MOM6

Generated by Doxygen 1.8.13

Contents

Chapter 1

ALE

Basics of the Vertical Lagrangian-Remap Method in MOM6

As discussed by **?**, there are two general classes of algorithms that frame how hydrostatic ocean models are formulated. The two classes differ in how they treat the vertical direction. Quasi-Eulerian methods follow the approach traditionally used in geopotential coordinate models, whereby vertical motion is diagnosed via the continuity equation. Quasi-Lagrangian methods are traditionally used by layered isopycnal models, with the Lagrangian approach specifying motion that crosses coordinate surfaces. Indeed, such dia-surface flow can be set to zero using Lagrangian methods for studies of adiabatic dynamics. MOM6 makes use of the vertical Lagrangian remap method, as pioneered for ocean modeling by **?**, which is a limit case of the Arbitrary-Lagrangian-Eulerian method (**?**). Dia-surface transport is implemented via a remapping so that the method can be summarized as the Lagrangian plus remap approach and is essentially a one-dimensional version of the incremental remapping of **?**.

The MOM6 implementation of the vertical Lagrangian-remap method makes use of two general steps. The first evolves the ocean state forward in time according to a vertical Lagrangian limit with $\dot{r} = 0$. Hence, the horizontal momentum, thickness, and tracers are time stepped with the red terms removed in equations **??**, **??**, **??**, and **??**. All advective transport thus occurs within a layer as defined by constant r-surfaces so that the volume within each layer is fixed. All other terms are retained in their full form, including subgrid scale terms that contribute to the transfer of tracer and momentum into distinct r layers (e.g., dia-surface diffusion of tracer and velocity). Maintaining constant volume within a layer yet allowing for tracers to move between layers engenders no inconsistency between tracer and thickness evolution. The reason is that tracer diffusion, even dia-surface diffusion, does not transfer volume.

The second step in the algorithm comprises the generation of a new vertical grid following a prescription, such as whether the grid should align with isopcynals or constant z^* or a combination. The ocean state is then vertically remapped to the newly generated vertical grid. The remapping step incorporates dia-surface transfer of properties, with such transfer depending on the prescription given for the vertical grid generation. To minimize discretization errors and the associated spurious mixing, the remapping step makes use of the high order accurate methods developed by **?** and **?**.

The underlying algorithm for treatment of the vertical can be related to operator-splitting of the red terms in equations **??–??**. If we consider, for simplicity, an Euler-forward update for a time-step Δt, the time-stepping for the continuity and temperature equation can be summarized as

$$
\begin{aligned}
h^\dagger &= h^{(n)} - \Delta t \left[\nabla_r \cdot (h\,\mathbf{u}) \right] && \text{thickness} && (1.2)\\
\theta^\dagger\, h^\dagger &= \theta^{(n)}\, h^{(n)} - \Delta t \left[\nabla_r \cdot (\theta h\,\mathbf{u}) - h \boldsymbol{\mathcal{N}}^\gamma_\theta + \delta_r J^{(z)}_\theta \right] && \text{potential temp} && (1.3)\\
h^{(n+1)} &= h^\dagger - \Delta t\, \delta_r \left(z_r \dot{r} \right) && \text{move grid} && (1.4)\\
\theta^{(n+1)} h^{(n+1)} &= \theta^\dagger h^\dagger - \Delta t\, \delta_r \left(z_r \dot{r}\, \theta^\dagger \right) && \text{remap temperature.} && (1.5)
\end{aligned}
$$

Substituting **??** into **??** recovers a time-discrete form of **??**. The intermediate quantities indicated by †-symbols are the result of the vertical Lagrangian step of the algorithm. What were the red terms in the continuous-in-time equations are used to evolve the the intermediate quantities to the final updated quantities each step. In MO$\hookleftarrow$ M6, equation **??** is essentially used to define the dia-surface transport $z_r\dot{r}$ by prescribing $h^{(n+1)}$. For example, to recover a z-coordinate model, $h^{(n+1)} = \Delta z$, and $z_r\dot{r}$ becomes the Eulerian vertical velocity, w.

Within the above framework for evolving the ocean state, we make use of a standard split-explicit time stepping method by decomposing the horizontal momentum equation into its fast (depth integrated) and slow (deviation from depth integrated) components. Furthermore, we follow the methods of **?** to ensure that the free surface resulting from time stepping the depth integrated thickness equation (i.e., the free surface equation) is consistent with the sum of the thicknesses that result from time stepping the layer thickness equations for each of the discretized layers; i.e., $\sum_k h = H + \eta$.

Chapter 2

ALE Timestep

Explanation of ALE remapping

The Arbitrary Lagrangian-Eulerian (ALE) remapping is not a timestep in the traditional sense, but rather an operation performed to bring the vertical coordinate back to the target specification. This remapping can be less frequent than the momentum or thermodynamic timesteps, but must be done before the layer interfaces become entangled with each other.

Assuming the target vertical grid is level z-surfaces, the initial state is shown on the left in the following figure:

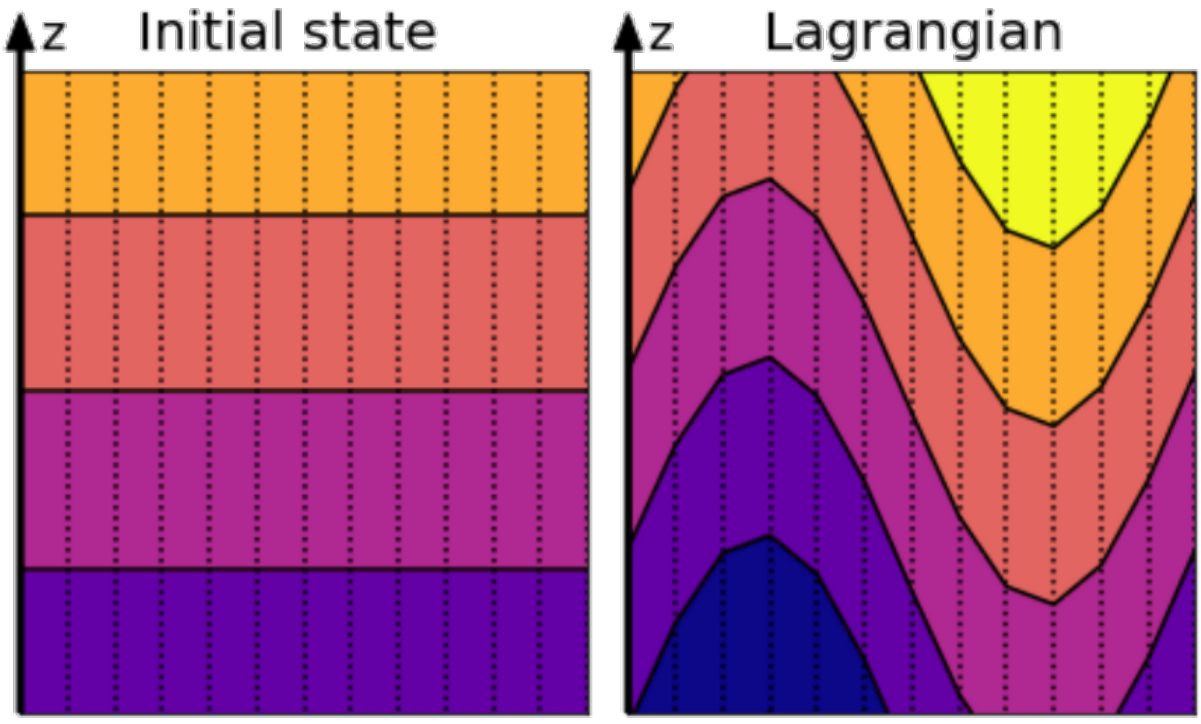


Figure 2.1 The initial state with level surface (left) and the perturbed state after a wave has come through (right).

Some time later, a wave has perturbed the surfaces which move with the fluid and it has been determined that a remapping operation is needed. The target vertical grid is still level z-surfaces, so this new target grid is shown overlaid on the left as regrid:

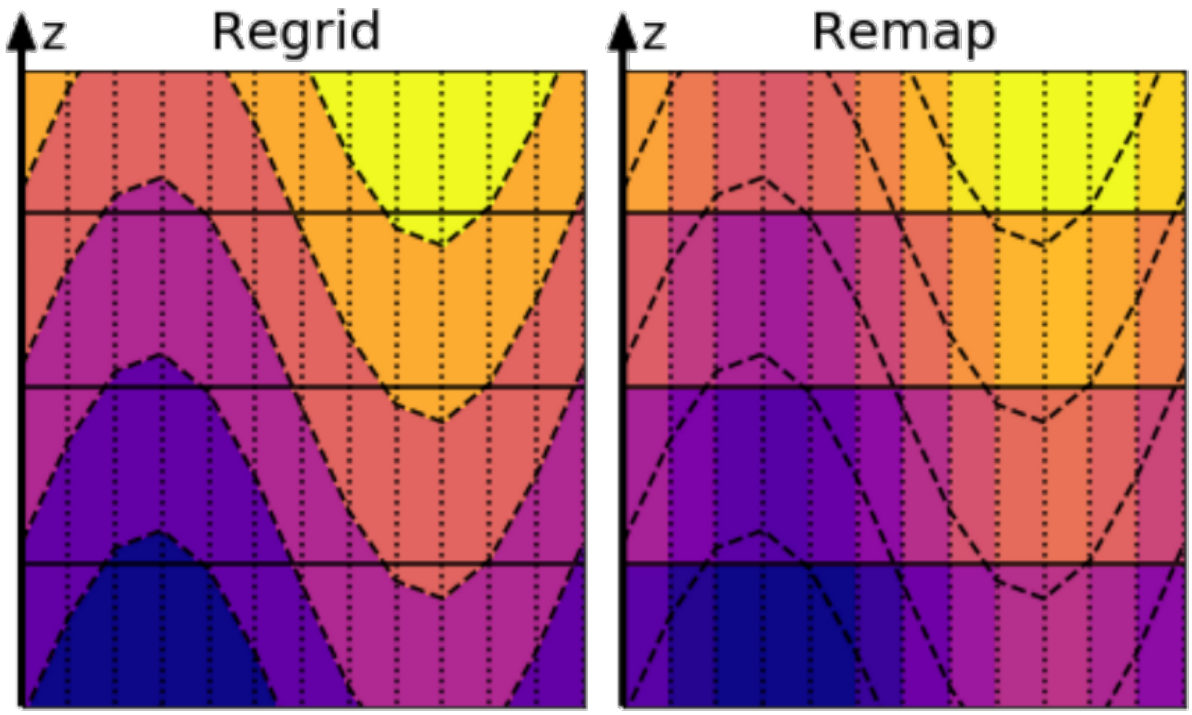


Figure 2.2 The regrid operation (left) and the remap operation (right).

The complex part of the operation is remapping the wavy field onto the new grid as shown on the right and again in the final frame after the old deformed coordinate system has been deleted:

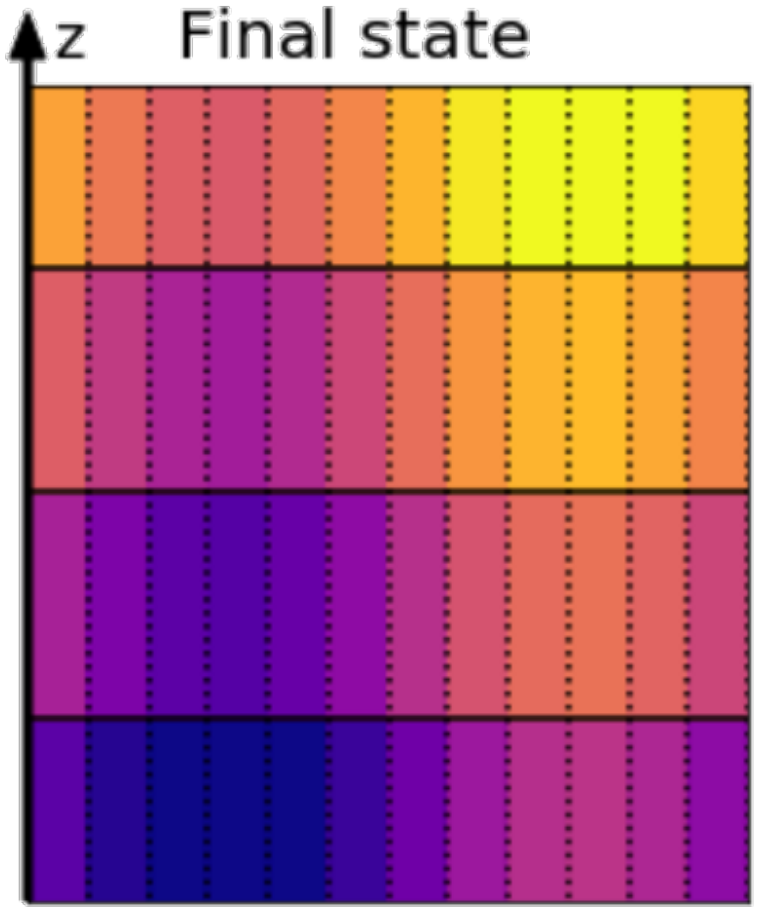


Figure 2.3 The final state after remapping.

Mathematically, the new layer thicknesses, h_k, are computed and then populated with the new velocities and tracers:

$$h_k^{\text{new}} = \nabla_k z_{\text{coord}}$$

$$\sum h_k^{\text{new}} = \sum h_k^{\text{old}}$$

$$\vec{u}_k^{\text{new}} = \frac{1}{h_k} \int_{z_{k+\frac{1}{2}}}^{z_{k+\frac{1}{2}}+h_k} \vec{u}^{\text{old}}(z')dz'$$

$$\theta^{\text{new}} = \frac{1}{h_k} \int_{z_{k+\frac{1}{2}}}^{z_{k+\frac{1}{2}}+h_k} \theta^{\text{old}}(z')dz'$$

Chapter 3

Baroclinic Momentum Equations

Baroclinic Momentum Equations

The baroclinic momentum equations are the stacked shallow water equations:

$$\frac{\partial \vec{u}_k}{\partial t} + (f + \nabla_s \times \vec{u}_k)\hat{z} \times \vec{u}_k = -\frac{\nabla_s p_k}{\rho} - \nabla_s(\phi_k + \frac{1}{2}||\vec{u}_k||^2) + \frac{\nabla \cdot \tilde{\tau}_k}{\rho}$$

$$\frac{\partial h_k}{\partial t} + \nabla_s \cdot (\vec{u} h_k) = 0$$

The timestepping for these equations is a (quasi?) second-order Runge-Kutta step for the inertial oscillations and a forward-backward Euler step for the pressure (gravity) waves. Using the graphical notation from **?**, it looks like:

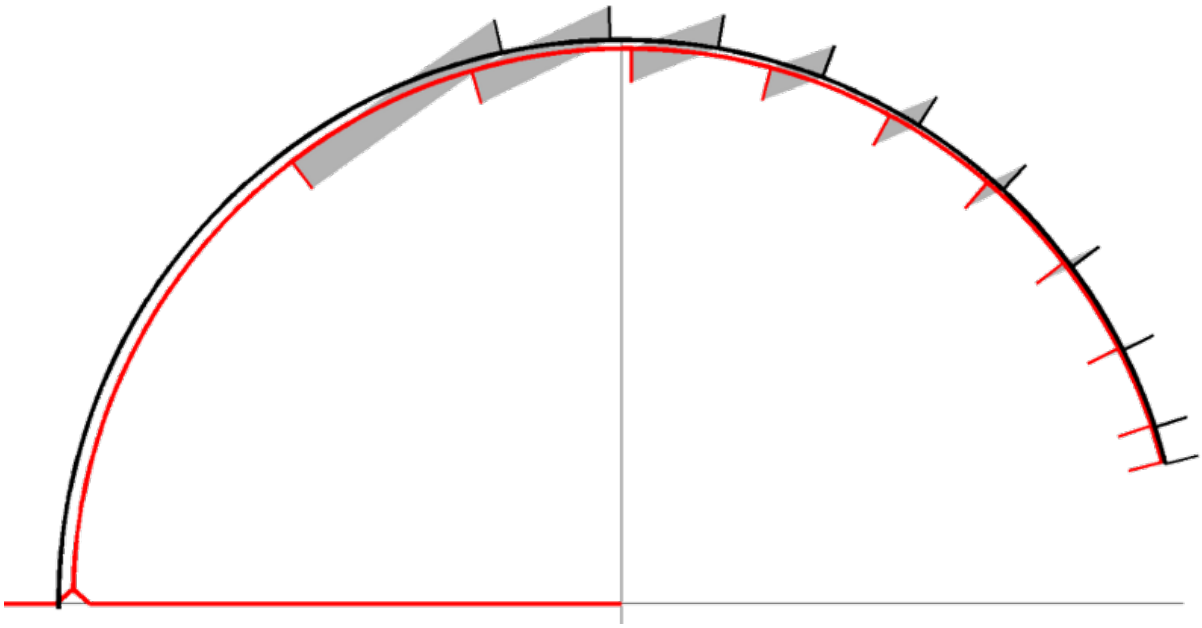


Figure 3.1 Graphical notation for timestepping schemes in which the black line represents the ideal solution and the red line shows the actual solution. Phase errors are represented by the grey shapes between the bars normal to the circle.

The timestep used in ROMS looks instead like:

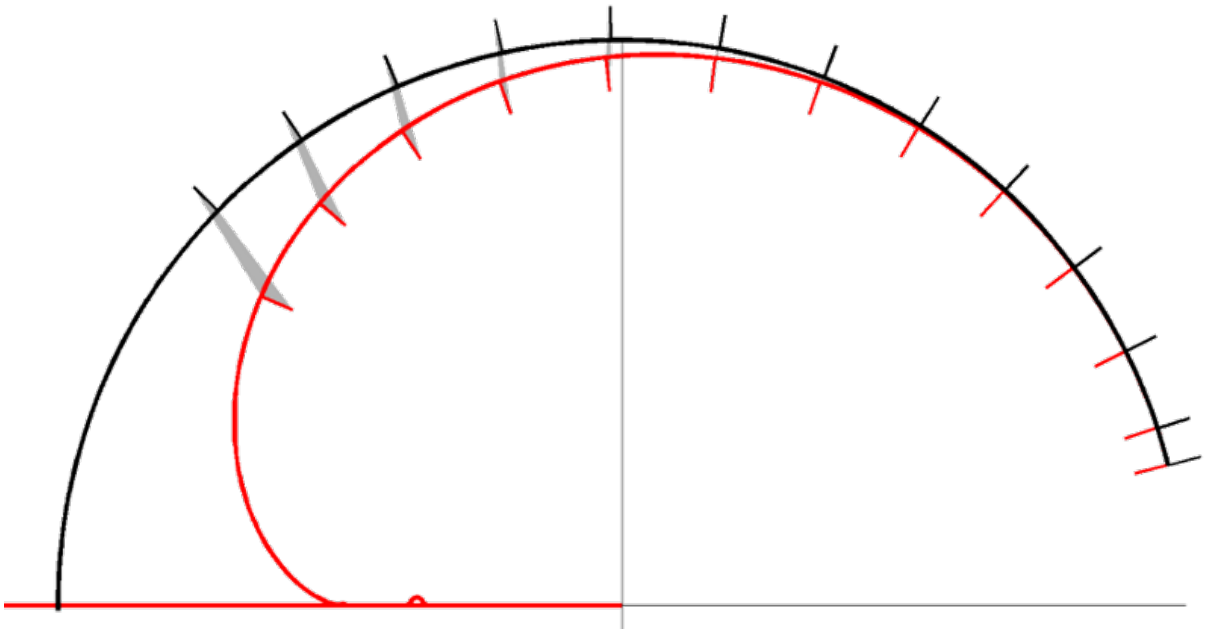


Figure 3.2 Graphical notation for the Adams-Bashforth technique used in the ROMS model.

The ROMS timestepping has smaller phase errors, strong damping at high frequency. The MOM6 use as a global climate model has made the phase errors of lower priority. However, the phase errors may become more problematic for future uses of MOM6. While the MOM6 use of the ALE remapping makes an Adams-Bashforth scheme impractical, there may be a better timestepping scheme out there for MOM6. Please let the MOM6 developers know if you would like to work on this problem.

Chapter 4

Barotropic-Baroclinic Coupling

Time-averaged accelerations

The barotropic equations are timestepped with a timestep to resolve the surface gravity waves. With care, the baroclinic timestep only need resolve the inertial oscillations. The barotropic accelerations are averaged over the many barotropic timesteps taken between baroclinic steps. At time n, the baroclinic accelerations are computed. The vertical average of that acceleration is subtracted off and replaced by the time-averaged acceleration from the group of barotropic timesteps:

$$\Delta t \frac{\partial \vec{u}}{\partial t} = \Delta t \left(\frac{\partial \vec{u}}{\partial t} - \frac{\partial \vec{u}_{BT}}{\partial t} \right)^n + \Delta t \overline{\frac{\partial \vec{u}_{BT}}{\partial t}}^{\Delta t}$$

Similarly, the velocities used in the tracer equation are a careful blend of the barotropic and baroclinic solutions:

$$\Delta t \frac{\partial \theta}{\partial t} + \Delta t \left(\tilde{\vec{u}} \cdot \nabla \theta + \tilde{w} \frac{\partial \theta}{\partial z} \right)$$

with

$$\tilde{\vec{u}} = \vec{u}_{BC} + \overline{\vec{u}_{BT}}^{\Delta t}$$
$$\frac{\partial \tilde{w}}{\partial z} = -\nabla \cdot \tilde{\vec{u}}$$

4.1 Two estimates of the free surface height

The vertically discrete, temporally continuous layer continuity equations are:

$$\frac{\partial h_k}{\partial t} = -\nabla \cdot (\vec{u} h_k) = -\nabla \cdot \mathbf{F}(u_k, h_k)$$

The relationship between the free surface height and the layer thicknesses h_k is:

$$\eta = \sum_{k=1}^{N} h_k - D$$

We get an evolution equation for the free surface height:

$$\frac{\partial \eta}{\partial t} = \sum_{k=1}^{N} \frac{\partial h_k}{\partial t} = -\nabla \cdot \sum_{k=1}^{N} \mathbf{F}(u_k, h_k)$$

If the algorithms for the fluxes in the continuity equations are *linear* in the velocity, the free surface height can be rewritten as:

$$\frac{\partial \eta}{\partial t} \& = -\nabla \cdot \sum_{k=1}^{N} \mathbf{F}(u_k, h_k) = -\nabla \cdot \sum_{k=1}^{N} (\vec{u}_k h_k) \& = -\nabla \cdot \left[\sum_{k=1}^{N} h_k \frac{\sum_{k=1}^{N} (\vec{u}_k h_k)}{\sum_{k=1}^{M} k_k} \right] \equiv -\nabla \cdot H\mathbf{U}$$

where

$$\mathbf{U} \equiv \frac{\sum_{k=1}^{N} (\vec{u}_k h_k)}{\sum_{k=1}^{M} k_k}$$

$$H \equiv \sum_{k=1}^{N} h_k$$

However, ALE models like MOM6 require positive-definite, nonlinear continuity solvers (MOM6 uses PPM Advection Scheme). We need a different way to reconcile this estimate of free surface height with the one coming from the barotropic equations. After rejecting several other options, MOM6 is now using the scheme from **?**. The barotropic update of η is given by:

$$\frac{\eta^{n+1} - \eta^n}{\Delta t} + \nabla \cdot \left(\overline{UH} \right) = 0$$

Determine the ΔU such that:

$$\sum_{k=1}^{N} \mathbf{F}(\tilde{u}_k, h_k) = \left(\overline{UH} \right)$$

where

$$\tilde{u}_k = u_k + \Delta U$$

The layer timestep equation becomes:

$$h_k^{n+1} = h_k^n - \Delta t \nabla \cdot \mathbf{F}(\tilde{u}_k, h_k)$$

This scheme has these properties:

- Total mass is conserved layer-wise.
- Layer equations retain their flux form.
- Total salt, heat, and other tracers are exactly conserved.
- Free surface heights exactly agree.
- Requires (very few) completely local iterations.
- The velocity corrections are barotropic, and more likely to correspond to the layers whose flow was deficient than in some older schemes.

The solution is unique provided that:

$$\frac{\partial}{\partial \tilde{u}_k}\mathbf{F}(\tilde{u}_k, h_k) > 0$$

This is a reasonable requirement for any discretization of the continuity equation. In the continuous limit, $\mathbf{F}(u,h) = uh$, so one interpretation is:

$$\frac{\partial}{\partial \tilde{u}_k}\mathbf{F}(\tilde{u}_k, h_k) = h_{k,\text{Marginal}}$$

With the PPM continuity scheme:

$$F_{i+1/2} = \frac{1}{\Delta t}\int_{x_{i+1/2}-u\Delta t}^{x_{i+1/2}} h_i^n(x)dx$$

leads to:

$$\frac{\partial F_{i+1/2}}{\partial u_{i+1/2}} = h_i^n(x_{i+1/2} - u_{i+1/2}\Delta t) \equiv h_{k,\text{Marginal}}$$

$h_i(x) > 0$ is already required for positive definiteness.

Newton's method iterations quickly give ΔU:

$$\Delta U^{m+1} = \Delta U^m + \frac{(\overline{UH}) - \sum_{k=1}^{N} F(u_k + \Delta U^m, h_k)}{\sum_{k=1}^{N} h_{k,\text{Marginal}}}$$

4.1.1 How practical is this iterative approach?

The piecewise parabolic method continuity solver uses two steps:

- Set up the positive-definite subgridscale profiles, $h_{PPM}(x)$.

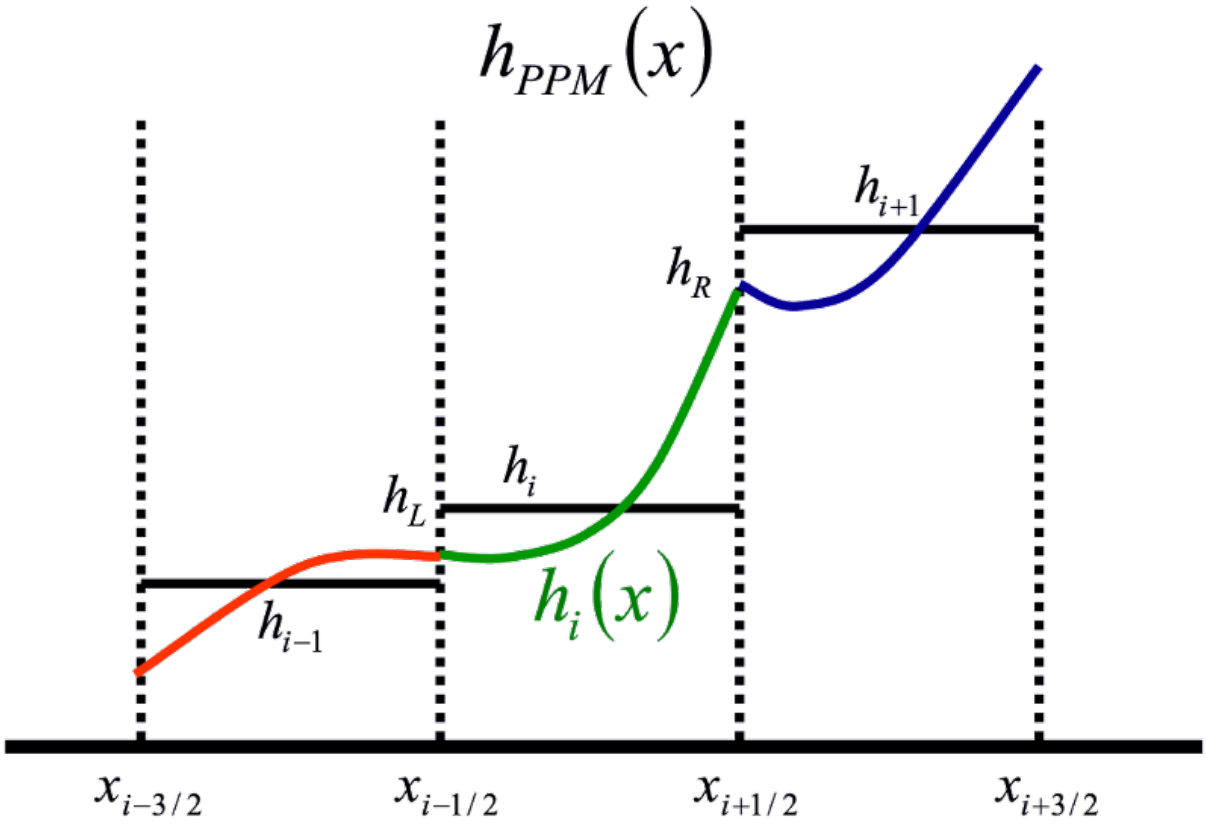


Figure 4.1 Piecewise parabolic reconstructions of $h(x)$.

- Integrating the profiles to determine F. Step 1 is typically ~ 3 times as expensive as step 2. $F(u,h)$ is piecewise cubic in u, but often nearly linear. Newton's method iterations converge quickly:

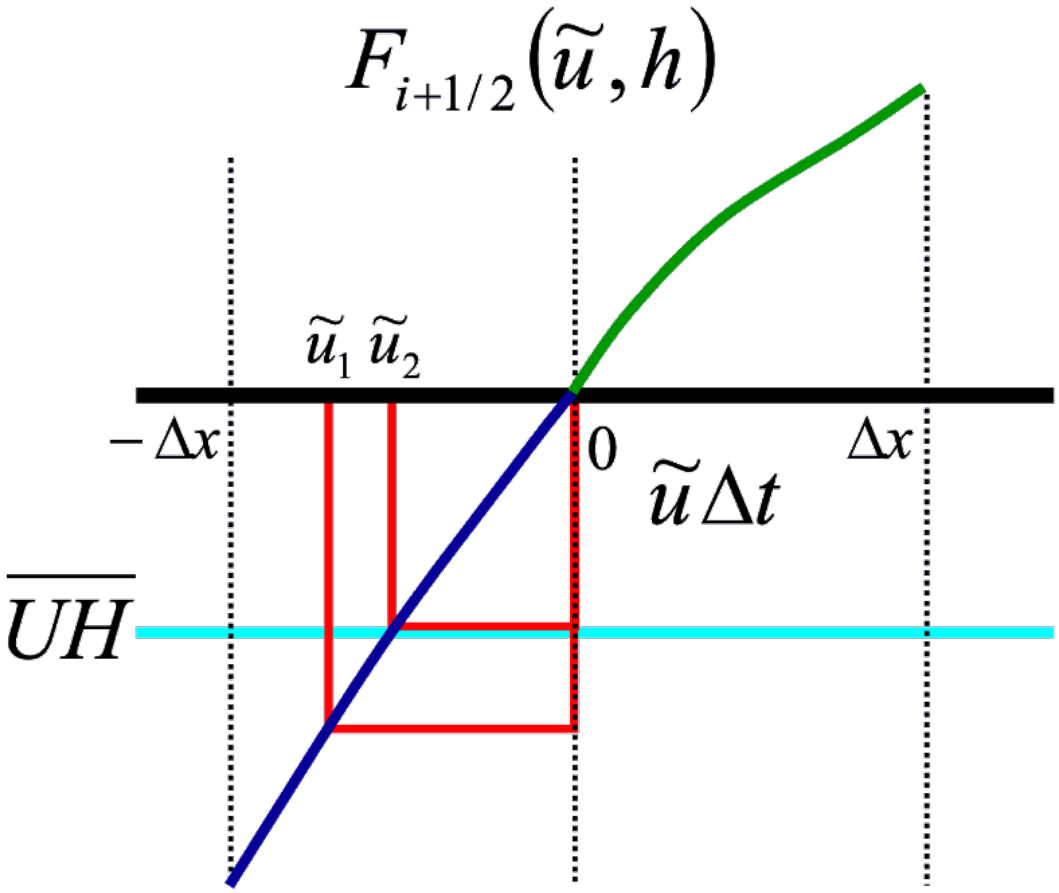


Figure 4.2 Newton's method iterations for finding $ U$.

In a global model the sea surface heights converge everywhere to a tolerance of 10^{-6} m within five iterations. These five iterations add ~ 1.6 times more CPU time to the PPM continuity solver and the continuity solver is just 12% of the total model time.

4.1.2 A note on bottom drag

Barotropic accelerations do not lead to barotropic flows after a timestep when vertical viscosity is taken into account.

$$u_k^{n+1} = u_k^n + \Delta t A_k + \Delta t \frac{\tau_{k-1/2} - \tau_{k+1/2}}{h_k}$$

$$\tau_{1/2} = \tau_{Wind}$$

$$\tau_{k+1/2} = \nu_{k+1/2} \frac{u_k^{n+1} - u_{k+1}^{n+1}}{h_{k+1/2}}$$

$$\tau_{N+1/2} = \nu_{N+1/2} \frac{2u_N^{n+1}}{h_{N+1/2}}$$

$$\gamma_k \equiv \frac{1}{\Delta t}\frac{\partial u_k^{n+1}}{\partial A}$$

A tridiagonal equation for γ_k results, going from 0 for thin layers near the bottom to 1 far above the bottom.

$$\gamma_1 = 1 + \frac{1}{h_1}\left[-\frac{\nu_{3/2}\Delta t}{h_{3/2}}(\gamma_1 - \gamma_2)\right]$$
$$\gamma_k = 1 + \frac{1}{h_k}\left[\frac{\nu_{k-1/2}\Delta t}{h_{k-1/2}}(\gamma_{k-1} - \gamma_k) - \frac{\nu_{k+1/2}\Delta t}{h_{k+1/2}}(\gamma_k - \gamma_{k+1})\right]$$
$$\gamma_N = 1 + \frac{1}{h_N}\left[\frac{\nu_{N-1/2}\Delta t}{h_{N-1/2}}(\gamma_{N-1} - \gamma_N) - \frac{2\nu_{N+1/2}\Delta t}{h_{N+1/2}}\gamma_N\right]$$

In the continuous limit:

$$\gamma(z) = 1 + \Delta t \frac{d}{dz}\left(\nu\frac{d\gamma}{dz}\right)$$

with boundary conditions:

$$\frac{d\gamma}{dz}(0) = 0$$
$$\gamma(-D) = 0$$

For deep water and contant ν, we get:

$$\gamma(z) = 1 - \exp\left(-\sqrt{\nu\Delta t}(z + D)\right)$$

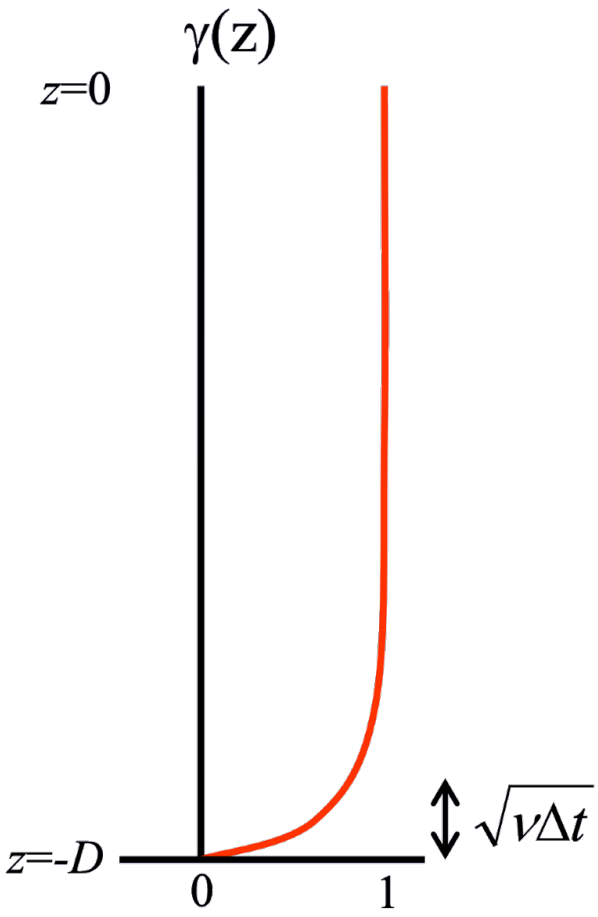


Figure 4.3 The continuous solution for barotropic flow plus a no-slip condition at the bottom.

We want a scheme in which the split model looks exactly like the unsplit model would if we had taken all those short 3D timesteps. Rather than applying a barotropic velocity, we use a barotropic acceleration and allow the continuity solver to determine the tranports consistent with a no-slip bottom boundary layer and perhaps also a no-slip surface boundary layer under an ice shelf.

From above, the barotropic equation is:

$$\frac{\eta^{n+1} - \eta^n}{\Delta t} + \nabla \cdot \left(\overline{UH}\right) = 0$$

We need to determine the $\Delta\overline{A}$ such that:

$$\sum_{k=1}^{N} \mathbf{F}(\tilde{u}_k, h_k) = \left(\overline{UH}\right)$$

where

$$\tilde{u}_k = u_k + \gamma_k \Delta\overline{A} \Delta t$$

4.2 Additional details about the split time stepping

- Transports are used as input and output to the barotropic solver. The continuity solver is inverted to determine velocities.

$$\frac{\partial \eta}{\partial t} = \nabla \cdot \overline{U} + M$$

$$\overline{U}(\overline{u}) = \frac{1}{\Delta t} \int_0^{\overline{u}\Delta t} H(x) dx$$

$$\overline{u}^n = \overline{U}^{-1} \left(\sum_{k=1}^{N} U_k^n \right)$$

$$u_k^{n+1} = \tilde{u}_k^{n+1} + \Delta\overline{u}$$

We need to find $\Delta\overline{u}$ such that:

$$\sum_{k=1}^{N} U_k \left(\tilde{u}_k^{n+1} + \Delta\overline{u} \right) = \overline{U}^{n+1}$$

- Barotropic accelerations are treates as anomalies from the baroclinic state:

$$\frac{\partial \overline{u}}{\partial t} \& = -f\hat{k} \times (\overline{u} - \overline{u}_{Cor}) - \nabla\overline{g}(\eta - \eta_{PF}) - \& \frac{c_D \left(||u_{Bot}|| + ||u_{Shelf}|| \right)}{\sum_{k=1}^{N} h_k} (\overline{u} - \overline{u}_{Drag}) + \frac{\sum_{k=1}^{N} h_k \frac{\partial u_k}{\partial t}}{\sum_{k=1}^{N} h_k}$$

- Bottom drag (and under ice surface-drag) is treated implicitly.
- The barotropic continuity solver uses flow-dependent thickness fits which approximate the sum of the layer thicness transports, to accommodate wetting and drying. An image of this is shown here:

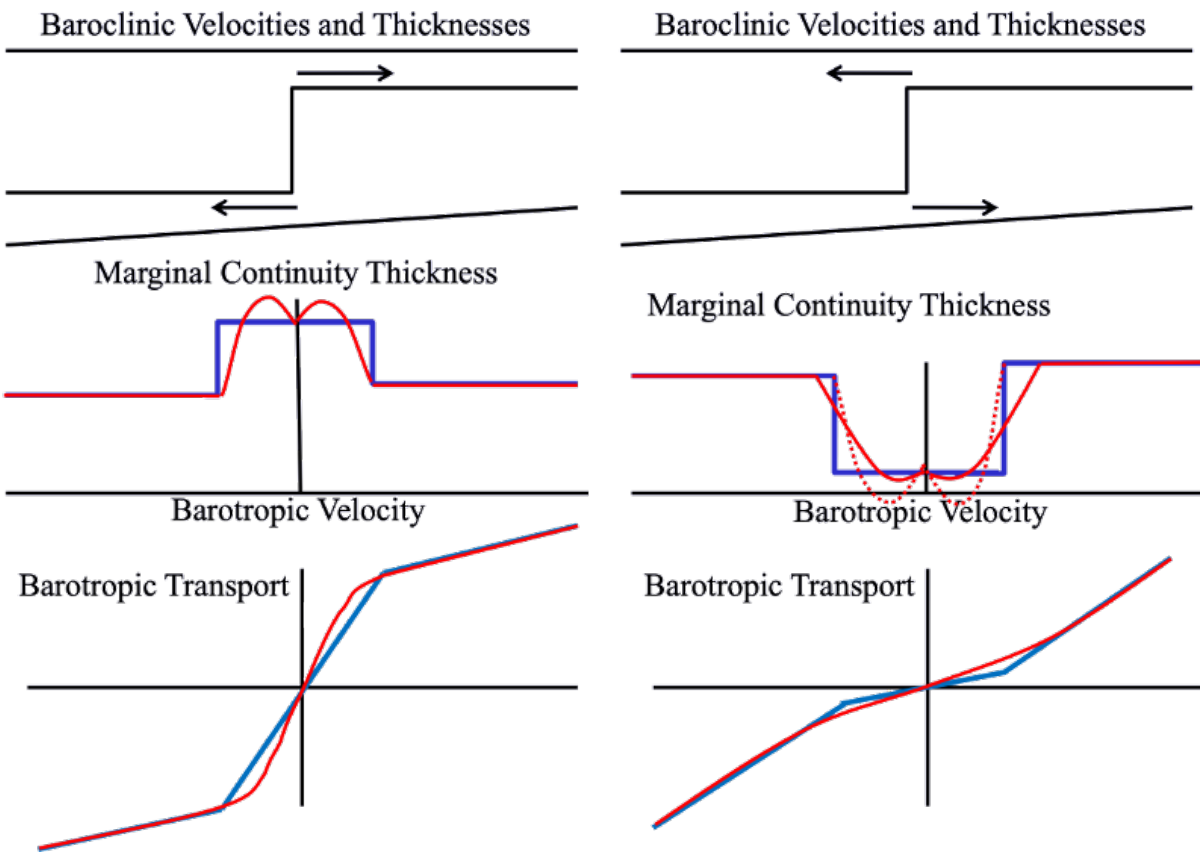


Figure 4.4 The barotropic transports depend on the baroclinic flows and thicknesses.

4.3 Summary of MOM6 split time stepping

- We use an efficient approach for handling fast modes via simplified 2-d equations, while the 3-d baroclinic timestep is determined by baroclinic dynamics.
- The barotropic solver determines free surface height and time-averaged depth-integrated transports.
- By using anomalies, the MOM6 split solver gives identical answers to an equivalent unsplit scheme for short timesteps.
- This scheme has been demonstrated to work with wetting and drying, as well as under ice-shelves.
- The linear barotropic solver allows MOM6 to automatically set a stable barotropic timestep (e.g. to 98% of maximum).

Chapter 5

Barotropic Momentum Equations

Barotropic Momentum Equations

The barotropic equations are timestepped on a relatively short timestep compared to the rest of the model. Since the timestep constraints for this are known, the barotropic timestep is computed at runtime.

The 2-d linear momentum equations with integrated continuity are:

$$\frac{\partial \eta}{\partial t} + \nabla \cdot ((D+\eta)\vec{u}_{BT} h_k) = P - E$$
$$\frac{\partial \vec{u}_{BT}}{\partial t} = -g\nabla\eta - f\hat{z} \times \vec{u}_{BT} + \vec{F}_{BT}$$

where

$$\vec{u}_{BT} \equiv \frac{1}{D+\eta}\int_{-D}^{\eta} \vec{u} dz$$

and $\vec{F}_{BT}$ is the barotropic momentum forcing from baroclinic processes. Note that explicit mass fluxes such as evaporation and precipitation change the model volume explicitly.

In the mode splitting between baroclinic and barotropic processes, it is important to include the contribution of free surface waves on the internal interface heights on the pressure gradient force, shown here as g_{Eff}:

$$\frac{\partial p}{\partial z} = -\rho g$$
$$g_{Eff} = g + \frac{\partial}{\partial \eta}\left[\frac{1}{D+\eta}\int_{-D}^{\eta} p dz\right]$$

The barotropic momentum equation then becomes:

$$\frac{\partial \vec{u}_{BT}}{\partial t} + f\hat{z} \times \vec{u}_{BT} + \frac{1}{\rho_0}\nabla g_{Eff}\eta = \text{Residual}$$

Without including the internal wave motion in the barotropic equations, one can generate instabilities (**?**, **?**).

Chapter 6

Discrete Coriolis Term

6.1 Coriolis Term

In general, the discrete equations are written as simple difference equations based on the Arakawa C-grid as described in section Horizontal grids. One of the more interesting exceptions is the Coriolis term. It is computed in the form shown in **??**, or:

$$\frac{(f+\zeta)}{h}\,\hat{\mathbf{z}}\times h\,\mathbf{u}$$

This term needs to be evaluated at u points for the v equation and vice versa, plus we need to keep the thickness, h, positive definite. MOM6 contains a number of options for how to compute this term.

- SADOURNY75_ENERGY Sadourny **?** figured out how to conserve energy or enstrophy but not both. This option is energy conserving. The term in the u equation becomes:

  $$\frac{1}{4dx}\left(q_{i,j}(vh_{i+1,j}+vh_{i,j})+q_{i,j-1}(vh_{i+1,j-1}+vh_{i,j-1})\right)$$

  where $q=\frac{f+\zeta}{h}$ and h is an area-weighted average of the four thicknesses surrounding the q point, such that it is guaranteed to be positive definite.

There is a variant on this scheme with the CORIOLIS_EN_DIS option. If true, two estimates of the thickness fluxes vh are used to estimate the Coriolis term, and the one that dissipates energy relative to the other one is used.

- SADOURNY75_ENSTRO Also from **?**, this option is enstrophy conserving.

  $$\frac{1}{8dx}(q_{i,j}+q_{i,j-1})((vh_{i+1,j}+vh_{i,j})+(vh_{i+1,j-1}+vh_{i,j-1}))$$

- ARAKAWA_LAMB81 From **?** is a scheme which is both energy and enstrophy conserving. Its weaknesses are a large stencil and differing thickness stencils in the numerator and denominator. This scheme and several others (with differing values of a, b, c, d and ep) are implemented as:

  $$\frac{1}{dx}(a_{i,j}vh_{i+1,j} \quad +b_{i,j}vh_{i,j}+d_{i,j}vh_{i+1,j-1}+c_{i,j}vh_{i,j-1} \tag{6.1}$$

  $$+ep_{i,j}*uh_{i-1,j}-ep_{i+1,j}*uh_{i+1,j}) \tag{6.2}$$

  with

  $$a_{i,j} = \tfrac{1}{24}(2.0*(q_{i+1,j}+q_{i,j-1})+(q_{i,j}+q_{i+1,j-1})) \tag{6.3}$$

  $$b_{i,j} = \tfrac{1}{24}((q_{i,j}+q_{i-1,j-1})+2.0*(q_{i-1,j}+q_{i,j-1})) \tag{6.4}$$

  $$c_{i,j} = \tfrac{1}{24}(2.0*(q_{i,j}+q_{i-1,j-1})+(q_{i-1,j}+q_{i,j-1})) \tag{6.5}$$

  $$d_{i,j} = \tfrac{1}{24}((q_{i+1,j}+q_{i,j-1})+2.0*(q_{i,j}+q_{i+1,j-1})) \tag{6.6}$$

  $$ep_{i,j} = \tfrac{1}{24}((q_{i,j}-q_{i-1,j-1})+(q_{i-1,j}-q_{i,j-1})) \tag{6.7}$$

- ARAKAWA_HSU90 From **?** is a scheme which always conserves energy and conserves enstrophy in the limit of non-divergent flow. This one has a larger stencil than Sadourny's energy scheme, but it's much better behaved in terms of handling vanishing layers than Arakawa and Lamb. This scheme is implemented with:

$$\frac{1}{dx}(a_{i,j}vh_{i+1,j} + b_{i,j}vh_{i,j} + d_{i,j}vh_{i+1,j-1} + c_{i,j}vh_{i,j-1})$$

and

$$a_{i,j} = \tfrac{1}{12}(q_{i,j} + (q_{i+1,j} + q_{i,j-1})) \tag{6.8}$$
$$b_{i,j} = \tfrac{1}{12}(q_{i,j} + (q_{i-1,j} + q_{i,j-1})) \tag{6.9}$$
$$c_{i,j} = \tfrac{1}{12}(q_{i,j} + (q_{i-1,j-1} + q_{i,j-1})) \tag{6.10}$$
$$d_{i,j} = \tfrac{1}{12}(q_{i,j} + (q_{i+1,j-1} + q_{i,j-1})) \tag{6.11}$$

- ARAKAWA_LAMB_BLEND This is a blending of Arakawa and Lamb, Arakawa and Hsu, and the Sadourny Energy scheme. There are weights CORIOLIS_BLEND_WT_LIN and CORIOLIS_BLEND_F_EFF_MAX to control this scheme. The equation is the same as for Arakawa and Lamb **??**, but the values of a, b, c, d and ep differ when the pure Arakawa and Lamb scheme breaks down due to thickness variations.

- ROBUST_ENSTRO An enstrophy-conserving scheme which is robust to vanishing layers.

Some of these options also support the BOUND_CORIOLIS flag. If true, the Coriolis terms in the u equation are bounded by the four estimates of $\frac{(f+\zeta)}{h}vh$ from the four neighboring v points, and similarly in the v equation. This option would have no effect on the SADOURNY75_ENERGY scheme if it were possible to use centered difference thickness fluxes.

6.1.1 Wall boundary conditions

Two sets of boundary conditions have been coded in the definition of relative vorticity. These are written as:

NOSLIP defined (in spherical coordinates):

$$\text{relvort} = dv/dx \text{ (east } \& \text{ west), with } v = 0. \tag{6.12}$$
$$\text{relvort} = -\sec(\phi) * d(u\cos(\phi))/dy \text{ (north } \& \text{ south), with } u = 0. \tag{6.13}$$

Free slip (NOSLIP not defined):

$$\text{relvort} = 0 \text{ (all boundaries)}$$

with ϕ defined as latitude. The free slip boundary condition is much more natural on a C-grid.

Chapter 7

Discrete Horizontal and Vertical Grids

Discrete Horizontal and Vertical Grids

7.1 Horizontal grids

The placement of model variables on the horizontal C-grid is illustrated here:

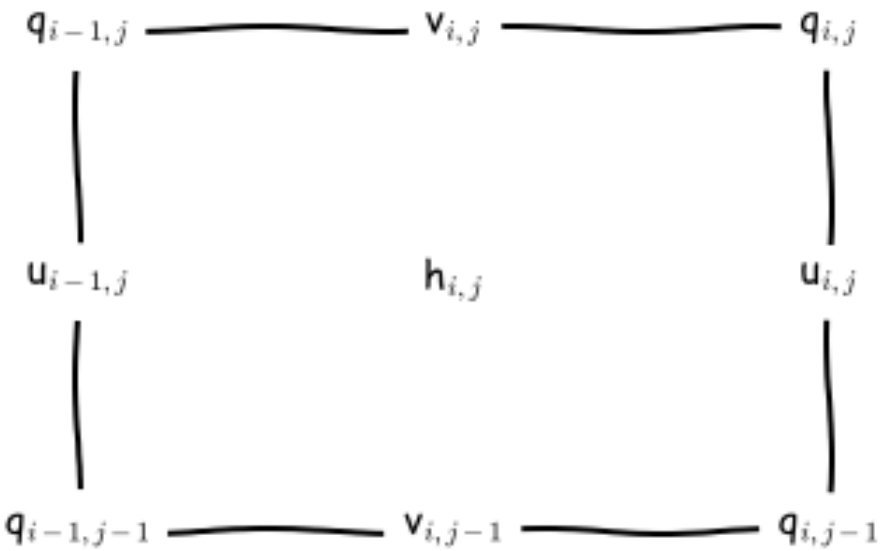


Figure 7.1 MOM6 uses an Arakawa C grid staggering of variables with a North-East indexing convention.

Scalars are located at the h-points, velocities are staggered such that u-points and v-points are not co-located, and vorticities are located at q-points. The indexing for points (i, j) in the logically-rectangular domain is such that i

increases in the x direction (eastward for spherical polar coordinates), and j increases in the y direction (northward for spherical polar coordinates). A q-point with indices (i, j) lies to the upper right (northeast) of the h-point with the same indices. The index for the vertical dimension k increases with depth, although the vertical coordinate z, measured from the mean surface level $z = 0$, decreases with depth.

When the horizontal grid is generated, it is actually computed on the "supergrid" at twice the nominal resolution of the model. The grid file contains the grid metrics and the areas of this fine grid. The model then decomposes it into the four staggered grids, along with computing the grid metrics as shown here:

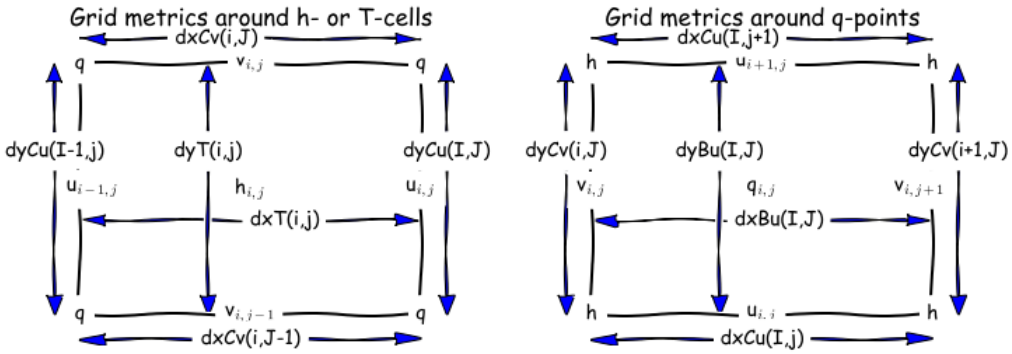


Figure 7.2 The grid metrics around both h-points and q-points.

The model carries both the metrics as well as their inverses, for instance, IdyT = 1/dyT. There are also the areas and the inverse areas for all four grid locations. areaT and areaBu are the sum of the four areas from the supergrid surrounding each h-point and each q-point, respectively. The velocity faces can be partially blocked and their areas are adjusted accordingly, where dy_Cu and dx_Cv are the blocked distances at u and v points, respectively.

$$\text{areaCu}_{i,j} = dxCu_{i,j} * dy_Cu_{i,j} \tag{7.1}$$
$$\text{areaCv}_{i,j} = dx_Cv_{i,j} * dyCv_{i,j} \tag{7.2}$$
$$\text{IareaCu}_{i,j} = 1/\text{areaCu}_{i,j} \tag{7.3}$$
$$\text{IareaCv}_{i,j} = 1/\text{areaCv}_{i,j} \tag{7.4}$$

The horizontal grids can be spherical, tripole, regional, or cubed sphere. The default is for grids to be re-entrant in the x-direction; this needs to be turned off for regional grids.

7.2 Vertical grids

The placement of model variables in the vertical is illustrated here:

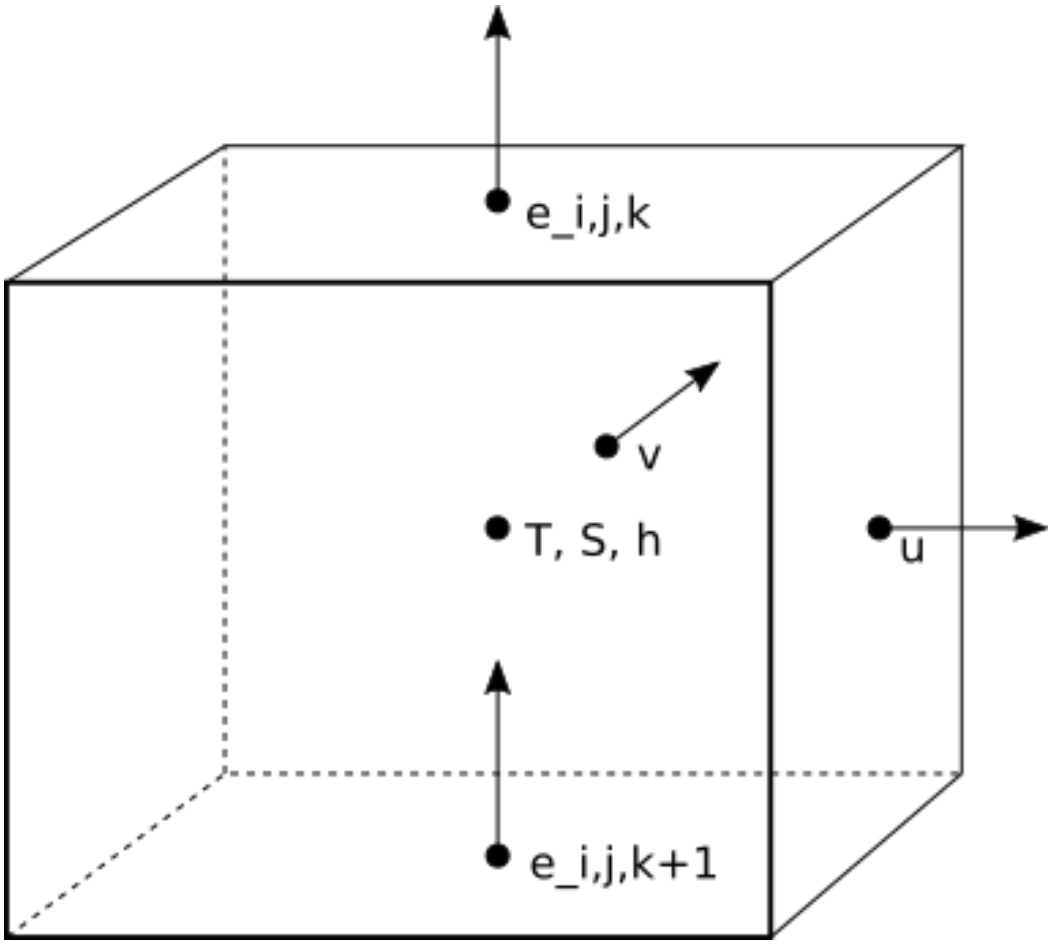


Figure 7.3 The MOM6 interfaces are at vertical location e which are separated by the layer thicknesses h.

The vertical coordinate is Lagrangian in that the interfaces between the layers are free to move up and down with time. The interfaces have target depths or target densities, depeneding on the desired vertical coordinate system. They can even have target sigma values for terrain-following coordinates or you can design a hybrid coordinate in which different interfaces have differing behavior. In any case, the interfaces move with the fluid during the dynamic timesteps and then get reset during a remapping operation. See section ALE Timestep for details.

Chapter 8

Discrete Open Boundary Conditions

Discrete Open Boundary Conditions

Chapter 9

Discrete Pressure Gradient Term

9.1 Pressure Gradient Term

Following **?**, the horizontal momentum equation in the general coordinate r can be written as:

$$\frac{\partial \vec{u}}{\partial t} + \nabla_r \Phi + \alpha \nabla_r p = \mathcal{F}$$

where the vector $\mathcal{F}$ represents all the forcing terms other than the pressure gradient. Here, $\vec{u}$ is the horizontal component of the velocity, Φ is the geopotential:

$$\Phi = gz$$

$\alpha = 1/\rho$ is the specific volume and p is the pressure. The gradient operator is a gradient along the coordinate surface r.

MOM6 offers two options, an older one using a Montgomery potential as described in **[hallberg1997]** and **?**. However, it can have the instability described in **?**. The version described here is that in **?** and is the recommended option (ANALYTIC_FV_PGF = True). The paper describes the Boussinesq form while the code supports that and also a non-Boussinesq form.

In two dimensions (x and p), we can integrate the zonal component of the momentum equation above over a finite volume:

$$-\int dx \int dp \frac{\partial u}{\partial t} = \int dx \int dp \, \frac{\partial \Phi}{\partial x}\Big|_p \tag{9.1}$$

$$= \int_{p_{br}}^{p_{tr}} \Phi dp + \int_{p_{tr}}^{p_{tl}} \Phi dp + \int_{p_{tl}}^{p_{bl}} \Phi dp \quad + \int_{p_{bl}}^{p_{br}} \Phi dp \tag{9.2}$$

We convert to line integrals thanks to the Leibniz rule. See the figure for the location of the line integral ranges:

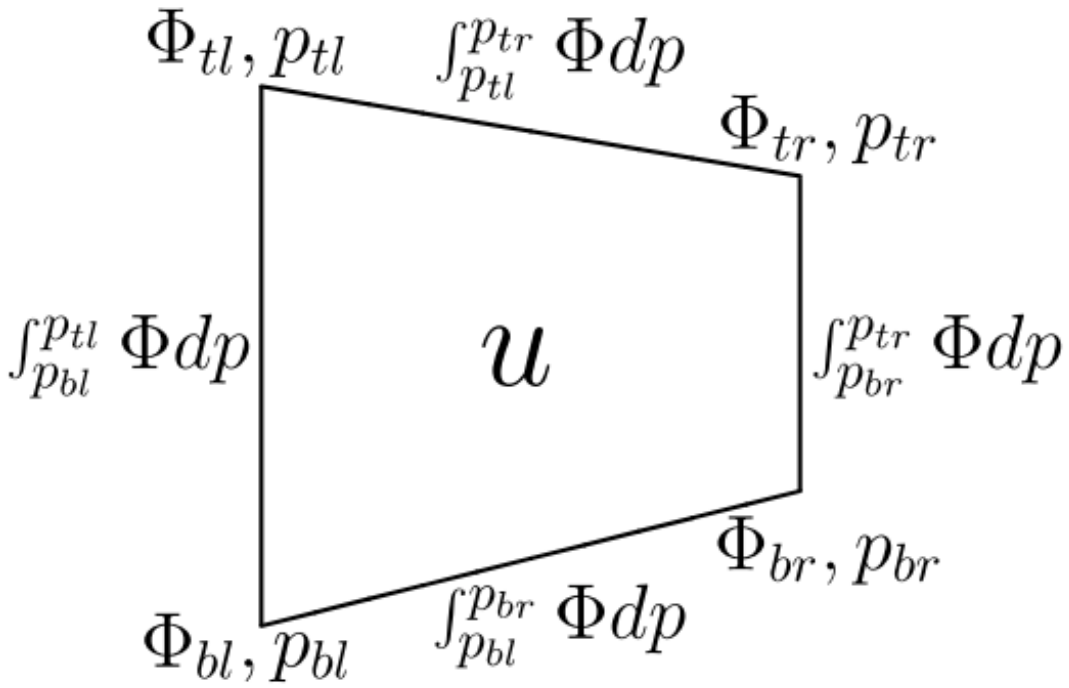


Figure 9.1 Schematic of the finite volume used for integrating the u-component of momentum. The thermodynamic variables $$ and s reside on the sides of the depicted volume and are considered uniform for the vertical extent of the volume but with linear variation in the horizontal. The volume is depicted in (x, p) space so p is linear around the volume but $$ can vary arbitrarily along the edges.

The only approximations made are (i) that the potential temperature θ and the salinity s can be represented continuously in the vertical within each layer although discontinuities between layers are allowed and (ii) that θ and s can be represented continuously along each layer. MOM6 has options for piecewise constant (PCM), piecewise linear (PLM), and piecewise parabolic (PPM) in the vertical.

If we use the Wright equation of state (**?**), we can integrate the above integrals analytically. This equation of state can be written as:

$$\alpha(s,\theta,p) = A(s,\theta) + \frac{\lambda(s,\theta)}{P(s,\theta)+p}$$

where A, λ and P are functions only of s and θ. The integral form of hydrostatic balance is:

$$\Phi(p_t) - \Phi(p_b) = \int_{p_t}^{p_b} \alpha(s,\theta,p)dp$$

Assuming piecewise constant values for θ and s and the above equation of state, we get:

$$\Phi(p_t) - \Phi(p_b) = \int_{p_t}^{p_b} \alpha(s,\theta,p)dp \tag{9.3}$$

$$= (p_b - p_t)A + \lambda \ln\left|\frac{P+p_b}{P+p_t}\right| \tag{9.4}$$

$$= \Delta p\left(A + \frac{\lambda}{P+\overline{p}}\frac{1}{2\epsilon}\ln\left|\frac{1+\epsilon}{1-\epsilon}\right|\right) \tag{9.5}$$

which is the exact solution for the continuum only if θ and s are uniform in the interval p_t to p_b. Here, we have introduced the variables:

$$\Delta p = p_b - p_t$$

$$\overline{p} = \frac{1}{2}(p_t + p_b)$$

and

$$\epsilon = \frac{\Delta p}{2(P + \overline{p})}$$

We will show later that $\epsilon \ll 1$. Note the series expansion:

$$\frac{1}{2\epsilon} \ln \left| \frac{1+\epsilon}{1-\epsilon} \right| = \sum_{n=1}^{\infty} \frac{\epsilon^{2n-2}}{2n-1} = 1 + \frac{\epsilon^2}{3} + \frac{\epsilon^4}{5} + \cdots \forall |\epsilon| \leq 1$$

Typical values for the deep ocean with 100 m layer thickness are 6×10^8 Pa and 10^6 Pa, respectively, yielding $\epsilon \sim 8 \times 10^{-4}$ and a corresponding accuracy in the geopotential height calculation of $\frac{\lambda \epsilon^3}{g} \sim 10^{-5}$ m. For this value of ϵ, the series converges with just three terms. In MOM6, we use series rather than the intrinsic log function , since the log is machine dependent and insufficiently accurate. In extreme circumstances, $\Delta p \sim 6 \times 10^7$ Pa (limited by the depth of the ocean) for which $\epsilon \sim 0.04$ with geopotential height errors of order 1 m. In this case, the series converges to machine precision with six terms.

The finite volume acceleration is expression terms of four integrals around the volume, $\int \Phi dp$. The side integrals can be calculated by direct integration of **??**, which gives:

$$\int_{p_t}^{p_b} \Phi dp = \Delta p \left(\Phi_b + \tfrac{1}{2} A \Delta p + \lambda \left(1 - \tfrac{1-\epsilon}{2\epsilon} \ln \left| \tfrac{1+\epsilon}{1-\epsilon} \right| \right) \right) \tag{9.6}$$

$$= \Delta p \left(\Phi_b + \tfrac{1}{2} A \Delta p + \lambda \left(1 - (1-\epsilon) \left(1 + \tfrac{\epsilon^2}{3} + \tfrac{\epsilon^4}{5} + \cdots \right) \right) \right) \tag{9.7}$$

$$= \Delta p \left(\Phi_b + \tfrac{1}{2} A \Delta p + \lambda \left(\epsilon - (1-\epsilon)\epsilon^2 \left(\tfrac{1}{3} + \tfrac{\epsilon^2}{5} + \cdots \right) \right) \right) \tag{9.8}$$

where $\Phi, \Delta p, P, A$ and λ are each evaluated on the left or right side of the volume.

The top and bottom integrals in **??** must allow for the effect of varying θ and s on A, λ and P. We evaluate these integrals numerically using sixth-order quadrature; Boole's rule requires evaluating the coefficients in the equation of state at five points, two of which have already been evaluated for the side integrals. For efficiency, we linearly interpolate the coefficients A, P and λ between the end points, which seems to make very little difference to the solution. We also verified that tenth-order quadrature makes little difference to the solution. The values of the top and bottom integrals are carried upward in a hydrostatic-like integration, obtained as follows:

$$\int_{p_{tl}}^{p_{tr}} \Phi_t dp = (p_{tr} - p_{tl}) \int_0^1 \Phi_t dx \tag{9.9}$$

$$= (p_{tr} - p_{tl}) \int_0^1 \left(\Phi_b + A(x)\Delta p(x) + \lambda(x) \ln \left| \tfrac{1+\epsilon(x)}{1-\epsilon(x)} \right| \right) dx \tag{9.10}$$

$$= (p_{tr} - p_{tl}) \int_0^1 \Phi_b dx \tag{9.11}$$

$$+ \int_0^1 \Delta p(x) \left(A(x) + \tfrac{\lambda(x)}{P(x)+\overline{p}(x)} \sum_{n=1}^{\infty} \tfrac{\epsilon^{2n-2}}{2n-1} \right) dx \tag{9.12}$$

The first integral is either known from the top integral of the layer below or the boundary condition at the ocean bottom. The second integral is evaluated numerically.

All the above definite integrals are specific to the Wright equation of state; the use of a different equation of state requires analytic integration of the appropriate equations. We have found, however, that high-order numerical integration appears to be sufficient. Although the numerical implementation is more general (allowing the use of arbitrary equations of state), it is significantly more expensive and so we advocate the analytic implementation for efficiency.

Chapter 10

Energetic Consistency

Energetic Consistency

Chapter 11

Finite Difference Operators

Finite Difference Operators

Chapter 12

General coordinate equations

General coordinate equations

Transforming to a vertical coordinate $r(z, x, y, t)$, with $\dot{r} = \frac{\partial r}{\partial t}$ ...

The Boussinesq hydrostatic equations of motion in general-coordinate r are:

$$
\begin{aligned}
&&&&(12.1)\\
\rho_0 \left(\frac{\partial \mathbf{u}}{\partial t} + (f+\zeta)\,\hat{\mathbf{z}} \times \mathbf{u} + \dot{r}\,\frac{\partial \mathbf{u}}{\partial r} + \nabla_r K \right) &= -\nabla_r p - \rho \nabla_r \Phi + \boldsymbol{\mathcal{F}} && \text{momentum} && (12.2)\\
\rho\,\frac{\partial \Phi}{\partial r} + \frac{\partial p}{\partial r} &= 0 && \text{hydrostatic} && (12.3)\\
\frac{\partial z_r}{\partial t} + \nabla_r \cdot\,(z_r\,\mathbf{u}) + \frac{\partial (z_r\,\dot{r})}{\partial r} &= 0 && \text{thickness} && (12.4)\\
\frac{\partial (\theta\, z_r)}{\partial t} + \nabla_r \cdot (\theta z_r\,\mathbf{u}) + \frac{\partial (\theta\, z_r\,\dot{r})}{\partial r} &= z_r \boldsymbol{\mathcal{N}}^{\gamma}_{\theta} - \tfrac{\partial J^{(z)}_{\theta}}{\partial r} && \text{potential temp} && (12.5)\\
\frac{\partial (S\, z_r)}{\partial t} + \nabla_r \cdot (S\, z_r\,\mathbf{u}) + \frac{\partial (S\, z_r\,\dot{r})}{\partial r} &= z_r \boldsymbol{\mathcal{N}}^{\gamma}_{S} - \tfrac{\partial J^{(z)}_{S}}{\partial r} && \text{salinity} && (12.6)\\
\rho &= \rho\,(S, \theta, -g\rho_0 z(r)) && \text{equation of state.} && (12.7)
\end{aligned}
$$

The time derivatives are now computed with the generalized vertical coordinate fixed rather than the geopotential. We introduced the specific thickness, $z_r = \partial z / \partial r$, which measures the inverse vertical stratification of the vertical coordinate surfaces.

Similar to **?**, MOM6 is discretized in the vertical by integrating between surfaces of r to yield layer equations where the layer thickness is $h = \int z_r dr$ and variables are treated as finite volume averages over each layer:

$$
\begin{aligned}
&&&&(12.8)\\
\rho_0 \left(\frac{\partial \mathbf{u}}{\partial t} + \frac{(f+\zeta)}{h}\,\hat{\mathbf{z}} \times h\,\mathbf{u} + \underbrace{\dot{r}\,\frac{\partial \mathbf{u}}{\partial r}} + \nabla_r K \right) &= -\nabla_r p - \rho \nabla_r \Phi + \boldsymbol{\mathcal{F}} && \text{momentum} && (12.9)\\
\rho\,\delta_r \Phi + \delta_r p &= 0 && \text{hydrostatic} && (12.10)\\
\frac{\partial h}{\partial t} + \nabla_r \cdot (h\,\mathbf{u}) + \underbrace{\delta_r (z_r \dot{r})} &= 0 && \text{thickness} && (12.11)\\
\frac{\partial (\theta\, h)}{\partial t} + \nabla_r \cdot (\theta h\,\mathbf{u}) + \underbrace{\delta_r (\theta\, z_r \dot{r})} &= h \boldsymbol{\mathcal{N}}^{\gamma}_{\theta} - \delta_r J^{(z)}_{\theta} && \text{potential temp} && (12.12)\\
\frac{\partial (S\, h)}{\partial t} + \nabla_r \cdot (S\, h\,\mathbf{u}) + \underbrace{\delta_r (S\, z_r \dot{r})} &= h \boldsymbol{\mathcal{N}}^{\gamma}_{S} - \delta_r J^{(z)}_{S} && \text{salinity} && (12.13)\\
\rho &= \rho\,(S, \theta, -g\rho_0 z(r)) && \text{equation of state,} && (12.14)
\end{aligned}
$$

where $\delta_r = \mathrm{d}r\,(\partial/\partial r)$ is the discrete vertical difference operator. The pressure gradient accelerations in the momentum equation **??** are written in continuous-in-the-vertical form for brevity; the exact discretization is detailed in **?**. The MOM6 time-stepping algorithm integrates the above layer-averaged equations forward allowing the vertical grid to follow the motion, i.e. $\dot{r} = 0$, so that the underbraced terms are dropped. This approach is generally known as the Lagrangian method but here the Lagrangian method is used only in the vertical direction. After each Lagrangian step, a remap step is applied that generates a new vertical grid of the user's choosing. The ocean state is then mapped from the old to the new grid. The physical state is not meant to change during the remap step, yet truncation errors make remapping imperfect. We employ high-order accurate reconstructions to minimize errors introduced during the remap step (**?**, **?**). The connection between time-stepping and remapping is described in section ALE Timestep.

Chapter 13

Governing Equations

The Boussinesq hydrostatic equations of motion in height coordinates are

$$D_t \boldsymbol{u} + f\widehat{\boldsymbol{k}} \times \boldsymbol{u} + \frac{\rho}{\rho_o}\boldsymbol{\nabla}_z \Phi + \frac{1}{\rho_o}\boldsymbol{\nabla}_z p = \boldsymbol{\mathcal{F}} \quad \text{momentum} \tag{13.1}$$

$$\rho \frac{\partial \Phi}{\partial z} + \frac{\partial p}{\partial z} = 0 \quad \text{hydrostatic} \tag{13.2}$$

$$\boldsymbol{\nabla}_z \cdot \boldsymbol{u} + \frac{\partial w}{\partial z} = 0 \quad \text{thickness} \tag{13.3}$$

$$D_t \theta = \boldsymbol{\mathcal{N}}^{\gamma}_{\theta} - \frac{\partial J^{(z)}_{\theta}}{\partial z} \quad \text{potential temp} \tag{13.4}$$

$$D_t S = \boldsymbol{\mathcal{N}}^{\gamma}_{S} - \frac{\partial J^{(z)}_{S}}{\partial z} \quad \text{salinity} \tag{13.5}$$

$$\rho = \rho(S, \theta, z) \quad \text{equation of state.} \tag{13.6}$$

where notation is described in Notation for equations, $\boldsymbol{\mathcal{F}}$ represents the accelerations due to the divergence of stresses including those provided through boundary interactions.

The prognostic thermodynamic variables are potential temperature, θ, and salinity S, which are related to *in situ* density ρ through the **?** equation of state. In the potential temperature and salinity equations, fluxes due to diabatic, vertically oriented processes are indicated by $J^{(z)}$. The tendency due to the convergence of fluxes oriented along neutral directions is indicated by $\boldsymbol{\mathcal{N}}^{\gamma}$. Our implementation of this *neutral diffusion* parameterization is detailed in Shao et al. (personal comm.)

The total derivative is

$$D_t \equiv \frac{\partial}{\partial t} + \boldsymbol{v} \cdot \boldsymbol{\nabla} \tag{13.7}$$

$$= \frac{\partial}{\partial t} + \boldsymbol{u} \cdot \boldsymbol{\nabla}_z + w \frac{\partial}{\partial z}. \tag{13.8}$$

The non-divergence of flow allows a total derivative to be re-written in flux form:

$$D_t \theta = \frac{\partial}{\partial t} + \boldsymbol{\nabla} \cdot (\boldsymbol{v}\theta) \tag{13.9}$$

$$= \frac{\partial}{\partial t} + \boldsymbol{\nabla}_z \cdot (\boldsymbol{u}\theta) + \frac{\partial (w\theta)}{\partial z}. \tag{13.10}$$

The above equations of motion can thus be written as:

$$D_t \boldsymbol{u} + f\widehat{\boldsymbol{k}} \times \boldsymbol{u} + \frac{\rho}{\rho_o}\boldsymbol{\nabla}_z \Phi + \frac{1}{\rho_o}\boldsymbol{\nabla}_z p = \boldsymbol{\mathcal{F}} \quad \text{momentum} \tag{13.11}$$

$$\rho \frac{\partial \Phi}{\partial z} + \frac{\partial p}{\partial z} = 0 \quad \text{hydrostatic} \tag{13.12}$$

$$\boldsymbol{\nabla}_z \cdot \boldsymbol{u} + \frac{\partial w}{\partial z} = 0 \quad \text{thickness} \tag{13.13}$$

$$\frac{\partial \theta}{\partial t} + \boldsymbol{\nabla}_z \cdot (\boldsymbol{u}\theta) + \frac{\partial (w\theta)}{\partial z} = \boldsymbol{\mathcal{N}}_\theta^\gamma - \frac{\partial J_\theta^{(z)}}{\partial z} \quad \text{potential temp} \tag{13.14}$$

$$\frac{\partial S}{\partial t} + \boldsymbol{\nabla}_z \cdot (\boldsymbol{u}S) + \frac{\partial (wS)}{\partial z} = \boldsymbol{\mathcal{N}}_S^\gamma - \frac{\partial J_S^{(z)}}{\partial z} \quad \text{salinity} \tag{13.15}$$

$$\rho = \rho(S, \theta, z) \quad \text{equation of state.} \tag{13.16}$$

13.1 Vector Invariant Equations

MOM6 solves the momentum equations written in vector-invariant form.

A vector identity allows the total derivative of velocity to be written in the vector-invariant form:

$$D_t \boldsymbol{u} = \partial_t \boldsymbol{u} + \boldsymbol{v} \cdot \boldsymbol{\nabla} \boldsymbol{u} \tag{13.17}$$

$$= \partial_t \boldsymbol{u} + \boldsymbol{u} \cdot \boldsymbol{\nabla}_z \boldsymbol{u} + w \partial_z \boldsymbol{u} \tag{13.18}$$

$$= \partial_t \boldsymbol{u} + (\boldsymbol{\nabla} \times \boldsymbol{u}) \times \boldsymbol{v} + \boldsymbol{\nabla} \underbrace{\frac{1}{2} |\boldsymbol{u}|^2}_{\equiv K}. \tag{13.19}$$

The flux-form equations of motion in height coordinates can thus be written succinctly as:

$$\partial_t \boldsymbol{u} + \left(f\widehat{\boldsymbol{k}} + \boldsymbol{\nabla} \times \boldsymbol{u} \right) \times \boldsymbol{v} + \boldsymbol{\nabla} K + \frac{\rho}{\rho_o}\boldsymbol{\nabla}\Phi + \frac{1}{\rho_o}\boldsymbol{\nabla} p = \boldsymbol{\mathcal{F}} \quad \text{momentum} \tag{13.20}$$

$$\boldsymbol{\nabla}_z \cdot \boldsymbol{u} + \partial_z w = 0 \quad \text{thickness} \tag{13.21}$$

$$\partial_t \theta + \boldsymbol{\nabla}_z \cdot (\boldsymbol{u}\theta) + \partial_z (w\theta) = \boldsymbol{\mathcal{N}}_\theta^\gamma - \frac{\partial J_\theta^{(z)}}{\partial z} \quad \text{potential temp} \tag{13.22}$$

$$\partial_t S + \boldsymbol{\nabla}_z \cdot (\boldsymbol{u}S) + \partial_z (wS) = \boldsymbol{\mathcal{N}}_S^\gamma - \frac{\partial J_S^{(z)}}{\partial z} \quad \text{salinity} \tag{13.23}$$

$$\rho = \rho(S, \theta, z) \quad \text{equation of state} \tag{13.24}$$

where the horizontal momentum equations and vertical hydrostatic balance equation have been written as a single three-dimensional equation.

Chapter 14

Notation for equations

14.1 Symbols for variables

z refers to elevation (or height), increasing upward so that for much of the ocean z is negative.

x and y are the Cartesian horizontal coordinates.

λ and ϕ are the geographic coordinates on a sphere (longitude and latitude respectively).

Horizontal components of velocity are indicated by u and v and vertical component by w.

p is pressure and Φ is geo-potential:

$$\Phi = gz.$$

The thermodynamic state variables are usually salinity, S, and potential temperature, θ or the absolute salinity and conservative temperature, depending on the equation of state. ρ is in-situ density.

14.2 Vector notation

The three-dimensional velocity vector is denoted $\boldsymbol{v}$

$$\boldsymbol{v} = \boldsymbol{u} + \widehat{\boldsymbol{k}} w,$$

where $\widehat{\boldsymbol{k}}$ is the unit vector pointed in the upward vertical direction and $\boldsymbol{u} = (u, v, 0)$ is the horizontal component of velocity normal to the vertical.

The gradient operator without a suffix is three dimensional:

$$\boldsymbol{\nabla} = (\boldsymbol{\nabla}_z, \partial_z).$$

but a suffix indicates a lateral gradient along a surface of constant property indicated by the suffix:

$$\boldsymbol{\nabla}_z = \left(\partial_x|_z\,, \partial_y|_z\,, 0\right).$$

Chapter 15

PPM Advection Scheme

PPM Advection Scheme

Following **?** and **?**, we use the Piecewise Parabolic Method (PPM) to represent values within the model cells. Each cell is assumed to have a piecewise parabolic representation, which is uniquely prescribed by conservation and the two edge values. This method has the following features:

- The PPM approach is conservative.
- The (unlimited) order of accuracy is determined by the estimates of the edge values.
- Monotonicity is ensured by adjusting the edge values to flatten the profile.

An example is shown in this figure:

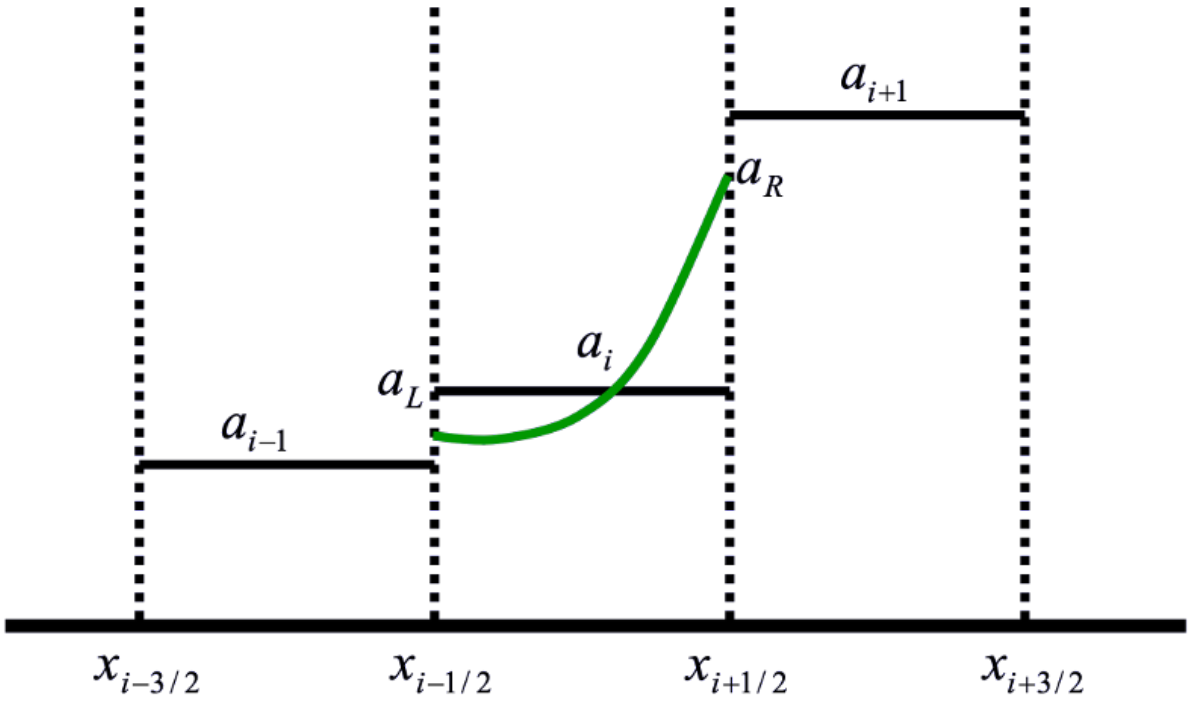


Figure 15.1 The parabolic representation of a field within a cell.

$$x_i' \equiv \frac{x - x_{i-1/2}}{\Delta x_i}$$

$$\Delta x_i \equiv x_{i+1/2} - x_{i-1/2}$$

$$c \equiv u\Delta t / \Delta x_i$$

$$A_i(x') = a_L + (a_R - a_L)x_i' + a_6 x_i'(1 - x_i')$$

$$a_6 = 6a_i - 3(a_R + a_L)$$

$$\begin{aligned} a_i &= \int_0^1 A_i(x_i')dx_i' = \int_0^1 a_L + (a_R - a_L)x_i' + a_6 x_i'(1 - x_i')dx_i' && (15.1)\\ &= \left[a_L x_i' + \tfrac{1}{2}(a_R - a_L)x_i'^2 + a_6\left(\tfrac{1}{2}x_i'^2 - \tfrac{1}{3}x_i'^3\right)\right]_0^1 && (15.2)\\ &= \tfrac{1}{2}(a_R + a_L) + \tfrac{1}{6}a_6 && (15.3) \end{aligned}$$

$$\begin{aligned} F_{i+1/2} &= \tfrac{1}{\Delta t}\int_{x_{i+1/2}-u\Delta t}^{x_{i+1/2}} A_i^n(x)dx = \tfrac{\Delta x}{\Delta t}\int_{1-c}^1 A_i(x_i')dx_i' && (15.4)\\ &= \tfrac{\Delta x}{\Delta t}\left[a_L x_i' + \tfrac{1}{2}(a_R - a_L)x_i'^2 + a_6\left(\tfrac{1}{2}x_i'^2 - \tfrac{1}{3}x_i'^3\right)\right]_{1-c}^1 && (15.5)\\ &= \tfrac{\Delta x}{\Delta t}\left[a_L c + (a_R - a_L + a_6)\left(c - \tfrac{1}{2}c^2\right) - a_6\left(c - c^2 + \tfrac{1}{3}c^3\right)\right] && (15.6)\\ &= u\left[a_R + \tfrac{1}{2}(a_L - a_R)c + a_6\left(\tfrac{1}{2}c - \tfrac{1}{3}c^2\right)\right] && (15.7) \end{aligned}$$

The choice of a_L and a_R is not unique, but can be done according to **?** (CW84) or **?** (H3) as mentioned in Tracer Advection.

Chapter 16

Sea Ice Considerations

Sea Ice Coupling

16.1 Ice Formation

Chapter 17

Solar Radiation

Penetration of Solar Radiation

17.1 Jerlov water type

17.2 Absorption by Chlorophyll

Chapter 18

Specifics

Specifics of the Ocean Model Equations

We here provide more details of the terms appearing in the ocean model equations described in General coordinate equations.

18.1 Horizontal Momentum Equation

Equation **??** is the horizontal momentum equation written in its vector-invariant advective form[1] with $\mathbf{u} = \hat{\mathbf{x}}\, u + \hat{\mathbf{y}}\, v$ the horizontal velocity, p the hydrostatic pressure, f the Coriolis parameter, and

$$\zeta^{(r)} = \hat{\mathbf{z}} \cdot (\nabla_r \times \mathbf{u})$$

the vertical component of the vorticity using ∇_r for the curl operator. The discretization of the Coriolis term is the enstrophy conserving scheme of **?**. The geopotential coordinate, z, has a value $z = 0$ at a resting ocean surface, $z = \eta(x, y, t)$ at the ocean free surface, and $z = -H(x, y)$ at the ocean bottom. We use the Boussinesq approximation (volume conserving kinematics) with $\rho_0 = 1035$ kg m^{-3} the reference density.[2] Time and horizontal derivatives are computed holding the generalized vertical coordinate fixed rather than the geopotential

$$\frac{\partial}{\partial t} = \left[\frac{\partial}{\partial t}\right]_r \qquad \nabla_r = \hat{\mathbf{x}} \left[\frac{\partial}{\partial x}\right]_r + \hat{\mathbf{y}} \left[\frac{\partial}{\partial y}\right]_r .$$

The transport of seawater crossing surfaces of constant r is measured by the dia-surface velocity component (see section 6.7 of **?**)

$$\frac{\partial z}{\partial r} \frac{\mathrm{D}r}{\mathrm{D}t} = z_r \dot{r},$$

with z_r the specific thickness that is assumed one-signed throughout the ocean, and $\mathrm{D}/\mathrm{D}t$ the material time derivative operator. In the ocean interior where r is aligned with isopycnals, the dia-surface velocity becomes the diapycnal velocity whose value is directly related to irreversible processes such as mixing that act on potential temperature and salinity. In the unstratified mixed layers, $r = z^*$ so that $z_r \dot{r} = (\partial z/\partial z^*)\, \mathrm{D}z^*/\mathrm{D}t$, which is close to the familiar vertical velocity component $\mathrm{D}z/\mathrm{D}t$.

Viscous dissipation (Laplacian and biharmonic friction following **?**) and mechanical boundary forces (winds, bottom stress) contribute to the divergence of the deviatoric (symmetric and trace-free) stress tensor, $\boldsymbol{\mathcal{F}} = \nabla \cdot \tau$. MOM6 and the real ocean have no vertical sidewalls, and MOM6 treats all solid-earth boundaries with bottom stress parameterized as a quadratic drag.

[1] The vector-invariant advective form is commonly used in models such as MOM6 using an Arakawa C-grid (e.g., see section 10 of @cite griffies2000-2).

[2] MOM6 has an option for compressible non-Boussinesq flow (mass conserving kinematics). We chose the Boussinesq option largely based on legacy.

18.2 Hydrostatic balance

Equation **??** is the discrete version of the hydrostatic balance. The horizontal pressure gradient force is implemented as a contact force following the method of **?**. These equations differ from **?** who uses the Montgomery potential to calculate pressure gradient accelerations.

18.3 Thickness and tracer equations

Volume conservation appears in the form of a prognostic flux-form layer thickness equation **??**, with the non-negative layer thickness given by

$$h = \frac{\partial z}{\partial r}\,\mathrm{d}r,$$

where $\mathrm{d}r$ is the thickness of a layer in r-space (e.g., the density difference between target density classes or the thickness between target depths). The layer thickness increases where horizontal thickness fluxes converge, $\nabla_r \cdot (h_k\,\mathbf{u}) < 0$, and where dia-surface flow converges, $\delta_r(z_r\,\dot{r}) < 0$. The volume flux $h_k\mathbf{u}$ is computed using the quasi-third order PPM scheme (**?**) using a positive-definite limiter rather than the monotonic limiter. This last choice avoids limiting of positive extrema and thus retains third-order accuracy everywhere except near vanishing layers.

Transport in the thickness equation is discretized compatibly with that in the flux-form potential temperature and salinity equations **??** and **??**. Compatibility is required to maintain global and layer integrated conservation properties for volume, heat, and salt. Tracer reconstruction for transport uses PPM with monotonic limiters but using third order interpolation for edge values. This reduces the size of the stencil which helps the computational efficiency of the transport scheme. The flux convergences, $\boldsymbol{\mathcal{N}}^{\gamma}_{\theta}$ and $\boldsymbol{\mathcal{N}}^{\gamma}_{S}$, provide subgrid scale neutral diffusion for the potential temperature and salinity, whereas $\delta_r J^{(z)}_{\theta}$ and $\delta_r J^{(z)}_{S}$ provide subgrid scale vertical diffusion as well as boundary fluxes. In the interior, both subgrid fluxes vanish when their respective tracers are spatially uniform, thus ensuring that the tracer equation reduces to the thickness equation when the tracer is uniform.

Parameterized subgrid scale advection from the submesoscale (**?**) and mesoscale (**?**) parameterizations are combined with the lateral advection of thickness and tracer, thus providing a residual mean advective transport for the scalar fields. Furthermore, we implement subgrid advective terms solely as lateral transports, thus interpreting them as layer bolus transport as appropriate for vertical Lagrangian models rather than a three-dimensional eddy-induced advection as appropriate for vertical Eulerian models (see **?** for details).

18.4 Equation of state

The equation of state, **??**, determines *in situ* density as a function of potential temperature, salinity, and pressure. We evaluate the pressure in the equation of state according to $-g\,\rho_0\,z$. Doing so maintains energetic consistency for the Boussinesq fluid according to section 2.4.3 of **?**. We make use of the **?** equation of state so that θ is potential temperature and S is the practical salinity. Although MOM6 has the more updated equation of state from **?**, the required changes for thermodynamic variables were implemented only after the basic model configuration was developed. Time constraints on model development prompted us to retain usage of **?** for OM4.

The freezing point of seawater is approximated as

$$T_f = -0.054S - 7.75 \times 10^{-08} p,$$

where p is in units of Pascals and S is in units of 1×10^{-3}. When the local temperature anywhere in the ocean column falls below the freezing point, the water-equivalent volume of ice is calculated and the fusion heat locally added back to the ocean to raise the liquid seawater temperature back to the freezing point. The frozen water and salt are sent to the sea-ice model.

Chapter 19

Filtering Timesteps

Filtering Timesteps

Chapter 20

Equation of State

Within MOM6, there is a wrapper for the equation of state, so that all calls look the same from the rest of the model. The equation of state code has to calculate not just in situ density, but also the compressibility and various derivatives of the density. There is also code for computing specific volume and the freezing temperature.

20.1 Linear Equation of State

Compute the required quantities with uniform values for $\alpha = \frac{\partial \rho}{\partial T}$ and $\beta = \frac{\partial \rho}{\partial S}$, (DRHO_DT, DRHO_DS in MOM_↩ input, also uses RHO_T0_S0).

20.2 Wright Equation of State

Compute the required quantities using the equation of state from **?**. This equation of state is in the form:

$$\alpha(s, \theta, p) = A(s, \theta) +$$

Chapter 21

Diagnostics

Controlling run-time diagnostics and how to add diagnostics to the code

MOM6 diagnostics are orchestrated via the FMS diag_manager, as for previous versions of MOM. However, because MOM6 is a general coordinate model, the model native-coordinae output can be less familiar to users of earlier generations of MOM. To alleviate this problem, MOM6 provides both "native" and "remapped" diagnostics; the former being diagnostics in the actual model coordinate space, and the latter in user-defined coordinates.

21.1 The "diag_table"

At run-time, diagnostics are controlled by the input file `diag_table` which is interpreted but the FMS package diag_manager.

The diag_table file has three kinds of section: Title, File and Field. The title section is mandatory and always the first. There can be multiple file and field sections, typically either in pairs or grouped in to all files and all fields, but always with the file section preceding the corresponding field section.

21.1.1 Title section

The first two lines are mandatory and comprise a line with a title and a line with six integers defining a base date against which time will be referenced.

```
"My ocean-only test case"
1900 1 1 0 0 0
```

21.1.2 File section

This section defines an arbitrary number of files that will be created. Each file is limited to a single rate of either sampling or time-averaging.

```
"file_name", output_freq, "output_freq_units", file_format, "time_axis_units", "time_axis_name"
```

- `file_name` : The name of the file that contains diagnostics at the given frequency (excluding the ".nc" extension).

- output_freq : The period between records in file_name, if positive. Special values of 0 mean write every time step and -1 write only at the end of the run.
- output_freq_units : The units in which output_freq is given. Valid values are "years", "months", "days", "hours", "minutes" or "seconds".
- file_format : Always set to 1, meaning netcdf.
- time_axis_units : The units to use for the time-axis in the file. Valid values are "years", "months", "days", "hours", "minutes" or "seconds".
- time_axis_name : The name of the time-axis (usually "Time").

Optional entries in the file line allow the generation of multiple files are intervals:

```
"file_name", output_freq, "output_freq_units", file_format, "time_axis_units", "time_axis_name"[,
      new_file_freq, "new_file_freq_units"[, "start_time"[, file_duration, "file_duration_units"]]]
```

- file_name : The base name of the file that contains diagnostics at the given frequency (excluding the ".nc" extension). The strings %yr, %mo, %dy, %hr %mi, %sc are expanded to the current year, month, day, hour, minute and second respectively, with new files created every new_file_freq.
- new_file_freq : The period between generation of new files.
- new_file_freq_units : The units in which new_file_freq is given.
- start_time, file_duration, file_duration_units : Even finer grain control of output files.

21.1.3 Field section

An arbitrary number of lines, one per diagnostic field:

```
"module_name", "field_name", "output_name", "file_name", "time_sampling", "reduction_method", "
      regional_section", packing
```

- module_name : Name of the component model. For native ocean variables this should be "ocean_model". See Vertically remapped diagnostics for non-native vertical-grid diagnostics in the ocean model.
- field_name : The name of the variable as registered in the model.
- output_name " The name of the variable as it will appear in the file. This is usually the same as the field_name but can be used to rename a diagnostic.
- file_name : One of the files defined above in the section File section.
- time_sampling : Always set to "all".
- reduction_method : "none" means sample or snapshot. "average" or "mean" performs a time-average. "min" or "max" diagnose the minium or maxium over each time period.
- regional_section : "none" means global output. A string of six space separated numbers, "lat_min, lat_max, lon_min, lon_max, vert_min, vert_max", limits the diagnostic to a region.
- packing : Data representation in the file. 1 means "real∗8", 2 means "real∗4", 4 mean 16-bit integers, 8 means 1-byte.

21.1.4 Example

```
"OM4 1/4 degree"
1900 1 1 0 0 0

# Static file
"ocean_static", -1, "months", 1, "days", "time" # ocean_static is a protected name. Do not change this line
      .
"ocean_model", "deptho",    "deptho",    "ocean_static", "all", "none", "none", 2
"ocean_model", "geolon",    "geolon",    "ocean_static", "all", "none", "none", 2
"ocean_model", "geolat",    "geolat",    "ocean_static", "all", "none", "none", 2
"ocean_model", "wet",       "wet",       "ocean_static", "all", "none", "none", 2

# High-frequency file
"surf_%4yr_%3dy", 1, "hours", 1, "days", "time", 1, "months"
"ocean_model","SSH","SSH","surf_%4yr_%3dy","all","none","none",2

# Daily averages
"ocean_daily", 1, "days", 1, "days", "time"
"ocean_model", "tos", "tos", "ocean_daily", "all", "mean", "none",2

# Monthly averages
"ocean_month", 1, "months", 1, "days", "time"
"ocean_model", "thetao", "thetao", "ocean_month", "all", "mean", "none",2

# Annual averages
"ocean_annual", 12, "months", 1, "days", "time"
"ocean_model", "thetao", "thetao", "ocean_annual", "all", "mean", "none",2

# Vertical section
"ocean_Bering_Strait", 5, "days", 1, "days", "time"
"ocean_model", "thetao","thetao", "ocean_Bering_Strait", "all", "mean", "-171.4 -168.7  66.1 66.1 -1 -1",2
```

21.2 Native diagnostics

The list of available diagnostics is dependent on the particular configuration of the model. For this reason the model writes a record of the available diagnostic fields at run-time into a file "available_diags.∗". See, for example, `available_diags.000000` for the global_ALE z-coordinate ocean-only test case.

Diagnostic fields in the module "ocean_model" refer to the native variables or diagnostics in the native grid. Since the model can be run in arbitrary coordinates, say in hybrid-coordinate mode, then native-space diagnostics can be potentially confusing. Native diagnostics are useful when examining exactly what the model is doing, or if the vertical coordinate of the model is configured to be a natural coordinate such as pure isopycnal or z∗ geopotential.

21.3 Vertically remapped diagnostics

Alternative vertical coordinates can be configured for diagnostic purposes only.

The run-time parameter `NUM_DIAG_COORDS` controls how many diagnostic coordinates to use.

The run-time parameter `DIAG_COORDS` defines the mapping between each coordinate, the name of the module in the diag_table and run-time parameter names that define the coordinate. A list of string tuples, separated by commas, with each tuple in the form of MODULE_SUFFIX PARAMETER_SUFFIX COORDINATE_NAME. `MOD↩ ULE_SUFFIX` is the string appended to "ocean_model" to create a module in the diag_table. `PARAMETER_SU↩ FFIX` is the string appended to "DiAG_COORD_DEF", and other parameters, used to control the generation of the named coordinate. `COORDINATE_NAME` is a name understood by the MOM6 regridding module. Valid examples are "ZSTAR", "SIGMA", "RHO", etc.

By default, `NUM_DIAG_COORDS=1` and `DIAG_COORDS="z Z ZSTAR"`, meaning the module "ocean_↩ model_z" provides diagnostics in "z∗" coordinates and uses the parameter `DIAG_COORD_DEF_Z`.

For example, multiple z∗-coordinates could be used for diagnostics with

```
NUM_DIAG_COORDS = 2
DIAG_COORDS = "z 01 ZSTAR,abc 02 ZSTAR"
DIAG_COORD_DEF_01 = "WOA09"
DIAG_COORD_DEF_02 = "UNIFORM:10,20."
```

would create the diag_manager modules "ocean_model_z" and "ocean_model_abc".

The above is equivalent to

```
NUM_DIAG_COORDS = 2
DIAG_COORDS = "z ZA ZSTAR,abc ZB ZSTAR"
DIAG_COORD_DEF_ZA = "WOA09"
DIAG_COORD_DEF_ZB = "UNIFORM:10,20."
```

To obtain a diagnostic of monthly-averaged potential temperature in both these coordinate systems the diag_table must include the lines

```
"ocean_month_z", 1, "months", 1, "days", "time"
"ocean_month_abc", 1, "months", 1, "days", "time"
"ocean_model_z", "temp", "temp", "ocean_month_z", "all", "mean", "none",2
"ocean_model_abc", "temp", "temp", "ocean_month_abc", "all", "mean", "none",2
```

21.3.1 Diagnostic vertical coordinates

For each of the NUM_DIAG_COORDS vertical coordinates listed in DIAG_COORDS the corresponding DIAG_C↩OORD_DEF_% parameter must be provided. It can take the following values:

- PARAM : In this case, a corresponding parameter DIAG_COORD_RES_% is read that lists the deltas for each level in the coordinate. For example, DIAG_COODS="z Z ZTAR", DIAG_COORD_DEF_Z="PARAM", DIAG_COORD_RES_Z=5,10,10,15 creates z∗-level with 5,10,10,15 meters thicknesses.
- UNIFORM : Uniform distribution down to the maximum depth of the model using the same number of levels as he model.
- UNIFORM:N : Uniform distribution down to the maximum depth of the model using N levels.
- UNIFORM:N,D : Uniform distribution of N levels with thickness D.
- FILE:filename,varname : Reads vector of coordinate thicknesses from field "varname" from file "filename".
- FILE:filename,interfaces=varname : Reads vector of coordinate positions from field "varname" from file "filename".
- WOA09 : Z-levels that correspond to the World Ocean Atlas, 2009, standard levels down to and including the maximum depth of the model.
- WOA09:N : The first N levels of the World Ocean Atlas, 2009, standard levels.

21.4 APIs for diagnostics

The multiple diagnostic-coordinates are implemented in a layer that sits on top of the FMS diag_manager known as the mom6_diag_mediator.

A diagnostic is registered with register_diag_field() which is an API that looks similar to the FMS diag_manager routine of the same name:

```
diag_id = register_diag_field(module_name, diag_name, axes, ...)
```

The MOM6 version of this routine optionally allows the specification of CMOR names in addition to the native names which then registers the diagnostic twice, once with each name.

For each of the native and CMOR names, the diagnostic is registered in the native module "ocean_model", corresponding to the native model coordinate, and a module associated with each of the diagnostic coordinates.

For each diagnostic coordinate, a horizontally-averaged diagnostic is also registered.

In all, for each 3D diagnostic, the are 2 + 4N diagnostics registered, where N is the number of diagnostic coordinates. As a result, the global_ALE examples have of order 900 total diagnostics available in the shipped configuration.

The data is made available to the diag_manager via a call to post_data() which is a wrapper that does all the vertical remapping before calling FMS's send_data():

```
call post_data(diag_id, data, diag_control)
```

21.4.1 Artifacts of posting frequency for diagnostics

Variables area "posted" for i/o or averaging to the diag_manager (via MOM6's diag_mediator) at different frequencies relative to each other. This is because the MOM6 algorithm takes the form of nested sub-cycles with different time-steps in each loop (e.g. barotropic solver with dynamics). A consequence of this is that a time average of a related quantities may appear to be inconsistent since the diagnostic posted with higher frequency may not vary linearly between the end-points seen on the longer time-step. The differences are usually small, but if you see large differences it might indicate you should re-examine the time-steps used for the various sub-cycles.

Chapter 22

Domain Decomposition

22.1 Domain Decomposition

MOM6 supports serial, OpenMP, and MPI computations, with the user choosing between them at run time. All are accomplished through domain decomposition in the horizontal. All of the horizontal operations are explicit with a relatively small footprint, so the tiling is a logical choice. Some goals in the parallel design of MOM6 were:

- Don't hard-code the number of processes.
- MPI and OpenMP share the same basic structure.
- Don't break the serial optimizations.
- Same result as serial code for any number of processes.
- Portability - able to run on any (Unix) system.

The whole horizontal MOM6 grid is shown in Declaration of variables. The computations are done over the cells inside the darker line; the cells are numbered 1 to NIGLOBAL in the x-direction and 1 to NJGLOBAL in the y-direction. Those looking ahead to running in parallel would be wise to include factors of two and three in their choice of NIGLOBAL and NJGLOBAL when building new grids. MOM6 will run in parallel with any values of these, but the computations might not be load-balanced.

22.2 Wide Halos

Chapter 23

Global Orthogonal Grids

Global Orthogonal Grids

23.1 Dipole Grids

23.2 Tripole Grids

Chapter 24

Horizontal indexing and memory

Conventions for staggering of variables and loops over 2d/3d arrays

MOM6 is written in Fortran90 and uses the `i,j,k` order of indexing. `i` corresponds to the fastest index (stride-1 in memory) and thus should be the inner-most loop variable. We often refer to the i-direction as the x- or zonal direction, and similarly to the j-direction as y- or meridional direction. The model can use curvilinear grids/coordinates in the horizontal and so these labels have loose meanings but convenient.

24.1 Loops and staggered variables

Many variables are staggered horizontally with respect to each other. The dynamics and tracer equations are discretized on an Arakawa C grid. Staggered variables must still have integer indices and we use a north-east convention centered on the h-points. These means a variable with indices `i,j` will be either collocated, to the east, to the north, or to the north-east of the h-point with the same indices.

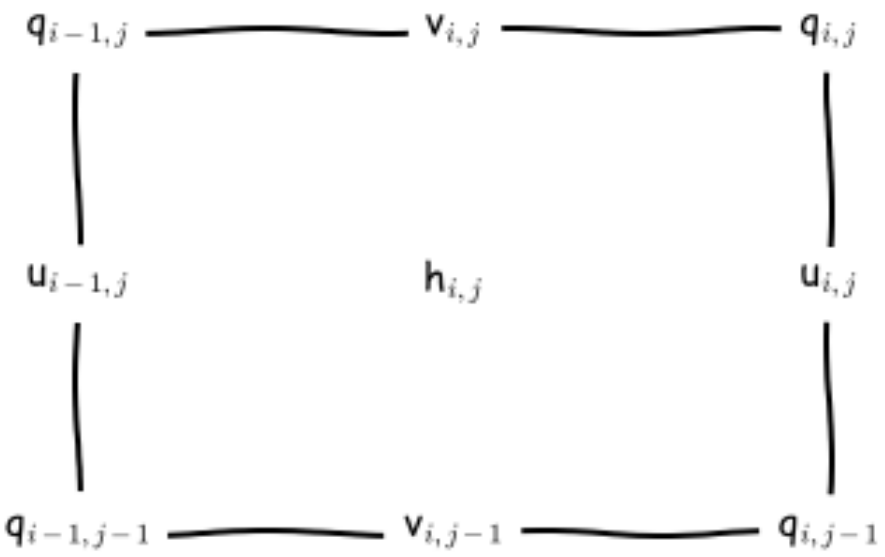


Figure 24.1 MOM6 uses an Arakawa C grid staggering of variables with a North-East indexing convention. "Cells" refer to the control volumes around tracer- or h-point located variables unless labelled otherwise.

24.1.1 Soft convention for loop variables

To ease reading the code we use a "soft" convection (soft because there is no syntax checking) where an upper-case index variable can be interpreted as the lower-case index variable plus $\frac{1}{2}$.

For example, when a loop is over h-points collocated variables

- the do-loop statements will be for lower-case `i,j` variables
- references to h-point variables will be `h(i,j)`, `D(i+1,j)`, etc.
- references to u-point variables will be `u(I,j)` (meaning $u_{i+\frac{1}{2},j}$), `u(I-1,j)` (meaning $u_{i-\frac{1}{2},j}$), etc.
- references to v-point variables will be `v(i,J)` (meaning $v_{i,j+\frac{1}{2}}$), `u(I-1,j)` (meaning $u_{i,j-\frac{1}{2}}$), etc.
- references to q-point variables will be `q(I,J)` (meaning $q_{i+\frac{1}{2},j+\frac{1}{2}}$), etc.

In contrast, when a loop is over u-points collocated variables

- the do-loop statements will be for upper-case `I` and lower-case `j` variables
- the expression $u_{i+\frac{1}{2},j}(h_{i,j} + h_{i+1,j})$ is `u(I,j) * ( h(i,j) + h(i+1,j).`

24.2 Declaration of variables

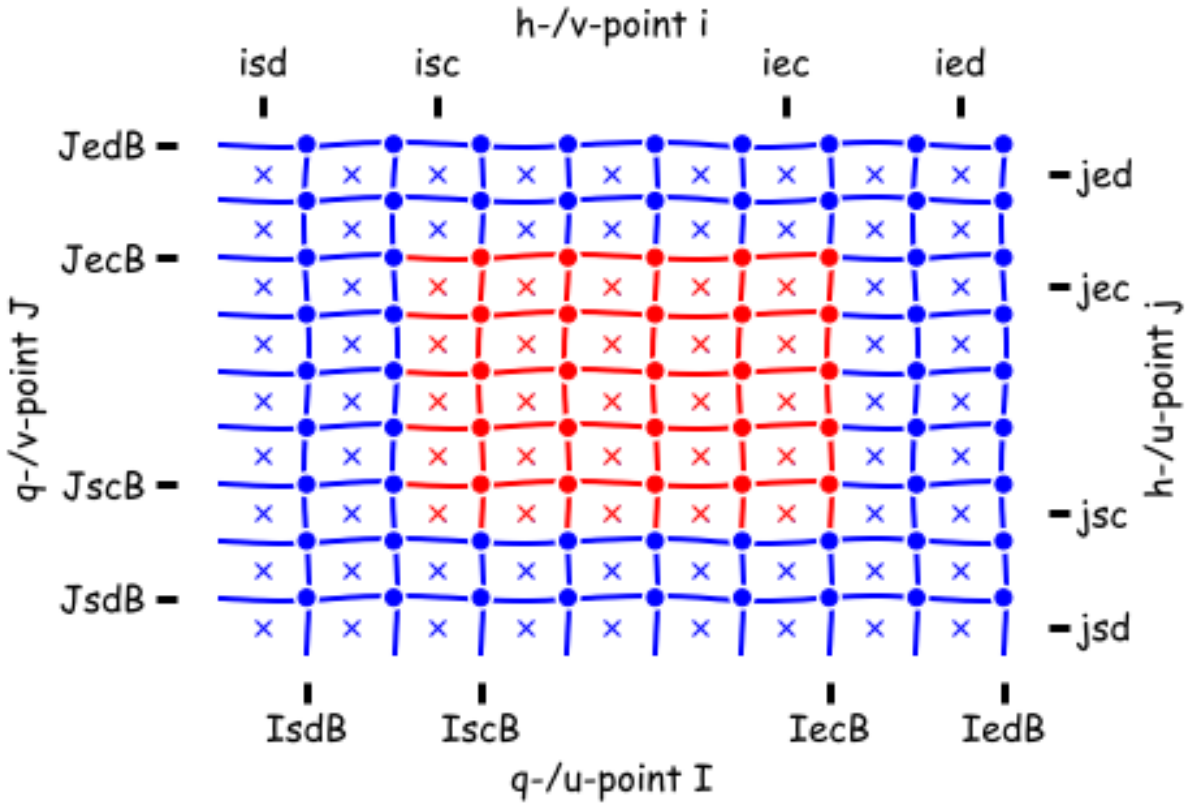


Figure 24.2 Non-symmetric mode: All arrays are declared with the same shape (isd:ied,jsd:jed).

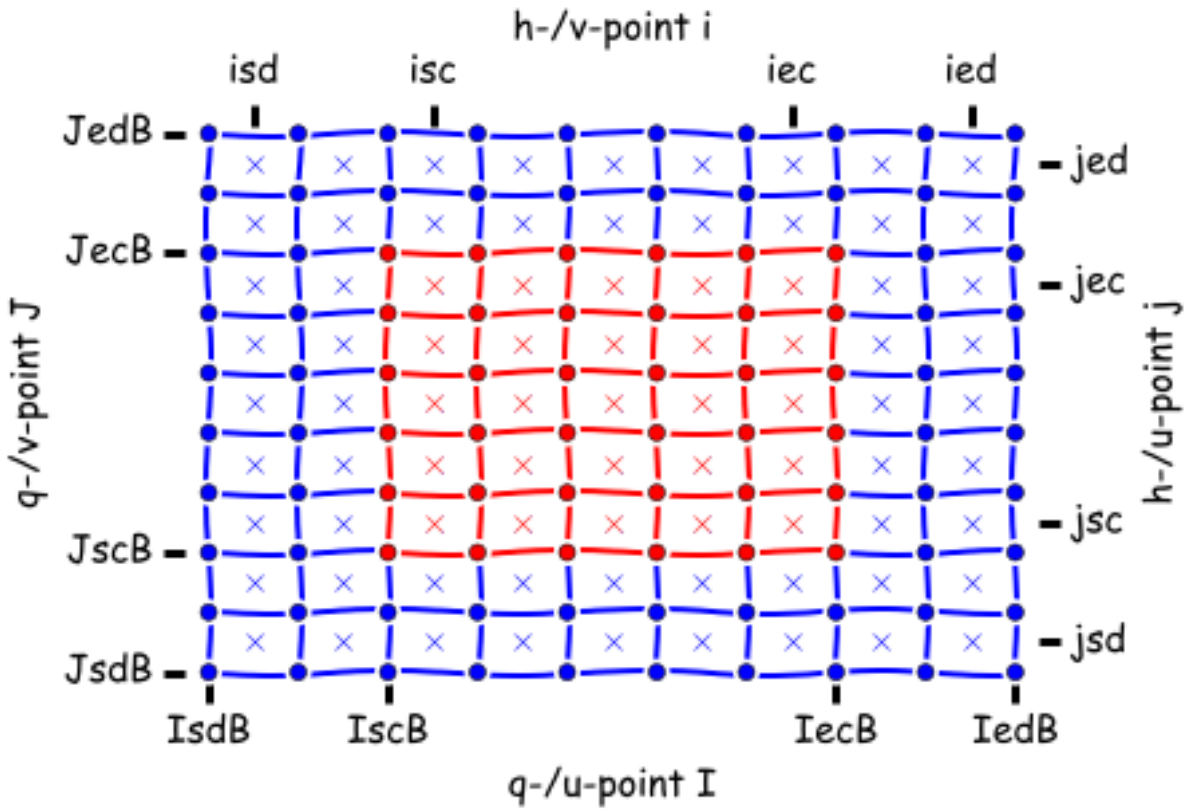


Figure 24.3 Symmetric mode: Arrays have different shapes depending on their staggering location on the Arakawa C grid.

A field is described by 2D or 3D arrays which are distributed across parallel processors. Each processor only sees a small window of the global field. The processor "owns" the computational domain (red in above figure) but arrays are extended horizontally with halos which are intermittently updated with the values from neighboring processors. The halo regions (blue in above figure) may not always be up-to-date. Data in halo regions (blue in above figure) will be overwritten my mpp_updates.

MOM6 has two memory models, "symmetric" and "non-symmetric". In non-symmetric mode all arrays are given the same shape. The consequence of this is that there are fewer staggered variables to the south-west of the computational domain. An operator applied at h-point locations involving u- or v- point data can not have as wide a stencil on the south-west side of the processor domain as it can on the north-east side.

In symmetric mode, declarations are dependent on the variables staggered location on the Arakawa C grid. This allows loops to be symmetric and stencils to be applied more uniformly.

In the code, declarations are consistent with the symmetric memory model. The non-symmetric mode is implemented by setting the start values of the staggered data domain to be the same as the non-staggered start value.

The horizontal index type (mom_hor_index::hor_index_type) provides the data domain start values. The values are also copied into the mom_grid::ocean_grid_type for convenience although we might deprecate this convenience in the future.

Declarations of h-point data take the form:

- `real, dimension(HI%isd:HI%ied, HI%jsd:HI%jed) :: D !< Depth at h-points (m)`

- `real, dimension(HI%isd:HI%ied, HI%jsd:HI%jed, GV%ke) :: h !< Layer thickness (H units)`

Declarations of u-point data take the form:

- `real, dimension(HI%IsdB:HI%IedB, HI%jsd:HI%jed) :: Du !< Depth at u-points (m)`
- `real, dimension(HI%IsdB:HI%IedB, HI%jsd:HI%jed, GV%ke) :: h !< Zonal flow (m/s)`

Declarations of v-point data take the form:

- `real, dimension(HI%isd:HI%ied, HI%JsdB:HI%JedB) :: Dv !< Depth at v-points (m)`
- `real, dimension(HI%isd:HI%ied, HI%JsdB:HI%JedB, GV%ke) :: h !< Zonal flow (m/s)`

Declarations of q-point data take the form:

- `real, dimension(HI%IsdB:HI%IedB, HI%JsdB:HI%JedB) :: Dq !< Depth at q-points (m)`
- `real, dimension(HI%IsdB:HI%IedB, HI%JsdB:HI%JedB, GV%ke) :: vort !< Vertical componentof vorticity (s-1)`

The file MOM_memory_macros.h provides the macros `SZI_`, `SZJ_`, `SZIB_` and `SZJB_` that help make the above more concise:

- `real, dimension(SZI_(HI), SZJ_(HI)) :: D !< Depth at h-points (m)`
- `real, dimension(SZIB_(HI), SZJ_(HI)) :: Du !< Depth at u-points (m)`
- `real, dimension(SZI_(HI), SZJB_(HI)) :: Dv !< Depth at v-points (m)`
- `real, dimension(SZIB_(HI), SZJB_(HI)) :: Dq !< Depth at q-points (m)`

See MOM_memory_macros.h for the complete list of macros used in various memory modes.

24.3 Calculating a global index

For the most part MOM6 code should be independent of an equivalent absolute global index. There are exceptions and when the global index of a cell `i, j` is needed is can be calculated as follows:

```
i_global = i + HI%idg_offset
```

Before the mom_hor_index::hor_index_type was introduced, this conversion was done use variables in the mom↩ _grid::ocean_grid_type:

```
i_global = (i-G%isd) + G%isd_global
```

which is no longer preferred.

Note that a global index only makes sense for a rectangular global domain. If the domain is a Mosaic of connected tiles (e.g. size tiles of a cube) the global indices (i,j) become meaningless.

Chapter 25

Parallel I/O

Parallel I/O

The model can be told to write a different output file per process. This may or may not save time, and is a bad idea on Lustre filesystems. If the model is writing individual files per process, one can combine them using the mppnccombine program from the FRE-nctools package.

Chapter 26

Regional Orthogonal Grids

Regional Orthogonal Grids

26.1 Map Projections

26.2 Open Boundary Segments

Chapter 27

Run-time Parameter System

How run-time parameters work in MOM6

MOM6 has an extensive set of parameters that are set at run-time by parsing an input file. Many parameters have default values and are not required to be in the input file, although there are a number of parameters that must be set for the model to run. The numerous examples provided with the MOM6 code mostly differ in their run-time parameters (although some add other components, like sea-ice), and comparison between these examples is an excellent way to get a broad overview of many of MOM6's parameters and how they might be set.

27.1 Getting parameters into MOM6

Run-time parameters are provided to the model in two phases:

1. A very small set of logistical parameters are read as namelist variables from the FMS parameter file `input.nml`. One of these logistical parameters is a list of ascii files that contain all the other run-time parameters.
2. All of the above-named parameter files are scanned for MOM6 model parameters, default values assigned and replaced, conflicts detected and various parameter summaries logged to files and/or the standard output.

27.1.1 Namelist parameters (‘input.nml‘)

All FMS derived MOM6 parameters reside in the namelist `MOM_input_nml` in the file `input.nml`. The parameters are:

- `input_filename` - If equal to "n" will run a new run (i.e. will not read a restart file). If equal to "r" MOM6 will attempt to read a restart file.
- `parameter_filename` - A list of file names containing the MOM6 internal run-time parameters. Typically `param_files="MOM6_input","MOM6_override"` where the file MOM6_input contains all the non-default parameters that define a "baseline" experiment and MOM6_override will be either empty (for baseline) or contain a few parameters that define a "derived" experiment (that differs from the baseline). This helps keep the parameter lists concise and enables easy comparison of parameters in related experiments.
- `restart_input_dir`, `restart_output_dir`, and `output_directory` - These specify the directories for reading input files, writing restart files, and writing many non-restart files.

27.1.2 Other MOM6-relevant FMS parameters

The namelist ocean_solo_nml may have the integer parameters secs, hours, days, months and years, which dictate how long the FMS ocean driver will try to run the model each run-segment.

27.1.3 MOM6 parameter file syntax

The general syntax for an entry in a MOM6 parameter file is

```
[!]#[override] PARAMETER_NAME = value[,value][...][!comments]
```

Parameter names must be constructed from the characters `[A-Za-z0-9_]` and by soft convention are upper case. The `!` character is a remark or comment indicator; all subsequent text on that line is ignored.

Parameters that are not specified in the parameter files may assume a default value. It is not an error to specify a parameter more than once with the same value. It is an error to specify different values.

The keyword #override indicates that this parameter specification takes precedence over other specifications. It is **not** an error to have two #override specifications for a single parameter with the same values. It is an error to have two #override statements with different values.

Some illustrations:

```
DO_THIS = True  ! Set the Boolean to .TRUE.
DO_THAT = False ! Set the Boolean to .FALSE.
NXYZ = 5  ! Set the value to NXYZ to 5
HALF = 0.5  ! Set the value of HALF to 0.5
NAME = "abc"  ! Set the string NAME to 'abc'
VECTOR = 1.0,2.0  ! Set the array VECTOR to [1.0, 2.0]
NAMES = 'abc','xyz' ! Set the strings NAMES to 'abc','xyz'
#override DO_THIS = False ! Set the Boolean to .FALSE., ignoring the above specification
#override HALF = 0.25 ! Set the value of HALF to 0.25, ignoring the above value
#override HALF = 0.25 ! Same as the above value of HALF to 0.25 so is accepted
```

27.1.4 Logging of parameters

The subroutine that reads MOM6 parameters has also serves to log every parameter to a file set by DOCUMENT_FILE, usually "MOM6_parameter_doc". In addition to the name of the variable being read, these calls contain a brief description, along with a description of the units and the default value (if any) or an indication that there is no default and that the variable must be present. For example, `DT` is always required to be present:

```
call get_param(param_file, module, "DT", CS%dt, &
             "The (baroclinic) dynamics time step.  The time-step that \n"//&
             "is actually used will be an integer fraction of the \n"//&
             "forcing time-step (DT_FORCING in ocean-only mode or the \n"//&
             "coupling timestep in coupled mode.)", units="s", &
             fail_if_missing=.true.)
```

At run-time, two levels of logging are performed, depending on the value of the parameter `MINIMAL_DOCUMENTATION`:

- (TRUE) The end result of the combination of default values, assignments and overrides are recorded with default and current values, description and units, for all parameters.
- (FALSE) The minimal list of required and non-default value parameters are recorded with current values, description and units only for those parameters needed to reproduce the configuration.

Either of the generated logging files can be used as inputs and yield the same configuration.

In addition, there are also calls that log derived quantities (e.g., a time-step that is derived from a CFL number, or the full path to an input file) without reading anything in.

27.1.5 Error checking of parameters and parameter files

There are several techniques that are used for error checking on MOM6 parameters:

- Some parameters have internal error messages if they are set to nonsensical values.
- No parameter can be set twice inconsistently without an explicit #override specification.
- If the run-time parameter REPORT_UNUSED_PARAMS is true, a warning will be issued for any entries in the input parameter files that are not read in, for instance if they are misspelled.
- Setting the run-time parameter FATAL_UNUSED_PARAMS to true causes a fatal error that will bring down the model if there are any unused entries in the input parameter files.

Chapter 28

Testing

MOM6 Validation and Verification

In the software engineering world, people talk about validation and verification of their codes. Verification is the confirmation of design specifications, such as:

- Does it compile on the target platform?
- Is it dimensionally consistent?
- Do answers change with the number of processes?
- Do answers change after a restart?

Validation is a little trickier:

- Does the model meet operational needs?
- Does it produce realistic simulations?
- Are relevant physical features present?
- Can I reproduce my old simulations?

There are a number of ways in which MOM6 is tested before each commit, especially commits to the shared dev/main branch.

28.1 Travis Testing

When pushing code to github, it is possible to set it up so that testing is performed automatically by travis. For MOM6, the .travis.yml file is executed, causing the code to be compiled and then run on all the tests in the .testing directory. It is also possible to run these tests on your local machine, but you might have to do some setup first. See

See also

../../../.testing/README.md for more information.

28.2 Consortium Testing

For commits to the dev/main branch, there is an opportunity for all consortium members to weigh in on proposed updates. A view of the consortium is shown here:

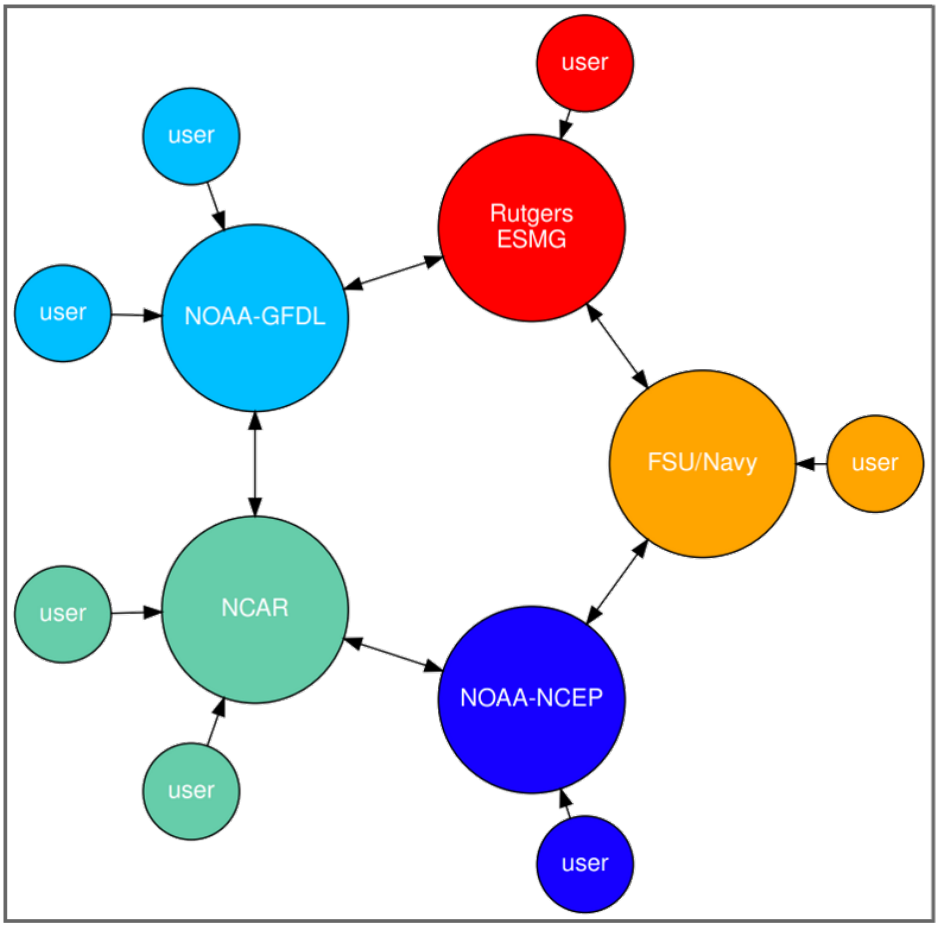


Figure 28.1 The MOM6 consortium.

Each group is expected to have their own tests and to keep track of expected answers when these tests are run to be compared to prior answers after the code is updated. Answer-changing updates have to be evaluated carefully, though there are circumstances in which the new answers may well be "better".

28.3 Novel Tests

There are two classes of tests which MOM6 performs within the .testing suite which could be considered unusual, but which can be quite useful for finding bugs.

28.3.1 Scaling tests

The equations of motion can be multiplied by factors of two without changing answers. One can use that to scale each of six units by a different factor of two to check for consistent use of units. For instance, the equation:

$$u^{n+1} = u^n + \Delta t \times \mathcal{F}$$

can be scaled as:

$$2^{L-T} u^{n+1} = 2^{L-T} u^n + 2^T \Delta t \times 2^{L-2T} \mathcal{F}$$

MOM6 has been recoded to include six different scale factors:

Table 28.1 Dimensional scale factors

Unit	Scaling	Name
s	T	Time
m	L	Horizontal length
m	H	Layer thickness
m	Z	Vertical length
kg/m3	R	Density
J/kg	Q	Enthalpy

You can add these integer scaling factors through the runtime parameters X_RESCALE_POWER, where X is one of T, L, H, Z, R, or Q. The valid range for these is -300 to 300.

When adding contributions to MOM6, this coding style with the scale factors must be maintained. For example, if you add new parameters to read from the input file:

```
call get_param(..., "DT", ... , scale=US%s_to_T)
```

This is also required for explicit contants, though we are trying to move those out of the code:

```
ustar = 0.01 * US%m_to_Z * US%T_to_s
```

or for adding diagnostics:

```
call register_diag_field(..., "u", ... , &
      conversion=US%L_T_to_m_s)
```

See also

mom_unit_scaling

28.3.2 Rotational tests

By setting the runtime option ROTATE_INDEX to True, the model rotates the domain by some number of 90 degree turns. This option can be used to look for bugs in which east-west operations do not match north-south operations. It changes the order of array elements as shown here:

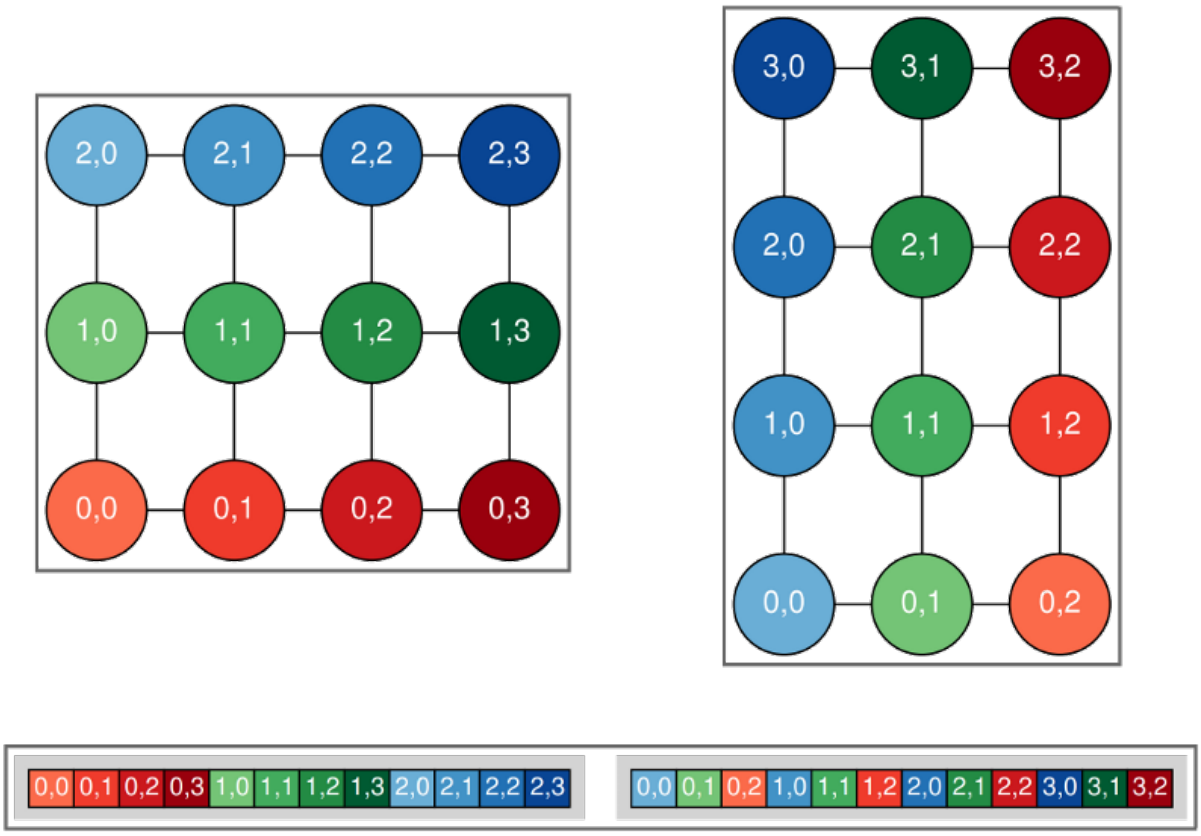


Figure 28.2 The original non-rotated domain is shown on the left while the right shows the domain rotated counterclockwise by 90 degrees. The array values are shown by the (invariant) colors, while the array indices (and dimensions) change.

It only currently runs in serial mode. One can ask for rotations of 90, 180, or 270 degrees, but only 90 degree turns are supported if there are open boundaries.

Because order matters in numerical computations, care must be taken for four-way averages to match between rotated and non-rotated runs. Say you want to compute the following quantity:

$$\phi_{i,j}^{(c)} = \frac{1}{4}(\phi_A + \phi_B + \phi_C + \phi_D)$$

as shown in this diagram:

You might write this first as:

$$\frac{1}{4}((\phi_A + \phi_B) + (\phi_C + \phi_D))$$

as shown on the left in this figure:

However, the round-off errors could give differing answers when rotated. Instead, you want to group the terms on the diagonal as shown in the right of the above figure and here:

$$\frac{1}{4}((\phi_A + \phi_D) + (\phi_B + \phi_C))$$

Chapter 29

Vertical Orthogonal Grids

Vertical Orthogonal Grids

29.1 Layered

29.2 Z-Star

29.3 Sigma

29.4 Rho

29.5 Hybrid

Chapter 30

The K-Profile Parameterization

The K-Profile Parameterization (KPP) of **?** is implemented via the Community Vertical Mixing package, CVMix, which is called directly by this module.

The formulation and implementation of KPP is described in great detail in the CVMix manual (written by our own Steve Griffies).

30.1 KPP in a nutshell

Large et al., **?**, decompose the parameterized boundary layer turbulent flux of a scalar, s, as

$$\overline{w's'} = -K\partial_z s + K\gamma_s(\sigma),$$

where $\sigma = -z/h$ is a non-dimensional coordinate within the boundary layer of depth h. K is the eddy diffusivity and is a function of position within the boundary layer as well as a function of the surface forcing:

$$K = hw_s(\sigma)G(\sigma).$$

Here, w_s is the vertical velocity scale of the boundary layer turbulence and $G(\sigma)$ is a "shape function" which is described later. The last term is the "non-local transport" which involves a function $\gamma_s(\sigma)$ that is matched to the forcing but is not actually needed in the final implementation. Instead, the entire non-local transport term can be equivalently written

$$K\gamma_s(\sigma) = C_s G(\sigma) Q_s$$

where Q_s is the surface flux of s and C_s is a constant. The vertical structure of the redistribution (non-local) term is solely due to the shape function, $G(\sigma)$. In our implementation of KPP, we allow the shape functions used for K and for the non-local transport to be chosen independently.

The particular shape function most widely used in the atmospheric community is

$$G(\sigma) = \sigma(1-\sigma)^2$$

which satisfies the boundary conditions $G(0) = 0$, $G(1) = 0$, $G'(0) = 1$, and $G'(1) = 0$. Large et al, 1994, alter the function so as to match interior diffusivities but we have found that this leads to inconsistencies within the formulation (see google groups thread Extreme values of non-local transport). Instead, we use either the above form, or even simpler forms that use alternative upper boundary conditions.

The KPP boundary layer depth is a function of the bulk Richardson number, Rib. But to compute Rib, we need the boundary layer depth. To address this circular logic, we compute Rib for each vertical cell in a column, assuming the BL depth equals to the depth of the given grid cell. Once we have a vertical array of Rib(k), we then call the OBLdepth routine from CVMix to compute the actual OBLdepth. We optionally then "correct" the OBLdepth by cycling through once more, this time knowing the OBLdepth from the first pass. This "correction" step is not used by NCAR. It has been found in idealized MOM6 tests to not be necessary.

See also

kpp_calculate(), kpp_applynonlocaltransport()

Chapter 31

Tracer Advection

Tracer transport schemes

MOM6 implements a generalised tracer advection scheme, which is a combination of the modified flux advection scheme **?** with reconstructed tracer distributions. The tracer distributions may be piecewise linear (PLM) or piecewise parabolic (PPM), which may itself use either the **?** (CW84) or **?** (H3) reconstruction.

31.1 Flux advection

The modified flux advection scheme preserves the tracer mixing ratio in a cell across directional splitting by accounting for changes in mass changes. Fluxes are applied to alternate directions in turn, restricting the applied flux so as not to evacuate all mass out of a cell. Because of this, we need to know the stencil used during the calculation of the reconstruction. Every iteration of the splitting algorithm, cells at the edge of a processor's data domain are invalidated. When this invalidation region extends below the halo, a group pass is required to refresh the halo. A larger stencil (such as for the CW84 reconstruction) therefore introduces more frequent updates, and may impact performance.

31.2 Tracer reconstruction

While MOM6 only carries the mean tracer concentration in a cell, a higher order reconstruction is computed for the purpose of advection. Reconstructions are also modified to ensure that monotonicity is preserved (i.e. spurious minima or maxima cannot be introduced).

The piecewise linear (PLM) reconstruction uses the monotonic modified van Leer scheme **?**. One might think to use the average of the one-sided differences of mean tracer concentration within a cell to calculate the slope of the linear reconstruction, however this method guarantees neither monotonicity, nor positive definiteness. Instead, the method is locally limited to the minimum of this average slope and each of the one-sided slopes, i.e.

$$\Delta\Phi_i = \min\left\{ \left|[\Delta\Phi_i]_{\text{avg}}\right|, 2\left(\Phi_i - \Phi_i^{\text{min}}\right), 2\left(\Phi_i^{\text{max}} - \Phi_i\right) \right\}$$

(where Φ_i^{min} is the minimum in the 3-point stencil).

In a PPM scheme (PPM Advection Scheme), for a cell with mean tracer concentration Φ_i, the values at the left and right interfaces, $\Phi_{L,i}$ and $\Phi_{R,i}$ must be estimated. First, an interpolation is used to calculate $\Phi_{i-1/2}$ and $\Phi_{i+1/2}$. These values are then modified to preserve monotonicity in each cell, which introduces discontinuities between cell edges (e.g. $\Phi_{R,i}$ and $\Phi_{L,i+1}$).

The reconstruction $\Phi_i(\xi)$ then satisfies three properties:

- total amount of tracer is conserved, $\int_{\xi_{i-1/2}}^{\xi_{i+1/2}} \Phi_i(\xi') \, \mathrm{d}\xi' = \Phi_i$
- left interface value matches, $\Phi(\xi_{i-1/2}) = \Phi_{L,i}$
- right interface value matches, $\Phi(\xi_{i+1/2}) = \Phi_{R,i}$

There are two methods of reconstruction for a piecewise parabolic (PPM) profile. They differ in the estimate of interface values $\Phi_{i+1/2}$ prior to monotonicity limiting. The CW84 scheme makes use of the limited slope $\Delta\Phi_i$ from PLM, above. This has the effect of requiring a larger stencil for each reconstruction. On the other hand, the H3 scheme reduces the requirement of this stencil, by only examining the tracer concentrations in adjacent cells, at the same time reducing order of accuracy of the reconstruction.

Chapter 32

Discrete Tracer Transport Equations

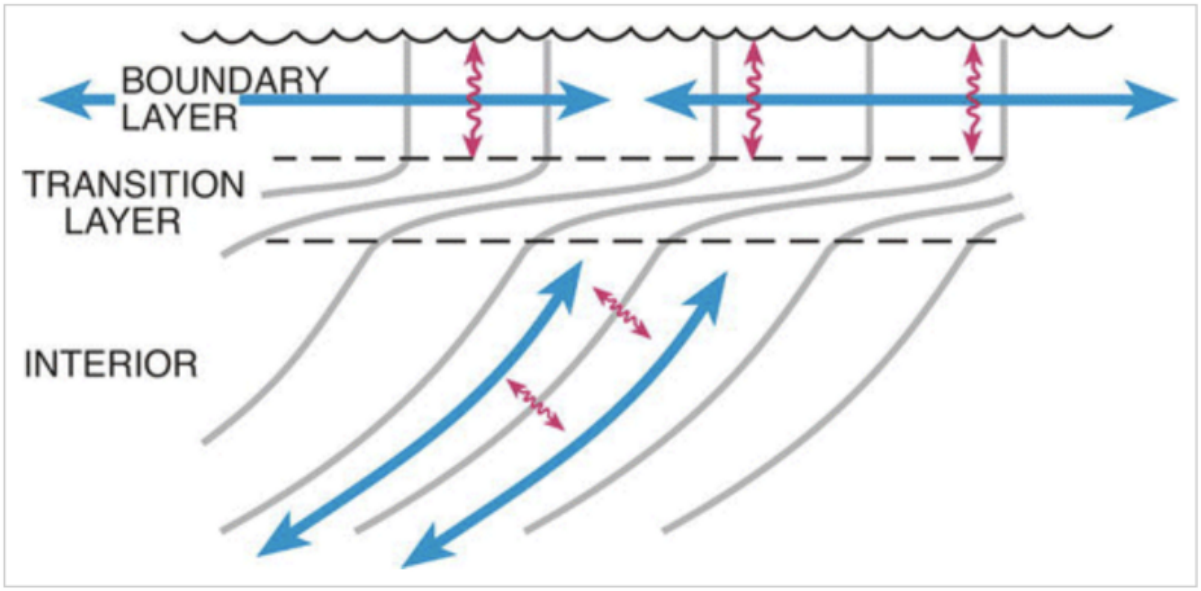
Discrete Tracer Transport Equations

Chapter 33

Horizontal Diffusion

Horizontal diffusion of tracers

Lateral mixing due to mesoscale eddies is believed to occur according to this figure:



From: Ferrari et al., 2008

Figure 33.1 Horizontal surface boundary layer fluxes and interior epineutral fluxes.

We start by describing an implementation of the mixing in the interior and then introduce a surface mixed layer implementation. A bottom mixed layer implementation is planned for the future.

33.1 Epineutral Diffusion

For the interior of the ocean, we would like to have horizontal diffusion with the following properties:

- Suitable for general coordinate models

- Preserves extrema
- Has no need for regularization or tapering (such as needed by rotated mixing tensors)

The algorithm used in MOM6 is described by **?** and will be introduced here. The aim is to allow lateral mixing of tracers within isopycnal layers. It is appropriate for the adiabatic interior of the ocean while a lateral mixing scheme for the surface boundary layer is described below.

Before presenting this scheme, a quick review of polynomial reconstructions is in order. Some choices for the vertical representation of a finite volume quantity are shown here:

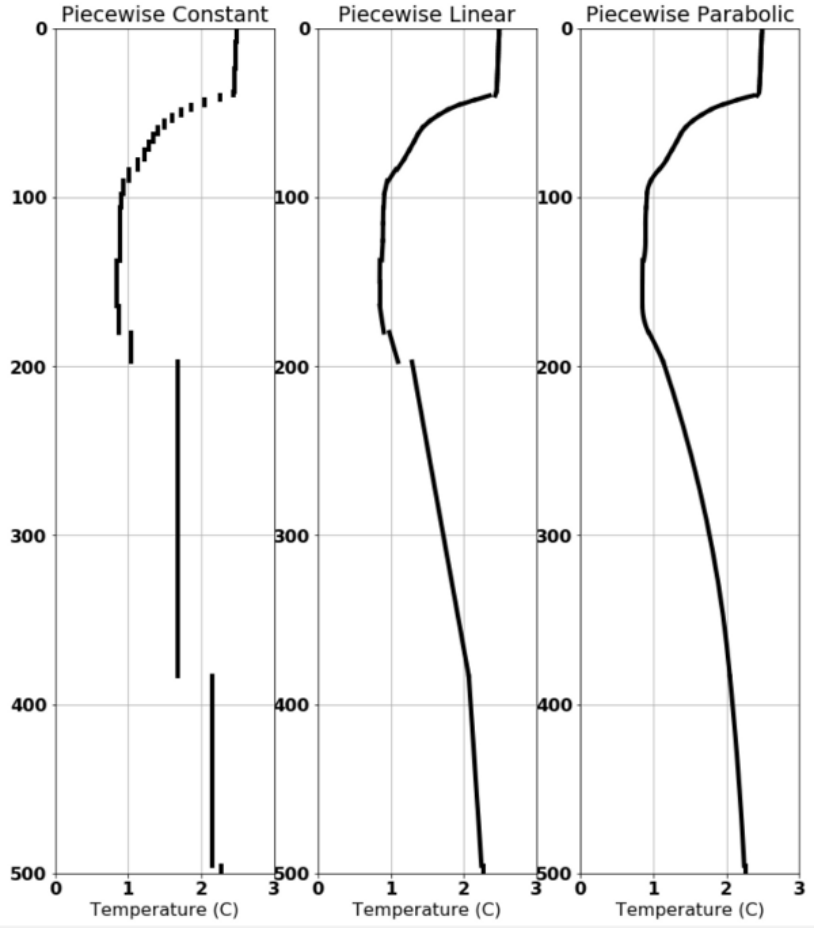


Figure 33.2 Polynomial reconstructions, starting with piecewise constant on the left, piecewise linear in the middle and piecewise parabolic on the right.

Some desired quantities for the polynomial reconstructions to be used are:

- Tracer concentrations represent the cell-averages in vertical discretization.
- Must be monotonic and introduce no new extrema.
- Discontinuous reconstructions are desirable to limit intracell slopes.

The algorithm has three phases: initialization, sorting, and flux calculation.

33.1.1 Initialization

We begin by generating polynomial reconstructions of the vertical tracer quantities such as shown by the blue lines here:

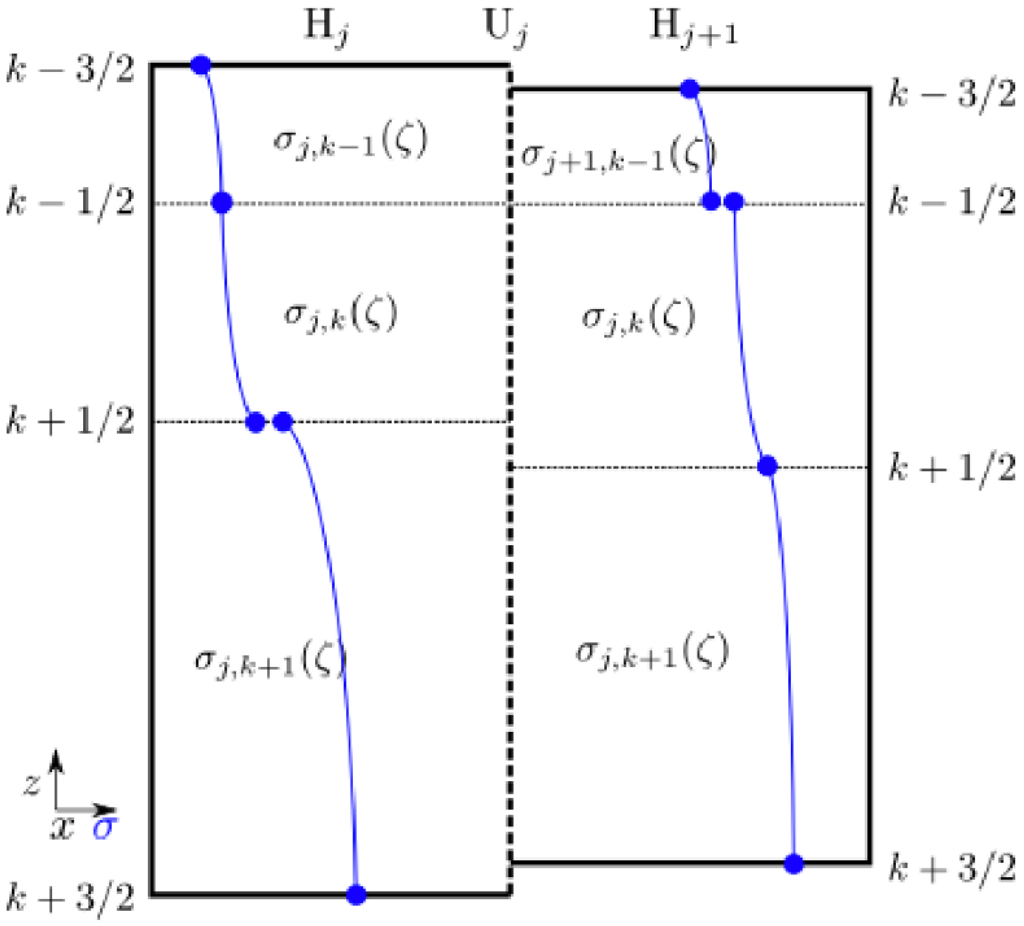


Figure 33.3 Polynomial reconstructions of two adjacent water columns.

Because we are looking to mix along epineutral surfaces, we will need to find surfaces of uniform density by using the temperature, salinity, and their effect on the density, α and β. The next step is to find the values of α and β at the interfaces.

Also during the initialization, the unstable parts of the water column are set aside to be skipped by this algorithm.

33.1.2 Sorting

The epineutral surfaces have constant density, where we use this equation:

$$\Delta\rho = \rho_1 - \rho_2 = \frac{\alpha_1 + \alpha_2}{2}(T_1 - T_2) + \frac{\beta_1 + \beta_2}{2}(S_1 - S_2)$$

When calculating α and β, there's more than one way to do it. Using a midpoint pressure gives neutral density while using a reference pressure gives isopycnal values.

Given two adjacent water columns, we are going to be looking to match densities. The match does not need to be at the same level or even near each other in depth. Starting from the top two interfaces, search the column with the lighter surface water (second column) downward to find which layer contains water matching that of the first column at the surface:

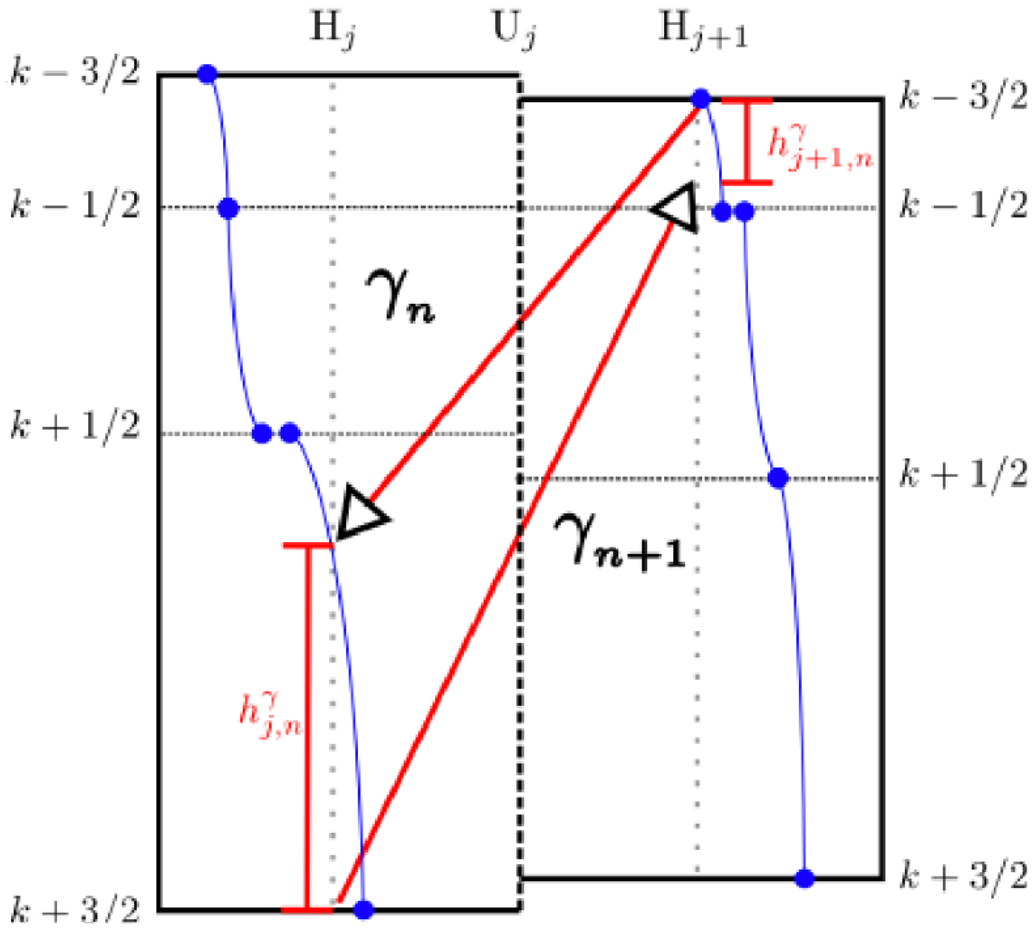


Figure 33.4 Searching the column with the lighter surface for the water matching the other column's surface water.

If the surface density matches that of an interface, point to the interface. Otherwise, solve for the matching density along the polynomial reconstruction for that layer. There are again some choices:

- Use Newton's method to find the root with higher order polynomials.
- Assume α and β vary linearly from top to bottom (cubic if T and S are parabolic).
- Equation of state is linear from top to bottom interface (parabolic of T and S are parabolic).
- $\Delta\rho$ is linear in the vertical.

Once the location of the first column's surface density is found in the second column, one goes to the next interface below to find the bottom density of the water to be mixed. Then find that density within the first column. Iterate downward until no more matches are found. These pairs of surfaces make up what is known as a sublayer along which the diffusion can take place.

33.1.3 Flux Calculation

For each sublayer, the fluxes are based on the mean tracer quantities within that sublayer in each column. For a tracer C, compute the vertical average of that tracer within the sublayer to form $\overline{C}$. The flux can then be computed based on:

$$F = K h_{\text{eff}} \frac{\overline{C_{j,k+1}} - \overline{C_{j+1,k-1}}}{\Delta x} \Delta t$$

where the effective thickness of the sublayer is:

$$h_{\text{eff}} = \frac{2 h^{\gamma}_{j,n} h^{\gamma}_{j,n+1}}{h^{\gamma}_{j,n} + h^{\gamma}_{j,n+1}}$$

and as shown in this figure:

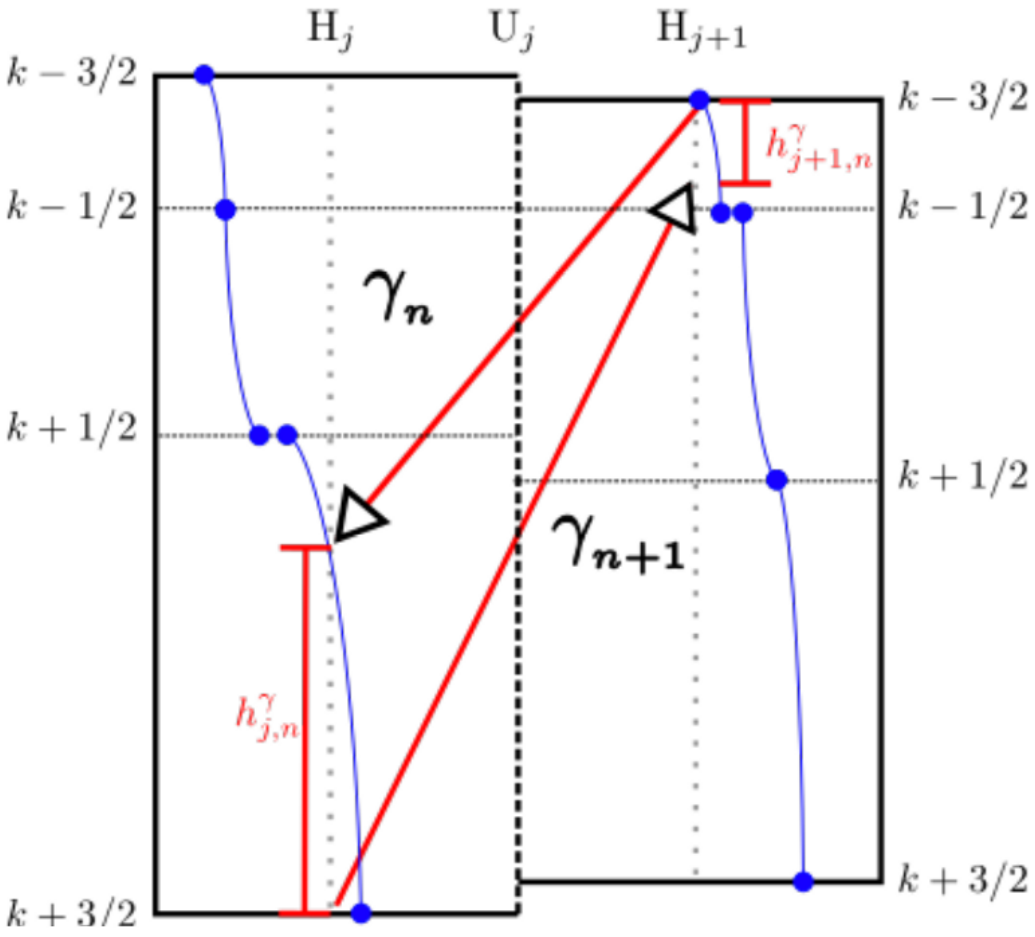


Figure 33.5 Diagram of sublayer thickness for the sublayer bounded by surfaces $$ and ${n+1}$.

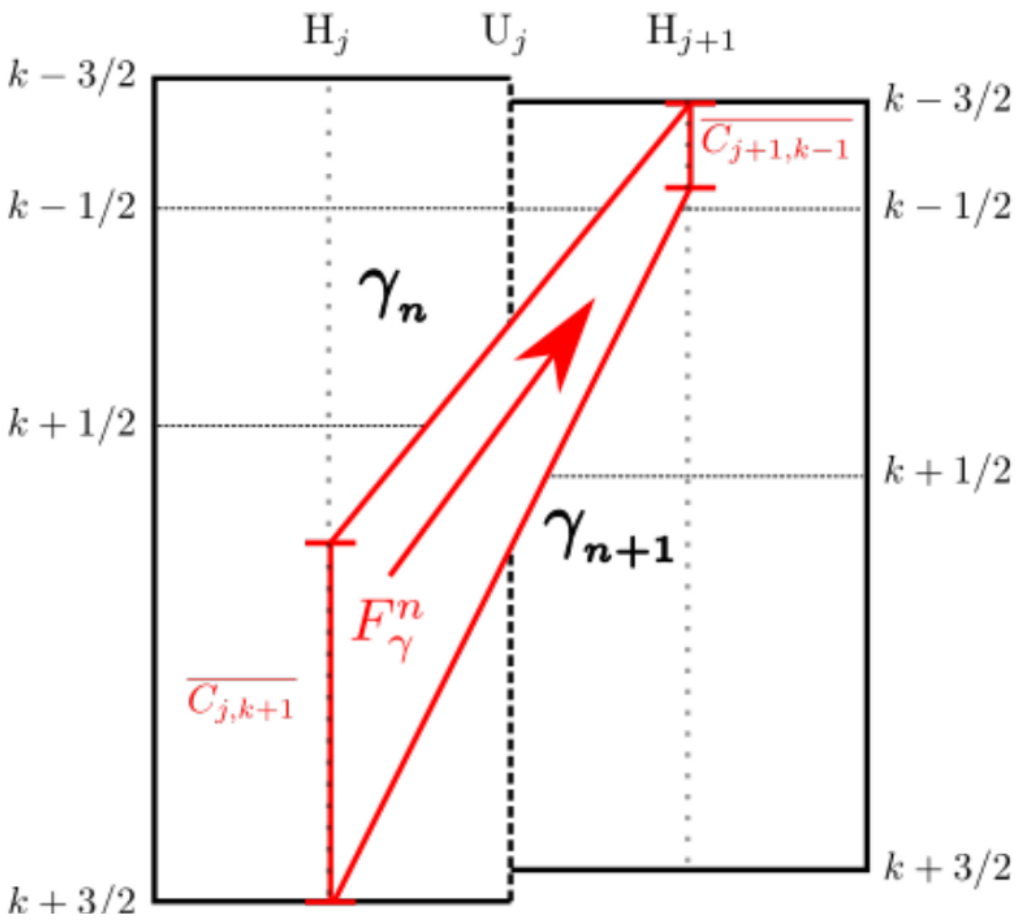


Figure 33.6 Flux of tracer C along the sublayer.

When updating the tracer state, one needs to accumulate all the fluxes through each face as shown here:

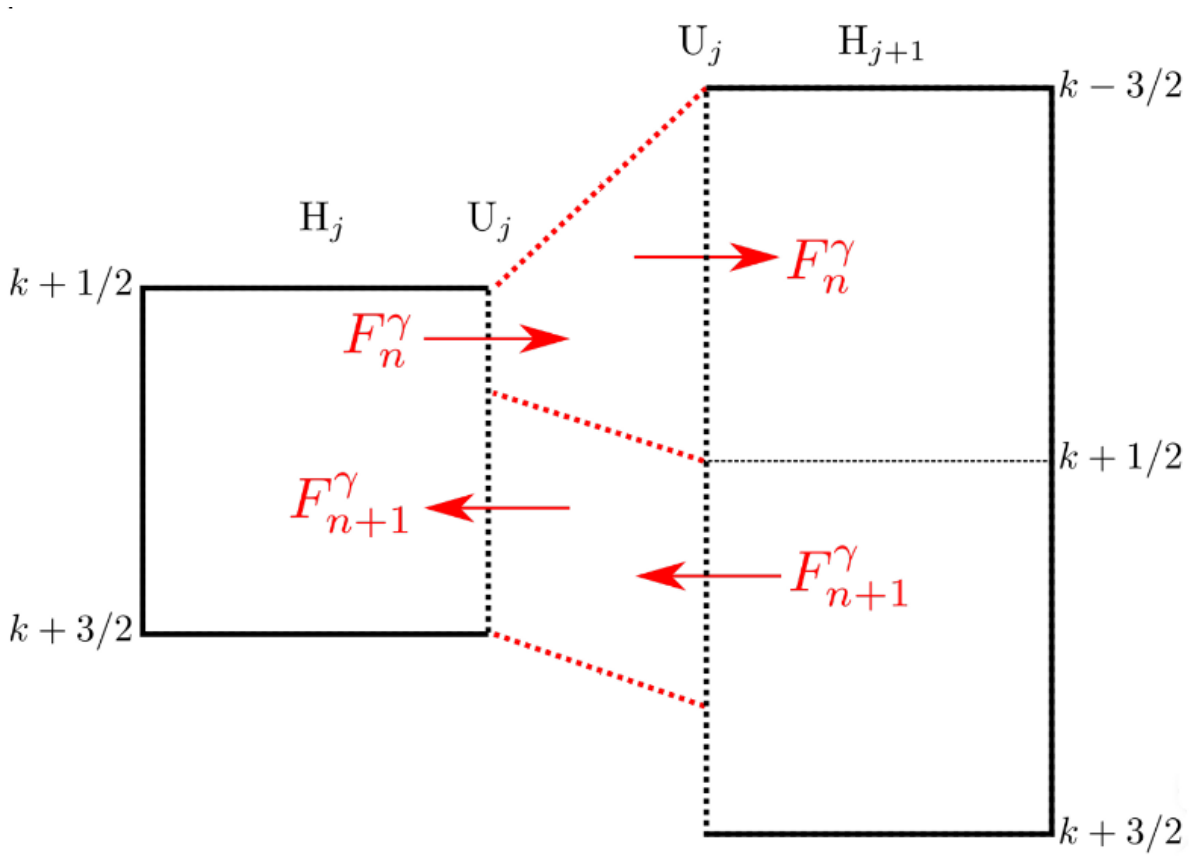


Figure 33.7 Accumulate all the fluxes across a face from all the layers in the next column contributing to it.

33.2 Surface Diffusion

As shown in figure eddy_flux of the eddy fluxes, the diffusion in the surface boundary layer is assumed to be purely horizontal. A bulk scheme was explored but found wanting, so a layer-by-layer approach has been implemented instead. It is this layer-by-layer code which is described here.

For each water column, the boundary layer thickness is determined first. This can be either via the CVMIX boundary layer thickness or through some other means. Next, determine how many of the model layers are within this boundary layer thickness. It is common for neighboring cells to have differing numbers of layers within the surface boundary layer, such as shown here:

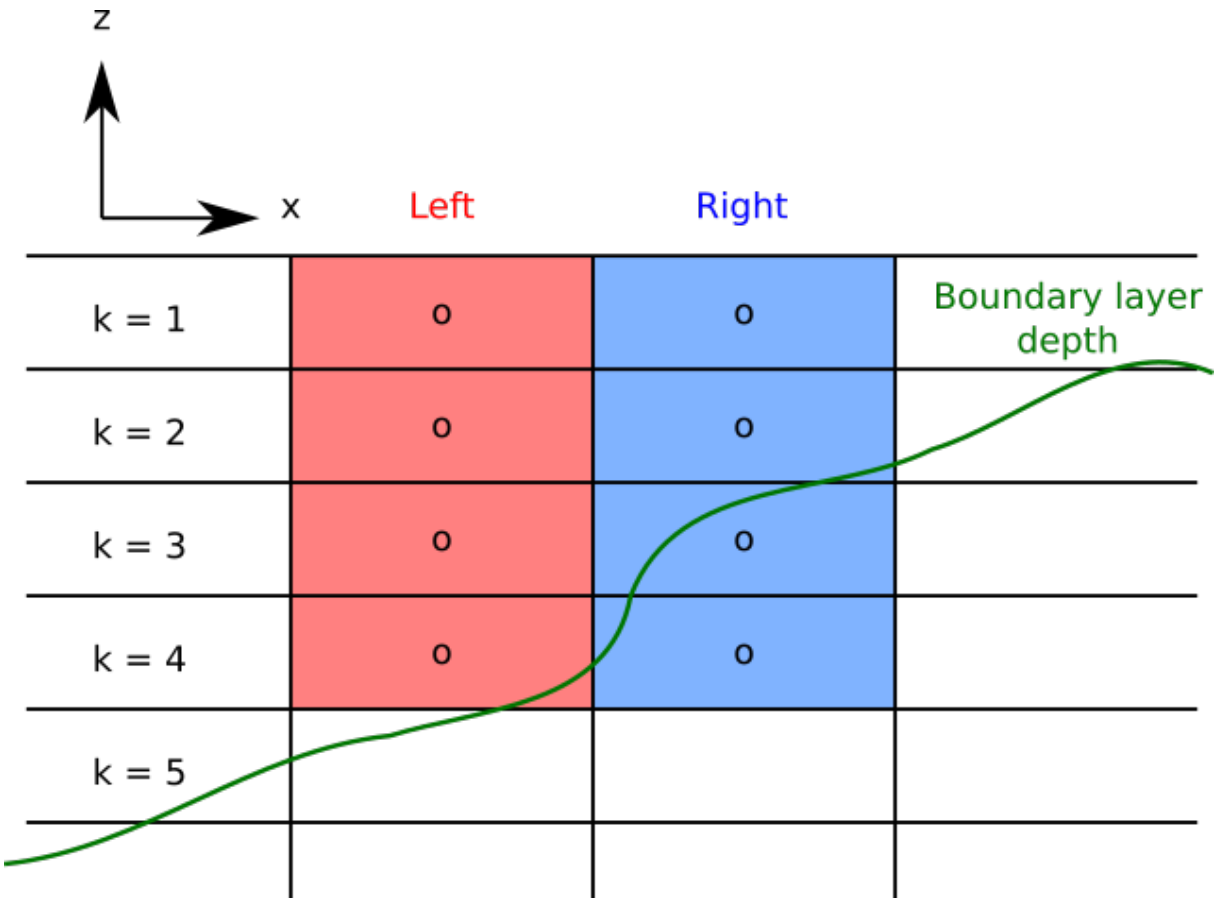


Figure 33.8 Two cells within the surface mixed layer, red on the left, blue on the right. The mixed layer depth is shown in green.

In this case, the cell on the left has four layers within the boundary layer while the cell on the right has just two. The layer-by-layer scheme computes fluxes for the first two layers, then has linearly reduced fluxes for the next two layers below as shown here:

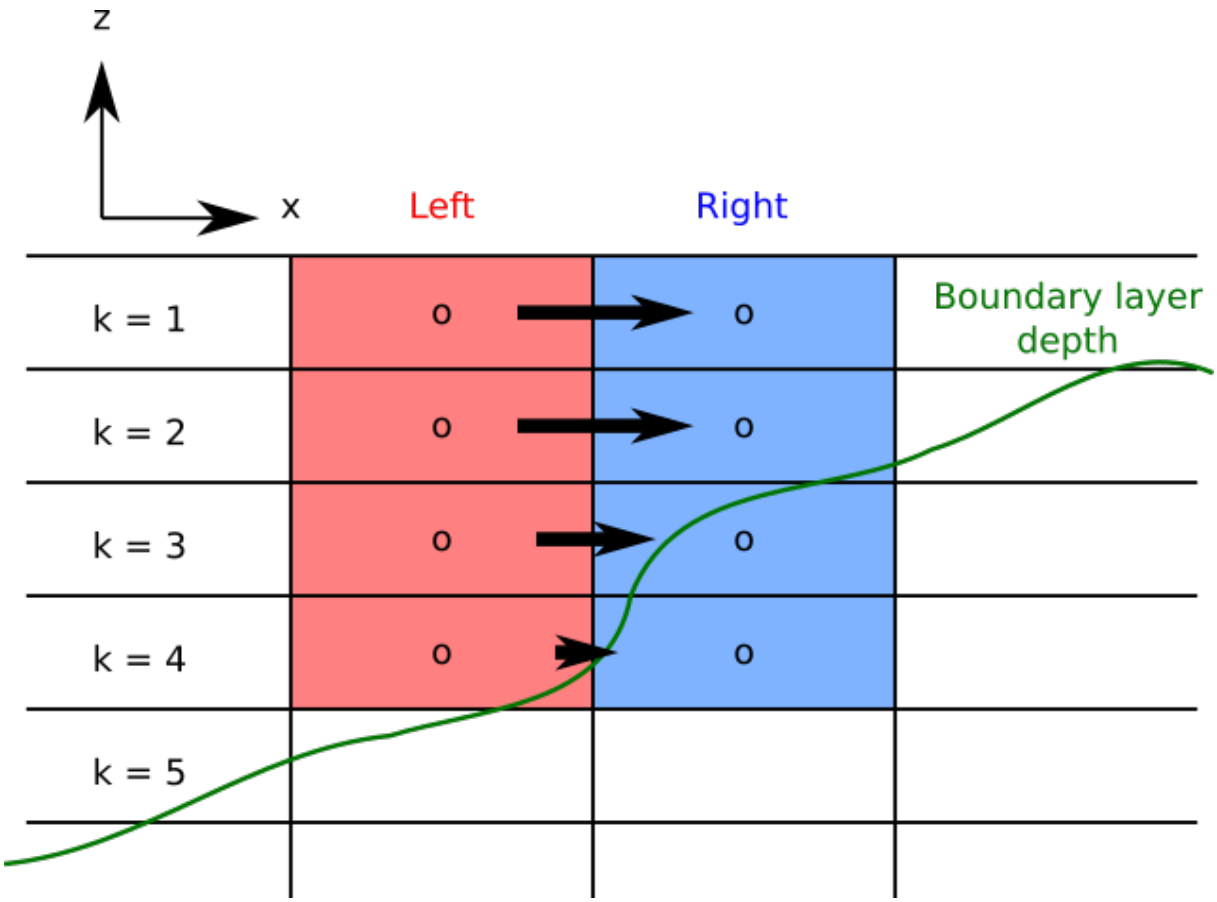


Figure 33.9 Two cells within the surface mixed layer with down-gradient fluxes as shown by the black arrows.

In all cases, the tracer flux is always down-gradient.

$$F(k) = K h_{\text{eff}(k)} \left[\phi_L(k) - \phi_R(k)\right]$$

where the effective thickness of the layer k is:

$$h_{\text{eff}(k)} = \frac{2 h_L(k) h_R(k)}{h_L(k) + h_R(k)}$$

Chapter 34

Passive and Other User-defined Tracers

Tracers beyond temperature and salinity

34.1 Passive Tracers

34.2 Generic Tracers

34.3 User-defined Tracers

Chapter 35

Tracer Fluxes

Tracer Fluxes

35.1 Tracer Fluxes

35.2 River Runoff

35.3 Ice Runoff

Chapter 36

Tracer Timestep

Overview of Tracer Timestep

The MOM6 code handles advection and lateral diffusion of all tracers. For potential temperature and salinity, it also timesteps the thermodynamics and vertical mixing (column physics). Since evaporation and precipitation are handled as volume changes, the layer thicknesses need to be updated:

$$\frac{\partial h_k}{\partial t} = (P - E)_k$$

The full tracer equation for tracer θ is:

$$\frac{\partial}{\partial t}(h_k \theta_k) + \nabla_s \cdot (\vec{u} h_k \theta_k) = Q_k^\theta h_k + \frac{1}{h_k} \Delta \left(\kappa \frac{\partial \theta}{\partial z} \right) + \frac{1}{h_k} \nabla_s (h_k K \nabla_s \theta)$$

Here, the advection is on the left hand side of the equation while the right hand side contains thermodynamic processes, vertical diffusion, and horizontal diffusion. There is more than one choice for vertical diffusion; these will be described elsewhere. Also, the lateral diffusion is handled in novel ways so as to avoid introduction of new extrema and to avoid instabilities associated with rotated mixing tensors. The lateral diffusion is described in Horizontal Diffusion.

Chapter 37

Tracer Transport Equations

Tracer Transport Equations

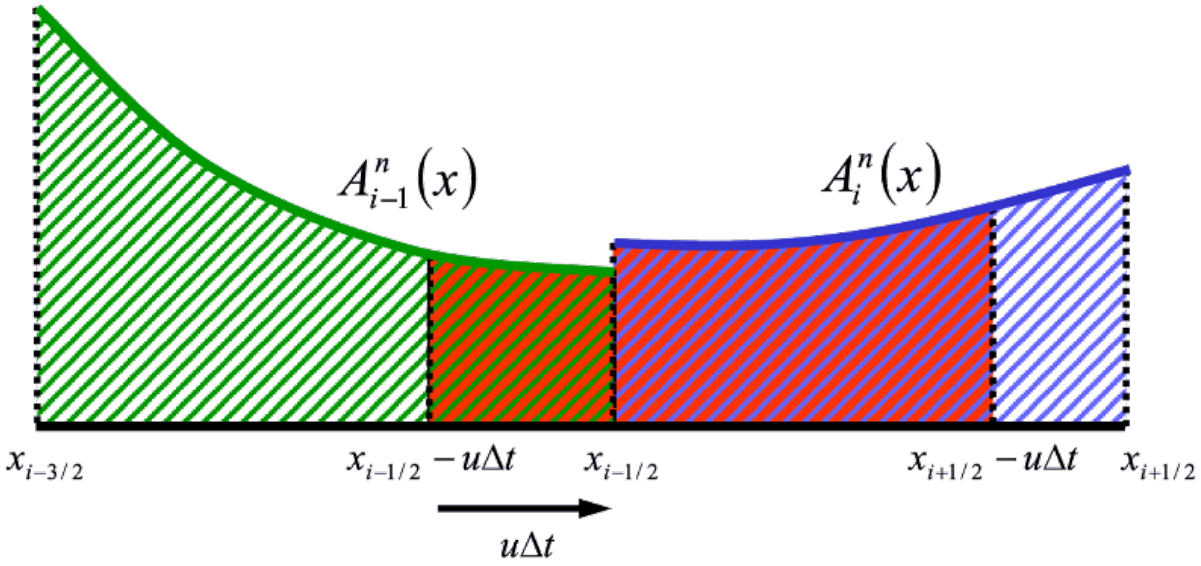


Figure 37.1 The 1-D finite volume advection of tracers. The reddish fluid will be in the cell at the end of the timestep.

Given a piecewise polynomial description of the tracer concentration, the new tracer cell concentration is the average of the fluid that will be in the cell after a timestep.

$$\int_{x_{i-1/2}}^{x_{i+1/2}} A_i^{n+1}(x)dx = \int_{x_{i-1/2-u\Delta t}}^{x_{i+1/2-u\Delta t}} A_i^n(x)dx \qquad = \tag{37.1}$$

$$\int_{x_{i-1/2}}^{x_{i+1/2}} A_i^n(x)dx - \int_{x_{i+1/2-u\Delta t}}^{x_{i+1/2}} A_i^n(x)dx \quad + \int_{x_{i-1/2-u\Delta t}}^{x_{i-1/2}} A_i^n(x)dx \tag{37.2}$$

Fluxes are found by analytically integrating the profile over the distance that is swept past the face within a timestep.

$$a_i^n = \frac{1}{\Delta x} \int_{x_{i-1/2}}^{x_{i+1/2}} A_i^n(x)dx$$

$$a_i^{n+1} = a_i^n - \frac{\Delta t}{\Delta x}(F_{i+1/2} - F_{i-1/2})$$
$$F_{i+1/2} = \frac{1}{\Delta t}\int_{x_{i+1/2-u\Delta t}}^{x_{i+1/2}} A_i^n(x)dx$$
$$F_{i-1/2} = \frac{1}{\Delta t}\int_{x_{i-1/2-u\Delta t}}^{x_{i-1/2}} A_i^n(x)dx$$

With piecewise constant profiles, this approach give first order upwind advection. Higher order polynomials (e.g., parabolas) can give higher order accuracy.

37.1 Multidimensional Tracer Advection

Using "Easter's Pseudo-compressibility" (easter1993), we start with these basic equations for a tracer ψ:

$$\frac{\partial h}{\partial t} + \vec{\nabla}\cdot(\vec{u}h) = 0 \equiv \frac{\partial h}{\partial t} + \vec{\nabla}\cdot(\vec{U})$$

$$\frac{\partial}{\partial t}(h\psi) + \vec{\nabla}\cdot(\vec{U}\psi) = 0$$

$$\frac{\partial \psi}{\partial t} + \vec{u}\cdot\vec{\nabla}\psi = 0$$

We discretize the first of these equations in space:

$$\frac{\partial h}{\partial t} = \frac{1}{\Delta x}\left(U_{i-\frac{1}{2},j} - U_{i+\frac{1}{2},j}\right) + \frac{1}{\Delta y}\left(V_{i,j-\frac{1}{2}} - V_{i,j+\frac{1}{2}}\right)$$

Using our monotonic one-dimensional flux:

$$F_{i+\frac{1}{2},j}(\psi) = U_{i+\frac{1}{2},j}\psi_{i+\frac{1}{2},j}$$

we come up with an estimate based only on an update in the x direction:

$$\tilde{h}_{i,j}\tilde{\psi}_{i,j} = h_{i,j}^n\psi_{i,j} + \frac{\Delta t}{\Delta x}\left(F_{i-\frac{1}{2},j}(\psi^n) - F_{i+\frac{1}{2},j}(\psi^n)\right)$$

$$\tilde{h}_{i,j} = h_{i,j}^n + \frac{\Delta t}{\Delta x}\left(U_{i-\frac{1}{2},j} - U_{i+\frac{1}{2},j}\right)$$

$$\tilde{\psi}_{i,j} = \frac{\tilde{h}_{i,j}\tilde{\psi}_{i,j}}{\tilde{h}_{i,j}}$$

Next, we update in the y direction:

$$h_{i,j}^{n+1}\psi_{i,j}^{n+1} = \tilde{h}_{i,j}\tilde{\psi}_{i,j} + \frac{\Delta t}{\Delta y}\left(G_{i,j-\frac{1}{2}}(\tilde{\psi}) - G_{i,j+\frac{1}{2}}(\tilde{\psi})\right)$$

$$h_{i,j}^{n+1} = \tilde{h}_{i,j} + \frac{\Delta t}{\Delta y}\left(V_{i,j-\frac{1}{2}} - V_{i,j+\frac{1}{2}}\right)$$

$$\psi_{i,j}^{n+1} = \frac{h_{i,j}^{n+1}\psi_{i,j}^{n+1}}{h_{i,j}^{n+1}}$$

- This method ensures monotonicity. Strang splitting can reduce directional splitting error. See **?**, **?** (section 5.9.4), and **?** .

- Flux-form pseudo-compressibility advection is based on accumulated mass (or volume) fluxes, not velocities.

- Additional pseudo-compressibility passes can be added to accommodate transports exceeding cell masses. Extra passes of tracer advection are used in MOM6 in the small fraction of cells where this is needed.

- Explicit layered dynamics time-steps are limited by Doppler-shifted internal gravity wave speeds or inertial oscillations. Flow speeds in most of the ocean volume are much smaller than the peak internal wave speeds so that the advective time-steps can be longer.

- Advective mass fluxes in MOM6 are often accumulated over multiple dynamic steps. The goal is that as we go to higher resolution, this tracer advection will remain stable at relatively long time-steps, allowing for the inclusion of many biogeochemical tracers without adding an undue burden in computational cost.

Chapter 38

Vertical Diffusion

Vertical diffusion of tracers

The MOM6 tracer registry takes care of the tracer advection as well as horizontal diffusion, but it is up to each individual tracer package to define its own vertical diffusion.

Chapter 39

Modules Index

39.1 Modules List

Here is a list of all modules with brief descriptions:

mom
: The central module of the MOM6 ocean model . . . . . ??

mom_ale
: This module contains the main regridding routines . . . . . ??

mom_eos
: Provides subroutines for quantities specific to the equation of state . . . . . ??

mom_ice_shelf
: Implements the thermodynamic aspects of ocean / ice-shelf interactions, along with a crude placeholder for a later implementation of full ice shelf dynamics, all using the MOM framework and coding style . . . . . ??

mom_meke
: Implements the Mesoscale Eddy Kinetic Energy framework with topographic beta effect included in computing beta in Rhines scale . . . . . ??

mom_unit_scaling
: Provides a transparent unit rescaling type to facilitate dimensional consistency testing . . . . . ??

Chapter 40

Data Type Index

40.1 Data Types List

Here are the data types with brief descriptions:

mom_ale::ale_cs
ALE control structure . . . . . ??
mom_eos::calculate_compress
Calculates the compressibility of water from T, S, and P . . . . . ??
mom_eos::calculate_density
Calculates density of sea water from T, S and P . . . . . ??
mom_eos::calculate_density_derivs
Calculate the derivatives of density with temperature and salinity from T, S, and P . . . . . ??
mom_eos::calculate_density_second_derivs
Calculates the second derivatives of density with various combinations of temperature, salinity, and pressure from T, S and P . . . . . ??
mom_eos::calculate_spec_vol
Calculates specific volume of sea water from T, S and P . . . . . ??
mom_eos::calculate_specific_vol_derivs
Calculate the derivatives of specific volume with temperature and salinity from T, S, and P . . . ??
mom_eos::calculate_tfreeze
Calculates the freezing point of sea water from T, S and P . . . . . ??
mom_eos::eos_type
A control structure for the equation of state . . . . . ??
mom_ice_shelf::ice_shelf_cs
Control structure that contains ice shelf parameters and diagnostics handles . . . . . ??
mom_meke::meke_cs
Control structure that contains MEKE parameters and diagnostics handles . . . . . ??
mom::mom_control_struct
Control structure for the MOM module, including the variables that describe the state of the ocean ??
mom::mom_diag_ids
A structure with diagnostic IDs of the state variables . . . . . ??
mom_unit_scaling::unit_scale_type
Describes various unit conversion factors . . . . . ??

Chapter 41

File Index

41.1 File List

Here is a list of all files with brief descriptions:

/home/cermak/src/MOM6/src/ALE/MOM_ALE.F90 . . . . . ??
/home/cermak/src/MOM6/src/core/MOM.F90 . . . . . ??
/home/cermak/src/MOM6/src/equation_of_state/MOM_EOS.F90 . . . . . ??
/home/cermak/src/MOM6/src/framework/MOM_unit_scaling.F90 . . . . . ??
/home/cermak/src/MOM6/src/ice_shelf/MOM_ice_shelf.F90 . . . . . ??
/home/cermak/src/MOM6/src/parameterizations/lateral/MOM_MEKE.F90 . . . . . ??

Chapter 42

Module Documentation

42.1 mom Module Reference

The central module of the MOM6 ocean model.

Data Types

- type mom_control_struct

  Control structure for the MOM module, including the variables that describe the state of the ocean.
- type mom_diag_ids

  A structure with diagnostic IDs of the state variables.

Functions/Subroutines

- subroutine, public step_mom (forces_in, fluxes_in, sfc_state, Time_start, time_int_in, CS, Waves, do_↩ dynamics, do_thermodynamics, start_cycle, end_cycle, cycle_length, reset_therm)

  This subroutine orchestrates the time stepping of MOM. The adiabatic dynamics are stepped by calls to one of the step_MOM_dyn_...routines. The action of lateral processes on tracers occur in calls to advect_tracer and tracer_↩ hordiff. Vertical mixing and possibly remapping occur inside of diabatic.
- subroutine step_mom_dynamics (forces, p_surf_begin, p_surf_end, dt, dt_thermo, bbl_time_int, CS, Time↩ _local, Waves)

  Time step the ocean dynamics, including the momentum and continuity equations.
- subroutine step_mom_tracer_dyn (CS, G, GV, US, h, Time_local)

  step_MOM_tracer_dyn does tracer advection and lateral diffusion, bringing the tracers up to date with the changes in state due to the dynamics. Surface sources and sinks and remapping are handled via step_MOM_thermo.
- subroutine step_mom_thermo (CS, G, GV, US, u, v, h, tv, fluxes, dtdia, Time_end_thermo, update_BBL, Waves)

  MOM_step_thermo orchestrates the thermodynamic time stepping and vertical remapping, via calls to diabatic (or adiabatic) and ALE_main.
- subroutine, public step_offline (forces, fluxes, sfc_state, Time_start, time_interval, CS)

  step_offline is the main driver for running tracers offline in MOM6. This has been primarily developed with ALE configurations in mind. Some work has been done in isopycnal configuration, but the work is very preliminary. Some more detail about this capability along with some of the subroutines called here can be found in tracers/MOM_↩ offline_control.F90
- subroutine, public initialize_mom (Time, Time_init, param_file, dirs, CS, restart_CSp, Time_in, offline_↩ tracer_mode, input_restart_file, diag_ptr, count_calls, tracer_flow_CSp)

Initialize MOM, including memory allocation, setting up parameters and diagnostics, initializing the ocean state variables, and initializing subsidiary modules.

- subroutine, public finish_mom_initialization (Time, dirs, CS, restart_CSp)

  Finishes initializing MOM and writes out the initial conditions.
- subroutine register_diags (Time, G, GV, US, IDs, diag)

  Register certain diagnostics.
- subroutine mom_timing_init (CS)

  Set up CPU clock IDs for timing various subroutines.
- subroutine set_restart_fields (GV, US, param_file, CS, restart_CSp)

  Set the fields that are needed for bitwise identical restarting the time stepping scheme. In addition to those specified here directly, there may be fields related to the forcing or to the barotropic solver that are needed; these are specified in sub- routines that are called from this one.
- subroutine adjust_ssh_for_p_atm (tv, G, GV, US, ssh, p_atm, use_EOS)

  Apply a correction to the sea surface height to compensate for the atmospheric pressure (the inverse barometer).
- subroutine, public extract_surface_state (CS, sfc_state_in)

  Set the surface (return) properties of the ocean model by setting the appropriate fields in sfc_state. Unused fields are set to NULL or are unallocated.
- subroutine rotate_initial_state (u_in, v_in, h_in, T_in, S_in, use_temperature, turns, u, v, h, T, S)

  Rotate initialization fields from input to rotated arrays.
- logical function, public mom_state_is_synchronized (CS, adv_dyn)

  Return true if all phases of step_MOM are at the same point in time.
- subroutine, public get_mom_state_elements (CS, G, GV, US, C_p, C_p_scaled, use_temp)

  This subroutine offers access to values or pointers to other types from within the MOM_control_struct, allowing the MOM_control_struct to be opaque.
- subroutine, public get_ocean_stocks (CS, mass, heat, salt, on_PE_only)

  Find the global integrals of various quantities.
- subroutine, public mom_end (CS)

  End of ocean model, including memory deallocation.

Variables

- integer id_clock_ocean

  CPU time clock IDs.
- integer id_clock_dynamics

  CPU time clock IDs.
- integer id_clock_thermo

  CPU time clock IDs.
- integer id_clock_tracer

  CPU time clock IDs.
- integer id_clock_diabatic

  CPU time clock IDs.
- integer id_clock_adiabatic

  CPU time clock IDs.
- integer id_clock_continuity

  CPU time clock IDs.
- integer id_clock_thick_diff

  CPU time clock IDs.
- integer id_clock_bbl_visc

CPU time clock IDs.

- integer id_clock_ml_restrat

  CPU time clock IDs.
- integer id_clock_diagnostics

  CPU time clock IDs.
- integer id_clock_z_diag

  CPU time clock IDs.
- integer id_clock_init

  CPU time clock IDs.
- integer id_clock_mom_init

  CPU time clock IDs.
- integer id_clock_pass

  CPU time clock IDs.
- integer id_clock_pass_init

  CPU time clock IDs.
- integer id_clock_ale

  CPU time clock IDs.
- integer id_clock_other

  CPU time clock IDs.
- integer id_clock_offline_tracer

  CPU time clock IDs.
- integer id_clock_unit_tests

  CPU time clock IDs.

42.1.1 Detailed Description

The central module of the MOM6 ocean model.

Modular Ocean Model (MOM) Version 6.0 (MOM6)

Authors

Alistair Adcroft, Robert Hallberg, and Stephen Griffies

Additional contributions from:

- Whit Anderson
- Brian Arbic
- Will Cooke
- Anand Gnanadesikan
- Matthew Harrison
- Mehmet Ilicak
- Laura Jackson
- Jasmine John
- John Krasting
- Zhi Liang

- Bonnie Samuels
- Harper Simmons
- Laurent White
- Niki Zadeh

MOM ice-shelf code was developed by

- Daniel Goldberg
- Robert Hallberg
- Chris Little
- Olga Sergienko

42.1.2 Overview of MOM

This program (MOM) simulates the ocean by numerically solving the hydrostatic primitive equations in generalized Lagrangian vertical coordinates, typically tracking stretched pressure (p$*$) surfaces or following isopycnals in the ocean's interior, and general orthogonal horizontal coordinates. Unlike earlier versions of MOM, in MOM6 these equations are horizontally discretized on an Arakawa C-grid. (It remains to be seen whether a B-grid dynamic core will be revived in MOM6 at a later date; for now applications requiring a B-grid discretization should use MOM5.1.) MOM6 offers a range of options for the physical parameterizations, from those most appropriate to highly idealized models for geophysical fluid dynamics studies to a rich suite of processes appropriate for realistic ocean simulations. The thermodynamic options typically use conservative temperature and preformed salinity as conservative state variables and a full nonlinear equation of state, but there are also idealized adiabatic configurations of the model that use fixed density layers. Version 6.0 of MOM continues in the long tradition of a commitment to climate-quality ocean simulations embodied in previous versions of MOM, even as it draws extensively on the lessons learned in the development of the Generalized Ocean Layered Dynamics (GOLD) ocean model, which was also primarily developed at NOAA/GFDL. MOM has also benefited tremendously from the FMS infrastructure, which it utilizes and shares with other component models developed at NOAA/GFDL.

When run is isopycnal-coordinate mode, the uppermost few layers are often used to describe a bulk mixed layer, including the effects of penetrating shortwave radiation. Either a split- explicit time stepping scheme or a non-split scheme may be used for the dynamics, while the time stepping may be split (and use different numbers of steps to cover the same interval) for the forcing, the thermodynamics, and for the dynamics. Most of the numerics are second order accurate in space. MOM can run with an absurdly thin minimum layer thickness. A variety of non-isopycnal vertical coordinate options are under development, but all exploit the advantages of a Lagrangian vertical coordinate, as discussed in detail by Adcroft and Hallberg (Ocean Modelling, 2006).

Details of the numerics and physical parameterizations are provided in the appropriate source files. All of the available options are selected at run-time by parsing the input files, usually MOM_input and MOM_override, and the options choices are then documented for each run in MOM_param_docs.

MOM6 integrates the equations forward in time in three distinct phases. In one phase, the dynamic equations for the velocities and layer thicknesses are advanced, capturing the propagation of external and internal inertia-gravity waves, Rossby waves, and other strictly adiabatic processes, including lateral stresses, vertical viscosity and momentum forcing, and interface height diffusion (commonly called Gent-McWilliams diffusion in depth- coordinate models). In the second phase, all tracers are advected and diffused along the layers. The third phase applies diabatic processes, vertical mixing of water properties, and perhaps vertical remapping to cause the layers to track the desired vertical coordinate.

The present file (MOM.F90) orchestrates the main time stepping loops. One time integration option for the dynamics uses a split explicit time stepping scheme to rapidly step the barotropic pressure and velocity fields. The barotropic velocities are averaged over the baroclinic time step before they are used to advect thickness and determine the

baroclinic accelerations. As described in Hallberg and Adcroft (2009), a barotropic correction is applied to the time-mean layer velocities to ensure that the sum of the layer transports agrees with the time-mean barotropic transport, thereby ensuring that the estimates of the free surface from the sum of the layer thicknesses agrees with the final free surface height as calculated by the barotropic solver. The barotropic and baroclinic velocities are kept consistent by recalculating the barotropic velocities from the baroclinic transports each time step. This scheme is described in Hallberg, 1997, J. Comp. Phys. 135, 54-65 and in Hallberg and Adcroft, 2009, Ocean Modelling, 29, 15-26.

The other time integration options use non-split time stepping schemes based on the 3-step third order Runge-Kutta scheme described in Matsuno, 1966, J. Met. Soc. Japan, 44, 85-88, or on a two-step quasi-2nd order Runge-Kutta scheme. These are much slower than the split time-stepping scheme, but they are useful for providing a more robust solution for debugging cases where the more complicated split time-stepping scheme may be giving suspect solutions.

There are a range of closure options available. Horizontal velocities are subject to a combination of horizontal bi-harmonic and Laplacian friction (based on a stress tensor formalism) and a vertical Fickian viscosity (perhaps using the kinematic viscosity of water). The horizontal viscosities may be constant, spatially varying or may be dynamically calculated using Smagorinsky's approach. A diapycnal diffusion of density and thermodynamic quantities is also allowed, but not required, as is horizontal diffusion of interface heights (akin to the Gent-McWilliams closure of geopotential coordinate models). The diapycnal mixing may use a fixed diffusivity or it may use the shear Richardson number dependent closure, like that described in Jackson et al. (JPO, 2008). When there is diapycnal diffusion, it applies to momentum as well. As this is in addition to the vertical viscosity, the vertical Prandtl always exceeds 1. A refined bulk-mixed layer is often used to describe the planetary boundary layer in realistic ocean simulations.

MOM has a number of noteworthy debugging capabilities. Excessively large velocities are truncated and MOM will stop itself after a number of such instances to keep the model from crashing altogether. This is useful in diagnosing failures, or (by accepting some truncations) it may be useful for getting the model past the adjustment from an ill-balanced initial condition. In addition, all of the accelerations in the columns with excessively large velocities may be directed to a text file. Parallelization errors may be diagnosed using the DEBUG option, which causes extensive checksums to be written out along with comments indicating where in the algorithm the sums originate and what variable is being summed. The point where these checksums differ between runs is usually a good indication of where in the code the problem lies. All of the test cases provided with MOM are routinely tested to ensure that they give bitwise identical results regardless of the domain decomposition, or whether they use static or dynamic memory allocation.

42.1.3 Structure of MOM

About 115 other files of source code and 4 header files comprise the MOM code, although there are several hundred more files that make up the FMS infrastructure upon which MOM is built. Each of the MOM files contains comments documenting what it does, and most of the file names are fairly self-evident. In addition, all subroutines and data types are referenced via a module use, only statement, and the module names are consistent with the file names, so it is not too hard to find the source file for a subroutine.

The typical MOM directory tree is as follows:

```
../MOM
|-- config_src
|   |-- coupled_driver
|   |-- dynamic
|   `-- solo_driver
|-- examples
|   |-- CM2G
|   |-- ...
|   `-- torus_advection_test
`-- src
    |-- core
    |-- diagnostics
    |-- equation_of_state
    |-- framework
    |-- ice_shelf
```

```
|-- initialization
|-- parameterizations
|   |-- lateral
|   `-- vertical
|-- tracer
`-- user
```

Rather than describing each file here, each directory contents will be described to give a broad overview of the MOM code structure.

The directories under config_src contain files that are used for configuring the code, for instance for coupled or ocean-only runs. Only one or two of these directories are used in compiling any, particular run.

- config_src/coupled_driver: The files here are used to couple MOM as a component in a larger run driven by the FMS coupler. This includes code that converts various forcing fields into the code structures and flux and unit conventions used by MOM, and converts the MOM surface fields back to the forms used by other FMS components.
- config_src/dynamic: The only file here is the version of MOM_memory.h that is used for dynamic memory configurations of MOM.
- config_src/solo_driver: The files here are include the _main driver that is used when MOM is configured as an ocean-only model, as well as the files that specify the surface forcing in this configuration.

  The directories under examples provide a large number of working configurations of MOM, along with reference solutions for several different compilers on GFDL's latest large computer. The versions of MOM_↩ memory.h in these directories need not be used if dynamic memory allocation is desired, and the answers should be unchanged.

  The directories under src contain most of the MOM files. These files are used in every configuration using MOM.
- src/core: The files here constitute the MOM dynamic core. This directory also includes files with the types that describe the model's lateral grid and have defined types that are shared across various MOM modules to allow for more succinct and flexible subroutine argument lists.
- src/diagnostics: The files here calculate various diagnostics that are anciliary to the model itself. While most of these diagnostics do not directly affect the model's solution, there are some, like the calculation of the deformation radius, that are used in some of the process parameterizations.
- src/equation_of_state: These files describe the physical properties of sea-water, including both the equation of state and when it freezes.
- src/framework: These files provide infrastructure utilities for MOM. Many are simply wrappers for capabilities provided by FMS, although others provide capabilities (like the file_parser) that are unique to MOM. When MOM is adapted to use a modeling infrastructure distinct from FMS, most of the required changes are in this directory.
- src/initialization: These are the files that are used to initialize the MOM grid or provide the initial physical state for MOM. These files are not intended to be modified, but provide a means for calling user-specific initialization code like the examples in src/user.
- src/parameterizations/lateral: These files implement a number of quasi-lateral (along-layer) process parameterizations, including lateral viscosities, parameterizations of eddy effects, and the calculation of tidal forcing.
- src/parameterizations/vertical: These files implement a number of vertical mixing or diabatic processes, including the effects of vertical viscosity and code to parameterize the planetary boundary layer. There is a separate driver that orchestrates this portion of the algorithm, and there is a diversity of parameterizations to be found here.
- src/tracer: These files handle the lateral transport and diffusion of tracers, or are the code to implement various passive tracer packages. Additional tracer packages are readily accommodated.

- src/user: These are either stub routines that a user could use to change the model's initial conditions or forcing, or are examples that implement specific test cases. These files can easily be hand edited to create new analytically specified configurations.

Most simulations can be set up by modifying only the files MOM_input, and possibly one or two of the files in src/user. In addition, the diag_table (MOM_diag_table) will commonly be modified to tailor the output to the needs of the question at hand. The FMS utility mkmf works with a file called path_names to build an appropriate makefile, and path_names should be edited to reflect the actual location of the desired source code.

There are 3 publicly visible subroutines in this file (MOM.F90).

- step_MOM steps MOM over a specified interval of time.
- MOM_initialize calls initialize and does other initialization that does not warrant user modification.
- extract_surface_state determines the surface (bulk mixed layer if traditional isoycnal vertical coordinate) properties of the current model state and packages pointers to these fields into an exported structure.

  The remaining subroutines in this file (src/core/MOM.F90) are:
- find_total_transport determines the barotropic mass transport.
- register_diags registers many diagnostic fields for the dynamic solver, or of the main model variables.
- MOM_timing_init initializes various CPU time clocks.
- write_static_fields writes out various time-invariant fields.
- set_restart_fields is used to specify those fields that are written to and read from the restart file.

42.1.4 Diagnosing MOM heat budget

Here are some example heat budgets for the ALE version of MOM6.

42.1.4.1 Depth integrated heat budget

Depth integrated heat budget diagnostic for MOM.

- OPOTTEMPTEND_2d = T_ADVECTION_XY_2d + OPOTTEMPPMDIFF_2d + HFDS + HFGEOU
- T_ADVECTION_XY_2d = horizontal advection
- OPOTTEMPPMDIFF_2d = neutral diffusion
- HFDS = net surface boundary heat flux
- HFGEOU = geothermal heat flux
- HFDS = net surface boundary heat flux entering the ocean = rsntds + rlntds + hfls + hfss + heat_pme + hfsifrazil
- More heat flux cross-checks
  - hfds = net_heat_coupler + hfsifrazil + heat_pme
  - heat_pme = heat_content_surfwater = heat_content_massin + heat_content_massout = heat_content_fprec + heat_content_cond + heat_content_vprec
    - hfrunoffds + hfevapds + hfrainds

42.1.4.2 Depth integrated heat budget

Here is an example 3d heat budget diagnostic for MOM.

- OPOTTEMPTEND = T_ADVECTION_XY + TH_TENDENCY_VERT_REMAP + OPOTTEMPDIFF + OPO↩ TTEMPPMDIFF
  - BOUNDARY_FORCING_HEAT_TENDENCY + FRAZIL_HEAT_TENDENCY
- OPOTTEMPTEND = net tendency of heat as diagnosed in MOM.F90
- T_ADVECTION_XY = heating of a cell from lateral advection
- TH_TENDENCY_VERT_REMAP = heating of a cell from vertical remapping
- OPOTTEMPDIFF = heating of a cell from diabatic diffusion
- OPOTTEMPPMDIFF = heating of a cell from neutral diffusion
- BOUNDARY_FORCING_HEAT_TENDENCY = heating of cell from boundary fluxes
- FRAZIL_HEAT_TENDENCY = heating of cell from frazil
- TH_TENDENCY_VERT_REMAP has zero vertical sum, as it redistributes heat in vertical.
- OPOTTEMPDIFF has zero vertical sum, as it redistributes heat in the vertical.
- BOUNDARY_FORCING_HEAT_TENDENCY generally has 3d structure, with k $>$ 1 contributions from penetrative shortwave, and from other fluxes for the case when layers are tiny, in which case MOM6 partitions tendencies into k $>$ 1 layers.
- FRAZIL_HEAT_TENDENCY generally has 3d structure, since MOM6 frazil calculation checks the full ocean column.
- FRAZIL_HEAT_TENDENCY[k=@sum] = HFSIFRAZIL = column integrated frazil heating.
- HFDS = FRAZIL_HEAT_TENDENCY[k=@sum] + BOUNDARY_FORCING_HEAT_TENDENCY[k=@sum]

  Here is an example 2d heat budget (depth summed) diagnostic for MOM.
- OPOTTEMPTEND_2d = T_ADVECTION_XY_2d + OPOTTEMPPMDIFF_2d + HFDS

  Here is an example 3d salt budget diagnostic for MOM.
- OSALTTEND = S_ADVECTION_XY + SH_TENDENCY_VERT_REMAP + OSALTDIFF + OSALTPMDIFF
  - BOUNDARY_FORCING_SALT_TENDENCY
- OSALTTEND = net tendency of salt as diagnosed in MOM.F90
- S_ADVECTION_XY = salt convergence to cell from lateral advection
- SH_TENDENCY_VERT_REMAP = salt convergence to cell from vertical remapping
- OSALTDIFF = salt convergence to cell from diabatic diffusion
- OSALTPMDIFF = salt convergence to cell from neutral diffusion
- BOUNDARY_FORCING_SALT_TENDENCY = salt convergence to cell from boundary fluxes
- SH_TENDENCY_VERT_REMAP has zero vertical sum, as it redistributes salt in vertical.
- OSALTDIFF has zero vertical sum, as it redistributes salt in the vertical.
- BOUNDARY_FORCING_SALT_TENDENCY generally has 3d structure, with k $>$ 1 contributions from the case when layers are tiny, in which case MOM6 partitions tendencies into k $>$ 1 layers.
- SFDSI = BOUNDARY_FORCING_SALT_TENDENCY[k=@sum]

  Here is an example 2d salt budget (depth summed) diagnostic for MOM.
- OSALTTEND_2d = S_ADVECTION_XY_2d + OSALTPMDIFF_2d + SFDSI (+ SALT_FLUX_RESTORE)

42.1.5 Function/Subroutine Documentation

42.1.5.1 adjust_ssh_for_p_atm()

```
subroutine mom::adjust_ssh_for_p_atm (
            type(thermo_var_ptrs), intent(in) tv,
            type(ocean_grid_type), intent(in) G,
            type(verticalgrid_type), intent(in) GV,
            type(unit_scale_type), intent(in) US,
            real, dimension( g %isd:  g %ied, g %jsd:  g %jed), intent(inout) ssh,
            real, dimension(:,:), optional, pointer p_atm,
            logical, intent(in), optional use_EOS )  [private]
```

Apply a correction to the sea surface height to compensate for the atmospheric pressure (the inverse barometer).

Parameters

in	*tv*	A structure pointing to various thermodynamic variables
in	*g*	ocean grid structure
in	*gv*	ocean vertical grid structure
in	*us*	A dimensional unit scaling type
in,out	*ssh*	time mean surface height [m]
	p_atm	Ocean surface pressure [R L2 T-2 ~> Pa]
in	*use_eos*	If true, calculate the density for the SSH correction using the equation of state.

Definition at line 2964 of file MOM.F90.

Referenced by step_mom(), and step_offline().

```
02964   type(thermo_var_ptrs),             intent(in)    :: tv  !< A structure pointing to various thermodynamic
        variables
02965   type(ocean_grid_type),             intent(in)    :: g   !< ocean grid structure
02966   type(verticalgrid_type),           intent(in)    :: gv  !< ocean vertical grid structure
02967   type(unit_scale_type),             intent(in)    :: us  !< A dimensional unit scaling type
02968   real, dimension(SZI_(G),SZJ_(G)),  intent(inout) :: ssh !< time mean surface height [m]
02969   real, dimension(:,:),    optional, pointer       :: p_atm !< Ocean surface pressure [R L2 T-2 ~> Pa]
02970   logical,                 optional, intent(in)    :: use_eos !< If true, calculate the density for
02971                                                          !! the SSH correction using the equation of state.
02972
02973   real :: rho_conv(szi_(g))  ! The density used to convert surface pressure to
02974                              ! a corrected effective SSH [R ~> kg m-3].
02975   real :: igr0       ! The SSH conversion factor from R L2 T-2 to m [m T2 R-1 L-2 ~> m Pa-1].
02976   logical :: calc_rho
02977   integer, dimension(2) :: eosdom ! The i-computational domain for the equation of state
02978   integer :: i, j, is, ie, js, je
02979
02980   is = g%isc ; ie = g%iec ; js = g%jsc ; je = g%jec
02981   eosdom(:) = eos_domain(g%HI)
02982   if (present(p_atm)) then ; if (associated(p_atm)) then
02983     calc_rho = associated(tv%eqn_of_state)
02984     if (present(use_eos) .and. calc_rho) calc_rho = use_eos
02985     ! Correct the output sea surface height for the contribution from the ice pressure.
02986     do j=js,je
02987       if (calc_rho) then
02988         call calculate_density(tv%T(:,j,1), tv%S(:,j,1), 0.5*p_atm(:,j), rho_conv, &
02989                                tv%eqn_of_state, eosdom)
02990         do i=is,ie
02991           igr0 = us%Z_to_m / (rho_conv(i) * gv%g_Earth)
02992           ssh(i,j) = ssh(i,j) + p_atm(i,j) * igr0
02993         enddo
02994       else
```

```
02995         do i=is,ie
02996           ssh(i,j) = ssh(i,j) + p_atm(i,j) * (us%Z_to_m / (gv%Rho0 * gv%g_Earth))
02997         enddo
02998       endif
02999     enddo
03000   endif ; endif
03001
```

42.1.5.2 extract_surface_state()

```
subroutine, public mom::extract_surface_state (
             type(mom_control_struct), pointer CS,
             type(surface), intent(inout), target sfc_state_in )
```

Set the surface (return) properties of the ocean model by setting the appropriate fields in sfc_state. Unused fields are set to NULL or are unallocated.

Parameters

	cs	Master MOM control structure
in,out	*sfc_state_in*	transparent ocean surface state structure shared with the calling routine data in this structure is intent out.

Definition at line 3008 of file MOM.F90.

Referenced by step_mom(), and step_offline().

```
03008   type(mom_control_struct), pointer    :: cs !< Master MOM control structure
03009   type(surface), target, intent(inout) :: sfc_state_in !< transparent ocean surface state
03010                                           !! structure shared with the calling routine
03011                                           !! data in this structure is intent out.
03012
03013   ! Local variables
03014   real :: hu, hv  ! Thicknesses interpolated to velocity points [H ~> m or kg m-2]
03015   type(ocean_grid_type),   pointer :: g => null() !< pointer to a structure containing
03016                                                   !! metrics and related information
03017   type(ocean_grid_type),   pointer :: g_in => null() !< Input grid metric
03018   type(verticalgrid_type), pointer :: gv => null() !< structure containing vertical grid info
03019   type(unit_scale_type),   pointer :: us => null() !< structure containing various unit conversion factors
03020   type(surface),           pointer :: sfc_state => null()  ! surface state on the model grid
03021   real, dimension(:,:,:),  pointer :: &
03022     h => null()    !< h : layer thickness [H ~> m or kg m-2]
03023   real :: depth(szi_(cs%g))  !< Distance from the surface in depth units [Z ~> m] or [H ~> m or kg m-2]
03024   real :: depth_ml           !< Depth over which to average to determine mixed
03025                              !! layer properties [Z ~> m] or [H ~> m or kg m-2]
03026   real :: dh                 !< Thickness of a layer within the mixed layer [Z ~> m] or [H ~> m or kg m-2]
03027   real :: mass               !< Mass per unit area of a layer [R Z ~> kg m-2]
03028   real :: t_freeze           !< freezing temperature [degC]
03029   real :: i_depth            !< The inverse of depth [Z-1 ~> m-1] or [H-1 ~> m-1 or m2 kg-1]
03030   real :: missing_depth      !< The portion of depth_ml that can not be found in a column [H ~> m or kg
      m-2]
03031   real :: h_rescale          !< A conversion factor from thickness units to the units used in the
03032                              !! calculation of properties of the uppermost ocean [nondim] or [Z H-1 ~> 1 or
      m3 kg-1]
03033                              !  After the ANSWERS_2018 flag has been obsoleted, H_rescale will be 1.
03034   real :: delt(szi_(cs%g))   !< Depth integral of T-T_freeze [Z degC ~> m degC]
03035   logical :: use_temperature !< If true, temp and saln used as state variables.
03036   integer :: i, j, k, is, ie, js, je, nz, numberoferrors, ig, jg
03037   integer :: isd, ied, jsd, jed
03038   integer :: iscb, iecb, jscb, jecb, isdb, iedb, jsdb, jedb
03039   logical :: localerror
03040   character(240) :: msg
03041   integer :: turns    ! Number of quarter turns
03042
03043   call calltree_enter("extract_surface_state(), MOM.F90")
03044   g => cs%G ; g_in => cs%G_in ; gv => cs%GV ; us => cs%US
03045   is  = g%isc ; ie  = g%iec ; js  = g%jsc ; je  = g%jec ; nz = gv%ke
```

```
03046   isd = g%isd ; ied = g%ied ; jsd = g%jsd ; jed = g%jed
03047   iscb = g%iscB ; iecb = g%iecB; jscb = g%jscB ; jecb = g%jecB
03048   isdb = g%isdB ; iedb = g%iedB; jsdb = g%jsdB ; jedb = g%jedB
03049   h => cs%h
03050
03051   use_temperature = associated(cs%tv%T)
03052
03053   turns = 0
03054   if (cs%rotate_index) &
03055     turns = g%HI%turns
03056
03057   if (.not.sfc_state_in%arrays_allocated) &
03058     !  Consider using a run-time flag to determine whether to do the vertical
03059     ! integrals, since the 3-d sums are not negligible in cost.
03060     call allocate_surface_state(sfc_state_in, g_in, use_temperature, &
03061         do_integrals=.true., omit_frazil=.not.associated(cs%tv%frazil))
03062
03063   if (cs%rotate_index) then
03064     allocate(sfc_state)
03065     call allocate_surface_state(sfc_state, g, use_temperature, &
03066         do_integrals=.true., omit_frazil=.not.associated(cs%tv%frazil))
03067   else
03068     sfc_state => sfc_state_in
03069   endif
03070
03071   sfc_state%T_is_conT = cs%tv%T_is_conT
03072   sfc_state%S_is_absS = cs%tv%S_is_absS
03073
03074   do j=js,je ; do i=is,ie
03075     sfc_state%sea_lev(i,j) = us%m_to_Z*cs%ave_ssh_ibc(i,j)
03076   enddo ; enddo
03077
03078   if (allocated(sfc_state%frazil) .and. associated(cs%tv%frazil)) then ; do j=js,je ; do i=is,ie
03079     sfc_state%frazil(i,j) = cs%tv%frazil(i,j)
03080   enddo ; enddo ; endif
03081
03082   ! copy Hml into sfc_state, so that caps can access it
03083   if (associated(cs%Hml)) then
03084     do j=js,je ; do i=is,ie
03085       sfc_state%Hml(i,j) = cs%Hml(i,j)
03086     enddo ; enddo
03087   endif
03088
03089   if (cs%Hmix < 0.0) then  ! A bulk mixed layer is in use, so layer 1 has the properties
03090     if (use_temperature) then ; do j=js,je ; do i=is,ie
03091       sfc_state%SST(i,j) = cs%tv%T(i,j,1)
03092       sfc_state%SSS(i,j) = cs%tv%S(i,j,1)
03093     enddo ; enddo ; endif
03094     do j=js,je ; do i=is-1,ie
03095       sfc_state%u(i,j) = cs%u(i,j,1)
03096     enddo ; enddo
03097     do j=js-1,je ; do i=is,ie
03098       sfc_state%v(i,j) = cs%v(i,j,1)
03099     enddo ; enddo
03100
03101   else  ! (CS%Hmix >= 0.0)
03102     h_rescale = 1.0 ; if (cs%answers_2018) h_rescale = gv%H_to_Z
03103     depth_ml = cs%Hmix
03104     if (.not.cs%answers_2018) depth_ml = cs%Hmix*gv%Z_to_H
03105     ! Determine the mean tracer properties of the uppermost depth_ml fluid.
03106
03107     !$OMP parallel do default(shared) private(depth,dh)
03108     do j=js,je
03109       do i=is,ie
03110         depth(i) = 0.0
03111         if (use_temperature) then
03112           sfc_state%SST(i,j) = 0.0 ; sfc_state%SSS(i,j) = 0.0
03113         else
03114           sfc_state%sfc_density(i,j) = 0.0
03115         endif
03116       enddo
03117
03118       do k=1,nz ; do i=is,ie
03119         if (depth(i) + h(i,j,k)*h_rescale < depth_ml) then
03120           dh = h(i,j,k)*h_rescale
03121         elseif (depth(i) < depth_ml) then
03122           dh = depth_ml - depth(i)
03123         else
03124           dh = 0.0
03125         endif
03126         if (use_temperature) then
03127           sfc_state%SST(i,j) = sfc_state%SST(i,j) + dh * cs%tv%T(i,j,k)
03128           sfc_state%SSS(i,j) = sfc_state%SSS(i,j) + dh * cs%tv%S(i,j,k)
03129         else
03130           sfc_state%sfc_density(i,j) = sfc_state%sfc_density(i,j) + dh * gv%Rlay(k)
03131         endif
03132         depth(i) = depth(i) + dh
```

```
03133       enddo ; enddo
03134     ! Calculate the average properties of the mixed layer depth.
03135       do i=is,ie
03136         if (cs%answers_2018) then
03137           if (depth(i) < gv%H_subroundoff*h_rescale) &
03138               depth(i) = gv%H_subroundoff*h_rescale
03139           if (use_temperature) then
03140             sfc_state%SST(i,j) = sfc_state%SST(i,j) / depth(i)
03141             sfc_state%SSS(i,j) = sfc_state%SSS(i,j) / depth(i)
03142           else
03143             sfc_state%sfc_density(i,j) = sfc_state%sfc_density(i,j) / depth(i)
03144           endif
03145         else
03146           if (depth(i) < gv%H_subroundoff*h_rescale) then
03147             i_depth = 1.0 / (gv%H_subroundoff*h_rescale)
03148             missing_depth = gv%H_subroundoff*h_rescale - depth(i)
03149             if (use_temperature) then
03150               sfc_state%SST(i,j) = (sfc_state%SST(i,j) + missing_depth*cs%tv%T(i,j,1)) * i_depth
03151               sfc_state%SSS(i,j) = (sfc_state%SSS(i,j) + missing_depth*cs%tv%S(i,j,1)) * i_depth
03152             else
03153               sfc_state%sfc_density(i,j) = (sfc_state%sfc_density(i,j) + &
03154                                             missing_depth*gv%Rlay(1)) * i_depth
03155             endif
03156           else
03157             i_depth = 1.0 / depth(i)
03158             if (use_temperature) then
03159               sfc_state%SST(i,j) = sfc_state%SST(i,j) * i_depth
03160               sfc_state%SSS(i,j) = sfc_state%SSS(i,j) * i_depth
03161             else
03162               sfc_state%sfc_density(i,j) = sfc_state%sfc_density(i,j) * i_depth
03163             endif
03164           endif
03165         endif
03166       enddo
03167     enddo ! end of j loop
03168
03169 !   Determine the mean velocities in the uppermost depth_ml fluid.
03170     ! NOTE: Velocity loops start on `[ij]s-1` in order to update halo values
03171     !       required by the speed diagnostic on the non-symmetric grid.
03172     !       This assumes that u and v halos have already been updated.
03173     if (cs%Hmix_UV>0.) then
03174       depth_ml = cs%Hmix_UV
03175       if (.not.cs%answers_2018) depth_ml = cs%Hmix_UV*gv%Z_to_H
03176       !$OMP parallel do default(shared) private(depth,dh,hv)
03177       do j=js-1,ie
03178         do i=is,ie
03179           depth(i) = 0.0
03180           sfc_state%v(i,j) = 0.0
03181         enddo
03182         do k=1,nz ; do i=is,ie
03183           hv = 0.5 * (h(i,j,k) + h(i,j+1,k)) * h_rescale
03184           if (depth(i) + hv < depth_ml) then
03185             dh = hv
03186           elseif (depth(i) < depth_ml) then
03187             dh = depth_ml - depth(i)
03188           else
03189             dh = 0.0
03190           endif
03191           sfc_state%v(i,j) = sfc_state%v(i,j) + dh * cs%v(i,j,k)
03192           depth(i) = depth(i) + dh
03193         enddo ; enddo
03194         ! Calculate the average properties of the mixed layer depth.
03195         do i=is,ie
03196           sfc_state%v(i,j) = sfc_state%v(i,j) / max(depth(i), gv%H_subroundoff*h_rescale)
03197         enddo
03198       enddo ! end of j loop
03199
03200       !$OMP parallel do default(shared) private(depth,dh,hu)
03201       do j=js,je
03202         do i=is-1,ie
03203           depth(i) = 0.0
03204           sfc_state%u(i,j) = 0.0
03205         enddo
03206         do k=1,nz ; do i=is-1,ie
03207           hu = 0.5 * (h(i,j,k) + h(i+1,j,k)) * h_rescale
03208           if (depth(i) + hu < depth_ml) then
03209             dh = hu
03210           elseif (depth(i) < depth_ml) then
03211             dh = depth_ml - depth(i)
03212           else
03213             dh = 0.0
03214           endif
03215           sfc_state%u(i,j) = sfc_state%u(i,j) + dh * cs%u(i,j,k)
03216           depth(i) = depth(i) + dh
03217         enddo ; enddo
03218         ! Calculate the average properties of the mixed layer depth.
03219         do i=is-1,ie
```

```
03220           sfc_state%u(i,j) = sfc_state%u(i,j) / max(depth(i), gv%H_subroundoff*h_rescale)
03221         enddo
03222       enddo ! end of j loop
03223     else ! Hmix_UV<=0.
03224       do j=js,je ; do i=is-1,ie
03225         sfc_state%u(i,j) = cs%u(i,j,1)
03226       enddo ; enddo
03227       do j=js-1,je ; do i=is,ie
03228         sfc_state%v(i,j) = cs%v(i,j,1)
03229       enddo ; enddo
03230     endif
03231   endif  ! (CS%Hmix >= 0.0)
03232
03233
03234   if (allocated(sfc_state%melt_potential)) then
03235     !$OMP parallel do default(shared) private(depth_ml, dh, T_freeze, depth, delT)
03236     do j=js,je
03237       do i=is,ie
03238         depth(i) = 0.0
03239         delt(i) = 0.0
03240       enddo
03241
03242       do k=1,nz ; do i=is,ie
03243         depth_ml = min(cs%HFrz, cs%visc%MLD(i,j))
03244         if (depth(i) + h(i,j,k)*gv%H_to_Z < depth_ml) then
03245           dh = h(i,j,k)*gv%H_to_Z
03246         elseif (depth(i) < depth_ml) then
03247           dh = depth_ml - depth(i)
03248         else
03249           dh = 0.0
03250         endif
03251
03252         ! p=0 OK, HFrz ~ 10 to 20m
03253         call calculate_tfreeze(cs%tv%S(i,j,k), 0.0, t_freeze, cs%tv%eqn_of_state)
03254         depth(i) = depth(i) + dh
03255         delt(i) =  delt(i) + dh * (cs%tv%T(i,j,k) - t_freeze)
03256       enddo ; enddo
03257
03258       do i=is,ie
03259        ! set melt_potential to zero to avoid passing previous values
03260        sfc_state%melt_potential(i,j) = 0.0
03261
03262        if (g%mask2dT(i,j)>0.) then
03263          ! instantaneous melt_potential [Q R Z ~> J m-2]
03264          sfc_state%melt_potential(i,j) = cs%tv%C_p * gv%Rho0 * delt(i)
03265        endif
03266       enddo
03267     enddo ! end of j loop
03268   endif   ! melt_potential
03269
03270   if (allocated(sfc_state%salt_deficit) .and. associated(cs%tv%salt_deficit)) then
03271     !$OMP parallel do default(shared)
03272     do j=js,je ; do i=is,ie
03273       ! Convert from gSalt to kgSalt
03274       sfc_state%salt_deficit(i,j) = 0.001 * cs%tv%salt_deficit(i,j)
03275     enddo ; enddo
03276   endif
03277   if (allocated(sfc_state%TempxPmE) .and. associated(cs%tv%TempxPmE)) then
03278     !$OMP parallel do default(shared)
03279     do j=js,je ; do i=is,ie
03280       sfc_state%TempxPmE(i,j) = cs%tv%TempxPmE(i,j)
03281     enddo ; enddo
03282   endif
03283   if (allocated(sfc_state%internal_heat) .and. associated(cs%tv%internal_heat)) then
03284     !$OMP parallel do default(shared)
03285     do j=js,je ; do i=is,ie
03286       sfc_state%internal_heat(i,j) = cs%tv%internal_heat(i,j)
03287     enddo ; enddo
03288   endif
03289   if (allocated(sfc_state%taux_shelf) .and. associated(cs%visc%taux_shelf)) then
03290     !$OMP parallel do default(shared)
03291     do j=js,je ; do i=is-1,ie
03292       sfc_state%taux_shelf(i,j) = cs%visc%taux_shelf(i,j)
03293     enddo ; enddo
03294   endif
03295   if (allocated(sfc_state%tauy_shelf) .and. associated(cs%visc%tauy_shelf)) then
03296     !$OMP parallel do default(shared)
03297     do j=js-1,je ; do i=is,ie
03298       sfc_state%tauy_shelf(i,j) = cs%visc%tauy_shelf(i,j)
03299     enddo ; enddo
03300   endif
03301
03302   if (allocated(sfc_state%ocean_mass) .and. allocated(sfc_state%ocean_heat) .and. &
03303       allocated(sfc_state%ocean_salt)) then
03304     !$OMP parallel do default(shared)
03305     do j=js,je ; do i=is,ie
03306       sfc_state%ocean_mass(i,j) = 0.0
```

```
03307       sfc_state%ocean_heat(i,j) = 0.0 ; sfc_state%ocean_salt(i,j) = 0.0
03308     enddo ; enddo
03309     !$OMP parallel do default(shared) private(mass)
03310     do j=js,je ; do k=1,nz; do i=is,ie
03311       mass = gv%H_to_RZ*h(i,j,k)
03312       sfc_state%ocean_mass(i,j) = sfc_state%ocean_mass(i,j) + mass
03313       sfc_state%ocean_heat(i,j) = sfc_state%ocean_heat(i,j) + mass * cs%tv%T(i,j,k)
03314       sfc_state%ocean_salt(i,j) = sfc_state%ocean_salt(i,j) + mass * (1.0e-3*cs%tv%S(i,j,k))
03315     enddo ; enddo ; enddo
03316   else
03317     if (allocated(sfc_state%ocean_mass)) then
03318       !$OMP parallel do default(shared)
03319       do j=js,je ; do i=is,ie ; sfc_state%ocean_mass(i,j) = 0.0 ; enddo ; enddo
03320       !$OMP parallel do default(shared)
03321       do j=js,je ; do k=1,nz ; do i=is,ie
03322         sfc_state%ocean_mass(i,j) = sfc_state%ocean_mass(i,j) + gv%H_to_RZ*h(i,j,k)
03323       enddo ; enddo ; enddo
03324     endif
03325     if (allocated(sfc_state%ocean_heat)) then
03326       !$OMP parallel do default(shared)
03327       do j=js,je ; do i=is,ie ; sfc_state%ocean_heat(i,j) = 0.0 ; enddo ; enddo
03328       !$OMP parallel do default(shared) private(mass)
03329       do j=js,je ; do k=1,nz ; do i=is,ie
03330         mass = gv%H_to_RZ*h(i,j,k)
03331         sfc_state%ocean_heat(i,j) = sfc_state%ocean_heat(i,j) + mass*cs%tv%T(i,j,k)
03332       enddo ; enddo ; enddo
03333     endif
03334     if (allocated(sfc_state%ocean_salt)) then
03335       !$OMP parallel do default(shared)
03336       do j=js,je ; do i=is,ie ; sfc_state%ocean_salt(i,j) = 0.0 ; enddo ; enddo
03337       !$OMP parallel do default(shared) private(mass)
03338       do j=js,je ; do k=1,nz ; do i=is,ie
03339         mass = gv%H_to_RZ*h(i,j,k)
03340         sfc_state%ocean_salt(i,j) = sfc_state%ocean_salt(i,j) + mass * (1.0e-3*cs%tv%S(i,j,k))
03341       enddo ; enddo ; enddo
03342     endif
03343   endif
03344
03345   if (associated(cs%tracer_flow_CSp)) then
03346     call call_tracer_surface_state(sfc_state, h, g, cs%tracer_flow_CSp)
03347   endif
03348
03349   if (cs%check_bad_sfc_vals) then
03350     numberoferrors=0 ! count number of errors
03351     do j=js,je; do i=is,ie
03352       if (g%mask2dT(i,j)>0.) then
03353         localerror = sfc_state%sea_lev(i,j) <= -g%bathyT(i,j) &
03354                 .or. sfc_state%sea_lev(i,j) >=  cs%bad_val_ssh_max  &
03355                 .or. sfc_state%sea_lev(i,j) <= -cs%bad_val_ssh_max  &
03356                 .or. sfc_state%sea_lev(i,j) + g%bathyT(i,j) < cs%bad_val_col_thick
03357         if (use_temperature) localerror = localerror &
03358                 .or. sfc_state%SSS(i,j)<0.                          &
03359                 .or. sfc_state%SSS(i,j)>=cs%bad_val_sss_max         &
03360                 .or. sfc_state%SST(i,j)< cs%bad_val_sst_min         &
03361                 .or. sfc_state%SST(i,j)>=cs%bad_val_sst_max
03362         if (localerror) then
03363           numberoferrors=numberoferrors+1
03364           if (numberoferrors<9) then ! Only report details for the first few errors
03365             ig = i + g%HI%idg_offset ! Global i-index
03366             jg = j + g%HI%jdg_offset ! Global j-index
03367             if (use_temperature) then
03368               write(msg(1:240),'(2(a,i4,x),4(a,f8.3,x),8(a,es11.4,x))') &
03369                 'Extreme surface sfc_state detected: i=',ig,'j=',jg, &
03370                 'lon=',g%geoLonT(i,j), 'lat=',g%geoLatT(i,j), &
03371                 'x=',g%gridLonT(ig), 'y=',g%gridLatT(jg), &
03372                 'D=',cs%US%Z_to_m*g%bathyT(i,j),  'SSH=',cs%US%Z_to_m*sfc_state%sea_lev(i,j), &
03373                 'SST=',sfc_state%SST(i,j), 'SSS=',sfc_state%SSS(i,j), &
03374                 'U-=',us%L_T_to_m_s*sfc_state%u(i-1,j), 'U+=',us%L_T_to_m_s*sfc_state%u(i,j), &
03375                 'V-=',us%L_T_to_m_s*sfc_state%v(i,j-1), 'V+=',us%L_T_to_m_s*sfc_state%v(i,j)
03376             else
03377               write(msg(1:240),'(2(a,i4,x),4(a,f8.3,x),6(a,es11.4))') &
03378                 'Extreme surface sfc_state detected: i=',ig,'j=',jg, &
03379                 'lon=',g%geoLonT(i,j), 'lat=',g%geoLatT(i,j), &
03380                 'x=',g%gridLonT(i), 'y=',g%gridLatT(j), &
03381                 'D=',cs%US%Z_to_m*g%bathyT(i,j),  'SSH=',cs%US%Z_to_m*sfc_state%sea_lev(i,j), &
03382                 'U-=',us%L_T_to_m_s*sfc_state%u(i-1,j), 'U+=',us%L_T_to_m_s*sfc_state%u(i,j), &
03383                 'V-=',us%L_T_to_m_s*sfc_state%v(i,j-1), 'V+=',us%L_T_to_m_s*sfc_state%v(i,j)
03384             endif
03385             call mom_error(warning, trim(msg), all_print=.true.)
03386           elseif (numberoferrors==9) then ! Indicate once that there are more errors
03387             call mom_error(warning, 'There were more unreported extreme events!', all_print=.true.)
03388           endif ! numberOfErrors
03389         endif ! localError
03390       endif ! mask2dT
03391     enddo ; enddo
03392     call sum_across_pes(numberoferrors)
03393     if (numberoferrors>0) then
```

```
03394       write(msg(1:240),'(3(a,i9,x))') 'There were a total of ',numberoferrors, &
03395           'locations detected with extreme surface values!'
03396       call mom_error(fatal, trim(msg))
03397     endif
03398   endif
03399
03400   if (cs%debug) call mom_surface_chksum("Post extract_sfc", sfc_state, g, us, haloshift=0)
03401
03402   ! Rotate sfc_state back onto the input grid, sfc_state_in
03403   if (cs%rotate_index) then
03404     call rotate_surface_state(sfc_state, g, sfc_state_in, g_in, -turns)
03405     call deallocate_surface_state(sfc_state)
03406   endif
03407
03408   call calltree_leave("extract_surface_sfc_state()")
```

42.1.5.3 finish_mom_initialization()

```
subroutine, public mom::finish_mom_initialization (
              type(time_type), intent(in) Time,
              type(directories), intent(in) dirs,
              type(mom_control_struct), pointer CS,
              type(mom_restart_cs), pointer restart_CSp )
```

Finishes initializing MOM and writes out the initial conditions.

Parameters

in	*time*	model time, used in this routine
in	*dirs*	structure with directory paths
	cs	pointer to MOM control structure
	restart_csp	pointer to the restart control structure that will be used for MOM.

Definition at line 2774 of file MOM.F90.

References id_clock_init.

```
02774   type(time_type),          intent(in)    :: time        !< model time, used in this routine
02775   type(directories),        intent(in)    :: dirs        !< structure with directory paths
02776   type(mom_control_struct), pointer       :: cs          !< pointer to MOM control structure
02777   type(mom_restart_cs),     pointer       :: restart_csp !< pointer to the restart control
02778                                                          !! structure that will be used for MOM.
02779   ! Local variables
02780   type(ocean_grid_type),   pointer :: g => null()  ! pointer to a structure containing
02781                                                    ! metrics and related information
02782   type(verticalgrid_type), pointer :: gv => null() ! Pointer to the vertical grid structure
02783   type(unit_scale_type),   pointer :: us => null() ! Pointer to a structure containing
02784                                                    ! various unit conversion factors
02785   type(mom_restart_cs),    pointer :: restart_csp_tmp => null()
02786   real, allocatable :: z_interface(:,:,:) ! Interface heights [m]
02787   type(vardesc) :: vd
02788
02789   call cpu_clock_begin(id_clock_init)
02790   call calltree_enter("finish_MOM_initialization()")
02791
02792   ! Pointers for convenience
02793   g => cs%G ; gv => cs%GV ; us => cs%US
02794
02795   !### Move to initialize_MOM?
02796   call fix_restart_scaling(gv)
02797   call fix_restart_unit_scaling(us)
02798
02799   ! Write initial conditions
02800   if (cs%write_IC) then
02801     allocate(restart_csp_tmp)
```

```
02802     restart_csp_tmp = restart_csp
02803     allocate(z_interface(szi_(g),szj_(g),szk_(g)+1))
02804     call find_eta(cs%h, cs%tv, g, gv, us, z_interface, eta_to_m=1.0)
02805     call register_restart_field(z_interface, "eta", .true., restart_csp_tmp, &
02806                                 "Interface heights", "meter", z_grid='i')
02807
02808     call save_restart(dirs%output_directory, time, cs%G_in, &
02809                       restart_csp_tmp, filename=cs%IC_file, gv=gv)
02810     deallocate(z_interface)
02811     deallocate(restart_csp_tmp)
02812   endif
02813
02814   call write_energy(cs%u, cs%v, cs%h, cs%tv, time, 0, g, gv, us, &
02815                     cs%sum_output_CSp, cs%tracer_flow_CSp)
02816
02817   call calltree_leave("finish_MOM_initialization()")
02818   call cpu_clock_end(id_clock_init)
02819
```

42.1.5.4 get_mom_state_elements()

```
subroutine, public mom::get_mom_state_elements (
             type(mom_control_struct), pointer CS,
             type(ocean_grid_type), optional, pointer G,
             type(verticalgrid_type), optional, pointer GV,
             type(unit_scale_type), optional, pointer US,
             real, intent(out), optional C_p,
             real, intent(out), optional C_p_scaled,
             logical, intent(out), optional use_temp )
```

This subroutine offers access to values or pointers to other types from within the MOM_control_struct, allowing the MOM_control_struct to be opaque.

Parameters

	cs	MOM control structure
	g	structure containing metrics and grid info
	gv	structure containing vertical grid info
	us	A dimensional unit scaling type
out	*c_p*	The heat capacity [J kg degC-1]
out	*c_p_scaled*	The heat capacity in scaled units [Q degC-1 ~> J kg degC-1]
out	*use_temp*	True if temperature is a state variable

Definition at line 3450 of file MOM.F90.

```
03450   type(mom_control_struct),          pointer     :: cs !< MOM control structure
03451   type(ocean_grid_type),   optional, pointer     :: g    !< structure containing metrics and grid info
03452   type(verticalgrid_type), optional, pointer     :: gv   !< structure containing vertical grid info
03453   type(unit_scale_type),   optional, pointer     :: us   !< A dimensional unit scaling type
03454   real,                    optional, intent(out) :: c_p  !< The heat capacity [J kg degC-1]
03455   real,                    optional, intent(out) :: c_p_scaled !< The heat capacity in scaled
03456                                                            !! units [Q degC-1 ~> J kg degC-1]
03457   logical,                 optional, intent(out) :: use_temp !< True if temperature is a state variable
03458
03459   if (present(g)) g => cs%G_in
03460   if (present(gv)) gv => cs%GV
03461   if (present(us)) us => cs%US
03462   if (present(c_p)) c_p = cs%US%Q_to_J_kg * cs%tv%C_p
03463   if (present(c_p_scaled)) c_p_scaled = cs%tv%C_p
03464   if (present(use_temp)) use_temp = associated(cs%tv%T)
```

42.1.5.5 get_ocean_stocks()

```
subroutine, public mom::get_ocean_stocks (
             type(mom_control_struct), pointer CS,
             real, intent(out), optional mass,
             real, intent(out), optional heat,
             real, intent(out), optional salt,
             logical, intent(in), optional on_PE_only )
```

Find the global integrals of various quantities.

Parameters

	cs	MOM control structure
out	*heat*	The globally integrated integrated ocean heat [J].
out	*salt*	The globally integrated integrated ocean salt [kg].
out	*mass*	The globally integrated integrated ocean mass [kg].
in	*on_pe_only*	If present and true, only sum on the local PE.

Definition at line 3469 of file MOM.F90.

```
03469   type(mom_control_struct), pointer :: cs !< MOM control structure
03470   real,    optional, intent(out) :: heat  !< The globally integrated integrated ocean heat [J].
03471   real,    optional, intent(out) :: salt  !< The globally integrated integrated ocean salt [kg].
03472   real,    optional, intent(out) :: mass  !< The globally integrated integrated ocean mass [kg].
03473   logical, optional, intent(in)  :: on_pe_only !< If present and true, only sum on the local PE.
03474
03475   if (present(mass)) &
03476     mass = global_mass_integral(cs%h, cs%G, cs%GV, on_pe_only=on_pe_only)
03477   if (present(heat)) &
03478     heat = cs%US%Q_to_J_kg*cs%tv%C_p * global_mass_integral(cs%h, cs%G, cs%GV, cs%tv%T, on_pe_only=
      on_pe_only)
03479   if (present(salt)) &
03480     salt = 1.0e-3 * global_mass_integral(cs%h, cs%G, cs%GV, cs%tv%S, on_pe_only=on_pe_only)
03481
```

42.1.5.6 initialize_mom()

```
subroutine, public mom::initialize_mom (
             type(time_type), intent(inout), target Time,
             type(time_type), intent(in) Time_init,
             type(param_file_type), intent(out) param_file,
             type(directories), intent(out) dirs,
             type(mom_control_struct), pointer CS,
             type(mom_restart_cs), pointer restart_CSp,
             type(time_type), intent(in), optional Time_in,
             logical, intent(out), optional offline_tracer_mode,
             character(len=*), intent(in), optional input_restart_file,
             type(diag_ctrl), optional, pointer diag_ptr,
             logical, intent(in), optional count_calls,
             type(tracer_flow_control_cs), optional, pointer tracer_flow_CSp )
```

Initialize MOM, including memory allocation, setting up parameters and diagnostics, initializing the ocean state variables, and initializing subsidiary modules.

Parameters

in,out	*time*	model time, set in this routine
in	*time_init*	The start time for the coupled model's calendar
out	*param_file*	structure indicating parameter file to parse
out	*dirs*	structure with directory paths
	cs	pointer set in this routine to MOM control structure
	restart_csp	pointer set in this routine to the restart control structure that will be used for MOM.
in	*time_in*	time passed to MOM_initialize_state when model is not being started from a restart file
out	*offline_tracer_mode*	True is returned if tracers are being run offline
in	*input_restart_file*	If present, name of restart file to read
	diag_ptr	A pointer set in this routine to the diagnostic regulatory structure
	tracer_flow_csp	A pointer set in this routine to
in	*count_calls*	If true, nstep_tot counts the number of calls to step_MOM instead of the number of dynamics timesteps.

Definition at line 1604 of file MOM.F90.

References id_clock_init, id_clock_mom_init, id_clock_pass_init, id_clock_unit_tests, mom_timing_init(), register_diags(), rotate_initial_state(), and set_restart_fields().

```
01604   type(time_type), target,   intent(inout) :: time        !< model time, set in this routine
01605   type(time_type),           intent(in)    :: time_init   !< The start time for the coupled model's
      calendar
01606   type(param_file_type),     intent(out)   :: param_file  !< structure indicating parameter file to parse
01607   type(directories),         intent(out)   :: dirs        !< structure with directory paths
01608   type(mom_control_struct), pointer        :: cs          !< pointer set in this routine to MOM control
      structure
01609   type(mom_restart_cs),      pointer       :: restart_csp !< pointer set in this routine to the
01610                                                           !! restart control structure that will
01611                                                           !! be used for MOM.
01612   type(time_type), optional, intent(in)    :: time_in     !< time passed to MOM_initialize_state when
01613                                                           !! model is not being started from a restart file
01614   logical,         optional, intent(out)   :: offline_tracer_mode !< True is returned if tracers are being
      run offline
01615   character(len=*),optional, intent(in)    :: input_restart_file !< If present, name of restart file to
      read
01616   type(diag_ctrl), optional, pointer       :: diag_ptr    !< A pointer set in this routine to the
      diagnostic
01617                                                           !! regulatory structure
01618   type(tracer_flow_control_cs), &
01619                    optional, pointer       :: tracer_flow_csp !< A pointer set in this routine to
01620                                                           !! the tracer flow control structure.
01621   logical,         optional, intent(in)    :: count_calls !< If true, nstep_tot counts the number of
01622                                                           !! calls to step_MOM instead of the number of
01623                                                           !! dynamics timesteps.
01624   ! local variables
01625   type(ocean_grid_type),  pointer :: g => null()    ! A pointer to the metric grid use for the run
01626   type(ocean_grid_type),  pointer :: g_in => null() ! Pointer to the input grid
01627   type(hor_index_type),   pointer :: hi => null()   ! A hor_index_type for array extents
01628   type(hor_index_type),   target  :: hi_in          ! HI on the input grid
01629   type(verticalgrid_type), pointer :: gv => null()
01630   type(dyn_horgrid_type), pointer :: dg => null()
01631   type(dyn_horgrid_type), pointer :: dg_in => null()
01632   type(diag_ctrl),        pointer :: diag => null()
01633   type(unit_scale_type),  pointer :: us => null()
01634   character(len=4), parameter :: vers_num = 'v2.0'
01635   integer :: turns   ! Number of grid quarter-turns
01636
01637   ! Initial state on the input index map
01638   real, allocatable, dimension(:,:,:) :: u_in, v_in, h_in
01639   real, allocatable, dimension(:,:,:), target :: t_in, s_in
01640   type(ocean_obc_type), pointer :: obc_in => null()
01641   type(sponge_cs), pointer :: sponge_in_csp => null()
01642   type(ale_sponge_cs), pointer :: ale_sponge_in_csp => null()
01643
01644   ! This include declares and sets the variable "version".
01645 # include "version_variable.h"
```

```
01646
01647   integer :: i, j, k, is, ie, js, je, isd, ied, jsd, jed, nz
01648   integer :: isdb, iedb, jsdb, jedb
01649   real    :: dtbt        ! The barotropic timestep [s]
01650
01651   real, allocatable, dimension(:,:)   :: eta ! free surface height or column mass [H ~> m or kg m-2]
01652   real, allocatable, dimension(:,:)   :: area_shelf_h ! area occupied by ice shelf [L2 ~> m2]
01653   real, dimension(:,:), allocatable, target  :: frac_shelf_h ! fraction of total area occupied by ice shelf
      [nondim]
01654   real, dimension(:,:), pointer :: shelf_area => null()
01655   type(mom_restart_cs),  pointer      :: restart_csp_tmp => null()
01656   type(group_pass_type) :: tmp_pass_uv_t_s_h, pass_uv_t_s_h
01657
01658   real    :: default_val       ! default value for a parameter
01659   logical :: write_geom_files  ! If true, write out the grid geometry files.
01660   logical :: ensemble_ocean    ! If true, perform an ensemble gather at the end of step_MOM
01661   logical :: new_sim
01662   logical :: use_geothermal    ! If true, apply geothermal heating.
01663   logical :: use_eos           ! If true, density calculated from T & S using an equation of state.
01664   logical :: symmetric         ! If true, use symmetric memory allocation.
01665   logical :: save_ic           ! If true, save the initial conditions.
01666   logical :: do_unit_tests     ! If true, call unit tests.
01667   logical :: test_grid_copy = .false.
01668
01669   logical :: bulkmixedlayer    ! If true, a refined bulk mixed layer scheme is used
01670                                ! with nkml sublayers and nkbl buffer layer.
01671   logical :: use_temperature   ! If true, temp and saln used as state variables.
01672   logical :: use_frazil        ! If true, liquid seawater freezes if temp below freezing,
01673                                ! with accumulated heat deficit returned to surface ocean.
01674   logical :: bound_salinity    ! If true, salt is added to keep salinity above
01675                                ! a minimum value, and the deficit is reported.
01676   logical :: default_2018_answers ! The default setting for the various 2018_ANSWERS flags.
01677   logical :: use_cont_abss     ! If true, the prognostics T & S are conservative temperature
01678                                ! and absolute salinity. Care should be taken to convert them
01679                                ! to potential temperature and practical salinity before
01680                                ! exchanging them with the coupler and/or reporting T&S diagnostics.
01681   logical :: advect_ts         ! If false, then no horizontal advection of temperature
01682                                ! and salnity is performed
01683   logical :: use_ice_shelf     ! Needed for ALE
01684   logical :: global_indexing   ! If true use global horizontal index values instead
01685                                ! of having the data domain on each processor start at 1.
01686   logical :: bathy_at_vel      ! If true, also define bathymetric fields at the
01687                                ! the velocity points.
01688   logical :: calc_dtbt         ! Indicates whether the dynamically adjusted barotropic
01689                                ! time step needs to be updated before it is used.
01690   logical :: debug_truncations ! If true, turn on diagnostics useful for debugging truncations.
01691   integer :: first_direction   ! An integer that indicates which direction is to be
01692                                ! updated first in directionally split parts of the
01693                                ! calculation.  This can be altered during the course
01694                                ! of the run via calls to set_first_direction.
01695   integer :: nkml, nkbl, verbosity, write_geom
01696   integer :: dynamics_stencil  ! The computational stencil for the calculations
01697                                ! in the dynamic core.
01698   real :: conv2watt, conv2salt
01699   real :: rl2_t2_rescale, z_rescale, qrz_rescale ! Unit conversion factors
01700   character(len=48) :: flux_units, s_flux_units
01701
01702   type(vardesc) :: vd_t, vd_s  ! Structures describing temperature and salinity variables.
01703   type(time_type)                 :: start_time
01704   type(ocean_internal_state)      :: mom_internal_state
01705   character(len=200) :: area_varname, ice_shelf_file, inputdir, filename
01706
01707   if (associated(cs)) then
01708     call mom_error(warning, "initialize_MOM called with an associated "// &
01709                             "control structure.")
01710     return
01711   endif
01712   allocate(cs)
01713
01714   cs%Time => time
01715
01716   id_clock_init = cpu_clock_id('Ocean Initialization', grain=clock_subcomponent)
01717   call cpu_clock_begin(id_clock_init)
01718
01719   start_time = time ; if (present(time_in)) start_time = time_in
01720
01721   ! Read paths and filenames from namelist and store in "dirs".
01722   ! Also open the parsed input parameter file(s) and setup param_file.
01723   call get_mom_input(param_file, dirs, default_input_filename=input_restart_file)
01724
01725   verbosity = 2 ; call read_param(param_file, "VERBOSITY", verbosity)
01726   call mom_set_verbosity(verbosity)
01727   call calltree_enter("initialize_MOM(), MOM.F90")
01728
01729   call find_obsolete_params(param_file)
01730
01731   ! Determining the internal unit scaling factors for this run.
```

```
01732   call unit_scaling_init(param_file, cs%US)
01733   us => cs%US
01734
01735   ! Read relevant parameters and write them to the model log.
01736   call log_version(param_file, "MOM", version, "", log_to_all=.true., layout=.true., debugging=.true.)
01737   call get_param(param_file, "MOM", "VERBOSITY", verbosity,  &
01738                  "Integer controlling level of messaging\n" // &
01739                  "\t0 = Only FATAL messages\n" // &
01740                  "\t2 = Only FATAL, WARNING, NOTE [default]\n" // &
01741                  "\t9 = All)", default=2, debuggingparam=.true.)
01742   call get_param(param_file, "MOM", "DO_UNIT_TESTS", do_unit_tests, &
01743                  "If True, exercises unit tests at model start up.", &
01744                  default=.false., debuggingparam=.true.)
01745   if (do_unit_tests) then
01746     id_clock_unit_tests = cpu_clock_id('(Ocean unit tests)', grain=clock_module)
01747     call cpu_clock_begin(id_clock_unit_tests)
01748     call unit_tests(verbosity)
01749     call cpu_clock_end(id_clock_unit_tests)
01750   endif
01751
01752   call get_param(param_file, "MOM", "SPLIT", cs%split, &
01753                  "Use the split time stepping if true.", default=.true.)
01754   if (cs%split) then
01755     cs%use_RK2 = .false.
01756   else
01757     call get_param(param_file, "MOM", "USE_RK2", cs%use_RK2, &
01758                  "If true, use RK2 instead of RK3 in the unsplit time stepping.", &
01759                  default=.false.)
01760   endif
01761
01762   call get_param(param_file, "MOM", "CALC_RHO_FOR_SEA_LEVEL", cs%calc_rho_for_sea_lev, &
01763                  "If true, the in-situ density is used to calculate the "//&
01764                  "effective sea level that is returned to the coupler. If false, "//&
01765                  "the Boussinesq parameter RHO_0 is used.", default=.false.)
01766   call get_param(param_file, "MOM", "ENABLE_THERMODYNAMICS", use_temperature, &
01767                  "If true, Temperature and salinity are used as state "//&
01768                  "variables.", default=.true.)
01769   call get_param(param_file, "MOM", "USE_EOS", use_eos, &
01770                  "If true,  density is calculated from temperature and "//&
01771                  "salinity with an equation of state.  If USE_EOS is "//&
01772                  "true, ENABLE_THERMODYNAMICS must be true as well.", &
01773                  default=use_temperature)
01774   call get_param(param_file, "MOM", "DIABATIC_FIRST", cs%diabatic_first, &
01775                  "If true, apply diabatic and thermodynamic processes, "//&
01776                  "including buoyancy forcing and mass gain or loss, "//&
01777                  "before stepping the dynamics forward.", default=.false.)
01778   call get_param(param_file, "MOM", "USE_CONTEMP_ABSSAL", use_cont_abss, &
01779                  "If true, the prognostics T&S are the conservative temperature "//&
01780                  "and absolute salinity. Care should be taken to convert them "//&
01781                  "to potential temperature and practical salinity before "//&
01782                  "exchanging them with the coupler and/or reporting T&S diagnostics.", &
01783                  default=.false.)
01784   cs%tv%T_is_conT = use_cont_abss ; cs%tv%S_is_absS = use_cont_abss
01785   call get_param(param_file, "MOM", "ADIABATIC", cs%adiabatic, &
01786                  "There are no diapycnal mass fluxes if ADIABATIC is "//&
01787                  "true. This assumes that KD = KDML = 0.0 and that "//&
01788                  "there is no buoyancy forcing, but makes the model "//&
01789                  "faster by eliminating subroutine calls.", default=.false.)
01790   call get_param(param_file, "MOM", "DO_DYNAMICS", cs%do_dynamics, &
01791                  "If False, skips the dynamics calls that update u & v, as well as "//&
01792                  "the gravity wave adjustment to h. This may be a fragile feature, "//&
01793                  "but can be useful during development", default=.true.)
01794   call get_param(param_file, "MOM", "ADVECT_TS", advect_ts, &
01795                  "If True, advect temperature and salinity horizontally "//&
01796                  "If False, T/S are registered for advection. "//&
01797                  "This is intended only to be used in offline tracer mode "//&
01798                  "and is by default false in that case.", &
01799                  do_not_log = .true., default=.true. )
01800   if (present(offline_tracer_mode)) then ! Only read this parameter in enabled modes
01801     call get_param(param_file, "MOM", "OFFLINE_TRACER_MODE", cs%offline_tracer_mode, &
01802                  "If true, barotropic and baroclinic dynamics, thermodynamics "//&
01803                  "are all bypassed with all the fields necessary to integrate "//&
01804                  "the tracer advection and diffusion equation are read in from "//&
01805                  "files stored from a previous integration of the prognostic model. "//&
01806                  "NOTE: This option only used in the ocean_solo_driver.", default=.false.)
01807     if (cs%offline_tracer_mode) then
01808       call get_param(param_file, "MOM", "ADVECT_TS", advect_ts, &
01809                    "If True, advect temperature and salinity horizontally "//&
01810                    "If False, T/S are registered for advection. "//&
01811                    "This is intended only to be used in offline tracer mode."//&
01812                    "and is by default false in that case", &
01813                    default=.false. )
01814     endif
01815   endif
01816   call get_param(param_file, "MOM", "USE_REGRIDDING", cs%use_ALE_algorithm, &
01817                  "If True, use the ALE algorithm (regridding/remapping). "//&
01818                  "If False, use the layered isopycnal algorithm.", default=.false. )
```

```
01819   call get_param(param_file, "MOM", "BULKMIXEDLAYER", bulkmixedlayer, &
01820                  "If true, use a Kraus-Turner-like bulk mixed layer "//&
01821                  "with transitional buffer layers.  Layers 1 through "//&
01822                  "NKML+NKBL have variable densities. There must be at "//&
01823                  "least NKML+NKBL+1 layers if BULKMIXEDLAYER is true. "//&
01824                  "BULKMIXEDLAYER can not be used with USE_REGRIDDING. "//&
01825                  "The default is influenced by ENABLE_THERMODYNAMICS.", &
01826                  default=use_temperature .and. .not.cs%use_ALE_algorithm)
01827   call get_param(param_file, "MOM", "THICKNESSDIFFUSE", cs%thickness_diffuse, &
01828                  "If true, interface heights are diffused with a "//&
01829                  "coefficient of KHTH.", default=.false.)
01830   call get_param(param_file, "MOM",  "THICKNESSDIFFUSE_FIRST", &
01831                                      cs%thickness_diffuse_first, &
01832                  "If true, do thickness diffusion before dynamics. "//&
01833                  "This is only used if THICKNESSDIFFUSE is true.", &
01834                  default=.false.)
01835   if (.not.cs%thickness_diffuse) cs%thickness_diffuse_first = .false.
01836   call get_param(param_file, "MOM", "BATHYMETRY_AT_VEL", bathy_at_vel, &
01837                  "If true, there are separate values for the basin depths "//&
01838                  "at velocity points.  Otherwise the effects of topography "//&
01839                  "are entirely determined from thickness points.", &
01840                  default=.false.)
01841   call get_param(param_file, "MOM", "USE_WAVES", cs%UseWaves, default=.false., &
01842                  do_not_log=.true.)
01843
01844   call get_param(param_file, "MOM", "DEBUG", cs%debug, &
01845                  "If true, write out verbose debugging data.", &
01846                  default=.false., debuggingparam=.true.)
01847   call get_param(param_file, "MOM", "DEBUG_TRUNCATIONS", debug_truncations, &
01848                  "If true, calculate all diagnostics that are useful for "//&
01849                  "debugging truncations.", default=.false., debuggingparam=.true.)
01850
01851   call get_param(param_file, "MOM", "DT", cs%dt, &
01852                  "The (baroclinic) dynamics time step.  The time-step that "//&
01853                  "is actually used will be an integer fraction of the "//&
01854                  "forcing time-step (DT_FORCING in ocean-only mode or the "//&
01855                  "coupling timestep in coupled mode.)", units="s", scale=us%s_to_T, &
01856                  fail_if_missing=.true.)
01857   call get_param(param_file, "MOM", "DT_THERM", cs%dt_therm, &
01858                  "The thermodynamic and tracer advection time step. "//&
01859                  "Ideally DT_THERM should be an integer multiple of DT "//&
01860                  "and less than the forcing or coupling time-step, unless "//&
01861                  "THERMO_SPANS_COUPLING is true, in which case DT_THERM "//&
01862                  "can be an integer multiple of the coupling timestep.  By "//&
01863                  "default DT_THERM is set to DT.", &
01864                  units="s", scale=us%s_to_T, default=us%T_to_s*cs%dt)
01865   call get_param(param_file, "MOM", "THERMO_SPANS_COUPLING", cs%thermo_spans_coupling, &
01866                  "If true, the MOM will take thermodynamic and tracer "//&
01867                  "timesteps that can be longer than the coupling timestep. "//&
01868                  "The actual thermodynamic timestep that is used in this "//&
01869                  "case is the largest integer multiple of the coupling "//&
01870                  "timestep that is less than or equal to DT_THERM.", default=.false.)
01871
01872   if (bulkmixedlayer) then
01873     cs%Hmix = -1.0 ; cs%Hmix_UV = -1.0
01874   else
01875     call get_param(param_file, "MOM", "HMIX_SFC_PROP", cs%Hmix, &
01876                  "If BULKMIXEDLAYER is false, HMIX_SFC_PROP is the depth "//&
01877                  "over which to average to find surface properties like "//&
01878                  "SST and SSS or density (but not surface velocities).", &
01879                  units="m", default=1.0, scale=us%m_to_Z)
01880     call get_param(param_file, "MOM", "HMIX_UV_SFC_PROP", cs%Hmix_UV, &
01881                  "If BULKMIXEDLAYER is false, HMIX_UV_SFC_PROP is the depth "//&
01882                  "over which to average to find surface flow properties, "//&
01883                  "SSU, SSV. A non-positive value indicates no averaging.", &
01884                  units="m", default=0.0, scale=us%m_to_Z)
01885   endif
01886   call get_param(param_file, "MOM", "HFREEZE", cs%HFrz, &
01887                  "If HFREEZE > 0, melt potential will be computed. The actual depth "//&
01888                  "over which melt potential is computed will be min(HFREEZE, OBLD), "//&
01889                  "where OBLD is the boundary layer depth. If HFREEZE <= 0 (default), "//&
01890                  "melt potential will not be computed.", units="m", default=-1.0, scale=us%m_to_Z)
01891   call get_param(param_file, "MOM", "INTERPOLATE_P_SURF", cs%interp_p_surf, &
01892                  "If true, linearly interpolate the surface pressure "//&
01893                  "over the coupling time step, using the specified value "//&
01894                  "at the end of the step.", default=.false.)
01895
01896   if (cs%split) then
01897     call get_param(param_file, "MOM", "DTBT", dtbt, default=-0.98)
01898     default_val = us%T_to_s*cs%dt_therm ; if (dtbt > 0.0) default_val = -1.0
01899     cs%dtbt_reset_period = -1.0
01900     call get_param(param_file, "MOM", "DTBT_RESET_PERIOD", cs%dtbt_reset_period, &
01901                  "The period between recalculations of DTBT (if DTBT <= 0). "//&
01902                  "If DTBT_RESET_PERIOD is negative, DTBT is set based "//&
01903                  "only on information available at initialization.  If 0, "//&
01904                  "DTBT will be set every dynamics time step. The default "//&
01905                  "is set by DT_THERM.  This is only used if SPLIT is true.", &
```

```
01906                 units="s", default=default_val, do_not_read=(dtbt > 0.0))
01907   endif
01908
01909   ! This is here in case these values are used inappropriately.
01910   use_frazil = .false. ; bound_salinity = .false.
01911   cs%tv%P_Ref = 2.0e7*us%kg_m3_to_R*us%m_s_to_L_T**2
01912   if (use_temperature) then
01913     call get_param(param_file, "MOM", "FRAZIL", use_frazil, &
01914                  "If true, water freezes if it gets too cold, and the "//&
01915                  "accumulated heat deficit is returned in the "//&
01916                  "surface state.  FRAZIL is only used if "//&
01917                  "ENABLE_THERMODYNAMICS is true.", default=.false.)
01918     call get_param(param_file, "MOM", "DO_GEOTHERMAL", use_geothermal, &
01919                  "If true, apply geothermal heating.", default=.false.)
01920     call get_param(param_file, "MOM", "BOUND_SALINITY", bound_salinity, &
01921                  "If true, limit salinity to being positive. (The sea-ice "//&
01922                  "model may ask for more salt than is available and "//&
01923                  "drive the salinity negative otherwise.)", default=.false.)
01924     call get_param(param_file, "MOM", "MIN_SALINITY", cs%tv%min_salinity, &
01925                  "The minimum value of salinity when BOUND_SALINITY=True.", &
01926                  units="PPT", default=0.0, do_not_log=.not.bound_salinity)
01927     call get_param(param_file, "MOM", "C_P", cs%tv%C_p, &
01928                  "The heat capacity of sea water, approximated as a "//&
01929                  "constant. This is only used if ENABLE_THERMODYNAMICS is "//&
01930                  "true. The default value is from the TEOS-10 definition "//&
01931                  "of conservative temperature.", units="J kg-1 K-1", &
01932                  default=3991.86795711963, scale=us%J_kg_to_Q)
01933     call get_param(param_file, "MOM", "USE_PSURF_IN_EOS", cs%use_p_surf_in_EOS, &
01934                  "If true, always include the surface pressure contributions "//&
01935                  "in equation of state calculations.", default=.true.)
01936   endif
01937   if (use_eos) call get_param(param_file, "MOM", "P_REF", cs%tv%P_Ref, &
01938                  "The pressure that is used for calculating the coordinate "//&
01939                  "density.  (1 Pa = 1e4 dbar, so 2e7 is commonly used.) "//&
01940                  "This is only used if USE_EOS and ENABLE_THERMODYNAMICS are true.", &
01941                  units="Pa", default=2.0e7, scale=us%kg_m3_to_R*us%m_s_to_L_T**2)
01942
01943   if (bulkmixedlayer) then
01944     call get_param(param_file, "MOM", "NKML", nkml, &
01945                  "The number of sublayers within the mixed layer if "//&
01946                  "BULKMIXEDLAYER is true.", units="nondim", default=2)
01947     call get_param(param_file, "MOM", "NKBL", nkbl, &
01948                  "The number of layers that are used as variable density "//&
01949                  "buffer layers if BULKMIXEDLAYER is true.", units="nondim", &
01950                  default=2)
01951   endif
01952
01953   call get_param(param_file, "MOM", "GLOBAL_INDEXING", global_indexing, &
01954                  "If true, use a global lateral indexing convention, so "//&
01955                  "that corresponding points on different processors have "//&
01956                  "the same index. This does not work with static memory.", &
01957                  default=.false., layoutparam=.true.)
01958 #ifdef STATIC_MEMORY_
01959   if (global_indexing) call mom_error(fatal, "initialize_MOM: "//&
01960       "GLOBAL_INDEXING can not be true with STATIC_MEMORY.")
01961 #endif
01962   call get_param(param_file, "MOM", "FIRST_DIRECTION", first_direction, &
01963                  "An integer that indicates which direction goes first "//&
01964                  "in parts of the code that use directionally split "//&
01965                  "updates, with even numbers (or 0) used for x- first "//&
01966                  "and odd numbers used for y-first.", default=0)
01967
01968   call get_param(param_file, "MOM", "CHECK_BAD_SURFACE_VALS", cs%check_bad_sfc_vals, &
01969                  "If true, check the surface state for ridiculous values.", &
01970                  default=.false.)
01971   if (cs%check_bad_sfc_vals) then
01972     call get_param(param_file, "MOM", "BAD_VAL_SSH_MAX", cs%bad_val_ssh_max, &
01973                  "The value of SSH above which a bad value message is "//&
01974                  "triggered, if CHECK_BAD_SURFACE_VALS is true.", &
01975                  units="m", default=20.0, scale=us%m_to_Z)
01976     call get_param(param_file, "MOM", "BAD_VAL_SSS_MAX", cs%bad_val_sss_max, &
01977                  "The value of SSS above which a bad value message is "//&
01978                  "triggered, if CHECK_BAD_SURFACE_VALS is true.", units="PPT", &
01979                  default=45.0)
01980     call get_param(param_file, "MOM", "BAD_VAL_SST_MAX", cs%bad_val_sst_max, &
01981                  "The value of SST above which a bad value message is "//&
01982                  "triggered, if CHECK_BAD_SURFACE_VALS is true.", &
01983                  units="deg C", default=45.0)
01984     call get_param(param_file, "MOM", "BAD_VAL_SST_MIN", cs%bad_val_sst_min, &
01985                  "The value of SST below which a bad value message is "//&
01986                  "triggered, if CHECK_BAD_SURFACE_VALS is true.", &
01987                  units="deg C", default=-2.1)
01988     call get_param(param_file, "MOM", "BAD_VAL_COLUMN_THICKNESS", cs%bad_val_col_thick, &
01989                  "The value of column thickness below which a bad value message is "//&
01990                  "triggered, if CHECK_BAD_SURFACE_VALS is true.", &
01991                  units="m", default=0.0, scale=us%m_to_Z)
01992   endif
```

```
01993   call get_param(param_file, "MOM", "DEFAULT_2018_ANSWERS", default_2018_answers, &
01994                  "This sets the default value for the various _2018_ANSWERS parameters.", &
01995                  default=.false.)
01996   call get_param(param_file, "MOM", "SURFACE_2018_ANSWERS", cs%answers_2018, &
01997                  "If true, use expressions for the surface properties that recover the answers "//&
01998                  "from the end of 2018. Otherwise, use more appropriate expressions that differ "//&
01999                  "at roundoff for non-Boussinesq cases.", default=default_2018_answers)
02000
02001   call get_param(param_file, "MOM", "SAVE_INITIAL_CONDS", save_ic, &
02002                  "If true, write the initial conditions to a file given "//&
02003                  "by IC_OUTPUT_FILE.", default=.false.)
02004   call get_param(param_file, "MOM", "IC_OUTPUT_FILE", cs%IC_file, &
02005                  "The file into which to write the initial conditions.", &
02006                  default="MOM_IC")
02007   call get_param(param_file, "MOM", "WRITE_GEOM", write_geom, &
02008                  "If =0, never write the geometry and vertical grid files. "//&
02009                  "If =1, write the geometry and vertical grid files only for "//&
02010                  "a new simulation. If =2, always write the geometry and "//&
02011                  "vertical grid files. Other values are invalid.", default=1)
02012   if (write_geom<0 .or. write_geom>2) call mom_error(fatal,"MOM: "//&
02013          "WRITE_GEOM must be equal to 0, 1 or 2.")
02014   write_geom_files = ((write_geom==2) .or. ((write_geom==1) .and. &
02015      ((dirs%input_filename(1:1)=='n') .and. (len_trim(dirs%input_filename)==1))))
02016 ! If the restart file type had been initialized, this could become:
02017 !  write_geom_files = ((write_geom==2) .or. &
02018 !                     ((write_geom==1) .and. is_new_run(restart_CSp)))
02019
02020   ! Check for inconsistent parameter settings.
02021   if (cs%use_ALE_algorithm .and. bulkmixedlayer) call mom_error(fatal, &
02022     "MOM: BULKMIXEDLAYER can not currently be used with the ALE algorithm.")
02023   if (cs%use_ALE_algorithm .and. .not.use_temperature) call mom_error(fatal, &
02024      "MOM: At this time, USE_EOS should be True when using the ALE algorithm.")
02025   if (cs%adiabatic .and. use_temperature) call mom_error(warning, &
02026     "MOM: ADIABATIC and ENABLE_THERMODYNAMICS both defined is usually unwise.")
02027   if (use_eos .and. .not.use_temperature) call mom_error(fatal, &
02028     "MOM: ENABLE_THERMODYNAMICS must be defined to use USE_EOS.")
02029   if (cs%adiabatic .and. bulkmixedlayer) call mom_error(fatal, &
02030     "MOM: ADIABATIC and BULKMIXEDLAYER can not both be defined.")
02031   if (bulkmixedlayer .and. .not.use_eos) call mom_error(fatal, &
02032       "initialize_MOM: A bulk mixed layer can only be used with T & S as "//&
02033       "state variables. Add USE_EOS = True to MOM_input.")
02034
02035   call get_param(param_file, 'MOM', "ICE_SHELF", use_ice_shelf, default=.false., do_not_log=.true.)
02036   if (use_ice_shelf) then
02037     inputdir = "." ;  call get_param(param_file, 'MOM', "INPUTDIR", inputdir)
02038     inputdir = slasher(inputdir)
02039     call get_param(param_file, 'MOM', "ICE_THICKNESS_FILE", ice_shelf_file, &
02040                    "The file from which the ice bathymetry and area are read.", &
02041                    fail_if_missing=.true.)
02042     call get_param(param_file, 'MOM', "ICE_AREA_VARNAME", area_varname, &
02043                    "The name of the area variable in ICE_THICKNESS_FILE.", &
02044                    fail_if_missing=.true.)
02045   endif
02046
02047
02048   cs%ensemble_ocean=.false.
02049   call get_param(param_file, "MOM", "ENSEMBLE_OCEAN", cs%ensemble_ocean, &
02050                  "If False, The model is being run in serial mode as a single realization. "//&
02051                  "If True, The current model realization is part of a larger ensemble "//&
02052                  "and at the end of step MOM, we will perform a gather of the ensemble "//&
02053                  "members for statistical evaluation and/or data assimilation.", default=.false.)
02054
02055   call calltree_waypoint("MOM parameters read (initialize_MOM)")
02056
02057   ! Grid rotation test
02058   call get_param(param_file, "MOM", "ROTATE_INDEX", cs%rotate_index, &
02059       "Enable rotation of the horizontal indices.", default=.false., &
02060       debuggingparam=.true.)
02061   if (cs%rotate_index) then
02062     ! TODO: Index rotation currently only works when index rotation does not
02063     !   change the MPI rank of each domain.  Resolving this will require a
02064     !   modification to FMS PE assignment.
02065     !   For now, we only permit single-core runs.
02066
02067     if (num_pes() /= 1) &
02068       call mom_error(fatal, "Index rotation is only supported on one PE.")
02069
02070     call get_param(param_file, "MOM", "INDEX_TURNS", turns, &
02071         "Number of counterclockwise quarter-turn index rotations.", &
02072         default=1, debuggingparam=.true.)
02073   endif
02074
02075   ! Set up the model domain and grids.
02076 #ifdef SYMMETRIC_MEMORY_
02077   symmetric = .true.
02078 #else
02079   symmetric = .false.
```

```
02080 #endif
02081   g_in => cs%G_in
02082 #ifdef STATIC_MEMORY_
02083   call mom_domains_init(g_in%domain, param_file, symmetric=symmetric, &
02084             static_memory=.true., nihalo=nihalo_, njhalo=njhalo_, &
02085             niglobal=niglobal_, njglobal=njglobal_, niproc=niproc_, &
02086             njproc=njproc_)
02087 #else
02088   call mom_domains_init(g_in%domain, param_file, symmetric=symmetric, &
02089                         domain_name="MOM_in")
02090 #endif
02091
02092   ! Copy input grid (G_in) domain to active grid G
02093   ! Swap axes for quarter and 3-quarter turns
02094   if (cs%rotate_index) then
02095     allocate(cs%G)
02096     call clone_mom_domain(g_in%Domain, cs%G%Domain, turns=turns)
02097     first_direction = modulo(first_direction + turns, 2)
02098   else
02099     cs%G => g_in
02100   endif
02101
02102   ! TODO: It is unlikey that test_grid_copy and rotate_index would work at the
02103   !   same time.  It may be possible to enable both but for now we prevent it.
02104   if (test_grid_copy .and. cs%rotate_index) &
02105     call mom_error(fatal, "Grid cannot be copied during index rotation.")
02106
02107   if (test_grid_copy) then ; allocate(g)
02108   else ; g => cs%G ; endif
02109
02110   call calltree_waypoint("domains initialized (initialize_MOM)")
02111
02112   call mom_debugging_init(param_file)
02113   call diag_mediator_infrastructure_init()
02114   call mom_io_init(param_file)
02115
02116   ! Create HI and dG on the input index map.
02117   call hor_index_init(g_in%Domain, hi_in, param_file, &
02118                       local_indexing=.not.global_indexing)
02119   call create_dyn_horgrid(dg_in, hi_in, bathymetry_at_vel=bathy_at_vel)
02120   call clone_mom_domain(g_in%Domain, dg_in%Domain)
02121
02122   ! Allocate initialize time-invariant MOM variables.
02123   call mom_initialize_fixed(dg_in, us, obc_in, param_file, write_geom_files, &
02124                             dirs%output_directory)
02125
02126   call calltree_waypoint("returned from MOM_initialize_fixed() (initialize_MOM)")
02127
02128   ! Determine HI and dG for the model index map.
02129   if (cs%rotate_index) then
02130     allocate(hi)
02131     call rotate_hor_index(hi_in, turns, hi)
02132     call create_dyn_horgrid(dg, hi, bathymetry_at_vel=bathy_at_vel)
02133     call clone_mom_domain(g%Domain, dg%Domain)
02134     call rotate_dyngrid(dg_in, dg, us, turns)
02135     if (associated(obc_in)) then
02136       ! TODO: General OBC index rotations is not yet supported.
02137       if (modulo(turns, 4) /= 1) &
02138         call mom_error(fatal, "OBC index rotation of 180 and 270 degrees is " &
02139           // "not yet unsupported.")
02140       allocate(cs%OBC)
02141       call rotate_obc_config(obc_in, dg_in, cs%OBC, dg, turns)
02142     endif
02143   else
02144     hi => hi_in
02145     dg => dg_in
02146     cs%OBC => obc_in
02147   endif
02148
02149   call verticalgridinit( param_file, cs%GV, us )
02150   gv => cs%GV
02151
02152   ! Allocate the auxiliary non-symmetric domain for debugging or I/O purposes.
02153   if (cs%debug .or. dg%symmetric) &
02154     call clone_mom_domain(dg%Domain, dg%Domain_aux, symmetric=.false.)
02155
02156   call calltree_waypoint("grids initialized (initialize_MOM)")
02157
02158   call mom_timing_init(cs)
02159
02160   if (associated(cs%OBC)) call call_obc_register(param_file, cs%update_OBC_CSp, cs%OBC)
02161
02162   call tracer_registry_init(param_file, cs%tracer_Reg)
02163
02164   ! Allocate and initialize space for the primary time-varying MOM variables.
02165   is   = dg%isc   ; ie   = dg%iec  ; js   = dg%jsc  ; je   = dg%jec ; nz = gv%ke
02166   isd  = dg%isd   ; ied  = dg%ied  ; jsd  = dg%jsd  ; jed  = dg%jed
```

```
02167   isdb = dg%IsdB  ; iedb = dg%IedB ; jsdb = dg%JsdB ; jedb = dg%JedB
02168   alloc_(cs%u(isdb:iedb,jsd:jed,nz))   ; cs%u(:,:,:) = 0.0
02169   alloc_(cs%v(isd:ied,jsdb:jedb,nz))   ; cs%v(:,:,:) = 0.0
02170   alloc_(cs%h(isd:ied,jsd:jed,nz))     ; cs%h(:,:,:) = gv%Angstrom_H
02171   alloc_(cs%uh(isdb:iedb,jsd:jed,nz))  ; cs%uh(:,:,:) = 0.0
02172   alloc_(cs%vh(isd:ied,jsdb:jedb,nz))  ; cs%vh(:,:,:) = 0.0
02173   if (use_temperature) then
02174     alloc_(cs%T(isd:ied,jsd:jed,nz))   ; cs%T(:,:,:) = 0.0
02175     alloc_(cs%S(isd:ied,jsd:jed,nz))   ; cs%S(:,:,:) = 0.0
02176     cs%tv%T => cs%T ; cs%tv%S => cs%S
02177     if (cs%tv%T_is_conT) then
02178       vd_t = var_desc(name="contemp", units="Celsius", longname="Conservative Temperature", &
02179                       cmor_field_name="thetao", cmor_longname="Sea Water Potential Temperature", &
02180                       conversion=us%Q_to_J_kg*cs%tv%C_p)
02181     else
02182       vd_t = var_desc(name="temp", units="degC", longname="Potential Temperature", &
02183                       cmor_field_name="thetao", cmor_longname="Sea Water Potential Temperature", &
02184                       conversion=us%Q_to_J_kg*cs%tv%C_p)
02185     endif
02186     if (cs%tv%S_is_absS) then
02187       vd_s = var_desc(name="abssalt",units="g kg-1",longname="Absolute Salinity", &
02188                       cmor_field_name="so", cmor_longname="Sea Water Salinity", &
02189                       conversion=0.001)
02190     else
02191       vd_s = var_desc(name="salt",units="psu",longname="Salinity", &
02192                       cmor_field_name="so", cmor_longname="Sea Water Salinity", &
02193                       conversion=0.001)
02194     endif
02195
02196     if (advect_ts) then
02197       s_flux_units = get_tr_flux_units(gv, "psu") ! Could change to "kg m-2 s-1"?
02198       conv2watt    = gv%H_to_kg_m2 * us%Q_to_J_kg*cs%tv%C_p
02199       if (gv%Boussinesq) then
02200         conv2salt = gv%H_to_m ! Could change to GV%H_to_kg_m2 * 0.001?
02201       else
02202         conv2salt = gv%H_to_kg_m2
02203       endif
02204       call register_tracer(cs%tv%T, cs%tracer_Reg, param_file, dg%HI, gv, &
02205                            tr_desc=vd_t, registry_diags=.true., flux_nameroot='T', &
02206                            flux_units='W', flux_longname='Heat', &
02207                            flux_scale=conv2watt, convergence_units='W m-2', &
02208                            convergence_scale=conv2watt, cmor_tendprefix="opottemp", diag_form=2)
02209       call register_tracer(cs%tv%S, cs%tracer_Reg, param_file, dg%HI, gv, &
02210                            tr_desc=vd_s, registry_diags=.true., flux_nameroot='S', &
02211                            flux_units=s_flux_units, flux_longname='Salt', &
02212                            flux_scale=conv2salt, convergence_units='kg m-2 s-1', &
02213                            convergence_scale=0.001*gv%H_to_kg_m2, cmor_tendprefix="osalt", diag_form=2)
02214     endif
02215     ! NOTE: register_temp_salt_segments includes allocation of tracer fields
02216     !   along segments.  Bit reproducibility requires that MOM_initialize_state
02217     !   be called on the input index map, so we must setup both OBC and OBC_in.
02218     !
02219     ! XXX: This call on OBC_in allocates the tracer fields on the unrotated
02220     !   grid, but also incorrectly stores a pointer to a tracer_type for the
02221     !   rotated registry (e.g. segment%tr_reg%Tr(n)%Tr) from CS%tracer_reg.
02222     !
02223     !   While incorrect and potentially dangerous, it does not seem that this
02224     !   pointer is used during initialization, so we leave it for now.
02225     if (cs%rotate_index .and. associated(obc_in)) &
02226       call register_temp_salt_segments(gv, obc_in, cs%tracer_Reg, param_file)
02227     if (associated(cs%OBC)) &
02228       call register_temp_salt_segments(gv, cs%OBC, cs%tracer_Reg, param_file)
02229   endif
02230   if (use_frazil) then
02231     allocate(cs%tv%frazil(isd:ied,jsd:jed)) ; cs%tv%frazil(:,:) = 0.0
02232   endif
02233   if (bound_salinity) then
02234     allocate(cs%tv%salt_deficit(isd:ied,jsd:jed)) ; cs%tv%salt_deficit(:,:) = 0.0
02235   endif
02236
02237   if (bulkmixedlayer .or. use_temperature) then
02238     allocate(cs%Hml(isd:ied,jsd:jed)) ; cs%Hml(:,:) = 0.0
02239   endif
02240
02241   if (bulkmixedlayer) then
02242     gv%nkml = nkml ; gv%nk_rho_varies = nkml + nkbl
02243   else
02244     gv%nkml = 0 ; gv%nk_rho_varies = 0
02245   endif
02246   if (cs%use_ALE_algorithm) then
02247     call get_param(param_file, "MOM", "NK_RHO_VARIES", gv%nk_rho_varies, default=0) ! Will default to nz
      later... -AJA
02248   endif
02249
02250   alloc_(cs%uhtr(isdb:iedb,jsd:jed,nz)) ; cs%uhtr(:,:,:) = 0.0
02251   alloc_(cs%vhtr(isd:ied,jsdb:jedb,nz)) ; cs%vhtr(:,:,:) = 0.0
02252   cs%t_dyn_rel_adv = 0.0 ; cs%t_dyn_rel_thermo = 0.0 ; cs%t_dyn_rel_diag = 0.0
```

```
02253
02254   if (debug_truncations) then
02255     allocate(cs%u_prev(isdb:iedb,jsd:jed,nz)) ; cs%u_prev(:,:,:) = 0.0
02256     allocate(cs%v_prev(isd:ied,jsdb:jedb,nz)) ; cs%v_prev(:,:,:) = 0.0
02257     mom_internal_state%u_prev => cs%u_prev
02258     mom_internal_state%v_prev => cs%v_prev
02259     call safe_alloc_ptr(cs%ADp%du_dt_visc,isdb,iedb,jsd,jed,nz)
02260     call safe_alloc_ptr(cs%ADp%dv_dt_visc,isd,ied,jsdb,jedb,nz)
02261     if (.not.cs%adiabatic) then
02262       call safe_alloc_ptr(cs%ADp%du_dt_dia,isdb,iedb,jsd,jed,nz)
02263       call safe_alloc_ptr(cs%ADp%dv_dt_dia,isd,ied,jsdb,jedb,nz)
02264     endif
02265   endif
02266
02267   mom_internal_state%u => cs%u ; mom_internal_state%v => cs%v
02268   mom_internal_state%h => cs%h
02269   mom_internal_state%uh => cs%uh ; mom_internal_state%vh => cs%vh
02270   if (use_temperature) then
02271     mom_internal_state%T => cs%T ; mom_internal_state%S => cs%S
02272   endif
02273
02274   cs%CDp%uh => cs%uh ; cs%CDp%vh => cs%vh
02275
02276   if (cs%interp_p_surf) then
02277     allocate(cs%p_surf_prev(isd:ied,jsd:jed)) ; cs%p_surf_prev(:,:) = 0.0
02278   endif
02279
02280   alloc_(cs%ssh_rint(isd:ied,jsd:jed)) ; cs%ssh_rint(:,:) = 0.0
02281   alloc_(cs%ave_ssh_ibc(isd:ied,jsd:jed)) ; cs%ave_ssh_ibc(:,:) = 0.0
02282   alloc_(cs%eta_av_bc(isd:ied,jsd:jed)) ; cs%eta_av_bc(:,:) = 0.0
02283   cs%time_in_cycle = 0.0 ; cs%time_in_thermo_cycle = 0.0
02284
02285   ! Use the Wright equation of state by default, unless otherwise specified
02286   ! Note: this line and the following block ought to be in a separate
02287   ! initialization routine for tv.
02288   if (use_eos) call eos_init(param_file, cs%tv%eqn_of_state, us)
02289   if (use_temperature) then
02290     allocate(cs%tv%TempxPmE(isd:ied,jsd:jed)) ; cs%tv%TempxPmE(:,:) = 0.0
02291     if (use_geothermal) then
02292       allocate(cs%tv%internal_heat(isd:ied,jsd:jed)) ; cs%tv%internal_heat(:,:) = 0.0
02293     endif
02294   endif
02295   call calltree_waypoint("state variables allocated (initialize_MOM)")
02296
02297   ! Set the fields that are needed for bitwise identical restarting
02298   ! the time stepping scheme.
02299   call restart_init(param_file, restart_csp)
02300   call set_restart_fields(gv, us, param_file, cs, restart_csp)
02301   if (cs%split) then
02302     call register_restarts_dyn_split_rk2(dg%HI, gv, param_file, &
02303              cs%dyn_split_RK2_CSp, restart_csp, cs%uh, cs%vh)
02304   elseif (cs%use_RK2) then
02305     call register_restarts_dyn_unsplit_rk2(dg%HI, gv, param_file, &
02306            cs%dyn_unsplit_RK2_CSp, restart_csp)
02307   else
02308     call register_restarts_dyn_unsplit(dg%HI, gv, param_file, &
02309            cs%dyn_unsplit_CSp, restart_csp)
02310   endif
02311
02312   ! This subroutine calls user-specified tracer registration routines.
02313   ! Additional calls can be added to MOM_tracer_flow_control.F90.
02314   call call_tracer_register(dg%HI, gv, us, param_file, cs%tracer_flow_CSp, &
02315                             cs%tracer_Reg, restart_csp)
02316
02317   call meke_alloc_register_restart(dg%HI, param_file, cs%MEKE, restart_csp)
02318   call set_visc_register_restarts(dg%HI, gv, param_file, cs%visc, restart_csp)
02319   call mixedlayer_restrat_register_restarts(dg%HI, param_file, &
02320            cs%mixedlayer_restrat_CSp, restart_csp)
02321
02322   if (associated(cs%OBC)) &
02323     call open_boundary_register_restarts(dg%HI, gv, cs%OBC, cs%tracer_Reg, &
02324                           param_file, restart_csp, use_temperature)
02325
02326   call calltree_waypoint("restart registration complete (initialize_MOM)")
02327
02328   ! Initialize dynamically evolving fields, perhaps from restart files.
02329   call cpu_clock_begin(id_clock_mom_init)
02330   call mom_initialize_coord(gv, us, param_file, write_geom_files, &
02331                             dirs%output_directory, cs%tv, dg%max_depth)
02332   call calltree_waypoint("returned from MOM_initialize_coord() (initialize_MOM)")
02333
02334   if (cs%use_ALE_algorithm) then
02335     call ale_init(param_file, gv, us, dg%max_depth, cs%ALE_CSp)
02336     call calltree_waypoint("returned from ALE_init() (initialize_MOM)")
02337   endif
02338
02339   !   Shift from using the temporary dynamic grid type to using the final
```

```
02340   ! (potentially static) ocean-specific grid type.
02341   !   The next line would be needed if G%Domain had not already been init'd above:
02342   !     call clone_MOM_domain(dG%Domain, G%Domain)
02343
02344   ! NOTE: If indices are rotated, then G and G_in must both be initialized.
02345   !   If not rotated, then G_in and G are the same grid.
02346   if (cs%rotate_index) then
02347     call mom_grid_init(g, param_file, us, hi, bathymetry_at_vel=bathy_at_vel)
02348     call copy_dyngrid_to_mom_grid(dg, g, us)
02349     call destroy_dyn_horgrid(dg)
02350   endif
02351   call mom_grid_init(g_in, param_file, us, hi_in, bathymetry_at_vel=bathy_at_vel)
02352   call copy_dyngrid_to_mom_grid(dg_in, g_in, us)
02353   call destroy_dyn_horgrid(dg_in)
02354
02355   ! Set a few remaining fields that are specific to the ocean grid type.
02356   call set_first_direction(g, first_direction)
02357   ! Allocate the auxiliary non-symmetric domain for debugging or I/O purposes.
02358   if (cs%debug .or. g%symmetric) then
02359     call clone_mom_domain(g%Domain, g%Domain_aux, symmetric=.false.)
02360   else ; g%Domain_aux => g%Domain ; endif
02361   ! Copy common variables from the vertical grid to the horizontal grid.
02362   ! Consider removing this later?
02363   g%ke = gv%ke
02364
02365   if (cs%rotate_index) then
02366     g_in%ke = gv%ke
02367
02368     allocate(u_in(g_in%IsdB:g_in%IedB, g_in%jsd:g_in%jed, nz))
02369     allocate(v_in(g_in%isd:g_in%ied, g_in%JsdB:g_in%JedB, nz))
02370     allocate(h_in(g_in%isd:g_in%ied, g_in%jsd:g_in%jed, nz))
02371     u_in(:,:,:) = 0.0
02372     v_in(:,:,:) = 0.0
02373     h_in(:,:,:) = gv%Angstrom_H
02374
02375     if (use_temperature) then
02376       allocate(t_in(g_in%isd:g_in%ied, g_in%jsd:g_in%jed, nz))
02377       allocate(s_in(g_in%isd:g_in%ied, g_in%jsd:g_in%jed, nz))
02378       t_in(:,:,:) = 0.0
02379       s_in(:,:,:) = 0.0
02380
02381       cs%tv%T => t_in
02382       cs%tv%S => s_in
02383     endif
02384
02385     call mom_initialize_state(u_in, v_in, h_in, cs%tv, time, g_in, gv, us, &
02386         param_file, dirs, restart_csp, cs%ALE_CSp, cs%tracer_Reg, &
02387         sponge_in_csp, ale_sponge_in_csp, obc_in, time_in)
02388
02389     if (use_temperature) then
02390       cs%tv%T => cs%T
02391       cs%tv%S => cs%S
02392     endif
02393
02394     call rotate_initial_state(u_in, v_in, h_in, t_in, s_in, use_temperature, &
02395         turns, cs%u, cs%v, cs%h, cs%T, cs%S)
02396
02397     if (associated(sponge_in_csp)) then
02398       ! TODO: Implementation and testing of non-ALE spong rotation
02399       call mom_error(fatal, "Index rotation of non-ALE sponge is not yet implemented.")
02400     endif
02401
02402     if (associated(ale_sponge_in_csp)) then
02403       call rotate_ale_sponge(ale_sponge_in_csp, g_in, cs%ALE_sponge_CSp, g, turns, param_file)
02404       call update_ale_sponge_field(cs%ALE_sponge_CSp, t_in, g, gv, cs%T)
02405       call update_ale_sponge_field(cs%ALE_sponge_CSp, s_in, g, gv, cs%S)
02406     endif
02407
02408     if (associated(obc_in)) &
02409       call rotate_obc_init(obc_in, g, gv, us, param_file, cs%tv, restart_csp, cs%OBC)
02410
02411     deallocate(u_in)
02412     deallocate(v_in)
02413     deallocate(h_in)
02414     if (use_temperature) then
02415       deallocate(t_in)
02416       deallocate(s_in)
02417     endif
02418   else
02419     call mom_initialize_state(cs%u, cs%v, cs%h, cs%tv, time, g, gv, us, &
02420         param_file, dirs, restart_csp, cs%ALE_CSp, cs%tracer_Reg, &
02421         cs%sponge_CSp, cs%ALE_sponge_CSp, cs%OBC, time_in)
02422   endif
02423
02424   call cpu_clock_end(id_clock_mom_init)
02425   call calltree_waypoint("returned from MOM_initialize_state() (initialize_MOM)")
02426
```

```
02427 ! ! Need this after MOM_initialize_state for DOME OBC stuff.
02428 ! if (associated(CS%OBC)) &
02429 !   call open_boundary_register_restarts(G%HI, GV, CS%OBC, CS%tracer_Reg, &
02430 !                         param_file, restart_CSp, use_temperature)
02431
02432 ! call callTree_waypoint("restart registration complete (initialize_MOM)")
02433
02434   ! From this point, there may be pointers being set, so the final grid type
02435   ! that will persist throughout the run has to be used.
02436
02437   if (test_grid_copy) then
02438     !  Copy the data from the temporary grid to the dyn_hor_grid to CS%G.
02439     call create_dyn_horgrid(dg, g%HI)
02440     call clone_mom_domain(g%Domain, dg%Domain)
02441
02442     call clone_mom_domain(g%Domain, cs%G%Domain)
02443     call mom_grid_init(cs%G, param_file, us)
02444
02445     call copy_mom_grid_to_dyngrid(g, dg, us)
02446     call copy_dyngrid_to_mom_grid(dg, cs%G, us)
02447
02448     call destroy_dyn_horgrid(dg)
02449     call mom_grid_end(g) ; deallocate(g)
02450
02451     g => cs%G
02452     if (cs%debug .or. cs%G%symmetric) then
02453       call clone_mom_domain(cs%G%Domain, cs%G%Domain_aux, symmetric=.false.)
02454     else ; cs%G%Domain_aux => cs%G%Domain ; endif
02455     g%ke = gv%ke
02456   endif
02457
02458   ! At this point, all user-modified initialization code has been called.  The
02459   ! remainder of this subroutine is controlled by the parameters that have
02460   ! have already been set.
02461
02462   if (ale_remap_init_conds(cs%ALE_CSp) .and. .not. query_initialized(cs%h,"h",restart_csp)) then
02463     ! This block is controlled by the ALE parameter REMAP_AFTER_INITIALIZATION.
02464     ! \todo This block exists for legacy reasons and we should phase it out of
02465     ! all examples. !###
02466     if (cs%debug) then
02467       call uvchksum("Pre ALE adjust init cond [uv]", &
02468                     cs%u, cs%v, g%HI, haloshift=1)
02469       call hchksum(cs%h,"Pre ALE adjust init cond h", g%HI, haloshift=1, scale=gv%H_to_m)
02470     endif
02471     call calltree_waypoint("Calling adjustGridForIntegrity() to remap initial conditions (initialize_MOM)")
02472     call adjustgridforintegrity(cs%ALE_CSp, g, gv, cs%h )
02473     call calltree_waypoint("Calling ALE_main() to remap initial conditions (initialize_MOM)")
02474     if (use_ice_shelf) then
02475       filename = trim(inputdir)//trim(ice_shelf_file)
02476       if (.not.file_exists(filename, g%Domain)) call mom_error(fatal, &
02477         "MOM: Unable to open "//trim(filename))
02478
02479       allocate(area_shelf_h(isd:ied,jsd:jed))
02480       allocate(frac_shelf_h(isd:ied,jsd:jed))
02481       call mom_read_data(filename, trim(area_varname), area_shelf_h, g%Domain, scale=us%m_to_L**2)
02482       ! initialize frac_shelf_h with zeros (open water everywhere)
02483       frac_shelf_h(:,:) = 0.0
02484       ! compute fractional ice shelf coverage of h
02485       do j=jsd,jed ; do i=isd,ied
02486         if (g%areaT(i,j) > 0.0) &
02487           frac_shelf_h(i,j) = area_shelf_h(i,j) / g%areaT(i,j)
02488       enddo ; enddo
02489       ! pass to the pointer
02490       shelf_area => frac_shelf_h
02491       call ale_main(g, gv, us, cs%h, cs%u, cs%v, cs%tv, cs%tracer_Reg, cs%ALE_CSp, &
02492                     cs%OBC, frac_shelf_h=shelf_area)
02493     else
02494       call ale_main( g, gv, us, cs%h, cs%u, cs%v, cs%tv, cs%tracer_Reg, cs%ALE_CSp, cs%OBC)
02495     endif
02496
02497     call cpu_clock_begin(id_clock_pass_init)
02498     call create_group_pass(tmp_pass_uv_t_s_h, cs%u, cs%v, g%Domain)
02499     if (use_temperature) then
02500       call create_group_pass(tmp_pass_uv_t_s_h, cs%tv%T, g%Domain, halo=1)
02501       call create_group_pass(tmp_pass_uv_t_s_h, cs%tv%S, g%Domain, halo=1)
02502     endif
02503     call create_group_pass(tmp_pass_uv_t_s_h, cs%h, g%Domain, halo=1)
02504     call do_group_pass(tmp_pass_uv_t_s_h, g%Domain)
02505     call cpu_clock_end(id_clock_pass_init)
02506
02507     if (cs%debug) then
02508       call uvchksum("Post ALE adjust init cond [uv]", cs%u, cs%v, g%HI, haloshift=1)
02509       call hchksum(cs%h, "Post ALE adjust init cond h", g%HI, haloshift=1, scale=gv%H_to_m)
02510     endif
02511   endif
02512   if ( cs%use_ALE_algorithm ) call ale_updateverticalgridtype( cs%ALE_CSp, gv )
02513
```

```
02514   diag => cs%diag
02515   ! Initialize the diag mediator.
02516   call diag_mediator_init(g, gv, us, gv%ke, param_file, diag, doc_file_dir=dirs%output_directory)
02517   if (present(diag_ptr)) diag_ptr => cs%diag
02518
02519   ! Initialize the diagnostics masks for native arrays.
02520   ! This step has to be done after call to MOM_initialize_state
02521   ! and before MOM_diagnostics_init
02522   call diag_masks_set(g, gv%ke, diag)
02523
02524   ! Set up pointers within diag mediator control structure,
02525   ! this needs to occur _after_ CS%h etc. have been allocated.
02526   call diag_set_state_ptrs(cs%h, cs%T, cs%S, cs%tv%eqn_of_state, diag)
02527
02528   ! This call sets up the diagnostic axes. These are needed,
02529   ! e.g. to generate the target grids below.
02530   call set_axes_info(g, gv, us, param_file, diag)
02531
02532   ! Whenever thickness/T/S changes let the diag manager know, target grids
02533   ! for vertical remapping may need to be regenerated.
02534   ! FIXME: are h, T, S updated at the same time? Review these for T, S updates.
02535   call diag_update_remap_grids(diag)
02536
02537   ! Setup the diagnostic grid storage types
02538   call diag_grid_storage_init(cs%diag_pre_sync, g, diag)
02539   call diag_grid_storage_init(cs%diag_pre_dyn, g, diag)
02540
02541   ! Calculate masks for diagnostics arrays in non-native coordinates
02542   ! This step has to be done after set_axes_info() because the axes needed
02543   ! to be configured, and after diag_update_remap_grids() because the grids
02544   ! must be defined.
02545   call set_masks_for_axes(g, diag)
02546
02547   ! Diagnose static fields AND associate areas/volumes with axes
02548   call write_static_fields(g, gv, us, cs%tv, cs%diag)
02549   call calltree_waypoint("static fields written (initialize_MOM)")
02550
02551   ! Register the volume cell measure (must be one of first diagnostics)
02552   call register_cell_measure(g, cs%diag, time)
02553
02554   call cpu_clock_begin(id_clock_mom_init)
02555   if (cs%use_ALE_algorithm) then
02556     call ale_writecoordinatefile( cs%ALE_CSp, gv, dirs%output_directory )
02557   endif
02558   call cpu_clock_end(id_clock_mom_init)
02559   call calltree_waypoint("ALE initialized (initialize_MOM)")
02560
02561   cs%useMEKE = meke_init(time, g, us, param_file, diag, cs%MEKE_CSp, cs%MEKE, restart_csp)
02562
02563   call varmix_init(time, g, gv, us, param_file, diag, cs%VarMix)
02564   call set_visc_init(time, g, gv, us, param_file, diag, cs%visc, cs%set_visc_CSp, restart_csp, cs%OBC)
02565   call thickness_diffuse_init(time, g, gv, us, param_file, diag, cs%CDp, cs%thickness_diffuse_CSp)
02566
02567   if (cs%split) then
02568     allocate(eta(szi_(g),szj_(g))) ; eta(:,:) = 0.0
02569     call initialize_dyn_split_rk2(cs%u, cs%v, cs%h, cs%uh, cs%vh, eta, time, &
02570               g, gv, us, param_file, diag, cs%dyn_split_RK2_CSp, restart_csp, &
02571               cs%dt, cs%ADp, cs%CDp, mom_internal_state, cs%VarMix, cs%MEKE, &
02572               cs%thickness_diffuse_CSp,                                      &
02573               cs%OBC, cs%update_OBC_CSp, cs%ALE_CSp, cs%set_visc_CSp,        &
02574               cs%visc, dirs, cs%ntrunc, calc_dtbt=calc_dtbt, cont_stencil=cs%cont_stencil)
02575     if (cs%dtbt_reset_period > 0.0) then
02576       cs%dtbt_reset_interval = real_to_time(cs%dtbt_reset_period)
02577       ! Set dtbt_reset_time to be the next even multiple of dtbt_reset_interval.
02578       cs%dtbt_reset_time = time_init + cs%dtbt_reset_interval * &
02579                                ((time - time_init) / cs%dtbt_reset_interval)
02580       if ((cs%dtbt_reset_time > time) .and. calc_dtbt) then
02581         ! Back up dtbt_reset_time one interval to force dtbt to be calculated,
02582         ! because the restart was not aligned with the interval to recalculate
02583         ! dtbt, and dtbt was not read from a restart file.
02584         cs%dtbt_reset_time = cs%dtbt_reset_time - cs%dtbt_reset_interval
02585       endif
02586     endif
02587   elseif (cs%use_RK2) then
02588     call initialize_dyn_unsplit_rk2(cs%u, cs%v, cs%h, time, g, gv, us,     &
02589             param_file, diag, cs%dyn_unsplit_RK2_CSp, restart_csp,         &
02590             cs%ADp, cs%CDp, mom_internal_state, cs%MEKE, cs%OBC,           &
02591             cs%update_OBC_CSp, cs%ALE_CSp, cs%set_visc_CSp, cs%visc, dirs, &
02592             cs%ntrunc, cont_stencil=cs%cont_stencil)
02593   else
02594     call initialize_dyn_unsplit(cs%u, cs%v, cs%h, time, g, gv, us,         &
02595             param_file, diag, cs%dyn_unsplit_CSp, restart_csp,             &
02596             cs%ADp, cs%CDp, mom_internal_state, cs%MEKE, cs%OBC,           &
02597             cs%update_OBC_CSp, cs%ALE_CSp, cs%set_visc_CSp, cs%visc, dirs, &
02598             cs%ntrunc, cont_stencil=cs%cont_stencil)
02599   endif
02600
```

```
02601   call calltree_waypoint("dynamics initialized (initialize_MOM)")
02602
02603   cs%mixedlayer_restrat = mixedlayer_restrat_init(time, g, gv, us, param_file, diag, &
02604                                                   cs%mixedlayer_restrat_CSp, restart_csp)
02605   if (cs%mixedlayer_restrat) then
02606     if (.not.(bulkmixedlayer .or. cs%use_ALE_algorithm)) &
02607       call mom_error(fatal, "MOM: MIXEDLAYER_RESTRAT true requires a boundary layer scheme.")
02608     ! When DIABATIC_FIRST=False and using CS%visc%ML in mixedlayer_restrat we need to update after a
      restart
02609     if (.not. cs%diabatic_first .and. associated(cs%visc%MLD)) &
02610       call pass_var(cs%visc%MLD, g%domain, halo=1)
02611   endif
02612
02613   call mom_diagnostics_init(mom_internal_state, cs%ADp, cs%CDp, time, g, gv, us, &
02614                             param_file, diag, cs%diagnostics_CSp, cs%tv)
02615   call diag_copy_diag_to_storage(cs%diag_pre_sync, cs%h, cs%diag)
02616
02617   if (associated(cs%sponge_CSp)) &
02618     call init_sponge_diags(time, g, gv, us, diag, cs%sponge_CSp)
02619
02620   if (associated(cs%ALE_sponge_CSp)) &
02621     call init_ale_sponge_diags(time, g, diag, cs%ALE_sponge_CSp)
02622
02623   if (cs%adiabatic) then
02624     call adiabatic_driver_init(time, g, param_file, diag, cs%diabatic_CSp, &
02625                                cs%tracer_flow_CSp)
02626   else
02627     call diabatic_driver_init(time, g, gv, us, param_file, cs%use_ALE_algorithm, diag, &
02628                               cs%ADp, cs%CDp, cs%diabatic_CSp, cs%tracer_flow_CSp, &
02629                               cs%sponge_CSp, cs%ALE_sponge_CSp)
02630   endif
02631
02632   call tracer_advect_init(time, g, us, param_file, diag, cs%tracer_adv_CSp)
02633   call tracer_hor_diff_init(time, g, us, param_file, diag, cs%tv%eqn_of_state, cs%diabatic_CSp, &
02634                             cs%tracer_diff_CSp)
02635
02636   call lock_tracer_registry(cs%tracer_Reg)
02637   call calltree_waypoint("tracer registry now locked (initialize_MOM)")
02638
02639   ! now register some diagnostics since the tracer registry is now locked
02640   call register_surface_diags(time, g, us, cs%sfc_IDs, cs%diag, cs%tv)
02641   call register_diags(time, g, gv, us, cs%IDs, cs%diag)
02642   call register_transport_diags(time, g, gv, us, cs%transport_IDs, cs%diag)
02643   call register_tracer_diagnostics(cs%tracer_Reg, cs%h, time, diag, g, gv, us, &
02644                                    cs%use_ALE_algorithm)
02645   if (cs%use_ALE_algorithm) then
02646     call ale_register_diags(time, g, gv, us, diag, cs%ALE_CSp)
02647   endif
02648
02649   ! This subroutine initializes any tracer packages.
02650   new_sim = is_new_run(restart_csp)
02651   call tracer_flow_control_init(.not.new_sim, time, g, gv, us, cs%h, param_file, &
02652             cs%diag, cs%OBC, cs%tracer_flow_CSp, cs%sponge_CSp, &
02653             cs%ALE_sponge_CSp, cs%tv)
02654   if (present(tracer_flow_csp)) tracer_flow_csp => cs%tracer_flow_CSp
02655
02656   ! If running in offline tracer mode, initialize the necessary control structure and
02657   ! parameters
02658   if (present(offline_tracer_mode)) offline_tracer_mode=cs%offline_tracer_mode
02659
02660   if (cs%offline_tracer_mode) then
02661     ! Setup some initial parameterizations and also assign some of the subtypes
02662     call offline_transport_init(param_file, cs%offline_CSp, cs%diabatic_CSp, g, gv, us)
02663     call insert_offline_main( cs=cs%offline_CSp, ale_csp=cs%ALE_CSp, diabatic_csp=cs%diabatic_CSp, &
02664                               diag=cs%diag, obc=cs%OBC, tracer_adv_csp=cs%tracer_adv_CSp,               &
02665                               tracer_flow_csp=cs%tracer_flow_CSp, tracer_reg=cs%tracer_Reg,             &
02666                               tv=cs%tv, x_before_y = (mod(first_direction,2)==0), debug=cs%debug )
02667     call register_diags_offline_transport(time, cs%diag, cs%offline_CSp)
02668   endif
02669
02670   !--- set up group pass for u,v,T,S and h. pass_uv_T_S_h also is used in step_MOM
02671   call cpu_clock_begin(id_clock_pass_init)
02672   dynamics_stencil = min(3, g%Domain%nihalo, g%Domain%njhalo)
02673   call create_group_pass(pass_uv_t_s_h, cs%u, cs%v, g%Domain, halo=dynamics_stencil)
02674   if (use_temperature) then
02675     call create_group_pass(pass_uv_t_s_h, cs%tv%T, g%Domain, halo=dynamics_stencil)
02676     call create_group_pass(pass_uv_t_s_h, cs%tv%S, g%Domain, halo=dynamics_stencil)
02677   endif
02678   call create_group_pass(pass_uv_t_s_h, cs%h, g%Domain, halo=dynamics_stencil)
02679
02680   call do_group_pass(pass_uv_t_s_h, g%Domain)
02681
02682   if (associated(cs%visc%Kv_shear)) &
02683     call pass_var(cs%visc%Kv_shear, g%Domain, to_all+omit_corners, halo=1)
02684
02685   if (associated(cs%visc%Kv_slow)) &
02686     call pass_var(cs%visc%Kv_slow, g%Domain, to_all+omit_corners, halo=1)
```

```
02687
02688   call cpu_clock_end(id_clock_pass_init)
02689
02690   call register_obsolete_diagnostics(param_file, cs%diag)
02691
02692   if (use_frazil) then
02693     if (query_initialized(cs%tv%frazil, "frazil", restart_csp)) then
02694       ! Test whether the dimensional rescaling has changed for heat content.
02695       if ((us%kg_m3_to_R_restart*us%m_to_Z_restart*us%J_kg_to_Q_restart /= 0.0) .and. &
02696           ((us%J_kg_to_Q*us%kg_m3_to_R*us%m_to_Z) /= &
02697            (us%J_kg_to_Q_restart*us%kg_m3_to_R_restart*us%m_to_Z_restart)) ) then
02698         qrz_rescale = (us%J_kg_to_Q*us%kg_m3_to_R*us%m_to_Z) / &
02699                       (us%J_kg_to_Q_restart*us%kg_m3_to_R_restart*us%m_to_Z_restart)
02700         do j=js,je ; do i=is,ie
02701           cs%tv%frazil(i,j) = qrz_rescale * cs%tv%frazil(i,j)
02702         enddo ; enddo
02703       endif
02704     else
02705       cs%tv%frazil(:,:) = 0.0
02706     endif
02707   endif
02708
02709   if (cs%interp_p_surf) then
02710     cs%p_surf_prev_set = query_initialized(cs%p_surf_prev,"p_surf_prev",restart_csp)
02711
02712     if (cs%p_surf_prev_set) then
02713       ! Test whether the dimensional rescaling has changed for pressure.
02714       if ((us%kg_m3_to_R_restart*us%s_to_T_restart*us%m_to_L_restart /= 0.0) .and. &
02715           ((us%kg_m3_to_R*(us%m_to_L*us%s_to_T_restart)**2) /= &
02716            (us%kg_m3_to_R_restart*(us%m_to_L_restart*us%s_to_T)**2)) ) then
02717         rl2_t2_rescale = (us%kg_m3_to_R*(us%m_to_L*us%s_to_T_restart)**2) / &
02718                          (us%kg_m3_to_R_restart*(us%m_to_L_restart*us%s_to_T)**2)
02719         do j=js,je ; do i=is,ie
02720           cs%p_surf_prev(i,j) = rl2_t2_rescale * cs%p_surf_prev(i,j)
02721         enddo ; enddo
02722       endif
02723
02724       call pass_var(cs%p_surf_prev, g%domain)
02725     endif
02726   endif
02727
02728   if (use_ice_shelf .and. associated(cs%Hml)) then
02729     if (query_initialized(cs%Hml, "hML", restart_csp)) then
02730       ! Test whether the dimensional rescaling has changed for depths.
02731       if ((us%m_to_Z_restart /= 0.0) .and. (us%m_to_Z /= us%m_to_Z_restart) ) then
02732         z_rescale = us%m_to_Z / us%m_to_Z_restart
02733         do j=js,je ; do i=is,ie
02734           cs%Hml(i,j) = z_rescale * cs%Hml(i,j)
02735         enddo ; enddo
02736       endif
02737     endif
02738   endif
02739
02740   if (.not.query_initialized(cs%ave_ssh_ibc,"ave_ssh",restart_csp)) then
02741     if (cs%split) then
02742       call find_eta(cs%h, cs%tv, g, gv, us, cs%ave_ssh_ibc, eta, eta_to_m=1.0)
02743     else
02744       call find_eta(cs%h, cs%tv, g, gv, us, cs%ave_ssh_ibc, eta_to_m=1.0)
02745     endif
02746   endif
02747   if (cs%split) deallocate(eta)
02748
02749   cs%nstep_tot = 0
02750   if (present(count_calls)) cs%count_calls = count_calls
02751   call mom_sum_output_init(g_in, us, param_file, dirs%output_directory, &
02752                            cs%ntrunc, time_init, cs%sum_output_CSp)
02753
02754   ! Flag whether to save initial conditions in finish_MOM_initialization() or not.
02755   cs%write_IC = save_ic .and. &
02756                 .not.((dirs%input_filename(1:1) == 'r') .and. &
02757                       (len_trim(dirs%input_filename) == 1))
02758
02759   if (cs%ensemble_ocean) then
02760     call init_oda(time, g, gv, cs%odaCS)
02761   endif
02762
02763   !### This could perhaps go here instead of in finish_MOM_initialization?
02764   ! call fix_restart_scaling(GV)
02765   ! call fix_restart_unit_scaling(US)
02766
02767   call calltree_leave("initialize_MOM()")
02768   call cpu_clock_end(id_clock_init)
02769
```

42.1.5.7 mom_end()

```
subroutine, public mom::mom_end (
             type(mom_control_struct), pointer CS )
```

End of ocean model, including memory deallocation.

Parameters

cs	MOM control structure

Definition at line 3486 of file MOM.F90.

```
03486   type(mom_control_struct), pointer :: cs   !< MOM control structure
03487
03488   if (cs%use_ALE_algorithm) call ale_end(cs%ALE_CSp)
03489
03490   dealloc_(cs%u) ; dealloc_(cs%v) ; dealloc_(cs%h)
03491   dealloc_(cs%uh) ; dealloc_(cs%vh)
03492
03493   if (associated(cs%tv%T)) then
03494     dealloc_(cs%T) ; cs%tv%T => null() ; dealloc_(cs%S) ; cs%tv%S => null()
03495   endif
03496   if (associated(cs%tv%frazil)) deallocate(cs%tv%frazil)
03497   if (associated(cs%tv%salt_deficit)) deallocate(cs%tv%salt_deficit)
03498   if (associated(cs%Hml)) deallocate(cs%Hml)
03499
03500   call tracer_advect_end(cs%tracer_adv_CSp)
03501   call tracer_hor_diff_end(cs%tracer_diff_CSp)
03502   call tracer_registry_end(cs%tracer_Reg)
03503   call tracer_flow_control_end(cs%tracer_flow_CSp)
03504
03505   call diabatic_driver_end(cs%diabatic_CSp)
03506
03507   if (cs%offline_tracer_mode) call offline_transport_end(cs%offline_CSp)
03508
03509   dealloc_(cs%uhtr) ; dealloc_(cs%vhtr)
03510   if (cs%split) then
03511     call end_dyn_split_rk2(cs%dyn_split_RK2_CSp)
03512   elseif (cs%use_RK2) then
03513     call end_dyn_unsplit_rk2(cs%dyn_unsplit_RK2_CSp)
03514   else
03515     call end_dyn_unsplit(cs%dyn_unsplit_CSp)
03516   endif
03517   dealloc_(cs%ave_ssh_ibc) ; dealloc_(cs%ssh_rint)
03518   if (associated(cs%update_OBC_CSp)) call obc_register_end(cs%update_OBC_CSp)
03519
03520   call verticalgridend(cs%GV)
03521   call unit_scaling_end(cs%US)
03522   call mom_grid_end(cs%G)
03523
03524   deallocate(cs)
03525
```

42.1.5.8 mom_state_is_synchronized()

```
logical function, public mom::mom_state_is_synchronized (
             type(mom_control_struct), pointer CS,
             logical, intent(in), optional adv_dyn )
```

Return true if all phases of step_MOM are at the same point in time.

Parameters

	cs	MOM control structure
in	*adv_dyn*	If present and true, only check whether the advection is up-to-date with the dynamics.

Returns

True if all phases of the update are synchronized.

Definition at line 3429 of file MOM.F90.

Referenced by step_mom().

```
03429   type(mom_control_struct), pointer :: cs !< MOM control structure
03430   logical,        optional, intent(in) :: adv_dyn  !< If present and true, only check
03431                                            !! whether the advection is up-to-date with
03432                                            !! the dynamics.
03433   logical :: in_synch !< True if all phases of the update are synchronized.
03434
03435   logical :: adv_only
03436
03437   adv_only = .false. ; if (present(adv_dyn)) adv_only = adv_dyn
03438
03439   if (adv_only) then
03440     in_synch = (cs%t_dyn_rel_adv == 0.0)
03441   else
03442     in_synch = ((cs%t_dyn_rel_adv == 0.0) .and. (cs%t_dyn_rel_thermo == 0.0))
03443   endif
03444
```

42.1.5.9 mom_timing_init()

```
subroutine mom::mom_timing_init (
             type(mom_control_struct), intent(in) CS )  [private]
```

Set up CPU clock IDs for timing various subroutines.

Parameters

in	*cs*	control structure set up by initialize_MOM.

Definition at line 2855 of file MOM.F90.

References id_clock_adiabatic, id_clock_ale, id_clock_bbl_visc, id_clock_continuity, id_clock_diabatic, id_clock_diagnostics, id_clock_dynamics, id_clock_ml_restrat, id_clock_mom_init, id_clock_ocean, id_clock_offline_tracer, id_clock_other, id_clock_pass, id_clock_pass_init, id_clock_thermo, id_clock_thick_diff, id_clock_tracer, and id_clock_z_diag.

Referenced by initialize_mom().

```
02855   type(mom_control_struct), intent(in) :: cs  !< control structure set up by initialize_MOM.
02856
02857  id_clock_ocean    = cpu_clock_id('Ocean', grain=clock_component)
02858  id_clock_dynamics = cpu_clock_id('Ocean dynamics', grain=clock_subcomponent)
02859  id_clock_thermo   = cpu_clock_id('Ocean thermodynamics and tracers', grain=clock_subcomponent)
```

```
02860  id_clock_other    = cpu_clock_id('Ocean Other', grain=clock_subcomponent)
02861  id_clock_tracer   = cpu_clock_id('(Ocean tracer advection)', grain=clock_module_driver)
02862  if (.not.cs%adiabatic) then
02863    id_clock_diabatic = cpu_clock_id('(Ocean diabatic driver)', grain=clock_module_driver)
02864  else
02865    id_clock_adiabatic = cpu_clock_id('(Ocean adiabatic driver)', grain=clock_module_driver)
02866  endif
02867
02868  id_clock_continuity = cpu_clock_id('(Ocean continuity equation *)', grain=clock_module)
02869  id_clock_bbl_visc   = cpu_clock_id('(Ocean set BBL viscosity)', grain=clock_module)
02870  id_clock_pass       = cpu_clock_id('(Ocean message passing *)', grain=clock_module)
02871  id_clock_mom_init   = cpu_clock_id('(Ocean MOM_initialize_state)', grain=clock_module)
02872  id_clock_pass_init  = cpu_clock_id('(Ocean init message passing *)', grain=clock_routine)
02873  if (cs%thickness_diffuse) &
02874    id_clock_thick_diff = cpu_clock_id('(Ocean thickness diffusion *)', grain=clock_module)
02875 !if (CS%mixedlayer_restrat) &
02876    id_clock_ml_restrat = cpu_clock_id('(Ocean mixed layer restrat)', grain=clock_module)
02877  id_clock_diagnostics  = cpu_clock_id('(Ocean collective diagnostics)', grain=clock_module)
02878  id_clock_z_diag       = cpu_clock_id('(Ocean Z-space diagnostics)', grain=clock_module)
02879  id_clock_ale          = cpu_clock_id('(Ocean ALE)', grain=clock_module)
02880  if (cs%offline_tracer_mode) then
02881   id_clock_offline_tracer = cpu_clock_id('Ocean offline tracers', grain=clock_subcomponent)
02882  endif
02883
```

42.1.5.10 register_diags()

```
subroutine mom::register_diags (
             type(time_type), intent(in) Time,
             type(ocean_grid_type), intent(in) G,
             type(verticalgrid_type), intent(in) GV,
             type(unit_scale_type), intent(inout) US,
             type(mom_diag_ids), intent(inout) IDs,
             type(diag_ctrl), intent(inout) diag )  [private]
```

Register certain diagnostics.

Parameters

in	*time*	current model time
in	*g*	ocean grid structure
in	*gv*	ocean vertical grid structure
in,out	*us*	A dimensional unit scaling type
in,out	*ids*	A structure with the diagnostic IDs.
in,out	*diag*	regulates diagnostic output

Definition at line 2824 of file MOM.F90.

Referenced by initialize_mom().

```
02824   type(time_type),         intent(in)    :: time  !< current model time
02825   type(ocean_grid_type),   intent(in)    :: g     !< ocean grid structure
02826   type(verticalgrid_type), intent(in)    :: gv    !< ocean vertical grid structure
02827   type(unit_scale_type),   intent(inout) :: us    !< A dimensional unit scaling type
02828   type(mom_diag_ids),      intent(inout) :: ids   !< A structure with the diagnostic IDs.
02829   type(diag_ctrl),         intent(inout) :: diag  !< regulates diagnostic output
02830
02831   real :: h_convert
02832   character(len=48) :: thickness_units
02833
02834   thickness_units = get_thickness_units(gv)
02835   if (gv%Boussinesq) then
```

```
02836     h_convert = gv%H_to_m
02837   else
02838     h_convert = gv%H_to_kg_m2
02839   endif
02840
02841   ! Diagnostics of the rapidly varying dynamic state
02842   ids%id_u = register_diag_field('ocean_model', 'u_dyn', diag%axesCuL, time, &
02843       'Zonal velocity after the dynamics update', 'm s-1', conversion=us%L_T_to_m_s)
02844   ids%id_v = register_diag_field('ocean_model', 'v_dyn', diag%axesCvL, time, &
02845       'Meridional velocity after the dynamics update', 'm s-1', conversion=us%L_T_to_m_s)
02846   ids%id_h = register_diag_field('ocean_model', 'h_dyn', diag%axesTL, time, &
02847       'Layer Thickness after the dynamics update', thickness_units, &
02848       v_extensive=.true., conversion=h_convert)
02849   ids%id_ssh_inst = register_diag_field('ocean_model', 'SSH_inst', diag%axesT1, &
02850       time, 'Instantaneous Sea Surface Height', 'm')
```

42.1.5.11 rotate_initial_state()

```
subroutine mom::rotate_initial_state (
             real, dimension(:,:,:), intent(in) u_in,
             real, dimension(:,:,:), intent(in) v_in,
             real, dimension(:,:,:), intent(in) h_in,
             real, dimension(:,:,:), intent(in) T_in,
             real, dimension(:,:,:), intent(in) S_in,
             logical, intent(in) use_temperature,
             integer, intent(in) turns,
             real, dimension(:,:,:), intent(out) u,
             real, dimension(:,:,:), intent(out) v,
             real, dimension(:,:,:), intent(out) h,
             real, dimension(:,:,:), intent(out) T,
             real, dimension(:,:,:), intent(out) S )  [private]
```

Rotate initialization fields from input to rotated arrays.

Definition at line 3414 of file MOM.F90.

Referenced by initialize_mom().

```
03414   real, dimension(:,:,:), intent(in) :: u_in, v_in, h_in, t_in, s_in
03415   logical, intent(in) :: use_temperature
03416   integer, intent(in) :: turns
03417   real, dimension(:,:,:), intent(out) :: u, v, h, t, s
03418
03419   call rotate_vector(u_in, v_in, turns, u, v)
03420   call rotate_array(h_in, turns, h)
03421   if (use_temperature) then
03422     call rotate_array(t_in, turns, t)
03423     call rotate_array(s_in, turns, s)
03424   endif
```

42.1.5.12 set_restart_fields()

```
subroutine mom::set_restart_fields (
             type(verticalgrid_type), intent(inout) GV,
             type(unit_scale_type), intent(inout) US,
             type(param_file_type), intent(in) param_file,
             type(mom_control_struct), intent(in) CS,
             type(mom_restart_cs), pointer restart_CSp )  [private]
```

Set the fields that are needed for bitwise identical restarting the time stepping scheme. In addition to those specified here directly, there may be fields related to the forcing or to the barotropic solver that are needed; these are specified in sub- routines that are called from this one.

This routine should be altered if there are any changes to the time stepping scheme. The CHECK_RESTART facility may be used to confirm that all needed restart fields have been included.

Parameters

in,out	*gv*	ocean vertical grid structure
in,out	*us*	A dimensional unit scaling type
in	*param_file*	opened file for parsing to get parameters
in	*cs*	control structure set up by initialize_MOM
	restart_csp	pointer to the restart control structure that will be used for MOM.

Definition at line 2896 of file MOM.F90.

Referenced by initialize_mom().

```
02896   type(verticalgrid_type),  intent(inout) :: gv         !< ocean vertical grid structure
02897   type(unit_scale_type),    intent(inout) :: us         !< A dimensional unit scaling type
02898   type(param_file_type),    intent(in) :: param_file    !< opened file for parsing to get parameters
02899   type(mom_control_struct), intent(in) :: cs            !< control structure set up by initialize_MOM
02900   type(mom_restart_cs),     pointer    :: restart_csp   !< pointer to the restart control
02901                                                         !! structure that will be used for MOM.
02902   ! Local variables
02903   logical :: use_ice_shelf ! Needed to determine whether to add CS%Hml to restarts
02904   character(len=48) :: thickness_units, flux_units
02905   type(vardesc) :: u_desc, v_desc
02906
02907   thickness_units = get_thickness_units(gv)
02908   flux_units = get_flux_units(gv)
02909
02910   u_desc = var_desc("u", "m s-1", "Zonal velocity", hor_grid='Cu')
02911   v_desc = var_desc("v", "m s-1", "Meridional velocity", hor_grid='Cv')
02912
02913   if (associated(cs%tv%T)) &
02914     call register_restart_field(cs%tv%T, "Temp", .true., restart_csp, &
02915                                 "Potential Temperature", "degC")
02916   if (associated(cs%tv%S)) &
02917     call register_restart_field(cs%tv%S, "Salt", .true., restart_csp, &
02918                                 "Salinity", "PPT")
02919
02920   call register_restart_field(cs%h, "h", .true., restart_csp, &
02921                               "Layer Thickness", thickness_units)
02922
02923   call register_restart_pair(cs%u, cs%v, u_desc, v_desc, .true., restart_csp)
02924
02925   if (associated(cs%tv%frazil)) &
02926     call register_restart_field(cs%tv%frazil, "frazil", .false., restart_csp, &
02927                                 "Frazil heat flux into ocean", "J m-2")
02928
02929   if (cs%interp_p_surf) then
02930     call register_restart_field(cs%p_surf_prev, "p_surf_prev", .false., restart_csp, &
02931                                 "Previous ocean surface pressure", "Pa")
02932   endif
02933
02934   call register_restart_field(cs%ave_ssh_ibc, "ave_ssh", .false., restart_csp, &
02935                               "Time average sea surface height", "meter")
02936
02937   ! hML is needed when using the ice shelf module
02938   call get_param(param_file, '', "ICE_SHELF", use_ice_shelf, default=.false., &
02939                  do_not_log=.true.)
02940   if (use_ice_shelf .and. associated(cs%Hml)) then
02941     call register_restart_field(cs%Hml, "hML", .false., restart_csp, &
02942                                 "Mixed layer thickness", "meter")
02943   endif
02944
02945   ! Register scalar unit conversion factors.
02946   call register_restart_field(us%m_to_Z_restart, "m_to_Z", .false., restart_csp, &
02947                               "Height unit conversion factor", "Z meter-1")
02948   call register_restart_field(gv%m_to_H_restart, "m_to_H", .false., restart_csp, &
02949                               "Thickness unit conversion factor", "H meter-1")
02950   call register_restart_field(us%m_to_L_restart, "m_to_L", .false., restart_csp, &
02951                               "Length unit conversion factor", "L meter-1")
02952   call register_restart_field(us%s_to_T_restart, "s_to_T", .false., restart_csp, &
02953                               "Time unit conversion factor", "T second-1")
02954   call register_restart_field(us%kg_m3_to_R_restart, "kg_m3_to_R", .false., restart_csp, &
02955                               "Density unit conversion factor", "R m3 kg-1")
02956   call register_restart_field(us%J_kg_to_Q_restart, "J_kg_to_Q", .false., restart_csp, &
02957                               "Heat content unit conversion factor.", units="Q kg J-1")
02958
```

42.1.5.13 step_mom()

```
subroutine, public mom::step_mom (
             type(mech_forcing), intent(inout), target forces_in,
             type(forcing), intent(inout), target fluxes_in,
             type(surface), intent(inout), target sfc_state,
             type(time_type), intent(in) Time_start,
             real, intent(in) time_int_in,
             type(mom_control_struct), pointer CS,
             type(wave_parameters_cs), optional, pointer Waves,
             logical, intent(in), optional do_dynamics,
             logical, intent(in), optional do_thermodynamics,
             logical, intent(in), optional start_cycle,
             logical, intent(in), optional end_cycle,
             real, intent(in), optional cycle_length,
             logical, intent(in), optional reset_therm )
```

This subroutine orchestrates the time stepping of MOM. The adiabatic dynamics are stepped by calls to one of the step_MOM_dyn_...routines. The action of lateral processes on tracers occur in calls to advect_tracer and tracer_↩ hordiff. Vertical mixing and possibly remapping occur inside of diabatic.

Parameters

in,out	*forces_in*	A structure with the driving mechanical forces
in,out	*fluxes_in*	A structure with pointers to themodynamic, tracer and mass exchange forcing fields
in,out	*sfc_state*	surface ocean state
in	*time_start*	starting time of a segment, as a time type
in	*time_int_in*	time interval covered by this run segment [s].
	cs	control structure from initialize_MOM
	waves	An optional pointer to a wave property CS
in	*do_dynamics*	Present and false, do not do updates due to the dynamics.
in	*do_thermodynamics*	Present and false, do not do updates due to the thermodynamics or remapping.
in	*start_cycle*	This indicates whether this call is to be treated as the first call to step_MOM in a time-stepping cycle; missing is like true.
in	*end_cycle*	This indicates whether this call is to be treated as the last call to step_MOM in a time-stepping cycle; missing is like true.
in	*cycle_length*	The amount of time in a coupled time stepping cycle [s].
in	*reset_therm*	This indicates whether the running sums of thermodynamic quantities should be reset. If missing, this is like start_cycle.

Definition at line 422 of file MOM.F90.

References adjust_ssh_for_p_atm(), extract_surface_state(), id_clock_diagnostics, id_clock_dynamics, id_clock↩ _ocean, id_clock_other, id_clock_pass, mom_state_is_synchronized(), step_mom_dynamics(), step_mom_↩ thermo(), and step_mom_tracer_dyn().

```
00422   type(mech_forcing), target, intent(inout) :: forces_in !< A structure with the driving mechanical forces
00423   type(forcing), target, intent(inout) :: fluxes_in  !< A structure with pointers to themodynamic,
00424                                                      !! tracer and mass exchange forcing fields
00425   type(surface), target, intent(inout) :: sfc_state  !< surface ocean state
00426   type(time_type),    intent(in)    :: time_start    !< starting time of a segment, as a time type
00427   real,               intent(in)    :: time_int_in   !< time interval covered by this run segment [s].
00428   type(mom_control_struct), pointer :: cs            !< control structure from initialize_MOM
00429   type(wave_parameters_cs), &
```

```
00430            optional, pointer       :: waves         !< An optional pointer to a wave property CS
00431   logical, optional, intent(in)    :: do_dynamics   !< Present and false, do not do updates due
00432                                                     !! to the dynamics.
00433   logical, optional, intent(in)    :: do_thermodynamics  !< Present and false, do not do updates due
00434                                                     !! to the thermodynamics or remapping.
00435   logical, optional, intent(in)    :: start_cycle   !< This indicates whether this call is to be
00436                                                     !! treated as the first call to step_MOM in a
00437                                                     !! time-stepping cycle; missing is like true.
00438   logical, optional, intent(in)    :: end_cycle     !< This indicates whether this call is to be
00439                                                     !! treated as the last call to step_MOM in a
00440                                                     !! time-stepping cycle; missing is like true.
00441   real,    optional, intent(in)    :: cycle_length  !< The amount of time in a coupled time
00442                                                     !! stepping cycle [s].
00443   logical, optional, intent(in)    :: reset_therm   !< This indicates whether the running sums of
00444                                                     !! thermodynamic quantities should be reset.
00445                                                     !! If missing, this is like start_cycle.
00446
00447   ! local variables
00448   type(ocean_grid_type),   pointer :: g => null()  ! pointer to a structure containing
00449                                                    ! metrics and related information
00450   type(ocean_grid_type),   pointer :: g_in => null()  ! Input grid metric
00451   type(verticalgrid_type), pointer :: gv => null() ! Pointer to the vertical grid structure
00452   type(unit_scale_type),   pointer :: us => null() ! Pointer to a structure containing
00453                                                    ! various unit conversion factors
00454   integer       :: ntstep ! time steps between tracer updates or diabatic forcing
00455   integer       :: n_max  ! number of steps to take in this call
00456
00457   integer :: i, j, k, is, ie, js, je, isq, ieq, jsq, jeq, nz, n
00458   integer :: isd, ied, jsd, jed, isdb, iedb, jsdb, jedb
00459
00460   real :: time_interval   ! time interval covered by this run segment [T ~> s].
00461   real :: dt              ! baroclinic time step [T ~> s]
00462   real :: dtdia           ! time step for diabatic processes [T ~> s]
00463   real :: dt_therm        ! a limited and quantized version of CS%dt_therm [T ~> s]
00464   real :: dt_therm_here   ! a further limited value of dt_therm [T ~> s]
00465
00466   real :: wt_end, wt_beg
00467   real :: bbl_time_int    ! The amount of time over which the calculated BBL
00468                           ! properties will apply, for use in diagnostics, or 0
00469                           ! if it is not to be calculated anew [T ~> s].
00470   real :: rel_time = 0.0  ! relative time since start of this call [T ~> s].
00471
00472   logical :: calc_dtbt                ! Indicates whether the dynamically adjusted
00473                                       ! barotropic time step needs to be updated.
00474   logical :: do_advection             ! If true, it is time to advect tracers.
00475   logical :: do_calc_bbl              ! If true, calculate the boundary layer properties.
00476   logical :: thermo_does_span_coupling ! If true, thermodynamic forcing spans
00477                                       ! multiple dynamic timesteps.
00478   logical :: do_dyn     ! If true, dynamics are updated with this call.
00479   logical :: do_thermo  ! If true, thermodynamics and remapping may be applied with this call.
00480   logical :: cycle_start ! If true, do calculations that are only done at the start of
00481                          ! a stepping cycle (whatever that may mean).
00482   logical :: cycle_end  ! If true, do calculations and diagnostics that are only done at
00483                          ! the end of a stepping cycle (whatever that may mean).
00484   logical :: therm_reset ! If true, reset running sums of thermodynamic quantities.
00485   real :: cycle_time    ! The length of the coupled time-stepping cycle [T ~> s].
00486   real, dimension(SZI_(CS%G),SZJ_(CS%G)) :: &
00487     ssh         ! sea surface height, which may be based on eta_av [m]
00488
00489   real, dimension(:,:,:), pointer :: &
00490     u => null(), & ! u : zonal velocity component [L T-1 ~> m s-1]
00491     v => null(), & ! v : meridional velocity component [L T-1 ~> m s-1]
00492     h => null()    ! h : layer thickness [H ~> m or kg m-2]
00493   real, dimension(:,:), pointer :: &
00494     p_surf => null() ! A pointer to the ocean surface pressure [R L2 T-2 ~> Pa].
00495   real :: i_wt_ssh  ! The inverse of the time weights [T-1 ~> s-1]
00496
00497   type(time_type) :: time_local, end_time_thermo, time_temp
00498   type(group_pass_type) :: pass_tau_ustar_psurf
00499   logical :: showcalltree
00500
00501   ! External forcing fields on the model index map
00502   type(mech_forcing), pointer :: forces     ! Mechanical forcing
00503   type(forcing), pointer :: fluxes          ! Boundary fluxes
00504   type(surface), pointer :: sfc_state_diag  ! Surface boundary fields
00505   integer :: turns  ! Number of quarter turns from input to model indexing
00506
00507   g => cs%G ; g_in => cs%G_in ; gv => cs%GV ; us => cs%US
00508   is   = g%isc  ; ie   = g%iec  ; js   = g%jsc  ; je   = g%jec ; nz = g%ke
00509   isq  = g%IscB ; ieq  = g%IecB ; jsq  = g%JscB ; jeq  = g%JecB
00510   isd  = g%isd  ; ied  = g%ied  ; jsd  = g%jsd  ; jed  = g%jed
00511   isdb = g%IsdB ; iedb = g%IedB ; jsdb = g%JsdB ; jedb = g%JedB
00512   u => cs%u ; v => cs%v ; h => cs%h
00513
00514   time_interval = us%s_to_T*time_int_in
00515   do_dyn = .true. ; if (present(do_dynamics)) do_dyn = do_dynamics
00516   do_thermo = .true. ; if (present(do_thermodynamics)) do_thermo = do_thermodynamics
```

```
00517   if (.not.(do_dyn .or. do_thermo)) call mom_error(fatal,"Step_MOM: "//&
00518     "Both do_dynamics and do_thermodynamics are false, which makes no sense.")
00519   cycle_start = .true. ; if (present(start_cycle)) cycle_start = start_cycle
00520   cycle_end = .true. ; if (present(end_cycle)) cycle_end = end_cycle
00521   cycle_time = time_interval ; if (present(cycle_length)) cycle_time = us%s_to_T*cycle_length
00522   therm_reset = cycle_start ; if (present(reset_therm)) therm_reset = reset_therm
00523
00524   call cpu_clock_begin(id_clock_ocean)
00525   call cpu_clock_begin(id_clock_other)
00526
00527   if (cs%debug) then
00528     call mom_state_chksum("Beginning of step_MOM ", u, v, h, cs%uh, cs%vh, g, gv, us)
00529   endif
00530
00531   showcalltree = calltree_showquery()
00532   if (showcalltree) call calltree_enter("step_MOM(), MOM.F90")
00533
00534   ! Rotate the forces from G_in to G
00535   if (cs%rotate_index) then
00536     turns = g%HI%turns
00537     allocate(forces)
00538     call allocate_mech_forcing(forces_in, g, forces)
00539     call rotate_mech_forcing(forces_in, turns, forces)
00540
00541     allocate(fluxes)
00542     call allocate_forcing_type(fluxes_in, g, fluxes)
00543     call rotate_forcing(fluxes_in, fluxes, turns)
00544   else
00545     forces => forces_in
00546     fluxes => fluxes_in
00547   endif
00548
00549   ! First determine the time step that is consistent with this call and an
00550   ! integer fraction of time_interval.
00551   if (do_dyn) then
00552     n_max = 1
00553     if (time_interval > cs%dt) n_max = ceiling(time_interval/cs%dt - 0.001)
00554     dt = time_interval / real(n_max)
00555     thermo_does_span_coupling = (cs%thermo_spans_coupling .and. &
00556                                  (cs%dt_therm > 1.5*cycle_time))
00557     if (thermo_does_span_coupling) then
00558       ! Set dt_therm to be an integer multiple of the coupling time step.
00559       dt_therm = cycle_time * floor(cs%dt_therm / cycle_time + 0.001)
00560       ntstep = floor(dt_therm/dt + 0.001)
00561     elseif (.not.do_thermo) then
00562       dt_therm = cs%dt_therm
00563       if (present(cycle_length)) dt_therm = min(cs%dt_therm, us%s_to_T*cycle_length)
00564       ! ntstep is not used.
00565     else
00566       ntstep = max(1, min(n_max, floor(cs%dt_therm/dt + 0.001)))
00567       dt_therm = dt*ntstep
00568     endif
00569
00570     if (associated(forces%p_surf)) p_surf => forces%p_surf
00571     if (.not.associated(forces%p_surf)) cs%interp_p_surf = .false.
00572     cs%tv%p_surf => null()
00573     if (cs%use_p_surf_in_EOS .and. associated(forces%p_surf)) cs%tv%p_surf => forces%p_surf
00574
00575     !---------- Initiate group halo pass of the forcing fields
00576     call cpu_clock_begin(id_clock_pass)
00577     call create_group_pass(pass_tau_ustar_psurf, forces%taux, forces%tauy, g%Domain)
00578     if (associated(forces%ustar)) &
00579       call create_group_pass(pass_tau_ustar_psurf, forces%ustar, g%Domain)
00580     if (associated(forces%p_surf)) &
00581       call create_group_pass(pass_tau_ustar_psurf, forces%p_surf, g%Domain)
00582     if (g%nonblocking_updates) then
00583       call start_group_pass(pass_tau_ustar_psurf, g%Domain)
00584     else
00585       call do_group_pass(pass_tau_ustar_psurf, g%Domain)
00586     endif
00587     call cpu_clock_end(id_clock_pass)
00588   else
00589     ! This step only updates the thermodynamics so setting timesteps is simpler.
00590     n_max = 1
00591     if ((time_interval > cs%dt_therm) .and. (cs%dt_therm > 0.0)) &
00592       n_max = ceiling(time_interval/cs%dt_therm - 0.001)
00593
00594     dt = time_interval / real(n_max)
00595     dt_therm = dt ; ntstep = 1
00596     if (associated(fluxes%p_surf)) p_surf => fluxes%p_surf
00597     cs%tv%p_surf => null()
00598     if (associated(fluxes%p_surf)) then
00599       if (cs%use_p_surf_in_EOS) cs%tv%p_surf => fluxes%p_surf
00600     endif
00601     if (cs%UseWaves) call pass_var(fluxes%ustar, g%Domain, clock=id_clock_pass)
00602   endif
00603
```

```
00604   if (therm_reset) then
00605     cs%time_in_thermo_cycle = 0.0
00606     if (associated(cs%tv%frazil))        cs%tv%frazil(:,:)        = 0.0
00607     if (associated(cs%tv%salt_deficit))  cs%tv%salt_deficit(:,:)  = 0.0
00608     if (associated(cs%tv%TempxPmE))      cs%tv%TempxPmE(:,:)      = 0.0
00609     if (associated(cs%tv%internal_heat)) cs%tv%internal_heat(:,:) = 0.0
00610   endif
00611
00612   if (cycle_start) then
00613     cs%time_in_cycle = 0.0
00614     do j=js,je ; do i=is,ie ; cs%ssh_rint(i,j) = 0.0 ; enddo ; enddo
00615
00616     if (associated(cs%VarMix)) then
00617       call enable_averages(cycle_time, time_start + real_to_time(us%T_to_s*cycle_time), cs%diag)
00618       call calc_resoln_function(h, cs%tv, g, gv, us, cs%VarMix)
00619       call calc_depth_function(g, cs%VarMix)
00620       call disable_averaging(cs%diag)
00621     endif
00622   endif
00623
00624   if (do_dyn) then
00625     if (g%nonblocking_updates) &
00626       call complete_group_pass(pass_tau_ustar_psurf, g%Domain, clock=id_clock_pass)
00627
00628     if (cs%interp_p_surf) then
00629       if (.not.associated(cs%p_surf_end))   allocate(cs%p_surf_end(isd:ied,jsd:jed))
00630       if (.not.associated(cs%p_surf_begin)) allocate(cs%p_surf_begin(isd:ied,jsd:jed))
00631       if (.not.cs%p_surf_prev_set) then
00632         do j=jsd,jed ; do i=isd,ied
00633           cs%p_surf_prev(i,j) = forces%p_surf(i,j)
00634         enddo ; enddo
00635         cs%p_surf_prev_set = .true.
00636       endif
00637     else
00638       cs%p_surf_end  => forces%p_surf
00639     endif
00640
00641     if (cs%UseWaves) then
00642       ! Update wave information, which is presently kept static over each call to step_mom
00643       call enable_averages(time_interval, time_start + real_to_time(us%T_to_s*time_interval), cs%diag)
00644       call update_stokes_drift(g, gv, us, waves, h, forces%ustar)
00645       call disable_averaging(cs%diag)
00646     endif
00647   else ! not do_dyn.
00648     if (cs%UseWaves) & ! Diagnostics are not enabled in this call.
00649       call update_stokes_drift(g, gv, us, waves, h, fluxes%ustar)
00650   endif
00651
00652   if (cs%debug) then
00653     if (cycle_start) &
00654       call mom_state_chksum("Before steps ", u, v, h, cs%uh, cs%vh, g, gv, us)
00655     if (cycle_start) call check_redundant("Before steps ", u, v, g)
00656     if (do_dyn) call mom_mech_forcing_chksum("Before steps", forces, g, us, haloshift=0)
00657     if (do_dyn) call check_redundant("Before steps ", forces%taux, forces%tauy, g)
00658   endif
00659   call cpu_clock_end(id_clock_other)
00660
00661   rel_time = 0.0
00662   do n=1,n_max
00663     rel_time = rel_time + dt ! The relative time at the end of the step.
00664     ! Set the universally visible time to the middle of the time step.
00665     cs%Time = time_start + real_to_time(us%T_to_s*(rel_time - 0.5*dt))
00666     ! Set the local time to the end of the time step.
00667     time_local = time_start + real_to_time(us%T_to_s*rel_time)
00668
00669     if (showcalltree) call calltree_enter("DT cycles (step_MOM) n=",n)
00670
00671     ! Update the vertically extensive diagnostic grids so that they are
00672     ! referenced to the beginning timestep
00673     call diag_update_remap_grids(cs%diag, update_intensive = .false., update_extensive = .true. )
00674
00675     !===========================================================================
00676     ! This is the first place where the diabatic processes and remapping could occur.
00677     if (cs%diabatic_first .and. (cs%t_dyn_rel_adv==0.0) .and. do_thermo) then ! do thermodynamics.
00678
00679       if (.not.do_dyn) then
00680         dtdia = dt
00681       elseif (thermo_does_span_coupling) then
00682         dtdia = dt_therm
00683         if ((fluxes%dt_buoy_accum > 0.0) .and. (dtdia > time_interval) .and. &
00684             (abs(fluxes%dt_buoy_accum - dtdia) > 1e-6*dtdia)) then
00685           call mom_error(fatal, "step_MOM: Mismatch between long thermodynamic "//&
00686             "timestep and time over which buoyancy fluxes have been accumulated.")
00687         endif
00688         call mom_error(fatal, "MOM is not yet set up to have restarts that work "//&
00689           "with THERMO_SPANS_COUPLING and DIABATIC_FIRST.")
00690       else
```

```
00691         dtdia = dt*min(ntstep,n_max-(n-1))
00692       endif
00693
00694       end_time_thermo = time_local
00695       if (dtdia > dt) then
00696         ! If necessary, temporarily reset CS%Time to the center of the period covered
00697         ! by the call to step_MOM_thermo, noting that they begin at the same time.
00698         cs%Time = cs%Time + real_to_time(0.5*us%T_to_s*(dtdia-dt))
00699         ! The end-time of the diagnostic interval needs to be set ahead if there
00700         ! are multiple dynamic time steps worth of thermodynamics applied here.
00701         end_time_thermo = Time_local + real_to_time(US%T_to_s*(dtdia-dt))
00702       endif
00703
00704       ! Apply diabatic forcing, do mixing, and regrid.
00705       call step_mom_thermo(cs, g, gv, us, u, v, h, cs%tv, fluxes, dtdia, &
00706                            end_time_thermo, .true., Waves=Waves)
00707       cs%time_in_thermo_cycle = cs%time_in_thermo_cycle + dtdia
00708
00709       ! The diabatic processes are now ahead of the dynamics by dtdia.
00710       cs%t_dyn_rel_thermo = -dtdia
00711       if (showcalltree) call calltree_waypoint("finished diabatic_first (step_MOM)")
00712
00713       if (dtdia > dt) & ! Reset CS%Time to its previous value.
00714         cs%Time = time_start + real_to_time(us%T_to_s*(rel_time - 0.5*dt))
00715     endif ! end of block "(CS%diabatic_first .and. (CS%t_dyn_rel_adv==0.0))"
00716
00717     if (do_dyn) then
00718       ! Store pre-dynamics thicknesses for proper diagnostic remapping for transports or
00719       ! advective tendencies.  If there are more than one dynamics steps per advective
00720       ! step (i.e DT_THERM > DT), this needs to be stored at the first dynamics call.
00721       if (.not.cs%preadv_h_stored .and. (cs%t_dyn_rel_adv == 0.)) then
00722         call diag_copy_diag_to_storage(cs%diag_pre_dyn, h, cs%diag)
00723         cs%preadv_h_stored = .true.
00724       endif
00725
00726       ! The pre-dynamics velocities might be stored for debugging truncations.
00727       if (associated(cs%u_prev) .and. associated(cs%v_prev)) then
00728         do k=1,nz ; do j=jsd,jed ; do i=isdb,iedb
00729           cs%u_prev(i,j,k) = u(i,j,k)
00730         enddo ; enddo ; enddo
00731         do k=1,nz ; do j=jsdb,jedb ; do i=isd,ied
00732           cs%v_prev(i,j,k) = v(i,j,k)
00733         enddo ; enddo ; enddo
00734       endif
00735
00736       dt_therm_here = dt_therm
00737       if (do_thermo .and. do_dyn .and. .not.thermo_does_span_coupling) &
00738         dt_therm_here = dt*min(ntstep, n_max-n+1)
00739
00740       ! Indicate whether the bottom boundary layer properties need to be
00741       ! recalculated, and if so for how long an interval they are valid.
00742       bbl_time_int = 0.0
00743       if (do_thermo) then
00744         if ((cs%t_dyn_rel_adv == 0.0) .or. (n==1)) &
00745           bbl_time_int = max(dt, min(dt_therm - cs%t_dyn_rel_adv, dt*(1+n_max-n)) )
00746       else
00747         if ((cs%t_dyn_rel_adv == 0.0) .or. ((n==1) .and. cycle_start)) &
00748           bbl_time_int = min(dt_therm, cycle_time)
00749       endif
00750
00751       if (cs%interp_p_surf) then
00752         wt_end = real(n) / real(n_max)
00753         wt_beg = real(n-1) / real(n_max)
00754         do j=jsd,jed ; do i=isd,ied
00755           cs%p_surf_end(i,j) = wt_end * forces%p_surf(i,j) + &
00756                                (1.0-wt_end) * cs%p_surf_prev(i,j)
00757           cs%p_surf_begin(i,j) = wt_beg * forces%p_surf(i,j) + &
00758                                  (1.0-wt_beg) * cs%p_surf_prev(i,j)
00759         enddo ; enddo
00760       endif
00761
00762       call step_mom_dynamics(forces, cs%p_surf_begin, cs%p_surf_end, dt, &
00763                              dt_therm_here, bbl_time_int, cs, &
00764                              time_local, waves=waves)
00765
00766       !===============================================================================
00767       ! This is the start of the tracer advection part of the algorithm.
00768
00769       if (thermo_does_span_coupling .or. .not.do_thermo) then
00770         do_advection = (cs%t_dyn_rel_adv + 0.5*dt > dt_therm)
00771       else
00772         do_advection = ((mod(n,ntstep) == 0) .or. (n==n_max))
00773       endif
00774
00775       if (do_advection) then ! Do advective transport and lateral tracer mixing.
00776         call step_mom_tracer_dyn(cs, g, gv, us, h, time_local)
00777         if (cs%diabatic_first .and. abs(cs%t_dyn_rel_thermo) > 1e-6*dt) call mom_error(fatal, &
```

```
00778                 "step_MOM: Mismatch between the dynamics and diabatic times "//&
00779                 "with DIABATIC_FIRST.")
00780       endif
00781     endif ! end of (do_dyn)
00782
00783     !============================================================================
00784     ! This is the second place where the diabatic processes and remapping could occur.
00785     if ((cs%t_dyn_rel_adv==0.0) .and. do_thermo .and. (.not.cs%diabatic_first)) then
00786
00787       dtdia = cs%t_dyn_rel_thermo
00788       ! If the MOM6 dynamic and thermodynamic time stepping is being orchestrated
00789       ! by the coupler, the value of diabatic_first does not matter.
00790       if ((cs%t_dyn_rel_thermo==0.0) .and. .not.do_dyn) dtdia = dt
00791
00792       if (cs%thermo_spans_coupling .and. (cs%dt_therm > 1.5*cycle_time) .and. &
00793           (abs(dt_therm - dtdia) > 1e-6*dt_therm)) then
00794         call mom_error(fatal, "step_MOM: Mismatch between dt_therm and dtdia "//&
00795                        "before call to diabatic.")
00796       endif
00797
00798       ! If necessary, temporarily reset CS%Time to the center of the period covered
00799       ! by the call to step_MOM_thermo, noting that they end at the same time.
00800       if (dtdia > dt) cs%Time = cs%Time - real_to_time(0.5*us%T_to_s*(dtdia-dt))
00801
00802       ! Apply diabatic forcing, do mixing, and regrid.
00803       call step_mom_thermo(cs, g, gv, us, u, v, h, cs%tv, fluxes, dtdia, &
00804                            time_local, .false., waves=waves)
00805       cs%time_in_thermo_cycle = cs%time_in_thermo_cycle + dtdia
00806
00807       if ((cs%t_dyn_rel_thermo==0.0) .and. .not.do_dyn) then
00808         ! The diabatic processes are now ahead of the dynamics by dtdia.
00809         cs%t_dyn_rel_thermo = -dtdia
00810       else ! The diabatic processes and the dynamics are synchronized.
00811         cs%t_dyn_rel_thermo = 0.0
00812       endif
00813
00814       if (dtdia > dt) & ! Reset CS%Time to its previous value.
00815         cs%Time = time_start + real_to_time(us%T_to_s*(rel_time - 0.5*dt))
00816     endif
00817
00818     if (do_dyn) then
00819       call cpu_clock_begin(id_clock_dynamics)
00820       ! Determining the time-average sea surface height is part of the algorithm.
00821       ! This may be eta_av if Boussinesq, or need to be diagnosed if not.
00822       cs%time_in_cycle = cs%time_in_cycle + dt
00823       call find_eta(h, cs%tv, g, gv, us, ssh, cs%eta_av_bc, eta_to_m=1.0)
00824       do j=js,je ; do i=is,ie
00825         cs%ssh_rint(i,j) = cs%ssh_rint(i,j) + dt*ssh(i,j)
00826       enddo ; enddo
00827       if (cs%IDs%id_ssh_inst > 0) call post_data(cs%IDs%id_ssh_inst, ssh, cs%diag)
00828       call cpu_clock_end(id_clock_dynamics)
00829     endif
00830
00831     !============================================================================
00832     ! Calculate diagnostics at the end of the time step if the state is self-consistent.
00833     if (mom_state_is_synchronized(cs)) then
00834     !### Perhaps this should be if (CS%t_dyn_rel_thermo == 0.0)
00835       call cpu_clock_begin(id_clock_other) ; call cpu_clock_begin(id_clock_diagnostics)
00836       ! Diagnostics that require the complete state to be up-to-date can be calculated.
00837
00838       call enable_averages(cs%t_dyn_rel_diag, time_local, cs%diag)
00839       call calculate_diagnostic_fields(u, v, h, cs%uh, cs%vh, cs%tv, cs%ADp,  &
00840                           cs%CDp, p_surf, cs%t_dyn_rel_diag, cs%diag_pre_sync,&
00841                           g, gv, us, cs%diagnostics_CSp)
00842       call post_tracer_diagnostics_at_sync(cs%Tracer_reg, h, cs%diag_pre_sync, cs%diag, g, gv, cs
    %t_dyn_rel_diag)
00843       call diag_copy_diag_to_storage(cs%diag_pre_sync, h, cs%diag)
00844       if (showcalltree) call calltree_waypoint("finished calculate_diagnostic_fields (step_MOM)")
00845       call disable_averaging(cs%diag)
00846       cs%t_dyn_rel_diag = 0.0
00847
00848       call cpu_clock_end(id_clock_diagnostics) ; call cpu_clock_end(id_clock_other)
00849     endif
00850
00851     if (do_dyn .and. .not.cs%count_calls) cs%nstep_tot = cs%nstep_tot + 1
00852     if (showcalltree) call calltree_leave("DT cycles (step_MOM)")
00853
00854   enddo ! complete the n loop
00855
00856   if (cs%count_calls .and. cycle_start) cs%nstep_tot = cs%nstep_tot + 1
00857
00858   call cpu_clock_begin(id_clock_other)
00859
00860   if (cs%time_in_cycle > 0.0) then
00861     i_wt_ssh = 1.0/cs%time_in_cycle
00862     do j=js,je ; do i=is,ie
00863       ssh(i,j) = cs%ssh_rint(i,j)*i_wt_ssh
```

```
00864       cs%ave_ssh_ibc(i,j) = ssh(i,j)
00865     enddo ; enddo
00866     if (do_dyn) then
00867       call adjust_ssh_for_p_atm(cs%tv, g, gv, us, cs%ave_ssh_ibc, forces%p_surf_SSH, &
00868                                 cs%calc_rho_for_sea_lev)
00869     elseif (do_thermo) then
00870       call adjust_ssh_for_p_atm(cs%tv, g, gv, us, cs%ave_ssh_ibc, fluxes%p_surf_SSH, &
00871                                 cs%calc_rho_for_sea_lev)
00872     endif
00873   endif
00874
00875   if (do_dyn .and. cs%interp_p_surf) then ; do j=jsd,jed ; do i=isd,ied
00876     cs%p_surf_prev(i,j) = forces%p_surf(i,j)
00877   enddo ; enddo ; endif
00878
00879   if (cs%ensemble_ocean) then
00880     ! update the time for the next analysis step if needed
00881     call set_analysis_time(cs%Time,cs%odaCS)
00882     ! store ensemble vector in odaCS
00883     call set_prior_tracer(cs%Time, g, gv, cs%h, cs%tv, cs%odaCS)
00884     ! call DA interface
00885     call oda(cs%Time,cs%odaCS)
00886   endif
00887
00888   if (showcalltree) call calltree_waypoint("calling extract_surface_state (step_MOM)")
00889   ! NOTE: sfc_state uses input indexing, since it is also used by drivers.
00890   call extract_surface_state(cs, sfc_state)
00891
00892   ! Do diagnostics that only occur at the end of a complete forcing step.
00893   if (cycle_end) then
00894     if (cs%rotate_index) then
00895       allocate(sfc_state_diag)
00896       call rotate_surface_state(sfc_state, g_in, sfc_state_diag, g, turns)
00897     else
00898       sfc_state_diag => sfc_state
00899     endif
00900
00901     call cpu_clock_begin(id_clock_diagnostics)
00902     if (cs%time_in_cycle > 0.0) then
00903       call enable_averages(cs%time_in_cycle, time_local, cs%diag)
00904       call post_surface_dyn_diags(cs%sfc_IDs, g, cs%diag, sfc_state_diag, ssh)
00905     endif
00906     if (cs%time_in_thermo_cycle > 0.0) then
00907       call enable_averages(cs%time_in_thermo_cycle, time_local, cs%diag)
00908       call post_surface_thermo_diags(cs%sfc_IDs, g, gv, us, cs%diag, cs%time_in_thermo_cycle, &
00909                                      sfc_state_diag, cs%tv, ssh, cs%ave_ssh_ibc)
00910     endif
00911     call disable_averaging(cs%diag)
00912     call cpu_clock_end(id_clock_diagnostics)
00913   endif
00914
00915   ! Accumulate the surface fluxes for assessing conservation
00916   if (do_thermo .and. fluxes%fluxes_used) &
00917     call accumulate_net_input(fluxes, sfc_state, cs%tv, fluxes%dt_buoy_accum, &
00918                               g, us, cs%sum_output_CSp)
00919
00920   if (mom_state_is_synchronized(cs)) &
00921     call write_energy(cs%u, cs%v, cs%h, cs%tv, time_local, cs%nstep_tot, &
00922                       g, gv, us, cs%sum_output_CSp, cs%tracer_flow_CSp, &
00923                       dt_forcing=real_to_time(us%T_to_s*time_interval) )
00924
00925   call cpu_clock_end(id_clock_other)
00926
00927   ! De-rotate fluxes and copy back to the input, since they can be changed.
00928   if (cs%rotate_index) then
00929     call rotate_forcing(fluxes, fluxes_in, -turns)
00930
00931     call deallocate_mech_forcing(forces)
00932     deallocate(forces)
00933
00934     call deallocate_forcing_type(fluxes)
00935     deallocate(fluxes)
00936   endif
00937
00938   if (showcalltree) call calltree_leave("step_MOM()")
00939   call cpu_clock_end(id_clock_ocean)
00940
```

42.1.5.14 step_mom_dynamics()

```
subroutine mom::step_mom_dynamics (
```

```
type(mech_forcing), intent(in) forces,
real, dimension(:,:), pointer p_surf_begin,
real, dimension(:,:), pointer p_surf_end,
real, intent(in) dt,
real, intent(in) dt_thermo,
real, intent(in) bbl_time_int,
type(mom_control_struct), pointer CS,
type(time_type), intent(in) Time_local,
type(wave_parameters_cs), optional, pointer Waves )  [private]
```

Time step the ocean dynamics, including the momentum and continuity equations.

Parameters

in	*forces*	A structure with the driving mechanical forces
	p_surf_begin	A pointer (perhaps NULL) to the surface pressure at the beginning of this dynamic step, intent in [R L2 T-2 ~> Pa].
	p_surf_end	A pointer (perhaps NULL) to the surface pressure at the end of this dynamic step, intent in [R L2 T-2 ~> Pa].
in	*dt*	time interval covered by this call [T ~> s].
in	*dt_thermo*	time interval covered by any updates that may span multiple dynamics steps [T ~> s].
in	*bbl_time_int*	time interval over which updates to the bottom boundary layer properties will apply [T ~> s], or zero not to update the properties.
	cs	control structure from initialize_MOM
in	*time_local*	End time of a segment, as a time type
	waves	Container for wave related parameters; the

Definition at line 946 of file MOM.F90.

References id_clock_bbl_visc, id_clock_diagnostics, id_clock_dynamics, id_clock_ml_restrat, id_clock_other, id_clock_pass, and id_clock_thick_diff.

Referenced by step_mom().

```
00946   type(mech_forcing), intent(in)    :: forces     !< A structure with the driving mechanical forces
00947   real, dimension(:,:), pointer     :: p_surf_begin !< A pointer (perhaps NULL) to the surface
00948                                                    !! pressure at the beginning of this dynamic
00949                                                    !! step, intent in [R L2 T-2 ~> Pa].
00950   real, dimension(:,:), pointer     :: p_surf_end !< A pointer (perhaps NULL) to the surface
00951                                                    !! pressure at the end of this dynamic step,
00952                                                    !! intent in [R L2 T-2 ~> Pa].
00953   real,                 intent(in)  :: dt         !< time interval covered by this call [T ~> s].
00954   real,                 intent(in)  :: dt_thermo  !< time interval covered by any updates that may
00955                                                    !! span multiple dynamics steps [T ~> s].
00956   real,                 intent(in)  :: bbl_time_int !< time interval over which updates to the
00957                                                    !! bottom boundary layer properties will apply [T ~> s],
00958                                                    !! or zero not to update the properties.
00959   type(mom_control_struct), pointer :: cs         !< control structure from initialize_MOM
00960   type(time_type),    intent(in)    :: time_local !< End time of a segment, as a time type
00961   type(wave_parameters_cs), &
00962             optional, pointer       :: waves      !< Container for wave related parameters; the
00963                                                    !! fields in Waves are intent in here.
00964
00965   ! local variables
00966   type(ocean_grid_type),   pointer :: g => null()  ! pointer to a structure containing
00967                                                    ! metrics and related information
00968   type(verticalgrid_type), pointer :: gv => null() ! Pointer to the vertical grid structure
00969   type(unit_scale_type),   pointer :: us => null() ! Pointer to a structure containing
00970                                                    ! various unit conversion factors
00971   type(mom_diag_ids), pointer :: ids => null() ! A structure with the diagnostic IDs.
00972   real, dimension(:,:,:), pointer :: &
00973     u => null(), & ! u : zonal velocity component [L T-1 ~> m s-1]
00974     v => null(), & ! v : meridional velocity component [L T-1 ~> m s-1]
00975     h => null()    ! h : layer thickness [H ~> m or kg m-2]
00976
```

```
00977   logical :: calc_dtbt  ! Indicates whether the dynamically adjusted
00978                         ! barotropic time step needs to be updated.
00979   logical :: showcalltree
00980
00981   integer :: i, j, k, is, ie, js, je, isq, ieq, jsq, jeq, nz
00982   integer :: isd, ied, jsd, jed, isdb, iedb, jsdb, jedb
00983
00984   g => cs%G ; gv => cs%GV ; us => cs%US ; ids => cs%IDs
00985   is   = g%isc  ; ie   = g%iec  ; js   = g%jsc  ; je   = g%jec ; nz = g%ke
00986   isq  = g%IscB ; ieq  = g%IecB ; jsq  = g%JscB ; jeq  = g%JecB
00987   isd  = g%isd  ; ied  = g%ied  ; jsd  = g%jsd  ; jed  = g%jed
00988   isdb = g%IsdB ; iedb = g%IedB ; jsdb = g%JsdB ; jedb = g%JedB
00989   u => cs%u ; v => cs%v ; h => cs%h
00990   showcalltree = calltree_showquery()
00991
00992   call cpu_clock_begin(id_clock_dynamics)
00993
00994   if ((cs%t_dyn_rel_adv == 0.0) .and. cs%thickness_diffuse .and. cs%thickness_diffuse_first) then
00995
00996     call enable_averages(dt_thermo, time_local+real_to_time(us%T_to_s*(dt_thermo-dt)), cs%diag)
00997     call cpu_clock_begin(id_clock_thick_diff)
00998     if (associated(cs%VarMix)) &
00999       call calc_slope_functions(h, cs%tv, dt, g, gv, us, cs%VarMix, obc=cs%OBC)
01000     call thickness_diffuse(h, cs%uhtr, cs%vhtr, cs%tv, dt_thermo, g, gv, us, &
01001                            cs%MEKE, cs%VarMix, cs%CDp, cs%thickness_diffuse_CSp)
01002     call cpu_clock_end(id_clock_thick_diff)
01003     call pass_var(h, g%Domain, clock=id_clock_pass, halo=max(2,cs%cont_stencil))
01004     call disable_averaging(cs%diag)
01005     if (showcalltree) call calltree_waypoint("finished thickness_diffuse_first (step_MOM)")
01006
01007     ! Whenever thickness changes let the diag manager know, target grids
01008     ! for vertical remapping may need to be regenerated.
01009     call diag_update_remap_grids(cs%diag)
01010   endif
01011
01012   ! The bottom boundary layer properties need to be recalculated.
01013   if (bbl_time_int > 0.0) then
01014     call enable_averages(bbl_time_int, &
01015              time_local + real_to_time(us%T_to_s*(bbl_time_int-dt)), cs%diag)
01016     ! Calculate the BBL properties and store them inside visc (u,h).
01017     call cpu_clock_begin(id_clock_bbl_visc)
01018     call set_viscous_bbl(cs%u, cs%v, cs%h, cs%tv, cs%visc, g, gv, us, &
01019                          cs%set_visc_CSp, symmetrize=.true.)
01020     call cpu_clock_end(id_clock_bbl_visc)
01021     if (showcalltree) call calltree_waypoint("done with set_viscous_BBL (step_MOM)")
01022     call disable_averaging(cs%diag)
01023   endif
01024
01025
01026   if (cs%do_dynamics .and. cs%split) then !--------------------------- start SPLIT
01027     ! This section uses a split time stepping scheme for the dynamic equations,
01028     ! basically the stacked shallow water equations with viscosity.
01029
01030     calc_dtbt = .false.
01031     if (cs%dtbt_reset_period == 0.0) calc_dtbt = .true.
01032     if (cs%dtbt_reset_period > 0.0) then
01033       if (time_local >= cs%dtbt_reset_time) then  !### Change >= to > here.
01034         calc_dtbt = .true.
01035         cs%dtbt_reset_time = cs%dtbt_reset_time + cs%dtbt_reset_interval
01036       endif
01037     endif
01038
01039     call step_mom_dyn_split_rk2(u, v, h, cs%tv, cs%visc, time_local, dt, forces, &
01040                 p_surf_begin, p_surf_end, cs%uh, cs%vh, cs%uhtr, cs%vhtr, &
01041                 cs%eta_av_bc, g, gv, us, cs%dyn_split_RK2_CSp, calc_dtbt, cs%VarMix, &
01042                 cs%MEKE, cs%thickness_diffuse_CSp, waves=waves)
01043     if (showcalltree) call calltree_waypoint("finished step_MOM_dyn_split (step_MOM)")
01044
01045   elseif (cs%do_dynamics) then ! ----------------------------------- not SPLIT
01046     !   This section uses an unsplit stepping scheme for the dynamic
01047     ! equations; basically the stacked shallow water equations with viscosity.
01048     ! Because the time step is limited by CFL restrictions on the external
01049     ! gravity waves, the unsplit is usually much less efficient that the split
01050     ! approaches. But because of its simplicity, the unsplit method is very
01051     ! useful for debugging purposes.
01052
01053     if (cs%use_RK2) then
01054       call step_mom_dyn_unsplit_rk2(u, v, h, cs%tv, cs%visc, time_local, dt, forces, &
01055                p_surf_begin, p_surf_end, cs%uh, cs%vh, cs%uhtr, cs%vhtr, &
01056                cs%eta_av_bc, g, gv, us, cs%dyn_unsplit_RK2_CSp, cs%VarMix, cs%MEKE)
01057     else
01058       call step_mom_dyn_unsplit(u, v, h, cs%tv, cs%visc, time_local, dt, forces, &
01059                p_surf_begin, p_surf_end, cs%uh, cs%vh, cs%uhtr, cs%vhtr, &
01060                cs%eta_av_bc, g, gv, us, cs%dyn_unsplit_CSp, cs%VarMix, cs%MEKE, waves=waves)
01061     endif
01062     if (showcalltree) call calltree_waypoint("finished step_MOM_dyn_unsplit (step_MOM)")
01063
```

```
01064   endif ! ------------------------------------------------ end SPLIT
01065
01066   if (cs%thickness_diffuse .and. .not.cs%thickness_diffuse_first) then
01067     call cpu_clock_begin(id_clock_thick_diff)
01068
01069     if (cs%debug) call hchksum(h,"Pre-thickness_diffuse h", g%HI, haloshift=0, scale=gv%H_to_m)
01070
01071     if (associated(cs%VarMix)) &
01072       call calc_slope_functions(h, cs%tv, dt, g, gv, us, cs%VarMix, obc=cs%OBC)
01073     call thickness_diffuse(h, cs%uhtr, cs%vhtr, cs%tv, dt, g, gv, us, &
01074                            cs%MEKE, cs%VarMix, cs%CDp, cs%thickness_diffuse_CSp)
01075
01076     if (cs%debug) call hchksum(h,"Post-thickness_diffuse h", g%HI, haloshift=1, scale=gv%H_to_m)
01077     call cpu_clock_end(id_clock_thick_diff)
01078     call pass_var(h, g%Domain, clock=id_clock_pass, halo=max(2,cs%cont_stencil))
01079     if (showcalltree) call calltree_waypoint("finished thickness_diffuse (step_MOM)")
01080   endif
01081
01082   ! apply the submesoscale mixed layer restratification parameterization
01083   if (cs%mixedlayer_restrat) then
01084     if (cs%debug) then
01085       call hchksum(h,"Pre-mixedlayer_restrat h", g%HI, haloshift=1, scale=gv%H_to_m)
01086       call uvchksum("Pre-mixedlayer_restrat uhtr", &
01087                     cs%uhtr, cs%vhtr, g%HI, haloshift=0, scale=gv%H_to_m*us%L_to_m**2)
01088     endif
01089     call cpu_clock_begin(id_clock_ml_restrat)
01090     call mixedlayer_restrat(h, cs%uhtr, cs%vhtr, cs%tv, forces, dt, cs%visc%MLD, &
01091                             cs%VarMix, g, gv, us, cs%mixedlayer_restrat_CSp)
01092     call cpu_clock_end(id_clock_ml_restrat)
01093     call pass_var(h, g%Domain, clock=id_clock_pass, halo=max(2,cs%cont_stencil))
01094     if (cs%debug) then
01095       call hchksum(h,"Post-mixedlayer_restrat h", g%HI, haloshift=1, scale=gv%H_to_m)
01096       call uvchksum("Post-mixedlayer_restrat [uv]htr", &
01097                     cs%uhtr, cs%vhtr, g%HI, haloshift=0, scale=gv%H_to_m*us%L_to_m**2)
01098     endif
01099   endif
01100
01101   ! Whenever thickness changes let the diag manager know, target grids
01102   ! for vertical remapping may need to be regenerated.
01103   call diag_update_remap_grids(cs%diag)
01104
01105   if (cs%useMEKE) call step_forward_meke(cs%MEKE, h, cs%VarMix%SN_u, cs%VarMix%SN_v, &
01106                                          cs%visc, dt, g, gv, us, cs%MEKE_CSp, cs%uhtr, cs%vhtr)
01107   call disable_averaging(cs%diag)
01108
01109   ! Advance the dynamics time by dt.
01110   cs%t_dyn_rel_adv = cs%t_dyn_rel_adv + dt
01111   cs%t_dyn_rel_thermo = cs%t_dyn_rel_thermo + dt
01112   if (abs(cs%t_dyn_rel_thermo) < 1e-6*dt) cs%t_dyn_rel_thermo = 0.0
01113   cs%t_dyn_rel_diag = cs%t_dyn_rel_diag + dt
01114
01115   call cpu_clock_end(id_clock_dynamics)
01116
01117   call cpu_clock_begin(id_clock_other) ; call cpu_clock_begin(id_clock_diagnostics)
01118   call enable_averages(dt, time_local, cs%diag)
01119   ! These diagnostics are available after every time dynamics step.
01120   if (ids%id_u > 0) call post_data(ids%id_u, u, cs%diag)
01121   if (ids%id_v > 0) call post_data(ids%id_v, v, cs%diag)
01122   if (ids%id_h > 0) call post_data(ids%id_h, h, cs%diag)
01123   call disable_averaging(cs%diag)
01124   call cpu_clock_end(id_clock_diagnostics) ; call cpu_clock_end(id_clock_other)
01125
```

42.1.5.15 step_mom_thermo()

```
subroutine mom::step_mom_thermo (
             type(mom_control_struct), intent(inout) CS,
             type(ocean_grid_type), intent(inout) G,
             type(verticalgrid_type), intent(inout) GV,
             type(unit_scale_type), intent(in) US,
             real, dimension(szib_(g),szj_(g),szk_(g)), intent(inout) u,
             real, dimension(szi_(g),szjb_(g),szk_(g)), intent(inout) v,
             real, dimension(szi_(g),szj_(g),szk_(g)), intent(inout) h,
             type(thermo_var_ptrs), intent(inout) tv,
```

```
type(forcing), intent(inout) fluxes,
real, intent(in) dtdia,
type(time_type), intent(in) Time_end_thermo,
logical, intent(in) update_BBL,
type(wave_parameters_cs), optional, pointer Waves )  [private]
```

MOM_step_thermo orchestrates the thermodynamic time stepping and vertical remapping, via calls to diabatic (or adiabatic) and ALE_main.

Parameters

in,out	*cs*	Master MOM control structure
in,out	*g*	ocean grid structure
in,out	*gv*	ocean vertical grid structure
in	*us*	A dimensional unit scaling type
in,out	*u*	zonal velocity [L T-1 ~> m s-1]
in,out	*v*	meridional velocity [L T-1 ~> m s-1]
in,out	*h*	layer thickness [H ~> m or kg m-2]
in,out	*tv*	A structure pointing to various thermodynamic variables
in,out	*fluxes*	pointers to forcing fields
in	*dtdia*	The time interval over which to advance [T ~> s]
in	*time_end_thermo*	End of averaging interval for thermo diags
in	*update_bbl*	If true, calculate the bottom boundary layer properties.
	waves	Container for wave related parameters

Definition at line 1213 of file MOM.F90.

References id_clock_adiabatic, id_clock_ale, id_clock_bbl_visc, id_clock_diabatic, id_clock_pass, and id_clock_thermo.

Referenced by step_mom().

```
01213   type(mom_control_struct), intent(inout) :: cs     !< Master MOM control structure
01214   type(ocean_grid_type),    intent(inout) :: g      !< ocean grid structure
01215   type(verticalgrid_type),  intent(inout) :: gv     !< ocean vertical grid structure
01216   type(unit_scale_type),    intent(in)    :: us     !< A dimensional unit scaling type
01217   real, dimension(SZIB_(G),SZJ_(G),SZK_(G)), &
01218                             intent(inout) :: u      !< zonal velocity [L T-1 ~> m s-1]
01219   real, dimension(SZI_(G),SZJB_(G),SZK_(G)), &
01220                             intent(inout) :: v      !< meridional velocity [L T-1 ~> m s-1]
01221   real, dimension(SZI_(G),SZJ_(G),SZK_(G)),  &
01222                             intent(inout) :: h      !< layer thickness [H ~> m or kg m-2]
01223   type(thermo_var_ptrs),    intent(inout) :: tv     !< A structure pointing to various thermodynamic
      variables
01224   type(forcing),            intent(inout) :: fluxes !< pointers to forcing fields
01225   real,                     intent(in)    :: dtdia  !< The time interval over which to advance [T ~> s]
01226   type(time_type),          intent(in)    :: time_end_thermo !< End of averaging interval for thermo diags
01227   logical,                  intent(in)    :: update_bbl !< If true, calculate the bottom boundary layer
      properties.
01228   type(wave_parameters_cs), &
01229                  optional, pointer        :: waves  !< Container for wave related parameters
01230                                                    !! the fields in Waves are intent in here.
01231
01232   logical :: use_ice_shelf ! Needed for selecting the right ALE interface.
01233   logical :: showcalltree
01234   type(group_pass_type) :: pass_t_s, pass_t_s_h, pass_uv_t_s_h
01235   integer :: dynamics_stencil  ! The computational stencil for the calculations
01236                                ! in the dynamic core.
01237   integer :: halo_sz ! The size of a halo where data must be valid.
01238   integer :: i, j, k, is, ie, js, je, nz
01239
01240   is = g%isc ; ie = g%iec ; js = g%jsc ; je = g%jec ; nz = g%ke
01241   showcalltree = calltree_showquery()
01242   if (showcalltree) call calltree_enter("step_MOM_thermo(), MOM.F90")
01243
```

```
01244   use_ice_shelf = .false.
01245   if (associated(fluxes%frac_shelf_h)) use_ice_shelf = .true.
01246
01247   call enable_averages(dtdia, time_end_thermo, cs%diag)
01248
01249   if (associated(cs%odaCS)) then
01250     call apply_oda_tracer_increments(us%T_to_s*dtdia,g,tv,h,cs%odaCS)
01251   endif
01252
01253   if (associated(fluxes%p_surf) .or. associated(fluxes%p_surf_full)) then
01254     call extract_diabatic_member(cs%diabatic_CSp, diabatic_halo=halo_sz)
01255     if (halo_sz > 0) then
01256       if (associated(fluxes%p_surf_full)) &
01257         call pass_var(fluxes%p_surf_full, g%Domain, &
01258                       clock=id_clock_pass, halo=halo_sz, complete=.not.associated(fluxes%p_surf))
01259       call pass_var(fluxes%p_surf, g%Domain, clock=id_clock_pass, halo=halo_sz, complete=.true.)
01260     endif
01261   endif
01262
01263   if (update_bbl) then
01264     !   Calculate the BBL properties and store them inside visc (u,h).
01265     ! This is here so that CS%visc is updated before diabatic() when
01266     ! DIABATIC_FIRST=True. Otherwise diabatic() is called after the dynamics
01267     ! and set_viscous_BBL is called as a part of the dynamic stepping.
01268     call cpu_clock_begin(id_clock_bbl_visc)
01269     call set_viscous_bbl(u, v, h, tv, cs%visc, g, gv, us, cs%set_visc_CSp, symmetrize=.true.)
01270     call cpu_clock_end(id_clock_bbl_visc)
01271     if (showcalltree) call calltree_waypoint("done with set_viscous_BBL (step_MOM_thermo)")
01272   endif
01273
01274   call cpu_clock_begin(id_clock_thermo)
01275   if (.not.cs%adiabatic) then
01276     if (cs%debug) then
01277       call uvchksum("Pre-diabatic [uv]", u, v, g%HI, haloshift=2, scale=us%L_T_to_m_s)
01278       call hchksum(h,"Pre-diabatic h", g%HI, haloshift=1, scale=gv%H_to_m)
01279       call uvchksum("Pre-diabatic [uv]h", cs%uhtr, cs%vhtr, g%HI, &
01280                     haloshift=0, scale=gv%H_to_m*us%L_to_m**2)
01281     ! call MOM_state_chksum("Pre-diabatic ", u, v, h, CS%uhtr, CS%vhtr, G, GV, vel_scale=1.0)
01282       call mom_thermo_chksum("Pre-diabatic ", tv, g, us, haloshift=0)
01283       call check_redundant("Pre-diabatic ", u, v, g)
01284       call mom_forcing_chksum("Pre-diabatic", fluxes, g, us, haloshift=0)
01285     endif
01286
01287     call cpu_clock_begin(id_clock_diabatic)
01288
01289     call diabatic(u, v, h, tv, cs%Hml, fluxes, cs%visc, cs%ADp, cs%CDp, dtdia, &
01290                   time_end_thermo, g, gv, us, cs%diabatic_CSp, obc=cs%OBC, waves=waves)
01291     fluxes%fluxes_used = .true.
01292
01293     if (showcalltree) call calltree_waypoint("finished diabatic (step_MOM_thermo)")
01294
01295     ! Regridding/remapping is done here, at end of thermodynamics time step
01296     ! (that may comprise several dynamical time steps)
01297     ! The routine 'ALE_main' can be found in 'MOM_ALE.F90'.
01298     if ( cs%use_ALE_algorithm ) then
01299       call enable_averages(dtdia, time_end_thermo, cs%diag)
01300 !       call pass_vector(u, v, G%Domain)
01301       call cpu_clock_begin(id_clock_pass)
01302       if (associated(tv%T)) &
01303         call create_group_pass(pass_t_s_h, tv%T, g%Domain, to_all+omit_corners, halo=1)
01304       if (associated(tv%S)) &
01305         call create_group_pass(pass_t_s_h, tv%S, g%Domain, to_all+omit_corners, halo=1)
01306       call create_group_pass(pass_t_s_h, h, g%Domain, to_all+omit_corners, halo=1)
01307       call do_group_pass(pass_t_s_h, g%Domain)
01308       call cpu_clock_end(id_clock_pass)
01309
01310       call preale_tracer_diagnostics(cs%tracer_Reg, g, gv)
01311
01312       if (cs%debug) then
01313         call mom_state_chksum("Pre-ALE ", u, v, h, cs%uh, cs%vh, g, gv, us)
01314         call hchksum(tv%T,"Pre-ALE T", g%HI, haloshift=1)
01315         call hchksum(tv%S,"Pre-ALE S", g%HI, haloshift=1)
01316         call check_redundant("Pre-ALE ", u, v, g)
01317       endif
01318       call cpu_clock_begin(id_clock_ale)
01319       if (use_ice_shelf) then
01320         call ale_main(g, gv, us, h, u, v, tv, cs%tracer_Reg, cs%ALE_CSp, cs%OBC, &
01321                       dtdia, fluxes%frac_shelf_h)
01322       else
01323         call ale_main(g, gv, us, h, u, v, tv, cs%tracer_Reg, cs%ALE_CSp, cs%OBC, dtdia)
01324       endif
01325
01326       if (showcalltree) call calltree_waypoint("finished ALE_main (step_MOM_thermo)")
01327       call cpu_clock_end(id_clock_ale)
01328     endif   ! endif for the block "if ( CS%use_ALE_algorithm )"
01329
01330     dynamics_stencil = min(3, g%Domain%nihalo, g%Domain%njhalo)
```

```
01331     call create_group_pass(pass_uv_t_s_h, u, v, g%Domain, halo=dynamics_stencil)
01332     if (associated(tv%T)) &
01333       call create_group_pass(pass_uv_t_s_h, tv%T, g%Domain, halo=dynamics_stencil)
01334     if (associated(tv%S)) &
01335       call create_group_pass(pass_uv_t_s_h, tv%S, g%Domain, halo=dynamics_stencil)
01336     call create_group_pass(pass_uv_t_s_h, h, g%Domain, halo=dynamics_stencil)
01337     call do_group_pass(pass_uv_t_s_h, g%Domain, clock=id_clock_pass)
01338
01339     if (cs%debug .and. cs%use_ALE_algorithm) then
01340       call mom_state_chksum("Post-ALE ", u, v, h, cs%uh, cs%vh, g, gv, us)
01341       call hchksum(tv%T, "Post-ALE T", g%HI, haloshift=1)
01342       call hchksum(tv%S, "Post-ALE S", g%HI, haloshift=1)
01343       call check_redundant("Post-ALE ", u, v, g)
01344     endif
01345
01346     ! Whenever thickness changes let the diag manager know, target grids
01347     ! for vertical remapping may need to be regenerated. This needs to
01348     ! happen after the H update and before the next post_data.
01349     call diag_update_remap_grids(cs%diag)
01350
01351     !### Consider moving this up into the if ALE block.
01352     call postale_tracer_diagnostics(cs%tracer_Reg, g, gv, cs%diag, dtdia)
01353
01354     if (cs%debug) then
01355       call uvchksum("Post-diabatic u", u, v, g%HI, haloshift=2, scale=us%L_T_to_m_s)
01356       call hchksum(h, "Post-diabatic h", g%HI, haloshift=1, scale=gv%H_to_m)
01357       call uvchksum("Post-diabatic [uv]h", cs%uhtr, cs%vhtr, g%HI, &
01358                     haloshift=0, scale=gv%H_to_m*us%L_to_m**2)
01359     ! call MOM_state_chksum("Post-diabatic ", u, v, &
01360     !                       h, CS%uhtr, CS%vhtr, G, GV, haloshift=1)
01361       if (associated(tv%T)) call hchksum(tv%T, "Post-diabatic T", g%HI, haloshift=1)
01362       if (associated(tv%S)) call hchksum(tv%S, "Post-diabatic S", g%HI, haloshift=1)
01363       if (associated(tv%frazil)) call hchksum(tv%frazil, "Post-diabatic frazil", g%HI, haloshift=0, &
01364                                               scale=g%US%Q_to_J_kg*g%US%RZ_to_kg_m2)
01365       if (associated(tv%salt_deficit)) call hchksum(tv%salt_deficit, &
01366                                 "Post-diabatic salt deficit", g%HI, haloshift=0, scale=us%RZ_to_kg_m2)
01367     ! call MOM_thermo_chksum("Post-diabatic ", tv, G, US)
01368       call check_redundant("Post-diabatic ", u, v, g)
01369     endif
01370     call disable_averaging(cs%diag)
01371
01372     call cpu_clock_end(id_clock_diabatic)
01373   else   ! complement of "if (.not.CS%adiabatic)"
01374
01375     call cpu_clock_begin(id_clock_adiabatic)
01376     call adiabatic(h, tv, fluxes, dtdia, g, gv, us, cs%diabatic_CSp)
01377     fluxes%fluxes_used = .true.
01378     call cpu_clock_end(id_clock_adiabatic)
01379
01380     if (associated(tv%T)) then
01381       call create_group_pass(pass_t_s, tv%T, g%Domain, to_all+omit_corners, halo=1)
01382       call create_group_pass(pass_t_s, tv%S, g%Domain, to_all+omit_corners, halo=1)
01383       call do_group_pass(pass_t_s, g%Domain, clock=id_clock_pass)
01384       if (cs%debug) then
01385         if (associated(tv%T)) call hchksum(tv%T, "Post-diabatic T", g%HI, haloshift=1)
01386         if (associated(tv%S)) call hchksum(tv%S, "Post-diabatic S", g%HI, haloshift=1)
01387       endif
01388     endif
01389
01390   endif   ! endif for the block "if (.not.CS%adiabatic)"
01391   call cpu_clock_end(id_clock_thermo)
01392
01393   call disable_averaging(cs%diag)
01394
01395   if (showcalltree) call calltree_leave("step_MOM_thermo(), MOM.F90")
01396
```

42.1.5.16 step_mom_tracer_dyn()

```
subroutine mom::step_mom_tracer_dyn (
             type(mom_control_struct), intent(inout) CS,
             type(ocean_grid_type), intent(inout) G,
             type(verticalgrid_type), intent(in) GV,
             type(unit_scale_type), intent(in) US,
             real, dimension( g %isd:  g %ied, g %jsd:  g %jed, g %ke), intent(in) h,
             type(time_type), intent(in) Time_local )  [private]
```

step_MOM_tracer_dyn does tracer advection and lateral diffusion, bringing the tracers up to date with the changes in state due to the dynamics. Surface sources and sinks and remapping are handled via step_MOM_thermo.

Parameters

in,out	*cs*	control structure
in,out	*g*	ocean grid structure
in	*gv*	ocean vertical grid structure
in	*us*	A dimensional unit scaling type
in	*h*	layer thicknesses after the transports [H ~> m or kg m-2]
in	*time_local*	The model time at the end of the time step.

Definition at line 1132 of file MOM.F90.

References id_clock_diagnostics, id_clock_other, id_clock_pass, id_clock_thermo, and id_clock_tracer.

Referenced by step_mom().

```
01132   type(mom_control_struct), intent(inout) :: cs     !< control structure
01133   type(ocean_grid_type),    intent(inout) :: g      !< ocean grid structure
01134   type(verticalgrid_type),  intent(in)    :: gv     !< ocean vertical grid structure
01135   type(unit_scale_type),    intent(in)    :: us     !< A dimensional unit scaling type
01136   real, dimension(SZI_(G),SZJ_(G),SZK_(G)),  &
01137                             intent(in)    :: h      !< layer thicknesses after the transports [H ~> m or kg
      m-2]
01138   type(time_type),          intent(in)    :: time_local !< The model time at the end
01139                                                     !! of the time step.
01140   type(group_pass_type) :: pass_t_s
01141   integer :: halo_sz ! The size of a halo where data must be valid.
01142   logical :: showcalltree
01143   showcalltree = calltree_showquery()
01144
01145   if (cs%debug) then
01146     call cpu_clock_begin(id_clock_other)
01147     call hchksum(h,"Pre-advection h", g%HI, haloshift=1, scale=gv%H_to_m)
01148     call uvchksum("Pre-advection uhtr", cs%uhtr, cs%vhtr, g%HI, &
01149                   haloshift=0, scale=gv%H_to_m*us%L_to_m**2)
01150     if (associated(cs%tv%T)) call hchksum(cs%tv%T, "Pre-advection T", g%HI, haloshift=1)
01151     if (associated(cs%tv%S)) call hchksum(cs%tv%S, "Pre-advection S", g%HI, haloshift=1)
01152     if (associated(cs%tv%frazil)) call hchksum(cs%tv%frazil, "Pre-advection frazil", g%HI, haloshift=0, &
01153                                                scale=g%US%Q_to_J_kg*g%US%RZ_to_kg_m2)
01154     if (associated(cs%tv%salt_deficit)) call hchksum(cs%tv%salt_deficit, &
01155                    "Pre-advection salt deficit", g%HI, haloshift=0, scale=us%RZ_to_kg_m2)
01156   ! call MOM_thermo_chksum("Pre-advection ", CS%tv, G, US)
01157     call cpu_clock_end(id_clock_other)
01158   endif
01159
01160   call cpu_clock_begin(id_clock_thermo) ; call cpu_clock_begin(id_clock_tracer)
01161   call enable_averages(cs%t_dyn_rel_adv, time_local, cs%diag)
01162
01163   call advect_tracer(h, cs%uhtr, cs%vhtr, cs%OBC, cs%t_dyn_rel_adv, g, gv, us, &
01164                      cs%tracer_adv_CSp, cs%tracer_Reg)
01165   call tracer_hordiff(h, cs%t_dyn_rel_adv, cs%MEKE, cs%VarMix, g, gv, us, &
01166                       cs%tracer_diff_CSp, cs%tracer_Reg, cs%tv)
01167   if (showcalltree) call calltree_waypoint("finished tracer advection/diffusion (step_MOM)")
01168   call update_segment_tracer_reservoirs(g, gv, cs%uhtr, cs%vhtr, h, cs%OBC, &
01169                      cs%t_dyn_rel_adv, cs%tracer_Reg)
01170   call cpu_clock_end(id_clock_tracer) ; call cpu_clock_end(id_clock_thermo)
01171
01172   call cpu_clock_begin(id_clock_other) ; call cpu_clock_begin(id_clock_diagnostics)
01173   call post_transport_diagnostics(g, gv, us, cs%uhtr, cs%vhtr, h, cs%transport_IDs, &
01174            cs%diag_pre_dyn, cs%diag, cs%t_dyn_rel_adv, cs%tracer_reg)
01175   ! Rebuild the remap grids now that we've posted the fields which rely on thicknesses
01176   ! from before the dynamics calls
01177   call diag_update_remap_grids(cs%diag)
01178
01179   call disable_averaging(cs%diag)
01180   call cpu_clock_end(id_clock_diagnostics) ; call cpu_clock_end(id_clock_other)
01181
01182   ! Reset the accumulated transports to 0 and record that the dynamics
01183   ! and advective times now agree.
01184   call cpu_clock_begin(id_clock_thermo) ; call cpu_clock_begin(id_clock_tracer)
01185   cs%uhtr(:,:,:) = 0.0
01186   cs%vhtr(:,:,:) = 0.0
```

```
01187   cs%t_dyn_rel_adv = 0.0
01188   call cpu_clock_end(id_clock_tracer) ; call cpu_clock_end(id_clock_thermo)
01189
01190   if (associated(cs%tv%T)) then
01191     call extract_diabatic_member(cs%diabatic_CSp, diabatic_halo=halo_sz)
01192     if (halo_sz > 0) then
01193       call create_group_pass(pass_t_s, cs%tv%T, g%Domain, to_all, halo=halo_sz)
01194       call create_group_pass(pass_t_s, cs%tv%S, g%Domain, to_all, halo=halo_sz)
01195       call do_group_pass(pass_t_s, g%Domain, clock=id_clock_pass)
01196     elseif (cs%diabatic_first) then
01197       ! Temperature and salinity need halo updates because they will be used
01198       ! in the dynamics before they are changed again.
01199       call create_group_pass(pass_t_s, cs%tv%T, g%Domain, to_all+omit_corners, halo=1)
01200       call create_group_pass(pass_t_s, cs%tv%S, g%Domain, to_all+omit_corners, halo=1)
01201       call do_group_pass(pass_t_s, g%Domain, clock=id_clock_pass)
01202     endif
01203   endif
01204
01205   cs%preadv_h_stored = .false.
01206
```

42.1.5.17 step_offline()

```
subroutine, public mom::step_offline (
             type(mech_forcing), intent(in) forces,
             type(forcing), intent(inout) fluxes,
             type(surface), intent(inout) sfc_state,
             type(time_type), intent(in) Time_start,
             real, intent(in) time_interval,
             type(mom_control_struct), pointer CS )
```

step_offline is the main driver for running tracers offline in MOM6. This has been primarily developed with ALE configurations in mind. Some work has been done in isopycnal configuration, but the work is very preliminary. Some more detail about this capability along with some of the subroutines called here can be found in tracers/M↩ OM_offline_control.F90

Parameters

in	*forces*	A structure with the driving mechanical forces
in,out	*fluxes*	pointers to forcing fields
in,out	*sfc_state*	surface ocean state
in	*time_start*	starting time of a segment, as a time type
in	*time_interval*	time interval
	cs	control structure from initialize_MOM

Definition at line 1405 of file MOM.F90.

References adjust_ssh_for_p_atm(), extract_surface_state(), id_clock_ale, and id_clock_offline_tracer.

```
01405   type(mech_forcing), intent(in)    :: forces        !< A structure with the driving mechanical forces
01406   type(forcing),      intent(inout) :: fluxes        !< pointers to forcing fields
01407   type(surface),      intent(inout) :: sfc_state     !< surface ocean state
01408   type(time_type),    intent(in)    :: time_start    !< starting time of a segment, as a time type
01409   real,               intent(in)    :: time_interval !< time interval
01410   type(mom_control_struct), pointer :: cs            !< control structure from initialize_MOM
01411
01412   ! Local pointers
01413   type(ocean_grid_type),      pointer :: g  => null() ! Pointer to a structure containing
01414                                                       ! metrics and related information
01415   type(verticalgrid_type),    pointer :: gv => null() ! Pointer to structure containing information
01416                                                       ! about the vertical grid
```

```
01417   type(unit_scale_type),      pointer :: us => null() ! Pointer to a structure containing
01418                                                       ! various unit conversion factors
01419
01420   logical :: first_iter    !< True if this is the first time step_offline has been called in a given
      interval
01421   logical :: last_iter     !< True if this is the last time step_tracer is to be called in an offline
      interval
01422   logical :: do_vertical   !< If enough time has elapsed, do the diabatic tracer sources/sinks
01423   logical :: adv_converged !< True if all the horizontal fluxes have been used
01424
01425   real :: dt_offline          ! The offline timestep for advection [T ~> s]
01426   real :: dt_offline_vertical ! The offline timestep for vertical fluxes and remapping [T ~> s]
01427   logical :: skip_diffusion
01428   integer :: id_eta_diff_end
01429
01430   type(time_type), pointer :: accumulated_time => null()
01431   type(time_type), pointer :: vertical_time => null()
01432   integer :: i,j,k
01433   integer :: is, ie, js, je, isd, ied, jsd, jed
01434
01435   ! 3D pointers
01436   real, dimension(:,:,:), pointer :: &
01437     uhtr => null(), vhtr => null(), &
01438     eatr => null(), ebtr => null(), &
01439     h_end => null()
01440
01441   ! 2D Array for diagnostics
01442   real, dimension(SZI_(CS%G),SZJ_(CS%G)) :: eta_pre, eta_end
01443   type(time_type) :: time_end    ! End time of a segment, as a time type
01444
01445   ! Grid-related pointer assignments
01446   g => cs%G ; gv => cs%GV ; us => cs%US
01447
01448   is  = g%isc  ; ie  = g%iec  ; js  = g%jsc  ; je  = g%jec
01449   isd = g%isd  ; ied = g%ied  ; jsd = g%jsd  ; jed = g%jed
01450
01451   call cpu_clock_begin(id_clock_offline_tracer)
01452   call extract_offline_main(cs%offline_CSp, uhtr, vhtr, eatr, ebtr, h_end, accumulated_time, &
01453                             vertical_time, dt_offline, dt_offline_vertical, skip_diffusion)
01454   time_end = increment_date(time_start, seconds=floor(time_interval+0.001))
01455
01456   call enable_averaging(time_interval, time_end, cs%diag)
01457
01458   ! Check to see if this is the first iteration of the offline interval
01459   if (accumulated_time == real_to_time(0.0)) then
01460     first_iter = .true.
01461   else ! This is probably unnecessary but is used to guard against unwanted behavior
01462     first_iter = .false.
01463   endif
01464
01465   ! Check to see if vertical tracer functions should be done
01466   if (first_iter .or. (accumulated_time >= vertical_time)) then
01467     do_vertical = .true.
01468     vertical_time = accumulated_time + real_to_time(us%T_to_s*dt_offline_vertical)
01469   else
01470     do_vertical = .false.
01471   endif
01472
01473   ! Increment the amount of time elapsed since last read and check if it's time to roll around
01474   accumulated_time = accumulated_time + real_to_time(time_interval)
01475
01476   last_iter = (accumulated_time >= real_to_time(us%T_to_s*dt_offline))
01477
01478   if (cs%use_ALE_algorithm) then
01479     ! If this is the first iteration in the offline timestep, then we need to read in fields and
01480     ! perform the main advection.
01481     if (first_iter) then
01482       call mom_mesg("Reading in new offline fields")
01483       ! Read in new transport and other fields
01484       ! call update_transport_from_files(G, GV, CS%offline_CSp, h_end, eatr, ebtr, uhtr, vhtr, &
01485       !     CS%tv%T, CS%tv%S, fluxes, CS%use_ALE_algorithm)
01486       ! call update_transport_from_arrays(CS%offline_CSp)
01487       call update_offline_fields(cs%offline_CSp, cs%h, fluxes, cs%use_ALE_algorithm)
01488
01489       ! Apply any fluxes into the ocean
01490       call offline_fw_fluxes_into_ocean(g, gv, cs%offline_CSp, fluxes, cs%h)
01491
01492       if (.not.cs%diabatic_first) then
01493         call offline_advection_ale(fluxes, time_start, time_interval, cs%offline_CSp, id_clock_ale, &
01494             cs%h, uhtr, vhtr, converged=adv_converged)
01495
01496         ! Redistribute any remaining transport
01497         call offline_redistribute_residual(cs%offline_CSp, cs%h, uhtr, vhtr, adv_converged)
01498
01499         ! Perform offline diffusion if requested
01500         if (.not. skip_diffusion) then
01501           if (associated(cs%VarMix)) then
```

```
01502             call pass_var(cs%h, g%Domain)
01503             call calc_resoln_function(cs%h, cs%tv, g, gv, us, cs%VarMix)
01504             call calc_depth_function(g, cs%VarMix)
01505             call calc_slope_functions(cs%h, cs%tv, dt_offline, g, gv, us, cs%VarMix, obc=cs%OBC)
01506           endif
01507           call tracer_hordiff(cs%h, dt_offline, cs%MEKE, cs%VarMix, g, gv, us, &
01508               cs%tracer_diff_CSp, cs%tracer_Reg, cs%tv)
01509         endif
01510       endif
01511     endif
01512     ! The functions related to column physics of tracers is performed separately in ALE mode
01513     if (do_vertical) then
01514       call offline_diabatic_ale(fluxes, time_start, time_end, cs%offline_CSp, cs%h, eatr, ebtr)
01515     endif
01516
01517     ! Last thing that needs to be done is the final ALE remapping
01518     if (last_iter) then
01519       if (cs%diabatic_first) then
01520         call offline_advection_ale(fluxes, time_start, time_interval, cs%offline_CSp, id_clock_ale, &
01521             cs%h, uhtr, vhtr, converged=adv_converged)
01522
01523         ! Redistribute any remaining transport and perform the remaining advection
01524         call offline_redistribute_residual(cs%offline_CSp, cs%h, uhtr, vhtr, adv_converged)
01525               ! Perform offline diffusion if requested
01526         if (.not. skip_diffusion) then
01527           if (associated(cs%VarMix)) then
01528             call pass_var(cs%h, g%Domain)
01529             call calc_resoln_function(cs%h, cs%tv, g, gv, us, cs%VarMix)
01530             call calc_depth_function(g, cs%VarMix)
01531             call calc_slope_functions(cs%h, cs%tv, dt_offline, g, gv, us, cs%VarMix, obc=cs%OBC)
01532           endif
01533           call tracer_hordiff(cs%h, dt_offline, cs%MEKE, cs%VarMix, g, gv, us, &
01534               cs%tracer_diff_CSp, cs%tracer_Reg, cs%tv)
01535         endif
01536       endif
01537
01538       call mom_mesg("Last iteration of offline interval")
01539
01540       ! Apply freshwater fluxes out of the ocean
01541       call offline_fw_fluxes_out_ocean(g, gv, cs%offline_CSp, fluxes, cs%h)
01542       ! These diagnostic can be used to identify which grid points did not converge within
01543       ! the specified number of advection sub iterations
01544       call post_offline_convergence_diags(cs%offline_CSp, cs%h, h_end, uhtr, vhtr)
01545
01546       ! Call ALE one last time to make sure that tracers are remapped onto the layer thicknesses
01547       ! stored from the forward run
01548       call cpu_clock_begin(id_clock_ale)
01549       call ale_offline_tracer_final( g, gv, cs%h, cs%tv, h_end, cs%tracer_Reg, cs%ALE_CSp, cs%OBC)
01550       call cpu_clock_end(id_clock_ale)
01551       call pass_var(cs%h, g%Domain)
01552     endif
01553   else ! NON-ALE MODE...NOT WELL TESTED
01554     call mom_error(warning, &
01555         "Offline tracer mode in non-ALE configuration has not been thoroughly tested")
01556     ! Note that for the layer mode case, the calls to tracer sources and sinks is embedded in
01557     ! main_offline_advection_layer. Warning: this may not be appropriate for tracers that
01558     ! exchange with the atmosphere
01559     if (abs(time_interval - us%T_to_s*dt_offline) > 1.0e-6) then
01560       call mom_error(fatal, &
01561           "For offline tracer mode in a non-ALE configuration, dt_offline must equal time_interval")
01562     endif
01563     call update_offline_fields(cs%offline_CSp, cs%h, fluxes, cs%use_ALE_algorithm)
01564     call offline_advection_layer(fluxes, time_start, time_interval, cs%offline_CSp, &
01565         cs%h, eatr, ebtr, uhtr, vhtr)
01566     ! Perform offline diffusion if requested
01567     if (.not. skip_diffusion) then
01568       call tracer_hordiff(h_end, dt_offline, cs%MEKE, cs%VarMix, g, gv, us, &
01569                           cs%tracer_diff_CSp, cs%tracer_Reg, cs%tv)
01570     endif
01571
01572     cs%h = h_end
01573
01574     call pass_var(cs%tv%T, g%Domain)
01575     call pass_var(cs%tv%S, g%Domain)
01576     call pass_var(cs%h, g%Domain)
01577
01578   endif
01579
01580   call adjust_ssh_for_p_atm(cs%tv, g, gv, us, cs%ave_ssh_ibc, forces%p_surf_SSH, &
01581                             cs%calc_rho_for_sea_lev)
01582   call extract_surface_state(cs, sfc_state)
01583
01584   call disable_averaging(cs%diag)
01585   call pass_var(cs%tv%T, g%Domain)
01586   call pass_var(cs%tv%S, g%Domain)
01587   call pass_var(cs%h, g%Domain)
01588
```

```
01589   fluxes%fluxes_used = .true.
01590
01591   if (last_iter) then
01592     accumulated_time = real_to_time(0.0)
01593   endif
01594
01595   call cpu_clock_end(id_clock_offline_tracer)
01596
```

42.1.6 Variable Documentation

42.1.6.1 id_clock_adiabatic

```
integer mom::id_clock_adiabatic  [private]
```

CPU time clock IDs.

Definition at line 395 of file MOM.F90.

Referenced by mom_timing_init(), and step_mom_thermo().

```
00395 integer :: id_clock_adiabatic
```

42.1.6.2 id_clock_ale

```
integer mom::id_clock_ale  [private]
```

CPU time clock IDs.

Definition at line 406 of file MOM.F90.

Referenced by mom_timing_init(), step_mom_thermo(), and step_offline().

```
00406 integer :: id_clock_ale
```

42.1.6.3 id_clock_bbl_visc

```
integer mom::id_clock_bbl_visc  [private]
```

CPU time clock IDs.

Definition at line 398 of file MOM.F90.

Referenced by mom_timing_init(), step_mom_dynamics(), and step_mom_thermo().

```
00398 integer :: id_clock_bbl_visc
```

42.1.6.4 id_clock_continuity

```
integer mom::id_clock_continuity  [private]
```

CPU time clock IDs.

Definition at line 396 of file MOM.F90.

Referenced by mom_timing_init().

```
00396 integer :: id_clock_continuity  ! also in dynamics s/r
```

42.1.6.5 id_clock_diabatic

```
integer mom::id_clock_diabatic  [private]
```

CPU time clock IDs.

Definition at line 394 of file MOM.F90.

Referenced by mom_timing_init(), and step_mom_thermo().

```
00394 integer :: id_clock_diabatic
```

42.1.6.6 id_clock_diagnostics

```
integer mom::id_clock_diagnostics  [private]
```

CPU time clock IDs.

Definition at line 400 of file MOM.F90.

Referenced by mom_timing_init(), step_mom(), step_mom_dynamics(), and step_mom_tracer_dyn().

```
00400 integer :: id_clock_diagnostics
```

42.1.6.7 id_clock_dynamics

```
integer mom::id_clock_dynamics  [private]
```

CPU time clock IDs.

Definition at line 391 of file MOM.F90.

Referenced by mom_timing_init(), step_mom(), and step_mom_dynamics().

```
00391 integer :: id_clock_dynamics
```

42.1.6.8 id_clock_init

```
integer mom::id_clock_init  [private]
```

CPU time clock IDs.

Definition at line 402 of file MOM.F90.

Referenced by finish_mom_initialization(), and initialize_mom().

```
00402 integer :: id_clock_init
```

42.1.6.9 id_clock_ml_restrat

```
integer mom::id_clock_ml_restrat  [private]
```

CPU time clock IDs.

Definition at line 399 of file MOM.F90.

Referenced by mom_timing_init(), and step_mom_dynamics().

```
00399 integer :: id_clock_ml_restrat
```

42.1.6.10 id_clock_mom_init

```
integer mom::id_clock_mom_init  [private]
```

CPU time clock IDs.

Definition at line 403 of file MOM.F90.

Referenced by initialize_mom(), and mom_timing_init().

```
00403 integer :: id_clock_mom_init
```

42.1.6.11 id_clock_ocean

```
integer mom::id_clock_ocean
```

CPU time clock IDs.

Definition at line 390 of file MOM.F90.

Referenced by mom_timing_init(), and step_mom().

```
00390 integer :: id_clock_ocean
```

42.1.6.12 id_clock_offline_tracer

```
integer mom::id_clock_offline_tracer  [private]
```

CPU time clock IDs.

Definition at line 408 of file MOM.F90.

Referenced by mom_timing_init(), and step_offline().

```
00408 integer :: id_clock_offline_tracer
```

42.1.6.13 id_clock_other

```
integer mom::id_clock_other  [private]
```

CPU time clock IDs.

Definition at line 407 of file MOM.F90.

Referenced by mom_timing_init(), step_mom(), step_mom_dynamics(), and step_mom_tracer_dyn().

```
00407 integer :: id_clock_other
```

42.1.6.14 id_clock_pass

```
integer mom::id_clock_pass  [private]
```

CPU time clock IDs.

Definition at line 404 of file MOM.F90.

Referenced by mom_timing_init(), step_mom(), step_mom_dynamics(), step_mom_thermo(), and step_mom_↩ tracer_dyn().

```
00404 integer :: id_clock_pass       ! also in dynamics d/r
```

42.1.6.15 id_clock_pass_init

```
integer mom::id_clock_pass_init  [private]
```

CPU time clock IDs.

Definition at line 405 of file MOM.F90.

Referenced by initialize_mom(), and mom_timing_init().

```
00405 integer :: id_clock_pass_init  ! also in dynamics d/r
```

42.1.6.16 id_clock_thermo

```
integer mom::id_clock_thermo  [private]
```

CPU time clock IDs.

Definition at line 392 of file MOM.F90.

Referenced by mom_timing_init(), step_mom_thermo(), and step_mom_tracer_dyn().

```
00392 integer :: id_clock_thermo
```

42.1.6.17 id_clock_thick_diff

```
integer mom::id_clock_thick_diff  [private]
```

CPU time clock IDs.

Definition at line 397 of file MOM.F90.

Referenced by mom_timing_init(), and step_mom_dynamics().

```
00397 integer :: id_clock_thick_diff
```

42.1.6.18 id_clock_tracer

```
integer mom::id_clock_tracer  [private]
```

CPU time clock IDs.

Definition at line 393 of file MOM.F90.

Referenced by mom_timing_init(), and step_mom_tracer_dyn().

```
00393 integer :: id_clock_tracer
```

42.1.6.19 id_clock_unit_tests

```
integer mom::id_clock_unit_tests  [private]
```

CPU time clock IDs.

Definition at line 409 of file MOM.F90.

Referenced by initialize_mom().

```
00409 integer :: id_clock_unit_tests
```

42.1.6.20 id_clock_z_diag

```
integer mom::id_clock_z_diag  [private]
```

CPU time clock IDs.

Definition at line 401 of file MOM.F90.

Referenced by mom_timing_init().

```
00401 integer :: id_clock_z_diag
```

42.2 mom_ale Module Reference

This module contains the main regridding routines.

Data Types

- type ale_cs

  ALE control structure.

Functions/Subroutines

- subroutine, public ale_init (param_file, GV, US, max_depth, CS)

  This routine is typically called (from initialize_MOM in file MOM.F90) before the main time integration loop to initialize the regridding stuff. We read the MOM_input file to register the values of different regridding/remapping parameters.
- subroutine, public ale_register_diags (Time, G, GV, US, diag, CS)

  Initialize diagnostics for the ALE module.
- subroutine, public adjustgridforintegrity (CS, G, GV, h)

  Crudely adjust (initial) grid for integrity. This routine is typically called (from initialize_MOM in file MOM.F90) before the main time integration loop to initialize the regridding stuff. We read the MOM_input file to register the values of different regridding/remapping parameters.
- subroutine, public ale_end (CS)

  End of regridding (memory deallocation). This routine is typically called (from MOM_end in file MOM.F90) after the main time integration loop to deallocate the regridding stuff.
- subroutine, public ale_main (G, GV, US, h, u, v, tv, Reg, CS, OBC, dt, frac_shelf_h)

  Takes care of (1) building a new grid and (2) remapping all variables between the old grid and the new grid. The creation of the new grid can be based on z coordinates, target interface densities, sigma coordinates or any arbitrary coordinate system.
- subroutine, public ale_main_offline (G, GV, h, tv, Reg, CS, OBC, dt)

  Takes care of (1) building a new grid and (2) remapping all variables between the old grid and the new grid. The creation of the new grid can be based on z coordinates, target interface densities, sigma coordinates or any arbitrary coordinate system.
- subroutine, public ale_offline_inputs (CS, G, GV, h, tv, Reg, uhtr, vhtr, Kd, debug, OBC)

  Regrid/remap stored fields used for offline tracer integrations. These input fields are assumed to have the same layer thicknesses at the end of the last offline interval (which should be a Zstar grid). This routine builds a grid on the runtime specified vertical coordinate.
- subroutine, public ale_offline_tracer_final (G, GV, h, tv, h_target, Reg, CS, OBC)

  Remaps all tracers from h onto h_target. This is intended to be called when tracers are done offline. In the case where transports don't quite conserve, we still want to make sure that layer thicknesses offline do not drift too far away from the online model.
- subroutine check_grid (G, GV, h, threshold)

  Check grid for negative thicknesses.
- subroutine, public ale_build_grid (G, GV, regridCS, remapCS, h, tv, debug, frac_shelf_h)

  Generates new grid.
- subroutine, public ale_regrid_accelerated (CS, G, GV, h, tv, n, u, v, OBC, Reg, dt, dzRegrid, initial)

  For a state-based coordinate, accelerate the process of regridding by repeatedly applying the grid calculation algorithm.
- subroutine remap_all_state_vars (CS_remapping, CS_ALE, G, GV, h_old, h_new, Reg, OBC, dxInterface, u, v, debug, dt)

  This routine takes care of remapping all variable between the old and the new grids. When velocity components need to be remapped, thicknesses at velocity points are taken to be arithmetic averages of tracer thicknesses. This routine is called during initialization of the model at time=0, to remap initiali conditions to the model grid. It is also called during a time step to update the state.
- subroutine, public ale_remap_scalar (CS, G, GV, nk_src, h_src, s_src, h_dst, s_dst, all_cells, old_remap, answers_2018)

  Remaps a single scalar between grids described by thicknesses h_src and h_dst. h_dst must be dimensioned as a model array with GVke layers while h_src can have an arbitrary number of layers specified by nk_src.
- subroutine, public ts_plm_edge_values (CS, S_t, S_b, T_t, T_b, G, GV, tv, h, bdry_extrap)

  Calculate edge values (top and bottom of layer) for T and S consistent with a PLM reconstruction in the vertical direction. Boundary reconstructions are PCM unless bdry_extrap is true.
- subroutine, public ale_plm_edge_values (CS, G, GV, h, Q, bdry_extrap, Q_t, Q_b)

  Calculate edge values (top and bottom of layer) 3d scalar array. Boundary reconstructions are PCM unless bdry_↩ extrap is true.
- subroutine, public ts_ppm_edge_values (CS, S_t, S_b, T_t, T_b, G, GV, tv, h, bdry_extrap)

Calculate edge values (top and bottom of layer) for T and S consistent with a PPM reconstruction in the vertical direction. Boundary reconstructions are PCM unless bdry_extrap is true.

- subroutine, public ale_initregridding (GV, US, max_depth, param_file, mdl, regridCS)

  Initializes regridding for the main ALE algorithm.
- real function, dimension(cs%nk+1), public ale_getcoordinate (CS)

  Query the target coordinate interfaces positions.
- character(len=20) function, public ale_getcoordinateunits (CS)

  Query the target coordinate units.
- logical function, public ale_remap_init_conds (CS)

  Returns true if initial conditions should be regridded and remapped.
- subroutine, public ale_update_regrid_weights (dt, CS)

  Updates the weights for time filtering the new grid generated in regridding.
- subroutine, public ale_updateverticalgridtype (CS, GV)

  Update the vertical grid type with ALE information. This subroutine sets information in the verticalGrid_type to be consistent with the use of ALE mode.
- subroutine, public ale_writecoordinatefile (CS, GV, directory)

  Write the vertical coordinate information into a file. This subroutine writes out a file containing any available data related to the vertical grid used by the MOM ocean model when in ALE mode.
- subroutine, public ale_initthicknesstocoord (CS, G, GV, h)

  Set h to coordinate values for fixed coordinate systems.

42.2.1 Detailed Description

This module contains the main regridding routines.

Regridding comprises two steps:

1. Interpolation and creation of a new grid based on target interface densities (or any other criterion).
2. Remapping of quantities between old grid and new grid.

Original module written by Laurent White, 2008.06.09

42.2.2 Function/Subroutine Documentation

42.2.2.1 adjustgridforintegrity()

```
subroutine, public mom_ale::adjustgridforintegrity (
             type(ale_cs), pointer CS,
             type(ocean_grid_type), intent(in) G,
             type(verticalgrid_type), intent(in) GV,
             real, dimension(szi_(g),szj_(g),szk_(gv)), intent(inout) h )
```

Crudely adjust (initial) grid for integrity. This routine is typically called (from initialize_MOM in file MOM.F90) before the main time integration loop to initialize the regridding stuff. We read the MOM_input file to register the values of different regridding/remapping parameters.

Parameters

	cs	Regridding parameters and options
in	*g*	Ocean grid informations
in	*gv*	Ocean vertical grid structure
in,out	*h*	Current 3D grid thickness that are to be adjusted [H ~> m or kg-2]

Definition at line 293 of file MOM_ALE.F90.

```
00293   type(ale_cs),                              pointer       :: cs  !< Regridding parameters and options
00294   type(ocean_grid_type),                     intent(in)    :: g   !< Ocean grid informations
00295   type(verticalgrid_type),                   intent(in)    :: gv  !< Ocean vertical grid structure
00296   real, dimension(SZI_(G),SZJ_(G),SZK_(GV)), intent(inout) :: h   !< Current 3D grid thickness that
00297                                                                     !! are to be adjusted [H ~> m or kg-2]
00298   call inflate_vanished_layers_old( cs%regridCS, g, gv, h(:,:,:) )
00299
```

42.2.2.2 ale_build_grid()

```
subroutine, public mom_ale::ale_build_grid (
              type(ocean_grid_type), intent(in) G,
              type(verticalgrid_type), intent(in) GV,
              type(regridding_cs), intent(in) regridCS,
              type(remapping_cs), intent(in) remapCS,
              real, dimension( g %isd:  g %ied, g %jsd:  g %jed, gv %ke), intent(inout) h,
              type(thermo_var_ptrs), intent(inout) tv,
              logical, intent(in), optional debug,
              real, dimension(:,:), optional, pointer frac_shelf_h )
```

Generates new grid.

Parameters

in	*g*	Ocean grid structure
in	*gv*	Ocean vertical grid structure
in	*regridcs*	Regridding parameters and options
in	*remapcs*	Remapping parameters and options
in,out	*tv*	Thermodynamical variable structure
in,out	*h*	Current 3D grid obtained after the last time step [H ~> m or kg-2]
in	*debug*	If true, show the call tree
	frac_shelf_h	Fractional ice shelf coverage

Definition at line 616 of file MOM_ALE.F90.

```
00616   type(ocean_grid_type),                     intent(in)    :: g        !< Ocean grid structure
00617   type(verticalgrid_type),                   intent(in)    :: gv       !< Ocean vertical grid structure
00618   type(regridding_cs),                       intent(in)    :: regridcs !< Regridding parameters and options
00619   type(remapping_cs),                        intent(in)    :: remapcs  !< Remapping parameters and options
00620   type(thermo_var_ptrs),                     intent(inout) :: tv       !< Thermodynamical variable structure
00621   real, dimension(SZI_(G),SZJ_(G), SZK_(GV)), intent(inout) :: h        !< Current 3D grid obtained after the
```

```
00622                                                                    !! last time step [H ~> m or kg-2]
00623   logical,                        optional, intent(in)     :: debug    !< If true, show the call tree
00624   real, dimension(:,:),           optional, pointer        :: frac_shelf_h !< Fractional ice shelf coverage
00625   ! Local variables
00626   integer :: nk, i, j, k
00627   real, dimension(SZI_(G), SZJ_(G), SZK_(GV)+1) :: dzregrid ! The change in grid interface positions
00628   real, dimension(SZI_(G), SZJ_(G), SZK_(GV)) :: h_new ! The new grid thicknesses
00629   logical :: show_call_tree, use_ice_shelf
00630
00631   show_call_tree = .false.
00632   if (present(debug)) show_call_tree = debug
00633   if (show_call_tree) call calltree_enter("ALE_build_grid(), MOM_ALE.F90")
00634   use_ice_shelf = .false.
00635   if (present(frac_shelf_h)) then
00636     if (associated(frac_shelf_h)) use_ice_shelf = .true.
00637   endif
00638
00639   ! Build new grid. The new grid is stored in h_new. The old grid is h.
00640   ! Both are needed for the subsequent remapping of variables.
00641   if (use_ice_shelf) then
00642      call regridding_main( remapcs, regridcs, g, gv, h, tv, h_new, dzregrid, frac_shelf_h )
00643   else
00644      call regridding_main( remapcs, regridcs, g, gv, h, tv, h_new, dzregrid )
00645   endif
00646
00647   ! Override old grid with new one. The new grid 'h_new' is built in
00648   ! one of the 'build_...' routines above.
00649 !$OMP parallel do default(none) shared(G,h,h_new)
00650   do j = g%jsc,g%jec ; do i = g%isc,g%iec
00651     if (g%mask2dT(i,j)>0.) h(i,j,:) = h_new(i,j,:)
00652   enddo ; enddo
00653
00654   if (show_call_tree) call calltree_leave("ALE_build_grid()")
```

42.2.2.3 ale_end()

```
subroutine, public mom_ale::ale_end (
              type(ale_cs), pointer CS )
```

End of regridding (memory deallocation). This routine is typically called (from MOM_end in file MOM.F90) after the main time integration loop to deallocate the regridding stuff.

Parameters

cs	module control structure

Definition at line 307 of file MOM_ALE.F90.

```
00307   type(ale_cs), pointer :: cs  !< module control structure
00308
00309   ! Deallocate memory used for the regridding
00310   call end_remapping( cs%remapCS )
00311   call end_regridding( cs%regridCS )
00312
00313   deallocate(cs)
00314
```

42.2.2.4 ale_getcoordinate()

```
real function, dimension(cs%nk+1), public mom_ale::ale_getcoordinate (
              type(ale_cs), pointer CS )
```

Query the target coordinate interfaces positions.

Parameters

cs	module control structure

Definition at line 1207 of file MOM_ALE.F90.

```
01207   type(ale_cs), pointer    :: cs                 !< module control structure
01208
01209   real, dimension(CS%nk+1) :: ale_getcoordinate
01210   ale_getcoordinate(:) = getcoordinateinterfaces( cs%regridCS, undo_scaling=.true. )
01211
```

42.2.2.5 ale_getcoordinateunits()

```
character(len=20) function, public mom_ale::ale_getcoordinateunits (
             type(ale_cs), pointer CS )
```

Query the target coordinate units.

Parameters

cs	module control structure

Definition at line 1217 of file MOM_ALE.F90.

```
01217   type(ale_cs), pointer :: cs   !< module control structure
01218
01219   character(len=20)     :: ale_getcoordinateunits
01220
01221   ale_getcoordinateunits = getcoordinateunits( cs%regridCS )
01222
```

42.2.2.6 ale_init()

```
subroutine, public mom_ale::ale_init (
             type(param_file_type), intent(in) param_file,
             type(verticalgrid_type), intent(in) GV,
             type(unit_scale_type), intent(in) US,
             real, intent(in) max_depth,
             type(ale_cs), pointer CS )
```

This routine is typically called (from initialize_MOM in file MOM.F90) before the main time integration loop to initialize the regridding stuff. We read the MOM_input file to register the values of different regridding/remapping parameters.

Parameters

in	*param_file*	Parameter file
in	*gv*	Ocean vertical grid structure
in	*us*	A dimensional unit scaling type
in	*max_depth*	The maximum depth of the ocean [Z ~> m].
	cs	Module control structure

Definition at line 140 of file MOM_ALE.F90.

References ale_initregridding().

```
00140   type(param_file_type),   intent(in) :: param_file !< Parameter file
00141   type(verticalgrid_type), intent(in) :: gv         !< Ocean vertical grid structure
00142   type(unit_scale_type),   intent(in) :: us         !< A dimensional unit scaling type
00143   real,                    intent(in) :: max_depth  !< The maximum depth of the ocean [Z ~> m].
00144   type(ale_cs),            pointer    :: cs         !< Module control structure
00145
00146   ! Local variables
00147   real, dimension(:), allocatable :: dz
00148   character(len=40)               :: mdl = "MOM_ALE" ! This module's name.
00149   character(len=80)               :: string ! Temporary strings
00150   real                            :: filter_shallow_depth, filter_deep_depth
00151   logical       :: default_2018_answers
00152   logical                         :: check_reconstruction
00153   logical                         :: check_remapping
00154   logical                         :: force_bounds_in_subcell
00155   logical                         :: local_logical
00156   logical                         :: remap_boundary_extrap
00157
00158   if (associated(cs)) then
00159     call mom_error(warning, "ALE_init called with an associated "// &
00160                             "control structure.")
00161     return
00162   endif
00163   allocate(cs)
00164
00165   cs%show_call_tree = calltree_showquery()
00166   if (cs%show_call_tree) call calltree_enter("ALE_init(), MOM_ALE.F90")
00167
00168   call get_param(param_file, mdl, "REMAP_UV_USING_OLD_ALG", cs%remap_uv_using_old_alg, &
00169                  "If true, uses the old remapping-via-a-delta-z method for "//&
00170                  "remapping u and v. If false, uses the new method that remaps "//&
00171                  "between grids described by an old and new thickness.", &
00172                  default=.false.)
00173
00174   ! Initialize and configure regridding
00175   call ale_initregridding(gv, us, max_depth, param_file, mdl, cs%regridCS)
00176
00177   ! Initialize and configure remapping
00178   call get_param(param_file, mdl, "REMAPPING_SCHEME", string, &
00179                  "This sets the reconstruction scheme used "//&
00180                  "for vertical remapping for all variables. "//&
00181                  "It can be one of the following schemes: "//&
00182                  trim(remappingschemesdoc), default=remappingdefaultscheme)
00183   call get_param(param_file, mdl, "FATAL_CHECK_RECONSTRUCTIONS", check_reconstruction, &
00184                  "If true, cell-by-cell reconstructions are checked for "//&
00185                  "consistency and if non-monotonicity or an inconsistency is "//&
00186                  "detected then a FATAL error is issued.", default=.false.)
00187   call get_param(param_file, mdl, "FATAL_CHECK_REMAPPING", check_remapping, &
00188                  "If true, the results of remapping are checked for "//&
00189                  "conservation and new extrema and if an inconsistency is "//&
00190                  "detected then a FATAL error is issued.", default=.false.)
00191   call get_param(param_file, mdl, "REMAP_BOUND_INTERMEDIATE_VALUES", force_bounds_in_subcell, &
00192                  "If true, the values on the intermediate grid used for remapping "//&
00193                  "are forced to be bounded, which might not be the case due to "//&
00194                  "round off.", default=.false.)
00195   call get_param(param_file, mdl, "REMAP_BOUNDARY_EXTRAP", remap_boundary_extrap, &
00196                  "If true, values at the interfaces of boundary cells are "//&
00197                  "extrapolated instead of piecewise constant", default=.false.)
00198   call get_param(param_file, mdl, "DEFAULT_2018_ANSWERS", default_2018_answers, &
00199                  "This sets the default value for the various _2018_ANSWERS parameters.", &
00200                  default=.false.)
00201   call get_param(param_file, mdl, "REMAPPING_2018_ANSWERS", cs%answers_2018, &
00202                  "If true, use the order of arithmetic and expressions that recover the "//&
00203                  "answers from the end of 2018.  Otherwise, use updated and more robust "//&
00204                  "forms of the same expressions.", default=default_2018_answers)
00205   call initialize_remapping( cs%remapCS, string, &
00206                              boundary_extrapolation=remap_boundary_extrap, &
00207                              check_reconstruction=check_reconstruction, &
00208                              check_remapping=check_remapping, &
00209                              force_bounds_in_subcell=force_bounds_in_subcell, &
00210                              answers_2018=cs%answers_2018)
00211
00212   call get_param(param_file, mdl, "REMAP_AFTER_INITIALIZATION", cs%remap_after_initialization, &
00213                  "If true, applies regridding and remapping immediately after "//&
00214                  "initialization so that the state is ALE consistent. This is a "//&
00215                  "legacy step and should not be needed if the initialization is "//&
00216                  "consistent with the coordinate mode.", default=.true.)
00217
00218   call get_param(param_file, mdl, "REGRID_TIME_SCALE", cs%regrid_time_scale, &
00219                  "The time-scale used in blending between the current (old) grid "//&
00220                  "and the target (new) grid. A short time-scale favors the target "//&
```

```
00221                 "grid (0. or anything less than DT_THERM) has no memory of the old "//&
00222                 "grid. A very long time-scale makes the model more Lagrangian.", &
00223                 units="s", default=0., scale=us%s_to_T)
00224   call get_param(param_file, mdl, "REGRID_FILTER_SHALLOW_DEPTH", filter_shallow_depth, &
00225                 "The depth above which no time-filtering is applied. Above this depth "//&
00226                 "final grid exactly matches the target (new) grid.", &
00227                 units="m", default=0., scale=gv%m_to_H)
00228   call get_param(param_file, mdl, "REGRID_FILTER_DEEP_DEPTH", filter_deep_depth, &
00229                 "The depth below which full time-filtering is applied with time-scale "//&
00230                 "REGRID_TIME_SCALE. Between depths REGRID_FILTER_SHALLOW_DEPTH and "//&
00231                 "REGRID_FILTER_SHALLOW_DEPTH the filter weights adopt a cubic profile.", &
00232                 units="m", default=0., scale=gv%m_to_H)
00233   call set_regrid_params(cs%regridCS, depth_of_time_filter_shallow=filter_shallow_depth, &
00234                          depth_of_time_filter_deep=filter_deep_depth)
00235   call get_param(param_file, mdl, "REGRID_USE_OLD_DIRECTION", local_logical, &
00236                 "If true, the regridding ntegrates upwards from the bottom for "//&
00237                 "interface positions, much as the main model does. If false "//&
00238                 "regridding integrates downward, consistant with the remapping "//&
00239                 "code.", default=.true., do_not_log=.true.)
00240   call set_regrid_params(cs%regridCS, integrate_downward_for_e=.not.local_logical)
00241
00242   ! Keep a record of values for subsequent queries
00243   cs%nk = gv%ke
00244
00245   if (cs%show_call_tree) call calltree_leave("ALE_init()")
```

42.2.2.7 ale_initregridding()

```
subroutine, public mom_ale::ale_initregridding (
             type(verticalgrid_type), intent(in) GV,
             type(unit_scale_type), intent(in) US,
             real, intent(in) max_depth,
             type(param_file_type), intent(in) param_file,
             character(len=*), intent(in) mdl,
             type(regridding_cs), intent(out) regridCS )
```

Initializes regridding for the main ALE algorithm.

Parameters

in	*gv*	Ocean vertical grid structure
in	*us*	A dimensional unit scaling type
in	*max_depth*	The maximum depth of the ocean [Z ~> m].
in	*param_file*	parameter file
in	*mdl*	Name of calling module
out	*regridcs*	Regridding parameters and work arrays

Definition at line 1186 of file MOM_ALE.F90.

Referenced by ale_init().

```
01186   type(verticalgrid_type), intent(in)  :: gv         !< Ocean vertical grid structure
01187   type(unit_scale_type),   intent(in)  :: us         !< A dimensional unit scaling type
01188   real,                    intent(in)  :: max_depth  !< The maximum depth of the ocean [Z ~> m].
01189   type(param_file_type),   intent(in)  :: param_file !< parameter file
01190   character(len=*),        intent(in)  :: mdl        !< Name of calling module
01191   type(regridding_cs),     intent(out) :: regridcs   !< Regridding parameters and work arrays
01192   ! Local variables
01193   character(len=30) :: coord_mode
01194
01195   call get_param(param_file, mdl, "REGRIDDING_COORDINATE_MODE", coord_mode, &
01196                 "Coordinate mode for vertical regridding. "//&
```

```
01197                 "Choose among the following possibilities: "//&
01198                 trim(regriddingcoordinatemodedoc), &
01199                 default=default_coordinate_mode, fail_if_missing=.true.)
01200
01201   call initialize_regridding(regridcs, gv, us, max_depth, param_file, mdl, coord_mode, '', '')
01202
```

42.2.2.8 ale_initthicknesstocoord()

```
subroutine, public mom_ale::ale_initthicknesstocoord (
             type(ale_cs), intent(inout) CS,
             type(ocean_grid_type), intent(in) G,
             type(verticalgrid_type), intent(in) GV,
             real, dimension(szi_(g),szj_(g),szk_(gv)), intent(out) h )
```

Set h to coordinate values for fixed coordinate systems.

Parameters

in,out	*cs*	module control structure
in	*g*	module grid structure
in	*gv*	Ocean vertical grid structure
out	*h*	layer thickness [H ~> m or kg m-2]

Definition at line 1306 of file MOM_ALE.F90.

```
01306   type(ale_cs), intent(inout)                            :: cs  !< module control structure
01307   type(ocean_grid_type), intent(in)                      :: g   !< module grid structure
01308   type(verticalgrid_type), intent(in)                    :: gv  !< Ocean vertical grid structure
01309   real, dimension(SZI_(G),SZJ_(G),SZK_(GV)), intent(out) :: h   !< layer thickness [H ~> m or kg m-2]
01310
01311   ! Local variables
01312   integer :: i, j, k
01313
01314   do j = g%jsd,g%jed ; do i = g%isd,g%ied
01315     h(i,j,:) = gv%Z_to_H * getstaticthickness( cs%regridCS, 0., g%bathyT(i,j) )
01316   enddo ; enddo
01317
```

42.2.2.9 ale_main()

```
subroutine, public mom_ale::ale_main (
             type(ocean_grid_type), intent(in) G,
             type(verticalgrid_type), intent(in) GV,
             type(unit_scale_type), intent(in) US,
             real, dimension(szi_(g),szj_(g),szk_(gv)), intent(inout) h,
             real, dimension(szib_(g),szj_(g),szk_(gv)), intent(inout) u,
             real, dimension(szi_(g),szjb_(g),szk_(gv)), intent(inout) v,
             type(thermo_var_ptrs), intent(inout) tv,
             type(tracer_registry_type), pointer Reg,
             type(ale_cs), pointer CS,
             type(ocean_obc_type), pointer OBC,
```

```
real, intent(in), optional dt,
real, dimension(:,:), optional, pointer frac_shelf_h )
```

Takes care of (1) building a new grid and (2) remapping all variables between the old grid and the new grid. The creation of the new grid can be based on z coordinates, target interface densities, sigma coordinates or any arbitrary coordinate system.

Parameters

in	*g*	Ocean grid informations
in	*gv*	Ocean vertical grid structure
in	*us*	A dimensional unit scaling type
in,out	*h*	Current 3D grid obtained after the last time step [H ~> m or kg m-2]
in,out	*u*	Zonal velocity field [L T-1 ~> m s-1]
in,out	*v*	Meridional velocity field [L T-1 ~> m s-1]
in,out	*tv*	Thermodynamic variable structure
	reg	Tracer registry structure
	cs	Regridding parameters and options
	obc	Open boundary structure
in	*dt*	Time step between calls to ALE_main [T ~> s]
	frac_shelf_h	Fractional ice shelf coverage

Definition at line 322 of file MOM_ALE.F90.

References ale_update_regrid_weights(), check_grid(), and remap_all_state_vars().

```
00322   type(ocean_grid_type),                      intent(in)    :: g   !< Ocean grid informations
00323   type(verticalgrid_type),                    intent(in)    :: gv  !< Ocean vertical grid structure
00324   type(unit_scale_type),                      intent(in)    :: us  !< A dimensional unit scaling type
00325   real, dimension(SZI_(G),SZJ_(G),SZK_(GV)),  intent(inout) :: h   !< Current 3D grid obtained after the
00326                                                                      !! last time step [H ~> m or kg m-2]
00327   real, dimension(SZIB_(G),SZJ_(G),SZK_(GV)), intent(inout) :: u   !< Zonal velocity field [L T-1 ~> m s-1]
00328   real, dimension(SZI_(G),SZJB_(G),SZK_(GV)), intent(inout) :: v   !< Meridional velocity field [L T-1 ~> m
      s-1]
00329   type(thermo_var_ptrs),                      intent(inout) :: tv  !< Thermodynamic variable structure
00330   type(tracer_registry_type),                 pointer       :: reg !< Tracer registry structure
00331   type(ale_cs),                               pointer       :: cs  !< Regridding parameters and options
00332   type(ocean_obc_type),                       pointer       :: obc !< Open boundary structure
00333   real,                              optional, intent(in)    :: dt  !< Time step between calls to ALE_main
      [T ~> s]
00334   real, dimension(:,:),              optional, pointer       :: frac_shelf_h !< Fractional ice shelf
      coverage
00335   ! Local variables
00336   real, dimension(SZI_(G),SZJ_(G),SZK_(GV)+1) :: dzregrid ! The change in grid interface positions
00337   real, dimension(SZI_(G),SZJ_(G),SZK_(GV)+1) :: eta_preale
00338   real, dimension(SZI_(G),SZJ_(G),SZK_(GV)) :: h_new ! New 3D grid obtained after last time step [H ~> m or
      kg-2]
00339   integer :: nk, i, j, k, isc, iec, jsc, jec
00340   logical :: ice_shelf
00341
00342   nk = gv%ke; isc = g%isc; iec = g%iec; jsc = g%jsc; jec = g%jec
00343
00344   ice_shelf = .false.
00345   if (present(frac_shelf_h)) then
00346     if (associated(frac_shelf_h)) ice_shelf = .true.
00347   endif
00348
00349   if (cs%show_call_tree) call calltree_enter("ALE_main(), MOM_ALE.F90")
00350
00351   ! These diagnostics of the state before ALE is applied are mostly used for debugging.
00352   if (cs%id_u_preale > 0) call post_data(cs%id_u_preale, u,    cs%diag)
00353   if (cs%id_v_preale > 0) call post_data(cs%id_v_preale, v,    cs%diag)
00354   if (cs%id_h_preale > 0) call post_data(cs%id_h_preale, h,    cs%diag)
00355   if (cs%id_T_preale > 0) call post_data(cs%id_T_preale, tv%T, cs%diag)
00356   if (cs%id_S_preale > 0) call post_data(cs%id_S_preale, tv%S, cs%diag)
00357   if (cs%id_e_preale > 0) then
00358     call find_eta(h, tv, g, gv, us, eta_preale)
```

```
00359     call post_data(cs%id_e_preale, eta_preale, cs%diag)
00360   endif
00361
00362   if (present(dt)) then
00363     call ale_update_regrid_weights( dt, cs )
00364   endif
00365   dzregrid(:,:,:) = 0.0
00366
00367   ! Build new grid. The new grid is stored in h_new. The old grid is h.
00368   ! Both are needed for the subsequent remapping of variables.
00369   if (ice_shelf) then
00370     call regridding_main( cs%remapCS, cs%regridCS, g, gv, h, tv, h_new, dzregrid, frac_shelf_h)
00371   else
00372     call regridding_main( cs%remapCS, cs%regridCS, g, gv, h, tv, h_new, dzregrid)
00373   endif
00374
00375   call check_grid( g, gv, h, 0. )
00376
00377   if (cs%show_call_tree) call calltree_waypoint("new grid generated (ALE_main)")
00378
00379   ! The presence of dt is used for expediency to distinguish whether ALE_main is being called during init
00380   ! or in the main loop. Tendency diagnostics in remap_all_state_vars also rely on this logic.
00381   if (present(dt)) then
00382     call diag_update_remap_grids(cs%diag)
00383   endif
00384   ! Remap all variables from old grid h onto new grid h_new
00385   call remap_all_state_vars( cs%remapCS, cs, g, gv, h, h_new, reg, obc, -dzregrid, &
00386                              u, v, cs%show_call_tree, dt )
00387
00388   if (cs%show_call_tree) call calltree_waypoint("state remapped (ALE_main)")
00389
00390   ! Override old grid with new one. The new grid 'h_new' is built in
00391   ! one of the 'build_...' routines above.
00392   !$OMP parallel do default(shared)
00393   do k = 1,nk ; do j = jsc-1,jec+1 ; do i = isc-1,iec+1
00394     h(i,j,k) = h_new(i,j,k)
00395   enddo ; enddo ; enddo
00396
00397   if (cs%show_call_tree) call calltree_leave("ALE_main()")
00398
00399   if (cs%id_dzRegrid>0 .and. present(dt)) call post_data(cs%id_dzRegrid, dzregrid, cs%diag)
00400
00401
```

42.2.2.10 ale_main_offline()

```
subroutine, public mom_ale::ale_main_offline (
              type(ocean_grid_type), intent(in) G,
              type(verticalgrid_type), intent(in) GV,
              real, dimension(szi_(g),szj_(g),szk_(gv)), intent(inout) h,
              type(thermo_var_ptrs), intent(inout) tv,
              type(tracer_registry_type), pointer Reg,
              type(ale_cs), pointer CS,
              type(ocean_obc_type), pointer OBC,
              real, intent(in), optional dt )
```

Takes care of (1) building a new grid and (2) remapping all variables between the old grid and the new grid. The creation of the new grid can be based on z coordinates, target interface densities, sigma coordinates or any arbitrary coordinate system.

Parameters

in	*g*	Ocean grid informations
in	*gv*	Ocean vertical grid structure
in,out	*h*	Current 3D grid obtained after the last time step [H ~> m or kg-2]
in,out	*tv*	Thermodynamic variable structure
	reg	Tracer registry structure
	cs	Regridding parameters and options
	obc	Open boundary structure
in	*dt*	Time step between calls to ALE_main [T ~> s]

Definition at line 409 of file MOM_ALE.F90.

References ale_update_regrid_weights(), check_grid(), and remap_all_state_vars().

```
00409   type(ocean_grid_type),                      intent(in)    :: g   !< Ocean grid informations
00410   type(verticalgrid_type),                    intent(in)    :: gv  !< Ocean vertical grid structure
00411   real, dimension(SZI_(G),SZJ_(G),SZK_(GV)),  intent(inout) :: h   !< Current 3D grid obtained after the
00412                                                                      !! last time step [H ~> m or kg-2]
00413   type(thermo_var_ptrs),                      intent(inout) :: tv  !< Thermodynamic variable structure
00414   type(tracer_registry_type),                 pointer       :: reg !< Tracer registry structure
00415   type(ale_cs),                               pointer       :: cs  !< Regridding parameters and options
00416   type(ocean_obc_type),                       pointer       :: obc !< Open boundary structure
00417   real,                             optional, intent(in)    :: dt  !< Time step between calls to ALE_main
      [T ~> s]
00418   ! Local variables
00419   real, dimension(SZI_(G), SZJ_(G), SZK_(GV)+1) :: dzregrid ! The change in grid interface positions
00420   real, dimension(SZI_(G),SZJ_(G),SZK_(GV)) :: h_new ! New 3D grid obtained after last time step [H ~> m or
      kg-2]
00421   integer :: nk, i, j, k, isc, iec, jsc, jec
00422
00423   nk = gv%ke; isc = g%isc; iec = g%iec; jsc = g%jsc; jec = g%jec
00424
00425   if (cs%show_call_tree) call calltree_enter("ALE_main_offline(), MOM_ALE.F90")
00426
00427   if (present(dt)) then
00428     call ale_update_regrid_weights( dt, cs )
00429   endif
00430   dzregrid(:,:,:) = 0.0
00431
00432   ! Build new grid. The new grid is stored in h_new. The old grid is h.
00433   ! Both are needed for the subsequent remapping of variables.
00434   call regridding_main( cs%remapCS, cs%regridCS, g, gv, h, tv, h_new, dzregrid )
00435
00436   call check_grid( g, gv, h, 0. )
00437
00438   if (cs%show_call_tree) call calltree_waypoint("new grid generated (ALE_main)")
00439
00440   ! Remap all variables from old grid h onto new grid h_new
00441
00442   call remap_all_state_vars(cs%remapCS, cs, g, gv, h, h_new, reg, obc, &
00443                             debug=cs%show_call_tree, dt=dt )
00444
00445   if (cs%show_call_tree) call calltree_waypoint("state remapped (ALE_main)")
00446
00447   ! Override old grid with new one. The new grid 'h_new' is built in
00448   ! one of the 'build_...' routines above.
00449   !$OMP parallel do default(shared)
00450   do k = 1,nk ; do j = jsc-1,jec+1 ; do i = isc-1,iec+1
00451     h(i,j,k) = h_new(i,j,k)
00452   enddo ; enddo ; enddo
00453
00454   if (cs%show_call_tree) call calltree_leave("ALE_main()")
00455   if (cs%id_dzRegrid>0 .and. present(dt)) call post_data(cs%id_dzRegrid, dzregrid, cs%diag)
00456
```

42.2.2.11 ale_offline_inputs()

```
subroutine, public mom_ale::ale_offline_inputs (
              type(ale_cs), pointer CS,
              type(ocean_grid_type), intent(in) G,
              type(verticalgrid_type), intent(in) GV,
              real, dimension(szi_(g),szj_(g),szk_(gv)), intent(inout) h,
              type(thermo_var_ptrs), intent(inout) tv,
              type(tracer_registry_type), pointer Reg,
              real, dimension(szib_(g),szj_(g),szk_(gv)), intent(inout) uhtr,
              real, dimension(szi_(g),szjb_(g),szk_(gv)), intent(inout) vhtr,
              real, dimension(szi_(g),szj_(g),szk_(gv)+1), intent(inout) Kd,
              logical, intent(in) debug,
              type(ocean_obc_type), pointer OBC )
```

Regrid/remap stored fields used for offline tracer integrations. These input fields are assumed to have the same layer thicknesses at the end of the last offline interval (which should be a Zstar grid). This routine builds a grid on the runtime specified vertical coordinate.

Parameters

	cs	Regridding parameters and options
in	*g*	Ocean grid informations
in	*gv*	Ocean vertical grid structure
in,out	*h*	Layer thicknesses
in,out	*tv*	Thermodynamic variable structure
	reg	Tracer registry structure
in,out	*uhtr*	Zonal mass fluxes
in,out	*vhtr*	Meridional mass fluxes
in,out	*kd*	Input diffusivites
in	*debug*	If true, then turn checksums
	obc	Open boundary structure

Definition at line 463 of file MOM_ALE.F90.

References ale_remap_scalar(), check_grid(), and remap_all_state_vars().

```
00463   type(ale_cs),                                pointer       :: cs    !< Regridding parameters and options
00464   type(ocean_grid_type),                       intent(in   ) :: g     !< Ocean grid informations
00465   type(verticalgrid_type),                     intent(in   ) :: gv    !< Ocean vertical grid structure
00466   real, dimension(SZI_(G),SZJ_(G),SZK_(GV)),   intent(inout) :: h     !< Layer thicknesses
00467   type(thermo_var_ptrs),                       intent(inout) :: tv    !< Thermodynamic variable structure
00468   type(tracer_registry_type),                  pointer       :: reg   !< Tracer registry structure
00469   real, dimension(SZIB_(G),SZJ_(G),SZK_(GV)),  intent(inout) :: uhtr  !< Zonal mass fluxes
00470   real, dimension(SZI_(G),SZJB_(G),SZK_(GV)),  intent(inout) :: vhtr  !< Meridional mass fluxes
00471   real, dimension(SZI_(G),SZJ_(G),SZK_(GV)+1), intent(inout) :: kd    !< Input diffusivites
00472   logical,                                     intent(in   ) :: debug !< If true, then turn checksums
00473   type(ocean_obc_type),                        pointer       :: obc   !< Open boundary structure
00474   ! Local variables
00475   integer :: nk, i, j, k, isc, iec, jsc, jec
00476   real, dimension(SZI_(G), SZJ_(G), SZK_(GV))   :: h_new    ! Layer thicknesses after regridding
00477   real, dimension(SZI_(G), SZJ_(G), SZK_(GV)+1) :: dzregrid ! The change in grid interface positions
00478   real, dimension(SZK_(GV)) :: h_src
00479   real, dimension(SZK_(GV)) :: h_dest, uh_dest
00480   real, dimension(SZK_(GV)) :: temp_vec
00481
00482   nk = gv%ke; isc = g%isc; iec = g%iec; jsc = g%jsc; jec = g%jec
00483   dzregrid(:,:,:) = 0.0
00484   h_new(:,:,:) = 0.0
00485
00486   if (debug) call mom_tracer_chkinv("Before ALE_offline_inputs", g, h, reg%Tr, reg%ntr)
00487
00488   ! Build new grid from the Zstar state onto the requested vertical coordinate. The new grid is stored
00489   ! in h_new. The old grid is h. Both are needed for the subsequent remapping of variables. Convective
00490   ! adjustment right now is not used because it is unclear what to do with vanished layers
00491   call regridding_main( cs%remapCS, cs%regridCS, g, gv, h, tv, h_new, dzregrid, conv_adjust = .false. )
00492   call check_grid( g, gv, h_new, 0. )
00493   if (cs%show_call_tree) call calltree_waypoint("new grid generated (ALE_offline_inputs)")
00494
00495   ! Remap all variables from old grid h onto new grid h_new
00496   call remap_all_state_vars( cs%remapCS, cs, g, gv, h, h_new, reg, obc, debug=cs%show_call_tree )
00497   if (cs%show_call_tree) call calltree_waypoint("state remapped (ALE_inputs)")
00498
00499   ! Reintegrate mass transports from Zstar to the offline vertical coordinate
00500   do j=jsc,jec ; do i=g%iscB,g%iecB
00501     if (g%mask2dCu(i,j)>0.) then
00502       h_src(:) = 0.5 * (h(i,j,:) + h(i+1,j,:))
00503       h_dest(:) = 0.5 * (h_new(i,j,:) + h_new(i+1,j,:))
00504       call reintegrate_column(nk, h_src, uhtr(i,j,:), nk, h_dest, 0., temp_vec)
00505       uhtr(i,j,:) = temp_vec
00506     endif
00507   enddo ; enddo
00508   do j=g%jscB,g%jecB ; do i=isc,iec
00509     if (g%mask2dCv(i,j)>0.) then
00510       h_src(:) = 0.5 * (h(i,j,:) + h(i,j+1,:))
00511       h_dest(:) = 0.5 * (h_new(i,j,:) + h_new(i,j+1,:))
00512       call reintegrate_column(nk, h_src, vhtr(i,j,:), nk, h_dest, 0., temp_vec)
00513       vhtr(i,j,:) = temp_vec
00514     endif
00515   enddo ; enddo
00516
00517   do j = jsc,jec ; do i=isc,iec
```

```
00518     if (g%mask2dT(i,j)>0.) then
00519       if (check_column_integrals(nk, h_src, nk, h_dest)) then
00520         call mom_error(fatal, "ALE_offline_inputs: Kd interpolation columns do not match")
00521       endif
00522       call interpolate_column(nk, h(i,j,:), kd(i,j,:), nk, h_new(i,j,:), 0., kd(i,j,:))
00523     endif
00524   enddo ; enddo
00525
00526   call ale_remap_scalar(cs%remapCS, g, gv, nk, h, tv%T, h_new, tv%T, answers_2018=cs%answers_2018)
00527   call ale_remap_scalar(cs%remapCS, g, gv, nk, h, tv%S, h_new, tv%S, answers_2018=cs%answers_2018)
00528
00529   if (debug) call mom_tracer_chkinv("After ALE_offline_inputs", g, h_new, reg%Tr, reg%ntr)
00530
00531   ! Copy over the new layer thicknesses
00532   do k = 1,nk  ; do j = jsc-1,jec+1 ; do i = isc-1,iec+1
00533       h(i,j,k) = h_new(i,j,k)
00534   enddo ; enddo ; enddo
00535
00536   if (cs%show_call_tree) call calltree_leave("ALE_offline_inputs()")
```

42.2.2.12 ale_offline_tracer_final()

```
subroutine, public mom_ale::ale_offline_tracer_final (
             type(ocean_grid_type), intent(in) G,
             type(verticalgrid_type), intent(in) GV,
             real, dimension( g %isd:  g %ied, g %jsd:  g %jed, gv %ke), intent(inout) h,
             type(thermo_var_ptrs), intent(inout) tv,
             real, dimension( g %isd:  g %ied, g %jsd:  g %jed, gv %ke), intent(inout) h_↩
target,
             type(tracer_registry_type), pointer Reg,
             type(ale_cs), pointer CS,
             type(ocean_obc_type), pointer OBC )
```

Remaps all tracers from h onto h_target. This is intended to be called when tracers are done offline. In the case where transports don't quite conserve, we still want to make sure that layer thicknesses offline do not drift too far away from the online model.

Parameters

in	*g*	Ocean grid informations
in	*gv*	Ocean vertical grid structure
in,out	*h*	Current 3D grid obtained after the last time step [H ~> m or kg-2]
in,out	*tv*	Thermodynamic variable structure
in,out	*h_target*	Current 3D grid obtained after last time step [H ~> m or kg-2]
	reg	Tracer registry structure
	cs	Regridding parameters and options
	obc	Open boundary structure

Definition at line 544 of file MOM_ALE.F90.

References check_grid(), and remap_all_state_vars().

```
00544   type(ocean_grid_type),                     intent(in)    :: g   !< Ocean grid informations
00545   type(verticalgrid_type),                   intent(in)    :: gv  !< Ocean vertical grid structure
00546   real, dimension(SZI_(G),SZJ_(G),SZK_(GV)), intent(inout) :: h   !< Current 3D grid obtained after the
00547                                                                    !! last time step [H ~> m or kg-2]
00548   type(thermo_var_ptrs),                     intent(inout) :: tv  !< Thermodynamic variable structure
00549   real, dimension(SZI_(G),SZJ_(G),SZK_(GV)), intent(inout) :: h_target !< Current 3D grid obtained after
```

```
00550                                                                    !! last time step  [H ~> m or kg-2]
00551   type(tracer_registry_type),                pointer       :: reg !< Tracer registry structure
00552   type(ale_cs),                              pointer       :: cs  !< Regridding parameters and options
00553   type(ocean_obc_type),                      pointer       :: obc !< Open boundary structure
00554   ! Local variables
00555
00556   real, dimension(SZI_(G), SZJ_(G), SZK_(GV)+1) :: dzregrid !< The change in grid interface positions
00557   real, dimension(SZI_(G), SZJ_(G), SZK_(GV))   :: h_new    !< Regridded target thicknesses
00558   integer :: nk, i, j, k, isc, iec, jsc, jec
00559
00560   nk = gv%ke; isc = g%isc; iec = g%iec; jsc = g%jsc; jec = g%jec
00561
00562   if (cs%show_call_tree) call calltree_enter("ALE_offline_tracer_final(), MOM_ALE.F90")
00563   ! Need to make sure that h_target is consistent with the current offline ALE confiuration
00564   call regridding_main( cs%remapCS, cs%regridCS, g, gv, h_target, tv, h_new, dzregrid )
00565   call check_grid( g, gv, h_target, 0. )
00566
00567
00568   if (cs%show_call_tree) call calltree_waypoint("Source and target grids checked
      (ALE_offline_tracer_final)")
00569
00570   ! Remap all variables from old grid h onto new grid h_new
00571
00572   call remap_all_state_vars( cs%remapCS, cs, g, gv, h, h_new, reg, obc, debug=cs%show_call_tree )
00573
00574   if (cs%show_call_tree) call calltree_waypoint("state remapped (ALE_offline_tracer_final)")
00575
00576   ! Override old grid with new one. The new grid 'h_new' is built in
00577   ! one of the 'build_...' routines above.
00578   !$OMP parallel do default(shared)
00579   do k = 1,nk
00580     do j = jsc-1,jec+1 ; do i = isc-1,iec+1
00581       h(i,j,k) = h_new(i,j,k)
00582     enddo ; enddo
00583   enddo
00584   if (cs%show_call_tree) call calltree_leave("ALE_offline_tracer_final()")
```

42.2.2.13 ale_plm_edge_values()

```
subroutine, public mom_ale::ale_plm_edge_values (
             type(ale_cs), intent(in) CS,
             type(ocean_grid_type), intent(in) G,
             type(verticalgrid_type), intent(in) GV,
             real, dimension(szi_(g),szj_(g),szk_(gv)), intent(in) h,
             real, dimension(szi_(g),szj_(g),szk_(gv)), intent(in) Q,
             logical, intent(in) bdry_extrap,
             real, dimension(szi_(g),szj_(g),szk_(gv)), intent(inout) Q_t,
             real, dimension(szi_(g),szj_(g),szk_(gv)), intent(inout) Q_b )
```

Calculate edge values (top and bottom of layer) 3d scalar array. Boundary reconstructions are PCM unless bdry↩ _extrap is true.

Parameters

in	*cs*	module control structure
in	*g*	ocean grid structure
in	*gv*	Ocean vertical grid structure
in	*h*	layer thickness [H ~> m or kg m-2]
in	*q*	3d scalar array
in	*bdry_extrap*	If true, use high-order boundary extrapolation within boundary cells
in,out	*q_t*	Scalar at the top edge of each layer
in,out	*q_b*	Scalar at the bottom edge of each layer

Definition at line 1039 of file MOM_ALE.F90.

Referenced by ts_plm_edge_values().

```
01039   type(ale_cs),            intent(in)    :: cs   !< module control structure
01040   type(ocean_grid_type),   intent(in)    :: g    !< ocean grid structure
01041   type(verticalgrid_type), intent(in)    :: gv   !< Ocean vertical grid structure
01042   real, dimension(SZI_(G),SZJ_(G),SZK_(GV)), &
01043                            intent(in)    :: h    !< layer thickness [H ~> m or kg m-2]
01044   real, dimension(SZI_(G),SZJ_(G),SZK_(GV)), &
01045                            intent(in)    :: q    !< 3d scalar array
01046   logical,                 intent(in)    :: bdry_extrap !< If true, use high-order boundary
01047                                                 !! extrapolation within boundary cells
01048   real, dimension(SZI_(G),SZJ_(G),SZK_(GV)), &
01049                            intent(inout) :: q_t  !< Scalar at the top edge of each layer
01050   real, dimension(SZI_(G),SZJ_(G),SZK_(GV)), &
01051                            intent(inout) :: q_b  !< Scalar at the bottom edge of each layer
01052   ! Local variables
01053   integer :: i, j, k
01054   real :: slp(gv%ke)
01055   real :: mslp
01056   real :: h_neglect
01057
01058   if (.not.cs%answers_2018) then
01059     h_neglect = gv%H_subroundoff
01060   elseif (gv%Boussinesq) then
01061     h_neglect = gv%m_to_H*1.0e-30
01062   else
01063     h_neglect = gv%kg_m2_to_H*1.0e-30
01064   endif
01065
01066   !$OMP parallel do default(shared) private(slp,mslp)
01067   do j = g%jsc-1,g%jec+1 ; do i = g%isc-1,g%iec+1
01068     slp(1) = 0.
01069     do k = 2, gv%ke-1
01070       slp(k) = plm_slope_wa(h(i,j,k-1), h(i,j,k), h(i,j,k+1), h_neglect, q(i,j,k-1), q(i,j,k), q(i,j,k+1))
01071     enddo
01072     slp(gv%ke) = 0.
01073
01074     do k = 2, gv%ke-1
01075       mslp = plm_monotonized_slope(q(i,j,k-1), q(i,j,k), q(i,j,k+1), slp(k-1), slp(k), slp(k+1))
01076       q_t(i,j,k) = q(i,j,k) - 0.5 * mslp
01077       q_b(i,j,k) = q(i,j,k) + 0.5 * mslp
01078     enddo
01079     if (bdry_extrap) then
01080       mslp = - plm_extrapolate_slope(h(i,j,2), h(i,j,1), h_neglect, q(i,j,2), q(i,j,1))
01081       q_t(i,j,1) = q(i,j,1) - 0.5 * mslp
01082       q_b(i,j,1) = q(i,j,1) + 0.5 * mslp
01083       mslp = plm_extrapolate_slope(h(i,j,gv%ke-1), h(i,j,gv%ke), h_neglect, q(i,j,gv%ke-1), q(i,j,gv%ke))
01084       q_t(i,j,gv%ke) = q(i,j,gv%ke) - 0.5 * mslp
01085       q_b(i,j,gv%ke) = q(i,j,gv%ke) + 0.5 * mslp
01086     else
01087       q_t(i,j,1) = q(i,j,1)
01088       q_b(i,j,1) = q(i,j,1)
01089       q_t(i,j,gv%ke) = q(i,j,gv%ke)
01090       q_b(i,j,gv%ke) = q(i,j,gv%ke)
01091     endif
01092
01093   enddo ; enddo
01094
```

42.2.2.14 ale_register_diags()

```
subroutine, public mom_ale::ale_register_diags (
             type(time_type), intent(in), target Time,
             type(ocean_grid_type), intent(in) G,
             type(verticalgrid_type), intent(in) GV,
             type(unit_scale_type), intent(in) US,
             type(diag_ctrl), intent(in), target diag,
             type(ale_cs), pointer CS )
```

Initialize diagnostics for the ALE module.

Parameters

in	*time*	Time structure
in	*g*	Grid structure
in	*us*	A dimensional unit scaling type
in	*gv*	Ocean vertical grid structure
in	*diag*	Diagnostics control structure
	cs	Module control structure

Definition at line 250 of file MOM_ALE.F90.

```
00250   type(time_type),target,     intent(in)  :: time  !< Time structure
00251   type(ocean_grid_type),      intent(in)  :: g     !< Grid structure
00252   type(unit_scale_type),      intent(in)  :: us    !< A dimensional unit scaling type
00253   type(verticalgrid_type),    intent(in)  :: gv    !< Ocean vertical grid structure
00254   type(diag_ctrl), target,    intent(in)  :: diag  !< Diagnostics control structure
00255   type(ale_cs), pointer                   :: cs    !< Module control structure
00256
00257   cs%diag => diag
00258
00259   ! These diagnostics of the state variables before ALE are useful for
00260   ! debugging the ALE code.
00261   cs%id_u_preale = register_diag_field('ocean_model', 'u_preale', diag%axesCuL, time, &
00262       'Zonal velocity before remapping', 'm s-1', conversion=us%L_T_to_m_s)
00263   cs%id_v_preale = register_diag_field('ocean_model', 'v_preale', diag%axesCvL, time, &
00264       'Meridional velocity before remapping', 'm s-1', conversion=us%L_T_to_m_s)
00265   cs%id_h_preale = register_diag_field('ocean_model', 'h_preale', diag%axesTL, time, &
00266       'Layer Thickness before remapping', get_thickness_units(gv), &
00267       conversion=gv%H_to_MKS, v_extensive=.true.)
00268   cs%id_T_preale = register_diag_field('ocean_model', 'T_preale', diag%axesTL, time, &
00269       'Temperature before remapping', 'degC')
00270   cs%id_S_preale = register_diag_field('ocean_model', 'S_preale', diag%axesTL, time, &
00271       'Salinity before remapping', 'PSU')
00272   cs%id_e_preale = register_diag_field('ocean_model', 'e_preale', diag%axesTi, time, &
00273       'Interface Heights before remapping', 'm', conversion=us%Z_to_m)
00274
00275   cs%id_dzRegrid = register_diag_field('ocean_model','dzRegrid',diag%axesTi,time, &
00276       'Change in interface height due to ALE regridding', 'm', &
00277       conversion=gv%H_to_m)
00278   cs%id_vert_remap_h = register_diag_field('ocean_model', 'vert_remap_h', &
00279       diag%axestl, time, 'layer thicknesses after ALE regridding and remapping', 'm', &
00280       conversion=gv%H_to_m, v_extensive=.true.)
00281   cs%id_vert_remap_h_tendency = register_diag_field('ocean_model','vert_remap_h_tendency',diag%axestl,time,
      &
00282       'Layer thicknesses tendency due to ALE regridding and remapping', 'm', &
00283       conversion=gv%H_to_m*us%s_to_T, v_extensive = .true.)
00284
```

42.2.2.15 ale_regrid_accelerated()

```
subroutine, public mom_ale::ale_regrid_accelerated (
              type(ale_cs), pointer CS,
              type(ocean_grid_type), intent(inout) G,
              type(verticalgrid_type), intent(in) GV,
              real, dimension( g %isd:  g %ied, g %jsd:  g %jed, gv %ke), intent(inout) h,
              type(thermo_var_ptrs), intent(inout) tv,
              integer, intent(in) n,
              real, dimension( g %isdb:  g %iedb, g %jsd:  g %jed, gv %ke), intent(inout) u,
              real, dimension( g %isd:  g %ied, g %jsdb:  g %jedb, gv %ke), intent(inout) v,
              type(ocean_obc_type), pointer OBC,
              type(tracer_registry_type), optional, pointer Reg,
              real, intent(in), optional dt,
              real, dimension( g %isd:  g %ied, g %jsd:  g %jed, gv %ke+1), intent(inout),
```

```
optional dzRegrid,
             logical, intent(in), optional initial )
```

For a state-based coordinate, accelerate the process of regridding by repeatedly applying the grid calculation algorithm.

Parameters

	cs	ALE control structure
in,out	*g*	Ocean grid
in	*gv*	Vertical grid
in,out	*h*	Original thicknesses [H ~> m or kg-2]
in,out	*tv*	Thermo vars (T/S/EOS)
in	*n*	Number of times to regrid
in,out	*u*	Zonal velocity [L T-1 ~> m s-1]
in,out	*v*	Meridional velocity [L T-1 ~> m s-1]
	obc	Open boundary structure
	reg	Tracer registry to remap onto new grid
in	*dt*	Model timestep to provide a timescale for regridding [T ~> s]
in,out	*dzregrid*	Final change in interface positions
in	*initial*	Whether we're being called from an initialization routine (and expect diagnostics to work)

Definition at line 660 of file MOM_ALE.F90.

References ale_update_regrid_weights(), and remap_all_state_vars().

```
00660   type(ale_cs),            pointer       :: cs     !< ALE control structure
00661   type(ocean_grid_type),   intent(inout) :: g      !< Ocean grid
00662   type(verticalgrid_type), intent(in)    :: gv     !< Vertical grid
00663   real, dimension(SZI_(G),SZJ_(G),SZK_(GV)), &
00664                            intent(inout) :: h      !< Original thicknesses [H ~> m or kg-2]
00665   type(thermo_var_ptrs),   intent(inout) :: tv     !< Thermo vars (T/S/EOS)
00666   integer,                 intent(in)    :: n      !< Number of times to regrid
00667   real, dimension(SZIB_(G),SZJ_(G),SZK_(GV)), &
00668                            intent(inout) :: u      !< Zonal velocity [L T-1 ~> m s-1]
00669   real, dimension(SZI_(G),SZJB_(G),SZK_(GV)), &
00670                            intent(inout) :: v      !< Meridional velocity [L T-1 ~> m s-1]
00671   type(ocean_obc_type),    pointer       :: obc    !< Open boundary structure
00672   type(tracer_registry_type), &
00673                  optional, pointer       :: reg    !< Tracer registry to remap onto new grid
00674   real,          optional, intent(in)    :: dt     !< Model timestep to provide a timescale for regridding
      [T ~> s]
00675   real, dimension(SZI_(G),SZJ_(G),SZK_(GV)+1), &
00676                  optional, intent(inout) :: dzregrid !< Final change in interface positions
00677   logical,       optional, intent(in)    :: initial !< Whether we're being called from an initialization
00678                                                     !! routine (and expect diagnostics to work)
00679
00680   ! Local variables
00681   integer :: i, j, k, nz
00682   type(thermo_var_ptrs) :: tv_local ! local/intermediate temp/salt
00683   type(group_pass_type) :: pass_t_s_h ! group pass if the coordinate has a stencil
00684   real, dimension(SZI_(G),SZJ_(G),SZK_(GV))         :: h_loc, h_orig ! A working copy of layer thicknesses
00685   real, dimension(SZI_(G),SZJ_(G),SZK_(GV)), target :: t, s ! local temporary state
00686   ! we have to keep track of the total dzInterface if for some reason
00687   ! we're using the old remapping algorithm for u/v
00688   real, dimension(SZI_(G),SZJ_(G),SZK_(GV)+1) :: dzinterface, dzinttotal
00689
00690   nz = gv%ke
00691
00692   ! initial total interface displacement due to successive regridding
00693   dzinttotal(:,:,:) = 0.
00694
00695   call create_group_pass(pass_t_s_h, t, g%domain)
00696   call create_group_pass(pass_t_s_h, s, g%domain)
00697   call create_group_pass(pass_t_s_h, h_loc, g%domain)
00698
00699   ! copy original temp/salt and set our local tv_pointers to them
00700   tv_local = tv
00701   t(:,:,:) = tv%T(:,:,:)
00702   s(:,:,:) = tv%S(:,:,:)
00703   tv_local%T => t
00704   tv_local%S => s
00705
00706   ! get local copy of thickness and save original state for remapping
00707   h_loc(:,:,:) = h(:,:,:)
00708   h_orig(:,:,:) = h(:,:,:)
00709
```

```
00710   ! Apply timescale to regridding (for e.g. filtered_grid_motion)
00711   if (present(dt)) &
00712     call ale_update_regrid_weights(dt, cs)
00713
00714   do k = 1, n
00715     call do_group_pass(pass_t_s_h, g%domain)
00716
00717     ! generate new grid
00718     call regridding_main(cs%remapCS, cs%regridCS, g, gv, h_loc, tv_local, h, dzinterface)
00719     dzinttotal(:,:,:) = dzinttotal(:,:,:) + dzinterface(:,:,:)
00720
00721     ! remap from original grid onto new grid
00722     do j = g%jsc-1,g%jec+1 ; do i = g%isc-1,g%iec+1
00723       call remapping_core_h(cs%remapCS, nz, h_orig(i,j,:), tv%S(i,j,:), nz, h(i,j,:), tv_local%S(i,j,:))
00724       call remapping_core_h(cs%remapCS, nz, h_orig(i,j,:), tv%T(i,j,:), nz, h(i,j,:), tv_local%T(i,j,:))
00725     enddo ; enddo
00726
00727     ! starting grid for next iteration
00728     h_loc(:,:,:) = h(:,:,:)
00729   enddo
00730
00731   ! remap all state variables (including those that weren't needed for regridding)
00732   call remap_all_state_vars(cs%remapCS, cs, g, gv, h_orig, h, reg, obc, dzinttotal, u, v)
00733
00734   ! save total dzregrid for diags if needed?
00735   if (present(dzregrid)) dzregrid(:,:,:) = dzinttotal(:,:,:)
```

42.2.2.16 ale_remap_init_conds()

```
logical function, public mom_ale::ale_remap_init_conds (
            type(ale_cs), pointer CS )
```

Returns true if initial conditions should be regridded and remapped.

Parameters

cs	module control structure

Definition at line 1228 of file MOM_ALE.F90.

```
01228   type(ale_cs), pointer :: cs   !< module control structure
01229
01230   ale_remap_init_conds = .false.
01231   if (associated(cs)) ale_remap_init_conds = cs%remap_after_initialization
```

42.2.2.17 ale_remap_scalar()

```
subroutine, public mom_ale::ale_remap_scalar (
            type(remapping_cs), intent(in) CS,
            type(ocean_grid_type), intent(in) G,
            type(verticalgrid_type), intent(in) GV,
            integer, intent(in) nk_src,
            real, dimension(szi_(g),szj_(g),nk_src), intent(in) h_src,
            real, dimension(szi_(g),szj_(g),nk_src), intent(in) s_src,
            real, dimension(szi_(g),szj_(g),szk_(gv)), intent(in) h_dst,
            real, dimension(szi_(g),szj_(g),szk_(gv)), intent(inout) s_dst,
            logical, intent(in), optional all_cells,
```

```
logical, intent(in), optional old_remap,
logical, intent(in), optional answers_2018 )
```

Remaps a single scalar between grids described by thicknesses h_src and h_dst. h_dst must be dimensioned as a model array with GVke layers while h_src can have an arbitrary number of layers specified by nk_src.

Parameters

in	*cs*	Remapping control structure
in	*g*	Ocean grid structure
in	*gv*	Ocean vertical grid structure
in	*nk_src*	Number of levels on source grid
in	*h_src*	Level thickness of source grid [H ~> m or kg-2]
in	*s_src*	Scalar on source grid
in	*h_dst*	Level thickness of destination grid [H ~> m or kg-2]
in,out	*s_dst*	Scalar on destination grid
in	*all_cells*	If false, only reconstruct for non-vanished cells. Use all vanished layers otherwise (default).
in	*old_remap*	If true, use the old "remapping_core_w" method, otherwise use "remapping_core_h".
in	*answers_2018*	If true, use the order of arithmetic and expressions that recover the answers for remapping from the end of 2018. Otherwise, use more robust forms of the same expressions.

Definition at line 945 of file MOM_ALE.F90.

Referenced by ale_offline_inputs().

```
00945   type(remapping_cs),                     intent(in)    :: cs         !< Remapping control structure
00946   type(ocean_grid_type),                  intent(in)    :: g          !< Ocean grid structure
00947   type(verticalgrid_type),                intent(in)    :: gv         !< Ocean vertical grid structure
00948   integer,                                intent(in)    :: nk_src     !< Number of levels on source grid
00949   real, dimension(SZI_(G),SZJ_(G),nk_src), intent(in)    :: h_src      !< Level thickness of source grid
00950                                                                         !! [H ~> m or kg-2]
00951   real, dimension(SZI_(G),SZJ_(G),nk_src), intent(in)    :: s_src      !< Scalar on source grid
00952   real, dimension(SZI_(G),SZJ_(G),SZK_(GV)),intent(in)   :: h_dst      !< Level thickness of destination
      grid
00953                                                                         !! [H ~> m or kg-2]
00954   real, dimension(SZI_(G),SZJ_(G),SZK_(GV)),intent(inout) :: s_dst     !< Scalar on destination grid
00955   logical, optional,                      intent(in)    :: all_cells  !< If false, only reconstruct for
00956                                                                         !! non-vanished cells. Use all
      vanished
00957                                                                         !! layers otherwise (default).
00958   logical, optional,                      intent(in)    :: old_remap  !< If true, use the old
      "remapping_core_w"
00959                                                                         !! method, otherwise use
      "remapping_core_h".
00960   logical,                        optional, intent(in)    :: answers_2018 !< If true, use the order of
      arithmetic
00961                                                                         !! and expressions that recover the
      answers for
00962                                                                         !! remapping from the end of 2018.
      Otherwise,
00963                                                                         !! use more robust forms of the same
      expressions.
00964   ! Local variables
00965   integer :: i, j, k, n_points
00966   real :: dx(gv%ke+1)
00967   real :: h_neglect, h_neglect_edge
00968   logical :: ignore_vanished_layers, use_remapping_core_w, use_2018_remap
00969
00970   ignore_vanished_layers = .false.
00971   if (present(all_cells)) ignore_vanished_layers = .not. all_cells
00972   use_remapping_core_w = .false.
00973   if (present(old_remap)) use_remapping_core_w = old_remap
00974   n_points = nk_src
00975   use_2018_remap = .true. ; if (present(answers_2018)) use_2018_remap = answers_2018
```

```
00976
00977   if (.not.use_2018_remap) then
00978     h_neglect = gv%H_subroundoff ; h_neglect_edge = gv%H_subroundoff
00979   elseif (gv%Boussinesq) then
00980     h_neglect = gv%m_to_H*1.0e-30 ; h_neglect_edge = gv%m_to_H*1.0e-10
00981   else
00982     h_neglect = gv%kg_m2_to_H*1.0e-30 ; h_neglect_edge = gv%kg_m2_to_H*1.0e-10
00983   endif
00984
00985   !$OMP parallel do default(shared) firstprivate(n_points,dx)
00986   do j = g%jsc,g%jec ; do i = g%isc,g%iec
00987     if (g%mask2dT(i,j) > 0.) then
00988       if (ignore_vanished_layers) then
00989         n_points = 0
00990         do k = 1, nk_src
00991           if (h_src(i,j,k)>0.) n_points = n_points + 1
00992         enddo
00993         s_dst(i,j,:) = 0.
00994       endif
00995       if (use_remapping_core_w) then
00996         call dzfromh1h2( n_points, h_src(i,j,1:n_points), gv%ke, h_dst(i,j,:), dx )
00997         call remapping_core_w(cs, n_points, h_src(i,j,1:n_points), s_src(i,j,1:n_points), &
00998                               gv%ke, dx, s_dst(i,j,:), h_neglect, h_neglect_edge)
00999       else
01000         call remapping_core_h(cs, n_points, h_src(i,j,1:n_points), s_src(i,j,1:n_points), &
01001                               gv%ke, h_dst(i,j,:), s_dst(i,j,:), h_neglect, h_neglect_edge)
01002       endif
01003     else
01004       s_dst(i,j,:) = 0.
01005     endif
01006   enddo ; enddo
01007
```

42.2.2.18 ale_update_regrid_weights()

```
subroutine, public mom_ale::ale_update_regrid_weights (
             real, intent(in) dt,
             type(ale_cs), pointer CS )
```

Updates the weights for time filtering the new grid generated in regridding.

Parameters

in	*dt*	Time-step used between ALE calls [T ~> s]
	cs	ALE control structure

Definition at line 1236 of file MOM_ALE.F90.

Referenced by ale_main(), ale_main_offline(), and ale_regrid_accelerated().

```
01236   real,         intent(in) :: dt !< Time-step used between ALE calls [T ~> s]
01237   type(ale_cs), pointer    :: cs !< ALE control structure
01238   ! Local variables
01239   real :: w  ! An implicit weighting estimate.
01240
01241   if (associated(cs)) then
01242     w = 0.0
01243     if (cs%regrid_time_scale > 0.0) then
01244       w = cs%regrid_time_scale / (cs%regrid_time_scale + dt)
01245     endif
01246     call set_regrid_params(cs%regridCS, old_grid_weight=w)
01247   endif
01248
```

42.2.2.19 ale_updateverticalgridtype()

```
subroutine, public mom_ale::ale_updateverticalgridtype (
              type(ale_cs), pointer CS,
              type(verticalgrid_type), pointer GV )
```

Update the vertical grid type with ALE information. This subroutine sets information in the verticalGrid_type to be consistent with the use of ALE mode.

Parameters

cs	ALE control structure
gv	vertical grid information

Definition at line 1255 of file MOM_ALE.F90.

```
01255   type(ale_cs),            pointer :: cs  !< ALE control structure
01256   type(verticalgrid_type), pointer :: gv  !< vertical grid information
01257
01258   integer :: nk
01259
01260   nk = gv%ke
01261   gv%sInterface(1:nk+1) = getcoordinateinterfaces( cs%regridCS, undo_scaling=.true. )
01262   gv%sLayer(1:nk) = 0.5*( gv%sInterface(1:nk) + gv%sInterface(2:nk+1) )
01263   gv%zAxisUnits = getcoordinateunits( cs%regridCS )
01264   gv%zAxisLongName = getcoordinateshortname( cs%regridCS )
01265   gv%direction = -1 ! Because of ferret in z* mode. Need method to set
01266                     ! as function of coordinae mode.
01267
```

42.2.2.20 ale_writecoordinatefile()

```
subroutine, public mom_ale::ale_writecoordinatefile (
              type(ale_cs), pointer CS,
              type(verticalgrid_type), intent(in) GV,
              character(len=*), intent(in) directory )
```

Write the vertical coordinate information into a file. This subroutine writes out a file containing any available data related to the vertical grid used by the MOM ocean model when in ALE mode.

Parameters

	cs	module control structure
in	*gv*	ocean vertical grid structure
in	*directory*	directory for writing grid info

Definition at line 1275 of file MOM_ALE.F90.

```
01275   type(ale_cs),            pointer     :: cs         !< module control structure
01276   type(verticalgrid_type), intent(in)  :: gv         !< ocean vertical grid structure
01277   character(len=*),        intent(in)  :: directory  !< directory for writing grid info
01278
01279   character(len=240) :: filepath
01280   type(vardesc)      :: vars(2)
01281   type(fieldtype)    :: fields(2)
```

```
01282   integer            :: unit
01283   real               :: ds(gv%ke), dsi(gv%ke+1)
01284
01285   filepath    = trim(directory) // trim("Vertical_coordinate")
01286   ds(:)       = getcoordinateresolution( cs%regridCS, undo_scaling=.true. )
01287   dsi(1)      = 0.5*ds(1)
01288   dsi(2:gv%ke) = 0.5*( ds(1:gv%ke-1) + ds(2:gv%ke) )
01289   dsi(gv%ke+1) = 0.5*ds(gv%ke)
01290
01291   vars(1) = var_desc('ds', getcoordinateunits( cs%regridCS ), &
01292                      'Layer Coordinate Thickness','1','L','1')
01293   vars(2) = var_desc('ds_interface', getcoordinateunits( cs%regridCS ), &
01294                      'Layer Center Coordinate Separation','1','i','1')
01295
01296   call create_file(unit, trim(filepath), vars, 2, fields, single_file, gv=gv)
01297   call write_field(unit, fields(1), ds)
01298   call write_field(unit, fields(2), dsi)
01299   call close_file(unit)
01300
```

42.2.2.21 check_grid()

```
subroutine mom_ale::check_grid (
             type(ocean_grid_type), intent(in) G,
             type(verticalgrid_type), intent(in) GV,
             real, dimension( g %isd:  g %ied, g %jsd:  g %jed, gv %ke), intent(in) h,
             real, intent(in) threshold )  [private]
```

Check grid for negative thicknesses.

Parameters

in	*g*	Ocean grid structure
in	*gv*	Ocean vertical grid structure
in	*h*	Current 3D grid obtained after the last time step [H ~> m or kg m-2]
in	*threshold*	Value below which to flag issues, [H ~> m or kg m-2]

Definition at line 589 of file MOM_ALE.F90.

Referenced by ale_main(), ale_main_offline(), ale_offline_inputs(), and ale_offline_tracer_final().

```
00589   type(ocean_grid_type),                    intent(in) :: g  !< Ocean grid structure
00590   type(verticalgrid_type),                  intent(in) :: gv !< Ocean vertical grid structure
00591   real, dimension(SZI_(G),SZJ_(G),SZK_(GV)), intent(in) :: h  !< Current 3D grid obtained after the
00592                                                             !! last time step [H ~> m or kg m-2]
00593   real,                                     intent(in) :: threshold !< Value below which to flag issues,
00594                                                             !! [H ~> m or kg m-2]
00595   ! Local variables
00596   integer :: i, j
00597
00598   do j = g%jsc,g%jec ; do i = g%isc,g%iec
00599     if (g%mask2dT(i,j)>0.) then
00600       if (minval(h(i,j,:)) < threshold) then
00601         write(0,*) 'check_grid: i,j=',i,j,'h(i,j,:)=',h(i,j,:)
00602         if (threshold <= 0.) then
00603           call mom_error(fatal,"MOM_ALE, check_grid: negative thickness encountered.")
00604         else
00605           call mom_error(fatal,"MOM_ALE, check_grid: too tiny thickness encountered.")
00606         endif
00607       endif
00608     endif
00609   enddo ; enddo
00610
00611
```

42.2.2.22 remap_all_state_vars()

```
subroutine mom_ale::remap_all_state_vars (
             type(remapping_cs), intent(in) CS_remapping,
             type(ale_cs), intent(in) CS_ALE,
             type(ocean_grid_type), intent(in) G,
             type(verticalgrid_type), intent(in) GV,
             real, dimension(szi_(g),szj_(g),szk_(gv)), intent(in) h_old,
             real, dimension(szi_(g),szj_(g),szk_(gv)), intent(in) h_new,
             type(tracer_registry_type), pointer Reg,
             type(ocean_obc_type), pointer OBC,
             real, dimension(szi_(g),szj_(g),szk_(gv)+1), intent(in), optional dxInterface,
             real, dimension(szib_(g),szj_(g),szk_(gv)), intent(inout), optional u,
             real, dimension(szi_(g),szjb_(g),szk_(gv)), intent(inout), optional v,
             logical, intent(in), optional debug,
             real, intent(in), optional dt )  [private]
```

This routine takes care of remapping all variable between the old and the new grids. When velocity components need to be remapped, thicknesses at velocity points are taken to be arithmetic averages of tracer thicknesses. This routine is called during initialization of the model at time=0, to remap initiali conditions to the model grid. It is also called during a time step to update the state.

Parameters

in	*cs_remapping*	Remapping control structure
in	*cs_ale*	ALE control structure
in	*g*	Ocean grid structure
in	*gv*	Ocean vertical grid structure
in	*h_old*	Thickness of source grid [H ~> m or kg-2]
in	*h_new*	Thickness of destination grid [H ~> m or kg-2]
	reg	Tracer registry structure
	obc	Open boundary structure
in	*dxinterface*	Change in interface position
in,out	*u*	Zonal velocity [L T-1 ~> m s-1]
in,out	*v*	Meridional velocity [L T-1 ~> m s-1]
in	*debug*	If true, show the call tree
in	*dt*	time step for diagnostics [T ~> s]

Definition at line 746 of file MOM_ALE.F90.

Referenced by ale_main(), ale_main_offline(), ale_offline_inputs(), ale_offline_tracer_final(), and ale_regrid_↩ accelerated().

```
00746   type(remapping_cs),                       intent(in)    :: cs_remapping !< Remapping control structure
00747   type(ale_cs),                             intent(in)    :: cs_ale       !< ALE control structure
00748   type(ocean_grid_type),                    intent(in)    :: g            !< Ocean grid structure
00749   type(verticalgrid_type),                  intent(in)    :: gv           !< Ocean vertical grid structure
00750   real, dimension(SZI_(G),SZJ_(G),SZK_(GV)), intent(in)    :: h_old        !< Thickness of source grid
00751                                                                             !! [H ~> m or kg-2]
00752   real, dimension(SZI_(G),SZJ_(G),SZK_(GV)), intent(in)    :: h_new        !< Thickness of destination grid
00753                                                                             !! [H ~> m or kg-2]
00754   type(tracer_registry_type),               pointer       :: reg          !< Tracer registry structure
00755   type(ocean_obc_type),                     pointer       :: obc          !< Open boundary structure
00756   real, dimension(SZI_(G),SZJ_(G),SZK_(GV)+1), &
00757                                    optional, intent(in)    :: dxinterface  !< Change in interface position
00758                                                                             !! [H ~> m or kg-2]
00759   real, dimension(SZIB_(G),SZJ_(G),SZK_(GV)), &
00760                                    optional, intent(inout) :: u      !< Zonal velocity [L T-1 ~> m s-1]
```

```
00761   real, dimension(SZI_(G),SZJB_(G),SZK_(GV)), &
00762                                 optional, intent(inout) :: v      !< Meridional velocity [L T-1 ~> m
      s-1]
00763   logical,                      optional, intent(in)    :: debug  !< If true, show the call tree
00764   real,                         optional, intent(in)    :: dt     !< time step for diagnostics [T ~> s]
00765   ! Local variables
00766   integer                                       :: i, j, k, m
00767   integer                                       :: nz, ntr
00768   real, dimension(GV%ke+1)                      :: dx
00769   real, dimension(GV%ke)                        :: h1, u_column
00770   real, dimension(SZI_(G), SZJ_(G), SZK_(GV))   :: work_conc
00771   real, dimension(SZI_(G), SZJ_(G), SZK_(GV))   :: work_cont
00772   real, dimension(SZI_(G), SZJ_(G))             :: work_2d
00773   real                                          :: idt ! The inverse of the timestep [T-1 ~> s-1]
00774   real                                          :: ppt2mks
00775   real, dimension(GV%ke)                        :: h2
00776   real :: h_neglect, h_neglect_edge
00777   logical                                       :: show_call_tree
00778   type(tracer_type), pointer                    :: tr => null()
00779
00780   show_call_tree = .false.
00781   if (present(debug)) show_call_tree = debug
00782   if (show_call_tree) call calltree_enter("remap_all_state_vars(), MOM_ALE.F90")
00783
00784   ! If remap_uv_using_old_alg is .true. and u or v is requested, then we must have dxInterface. Otherwise,
00785   ! u and v can be remapped without dxInterface
00786   if ( .not. present(dxinterface) .and. (cs_ale%remap_uv_using_old_alg .and. (present(u) .or. present(v)))
      ) then
00787     call mom_error(fatal, "remap_all_state_vars: dxInterface must be present if using old algorithm "// &
00788                           "and u/v are to be remapped")
00789   endif
00790
00791   if (.not.cs_ale%answers_2018) then
00792     h_neglect = gv%H_subroundoff ; h_neglect_edge = gv%H_subroundoff
00793   elseif (gv%Boussinesq) then
00794     h_neglect = gv%m_to_H*1.0e-30 ; h_neglect_edge = gv%m_to_H*1.0e-10
00795   else
00796     h_neglect = gv%kg_m2_to_H*1.0e-30 ; h_neglect_edge = gv%kg_m2_to_H*1.0e-10
00797   endif
00798
00799   nz      = gv%ke
00800   ppt2mks = 0.001
00801
00802   ntr = 0 ; if (associated(reg)) ntr = reg%ntr
00803
00804   if (present(dt)) then
00805     idt = 1.0/dt
00806     work_conc(:,:,:) = 0.0
00807     work_cont(:,:,:) = 0.0
00808   endif
00809
00810   ! Remap tracer
00811   if (ntr>0) then
00812     if (show_call_tree) call calltree_waypoint("remapping tracers (remap_all_state_vars)")
00813     !$OMP parallel do default(shared) private(h1,h2,u_column,Tr)
00814     do m=1,ntr ! For each tracer
00815       tr => reg%Tr(m)
00816       do j = g%jsc,g%jec ; do i = g%isc,g%iec ; if (g%mask2dT(i,j)>0.) then
00817         ! Build the start and final grids
00818         h1(:) = h_old(i,j,:)
00819         h2(:) = h_new(i,j,:)
00820         call remapping_core_h(cs_remapping, nz, h1, tr%t(i,j,:), nz, h2, &
00821                               u_column, h_neglect, h_neglect_edge)
00822
00823         ! Intermediate steps for tendency of tracer concentration and tracer content.
00824         if (present(dt)) then
00825           if (tr%id_remap_conc > 0) then
00826             do k=1,gv%ke
00827               work_conc(i,j,k) = (u_column(k) - tr%t(i,j,k)) * idt
00828             enddo
00829           endif
00830           if (tr%id_remap_cont > 0 .or. tr%id_remap_cont_2d > 0) then
00831             do k=1,gv%ke
00832               work_cont(i,j,k) = (u_column(k)*h2(k) - tr%t(i,j,k)*h1(k)) * idt
00833             enddo
00834           endif
00835         endif
00836         ! update tracer concentration
00837         tr%t(i,j,:) = u_column(:)
00838       endif ; enddo ; enddo
00839
00840       ! tendency diagnostics.
00841       if (present(dt)) then
00842         if (tr%id_remap_conc > 0) then
00843           call post_data(tr%id_remap_conc, work_conc, cs_ale%diag)
00844         endif
00845         if (tr%id_remap_cont > 0) then
```

```
00846           call post_data(tr%id_remap_cont, work_cont, cs_ale%diag)
00847         endif
00848         if (tr%id_remap_cont_2d > 0) then
00849           do j = g%jsc,g%jec ; do i = g%isc,g%iec
00850             work_2d(i,j) = 0.0
00851             do k = 1,gv%ke
00852               work_2d(i,j) = work_2d(i,j) + work_cont(i,j,k)
00853             enddo
00854           enddo ; enddo
00855           call post_data(tr%id_remap_cont_2d, work_2d, cs_ale%diag)
00856         endif
00857       endif
00858     enddo ! m=1,ntr
00859
00860   endif   ! endif for ntr > 0
00861
00862   if (show_call_tree) call calltree_waypoint("tracers remapped (remap_all_state_vars)")
00863
00864   ! Remap u velocity component
00865   if ( present(u) ) then
00866     !$OMP parallel do default(shared) private(h1,h2,dx,u_column)
00867     do j = g%jsc,g%jec ; do i = g%iscB,g%iecB ; if (g%mask2dCu(i,j)>0.) then
00868       ! Build the start and final grids
00869       h1(:) = 0.5 * ( h_old(i,j,:) + h_old(i+1,j,:) )
00870       if (cs_ale%remap_uv_using_old_alg) then
00871         dx(:) = 0.5 * ( dxinterface(i,j,:) + dxinterface(i+1,j,:) )
00872         do k = 1, nz
00873           h2(k) = max( 0., h1(k) + ( dx(k+1) - dx(k) ) )
00874         enddo
00875       else
00876         h2(:) = 0.5 * ( h_new(i,j,:) + h_new(i+1,j,:) )
00877       endif
00878       if (associated(obc)) then
00879         if (obc%segnum_u(i,j) .ne. 0) then
00880           if (obc%segment(obc%segnum_u(i,j))%direction == obc_direction_e) then
00881             h1(:) = h_old(i,j,:)
00882             h2(:) = h_new(i,j,:)
00883           else ! (OBC%segment(n)%direction == OBC_DIRECTION_W)
00884             h1(:) = h_old(i+1,j,:)
00885             h2(:) = h_new(i+1,j,:)
00886           endif
00887         endif
00888       endif
00889       call remapping_core_h(cs_remapping, nz, h1, u(i,j,:), nz, h2, &
00890                             u_column, h_neglect, h_neglect_edge)
00891       u(i,j,:) = u_column(:)
00892     endif ; enddo ; enddo
00893   endif
00894
00895   if (show_call_tree) call calltree_waypoint("u remapped (remap_all_state_vars)")
00896
00897   ! Remap v velocity component
00898   if ( present(v) ) then
00899     !$OMP parallel do default(shared) private(h1,h2,dx,u_column)
00900     do j = g%jscB,g%jecB ; do i = g%isc,g%iec ; if (g%mask2dCv(i,j)>0.) then
00901       ! Build the start and final grids
00902       h1(:) = 0.5 * ( h_old(i,j,:) + h_old(i,j+1,:) )
00903       if (cs_ale%remap_uv_using_old_alg) then
00904         dx(:) = 0.5 * ( dxinterface(i,j,:) + dxinterface(i,j+1,:) )
00905         do k = 1, nz
00906           h2(k) = max( 0., h1(k) + ( dx(k+1) - dx(k) ) )
00907         enddo
00908       else
00909         h2(:) = 0.5 * ( h_new(i,j,:) + h_new(i,j+1,:) )
00910       endif
00911       if (associated(obc)) then
00912         if (obc%segnum_v(i,j) .ne. 0) then
00913           if (obc%segment(obc%segnum_v(i,j))%direction == obc_direction_n) then
00914             h1(:) = h_old(i,j,:)
00915             h2(:) = h_new(i,j,:)
00916           else ! (OBC%segment(n)%direction == OBC_DIRECTION_S)
00917             h1(:) = h_old(i,j+1,:)
00918             h2(:) = h_new(i,j+1,:)
00919           endif
00920         endif
00921       endif
00922       call remapping_core_h(cs_remapping, nz, h1, v(i,j,:), nz, h2, &
00923                             u_column, h_neglect, h_neglect_edge)
00924       v(i,j,:) = u_column(:)
00925     endif ; enddo ; enddo
00926   endif
00927
00928   if (cs_ale%id_vert_remap_h > 0) call post_data(cs_ale%id_vert_remap_h, h_old, cs_ale%diag)
00929   if ((cs_ale%id_vert_remap_h_tendency > 0) .and. present(dt)) then
00930     do k = 1, nz ; do j = g%jsc,g%jec ; do i = g%isc,g%iec
00931       work_cont(i,j,k) = (h_new(i,j,k) - h_old(i,j,k))*idt
00932     enddo ; enddo ; enddo
```

```
00933     call post_data(cs_ale%id_vert_remap_h_tendency, work_cont, cs_ale%diag)
00934   endif
00935   if (show_call_tree) call calltree_waypoint("v remapped (remap_all_state_vars)")
00936   if (show_call_tree) call calltree_leave("remap_all_state_vars()")
00937
```

42.2.2.23 ts_plm_edge_values()

```
subroutine, public mom_ale::ts_plm_edge_values (
              type(ale_cs), intent(inout) CS,
              real, dimension(szi_(g),szj_(g),szk_(gv)), intent(inout) S_t,
              real, dimension(szi_(g),szj_(g),szk_(gv)), intent(inout) S_b,
              real, dimension(szi_(g),szj_(g),szk_(gv)), intent(inout) T_t,
              real, dimension(szi_(g),szj_(g),szk_(gv)), intent(inout) T_b,
              type(ocean_grid_type), intent(in) G,
              type(verticalgrid_type), intent(in) GV,
              type(thermo_var_ptrs), intent(in) tv,
              real, dimension(szi_(g),szj_(g),szk_(gv)), intent(in) h,
              logical, intent(in) bdry_extrap )
```

Calculate edge values (top and bottom of layer) for T and S consistent with a PLM reconstruction in the vertical direction. Boundary reconstructions are PCM unless bdry_extrap is true.

Parameters

in	*g*	ocean grid structure
in	*gv*	Ocean vertical grid structure
in,out	*cs*	module control structure
in,out	*s_t*	Salinity at the top edge of each layer
in,out	*s_b*	Salinity at the bottom edge of each layer
in,out	*t_t*	Temperature at the top edge of each layer
in,out	*t_b*	Temperature at the bottom edge of each layer
in	*tv*	thermodynamics structure
in	*h*	layer thickness [H ~> m or kg m-2]
in	*bdry_extrap*	If true, use high-order boundary extrapolation within boundary cells

Definition at line 1014 of file MOM_ALE.F90.

References ale_plm_edge_values().

```
01014   type(ocean_grid_type),   intent(in)    :: g    !< ocean grid structure
01015   type(verticalgrid_type), intent(in)    :: gv   !< Ocean vertical grid structure
01016   type(ale_cs),            intent(inout) :: cs   !< module control structure
01017   real, dimension(SZI_(G),SZJ_(G),SZK_(GV)), &
01018                            intent(inout) :: s_t  !< Salinity at the top edge of each layer
01019   real, dimension(SZI_(G),SZJ_(G),SZK_(GV)), &
01020                            intent(inout) :: s_b  !< Salinity at the bottom edge of each layer
01021   real, dimension(SZI_(G),SZJ_(G),SZK_(GV)), &
01022                            intent(inout) :: t_t  !< Temperature at the top edge of each layer
01023   real, dimension(SZI_(G),SZJ_(G),SZK_(GV)), &
01024                            intent(inout) :: t_b  !< Temperature at the bottom edge of each layer
01025   type(thermo_var_ptrs),   intent(in)    :: tv   !< thermodynamics structure
01026   real, dimension(SZI_(G),SZJ_(G),SZK_(GV)), &
01027                            intent(in)    :: h    !< layer thickness [H ~> m or kg m-2]
01028   logical,                 intent(in)    :: bdry_extrap !< If true, use high-order boundary
01029                                                  !! extrapolation within boundary cells
```

```
01030
01031   call ale_plm_edge_values( cs, g, gv, h, tv%S, bdry_extrap, s_t, s_b )
01032   call ale_plm_edge_values( cs, g, gv, h, tv%T, bdry_extrap, t_t, t_b )
01033
```

42.2.2.24 ts_ppm_edge_values()

```
subroutine, public mom_ale::ts_ppm_edge_values (
             type(ale_cs), intent(inout) CS,
             real, dimension(szi_(g),szj_(g),szk_(gv)), intent(inout) S_t,
             real, dimension(szi_(g),szj_(g),szk_(gv)), intent(inout) S_b,
             real, dimension(szi_(g),szj_(g),szk_(gv)), intent(inout) T_t,
             real, dimension(szi_(g),szj_(g),szk_(gv)), intent(inout) T_b,
             type(ocean_grid_type), intent(in) G,
             type(verticalgrid_type), intent(in) GV,
             type(thermo_var_ptrs), intent(in) tv,
             real, dimension(szi_(g),szj_(g),szk_(gv)), intent(in) h,
             logical, intent(in) bdry_extrap )
```

Calculate edge values (top and bottom of layer) for T and S consistent with a PPM reconstruction in the vertical direction. Boundary reconstructions are PCM unless bdry_extrap is true.

Parameters

in	*g*	ocean grid structure
in	*gv*	Ocean vertical grid structure
in,out	*cs*	module control structure
in,out	*s_t*	Salinity at the top edge of each layer
in,out	*s_b*	Salinity at the bottom edge of each layer
in,out	*t_t*	Temperature at the top edge of each layer
in,out	*t_b*	Temperature at the bottom edge of each layer
in	*tv*	thermodynamics structure
in	*h*	layer thicknesses [H ~> m or kg m-2]
in	*bdry_extrap*	If true, use high-order boundary extrapolation within boundary cells

Definition at line 1100 of file MOM_ALE.F90.

```
01100   type(ocean_grid_type),   intent(in)    :: g    !< ocean grid structure
01101   type(verticalgrid_type), intent(in)    :: gv   !< Ocean vertical grid structure
01102   type(ale_cs),            intent(inout) :: cs   !< module control structure
01103   real, dimension(SZI_(G),SZJ_(G),SZK_(GV)), &
01104                            intent(inout) :: s_t  !< Salinity at the top edge of each layer
01105   real, dimension(SZI_(G),SZJ_(G),SZK_(GV)), &
01106                            intent(inout) :: s_b  !< Salinity at the bottom edge of each layer
01107   real, dimension(SZI_(G),SZJ_(G),SZK_(GV)), &
01108                            intent(inout) :: t_t  !< Temperature at the top edge of each layer
01109   real, dimension(SZI_(G),SZJ_(G),SZK_(GV)), &
01110                            intent(inout) :: t_b  !< Temperature at the bottom edge of each layer
01111   type(thermo_var_ptrs),   intent(in)    :: tv   !< thermodynamics structure
01112   real, dimension(SZI_(G),SZJ_(G),SZK_(GV)), &
01113                            intent(in)    :: h    !< layer thicknesses [H ~> m or kg m-2]
01114   logical,                 intent(in)    :: bdry_extrap !< If true, use high-order boundary
01115                                                  !! extrapolation within boundary cells
01116
01117   ! Local variables
01118   integer :: i, j, k
01119   real    :: htmp(gv%ke) ! A 1-d copy of h [H ~> m or kg m-2]
```

```
01120   real    :: tmp(gv%ke)  ! A 1-d copy of a column of temperature [degC] or salinity [ppt]
01121   real, dimension(CS%nk,2) :: &
01122       ppol_e             ! Edge value of polynomial in [degC] or [ppt]
01123   real, dimension(CS%nk,3) :: &
01124       ppol_coefs         ! Coefficients of polynomial, all in [degC] or [ppt]
01125   real :: h_neglect, h_neglect_edge ! Tiny thicknesses [H ~> m or kg m-2]
01126
01127   if (.not.cs%answers_2018) then
01128     h_neglect = gv%H_subroundoff ; h_neglect_edge = gv%H_subroundoff
01129   elseif (gv%Boussinesq) then
01130     h_neglect = gv%m_to_H*1.0e-30 ; h_neglect_edge = gv%m_to_H*1.0e-10
01131   else
01132     h_neglect = gv%kg_m2_to_H*1.0e-30 ; h_neglect_edge = gv%kg_m2_to_H*1.0e-10
01133   endif
01134
01135   ! Determine reconstruction within each column
01136   !$OMP parallel do default(shared) private(hTmp,tmp,ppol_E,ppol_coefs)
01137   do j = g%jsc-1,g%jec+1 ; do i = g%isc-1,g%iec+1
01138
01139     ! Build current grid
01140     htmp(:) = h(i,j,:)
01141     tmp(:) = tv%S(i,j,:)
01142
01143     ! Reconstruct salinity profile
01144     ppol_e(:,:) = 0.0
01145     ppol_coefs(:,:) = 0.0
01146     call edge_values_implicit_h4( gv%ke, htmp, tmp, ppol_e, h_neglect=h_neglect_edge, &
01147                                   answers_2018=cs%answers_2018 )
01148     call ppm_reconstruction( gv%ke, htmp, tmp, ppol_e, ppol_coefs, h_neglect, &
01149                                   answers_2018=cs%answers_2018 )
01150     if (bdry_extrap) &
01151       call ppm_boundary_extrapolation( gv%ke, htmp, tmp, ppol_e, ppol_coefs, h_neglect )
01152
01153     do k = 1,gv%ke
01154       s_t(i,j,k) = ppol_e(k,1)
01155       s_b(i,j,k) = ppol_e(k,2)
01156     enddo
01157
01158     ! Reconstruct temperature profile
01159     ppol_e(:,:) = 0.0
01160     ppol_coefs(:,:) = 0.0
01161     tmp(:) = tv%T(i,j,:)
01162     if (cs%answers_2018) then
01163       call edge_values_implicit_h4( gv%ke, htmp, tmp, ppol_e, h_neglect=1.0e-10*gv%m_to_H, &
01164                                     answers_2018=cs%answers_2018 )
01165     else
01166       call edge_values_implicit_h4( gv%ke, htmp, tmp, ppol_e, h_neglect=gv%H_subroundoff, &
01167                                     answers_2018=cs%answers_2018 )
01168     endif
01169     call ppm_reconstruction( gv%ke, htmp, tmp, ppol_e, ppol_coefs, h_neglect, &
01170                                   answers_2018=cs%answers_2018 )
01171     if (bdry_extrap) &
01172       call ppm_boundary_extrapolation(gv%ke, htmp, tmp, ppol_e, ppol_coefs, h_neglect )
01173
01174     do k = 1,gv%ke
01175       t_t(i,j,k) = ppol_e(k,1)
01176       t_b(i,j,k) = ppol_e(k,2)
01177     enddo
01178
01179   enddo ; enddo
01180
```

42.3 mom_eos Module Reference

Provides subroutines for quantities specific to the equation of state.

Data Types

- interface calculate_compress

  Calculates the compressibility of water from T, S, and P.
- interface calculate_density

  Calculates density of sea water from T, S and P.
- interface calculate_density_derivs

Calculate the derivatives of density with temperature and salinity from T, S, and P.

- interface calculate_density_second_derivs

  Calculates the second derivatives of density with various combinations of temperature, salinity, and pressure from T, S and P.
- interface calculate_spec_vol

  Calculates specific volume of sea water from T, S and P.
- interface calculate_specific_vol_derivs

  Calculate the derivatives of specific volume with temperature and salinity from T, S, and P.
- interface calculate_tfreeze

  Calculates the freezing point of sea water from T, S and P.
- type eos_type

  A control structure for the equation of state.

Functions/Subroutines

- subroutine calculate_density_scalar (T, S, pressure, rho, EOS, rho_ref, scale)

  Calls the appropriate subroutine to calculate density of sea water for scalar inputs. If rho_ref is present, the anomaly with respect to rho_ref is returned. The pressure and density can be rescaled with the US. If both the US and scale arguments are present the density scaling uses the product of the two scaling factors.
- subroutine calculate_stanley_density_scalar (T, S, pressure, Tvar, TScov, Svar, rho, EOS, rho_ref, scale)

  Calls the appropriate subroutine to calculate density of sea water for scalar inputs including the variance of T, S and covariance of T-S. The calculation uses only the second order correction in a series as discussed in Stanley et al., 2020. If rho_ref is present, the anomaly with respect to rho_ref is returned. The density can be rescaled using rho_ref.
- subroutine calculate_density_array (T, S, pressure, rho, start, npts, EOS, rho_ref, scale)

  Calls the appropriate subroutine to calculate the density of sea water for 1-D array inputs. If rho_ref is present, the anomaly with respect to rho_ref is returned.
- subroutine calculate_stanley_density_array (T, S, pressure, Tvar, TScov, Svar, rho, start, npts, EOS, rho_ref, scale)

  Calls the appropriate subroutine to calculate the density of sea water for 1-D array inputs including the variance of T, S and covariance of T-S. The calculation uses only the second order correction in a series as discussed in Stanley et al., 2020. If rho_ref is present, the anomaly with respect to rho_ref is returned.
- subroutine calculate_density_1d (T, S, pressure, rho, EOS, dom, rho_ref, scale)

  Calls the appropriate subroutine to calculate the density of sea water for 1-D array inputs, potentially limiting the domain of indices that are worked on. If rho_ref is present, the anomaly with respect to rho_ref is returned.
- subroutine calculate_stanley_density_1d (T, S, pressure, Tvar, TScov, Svar, rho, EOS, dom, rho_ref, scale)

  Calls the appropriate subroutine to calculate the density of sea water for 1-D array inputs including the variance of T, S and covariance of T-S, potentially limiting the domain of indices that are worked on. The calculation uses only the second order correction in a series as discussed in Stanley et al., 2020. If rho_ref is present, the anomaly with respect to rho_ref is returned.
- subroutine calculate_spec_vol_array (T, S, pressure, specvol, start, npts, EOS, spv_ref, scale)

  Calls the appropriate subroutine to calculate the specific volume of sea water for 1-D array inputs.
- subroutine calc_spec_vol_scalar (T, S, pressure, specvol, EOS, spv_ref, scale)

  Calls the appropriate subroutine to calculate specific volume of sea water for scalar inputs.
- subroutine calc_spec_vol_1d (T, S, pressure, specvol, EOS, dom, spv_ref, scale)

  Calls the appropriate subroutine to calculate the specific volume of sea water for 1-D array inputs, potentially limiting the domain of indices that are worked on.
- subroutine calculate_tfreeze_scalar (S, pressure, T_fr, EOS, pres_scale)

  Calls the appropriate subroutine to calculate the freezing point for scalar inputs.
- subroutine calculate_tfreeze_array (S, pressure, T_fr, start, npts, EOS, pres_scale)

  Calls the appropriate subroutine to calculate the freezing point for a 1-D array.
- subroutine calculate_density_derivs_array (T, S, pressure, drho_dT, drho_dS, start, npts, EOS, scale)

  Calls the appropriate subroutine to calculate density derivatives for 1-D array inputs.
- subroutine calculate_density_derivs_1d (T, S, pressure, drho_dT, drho_dS, EOS, dom, scale)

Calls the appropriate subroutine to calculate density derivatives for 1-D array inputs.

- subroutine calculate_density_derivs_scalar (T, S, pressure, drho_dT, drho_dS, EOS, scale)

  Calls the appropriate subroutines to calculate density derivatives by promoting a scalar to a one-element array.
- subroutine calculate_density_second_derivs_array (T, S, pressure, drho_dS_dS, drho_dS_dT, drho_dT_dT, drho_dS_dP, drho_dT_dP, start, npts, EOS, scale)

  Calls the appropriate subroutine to calculate density second derivatives for 1-D array inputs.
- subroutine calculate_density_second_derivs_scalar (T, S, pressure, drho_dS_dS, drho_dS_dT, drho_dT_dT, drho_dS_dP, drho_dT_dP, EOS, scale)

  Calls the appropriate subroutine to calculate density second derivatives for scalar nputs.
- subroutine calculate_spec_vol_derivs_array (T, S, pressure, dSV_dT, dSV_dS, start, npts, EOS)

  Calls the appropriate subroutine to calculate specific volume derivatives for an array.
- subroutine calc_spec_vol_derivs_1d (T, S, pressure, dSV_dT, dSV_dS, EOS, dom, scale)

  Calls the appropriate subroutine to calculate specific volume derivatives for 1-d array inputs, potentially limiting the domain of indices that are worked on.
- subroutine calculate_compress_array (T, S, press, rho, drho_dp, start, npts, EOS)

  Calls the appropriate subroutine to calculate the density and compressibility for 1-D array inputs. If US is present, the units of the inputs and outputs are rescaled.
- subroutine calculate_compress_scalar (T, S, pressure, rho, drho_dp, EOS)

  Calculate density and compressibility for a scalar. This just promotes the scalar to an array with a singleton dimension and calls calculate_compress_array. If US is present, the units of the inputs and outputs are rescaled.
- integer function, dimension(2), public eos_domain (HI, halo)

  This subroutine returns a two point integer array indicating the domain of i-indices to work on in EOS calls based on information from a hor_index type.
- subroutine, public analytic_int_specific_vol_dp (T, S, p_t, p_b, alpha_ref, HI, EOS, dza, intp_dza, intx_dza, inty_dza, halo_size, bathyP, dP_tiny, useMassWghtInterp)

  Calls the appropriate subroutine to calculate analytical and nearly-analytical integrals in pressure across layers of geopotential anomalies, which are required for calculating the finite-volume form pressure accelerations in a non-Boussinesq model. There are essentially no free assumptions, apart from the use of Boole's rule to do the horizontal integrals, and from a truncation in the series for log(1-eps/1+eps) that assumes that $|eps| < 0.34$.
- subroutine, public analytic_int_density_dz (T, S, z_t, z_b, rho_ref, rho_0, G_e, HI, EOS, dpa, intz_dpa, intx_dpa, inty_dpa, bathyT, dz_neglect, useMassWghtInterp)

  This subroutine calculates analytical and nearly-analytical integrals of pressure anomalies across layers, which are required for calculating the finite-volume form pressure accelerations in a Boussinesq model.
- logical function, public query_compressible (EOS)

  Returns true if the equation of state is compressible (i.e. has pressure dependence)
- subroutine, public eos_init (param_file, EOS, US)

  Initializes EOS_type by allocating and reading parameters.
- subroutine, public eos_manual_init (EOS, form_of_EOS, form_of_TFreeze, EOS_quadrature, Compressible, Rho_T0_S0, drho_dT, dRho_dS, TFr_S0_P0, dTFr_dS, dTFr_dp)

  Manually initialized an EOS type (intended for unit testing of routines which need a specific EOS)
- subroutine, public eos_allocate (EOS)

  Allocates EOS_type.
- subroutine, public eos_end (EOS)

  Deallocates EOS_type.
- subroutine, public eos_use_linear (Rho_T0_S0, dRho_dT, dRho_dS, EOS, use_quadrature)

  Set equation of state structure (EOS) to linear with given coefficients.
- subroutine, public convert_temp_salt_for_teos10 (T, S, HI, kd, mask_z, EOS)

  Convert T&S to Absolute Salinity and Conservative Temperature if using TEOS10.
- logical function, public eos_quadrature (EOS)

  Return value of EOS_quadrature.
- subroutine, public extract_member_eos (EOS, form_of_EOS, form_of_TFreeze, EOS_quadrature, Compressible, Rho_T0_S0, drho_dT, dRho_dS, TFr_S0_P0, dTFr_dS, dTFr_dp)

  Extractor routine for the EOS type if the members need to be accessed outside this module.

Variables

- integer, parameter, public eos_linear = 1

  A named integer specifying an equation of state.
- integer, parameter, public eos_unesco = 2

  A named integer specifying an equation of state.
- integer, parameter, public eos_wright = 3

  A named integer specifying an equation of state.
- integer, parameter, public eos_teos10 = 4

  A named integer specifying an equation of state.
- integer, parameter, public eos_nemo = 5

  A named integer specifying an equation of state.
- character ∗(10), parameter eos_linear_string = "LINEAR"

  A string for specifying the equation of state.
- character ∗(10), parameter eos_unesco_string = "UNESCO"

  A string for specifying the equation of state.
- character ∗(10), parameter eos_wright_string = "WRIGHT"

  A string for specifying the equation of state.
- character ∗(10), parameter eos_teos10_string = "TEOS10"

  A string for specifying the equation of state.
- character ∗(10), parameter eos_nemo_string = "NEMO"

  A string for specifying the equation of state.
- character ∗(10), parameter eos_default = EOS_WRIGHT_STRING

  The default equation of state.
- integer, parameter tfreeze_linear = 1

  A named integer specifying a freezing point expression.
- integer, parameter tfreeze_millero = 2

  A named integer specifying a freezing point expression.
- integer, parameter tfreeze_teos10 = 3

  A named integer specifying a freezing point expression.
- character ∗(10), parameter tfreeze_linear_string = "LINEAR"

  A string for specifying the freezing point expression.
- character ∗(10), parameter tfreeze_millero_string = "MILLERO_78"

  A string for specifying freezing point expression.
- character ∗(10), parameter tfreeze_teos10_string = "TEOS10"

  A string for specifying the freezing point expression.
- character ∗(10), parameter tfreeze_default = TFREEZE_LINEAR_STRING

  The default freezing point expression.

42.3.1 Detailed Description

Provides subroutines for quantities specific to the equation of state.

The MOM_EOS module is a wrapper for various equations of state (e.g. Linear, Wright, UNESCO) and provides a uniform interface to the rest of the model independent of which equation of state is being used.

42.3.2 Function/Subroutine Documentation

42.3.2.1 analytic_int_density_dz()

```
subroutine, public mom_eos::analytic_int_density_dz (
             real, dimension(hi%isd:hi%ied,hi%jsd:hi%jed), intent(in) T,
             real, dimension(hi%isd:hi%ied,hi%jsd:hi%jed), intent(in) S,
             real, dimension(hi%isd:hi%ied,hi%jsd:hi%jed), intent(in) z_t,
             real, dimension(hi%isd:hi%ied,hi%jsd:hi%jed), intent(in) z_b,
             real, intent(in) rho_ref,
             real, intent(in) rho_0,
             real, intent(in) G_e,
             type(hor_index_type), intent(in) HI,
             type(eos_type), pointer EOS,
             real, dimension(hi%isd:hi%ied,hi%jsd:hi%jed), intent(inout) dpa,
             real, dimension(hi%isd:hi%ied,hi%jsd:hi%jed), intent(inout), optional intz_dpa,
             real, dimension(hi%isdb:hi%iedb,hi%jsd:hi%jed), intent(inout), optional intx_dpa,
             real, dimension(hi%isd:hi%ied,hi%jsdb:hi%jedb), intent(inout), optional inty_dpa,
             real, dimension(hi%isd:hi%ied,hi%jsd:hi%jed), intent(in), optional bathyT,
             real, intent(in), optional dz_neglect,
             logical, intent(in), optional useMassWghtInterp )
```

This subroutine calculates analytical and nearly-analytical integrals of pressure anomalies across layers, which are required for calculating the finite-volume form pressure accelerations in a Boussinesq model.

Parameters

in	*hi*	Ocean horizontal index structure
in	*t*	Potential temperature referenced to the surface [degC]
in	*s*	Salinity [ppt]
in	*z_t*	Height at the top of the layer in depth units [Z ~> m]
in	*z_b*	Height at the bottom of the layer [Z ~> m]
in	*rho_ref*	A mean density [R ~> kg m-3] or [kg m-3], that is subtracted out to reduce the magnitude of each of the integrals.
in	*rho_0*	A density [R ~> kg m-3] or [kg m-3], that is used to calculate the pressure (as p~=-z*rho_0*G_e) used in the equation of state.
in	*g_e*	The Earth's gravitational acceleration [L2 Z-1 T-2 ~> m s-2] or [m2 Z-1 s-2 ~> m s-2]
	eos	Equation of state structure
in,out	*dpa*	The change in the pressure anomaly
in,out	*intz_dpa*	The integral through the thickness of the
in,out	*intx_dpa*	The integral in x of the difference between
in,out	*inty_dpa*	The integral in y of the difference between
in	*bathyt*	The depth of the bathymetry [Z ~> m]
in	*dz_neglect*	A miniscule thickness change [Z ~> m]
in	*usemasswghtinterp*	If true, uses mass weighting to interpolate T/S for top and bottom integrals.

Definition at line 1257 of file MOM_EOS.F90.

References eos_linear, and eos_wright.

```
01257   type(hor_index_type), intent(in)  :: hi !< Ocean horizontal index structure
01258   real, dimension(HI%isd:HI%ied,HI%jsd:HI%jed), &
01259                         intent(in)  :: t   !< Potential temperature referenced to the surface [degC]
01260   real, dimension(HI%isd:HI%ied,HI%jsd:HI%jed), &
01261                         intent(in)  :: s   !< Salinity [ppt]
```

```
01262   real, dimension(HI%isd:HI%ied,HI%jsd:HI%jed), &
01263                         intent(in)  :: z_t !< Height at the top of the layer in depth units [Z ~> m]
01264   real, dimension(HI%isd:HI%ied,HI%jsd:HI%jed), &
01265                         intent(in)  :: z_b !< Height at the bottom of the layer [Z ~> m]
01266   real,                 intent(in)  :: rho_ref !< A mean density [R ~> kg m-3] or [kg m-3], that is
01267                                           !! subtracted out to reduce the magnitude of each of the
01268                                           !! integrals.
01269   real,                 intent(in)  :: rho_0 !< A density [R ~> kg m-3] or [kg m-3], that is used
01270                                           !! to calculate the pressure (as p~=-z*rho_0*G_e)
01271                                           !! used in the equation of state.
01272   real,                 intent(in)  :: g_e !< The Earth's gravitational acceleration
01273                                           !! [L2 Z-1 T-2 ~> m s-2] or [m2 Z-1 s-2 ~> m s-2]
01274   type(eos_type),       pointer     :: eos !< Equation of state structure
01275   real, dimension(HI%isd:HI%ied,HI%jsd:HI%jed), &
01276                       intent(inout) :: dpa !< The change in the pressure anomaly
01277                                           !! across the layer [R L2 T-2 ~> Pa] or [Pa]
01278   real, dimension(HI%isd:HI%ied,HI%jsd:HI%jed), &
01279             optional, intent(inout) :: intz_dpa !< The integral through the thickness of the
01280                                           !! layer of the pressure anomaly relative to the
01281                                           !! anomaly at the top of the layer [R L2 Z T-2 ~> Pa m]
01282   real, dimension(HI%IsdB:HI%IedB,HI%jsd:HI%jed), &
01283             optional, intent(inout) :: intx_dpa !< The integral in x of the difference between
01284                                          !! the pressure anomaly at the top and bottom of the
01285                                          !! layer divided by the x grid spacing [R L2 T-2 ~> Pa]
01286   real, dimension(HI%isd:HI%ied,HI%JsdB:HI%JedB), &
01287             optional, intent(inout) :: inty_dpa !< The integral in y of the difference between
01288                                          !! the pressure anomaly at the top and bottom of the
01289                                          !! layer divided by the y grid spacing [R L2 T-2 ~> Pa]
01290   real, dimension(HI%isd:HI%ied,HI%jsd:HI%jed), &
01291                optional, intent(in)  :: bathyt !< The depth of the bathymetry [Z ~> m]
01292   real,        optional, intent(in)  :: dz_neglect !< A miniscule thickness change [Z ~> m]
01293   logical,     optional, intent(in)  :: usemasswghtinterp !< If true, uses mass weighting to
01294                                           !! interpolate T/S for top and bottom integrals.
01295   ! Local variables
01296   real :: rho_scale  ! A multiplicative factor by which to scale density from kg m-3 to the
01297                      ! desired units [R m3 kg-1 ~> 1]
01298   real :: pres_scale ! A multiplicative factor to convert pressure into Pa [Pa T2 R-1 L-2 ~> 1]
01299
01300   if (.not.associated(eos)) call mom_error(fatal, &
01301     "int_density_dz called with an unassociated EOS_type EOS.")
01302
01303   ! We should never reach this point with quadrature. EOS_quadrature indicates that numerical
01304   ! integration be used instead of analytic. This is a safety check.
01305   if (eos%EOS_quadrature) call mom_error(fatal, "EOS_quadrature is set!")
01306
01307   select case (eos%form_of_EOS)
01308     case (eos_linear)
01309       rho_scale = eos%kg_m3_to_R
01310       if (rho_scale /= 1.0) then
01311         call int_density_dz_linear(t, s, z_t, z_b, rho_ref, rho_0, g_e, hi, &
01312                           rho_scale*eos%Rho_T0_S0, rho_scale*eos%dRho_dT, rho_scale*eos%dRho_dS, &
01313                           dpa, intz_dpa, intx_dpa, inty_dpa, bathyt, dz_neglect, usemasswghtinterp)
01314       else
01315         call int_density_dz_linear(t, s, z_t, z_b, rho_ref, rho_0, g_e, hi, &
01316                           eos%Rho_T0_S0, eos%dRho_dT, eos%dRho_dS, &
01317                           dpa, intz_dpa, intx_dpa, inty_dpa, bathyt, dz_neglect, usemasswghtinterp)
01318       endif
01319     case (eos_wright)
01320       rho_scale = eos%kg_m3_to_R
01321       pres_scale = eos%RL2_T2_to_Pa
01322       if ((rho_scale /= 1.0) .or. (pres_scale /= 1.0)) then
01323         call int_density_dz_wright(t, s, z_t, z_b, rho_ref, rho_0, g_e, hi, &
01324                                    dpa, intz_dpa, intx_dpa, inty_dpa, bathyt, &
01325                                    dz_neglect, usemasswghtinterp, rho_scale, pres_scale)
01326       else
01327         call int_density_dz_wright(t, s, z_t, z_b, rho_ref, rho_0, g_e, hi, &
01328                                    dpa, intz_dpa, intx_dpa, inty_dpa, bathyt, &
01329                                    dz_neglect, usemasswghtinterp)
01330       endif
01331     case default
01332       call mom_error(fatal, "No analytic integration option is available with this EOS!")
01333   end select
01334
```

42.3.2.2 analytic_int_specific_vol_dp()

```
subroutine, public mom_eos::analytic_int_specific_vol_dp (
             real, dimension(hi%isd:hi%ied,hi%jsd:hi%jed), intent(in) T,
```

```
real, dimension(hi%isd:hi%ied,hi%jsd:hi%jed), intent(in) S,
real, dimension(hi%isd:hi%ied,hi%jsd:hi%jed), intent(in) p_t,
real, dimension(hi%isd:hi%ied,hi%jsd:hi%jed), intent(in) p_b,
real, intent(in) alpha_ref,
type(hor_index_type), intent(in) HI,
type(eos_type), pointer EOS,
real, dimension(hi%isd:hi%ied,hi%jsd:hi%jed), intent(inout) dza,
real, dimension(hi%isd:hi%ied,hi%jsd:hi%jed), intent(inout), optional intp_dza,
real, dimension(hi%isdb:hi%iedb,hi%jsd:hi%jed), intent(inout), optional intx_dza,
real, dimension(hi%isd:hi%ied,hi%jsdb:hi%jedb), intent(inout), optional inty_dza,
integer, intent(in), optional halo_size,
real, dimension(hi%isd:hi%ied,hi%jsd:hi%jed), intent(in), optional bathyP,
real, intent(in), optional dP_tiny,
logical, intent(in), optional useMassWghtInterp )
```

Calls the appropriate subroutine to calculate analytical and nearly-analytical integrals in pressure across layers of geopotential anomalies, which are required for calculating the finite-volume form pressure accelerations in a non-↩ Boussinesq model. There are essentially no free assumptions, apart from the use of Boole's rule to do the horizontal integrals, and from a truncation in the series for log(1-eps/1+eps) that assumes that $|\text{eps}| < 0.34$.

Parameters

in	*hi*	The horizontal index structure
in	*t*	Potential temperature referenced to the surface [degC]
in	*s*	Salinity [ppt]
in	*p_t*	Pressure at the top of the layer [R L2 T-2 ~> Pa] or [Pa]
in	*p_b*	Pressure at the bottom of the layer [R L2 T-2 ~> Pa] or [Pa]
in	*alpha_ref*	A mean specific volume that is subtracted out to reduce the magnitude of each of the integrals [R-1 ~> m3 kg-1] The calculation is mathematically identical with different values of alpha_ref, but this reduces the effects of roundoff.
	eos	Equation of state structure
in,out	*dza*	The change in the geopotential anomaly across
in,out	*intp_dza*	The integral in pressure through the layer of the
in,out	*intx_dza*	The integral in x of the difference between the
in,out	*inty_dza*	The integral in y of the difference between the
in	*halo_size*	The width of halo points on which to calculate dza.
in	*bathyp*	The pressure at the bathymetry [R L2 T-2 ~> Pa] or [Pa]
in	*dp_tiny*	A miniscule pressure change with the same units as p_t [R L2 T-2 ~> Pa] or [Pa]
in	*usemasswghtinterp*	If true, uses mass weighting to interpolate T/S for top and bottom integrals.

Definition at line 1188 of file MOM_EOS.F90.

References eos_linear, and eos_wright.

```
01188   type(hor_index_type), intent(in)  :: hi  !< The horizontal index structure
01189   real, dimension(HI%isd:HI%ied,HI%jsd:HI%jed), &
01190                         intent(in)  :: t   !< Potential temperature referenced to the surface [degC]
01191   real, dimension(HI%isd:HI%ied,HI%jsd:HI%jed), &
01192                         intent(in)  :: s   !< Salinity [ppt]
01193   real, dimension(HI%isd:HI%ied,HI%jsd:HI%jed), &
01194                         intent(in)  :: p_t !< Pressure at the top of the layer [R L2 T-2 ~> Pa] or [Pa]
01195   real, dimension(HI%isd:HI%ied,HI%jsd:HI%jed), &
01196                         intent(in)  :: p_b !< Pressure at the bottom of the layer [R L2 T-2 ~> Pa] or [Pa]
01197   real,                 intent(in)  :: alpha_ref !< A mean specific volume that is subtracted out
01198                             !! to reduce the magnitude of each of the integrals [R-1 ~> m3 kg-1]
```

```
01199                           !! The calculation is mathematically identical with different values of
01200                           !! alpha_ref, but this reduces the effects of roundoff.
01201   type(eos_type),       pointer     :: eos !< Equation of state structure
01202   real, dimension(HI%isd:HI%ied,HI%jsd:HI%jed), &
01203                         intent(inout) :: dza !< The change in the geopotential anomaly across
01204                           !! the layer [L2 T-2 ~> m2 s-2] or [m2 s-2]
01205   real, dimension(HI%isd:HI%ied,HI%jsd:HI%jed), &
01206               optional, intent(inout) :: intp_dza !< The integral in pressure through the layer of the
01207                           !! geopotential anomaly relative to the anomaly at the bottom of the
01208                           !! layer [R L4 T-4 ~> Pa m2 s-2] or [Pa m2 s-2]
01209   real, dimension(HI%IsdB:HI%IedB,HI%jsd:HI%jed), &
01210               optional, intent(inout) :: intx_dza !< The integral in x of the difference between the
01211                           !! geopotential anomaly at the top and bottom of the layer divided by
01212                           !! the x grid spacing [L2 T-2 ~> m2 s-2] or [m2 s-2]
01213   real, dimension(HI%isd:HI%ied,HI%JsdB:HI%JedB), &
01214               optional, intent(inout) :: inty_dza !< The integral in y of the difference between the
01215                           !! geopotential anomaly at the top and bottom of the layer divided by
01216                           !! the y grid spacing [L2 T-2 ~> m2 s-2] or [m2 s-2]
01217   integer,    optional, intent(in)  :: halo_size !< The width of halo points on which to calculate dza.
01218   real, dimension(HI%isd:HI%ied,HI%jsd:HI%jed), &
01219               optional, intent(in)  :: bathyp  !< The pressure at the bathymetry [R L2 T-2 ~> Pa] or [Pa]
01220   real,       optional, intent(in)  :: dp_tiny !< A miniscule pressure change with
01221                           !! the same units as p_t [R L2 T-2 ~> Pa] or [Pa]
01222   logical,    optional, intent(in)  :: usemasswghtinterp !< If true, uses mass weighting
01223                           !! to interpolate T/S for top and bottom integrals.
01224   ! Local variables
01225   real :: pres_scale    ! A unit conversion factor from the rescaled units of pressure to Pa [Pa T2 R-1 L-2
      ~> 1]
01226   real :: sv_scale      ! A multiplicative factor by which to scale specific
01227                         ! volume from m3 kg-1 to the desired units [kg m-3 R-1 ~> 1]
01228
01229   if (.not.associated(eos)) call mom_error(fatal, &
01230     "int_specific_vol_dp called with an unassociated EOS_type EOS.")
01231
01232   ! We should never reach this point with quadrature. EOS_quadrature indicates that numerical
01233   ! integration be used instead of analytic. This is a safety check.
01234   if (eos%EOS_quadrature) call mom_error(fatal, "EOS_quadrature is set!")
01235
01236   select case (eos%form_of_EOS)
01237     case (eos_linear)
01238       call int_spec_vol_dp_linear(t, s, p_t, p_b, alpha_ref, hi, eos%kg_m3_to_R*eos%Rho_T0_S0, &
01239                                   eos%kg_m3_to_R*eos%dRho_dT, eos%kg_m3_to_R*eos%dRho_dS, dza, &
01240                                   intp_dza, intx_dza, inty_dza, halo_size, &
01241                                   bathyp, dp_tiny, usemasswghtinterp)
01242     case (eos_wright)
01243       call int_spec_vol_dp_wright(t, s, p_t, p_b, alpha_ref, hi, dza, intp_dza, intx_dza, &
01244                                     inty_dza, halo_size, bathyp, dp_tiny, usemasswghtinterp, &
01245                                     sv_scale=eos%R_to_kg_m3, pres_scale=eos%RL2_T2_to_Pa)
01246     case default
01247       call mom_error(fatal, "No analytic integration option is available with this EOS!")
01248   end select
01249
```

42.3.2.3 calc_spec_vol_1d()

```
subroutine mom_eos::calc_spec_vol_1d (
             real, dimension(:), intent(in) T,
             real, dimension(:), intent(in) S,
             real, dimension(:), intent(in) pressure,
             real, dimension(:), intent(inout) specvol,
             type(eos_type), pointer EOS,
             integer, dimension(2), intent(in), optional dom,
             real, intent(in), optional spv_ref,
             real, intent(in), optional scale )  [private]
```

Calls the appropriate subroutine to calculate the specific volume of sea water for 1-D array inputs, potentially limiting the domain of indices that are worked on.

Parameters

in	*t*	Potential temperature referenced to the surface [degC]
in	*s*	Salinity [ppt]
in	*pressure*	Pressure [R L2 T-2 ~> Pa]
in,out	*specvol*	In situ specific volume [R-1 ~> m3 kg-1]
	eos	Equation of state structure
in	*dom*	The domain of indices to work on, taking into account that arrays start at 1.
in	*spv_ref*	A reference specific volume [R-1 ~> m3 kg-1]
in	*scale*	A multiplicative factor by which to scale output specific volume in combination with scaling given by US [various]

Definition at line 571 of file MOM_EOS.F90.

References calculate_spec_vol_array().

```
00571   real, dimension(:),    intent(in)    :: t        !< Potential temperature referenced to the surface
      [degC]
00572   real, dimension(:),    intent(in)    :: s        !< Salinity [ppt]
00573   real, dimension(:),    intent(in)    :: pressure !< Pressure [R L2 T-2 ~> Pa]
00574   real, dimension(:),    intent(inout) :: specvol  !< In situ specific volume [R-1 ~> m3 kg-1]
00575   type(eos_type),        pointer       :: eos      !< Equation of state structure
00576   integer, dimension(2), optional, intent(in) :: dom   !< The domain of indices to work on, taking
00577                                                        !! into account that arrays start at 1.
00578   real,                  optional, intent(in) :: spv_ref !< A reference specific volume [R-1 ~> m3 kg-1]
00579   real,                  optional, intent(in) :: scale !< A multiplicative factor by which to scale
00580                                                        !! output specific volume in combination with
00581                                                        !! scaling given by US [various]
00582   ! Local variables
00583   real, dimension(size(specvol)) :: pres  ! Pressure converted to [Pa]
00584   real :: p_scale   ! A factor to convert pressure to units of Pa [Pa T2 R-1 L-2 ~> 1]
00585   real :: spv_unscale ! A factor to convert specific volume from R-1 to m3 kg-1 [m3 kg-1 R ~> 1]
00586   real :: spv_scale ! A factor to convert specific volume from m3 kg-1 to the desired units [kg m-3 R-1 ~>
      1]
00587   real :: spv_reference ! spv_ref converted to [m3 kg-1]
00588   integer :: i, is, ie, npts
00589
00590   if (.not.associated(eos)) call mom_error(fatal, &
00591     "calc_spec_vol_1d called with an unassociated EOS_type EOS.")
00592
00593   if (present(dom)) then
00594     is = dom(1) ; ie = dom(2) ; npts = 1 + ie - is
00595   else
00596     is = 1 ; ie = size(specvol) ; npts = 1 + ie - is
00597   endif
00598
00599   p_scale = eos%RL2_T2_to_Pa
00600   spv_unscale = eos%kg_m3_to_R
00601
00602   if ((p_scale == 1.0) .and. (spv_unscale == 1.0)) then
00603     call calculate_spec_vol_array(t, s, pressure, specvol, is, npts, eos, spv_ref)
00604   elseif (present(spv_ref)) then ! This is the same as above, but with some extra work to rescale
      variables.
00605     do i=is,ie ; pres(i) = p_scale * pressure(i) ; enddo
00606     spv_reference = spv_unscale*spv_ref
00607     call calculate_spec_vol_array(t, s, pres, specvol, is, npts, eos, spv_reference)
00608   else  ! There is rescaling of variables, but spv_ref is not present. Passing a 0 value of spv_ref
00609         ! changes answers at roundoff for some equations of state, like Wright and UNESCO.
00610     do i=is,ie ; pres(i) = p_scale * pressure(i) ; enddo
00611     call calculate_spec_vol_array(t, s, pres, specvol, is, npts, eos)
00612   endif
00613
00614   spv_scale = eos%R_to_kg_m3
00615   if (present(scale)) spv_scale = spv_scale * scale
00616   if (spv_scale /= 1.0) then ; do i=is,ie
00617     specvol(i) = spv_scale * specvol(i)
00618   enddo ; endif
00619
```

42.3.2.4 calc_spec_vol_derivs_1d()

```
subroutine mom_eos::calc_spec_vol_derivs_1d (
             real, dimension(:), intent(in) T,
             real, dimension(:), intent(in) S,
             real, dimension(:), intent(in) pressure,
             real, dimension(:), intent(inout) dSV_dT,
             real, dimension(:), intent(inout) dSV_dS,
             type(eos_type), pointer EOS,
             integer, dimension(2), intent(in), optional dom,
             real, intent(in), optional scale )  [private]
```

Calls the appropriate subroutine to calculate specific volume derivatives for 1-d array inputs, potentially limiting the domain of indices that are worked on.

Parameters

in	*t*	Potential temperature referenced to the surface [degC]
in	*s*	Salinity [ppt]
in	*pressure*	Pressure [R L2 T-2 ~> Pa]
in,out	*dsv_dt*	The partial derivative of specific volume with potential temperature [R-1 degC-1 ~> m3 kg-1 degC-1]
in,out	*dsv_ds*	The partial derivative of specific volume with salinity [R-1 ppt-1 ~> m3 kg-1 ppt-1]
	eos	Equation of state structure
in	*dom*	The domain of indices to work on, taking into account that arrays start at 1.
in	*scale*	A multiplicative factor by which to scale specific volume in combination with scaling given by US [various]

Definition at line 1040 of file MOM_EOS.F90.

References calculate_spec_vol_derivs_array().

```
01040   real, dimension(:), intent(in)    :: t        !< Potential temperature referenced to the surface [degC]
01041   real, dimension(:), intent(in)    :: s        !< Salinity [ppt]
01042   real, dimension(:), intent(in)    :: pressure !< Pressure [R L2 T-2 ~> Pa]
01043   real, dimension(:), intent(inout) :: dsv_dt   !< The partial derivative of specific volume with potential
01044                                                 !! temperature [R-1 degC-1 ~> m3 kg-1 degC-1]
01045   real, dimension(:), intent(inout) :: dsv_ds   !< The partial derivative of specific volume with salinity
01046                                                 !! [R-1 ppt-1 ~> m3 kg-1 ppt-1]
01047   type(eos_type),     pointer       :: eos      !< Equation of state structure
01048   integer, dimension(2), optional, intent(in) :: dom   !< The domain of indices to work on, taking
01049                                                        !! into account that arrays start at 1.
01050   real,                  optional, intent(in) :: scale !< A multiplicative factor by which to scale
      specific
01051                                                 !! volume in combination with scaling given by US [various]
01052
01053   ! Local variables
01054   real, dimension(size(dSV_dT)) :: press   ! Pressure converted to [Pa]
01055   real :: spv_scale ! A factor to convert specific volume from m3 kg-1 to the desired units [kg R-1 m-3 ~>
      1]
01056   real :: p_scale   ! A factor to convert pressure to units of Pa [Pa T2 R-1 L-2 ~> 1]
01057   integer :: i, is, ie, npts
01058
01059   if (.not.associated(eos)) call mom_error(fatal, &
01060     "calculate_spec_vol_derivs_1d called with an unassociated EOS_type EOS.")
01061
01062   if (present(dom)) then
01063     is = dom(1) ; ie = dom(2) ; npts = 1 + ie - is
01064   else
01065     is = 1 ; ie = size(dsv_dt) ; npts = 1 + ie - is
01066   endif
01067   p_scale = eos%RL2_T2_to_Pa
01068
01069   if (p_scale == 1.0) then
01070     call calculate_spec_vol_derivs_array(t, s, pressure, dsv_dt, dsv_ds, is, npts, eos)
```

```
01071   else
01072     do i=is,ie ; press(i) = p_scale * pressure(i) ; enddo
01073     call calculate_spec_vol_derivs_array(t, s, press, dsv_dt, dsv_ds, is, npts, eos)
01074   endif
01075
01076   spv_scale = eos%R_to_kg_m3
01077   if (present(scale)) spv_scale = spv_scale * scale
01078   if (spv_scale /= 1.0) then ; do i=is,ie
01079     dsv_dt(i) = spv_scale * dsv_dt(i)
01080     dsv_ds(i) = spv_scale * dsv_ds(i)
01081   enddo ; endif
01082
```

42.3.2.5 calc_spec_vol_scalar()

```
subroutine mom_eos::calc_spec_vol_scalar (
             real, intent(in) T,
             real, intent(in) S,
             real, intent(in) pressure,
             real, intent(out) specvol,
             type(eos_type), pointer EOS,
             real, intent(in), optional spv_ref,
             real, intent(in), optional scale )  [private]
```

Calls the appropriate subroutine to calculate specific volume of sea water for scalar inputs.

Parameters

in	*t*	Potential temperature referenced to the surface [degC]
in	*s*	Salinity [ppt]
in	*pressure*	Pressure [Pa] or [R L2 T-2 ~> Pa]
out	*specvol*	In situ? specific volume [m3 kg-1] or [R-1 ~> m3 kg-1]
	eos	Equation of state structure
in	*spv_ref*	A reference specific volume [m3 kg-1] or [R-1 m3 kg-1]
in	*scale*	A multiplicative factor by which to scale specific volume in combination with scaling given by US [various]

Definition at line 533 of file MOM_EOS.F90.

References calculate_spec_vol_array().

```
00533   real,           intent(in)  :: t        !< Potential temperature referenced to the surface [degC]
00534   real,           intent(in)  :: s        !< Salinity [ppt]
00535   real,           intent(in)  :: pressure !< Pressure [Pa] or [R L2 T-2 ~> Pa]
00536   real,           intent(out) :: specvol  !< In situ? specific volume [m3 kg-1] or [R-1 ~> m3 kg-1]
00537   type(eos_type), pointer     :: eos      !< Equation of state structure
00538   real, optional, intent(in)  :: spv_ref  !< A reference specific volume [m3 kg-1] or [R-1 m3 kg-1]
00539   real, optional, intent(in)  :: scale    !< A multiplicative factor by which to scale specific
00540                                           !! volume in combination with scaling given by US [various]
00541
00542   real, dimension(1) :: ta, sa, pres, spv  ! Rescaled single element array versions of the arguments.
00543   real :: spv_reference ! spv_ref converted to [m3 kg-1]
00544   real :: spv_scale ! A factor to convert specific volume from m3 kg-1 to the desired units [kg R-1 m-3 ~>
      1]
00545
00546   if (.not.associated(eos)) call mom_error(fatal, &
00547     "calc_spec_vol_scalar called with an unassociated EOS_type EOS.")
00548
00549   pres(1) = eos%RL2_T2_to_Pa*pressure
00550   ta(1) = t ; sa(1) = s
```

```
00551
00552   if (present(spv_ref)) then
00553     spv_reference = eos%kg_m3_to_R*spv_ref
00554     call calculate_spec_vol_array(ta, sa, pres, spv, 1, 1, eos, spv_reference)
00555   else
00556     call calculate_spec_vol_array(ta, sa, pres, spv, 1, 1, eos)
00557   endif
00558   specvol = spv(1)
00559
00560   spv_scale = eos%R_to_kg_m3
00561   if (present(scale)) spv_scale = spv_scale * scale
00562   if (spv_scale /= 1.0) then
00563     specvol = spv_scale * specvol
00564   endif
00565
```

42.3.2.6 calculate_compress_array()

```
subroutine mom_eos::calculate_compress_array (
             real, dimension(:), intent(in) T,
             real, dimension(:), intent(in) S,
             real, dimension(:), intent(in) press,
             real, dimension(:), intent(inout) rho,
             real, dimension(:), intent(inout) drho_dp,
             integer, intent(in) start,
             integer, intent(in) npts,
             type(eos_type), pointer EOS )  [private]
```

Calls the appropriate subroutine to calculate the density and compressibility for 1-D array inputs. If US is present, the units of the inputs and outputs are rescaled.

Parameters

in	*t*	Potential temperature referenced to the surface [degC]
in	*s*	Salinity [PSU]
in	*press*	Pressure [Pa] or [R L2 T-2 ~> Pa]
in,out	*rho*	In situ density [kg m-3] or [R ~> kg m-3]
in,out	*drho_dp*	The partial derivative of density with pressure (also the inverse of the square of sound speed) [s2 m-2] or [T2 L-2]
in	*start*	Starting index within the array
in	*npts*	The number of values to calculate
	eos	Equation of state structure

Definition at line 1089 of file MOM_EOS.F90.

References eos_linear, eos_nemo, eos_teos10, eos_unesco, and eos_wright.

Referenced by calculate_compress_scalar().

```
01089   real, dimension(:), intent(in)  :: t        !< Potential temperature referenced to the surface [degC]
01090   real, dimension(:), intent(in)  :: s        !< Salinity [PSU]
01091   real, dimension(:), intent(in)  :: press    !< Pressure [Pa] or [R L2 T-2 ~> Pa]
01092   real, dimension(:), intent(inout) :: rho      !< In situ density [kg m-3] or [R ~> kg m-3]
01093   real, dimension(:), intent(inout) :: drho_dp  !< The partial derivative of density with pressure
01094                                                 !! (also the inverse of the square of sound speed)
01095                                                 !! [s2 m-2] or [T2 L-2]
01096   integer,            intent(in)  :: start    !< Starting index within the array
01097   integer,            intent(in)  :: npts     !< The number of values to calculate
```

```
01098   type(eos_type),     pointer     :: eos      !< Equation of state structure
01099
01100   ! Local variables
01101   real, dimension(size(press)) :: pressure  ! Pressure converted to [Pa]
01102   integer :: i, is, ie
01103
01104   if (.not.associated(eos)) call mom_error(fatal, &
01105     "calculate_compress called with an unassociated EOS_type EOS.")
01106
01107   is = start ; ie = is + npts - 1
01108   do i=is,ie ; pressure(i) = eos%RL2_T2_to_Pa * press(i) ; enddo
01109
01110   select case (eos%form_of_EOS)
01111     case (eos_linear)
01112       call calculate_compress_linear(t, s, pressure, rho, drho_dp, start, npts, &
01113                                      eos%Rho_T0_S0, eos%dRho_dT, eos%dRho_dS)
01114     case (eos_unesco)
01115       call calculate_compress_unesco(t, s, pressure, rho, drho_dp, start, npts)
01116     case (eos_wright)
01117       call calculate_compress_wright(t, s, pressure, rho, drho_dp, start, npts)
01118     case (eos_teos10)
01119       call calculate_compress_teos10(t, s, pressure, rho, drho_dp, start, npts)
01120     case (eos_nemo)
01121       call calculate_compress_nemo(t, s, pressure, rho, drho_dp, start, npts)
01122     case default
01123       call mom_error(fatal, "calculate_compress: EOS%form_of_EOS is not valid.")
01124   end select
01125
01126   if (eos%kg_m3_to_R /= 1.0) then ; do i=is,ie
01127     rho(i) = eos%kg_m3_to_R * rho(i)
01128   enddo ; endif
01129   if (eos%L_T_to_m_s /= 1.0) then ; do i=is,ie
01130     drho_dp(i) = eos%L_T_to_m_s**2 * drho_dp(i)
01131   enddo ; endif
01132
```

42.3.2.7 calculate_compress_scalar()

```
subroutine mom_eos::calculate_compress_scalar (
             real, intent(in) T,
             real, intent(in) S,
             real, intent(in) pressure,
             real, intent(out) rho,
             real, intent(out) drho_dp,
             type(eos_type), pointer EOS )  [private]
```

Calculate density and compressibility for a scalar. This just promotes the scalar to an array with a singleton dimension and calls calculate_compress_array. If US is present, the units of the inputs and outputs are rescaled.

Parameters

in	*t*	Potential temperature referenced to the surface [degC]
in	*s*	Salinity [ppt]
in	*pressure*	Pressure [Pa] or [R L2 T-2 ~> Pa]
out	*rho*	In situ density [kg m-3] or [R ~> kg m-3]
out	*drho_dp*	The partial derivative of density with pressure (also the inverse of the square of sound speed) [s2 m-2] or [T2 L-2]
	eos	Equation of state structure

Definition at line 1139 of file MOM_EOS.F90.

References calculate_compress_array().

```
01139   real, intent(in)         :: t        !< Potential temperature referenced to the surface [degC]
01140   real, intent(in)         :: s        !< Salinity [ppt]
01141   real, intent(in)         :: pressure !< Pressure [Pa] or [R L2 T-2 ~> Pa]
01142   real, intent(out)        :: rho      !< In situ density [kg m-3] or [R ~> kg m-3]
01143   real, intent(out)        :: drho_dp  !< The partial derivative of density with pressure (also the
01144                                        !! inverse of the square of sound speed) [s2 m-2] or [T2 L-2]
01145   type(eos_type), pointer :: eos       !< Equation of state structure
01146
01147   ! Local variables
01148   real, dimension(1) :: ta, sa, pa, rhoa, drho_dpa
01149
01150   if (.not.associated(eos)) call mom_error(fatal, &
01151     "calculate_compress called with an unassociated EOS_type EOS.")
01152   ta(1) = t ; sa(1) = s; pa(1) = pressure
01153
01154   call calculate_compress_array(ta, sa, pa, rhoa, drho_dpa, 1, 1, eos)
01155   rho = rhoa(1) ; drho_dp = drho_dpa(1)
01156
```

42.3.2.8 calculate_density_1d()

```
subroutine mom_eos::calculate_density_1d (
             real, dimension(:), intent(in) T,
             real, dimension(:), intent(in) S,
             real, dimension(:), intent(in) pressure,
             real, dimension(:), intent(inout) rho,
             type(eos_type), pointer EOS,
             integer, dimension(2), intent(in), optional dom,
             real, intent(in), optional rho_ref,
             real, intent(in), optional scale )  [private]
```

Calls the appropriate subroutine to calculate the density of sea water for 1-D array inputs, potentially limiting the domain of indices that are worked on. If rho_ref is present, the anomaly with respect to rho_ref is returned.

Parameters

in	*t*	Potential temperature referenced to the surface [degC]
in	*s*	Salinity [ppt]
in	*pressure*	Pressure [R L2 T-2 ~> Pa]
in, out	*rho*	Density (in-situ if pressure is local) [R ~> kg m-3]
	eos	Equation of state structure
in	*dom*	The domain of indices to work on, taking into account that arrays start at 1.
in	*rho_ref*	A reference density [kg m-3]
in	*scale*	A multiplicative factor by which to scale density in combination with scaling given by US [various]

Definition at line 360 of file MOM_EOS.F90.

References calculate_density_array().

```
00360   real, dimension(:),    intent(in)    :: t        !< Potential temperature referenced to the surface
       [degC]
00361   real, dimension(:),    intent(in)    :: s        !< Salinity [ppt]
00362   real, dimension(:),    intent(in)    :: pressure !< Pressure [R L2 T-2 ~> Pa]
00363   real, dimension(:),    intent(inout) :: rho      !< Density (in-situ if pressure is local) [R ~> kg m-3]
00364   type(eos_type),        pointer       :: eos      !< Equation of state structure
00365   integer, dimension(2), optional, intent(in) :: dom   !< The domain of indices to work on, taking
00366                                                        !! into account that arrays start at 1.
```

```
00367   real,                 optional, intent(in) :: rho_ref !< A reference density [kg m-3]
00368   real,                 optional, intent(in) :: scale !< A multiplicative factor by which to scale density
00369                                                   !! in combination with scaling given by US [various]
00370   ! Local variables
00371   real :: p_scale   ! A factor to convert pressure to units of Pa [Pa T2 R-1 L-2 ~> 1]
00372   real :: rho_scale ! A factor to convert density from kg m-3 to the desired units [R m3 kg-1 ~> 1]
00373   real :: rho_unscale ! A factor to convert density from R to kg m-3 [kg m-3 R-1 ~> 1]
00374   real :: rho_reference ! rho_ref converted to [kg m-3]
00375   real, dimension(size(rho)) :: pres  ! Pressure converted to [Pa]
00376   integer :: i, is, ie, npts
00377
00378   if (.not.associated(eos)) call mom_error(fatal, &
00379     "calculate_density_1d called with an unassociated EOS_type EOS.")
00380
00381   if (present(dom)) then
00382     is = dom(1) ; ie = dom(2) ; npts = 1 + ie - is
00383   else
00384     is = 1 ; ie = size(rho) ; npts = 1 + ie - is
00385   endif
00386
00387   p_scale = eos%RL2_T2_to_Pa
00388   rho_unscale = eos%R_to_kg_m3
00389
00390   if ((p_scale == 1.0) .and. (rho_unscale == 1.0)) then
00391     call calculate_density_array(t, s, pressure, rho, is, npts, eos, rho_ref=rho_ref)
00392   elseif (present(rho_ref)) then ! This is the same as above, but with some extra work to rescale
      variables.
00393     do i=is,ie ; pres(i) = p_scale * pressure(i) ; enddo
00394     rho_reference = rho_unscale*rho_ref
00395     call calculate_density_array(t, s, pres, rho, is, npts, eos, rho_ref=rho_reference)
00396   else  ! There is rescaling of variables, but rho_ref is not present. Passing a 0 value of rho_ref
00397         ! changes answers at roundoff for some equations of state, like Wright and UNESCO.
00398     do i=is,ie ; pres(i) = p_scale * pressure(i) ; enddo
00399     call calculate_density_array(t, s, pres, rho, is, npts, eos)
00400   endif
00401
00402   rho_scale = eos%kg_m3_to_R
00403   if (present(scale)) rho_scale = rho_scale * scale
00404   if (rho_scale /= 1.0) then ; do i=is,ie
00405     rho(i) = rho_scale * rho(i)
00406   enddo ; endif
00407
```

42.3.2.9 calculate_density_array()

```
subroutine mom_eos::calculate_density_array (
              real, dimension(:), intent(in) T,
              real, dimension(:), intent(in) S,
              real, dimension(:), intent(in) pressure,
              real, dimension(:), intent(inout) rho,
              integer, intent(in) start,
              integer, intent(in) npts,
              type(eos_type), pointer EOS,
              real, intent(in), optional rho_ref,
              real, intent(in), optional scale )  [private]
```

Calls the appropriate subroutine to calculate the density of sea water for 1-D array inputs. If rho_ref is present, the anomaly with respect to rho_ref is returned.

Parameters

in	*t*	Potential temperature referenced to the surface [degC]
in	*s*	Salinity [ppt]
in	*pressure*	Pressure [Pa] or [R L2 T-2 ~> Pa]
in,out	*rho*	Density (in-situ if pressure is local) [kg m-3] or [R ~> kg m-3]
in	*start*	Start index for computation
in	*npts*	Number of point to compute

Parameters

	eos	Equation of state structure
in	*rho_ref*	A reference density [kg m-3]
in	*scale*	A multiplicative factor by which to scale density in combination with scaling given by US [various]

Definition at line 263 of file MOM_EOS.F90.

References eos_linear, eos_nemo, eos_teos10, eos_unesco, and eos_wright.

Referenced by calculate_density_1d().

```
00263   real, dimension(:), intent(in)    :: t        !< Potential temperature referenced to the surface [degC]
00264   real, dimension(:), intent(in)    :: s        !< Salinity [ppt]
00265   real, dimension(:), intent(in)    :: pressure !< Pressure [Pa] or [R L2 T-2 ~> Pa]
00266   real, dimension(:), intent(inout) :: rho      !< Density (in-situ if pressure is local) [kg m-3] or [R ~>
      kg m-3]
00267   integer,            intent(in)    :: start    !< Start index for computation
00268   integer,            intent(in)    :: npts     !< Number of point to compute
00269   type(eos_type),     pointer       :: eos      !< Equation of state structure
00270   real,                  optional, intent(in) :: rho_ref  !< A reference density [kg m-3]
00271   real,                  optional, intent(in) :: scale    !< A multiplicative factor by which to scale
      density
00272                                                 !! in combination with scaling given by US [various]
00273   integer :: j
00274
00275   if (.not.associated(eos)) call mom_error(fatal, &
00276     "calculate_density_array called with an unassociated EOS_type EOS.")
00277
00278   select case (eos%form_of_EOS)
00279     case (eos_linear)
00280       call calculate_density_linear(t, s, pressure, rho, start, npts, &
00281                                     eos%Rho_T0_S0, eos%dRho_dT, eos%dRho_dS, rho_ref)
00282     case (eos_unesco)
00283       call calculate_density_unesco(t, s, pressure, rho, start, npts, rho_ref)
00284     case (eos_wright)
00285       call calculate_density_wright(t, s, pressure, rho, start, npts, rho_ref)
00286     case (eos_teos10)
00287       call calculate_density_teos10(t, s, pressure, rho, start, npts, rho_ref)
00288     case (eos_nemo)
00289     call calculate_density_nemo(t, s, pressure, rho, start, npts, rho_ref)
00290     case default
00291       call mom_error(fatal, "calculate_density_array: EOS%form_of_EOS is not valid.")
00292   end select
00293
00294   if (present(scale)) then ; if (scale /= 1.0) then ; do j=start,start+npts-1
00295     rho(j) = scale * rho(j)
00296   enddo ; endif ; endif
00297
```

42.3.2.10 calculate_density_derivs_1d()

```
subroutine mom_eos::calculate_density_derivs_1d (
             real, dimension(:), intent(in) T,
             real, dimension(:), intent(in) S,
             real, dimension(:), intent(in) pressure,
             real, dimension(:), intent(inout) drho_dT,
             real, dimension(:), intent(inout) drho_dS,
             type(eos_type), pointer EOS,
             integer, dimension(2), intent(in), optional dom,
             real, intent(in), optional scale )  [private]
```

Calls the appropriate subroutine to calculate density derivatives for 1-D array inputs.

Parameters

in	*t*	Potential temperature referenced to the surface [degC]
in	*s*	Salinity [ppt]
in	*pressure*	Pressure [R L2 T-2 ~> Pa]
in,out	*drho_dt*	The partial derivative of density with potential temperature [R degC-1 ~> kg m-3 degC-1]
in,out	*drho_ds*	The partial derivative of density with salinity [R degC-1 ~> kg m-3 ppt-1]
	eos	Equation of state structure
in	*dom*	The domain of indices to work on, taking into account that arrays start at 1.
in	*scale*	A multiplicative factor by which to scale density in combination with scaling given by US [various]

Definition at line 751 of file MOM_EOS.F90.

References calculate_density_derivs_array().

```
00751   real, dimension(:),    intent(in)    :: t        !< Potential temperature referenced to the surface
      [degC]
00752   real, dimension(:),    intent(in)    :: s        !< Salinity [ppt]
00753   real, dimension(:),    intent(in)    :: pressure !< Pressure [R L2 T-2 ~> Pa]
00754   real, dimension(:),    intent(inout) :: drho_dt  !< The partial derivative of density with potential
00755                                                      !! temperature [R degC-1 ~> kg m-3 degC-1]
00756   real, dimension(:),    intent(inout) :: drho_ds  !< The partial derivative of density with salinity
00757                                                      !! [R degC-1 ~> kg m-3 ppt-1]
00758   type(eos_type),        pointer       :: eos      !< Equation of state structure
00759   integer, dimension(2), optional, intent(in) :: dom   !< The domain of indices to work on, taking
00760                                                        !! into account that arrays start at 1.
00761   real,                  optional, intent(in) :: scale !< A multiplicative factor by which to scale density
00762                                                        !! in combination with scaling given by US [various]
00763   ! Local variables
00764   real, dimension(size(drho_dT)) :: pres  ! Pressure converted to [Pa]
00765   real :: rho_scale ! A factor to convert density from kg m-3 to the desired units [R m3 kg-1 ~> 1]
00766   real :: p_scale   ! A factor to convert pressure to units of Pa [Pa T2 R-1 L-2 ~> 1]
00767   integer :: i, is, ie, npts
00768
00769   if (.not.associated(eos)) call mom_error(fatal, &
00770     "calculate_density_derivs called with an unassociated EOS_type EOS.")
00771
00772   if (present(dom)) then
00773     is = dom(1) ; ie = dom(2) ; npts = 1 + ie - is
00774   else
00775     is = 1 ; ie = size(drho_dt) ; npts = 1 + ie - is
00776   endif
00777
00778   p_scale = eos%RL2_T2_to_Pa
00779
00780   if (p_scale == 1.0) then
00781     call calculate_density_derivs_array(t, s, pressure, drho_dt, drho_ds, is, npts, eos)
00782   else
00783     do i=is,ie ; pres(i) = p_scale * pressure(i) ; enddo
00784     call calculate_density_derivs_array(t, s, pres, drho_dt, drho_ds, is, npts, eos)
00785   endif
00786
00787   rho_scale = eos%kg_m3_to_R
00788   if (present(scale)) rho_scale = rho_scale * scale
00789   if (rho_scale /= 1.0) then ; do i=is,ie
00790     drho_dt(i) = rho_scale * drho_dt(i)
00791     drho_ds(i) = rho_scale * drho_ds(i)
00792   enddo ; endif
00793
```

42.3.2.11 calculate_density_derivs_array()

```
subroutine mom_eos::calculate_density_derivs_array (
             real, dimension(:), intent(in) T,
```

```
real, dimension(:), intent(in) S,
real, dimension(:), intent(in) pressure,
real, dimension(:), intent(inout) drho_dT,
real, dimension(:), intent(inout) drho_dS,
integer, intent(in) start,
integer, intent(in) npts,
type(eos_type), pointer EOS,
real, intent(in), optional scale ) [private]
```

Calls the appropriate subroutine to calculate density derivatives for 1-D array inputs.

Parameters

in	*t*	Potential temperature referenced to the surface [degC]
in	*s*	Salinity [ppt]
in	*pressure*	Pressure [Pa] or [R L2 T-2 ~> Pa]
in,out	*drho_dt*	The partial derivative of density with potential temperature [kg m-3 degC-1] or [R degC-1 ~> kg m-3 degC-1]
in,out	*drho_ds*	The partial derivative of density with salinity, in [kg m-3 ppt-1] or [R degC-1 ~> kg m-3 ppt-1]
in	*start*	Starting index within the array
in	*npts*	The number of values to calculate
	eos	Equation of state structure
in	*scale*	A multiplicative factor by which to scale density in combination with scaling given by US [various]

Definition at line 706 of file MOM_EOS.F90.

References eos_linear, eos_nemo, eos_teos10, eos_unesco, and eos_wright.

Referenced by calculate_density_derivs_1d().

```
00706   real, dimension(:), intent(in)    :: t        !< Potential temperature referenced to the surface [degC]
00707   real, dimension(:), intent(in)    :: s        !< Salinity [ppt]
00708   real, dimension(:), intent(in)    :: pressure !< Pressure [Pa] or [R L2 T-2 ~> Pa]
00709   real, dimension(:), intent(inout) :: drho_dt  !< The partial derivative of density with potential
00710                                                 !! temperature [kg m-3 degC-1] or [R degC-1 ~> kg m-3
      degC-1]
00711   real, dimension(:), intent(inout) :: drho_ds  !< The partial derivative of density with salinity,
00712                                                 !! in [kg m-3 ppt-1] or [R degC-1 ~> kg m-3 ppt-1]
00713   integer,            intent(in)    :: start    !< Starting index within the array
00714   integer,            intent(in)    :: npts     !< The number of values to calculate
00715   type(eos_type),     pointer       :: eos      !< Equation of state structure
00716   real,     optional, intent(in)    :: scale !< A multiplicative factor by which to scale density
00717                                                 !! in combination with scaling given by US [various]
00718
00719   ! Local variables
00720   integer :: j
00721
00722   if (.not.associated(eos)) call mom_error(fatal, &
00723     "calculate_density_derivs called with an unassociated EOS_type EOS.")
00724
00725   select case (eos%form_of_EOS)
00726     case (eos_linear)
00727       call calculate_density_derivs_linear(t, s, pressure, drho_dt, drho_ds, eos%Rho_T0_S0, &
00728                                            eos%dRho_dT, eos%dRho_dS, start, npts)
00729     case (eos_unesco)
00730       call calculate_density_derivs_unesco(t, s, pressure, drho_dt, drho_ds, start, npts)
00731     case (eos_wright)
00732       call calculate_density_derivs_wright(t, s, pressure, drho_dt, drho_ds, start, npts)
00733     case (eos_teos10)
00734       call calculate_density_derivs_teos10(t, s, pressure, drho_dt, drho_ds, start, npts)
00735     case (eos_nemo)
00736       call calculate_density_derivs_nemo(t, s, pressure, drho_dt, drho_ds, start, npts)
00737     case default
00738       call mom_error(fatal, "calculate_density_derivs_array: EOS%form_of_EOS is not valid.")
```

```
00739   end select
00740
00741   if (present(scale)) then ; if (scale /= 1.0) then ; do j=start,start+npts-1
00742     drho_dt(j) = scale * drho_dt(j)
00743     drho_ds(j) = scale * drho_ds(j)
00744   enddo ; endif ; endif
00745
```

42.3.2.12 calculate_density_derivs_scalar()

```
subroutine mom_eos::calculate_density_derivs_scalar (
             real, intent(in) T,
             real, intent(in) S,
             real, intent(in) pressure,
             real, intent(out) drho_dT,
             real, intent(out) drho_dS,
             type(eos_type), pointer EOS,
             real, intent(in), optional scale )  [private]
```

Calls the appropriate subroutines to calculate density derivatives by promoting a scalar to a one-element array.

Parameters

in	*t*	Potential temperature referenced to the surface [degC]
in	*s*	Salinity [ppt]
in	*pressure*	Pressure [Pa] or [R L2 T-2 ~> Pa]
out	*drho_dt*	The partial derivative of density with potential temperature [kg m-3 degC-1] or [R degC-1 ~> kg m-3 degC-1]
out	*drho_ds*	The partial derivative of density with salinity, in [kg m-3 ppt-1] or [R ppt-1 ~> kg m-3 ppt-1]
	eos	Equation of state structure
in	*scale*	A multiplicative factor by which to scale density in combination with scaling given by US [various]

Definition at line 800 of file MOM_EOS.F90.

References eos_linear, eos_teos10, and eos_wright.

```
00800   real,           intent(in)  :: t !< Potential temperature referenced to the surface [degC]
00801   real,           intent(in)  :: s !< Salinity [ppt]
00802   real,           intent(in)  :: pressure !< Pressure [Pa] or [R L2 T-2 ~> Pa]
00803   real,           intent(out) :: drho_dt !< The partial derivative of density with potential
00804                                  !! temperature [kg m-3 degC-1] or [R degC-1 ~> kg m-3 degC-1]
00805   real,           intent(out) :: drho_ds !< The partial derivative of density with salinity,
00806                                  !! in [kg m-3 ppt-1] or [R ppt-1 ~> kg m-3 ppt-1]
00807   type(eos_type), pointer     :: eos     !< Equation of state structure
00808   real, optional, intent(in)  :: scale   !< A multiplicative factor by which to scale density
00809                                  !! in combination with scaling given by US [various]
00810   ! Local variables
00811   real :: rho_scale ! A factor to convert density from kg m-3 to the desired units [R m3 kg-1 ~> 1]
00812   real :: p_scale   ! A factor to convert pressure to units of Pa [Pa T2 R-1 L-2 ~> 1]
00813   integer :: j
00814
00815   if (.not.associated(eos)) call mom_error(fatal, &
00816     "calculate_density_derivs called with an unassociated EOS_type EOS.")
00817
00818   p_scale = eos%RL2_T2_to_Pa
00819
00820   select case (eos%form_of_EOS)
00821     case (eos_linear)
00822       call calculate_density_derivs_linear(t, s, p_scale*pressure, drho_dt, drho_ds, &
```

```
00823                                             eos%Rho_T0_S0, eos%dRho_dT, eos%dRho_dS)
00824     case (eos_wright)
00825       call calculate_density_derivs_wright(t, s, p_scale*pressure, drho_dt, drho_ds)
00826     case (eos_teos10)
00827       call calculate_density_derivs_teos10(t, s, p_scale*pressure, drho_dt, drho_ds)
00828     case default
00829       call mom_error(fatal, "calculate_density_derivs_scalar: EOS%form_of_EOS is not valid.")
00830   end select
00831
00832   rho_scale = eos%kg_m3_to_R
00833   if (present(scale)) rho_scale = rho_scale * scale
00834   if (rho_scale /= 1.0) then
00835     drho_dt = rho_scale * drho_dt
00836     drho_ds = rho_scale * drho_ds
00837   endif
00838
```

42.3.2.13 calculate_density_scalar()

```
subroutine mom_eos::calculate_density_scalar (
             real, intent(in) T,
             real, intent(in) S,
             real, intent(in) pressure,
             real, intent(out) rho,
             type(eos_type), pointer EOS,
             real, intent(in), optional rho_ref,
             real, intent(in), optional scale )  [private]
```

Calls the appropriate subroutine to calculate density of sea water for scalar inputs. If rho_ref is present, the anomaly with respect to rho_ref is returned. The pressure and density can be rescaled with the US. If both the US and scale arguments are present the density scaling uses the product of the two scaling factors.

Parameters

in	*t*	Potential temperature referenced to the surface [degC]
in	*s*	Salinity [ppt]
in	*pressure*	Pressure [Pa] or [R L2 T-2 ~> Pa]
out	*rho*	Density (in-situ if pressure is local) [kg m-3] or [R ~> kg m-3]
	eos	Equation of state structure
in	*rho_ref*	A reference density [kg m-3]
in	*scale*	A multiplicative factor by which to scale density in combination with scaling given by US [various]

Definition at line 166 of file MOM_EOS.F90.

References eos_linear, eos_nemo, eos_teos10, eos_unesco, and eos_wright.

```
00166   real,           intent(in)  :: t        !< Potential temperature referenced to the surface [degC]
00167   real,           intent(in)  :: s        !< Salinity [ppt]
00168   real,           intent(in)  :: pressure !< Pressure [Pa] or [R L2 T-2 ~> Pa]
00169   real,           intent(out) :: rho      !< Density (in-situ if pressure is local) [kg m-3] or [R ~> kg
      m-3]
00170   type(eos_type), pointer     :: eos      !< Equation of state structure
00171   real, optional, intent(in)  :: rho_ref  !< A reference density [kg m-3]
00172   real, optional, intent(in)  :: scale    !< A multiplicative factor by which to scale density in
00173                                           !! combination with scaling given by US [various]
00174
00175   real :: rho_scale ! A factor to convert density from kg m-3 to the desired units [R m3 kg-1 ~> 1]
00176   real :: p_scale   ! A factor to convert pressure to units of Pa [Pa T2 R-1 L-2 ~> 1]
```

```
00177
00178   if (.not.associated(eos)) call mom_error(fatal, &
00179     "calculate_density_scalar called with an unassociated EOS_type EOS.")
00180
00181   p_scale = eos%RL2_T2_to_Pa
00182
00183   select case (eos%form_of_EOS)
00184     case (eos_linear)
00185       call calculate_density_linear(t, s, p_scale*pressure, rho, &
00186                                     eos%Rho_T0_S0, eos%dRho_dT, eos%dRho_dS, rho_ref)
00187     case (eos_unesco)
00188       call calculate_density_unesco(t, s, p_scale*pressure, rho, rho_ref)
00189     case (eos_wright)
00190       call calculate_density_wright(t, s, p_scale*pressure, rho, rho_ref)
00191     case (eos_teos10)
00192       call calculate_density_teos10(t, s, p_scale*pressure, rho, rho_ref)
00193     case (eos_nemo)
00194       call calculate_density_nemo(t, s, p_scale*pressure, rho, rho_ref)
00195     case default
00196       call mom_error(fatal, "calculate_density_scalar: EOS is not valid.")
00197   end select
00198
00199   rho_scale = eos%kg_m3_to_R
00200   if (present(scale)) rho_scale = rho_scale * scale
00201   rho = rho_scale * rho
00202
```

42.3.2.14 calculate_density_second_derivs_array()

```
subroutine mom_eos::calculate_density_second_derivs_array (
              real, dimension(:), intent(in) T,
              real, dimension(:), intent(in) S,
              real, dimension(:), intent(in) pressure,
              real, dimension(:), intent(inout) drho_dS_dS,
              real, dimension(:), intent(inout) drho_dS_dT,
              real, dimension(:), intent(inout) drho_dT_dT,
              real, dimension(:), intent(inout) drho_dS_dP,
              real, dimension(:), intent(inout) drho_dT_dP,
              integer, intent(in) start,
              integer, intent(in) npts,
              type(eos_type), pointer EOS,
              real, intent(in), optional scale )  [private]
```

Calls the appropriate subroutine to calculate density second derivatives for 1-D array inputs.

Parameters

in	*t*	Potential temperature referenced to the surface [degC]
in	*s*	Salinity [ppt]
in	*pressure*	Pressure [Pa] or [R L2 T-2 ~> Pa]
in,out	*drho_ds_ds*	Partial derivative of beta with respect to S [kg m-3 ppt-2] or [R ppt-2 ~> kg m-3 ppt-2]
in,out	*drho_ds_dt*	Partial derivative of beta with respect to T [kg m-3 ppt-1 degC-1] or [R ppt-1 degC-1 ~> kg m-3 ppt-1 degC-1]
in,out	*drho_dt_dt*	Partial derivative of alpha with respect to T [kg m-3 degC-2] or [R degC-2 ~> kg m-3 degC-2]
in,out	*drho_ds_dp*	Partial derivative of beta with respect to pressure [kg m-3 ppt-1 Pa-1] or [R ppt-1 Pa-1 ~> kg m-3 ppt-1 Pa-1]
in,out	*drho_dt_dp*	Partial derivative of alpha with respect to pressure [kg m-3 degC-1 Pa-1] or [R degC-1 Pa-1 ~> kg m-3 degC-1 Pa-1]
in	*start*	Starting index within the array

Parameters

in	*npts*	The number of values to calculate
	eos	Equation of state structure
in	*scale*	A multiplicative factor by which to scale density in combination with scaling given by US [various]

Definition at line 844 of file MOM_EOS.F90.

References eos_linear, eos_teos10, and eos_wright.

```
00844   real, dimension(:), intent(in)  :: t !< Potential temperature referenced to the surface [degC]
00845   real, dimension(:), intent(in)  :: s !< Salinity [ppt]
00846   real, dimension(:), intent(in)  :: pressure   !< Pressure [Pa] or [R L2 T-2 ~> Pa]
00847   real, dimension(:), intent(inout) :: drho_ds_ds !< Partial derivative of beta with respect to S
00848                                                   !!  [kg m-3 ppt-2] or [R ppt-2 ~> kg m-3 ppt-2]
00849   real, dimension(:), intent(inout) :: drho_ds_dt !< Partial derivative of beta with respect to T
00850                                                   !! [kg m-3 ppt-1 degC-1] or [R ppt-1 degC-1 ~> kg m-3
      ppt-1 degC-1]
00851   real, dimension(:), intent(inout) :: drho_dt_dt !< Partial derivative of alpha with respect to T
00852                                                   !! [kg m-3 degC-2] or [R degC-2 ~> kg m-3 degC-2]
00853   real, dimension(:), intent(inout) :: drho_ds_dp !< Partial derivative of beta with respect to pressure
00854                                                   !! [kg m-3 ppt-1 Pa-1] or [R ppt-1 Pa-1 ~> kg m-3 ppt-1
      Pa-1]
00855   real, dimension(:), intent(inout) :: drho_dt_dp !< Partial derivative of alpha with respect to pressure
00856                                                   !! [kg m-3 degC-1 Pa-1] or [R degC-1 Pa-1 ~> kg m-3
      degC-1 Pa-1]
00857   integer,            intent(in)  :: start !< Starting index within the array
00858   integer,            intent(in)  :: npts  !< The number of values to calculate
00859   type(eos_type),     pointer     :: eos   !< Equation of state structure
00860   real,                  optional, intent(in) :: scale !< A multiplicative factor by which to scale density
00861                                                   !! in combination with scaling given by US [various]
00862   ! Local variables
00863   real, dimension(size(pressure)) :: pres  ! Pressure converted to [Pa]
00864   real :: rho_scale ! A factor to convert density from kg m-3 to the desired units [R m3 kg-1 ~> 1]
00865   real :: p_scale   ! A factor to convert pressure to units of Pa [Pa T2 R-1 L-2 ~> 1]
00866   real :: i_p_scale ! The inverse of the factor to convert pressure to units of Pa [R L2 T-2 Pa-1 ~> 1]
00867   integer :: j
00868
00869   if (.not.associated(eos)) call mom_error(fatal, &
00870     "calculate_density_derivs called with an unassociated EOS_type EOS.")
00871
00872   p_scale = eos%RL2_T2_to_Pa
00873
00874   if (p_scale == 1.0) then
00875     select case (eos%form_of_EOS)
00876       case (eos_linear)
00877         call calculate_density_second_derivs_linear(t, s, pressure, drho_ds_ds, drho_ds_dt, &
00878                                                     drho_dt_dt, drho_ds_dp, drho_dt_dp, start, npts)
00879       case (eos_wright)
00880         call calculate_density_second_derivs_wright(t, s, pressure, drho_ds_ds, drho_ds_dt, &
00881                                                     drho_dt_dt, drho_ds_dp, drho_dt_dp, start, npts)
00882       case (eos_teos10)
00883         call calculate_density_second_derivs_teos10(t, s, pressure, drho_ds_ds, drho_ds_dt, &
00884                                                     drho_dt_dt, drho_ds_dp, drho_dt_dp, start, npts)
00885       case default
00886         call mom_error(fatal, "calculate_density_derivs: EOS%form_of_EOS is not valid.")
00887     end select
00888   else
00889     do j=start,start+npts-1 ; pres(j) = p_scale * pressure(j) ; enddo
00890     select case (eos%form_of_EOS)
00891       case (eos_linear)
00892         call calculate_density_second_derivs_linear(t, s, pres, drho_ds_ds, drho_ds_dt, &
00893                                                     drho_dt_dt, drho_ds_dp, drho_dt_dp, start, npts)
00894       case (eos_wright)
00895         call calculate_density_second_derivs_wright(t, s, pres, drho_ds_ds, drho_ds_dt, &
00896                                                     drho_dt_dt, drho_ds_dp, drho_dt_dp, start, npts)
00897       case (eos_teos10)
00898         call calculate_density_second_derivs_teos10(t, s, pres, drho_ds_ds, drho_ds_dt, &
00899                                                     drho_dt_dt, drho_ds_dp, drho_dt_dp, start, npts)
00900       case default
00901         call mom_error(fatal, "calculate_density_derivs: EOS%form_of_EOS is not valid.")
00902     end select
00903   endif
00904
00905   rho_scale = eos%kg_m3_to_R
00906   if (present(scale)) rho_scale = rho_scale * scale
00907   if (rho_scale /= 1.0) then ; do j=start,start+npts-1
00908     drho_ds_ds(j) = rho_scale * drho_ds_ds(j)
```

```
00909     drho_ds_dt(j) = rho_scale * drho_ds_dt(j)
00910     drho_dt_dt(j) = rho_scale * drho_dt_dt(j)
00911     drho_ds_dp(j) = rho_scale * drho_ds_dp(j)
00912     drho_dt_dp(j) = rho_scale * drho_dt_dp(j)
00913   enddo ; endif
00914
00915   if (p_scale /= 1.0) then
00916     i_p_scale = 1.0 / p_scale
00917     do j=start,start+npts-1
00918       drho_ds_dp(j) = i_p_scale * drho_ds_dp(j)
00919       drho_dt_dp(j) = i_p_scale * drho_dt_dp(j)
00920     enddo
00921   endif
00922
```

42.3.2.15 calculate_density_second_derivs_scalar()

```
subroutine mom_eos::calculate_density_second_derivs_scalar (
             real, intent(in) T,
             real, intent(in) S,
             real, intent(in) pressure,
             real, intent(out) drho_dS_dS,
             real, intent(out) drho_dS_dT,
             real, intent(out) drho_dT_dT,
             real, intent(out) drho_dS_dP,
             real, intent(out) drho_dT_dP,
             type(eos_type), pointer EOS,
             real, intent(in), optional scale )  [private]
```

Calls the appropriate subroutine to calculate density second derivatives for scalar nputs.

Parameters

in	*t*	Potential temperature referenced to the surface [degC]
in	*s*	Salinity [ppt]
in	*pressure*	Pressure [Pa] or [R L2 T-2 ~> Pa]
out	*drho_ds_ds*	Partial derivative of beta with respect to S [kg m-3 ppt-2] or [R ppt-2 ~> kg m-3 ppt-2]
out	*drho_ds_dt*	Partial derivative of beta with respect to T [kg m-3 ppt-1 degC-1] or [R ppt-1 degC-1 ~> kg m-3 ppt-1 degC-1]
out	*drho_dt_dt*	Partial derivative of alpha with respect to T [kg m-3 degC-2] or [R degC-2 ~> kg m-3 degC-2]
out	*drho_ds_dp*	Partial derivative of beta with respect to pressure [kg m-3 ppt-1 Pa-1] or [R ppt-1 Pa-1 ~> kg m-3 ppt-1 Pa-1]
out	*drho_dt_dp*	Partial derivative of alpha with respect to pressure [kg m-3 degC-1 Pa-1] or [R degC-1 Pa-1 ~> kg m-3 degC-1 Pa-1]
	eos	Equation of state structure
in	*scale*	A multiplicative factor by which to scale density in combination with scaling given by US [various]

Definition at line 928 of file MOM_EOS.F90.

References eos_linear, eos_teos10, and eos_wright.

```
00928   real, intent(in)  :: t !< Potential temperature referenced to the surface [degC]
```

```
00929   real, intent(in)  :: s !< Salinity [ppt]
00930   real, intent(in)  :: pressure   !< Pressure [Pa] or [R L2 T-2 ~> Pa]
00931   real, intent(out) :: drho_ds_ds !< Partial derivative of beta with respect to S
00932                                   !! [kg m-3 ppt-2] or [R ppt-2 ~> kg m-3 ppt-2]
00933   real, intent(out) :: drho_ds_dt !< Partial derivative of beta with respect to T
00934                                   !! [kg m-3 ppt-1 degC-1] or [R ppt-1 degC-1 ~> kg m-3 ppt-1 degC-1]
00935   real, intent(out) :: drho_dt_dt !< Partial derivative of alpha with respect to T
00936                                   !! [kg m-3 degC-2] or [R degC-2 ~> kg m-3 degC-2]
00937   real, intent(out) :: drho_ds_dp !< Partial derivative of beta with respect to pressure
00938                                   !! [kg m-3 ppt-1 Pa-1] or [R ppt-1 Pa-1 ~> kg m-3 ppt-1 Pa-1]
00939   real, intent(out) :: drho_dt_dp !< Partial derivative of alpha with respect to pressure
00940                                   !! [kg m-3 degC-1 Pa-1] or [R degC-1 Pa-1 ~> kg m-3 degC-1 Pa-1]
00941   type(eos_type), pointer    :: eos !< Equation of state structure
00942   real, optional, intent(in) :: scale !< A multiplicative factor by which to scale density
00943                                   !! in combination with scaling given by US [various]
00944   ! Local variables
00945   real :: rho_scale ! A factor to convert density from kg m-3 to the desired units [R m3 kg-1 ~> 1]
00946   real :: p_scale   ! A factor to convert pressure to units of Pa [Pa T2 R-1 L-2 ~> 1]
00947   real :: i_p_scale ! The inverse of the factor to convert pressure to units of Pa [R L2 T-2 Pa-1 ~> 1]
00948
00949   if (.not.associated(eos)) call mom_error(fatal, &
00950     "calculate_density_derivs called with an unassociated EOS_type EOS.")
00951
00952   p_scale = eos%RL2_T2_to_Pa
00953
00954   select case (eos%form_of_EOS)
00955     case (eos_linear)
00956       call calculate_density_second_derivs_linear(t, s, p_scale*pressure, drho_ds_ds, drho_ds_dt, &
00957                                                   drho_dt_dt, drho_ds_dp, drho_dt_dp)
00958     case (eos_wright)
00959       call calculate_density_second_derivs_wright(t, s, p_scale*pressure, drho_ds_ds, drho_ds_dt, &
00960                                                   drho_dt_dt, drho_ds_dp, drho_dt_dp)
00961     case (eos_teos10)
00962       call calculate_density_second_derivs_teos10(t, s, p_scale*pressure, drho_ds_ds, drho_ds_dt, &
00963                                                   drho_dt_dt, drho_ds_dp, drho_dt_dp)
00964     case default
00965       call mom_error(fatal, "calculate_density_derivs: EOS%form_of_EOS is not valid.")
00966   end select
00967
00968   rho_scale = eos%kg_m3_to_R
00969   if (present(scale)) rho_scale = rho_scale * scale
00970   if (rho_scale /= 1.0) then
00971     drho_ds_ds = rho_scale * drho_ds_ds
00972     drho_ds_dt = rho_scale * drho_ds_dt
00973     drho_dt_dt = rho_scale * drho_dt_dt
00974     drho_ds_dp = rho_scale * drho_ds_dp
00975     drho_dt_dp = rho_scale * drho_dt_dp
00976   endif
00977
00978   if (p_scale /= 1.0) then
00979     i_p_scale = 1.0 / p_scale
00980     drho_ds_dp = i_p_scale * drho_ds_dp
00981     drho_dt_dp = i_p_scale * drho_dt_dp
00982   endif
00983
```

42.3.2.16 calculate_spec_vol_array()

```
subroutine mom_eos::calculate_spec_vol_array (
             real, dimension(:), intent(in) T,
             real, dimension(:), intent(in) S,
             real, dimension(:), intent(in) pressure,
             real, dimension(:), intent(inout) specvol,
             integer, intent(in) start,
             integer, intent(in) npts,
             type(eos_type), pointer EOS,
             real, intent(in), optional spv_ref,
             real, intent(in), optional scale )  [private]
```

Calls the appropriate subroutine to calculate the specific volume of sea water for 1-D array inputs.

Parameters

in	*t*	potential temperature relative to the surface [degC]
in	*s*	salinity [ppt]
in	*pressure*	pressure [Pa]
in,out	*specvol*	in situ specific volume [kg m-3]
in	*start*	the starting point in the arrays.
in	*npts*	the number of values to calculate.
	eos	Equation of state structure
in	*spv_ref*	A reference specific volume [m3 kg-1]
in	*scale*	A multiplicative factor by which to scale specific volume in combination with scaling given by US [various]

Definition at line 486 of file MOM_EOS.F90.

References eos_linear, eos_nemo, eos_teos10, eos_unesco, and eos_wright.

Referenced by calc_spec_vol_1d(), and calc_spec_vol_scalar().

```
00486   real, dimension(:), intent(in)    :: t        !< potential temperature relative to the surface [degC]
00487   real, dimension(:), intent(in)    :: s        !< salinity [ppt]
00488   real, dimension(:), intent(in)    :: pressure !< pressure [Pa]
00489   real, dimension(:), intent(inout) :: specvol  !< in situ specific volume [kg m-3]
00490   integer,            intent(in)    :: start    !< the starting point in the arrays.
00491   integer,            intent(in)    :: npts     !< the number of values to calculate.
00492   type(eos_type),     pointer       :: eos      !< Equation of state structure
00493   real,     optional, intent(in)    :: spv_ref  !< A reference specific volume [m3 kg-1]
00494   real,     optional, intent(in)    :: scale    !< A multiplicative factor by which to scale specific
00495                                                 !! volume in combination with scaling given by US [various]
00496
00497   real, dimension(size(specvol))  :: rho   ! Density [kg m-3]
00498   integer :: j
00499
00500   if (.not.associated(eos)) call mom_error(fatal, &
00501     "calculate_spec_vol_array called with an unassociated EOS_type EOS.")
00502
00503   select case (eos%form_of_EOS)
00504     case (eos_linear)
00505       call calculate_spec_vol_linear(t, s, pressure, specvol, start, npts, &
00506                eos%rho_T0_S0, eos%drho_dT, eos%drho_dS, spv_ref)
00507     case (eos_unesco)
00508       call calculate_spec_vol_unesco(t, s, pressure, specvol, start, npts, spv_ref)
00509     case (eos_wright)
00510       call calculate_spec_vol_wright(t, s, pressure, specvol, start, npts, spv_ref)
00511     case (eos_teos10)
00512       call calculate_spec_vol_teos10(t, s, pressure, specvol, start, npts, spv_ref)
00513     case (eos_nemo)
00514       call calculate_density_nemo(t, s, pressure, rho, start, npts)
00515       if (present(spv_ref)) then
00516         specvol(:) = 1.0 / rho(:) - spv_ref
00517       else
00518         specvol(:) = 1.0 / rho(:)
00519       endif
00520     case default
00521       call mom_error(fatal, "calculate_spec_vol_array: EOS%form_of_EOS is not valid.")
00522   end select
00523
00524   if (present(scale)) then ; if (scale /= 1.0) then ; do j=start,start+npts-1
00525     specvol(j) = scale * specvol(j)
00526   enddo ; endif ; endif
00527
```

42.3.2.17 calculate_spec_vol_derivs_array()

```
subroutine mom_eos::calculate_spec_vol_derivs_array (
             real, dimension(:), intent(in) T,
             real, dimension(:), intent(in) S,
             real, dimension(:), intent(in) pressure,
             real, dimension(:), intent(inout) dSV_dT,
             real, dimension(:), intent(inout) dSV_dS,
             integer, intent(in) start,
             integer, intent(in) npts,
             type(eos_type), pointer EOS )  [private]
```

Calls the appropriate subroutine to calculate specific volume derivatives for an array.

Parameters

in	*t*	Potential temperature referenced to the surface [degC]
in	*s*	Salinity [ppt]
in	*pressure*	Pressure [Pa]
in,out	*dsv_dt*	The partial derivative of specific volume with potential temperature [m3 kg-1 degC-1]
in,out	*dsv_ds*	The partial derivative of specific volume with salinity [m3 kg-1 ppt-1]
in	*start*	Starting index within the array
in	*npts*	The number of values to calculate
	eos	Equation of state structure

Definition at line 988 of file MOM_EOS.F90.

References eos_linear, eos_nemo, eos_teos10, eos_unesco, and eos_wright.

Referenced by calc_spec_vol_derivs_1d().

```
00988   real, dimension(:), intent(in)  :: t !< Potential temperature referenced to the surface [degC]
00989   real, dimension(:), intent(in)  :: s !< Salinity [ppt]
00990   real, dimension(:), intent(in)  :: pressure !< Pressure [Pa]
00991   real, dimension(:), intent(inout) :: dsv_dt !< The partial derivative of specific volume with potential
00992                                        !! temperature [m3 kg-1 degC-1]
00993   real, dimension(:), intent(inout) :: dsv_ds !< The partial derivative of specific volume with salinity
00994                                        !! [m3 kg-1 ppt-1]
00995   integer,            intent(in)  :: start  !< Starting index within the array
00996   integer,            intent(in)  :: npts   !< The number of values to calculate
00997   type(eos_type),     pointer     :: eos    !< Equation of state structure
00998
00999   ! Local variables
01000   real, dimension(size(T)) :: press   ! Pressure converted to [Pa]
01001   real, dimension(size(T)) :: rho     ! In situ density [kg m-3]
01002   real, dimension(size(T)) :: drho_dt ! Derivative of density with temperature [kg m-3 degC-1]
01003   real, dimension(size(T)) :: drho_ds ! Derivative of density with salinity [kg m-3 ppt-1]
01004   integer :: j
01005
01006   if (.not.associated(eos)) call mom_error(fatal, &
01007     "calculate_spec_vol_derivs_array called with an unassociated EOS_type EOS.")
01008
01009   select case (eos%form_of_EOS)
01010     case (eos_linear)
01011       call calculate_specvol_derivs_linear(t, s, pressure, dsv_dt, dsv_ds, start, &
01012                                            npts, eos%Rho_T0_S0, eos%dRho_dT, eos%dRho_dS)
01013     case (eos_unesco)
01014       call calculate_density_unesco(t, s, pressure, rho, start, npts)
01015       call calculate_density_derivs_unesco(t, s, pressure, drho_dt, drho_ds, start, npts)
01016       do j=start,start+npts-1
01017         dsv_dt(j) = -drho_dt(j)/(rho(j)**2)
01018         dsv_ds(j) = -drho_ds(j)/(rho(j)**2)
01019       enddo
01020     case (eos_wright)
01021       call calculate_specvol_derivs_wright(t, s, pressure, dsv_dt, dsv_ds, start, npts)
01022     case (eos_teos10)
```

```
01023       call calculate_specvol_derivs_teos10(t, s, pressure, dsv_dt, dsv_ds, start, npts)
01024     case (eos_nemo)
01025       call calculate_density_nemo(t, s, pressure, rho, start, npts)
01026       call calculate_density_derivs_nemo(t, s, pressure, drho_dt, drho_ds, start, npts)
01027       do j=start,start+npts-1
01028         dsv_dt(j) = -drho_dt(j)/(rho(j)**2)
01029         dsv_ds(j) = -drho_ds(j)/(rho(j)**2)
01030       enddo
01031     case default
01032       call mom_error(fatal, "calculate_spec_vol_derivs_array: EOS%form_of_EOS is not valid.")
01033   end select
01034
```

42.3.2.18 calculate_stanley_density_1d()

```
subroutine mom_eos::calculate_stanley_density_1d (
             real, dimension(:), intent(in) T,
             real, dimension(:), intent(in) S,
             real, dimension(:), intent(in) pressure,
             real, dimension(:), intent(in) Tvar,
             real, dimension(:), intent(in) TScov,
             real, dimension(:), intent(in) Svar,
             real, dimension(:), intent(inout) rho,
             type(eos_type), pointer EOS,
             integer, dimension(2), intent(in), optional dom,
             real, intent(in), optional rho_ref,
             real, intent(in), optional scale )  [private]
```

Calls the appropriate subroutine to calculate the density of sea water for 1-D array inputs including the variance of T, S and covariance of T-S, potentially limiting the domain of indices that are worked on. The calculation uses only the second order correction in a series as discussed in Stanley et al., 2020. If rho_ref is present, the anomaly with respect to rho_ref is returned.

Parameters

in	*t*	Potential temperature referenced to the surface [degC]
in	*s*	Salinity [ppt]
in	*pressure*	Pressure [R L2 T-2 ~> Pa]
in	*tvar*	Variance of potential temperature [degC2]
in	*tscov*	Covariance of potential temperature and salinity [degC ppt]
in	*svar*	Variance of salinity [ppt2]
in,out	*rho*	Density (in-situ if pressure is local) [R ~> kg m-3]
	eos	Equation of state structure
in	*dom*	The domain of indices to work on, taking into account that arrays start at 1.
in	*rho_ref*	A reference density [kg m-3]
in	*scale*	A multiplicative factor by which to scale density in combination with scaling given by US [various]

Definition at line 417 of file MOM_EOS.F90.

References eos_linear, eos_teos10, and eos_wright.

```
00417   real, dimension(:),    intent(in)    :: t        !< Potential temperature referenced to the surface
```

```
      [degC]
00418   real, dimension(:),    intent(in)    :: s        !< Salinity [ppt]
00419   real, dimension(:),    intent(in)    :: pressure !< Pressure [R L2 T-2 ~> Pa]
00420   real, dimension(:),    intent(in)    :: tvar     !< Variance of potential temperature [degC2]
00421   real, dimension(:),    intent(in)    :: tscov    !< Covariance of potential temperature and salinity
      [degC ppt]
00422   real, dimension(:),    intent(in)    :: svar     !< Variance of salinity [ppt2]
00423   real, dimension(:),    intent(inout) :: rho      !< Density (in-situ if pressure is local) [R ~> kg m-3]
00424   type(eos_type),        pointer       :: eos      !< Equation of state structure
00425   integer, dimension(2), optional, intent(in) :: dom   !< The domain of indices to work on, taking
00426                                                        !! into account that arrays start at 1.
00427   real,                  optional, intent(in) :: rho_ref !< A reference density [kg m-3]
00428   real,                  optional, intent(in) :: scale !< A multiplicative factor by which to scale density
00429                                                        !! in combination with scaling given by US [various]
00430   ! Local variables
00431   real :: p_scale   ! A factor to convert pressure to units of Pa [Pa T2 R-1 L-2 ~> 1]
00432   real :: rho_scale ! A factor to convert density from kg m-3 to the desired units [R m3 kg-1 ~> 1]
00433   real, dimension(size(rho)) :: pres  ! Pressure converted to [Pa]
00434   real, dimension(size(T)) :: d2rdtt, d2rdst, d2rdss, d2rdsp, d2rdtp ! Second derivatives of density wrt
      T,S,p
00435   integer :: i, is, ie, npts
00436
00437   if (.not.associated(eos)) call mom_error(fatal, &
00438     "calculate_density_1d called with an unassociated EOS_type EOS.")
00439
00440   if (present(dom)) then
00441     is = dom(1) ; ie = dom(2) ; npts = 1 + ie - is
00442   else
00443     is = 1 ; ie = size(rho) ; npts = 1 + ie - is
00444   endif
00445
00446   p_scale = eos%RL2_T2_to_Pa
00447   do i=is,ie
00448     pres(i) = p_scale * pressure(i)
00449   enddo
00450
00451   select case (eos%form_of_EOS)
00452     case (eos_linear)
00453       call calculate_density_linear(t, s, pres, rho, 1, npts, &
00454                                     eos%Rho_T0_S0, eos%dRho_dT, eos%dRho_dS, rho_ref)
00455       call calculate_density_second_derivs_linear(t, s, pres, d2rdss, d2rdst, &
00456                                                   d2rdtt, d2rdsp, d2rdtp, 1, npts)
00457     case (eos_wright)
00458       call calculate_density_wright(t, s, pres, rho, 1, npts, rho_ref)
00459       call calculate_density_second_derivs_wright(t, s, pres, d2rdss, d2rdst, &
00460                                                   d2rdtt, d2rdsp, d2rdtp, 1, npts)
00461     case (eos_teos10)
00462       call calculate_density_teos10(t, s, pres, rho, 1, npts, rho_ref)
00463       call calculate_density_second_derivs_teos10(t, s, pres, d2rdss, d2rdst, &
00464                                                   d2rdtt, d2rdsp, d2rdtp, 1, npts)
00465     case default
00466       call mom_error(fatal, "calculate_stanley_density_scalar: EOS is not valid.")
00467   end select
00468
00469   ! Equation 25 of Stanley et al., 2020.
00470   do i=is,ie
00471     rho(i) = rho(i) &
00472              + ( 0.5 * d2rdtt(i) * tvar(i) + ( d2rdst(i) * tscov(i) + 0.5 * d2rdss(i) * svar(i) ) )
00473   enddo
00474
00475   rho_scale = eos%kg_m3_to_R
00476   if (present(scale)) rho_scale = rho_scale * scale
00477   if (rho_scale /= 1.0) then ; do i=is,ie
00478     rho(i) = rho_scale * rho(i)
00479   enddo ; endif
00480
```

42.3.2.19 calculate_stanley_density_array()

```
subroutine mom_eos::calculate_stanley_density_array (
             real, dimension(:), intent(in) T,
             real, dimension(:), intent(in) S,
             real, dimension(:), intent(in) pressure,
             real, dimension(:), intent(in) Tvar,
             real, dimension(:), intent(in) TScov,
```

```
real, dimension(:), intent(in) Svar,
real, dimension(:), intent(inout) rho,
integer, intent(in) start,
integer, intent(in) npts,
type(eos_type), pointer EOS,
real, intent(in), optional rho_ref,
real, intent(in), optional scale )  [private]
```

Calls the appropriate subroutine to calculate the density of sea water for 1-D array inputs including the variance of T, S and covariance of T-S. The calculation uses only the second order correction in a series as discussed in Stanley et al., 2020. If rho_ref is present, the anomaly with respect to rho_ref is returned.

Parameters

in	*t*	Potential temperature referenced to the surface [degC]
in	*s*	Salinity [ppt]
in	*pressure*	Pressure [Pa]
in	*tvar*	Variance of potential temperature referenced to the surface [degC2]
in	*tscov*	Covariance of potential temperature and salinity [degC ppt]
in	*svar*	Variance of salinity [ppt2]
in,out	*rho*	Density (in-situ if pressure is local) [kg m-3]
in	*start*	Start index for computation
in	*npts*	Number of point to compute
	eos	Equation of state structure
in	*rho_ref*	A reference density [kg m-3].
in	*scale*	A multiplicative factor by which to scale density from kg m-3 to the desired units [R m3 kg-1]

Definition at line 306 of file MOM_EOS.F90.

References eos_linear, eos_teos10, and eos_wright.

```
00306   real, dimension(:), intent(in)    :: t        !< Potential temperature referenced to the surface [degC]
00307   real, dimension(:), intent(in)    :: s        !< Salinity [ppt]
00308   real, dimension(:), intent(in)    :: pressure !< Pressure [Pa]
00309   real, dimension(:), intent(in)    :: tvar     !< Variance of potential temperature referenced to the
    surface [degC2]
00310   real, dimension(:), intent(in)    :: tscov    !< Covariance of potential temperature and salinity [degC
    ppt]
00311   real, dimension(:), intent(in)    :: svar     !< Variance of salinity [ppt2]
00312   real, dimension(:), intent(inout) :: rho      !< Density (in-situ if pressure is local) [kg m-3]
00313   integer,            intent(in)    :: start    !< Start index for computation
00314   integer,            intent(in)    :: npts     !< Number of point to compute
00315   type(eos_type),     pointer       :: eos      !< Equation of state structure
00316   real,     optional, intent(in)    :: rho_ref  !< A reference density [kg m-3].
00317   real,     optional, intent(in)    :: scale    !< A multiplicative factor by which to scale density
00318                                                 !! from kg m-3 to the desired units [R m3 kg-1]
00319   ! Local variables
00320   real, dimension(size(T)) :: d2rdtt, d2rdst, d2rdss, d2rdsp, d2rdtp ! Second derivatives of density wrt
    T,S,p
00321   integer :: j
00322
00323   if (.not.associated(eos)) call mom_error(fatal, &
00324     "calculate_density_array called with an unassociated EOS_type EOS.")
00325
00326   select case (eos%form_of_EOS)
00327     case (eos_linear)
00328       call calculate_density_linear(t, s, pressure, rho, start, npts, &
00329                                     eos%Rho_T0_S0, eos%dRho_dT, eos%dRho_dS, rho_ref)
00330       call calculate_density_second_derivs_linear(t, s, pressure, d2rdss, d2rdst, &
00331                                                   d2rdtt, d2rdsp, d2rdtp, start, npts)
00332     case (eos_wright)
00333       call calculate_density_wright(t, s, pressure, rho, start, npts, rho_ref)
00334       call calculate_density_second_derivs_wright(t, s, pressure, d2rdss, d2rdst, &
00335                                                   d2rdtt, d2rdsp, d2rdtp, start, npts)
```

```
00336     case (eos_teos10)
00337       call calculate_density_teos10(t, s, pressure, rho, start, npts, rho_ref)
00338       call calculate_density_second_derivs_teos10(t, s, pressure, d2rdss, d2rdst, &
00339                                                   d2rdtt, d2rdsp, d2rdtp, start, npts)
00340     case default
00341       call mom_error(fatal, "calculate_stanley_density_array: EOS%form_of_EOS is not valid.")
00342   end select
00343
00344   ! Equation 25 of Stanley et al., 2020.
00345   do j=start,start+npts-1
00346     rho(j) = rho(j) &
00347              + ( 0.5 * d2rdtt(j) * tvar(j) + ( d2rdst(j) * tscov(j) + 0.5 * d2rdss(j) * svar(j) ) )
00348   enddo
00349
00350   if (present(scale)) then ; if (scale /= 1.0) then ; do j=start,start+npts-1
00351     rho(j) = scale * rho(j)
00352   enddo ; endif ; endif
00353
```

42.3.2.20 calculate_stanley_density_scalar()

```
subroutine mom_eos::calculate_stanley_density_scalar (
              real, intent(in) T,
              real, intent(in) S,
              real, intent(in) pressure,
              real, intent(in) Tvar,
              real, intent(in) TScov,
              real, intent(in) Svar,
              real, intent(out) rho,
              type(eos_type), pointer EOS,
              real, intent(in), optional rho_ref,
              real, intent(in), optional scale )  [private]
```

Calls the appropriate subroutine to calculate density of sea water for scalar inputs including the variance of T, S and covariance of T-S. The calculation uses only the second order correction in a series as discussed in Stanley et al., 2020. If rho_ref is present, the anomaly with respect to rho_ref is returned. The density can be rescaled using rho_ref.

Parameters

in	*t*	Potential temperature referenced to the surface [degC]
in	*s*	Salinity [ppt]
in	*tvar*	Variance of potential temperature referenced to the surface [degC2]
in	*tscov*	Covariance of potential temperature and salinity [degC ppt]
in	*svar*	Variance of salinity [ppt2]
in	*pressure*	Pressure [Pa]
out	*rho*	Density (in-situ if pressure is local) [kg m-3] or [R ~> kg m-3]
	eos	Equation of state structure
in	*rho_ref*	A reference density [kg m-3].
in	*scale*	A multiplicative factor by which to scale density from kg m-3 to the desired units [R m3 kg-1]

Definition at line 212 of file MOM_EOS.F90.

References eos_linear, eos_teos10, and eos_wright.

```
00212   real,           intent(in)  :: t        !< Potential temperature referenced to the surface [degC]
```

```
00213   real,           intent(in)  :: s        !< Salinity [ppt]
00214   real,           intent(in)  :: tvar     !< Variance of potential temperature referenced to the surface
      [degC2]
00215   real,           intent(in)  :: tscov    !< Covariance of potential temperature and salinity [degC ppt]
00216   real,           intent(in)  :: svar     !< Variance of salinity [ppt2]
00217   real,           intent(in)  :: pressure !< Pressure [Pa]
00218   real,           intent(out) :: rho      !< Density (in-situ if pressure is local) [kg m-3] or [R ~> kg
      m-3]
00219   type(eos_type), pointer     :: eos      !< Equation of state structure
00220   real, optional, intent(in)  :: rho_ref  !< A reference density [kg m-3].
00221   real, optional, intent(in)  :: scale    !< A multiplicative factor by which to scale density
00222                                           !! from kg m-3 to the desired units [R m3 kg-1]
00223   ! Local variables
00224   real :: d2rdtt, d2rdst, d2rdss, d2rdsp, d2rdtp ! Second derivatives of density wrt T,S,p
00225   real :: rho_scale ! A factor to convert density from kg m-3 to the desired units [R m3 kg-1 ~> 1]
00226   real :: p_scale   ! A factor to convert pressure to units of Pa [Pa T2 R-1 L-2 ~> 1]
00227
00228   if (.not.associated(eos)) call mom_error(fatal, &
00229     "calculate_stanley_density_scalar called with an unassociated EOS_type EOS.")
00230
00231   p_scale = eos%RL2_T2_to_Pa
00232
00233   select case (eos%form_of_EOS)
00234     case (eos_linear)
00235       call calculate_density_linear(t, s, p_scale*pressure, rho, &
00236                                     eos%Rho_T0_S0, eos%dRho_dT, eos%dRho_dS, rho_ref)
00237       call calculate_density_second_derivs_linear(t, s, pressure, d2rdss, d2rdst, &
00238                                                   d2rdtt, d2rdsp, d2rdtp)
00239     case (eos_wright)
00240       call calculate_density_wright(t, s, p_scale*pressure, rho, rho_ref)
00241       call calculate_density_second_derivs_wright(t, s, pressure, d2rdss, d2rdst, &
00242                                                   d2rdtt, d2rdsp, d2rdtp)
00243     case (eos_teos10)
00244       call calculate_density_teos10(t, s, p_scale*pressure, rho, rho_ref)
00245       call calculate_density_second_derivs_teos10(t, s, pressure, d2rdss, d2rdst, &
00246                                                   d2rdtt, d2rdsp, d2rdtp)
00247     case default
00248       call mom_error(fatal, "calculate_stanley_density_scalar: EOS is not valid.")
00249   end select
00250
00251   ! Equation 25 of Stanley et al., 2020.
00252   rho = rho + ( 0.5 * d2rdtt * tvar + ( d2rdst * tscov + 0.5 * d2rdss * svar ) )
00253
00254   rho_scale = eos%kg_m3_to_R
00255   if (present(scale)) rho_scale = rho_scale * scale
00256   rho = rho_scale * rho
00257
```

42.3.2.21 calculate_tfreeze_array()

```
subroutine mom_eos::calculate_tfreeze_array (
             real, dimension(:), intent(in) S,
             real, dimension(:), intent(in) pressure,
             real, dimension(:), intent(inout) T_fr,
             integer, intent(in) start,
             integer, intent(in) npts,
             type(eos_type), pointer EOS,
             real, intent(in), optional pres_scale )  [private]
```

Calls the appropriate subroutine to calculate the freezing point for a 1-D array.

Parameters

in	*s*	Salinity [ppt]
in	*pressure*	Pressure [Pa] or [other]
in,out	*t_fr*	Freezing point potential temperature referenced to the surface [degC]
in	*start*	Starting index within the array
in	*npts*	The number of values to calculate
	eos	Equation of state structure
in	*pres_scale*	A multiplicative factor to convert pressure into Pa.

Definition at line 656 of file MOM_EOS.F90.

References tfreeze_linear, tfreeze_millero, and tfreeze_teos10.

```
00656   real, dimension(:), intent(in)    :: s        !< Salinity [ppt]
00657   real, dimension(:), intent(in)    :: pressure !< Pressure [Pa] or [other]
00658   real, dimension(:), intent(inout) :: t_fr     !< Freezing point potential temperature referenced
00659                                                 !! to the surface [degC]
00660   integer,            intent(in)    :: start    !< Starting index within the array
00661   integer,            intent(in)    :: npts     !< The number of values to calculate
00662   type(eos_type),     pointer       :: eos      !< Equation of state structure
00663   real,     optional, intent(in)    :: pres_scale !< A multiplicative factor to convert pressure into Pa.
00664
00665   ! Local variables
00666   real, dimension(size(pressure)) :: pres  ! Pressure converted to [Pa]
00667   real :: p_scale ! A factor to convert pressure to units of Pa.
00668   integer :: j
00669
00670   if (.not.associated(eos)) call mom_error(fatal, &
00671     "calculate_TFreeze_scalar called with an unassociated EOS_type EOS.")
00672
00673   p_scale = 1.0 ; if (present(pres_scale)) p_scale = pres_scale
00674
00675   if (p_scale == 1.0) then
00676     select case (eos%form_of_TFreeze)
00677       case (tfreeze_linear)
00678         call calculate_tfreeze_linear(s, pressure, t_fr, start, npts, &
00679                                       eos%TFr_S0_P0, eos%dTFr_dS, eos%dTFr_dp)
00680       case (tfreeze_millero)
00681         call calculate_tfreeze_millero(s, pressure, t_fr, start, npts)
00682       case (tfreeze_teos10)
00683         call calculate_tfreeze_teos10(s, pressure, t_fr, start, npts)
00684       case default
00685         call mom_error(fatal, "calculate_TFreeze_scalar: form_of_TFreeze is not valid.")
00686     end select
00687   else
00688     do j=start,start+npts-1 ; pres(j) = p_scale * pressure(j) ; enddo
00689     select case (eos%form_of_TFreeze)
00690       case (tfreeze_linear)
00691         call calculate_tfreeze_linear(s, pres, t_fr, start, npts, &
00692                                       eos%TFr_S0_P0, eos%dTFr_dS, eos%dTFr_dp)
00693       case (tfreeze_millero)
00694         call calculate_tfreeze_millero(s, pres, t_fr, start, npts)
00695       case (tfreeze_teos10)
00696         call calculate_tfreeze_teos10(s, pres, t_fr, start, npts)
00697       case default
00698         call mom_error(fatal, "calculate_TFreeze_scalar: form_of_TFreeze is not valid.")
00699     end select
00700   endif
00701
```

42.3.2.22 calculate_tfreeze_scalar()

```
subroutine mom_eos::calculate_tfreeze_scalar (
             real, intent(in) S,
             real, intent(in) pressure,
             real, intent(out) T_fr,
             type(eos_type), pointer EOS,
             real, intent(in), optional pres_scale )  [private]
```

Calls the appropriate subroutine to calculate the freezing point for scalar inputs.

Parameters

in	*s*	Salinity [ppt]
in	*pressure*	Pressure [Pa] or [other]
out	*t_fr*	Freezing point potential temperature referenced to the surface [degC]
	eos	Equation of state structure
in	*pres_scale*	A multiplicative factor to convert pressure into Pa

Definition at line 625 of file MOM_EOS.F90.

References tfreeze_linear, tfreeze_millero, and tfreeze_teos10.

```
00625   real,           intent(in)  :: s !< Salinity [ppt]
00626   real,           intent(in)  :: pressure !< Pressure [Pa] or [other]
00627   real,           intent(out) :: t_fr !< Freezing point potential temperature referenced
00628                                       !! to the surface [degC]
00629   type(eos_type), pointer     :: eos !< Equation of state structure
00630   real, optional, intent(in)  :: pres_scale !< A multiplicative factor to convert pressure into Pa
00631
00632   ! Local variables
00633   real :: p_scale ! A factor to convert pressure to units of Pa.
00634
00635   if (.not.associated(eos)) call mom_error(fatal, &
00636     "calculate_TFreeze_scalar called with an unassociated EOS_type EOS.")
00637
00638   p_scale = 1.0 ; if (present(pres_scale)) p_scale = pres_scale
00639
00640   select case (eos%form_of_TFreeze)
00641     case (tfreeze_linear)
00642       call calculate_tfreeze_linear(s, p_scale*pressure, t_fr, eos%TFr_S0_P0, &
00643                                     eos%dTFr_dS, eos%dTFr_dp)
00644     case (tfreeze_millero)
00645       call calculate_tfreeze_millero(s, p_scale*pressure, t_fr)
00646     case (tfreeze_teos10)
00647       call calculate_tfreeze_teos10(s, p_scale*pressure, t_fr)
00648     case default
00649       call mom_error(fatal, "calculate_TFreeze_scalar: form_of_TFreeze is not valid.")
00650   end select
00651
```

42.3.2.23 convert_temp_salt_for_teos10()

```
subroutine, public mom_eos::convert_temp_salt_for_teos10 (
              real, dimension(hi%isd:hi%ied,hi%jsd:hi%jed,kd), intent(inout) T,
              real, dimension(hi%isd:hi%ied,hi%jsd:hi%jed,kd), intent(inout) S,
              type(hor_index_type), intent(in) HI,
              integer, intent(in) kd,
              real, dimension(hi%isd:hi%ied,hi%jsd:hi%jed,kd), intent(in) mask_z,
              type(eos_type), pointer EOS )
```

Convert T&S to Absolute Salinity and Conservative Temperature if using TEOS10.

Parameters

in	*kd*	The number of layers to work on
in	*hi*	The horizontal index structure
in,out	*t*	Potential temperature referenced to the surface [degC]
in,out	*s*	Salinity [ppt]
in	*mask_z*	3d mask regulating which points to convert.
	eos	Equation of state structure

Definition at line 1532 of file MOM_EOS.F90.

References eos_nemo, and eos_teos10.

```
01532   integer,              intent(in)    :: kd  !< The number of layers to work on
```

```
01533   type(hor_index_type),  intent(in)    :: hi       !< The horizontal index structure
01534   real, dimension(HI%isd:HI%ied,HI%jsd:HI%jed,kd), &
01535                          intent(inout) :: t   !< Potential temperature referenced to the surface [degC]
01536   real, dimension(HI%isd:HI%ied,HI%jsd:HI%jed,kd), &
01537                          intent(inout) :: s   !< Salinity [ppt]
01538   real, dimension(HI%isd:HI%ied,HI%jsd:HI%jed,kd), &
01539                          intent(in)    :: mask_z !< 3d mask regulating which points to convert.
01540   type(eos_type),        pointer       :: eos !< Equation of state structure
01541
01542   integer :: i, j, k
01543   real :: gsw_sr_from_sp, gsw_ct_from_pt, gsw_sa_from_sp
01544   real :: p
01545
01546   if (.not.associated(eos)) call mom_error(fatal, &
01547     "convert_temp_salt_to_TEOS10 called with an unassociated EOS_type EOS.")
01548
01549   if ((eos%form_of_EOS /= eos_teos10) .and. (eos%form_of_EOS /= eos_nemo)) return
01550
01551   do k=1,kd ; do j=hi%jsc,hi%jec ; do i=hi%isc,hi%iec
01552     if (mask_z(i,j,k) >= 1.0) then
01553      s(i,j,k) = gsw_sr_from_sp(s(i,j,k))
01554 !     Get absolute salinity from practical salinity, converting pressures from Pascal to dbar.
01555 !     If this option is activated, pressure will need to be added as an argument, and it should be
01556 !     moved out into module that is not shared between components, where the ocean_grid can be used.
01557 !     S(i,j,k) = gsw_sa_from_sp(S(i,j,k),pres(i,j,k)*1.0e-4,G%geoLonT(i,j),G%geoLatT(i,j))
01558      t(i,j,k) = gsw_ct_from_pt(s(i,j,k), t(i,j,k))
01559     endif
01560   enddo ; enddo ; enddo
```

42.3.2.24 eos_allocate()

```
subroutine, public mom_eos::eos_allocate (
              type(eos_type), pointer EOS )
```

Allocates EOS_type.

Parameters

eos	Equation of state structure

Definition at line 1491 of file MOM_EOS.F90.

Referenced by eos_init().

```
01491   type(eos_type), pointer :: eos !< Equation of state structure
01492
01493   if (.not.associated(eos)) allocate(eos)
```

42.3.2.25 eos_domain()

```
integer function, dimension(2), public mom_eos::eos_domain (
              type(hor_index_type), intent(in) HI,
              integer, intent(in), optional halo )
```

This subroutine returns a two point integer array indicating the domain of i-indices to work on in EOS calls based on information from a hor_index type.

Parameters

in	*hi*	The horizontal index structure
in	*halo*	The halo size to work on; missing is equivalent to 0.

Returns

The index domain that the EOS will work on, taking into account that the arrays inside the EOS routines will start at 1.

Definition at line 1163 of file MOM_EOS.F90.

```
01163   type(hor_index_type), intent(in)  :: hi    !< The horizontal index structure
01164   integer,    optional, intent(in)  :: halo  !< The halo size to work on; missing is equivalent to 0.
01165   integer, dimension(2) :: eosdom   !< The index domain that the EOS will work on, taking into account
01166                                     !! that the arrays inside the EOS routines will start at 1.
01167
01168   ! Local variables
01169   integer :: halo_sz
01170
01171   halo_sz = 0 ; if (present(halo)) halo_sz = halo
01172
01173   eosdom(1) = hi%isc - (hi%isd-1) - halo_sz
01174   eosdom(2) = hi%iec - (hi%isd-1) + halo_sz
01175
```

42.3.2.26 eos_end()

```
subroutine, public mom_eos::eos_end (
             type(eos_type), pointer EOS )
```

Deallocates EOS_type.

Parameters

eos	Equation of state structure

Definition at line 1498 of file MOM_EOS.F90.

```
01498   type(eos_type), pointer :: eos !< Equation of state structure
01499
01500   if (associated(eos)) deallocate(eos)
```

42.3.2.27 eos_init()

```
subroutine, public mom_eos::eos_init (
             type(param_file_type), intent(in) param_file,
             type(eos_type), pointer EOS,
             type(unit_scale_type), intent(in), optional US )
```

Initializes EOS_type by allocating and reading parameters.

Parameters

in	*param_file*	Parameter file structure
	eos	Equation of state structure
in	*us*	A dimensional unit scaling type

Definition at line 1349 of file MOM_EOS.F90.

References eos_allocate(), eos_default, eos_linear, eos_linear_string, eos_nemo, eos_nemo_string, eos_teos10, eos_teos10_string, eos_unesco, eos_unesco_string, eos_wright, eos_wright_string, tfreeze_default, tfreeze_linear, tfreeze_linear_string, tfreeze_millero, tfreeze_millero_string, tfreeze_teos10, and tfreeze_teos10_string.

Referenced by mom_ice_shelf::initialize_ice_shelf().

```
01349   type(param_file_type), intent(in) :: param_file !< Parameter file structure
01350   type(eos_type),        pointer    :: eos !< Equation of state structure
01351   type(unit_scale_type), intent(in) :: us  !< A dimensional unit scaling type
01352   optional :: us
01353   ! Local variables
01354 #include "version_variable.h"
01355   character(len=40)  :: mdl = "MOM_EOS" ! This module's name.
01356   character(len=40)  :: tmpstr
01357
01358   if (.not.associated(eos)) call eos_allocate(eos)
01359
01360   ! Read all relevant parameters and write them to the model log.
01361   call log_version(param_file, mdl, version, "")
01362
01363   call get_param(param_file, mdl, "EQN_OF_STATE", tmpstr, &
01364                  "EQN_OF_STATE determines which ocean equation of state "//&
01365                  "should be used.  Currently, the valid choices are "//&
01366                  '"LINEAR", "UNESCO", "WRIGHT", "NEMO" and "TEOS10". '//&
01367                  "This is only used if USE_EOS is true.", default=eos_default)
01368   select case (uppercase(tmpstr))
01369     case (eos_linear_string)
01370       eos%form_of_EOS = eos_linear
01371     case (eos_unesco_string)
01372       eos%form_of_EOS = eos_unesco
01373     case (eos_wright_string)
01374       eos%form_of_EOS = eos_wright
01375     case (eos_teos10_string)
01376       eos%form_of_EOS = eos_teos10
01377     case (eos_nemo_string)
01378       eos%form_of_EOS = eos_nemo
01379     case default
01380       call mom_error(fatal, "interpret_eos_selection: EQN_OF_STATE "//&
01381                              trim(tmpstr) // "in input file is invalid.")
01382   end select
01383   call mom_mesg('interpret_eos_selection: equation of state set to "' // &
01384                 trim(tmpstr)//'"', 5)
01385
01386   if (eos%form_of_EOS == eos_linear) then
01387     eos%Compressible = .false.
01388     call get_param(param_file, mdl, "RHO_T0_S0", eos%Rho_T0_S0, &
01389                    "When EQN_OF_STATE="//trim(eos_linear_string)//", "//&
01390                    "this is the density at T=0, S=0.", units="kg m-3", &
01391                    default=1000.0)
01392     call get_param(param_file, mdl, "DRHO_DT", eos%dRho_dT, &
01393                    "When EQN_OF_STATE="//trim(eos_linear_string)//", "//&
01394                    "this is the partial derivative of density with "//&
01395                    "temperature.", units="kg m-3 K-1", default=-0.2)
01396     call get_param(param_file, mdl, "DRHO_DS", eos%dRho_dS, &
01397                    "When EQN_OF_STATE="//trim(eos_linear_string)//", "//&
01398                    "this is the partial derivative of density with "//&
01399                    "salinity.", units="kg m-3 PSU-1", default=0.8)
01400   endif
01401
01402   call get_param(param_file, mdl, "EOS_QUADRATURE", eos%EOS_quadrature, &
01403                  "If true, always use the generic (quadrature) code "//&
01404                  "code for the integrals of density.", default=.false.)
01405
01406   call get_param(param_file, mdl, "TFREEZE_FORM", tmpstr, &
01407                  "TFREEZE_FORM determines which expression should be "//&
01408                  "used for the freezing point.  Currently, the valid "//&
01409                  'choices are "LINEAR", "MILLERO_78", "TEOS10"', &
01410                  default=tfreeze_default)
01411   select case (uppercase(tmpstr))
```

```
01412     case (tfreeze_linear_string)
01413       eos%form_of_TFreeze = tfreeze_linear
01414     case (tfreeze_millero_string)
01415       eos%form_of_TFreeze = tfreeze_millero
01416     case (tfreeze_teos10_string)
01417       eos%form_of_TFreeze = tfreeze_teos10
01418     case default
01419       call mom_error(fatal, "interpret_eos_selection:  TFREEZE_FORM "//&
01420                              trim(tmpstr) // "in input file is invalid.")
01421   end select
01422
01423   if (eos%form_of_TFreeze == tfreeze_linear) then
01424     call get_param(param_file, mdl, "TFREEZE_S0_P0",eos%TFr_S0_P0, &
01425                  "When TFREEZE_FORM="//trim(tfreeze_linear_string)//", "//&
01426                  "this is the freezing potential temperature at "//&
01427                  "S=0, P=0.", units="deg C", default=0.0)
01428     call get_param(param_file, mdl, "DTFREEZE_DS",eos%dTFr_dS, &
01429                  "When TFREEZE_FORM="//trim(tfreeze_linear_string)//", "//&
01430                  "this is the derivative of the freezing potential "//&
01431                  "temperature with salinity.", &
01432                  units="deg C PSU-1", default=-0.054)
01433     call get_param(param_file, mdl, "DTFREEZE_DP",eos%dTFr_dP, &
01434                  "When TFREEZE_FORM="//trim(tfreeze_linear_string)//", "//&
01435                  "this is the derivative of the freezing potential "//&
01436                  "temperature with pressure.", &
01437                  units="deg C Pa-1", default=0.0)
01438   endif
01439
01440   if ((eos%form_of_EOS == eos_teos10 .OR. eos%form_of_EOS == eos_nemo) .AND. &
01441       eos%form_of_TFreeze /= tfreeze_teos10) then
01442       call mom_error(fatal, "interpret_eos_selection:  EOS_TEOS10 or EOS_NEMO \n" //&
01443       "should only be used along with TFREEZE_FORM = TFREEZE_TEOS10 .")
01444   endif
01445
01446   ! Unit conversions
01447   eos%m_to_Z = 1. ; if (present(us)) eos%m_to_Z = us%m_to_Z
01448   eos%kg_m3_to_R = 1. ; if (present(us)) eos%kg_m3_to_R = us%kg_m3_to_R
01449   eos%R_to_kg_m3 = 1. ; if (present(us)) eos%R_to_kg_m3 = us%R_to_kg_m3
01450   eos%RL2_T2_to_Pa = 1. ; if (present(us)) eos%RL2_T2_to_Pa = us%RL2_T2_to_Pa
01451   eos%L_T_to_m_s = 1. ; if (present(us)) eos%L_T_to_m_s = us%L_T_to_m_s
01452
```

42.3.2.28 eos_manual_init()

```
subroutine, public mom_eos::eos_manual_init (
             type(eos_type), pointer EOS,
             integer, intent(in), optional form_of_EOS,
             integer, intent(in), optional form_of_TFreeze,
             logical, intent(in), optional EOS_quadrature,
             logical, intent(in), optional Compressible,
             real, intent(in), optional Rho_T0_S0,
             real, intent(in), optional drho_dT,
             real, intent(in), optional dRho_dS,
             real, intent(in), optional TFr_S0_P0,
             real, intent(in), optional dTFr_dS,
             real, intent(in), optional dTFr_dp )
```

Manually initialized an EOS type (intended for unit testing of routines which need a specific EOS)

Parameters

	eos	Equation of state structure
in	*form_of_eos*	A coded integer indicating the equation of state to use.
in	*form_of_tfreeze*	A coded integer indicating the expression for the potential temperature of the freezing point.
in	*eos_quadrature*	If true, always use the generic (quadrature) code for the integrals of density.

Parameters

in	*compressible*	If true, in situ density is a function of pressure.
in	*rho_t0_s0*	Density at T=0 degC and S=0 ppt [kg m-3]
in	*drho_dt*	Partial derivative of density with temperature in [kg m-3 degC-1]
in	*drho_ds*	Partial derivative of density with salinity in [kg m-3 ppt-1]
in	*tfr_s0_p0*	The freezing potential temperature at S=0, P=0 [degC]
in	*dtfr_ds*	The derivative of freezing point with salinity in [degC ppt-1]
in	*dtfr_dp*	The derivative of freezing point with pressure in [degC Pa-1]

Definition at line 1458 of file MOM_EOS.F90.

```
01458   type(eos_type),    pointer    :: eos !< Equation of state structure
01459   integer, optional, intent(in) :: form_of_eos !< A coded integer indicating the equation of state to use.
01460   integer, optional, intent(in) :: form_of_tfreeze !< A coded integer indicating the expression for
01461                                         !! the potential temperature of the freezing point.
01462   logical, optional, intent(in) :: eos_quadrature !< If true, always use the generic (quadrature)
01463                                         !! code for the integrals of density.
01464   logical, optional, intent(in) :: compressible  !< If true, in situ density is a function of pressure.
01465   real   , optional, intent(in) :: rho_t0_s0 !< Density at T=0 degC and S=0 ppt [kg m-3]
01466   real   , optional, intent(in) :: drho_dt   !< Partial derivative of density with temperature
01467                                             !! in [kg m-3 degC-1]
01468   real   , optional, intent(in) :: drho_ds   !< Partial derivative of density with salinity
01469                                             !! in [kg m-3 ppt-1]
01470   real   , optional, intent(in) :: tfr_s0_p0 !< The freezing potential temperature at S=0, P=0 [degC]
01471   real   , optional, intent(in) :: dtfr_ds   !< The derivative of freezing point with salinity
01472                                             !! in [degC ppt-1]
01473   real   , optional, intent(in) :: dtfr_dp   !< The derivative of freezing point with pressure
01474                                             !! in [degC Pa-1]
01475
01476   if (present(form_of_eos    ))  eos%form_of_EOS     = form_of_eos
01477   if (present(form_of_tfreeze))) eos%form_of_TFreeze = form_of_tfreeze
01478   if (present(eos_quadrature ))  eos%EOS_quadrature  = eos_quadrature
01479   if (present(compressible   ))  eos%Compressible    = compressible
01480   if (present(rho_t0_s0      ))  eos%Rho_T0_S0       = rho_t0_s0
01481   if (present(drho_dt        ))  eos%drho_dT         = drho_dt
01482   if (present(drho_ds        ))  eos%dRho_dS         = drho_ds
01483   if (present(tfr_s0_p0      ))  eos%TFr_S0_P0       = tfr_s0_p0
01484   if (present(dtfr_ds        ))  eos%dTFr_dS         = dtfr_ds
01485   if (present(dtfr_dp        ))  eos%dTFr_dp         = dtfr_dp
01486
```

42.3.2.29 eos_quadrature()

```
logical function, public mom_eos::eos_quadrature (
            type(eos_type), pointer EOS )
```

Return value of EOS_quadrature.

Parameters

eos	Equation of state structure

Definition at line 1565 of file MOM_EOS.F90.

```
01565   type(eos_type),    pointer     :: eos !< Equation of state structure
01566
01567   eos_quadrature  = eos%EOS_quadrature
01568
```

42.3.2.30 eos_use_linear()

```
subroutine, public mom_eos::eos_use_linear (
              real, intent(in) Rho_T0_S0,
              real, intent(in) dRho_dT,
              real, intent(in) dRho_dS,
              type(eos_type), pointer EOS,
              logical, intent(in), optional use_quadrature )
```

Set equation of state structure (EOS) to linear with given coefficients.

Note

This routine is primarily for testing and allows a local copy of the EOS_type (EOS argument) to be set to use the linear equation of state independent from the rest of the model.

Parameters

in	*rho_t0_s0*	Density at T=0 degC and S=0 ppt [kg m-3]
in	*drho_dt*	Partial derivative of density with temperature [kg m-3 degC-1]
in	*drho_ds*	Partial derivative of density with salinity [kg m-3 ppt-1]
in	*use_quadrature*	If true, always use the generic (quadrature) code for the integrals of density.
	eos	Equation of state structure

Definition at line 1509 of file MOM_EOS.F90.

References eos_linear.

```
01509   real,              intent(in) :: rho_t0_s0 !< Density at T=0 degC and S=0 ppt [kg m-3]
01510   real,              intent(in) :: drho_dt   !< Partial derivative of density with temperature [kg m-3
      degC-1]
01511   real,              intent(in) :: drho_ds   !< Partial derivative of density with salinity [kg m-3 ppt-1]
01512   logical, optional, intent(in) :: use_quadrature !< If true, always use the generic (quadrature)
01513                                                   !! code for the integrals of density.
01514   type(eos_type),    pointer    :: eos       !< Equation of state structure
01515
01516   if (.not.associated(eos)) call mom_error(fatal, &
01517     "MOM_EOS.F90: EOS_use_linear() called with an unassociated EOS_type EOS.")
01518
01519   eos%form_of_EOS = eos_linear
01520   eos%Compressible = .false.
01521   eos%Rho_T0_S0 = rho_t0_s0
01522   eos%dRho_dT = drho_dt
01523   eos%dRho_dS = drho_ds
01524   eos%EOS_quadrature = .false.
01525   if (present(use_quadrature)) eos%EOS_quadrature = use_quadrature
01526
```

42.3.2.31 extract_member_eos()

```
subroutine, public mom_eos::extract_member_eos (
              type(eos_type), pointer EOS,
              integer, intent(out), optional form_of_EOS,
              integer, intent(out), optional form_of_TFreeze,
              logical, intent(out), optional EOS_quadrature,
```

```
logical, intent(out), optional Compressible,
real, intent(out), optional Rho_T0_S0,
real, intent(out), optional drho_dT,
real, intent(out), optional dRho_dS,
real, intent(out), optional TFr_S0_P0,
real, intent(out), optional dTFr_dS,
real, intent(out), optional dTFr_dp )
```

Extractor routine for the EOS type if the members need to be accessed outside this module.

Parameters

	eos	Equation of state structure
out	*form_of_eos*	A coded integer indicating the equation of state to use.
out	*form_of_tfreeze*	A coded integer indicating the expression for the potential temperature of the freezing point.
out	*eos_quadrature*	If true, always use the generic (quadrature) code for the integrals of density.
out	*compressible*	If true, in situ density is a function of pressure.
out	*rho_t0_s0*	Density at T=0 degC and S=0 ppt [kg m-3]
out	*drho_dt*	Partial derivative of density with temperature in [kg m-3 degC-1]
out	*drho_ds*	Partial derivative of density with salinity in [kg m-3 ppt-1]
out	*tfr_s0_p0*	The freezing potential temperature at S=0, P=0 [degC]
out	*dtfr_ds*	The derivative of freezing point with salinity [degC PSU-1]
out	*dtfr_dp*	The derivative of freezing point with pressure [degC Pa-1]

Definition at line 1574 of file MOM_EOS.F90.

```
01574   type(eos_type),    pointer     :: eos !< Equation of state structure
01575   integer, optional, intent(out) :: form_of_eos !< A coded integer indicating the equation of state to use.
01576   integer, optional, intent(out) :: form_of_tfreeze !< A coded integer indicating the expression for
01577                                         !! the potential temperature of the freezing point.
01578   logical, optional, intent(out) :: eos_quadrature !< If true, always use the generic (quadrature)
01579                                         !! code for the integrals of density.
01580   logical, optional, intent(out) :: compressible !< If true, in situ density is a function of pressure.
01581   real   , optional, intent(out) :: rho_t0_s0 !< Density at T=0 degC and S=0 ppt [kg m-3]
01582   real   , optional, intent(out) :: drho_dt   !< Partial derivative of density with temperature
01583                                                !! in [kg m-3 degC-1]
01584   real   , optional, intent(out) :: drho_ds   !< Partial derivative of density with salinity
01585                                                !! in [kg m-3 ppt-1]
01586   real   , optional, intent(out) :: tfr_s0_p0 !< The freezing potential temperature at S=0, P=0 [degC]
01587   real   , optional, intent(out) :: dtfr_ds   !< The derivative of freezing point with salinity
01588                                                !! [degC PSU-1]
01589   real   , optional, intent(out) :: dtfr_dp   !< The derivative of freezing point with pressure
01590                                                !! [degC Pa-1]
01591
01592   if (present(form_of_eos    ))  form_of_eos     = eos%form_of_EOS
01593   if (present(form_of_tfreeze))  form_of_tfreeze = eos%form_of_TFreeze
01594   if (present(eos_quadrature ))  eos_quadrature  = eos%EOS_quadrature
01595   if (present(compressible   ))  compressible    = eos%Compressible
01596   if (present(rho_t0_s0      ))  rho_t0_s0       = eos%Rho_T0_S0
01597   if (present(drho_dt        ))  drho_dt         = eos%drho_dT
01598   if (present(drho_ds        ))  drho_ds         = eos%dRho_dS
01599   if (present(tfr_s0_p0      ))  tfr_s0_p0       = eos%TFr_S0_P0
01600   if (present(dtfr_ds        ))  dtfr_ds         = eos%dTFr_dS
01601   if (present(dtfr_dp        ))  dtfr_dp         = eos%dTFr_dp
01602
```

42.3.2.32 query_compressible()

```
logical function, public mom_eos::query_compressible (
            type(eos_type), pointer EOS )
```

Returns true if the equation of state is compressible (i.e. has pressure dependence)

Parameters

eos	Equation of state structure

Definition at line 1339 of file MOM_EOS.F90.

```
01339   type(eos_type), pointer :: eos !< Equation of state structure
01340
01341   if (.not.associated(eos)) call mom_error(fatal, &
01342     "query_compressible called with an unassociated EOS_type EOS.")
01343
01344   query_compressible = eos%compressible
```

42.3.3 Variable Documentation

42.3.3.1 eos_default

```
character*(10), parameter mom_eos::eos_default = EOS_WRIGHT_STRING  [private]
```

The default equation of state.

Definition at line 148 of file MOM_EOS.F90.

Referenced by eos_init().

```
00148 character*(10), parameter :: EOS_DEFAULT = EOS_WRIGHT_STRING !< The default equation of state
```

42.3.3.2 eos_linear

```
integer, parameter, public mom_eos::eos_linear = 1
```

A named integer specifying an equation of state.

Definition at line 137 of file MOM_EOS.F90.

Referenced by analytic_int_density_dz(), analytic_int_specific_vol_dp(), calculate_compress_array(), calculate_density_array(), calculate_density_derivs_array(), calculate_density_derivs_scalar(), calculate_density_scalar(), calculate_density_second_derivs_array(), calculate_density_second_derivs_scalar(), calculate_spec_vol_array(), calculate_spec_vol_derivs_array(), calculate_stanley_density_1d(), calculate_stanley_density_array(), calculate_stanley_density_scalar(), eos_init(), and eos_use_linear().

```
00137 integer, parameter, public :: eos_linear = 1 !< A named integer specifying an equation of state
```

42.3.3.3 eos_linear_string

```
character*(10), parameter mom_eos::eos_linear_string = "LINEAR"  [private]
```

A string for specifying the equation of state.

Definition at line 143 of file MOM_EOS.F90.

Referenced by eos_init().

```
00143 character*(10), parameter :: EOS_LINEAR_STRING = "LINEAR" !< A string for specifying the equation of state
```

42.3.3.4 eos_nemo

```
integer, parameter, public mom_eos::eos_nemo = 5
```

A named integer specifying an equation of state.

Definition at line 141 of file MOM_EOS.F90.

Referenced by calculate_compress_array(), calculate_density_array(), calculate_density_derivs_array(), calculate_density_scalar(), calculate_spec_vol_array(), calculate_spec_vol_derivs_array(), convert_temp_salt↩_for_teos10(), and eos_init().

```
00141 integer, parameter, public :: eos_nemo   = 5 !< A named integer specifying an equation of state
```

42.3.3.5 eos_nemo_string

```
character*(10), parameter mom_eos::eos_nemo_string = "NEMO"  [private]
```

A string for specifying the equation of state.

Definition at line 147 of file MOM_EOS.F90.

Referenced by eos_init().

```
00147 character*(10), parameter :: EOS_NEMO_STRING   = "NEMO"   !< A string for specifying the equation of state
```

42.3.3.6 eos_teos10

```
integer, parameter, public mom_eos::eos_teos10 = 4
```

A named integer specifying an equation of state.

Definition at line 140 of file MOM_EOS.F90.

Referenced by calculate_compress_array(), calculate_density_array(), calculate_density_derivs_array(), calculate_density_derivs_scalar(), calculate_density_scalar(), calculate_density_second_derivs_array(), calculate_density_second_derivs_scalar(), calculate_spec_vol_array(), calculate_spec_vol_derivs_array(), calculate_stanley_density_1d(), calculate_stanley_density_array(), calculate_stanley_density_scalar(), convert_temp_salt_for_teos10(), and eos_init().

```
00140 integer, parameter, public :: eos_teos10 = 4 !< A named integer specifying an equation of state
```

42.3.3.7 eos_teos10_string

```
character*(10), parameter mom_eos::eos_teos10_string = "TEOS10"  [private]
```

A string for specifying the equation of state.

Definition at line 146 of file MOM_EOS.F90.

Referenced by eos_init().

```
00146 character*(10), parameter :: EOS_TEOS10_STRING = "TEOS10" !< A string for specifying the equation of state
```

42.3.3.8 eos_unesco

```
integer, parameter, public mom_eos::eos_unesco = 2
```

A named integer specifying an equation of state.

Definition at line 138 of file MOM_EOS.F90.

Referenced by calculate_compress_array(), calculate_density_array(), calculate_density_derivs_array(), calculate_density_scalar(), calculate_spec_vol_array(), calculate_spec_vol_derivs_array(), and eos_init().

```
00138 integer, parameter, public :: eos_unesco = 2 !< A named integer specifying an equation of state
```

42.3.3.9 eos_unesco_string

```
character*(10), parameter mom_eos::eos_unesco_string = "UNESCO"  [private]
```

A string for specifying the equation of state.

Definition at line 144 of file MOM_EOS.F90.

Referenced by eos_init().

```
00144 character*(10), parameter :: EOS_UNESCO_STRING = "UNESCO" !< A string for specifying the equation of state
```

42.3.3.10 eos_wright

```
integer, parameter, public mom_eos::eos_wright = 3
```

A named integer specifying an equation of state.

Definition at line 139 of file MOM_EOS.F90.

Referenced by analytic_int_density_dz(), analytic_int_specific_vol_dp(), calculate_compress_array(), calculate_density_array(), calculate_density_derivs_array(), calculate_density_derivs_scalar(), calculate_density_scalar(), calculate_density_second_derivs_array(), calculate_density_second_derivs_scalar(), calculate_spec_vol_array(), calculate_spec_vol_derivs_array(), calculate_stanley_density_1d(), calculate_stanley_density_array(), calculate_stanley_density_scalar(), and eos_init().

```
00139 integer, parameter, public :: eos_wright = 3 !< A named integer specifying an equation of state
```

42.3.3.11 eos_wright_string

```
character*(10), parameter mom_eos::eos_wright_string = "WRIGHT"  [private]
```

A string for specifying the equation of state.

Definition at line 145 of file MOM_EOS.F90.

Referenced by eos_init().

```
00145 character*(10), parameter :: EOS_WRIGHT_STRING = "WRIGHT" !< A string for specifying the equation of state
```

42.3.3.12 tfreeze_default

```
character*(10), parameter mom_eos::tfreeze_default = TFREEZE_LINEAR_STRING  [private]
```

The default freezing point expression.

Definition at line 157 of file MOM_EOS.F90.

Referenced by eos_init().

```
00157 character*(10), parameter :: TFREEZE_DEFAULT = TFREEZE_LINEAR_STRING !< The default freezing point
       expression
```

42.3.3.13 tfreeze_linear

```
integer, parameter mom_eos::tfreeze_linear = 1  [private]
```

A named integer specifying a freezing point expression.

Definition at line 150 of file MOM_EOS.F90.

Referenced by calculate_tfreeze_array(), calculate_tfreeze_scalar(), and eos_init().

```
00150 integer, parameter :: tfreeze_linear = 1  !< A named integer specifying a freezing point expression
```

42.3.3.14 tfreeze_linear_string

```
character*(10), parameter mom_eos::tfreeze_linear_string = "LINEAR"  [private]
```

A string for specifying the freezing point expression.

Definition at line 153 of file MOM_EOS.F90.

Referenced by eos_init().

```
00153 character*(10), parameter :: TFREEZE_LINEAR_STRING = "LINEAR" !< A string for specifying the freezing point
       expression
```

42.3.3.15 tfreeze_millero

```
integer, parameter mom_eos::tfreeze_millero = 2  [private]
```

A named integer specifying a freezing point expression.

Definition at line 151 of file MOM_EOS.F90.

Referenced by calculate_tfreeze_array(), calculate_tfreeze_scalar(), and eos_init().

```
00151 integer, parameter :: tfreeze_millero = 2 !< A named integer specifying a freezing point expression
```

42.3.3.16 tfreeze_millero_string

```
character*(10), parameter mom_eos::tfreeze_millero_string = "MILLERO_78"  [private]
```

A string for specifying freezing point expression.

Definition at line 154 of file MOM_EOS.F90.

Referenced by eos_init().

```
00154 character*(10), parameter :: TFREEZE_MILLERO_STRING = "MILLERO_78" !< A string for specifying
```

42.3.3.17 tfreeze_teos10

```
integer, parameter mom_eos::tfreeze_teos10 = 3  [private]
```

A named integer specifying a freezing point expression.

Definition at line 152 of file MOM_EOS.F90.

Referenced by calculate_tfreeze_array(), calculate_tfreeze_scalar(), and eos_init().

```
00152 integer, parameter :: tfreeze_teos10 = 3  !< A named integer specifying a freezing point expression
```

42.3.3.18 tfreeze_teos10_string

```
character*(10), parameter mom_eos::tfreeze_teos10_string = "TEOS10"  [private]
```

A string for specifying the freezing point expression.

Definition at line 156 of file MOM_EOS.F90.

Referenced by eos_init().

```
00156 character*(10), parameter :: TFREEZE_TEOS10_STRING = "TEOS10" !< A string for specifying the freezing point
       expression
```

42.4 mom_ice_shelf Module Reference

Implements the thermodynamic aspects of ocean / ice-shelf interactions, along with a crude placeholder for a later implementation of full ice shelf dynamics, all using the MOM framework and coding style.

Data Types

- type ice_shelf_cs

  Control structure that contains ice shelf parameters and diagnostics handles.

Functions/Subroutines

- subroutine, public shelf_calc_flux (sfc_state, fluxes, Time, time_step, CS, forces)

  Calculates fluxes between the ocean and ice-shelf using the three-equations formulation (optional to use just two equations). See ICE_SHELF equations.
- subroutine change_thickness_using_melt (ISS, G, US, time_step, fluxes, density_ice, debug)

  Changes the thickness (mass) of the ice shelf based on sub-ice-shelf melting.
- subroutine, public add_shelf_forces (G, US, CS, forces, do_shelf_area)

  This subroutine adds the mechanical forcing fields and perhaps shelf areas, based on the ice state in ice_shelf_CS.
- subroutine add_shelf_pressure (G, US, CS, fluxes)

  This subroutine adds the ice shelf pressure to the fluxes type.
- subroutine, public add_shelf_flux (G, US, CS, sfc_state, fluxes)

  Updates surface fluxes that are influenced by sub-ice-shelf melting.
- subroutine, public initialize_ice_shelf (param_file, ocn_grid, Time, CS, diag, forces, fluxes, Time_in, solo_↩ ice_sheet_in)

  Initializes shelf model data, parameters and diagnostics.
- subroutine initialize_shelf_mass (G, param_file, CS, ISS, new_sim)

  Initializes shelf mass based on three options (file, zero and user)
- subroutine update_shelf_mass (G, US, CS, ISS, Time)

  Updates the ice shelf mass using data from a file.
- subroutine, public ice_shelf_save_restart (CS, Time, directory, time_stamped, filename_suffix)

  Save the ice shelf restart file.
- subroutine, public ice_shelf_end (CS)

  Deallocates all memory associated with this module.
- subroutine, public solo_step_ice_shelf (CS, time_interval, nsteps, Time, min_time_step_in)

  This routine is for stepping a stand-alone ice shelf model without an ocean.

Variables

- integer id_clock_shelf

  CPU Clock for the ice shelf code.
- integer id_clock_pass

  CPU Clock for group pass calls.

42.4.1 Detailed Description

Implements the thermodynamic aspects of ocean / ice-shelf interactions, along with a crude placeholder for a later implementation of full ice shelf dynamics, all using the MOM framework and coding style.

42.4.2 section_ICE_SHELF

This module implements the thermodynamic aspects of ocean/ice-shelf inter-actions using the MOM framework and coding style.

Derived from code by Chris Little, early 2010.

The ice-sheet dynamics subroutines do the following: initialize_shelf_mass - Initializes the ice shelf mass distribution.

- Initializes h_shelf, h_mask, area_shelf_h
- CURRENTLY: initializes mass_shelf as well, but this is unnecessary, as mass_shelf is initialized based on h↩ _shelf and density_ice immediately afterwards. Possibly subroutine should be renamed update_shelf_mass - updates ice shelf mass via netCDF file USER_update_shelf_mass (TODO). solo_step_ice_shelf - called only in ice-only mode. shelf_calc_flux - after melt rate & fluxes are calculated, ice dynamics are done. currently mass_shelf is updated immediately after ice_shelf_advect in fully dynamic mode.

NOTES: be aware that hmask(:,:) has a number of functions; it is used for front advancement, for subroutines in the velocity solve, and for thickness boundary conditions (this last one may be removed). in other words, interfering with its updates will have implications you might not expect.

Overall issues: Many variables need better documentation and units and the subgrid on which they are discretized.

42.4.2.1 ICE_SHELF equations

The three fundamental equations are: Heat flux

$$\rho_w C_{pw} \gamma_T (T_w - T_b) = \rho_i \dot{m} L_f$$

Salt flux

$$\rho_w \gamma_s (S_w - S_b) = \rho_i \dot{m} S_b$$

Freezing temperature

$$T_b = a S_b + b + cP$$

where ....

42.4.2.2 References

Asay-Davis, Xylar S., Stephen L. Cornford, Benjamin K. Galton-Fenzi, Rupert M. Gladstone, G. Hilmar Gudmundsson, David M. Holland, Paul R. Holland, and Daniel F. Martin. Experimental design for three interrelated marine ice sheet and ocean model intercomparison projects: MISMIP v. 3 (MISMIP+), ISOMIP v. 2 (ISOMIP+) and MISOMIP v. 1 (MISOMIP1). Geoscientific Model Development 9, no. 7 (2016): 2471.

Goldberg, D. N., et al. Investigation of land ice-ocean interaction with a fully coupled ice-ocean model: 1. Model description and behavior. Journal of Geophysical Research: Earth Surface 117.F2 (2012).

Goldberg, D. N., et al. Investigation of land ice-ocean interaction with a fully coupled ice-ocean model: 2. Sensitivity to external forcings. Journal of Geophysical Research: Earth Surface 117.F2 (2012).

Holland, David M., and Adrian Jenkins. Modeling thermodynamic ice-ocean interactions at the base of an ice shelf. Journal of Physical Oceanography 29.8 (1999): 1787-1800.

42.4.3 Function/Subroutine Documentation

42.4.3.1 add_shelf_flux()

```
subroutine, public mom_ice_shelf::add_shelf_flux (
             type(ocean_grid_type), intent(inout) G,
             type(unit_scale_type), intent(in) US,
             type(ice_shelf_cs), pointer CS,
             type(surface), intent(inout) sfc_state,
             type(forcing), intent(inout) fluxes )
```

Updates surface fluxes that are influenced by sub-ice-shelf melting.

Parameters

in,out	*g*	The ocean's grid structure.
in	*us*	A dimensional unit scaling type
	cs	This module's control structure.
in,out	*sfc_state*	Surface ocean state
in,out	*fluxes*	A structure of surface fluxes that may be used/updated.

Definition at line 885 of file MOM_ice_shelf.F90.

References add_shelf_pressure().

Referenced by shelf_calc_flux().

```
00885   type(ocean_grid_type), intent(inout) :: g    !< The ocean's grid structure.
00886   type(unit_scale_type), intent(in)    :: us   !< A dimensional unit scaling type
00887   type(ice_shelf_cs),    pointer       :: cs   !< This module's control structure.
00888   type(surface),         intent(inout) :: sfc_state !< Surface ocean state
00889   type(forcing),         intent(inout) :: fluxes  !< A structure of surface fluxes that may be
      used/updated.
00890
00891   ! local variables
00892   real :: frac_shelf       !< The fractional area covered by the ice shelf [nondim].
00893   real :: frac_open        !< The fractional area of the ocean that is not covered by the ice shelf
      [nondim].
00894   real :: delta_mass_shelf !< Change in ice shelf mass over one time step [R Z m2 T-1 ~> kg s-1]
00895   real :: balancing_flux   !< The fresh water flux that balances the integrated melt flux [R Z T-1 ~> kg
      m-2 s-1]
00896   real :: balancing_area   !< total area where the balancing flux is applied [m2]
00897   type(time_type) :: dtime !< The time step as a time_type
00898   type(time_type) :: time0 !< The previous time (Time-dt)
00899   real, dimension(SZDI_(G),SZDJ_(G)) :: bal_frac  !< Fraction of the cell where the mass flux
00900                            !! balancing the net melt flux occurs, 0 to 1 [nondim]
00901   real, dimension(SZDI_(G),SZDJ_(G)) :: last_mass_shelf !< Ice shelf mass
00902                            !! at at previous time (Time-dt) [R Z ~> kg m-2]
00903   real, dimension(SZDI_(G),SZDJ_(G)) :: delta_float_mass   !< The change in the floating mass between
00904                            !! the two timesteps at (Time) and (Time-dt) [R Z ~> kg m-2].
00905   real, dimension(SZDI_(G),SZDJ_(G))  :: last_h_shelf !< Ice shelf thickness [Z ~> m]
00906                            !! at at previous time (Time-dt)
00907   real, dimension(SZDI_(G),SZDJ_(G))  :: last_hmask !< Ice shelf mask [nondim]
00908                            !! at at previous time (Time-dt)
00909   real, dimension(SZDI_(G),SZDJ_(G))  :: last_area_shelf_h !< Ice shelf area [L2 ~> m2]
00910                            !! at at previous time (Time-dt)
00911   type(ice_shelf_state), pointer :: iss => null() !< A structure with elements that describe
00912                                          !! the ice-shelf state
00913
00914   character(len=160) :: mesg  ! The text of an error message
00915   integer :: i, j, is, ie, js, je, isd, ied, jsd, jed
00916   is = g%isc ; ie = g%iec ; js = g%jsc ; je = g%jec
00917   isd = g%isd ; jsd = g%jsd ; ied = g%ied ; jed = g%jed
```

```
00918
00919   if ((cs%grid%isc /= g%isc) .or. (cs%grid%iec /= g%iec) .or. &
00920       (cs%grid%jsc /= g%jsc) .or. (cs%grid%jec /= g%jec)) &
00921     call mom_error(fatal,"add_shelf_flux: Incompatible ocean and ice shelf grids.")
00922
00923   iss => cs%ISS
00924
00925
00926   call add_shelf_pressure(g, us, cs, fluxes)
00927
00928   ! Determine ustar and the square magnitude of the velocity in the
00929   ! bottom boundary layer. Together these give the TKE source and
00930   ! vertical decay scale.
00931
00932   if (cs%debug) then
00933     if (allocated(sfc_state%taux_shelf) .and. allocated(sfc_state%tauy_shelf)) then
00934       call uvchksum("tau[xy]_shelf", sfc_state%taux_shelf, sfc_state%tauy_shelf, &
00935                     g%HI, haloshift=0, scale=us%RZ_T_to_kg_m2s*us%L_T_to_m_s)
00936     endif
00937   endif
00938
00939   if (cs%active_shelf_dynamics .or. cs%override_shelf_movement) then
00940     do j=jsd,jed ; do i=isd,ied
00941       if (g%areaT(i,j) > 0.0) &
00942         fluxes%frac_shelf_h(i,j) = min(1.0, iss%area_shelf_h(i,j) * g%IareaT(i,j))
00943     enddo ; enddo
00944   endif
00945
00946   if (cs%debug) then
00947     call mom_forcing_chksum("Before adding shelf fluxes", fluxes, g, cs%US, haloshift=0)
00948   endif
00949
00950   do j=js,je ; do i=is,ie ; if (iss%area_shelf_h(i,j) > 0.0) then
00951     ! Replace fluxes intercepted by the ice shelf with fluxes from the ice shelf
00952     frac_shelf = min(1.0, iss%area_shelf_h(i,j) * g%IareaT(i,j))
00953     frac_open = max(0.0, 1.0 - frac_shelf)
00954
00955     if (associated(fluxes%sw)) fluxes%sw(i,j) = frac_open * fluxes%sw(i,j)
00956     if (associated(fluxes%sw_vis_dir)) fluxes%sw_vis_dir(i,j) = frac_open * fluxes%sw_vis_dir(i,j)
00957     if (associated(fluxes%sw_vis_dif)) fluxes%sw_vis_dif(i,j) = frac_open * fluxes%sw_vis_dif(i,j)
00958     if (associated(fluxes%sw_nir_dir)) fluxes%sw_nir_dir(i,j) = frac_open * fluxes%sw_nir_dir(i,j)
00959     if (associated(fluxes%sw_nir_dif)) fluxes%sw_nir_dif(i,j) = frac_open * fluxes%sw_nir_dif(i,j)
00960     if (associated(fluxes%lw)) fluxes%lw(i,j) = frac_open * fluxes%lw(i,j)
00961     if (associated(fluxes%latent)) fluxes%latent(i,j) = frac_open * fluxes%latent(i,j)
00962     if (associated(fluxes%evap)) fluxes%evap(i,j) = frac_open * fluxes%evap(i,j)
00963     if (associated(fluxes%lprec)) then
00964       if (iss%water_flux(i,j) > 0.0) then
00965         fluxes%lprec(i,j) =  frac_shelf*iss%water_flux(i,j)*cs%flux_factor + frac_open * fluxes%lprec(i,j)
00966       else
00967         fluxes%lprec(i,j) = frac_open * fluxes%lprec(i,j)
00968         fluxes%evap(i,j) = fluxes%evap(i,j) + frac_shelf*iss%water_flux(i,j)*cs%flux_factor
00969       endif
00970     endif
00971
00972     if (associated(fluxes%sens)) &
00973       fluxes%sens(i,j) = frac_shelf*iss%tflux_ocn(i,j)*cs%flux_factor + frac_open * fluxes%sens(i,j)
00974     ! The salt flux should be mostly from sea ice, so perhaps none should be intercepted and this should be
      changed.
00975     if (associated(fluxes%salt_flux)) &
00976       fluxes%salt_flux(i,j) = frac_shelf * iss%salt_flux(i,j)*cs%flux_factor + frac_open * fluxes%salt_flux
     (i,j)
00977   endif ; enddo ; enddo
00978
00979   if (cs%debug) then
00980     call hchksum(iss%water_flux, "water_flux add shelf fluxes", g%HI, haloshift=0, scale=us%RZ_T_to_kg_m2s)
00981     call hchksum(iss%tflux_ocn, "tflux_ocn add shelf fluxes", g%HI, haloshift=0, scale=us%QRZ_T_to_W_m2)
00982     call mom_forcing_chksum("After adding shelf fluxes", fluxes, g, cs%US, haloshift=0)
00983   endif
00984
00985   ! Keep sea level constant by removing mass via a balancing flux that might be applied
00986   ! in the open ocean or the sponge region (via virtual precip, vprec). Apply additional
00987   ! salt/heat fluxes so that the resultant surface buoyancy forcing is ~ 0.
00988   ! This is needed for some of the ISOMIP+ experiments.
00989
00990   if (cs%constant_sea_level) then
00991     if (.not. associated(fluxes%salt_flux)) allocate(fluxes%salt_flux(ie,je))
00992     if (.not. associated(fluxes%vprec)) allocate(fluxes%vprec(ie,je))
00993     fluxes%salt_flux(:,:) = 0.0 ; fluxes%vprec(:,:) = 0.0
00994
00995     ! take into account changes in mass (or thickness) when imposing ice shelf mass
00996     if (cs%override_shelf_movement .and. cs%mass_from_file) then
00997       dtime = real_to_time(cs%time_step)
00998
00999       ! Compute changes in mass after at least one full time step
01000       if (cs%Time > dtime) then
01001         time0 = cs%Time - dtime
01002         do j=js,je ; do i=is,ie
```

```
01003           last_hmask(i,j) = iss%hmask(i,j) ; last_area_shelf_h(i,j) = iss%area_shelf_h(i,j)
01004         enddo ; enddo
01005         call time_interp_external(cs%id_read_mass, time0, last_mass_shelf)
01006         do j=js,je ; do i=is,ie
01007         ! This should only be done if time_interp_external did an update.
01008           last_mass_shelf(i,j) = us%kg_m3_to_R*us%m_to_Z * last_mass_shelf(i,j) ! Rescale after time_interp
01009           last_h_shelf(i,j) = last_mass_shelf(i,j) / cs%density_ice
01010         enddo ; enddo
01011
01012         ! apply calving
01013         if (cs%min_thickness_simple_calve > 0.0) then
01014           call ice_shelf_min_thickness_calve(g, last_h_shelf, last_area_shelf_h, last_hmask, &
01015                                              cs%min_thickness_simple_calve, halo=0)
01016           ! convert to mass again
01017           do j=js,je ; do i=is,ie
01018             last_mass_shelf(i,j) = last_h_shelf(i,j) * cs%density_ice
01019           enddo ; enddo
01020         endif
01021
01022         ! get total ice shelf mass at (Time-dt) and (Time), in kg
01023         do j=js,je ; do i=is,ie
01024           ! Just consider the change in the mass of the floating shelf.
01025           if ((sfc_state%ocean_mass(i,j) > cs%min_ocean_mass_float) .and. &
01026               (iss%area_shelf_h(i,j) > 0.0)) then
01027             delta_float_mass(i,j) = iss%mass_shelf(i,j) - last_mass_shelf(i,j)
01028           else
01029             delta_float_mass(i,j) = 0.0
01030           endif
01031         enddo ; enddo
01032         delta_mass_shelf = us%kg_m2s_to_RZ_T*(global_area_integral(delta_float_mass, g, scale=us
     %RZ_to_kg_m2, &
01033                                                                    area=iss%area_shelf_h) / cs%time_step)
01034       else! first time step
01035         delta_mass_shelf = 0.0
01036       endif
01037     else ! ice shelf mass does not change
01038       delta_mass_shelf = 0.0
01039     endif
01040
01041     ! average total melt flux over sponge area
01042     do j=js,je ; do i=is,ie
01043       if ((g%mask2dT(i,j) > 0.0) .AND. (iss%area_shelf_h(i,j) * g%IareaT(i,j) < 1.0)) then
01044          ! Uncomment this for some ISOMIP cases:
01045          !  .AND. (G%geoLonT(i,j) >= 790.0) .AND. (G%geoLonT(i,j) <= 800.0)) then
01046         bal_frac(i,j) = max(1.0 - iss%area_shelf_h(i,j) * g%IareaT(i,j), 0.0)
01047       else
01048         bal_frac(i,j) = 0.0
01049       endif
01050     enddo ; enddo
01051
01052     balancing_area = global_area_integral(bal_frac, g)
01053     if (balancing_area > 0.0) then
01054       balancing_flux = ( us%kg_m2s_to_RZ_T*global_area_integral(iss%water_flux, g, scale=us%RZ_T_to_kg_m2s,
      &
01055                                                                 area=iss%area_shelf_h) + &
01056                          delta_mass_shelf ) / balancing_area
01057     else
01058       balancing_flux = 0.0
01059     endif
01060
01061     ! apply fluxes
01062     do j=js,je ; do i=is,ie
01063       if (bal_frac(i,j) > 0.0) then
01064         ! evap is negative, and vprec has units of [R Z T-1 ~> kg m-2 s-1]
01065         fluxes%vprec(i,j) = -balancing_flux
01066         fluxes%sens(i,j) = fluxes%vprec(i,j) * cs%Cp * cs%T0 ! [ Q R Z T-1 ~> W /m^2 ]
01067         fluxes%salt_flux(i,j) = fluxes%vprec(i,j) * cs%S0*1.0e-3 ! [kgSalt/kg R Z T-1 ~> kgSalt m-2 s-1]
01068       endif
01069     enddo ; enddo
01070
01071     if (cs%debug) then
01072       write(mesg,*) 'Balancing flux (kg/(m^2 s)), dt = ', balancing_flux*us%RZ_T_to_kg_m2s, cs%time_step
01073       call mom_mesg(mesg)
01074       call mom_forcing_chksum("After constant sea level", fluxes, g, cs%US, haloshift=0)
01075     endif
01076
01077   endif ! constant_sea_level
01078
```

42.4.3.2 add_shelf_forces()

```
subroutine, public mom_ice_shelf::add_shelf_forces (
```

```
type(ocean_grid_type), intent(inout) G,
type(unit_scale_type), intent(in) US,
type(ice_shelf_cs), pointer CS,
type(mech_forcing), intent(inout) forces,
logical, intent(in), optional do_shelf_area )
```

This subroutine adds the mechanical forcing fields and perhaps shelf areas, based on the ice state in ice_shelf_CS.

Parameters

in,out	*g*	The ocean's grid structure.
in	*us*	A dimensional unit scaling type
	cs	This module's control structure.
in,out	*forces*	A structure with the driving mechanical forces
in	*do_shelf_area*	If true find the shelf-covered areas.

Definition at line 776 of file MOM_ice_shelf.F90.

Referenced by initialize_ice_shelf(), and shelf_calc_flux().

```
00776   type(ocean_grid_type), intent(inout) :: g    !< The ocean's grid structure.
00777   type(unit_scale_type), intent(in)    :: us   !< A dimensional unit scaling type
00778   type(ice_shelf_cs),    pointer       :: cs   !< This module's control structure.
00779   type(mech_forcing),    intent(inout) :: forces !< A structure with the driving mechanical forces
00780   logical, optional,     intent(in)    :: do_shelf_area !< If true find the shelf-covered areas.
00781
00782   real :: kv_rho_ice ! The viscosity of ice divided by its density [L4 T-1 R-1 Z-2 ~> m5 kg-1 s-1].
00783   real :: press_ice  ! The pressure of the ice shelf per unit area of ocean (not ice) [R L2 T-2 ~> Pa].
00784   logical :: find_area ! If true find the shelf areas at u & v points.
00785   type(ice_shelf_state), pointer :: iss => null() ! A structure with elements that describe
00786                                           ! the ice-shelf state
00787
00788   integer :: i, j, is, ie, js, je, isd, ied, jsd, jed
00789   is = g%isc ; ie = g%iec ; js = g%jsc ; je = g%jec
00790   isd = g%isd ; jsd = g%jsd ; ied = g%ied ; jed = g%jed
00791
00792   if ((cs%grid%isc /= g%isc) .or. (cs%grid%iec /= g%iec) .or. &
00793       (cs%grid%jsc /= g%jsc) .or. (cs%grid%jec /= g%jec)) &
00794     call mom_error(fatal,"add_shelf_forces: Incompatible ocean and ice shelf grids.")
00795
00796   iss => cs%ISS
00797
00798   find_area = .true. ; if (present(do_shelf_area)) find_area = do_shelf_area
00799
00800   if (find_area) then
00801     ! The frac_shelf is set over the widest possible area. Could it be smaller?
00802     do j=jsd,jed ; do i=isd,ied-1
00803       forces%frac_shelf_u(i,j) = 0.0
00804       if ((g%areaT(i,j) + g%areaT(i+1,j) > 0.0)) & ! .and. (G%areaCu(I,j) > 0.0)) &
00805         forces%frac_shelf_u(i,j) = (iss%area_shelf_h(i,j) + iss%area_shelf_h(i+1,j)) / &
00806                                    (g%areaT(i,j) + g%areaT(i+1,j))
00807     enddo ; enddo
00808     do j=jsd,jed-1 ; do i=isd,ied
00809       forces%frac_shelf_v(i,j) = 0.0
00810       if ((g%areaT(i,j) + g%areaT(i,j+1) > 0.0)) & ! .and. (G%areaCv(i,J) > 0.0)) &
00811         forces%frac_shelf_v(i,j) = (iss%area_shelf_h(i,j) + iss%area_shelf_h(i,j+1)) / &
00812                                    (g%areaT(i,j) + g%areaT(i,j+1))
00813     enddo ; enddo
00814     call pass_vector(forces%frac_shelf_u, forces%frac_shelf_v, g%domain, to_all, cgrid_ne)
00815   endif
00816
00817   do j=js,je ; do i=is,ie
00818     press_ice = (iss%area_shelf_h(i,j) * g%IareaT(i,j)) * (cs%g_Earth * iss%mass_shelf(i,j))
00819     if (associated(forces%p_surf)) then
00820       if (.not.forces%accumulate_p_surf) forces%p_surf(i,j) = 0.0
00821       forces%p_surf(i,j) = forces%p_surf(i,j) + press_ice
00822     endif
00823     if (associated(forces%p_surf_full)) then
00824       if (.not.forces%accumulate_p_surf) forces%p_surf_full(i,j) = 0.0
00825       forces%p_surf_full(i,j) = forces%p_surf_full(i,j) + press_ice
00826     endif
00827   enddo ; enddo
00828
00829   ! For various reasons, forces%rigidity_ice_[uv] is always updated here. Note
```

```
00830   ! that it may have been zeroed out where IOB is translated to forces and
00831   ! contributions from icebergs and the sea-ice pack added subsequently.
00832   !### THE RIGIDITY SHOULD ALSO INCORPORATE AREAL-COVERAGE INFORMATION.
00833   kv_rho_ice = cs%kv_ice / cs%density_ice
00834   do j=js,je ; do i=is-1,ie
00835     if (.not.forces%accumulate_rigidity) forces%rigidity_ice_u(i,j) = 0.0
00836     forces%rigidity_ice_u(i,j) = forces%rigidity_ice_u(i,j) + &
00837             kv_rho_ice * min(iss%mass_shelf(i,j), iss%mass_shelf(i+1,j))
00838   enddo ; enddo
00839   do j=js-1,je ; do i=is,ie
00840     if (.not.forces%accumulate_rigidity) forces%rigidity_ice_v(i,j) = 0.0
00841     forces%rigidity_ice_v(i,j) = forces%rigidity_ice_v(i,j) + &
00842             kv_rho_ice * min(iss%mass_shelf(i,j), iss%mass_shelf(i,j+1))
00843   enddo ; enddo
00844
00845   if (cs%debug) then
00846     call uvchksum("rigidity_ice_[uv]", forces%rigidity_ice_u, forces%rigidity_ice_v, &
00847                   g%HI, symmetric=.true., scale=us%L_to_m**3*us%L_to_Z*us%s_to_T)
00848     call uvchksum("frac_shelf_[uv]", forces%frac_shelf_u, forces%frac_shelf_v, &
00849                   g%HI, symmetric=.true.)
00850   endif
00851
```

42.4.3.3 add_shelf_pressure()

```
subroutine mom_ice_shelf::add_shelf_pressure (
             type(ocean_grid_type), intent(inout) G,
             type(unit_scale_type), intent(in) US,
             type(ice_shelf_cs), intent(in) CS,
             type(forcing), intent(inout) fluxes )  [private]
```

This subroutine adds the ice shelf pressure to the fluxes type.

Parameters

in,out	*g*	The ocean's grid structure.
in	*us*	A dimensional unit scaling type
in	*cs*	This module's control structure.
in,out	*fluxes*	A structure of surface fluxes that may be updated.

Definition at line 856 of file MOM_ice_shelf.F90.

Referenced by add_shelf_flux(), and initialize_ice_shelf().

```
00856   type(ocean_grid_type), intent(inout) :: g    !< The ocean's grid structure.
00857   type(unit_scale_type), intent(in)    :: us   !< A dimensional unit scaling type
00858   type(ice_shelf_cs),    intent(in)    :: cs   !< This module's control structure.
00859   type(forcing),         intent(inout) :: fluxes  !< A structure of surface fluxes that may be updated.
00860
00861   real :: press_ice       !< The pressure of the ice shelf per unit area of ocean (not ice) [R L2 T-2 ~>
      Pa].
00862   integer :: i, j, is, ie, js, je, isd, ied, jsd, jed
00863   is = g%isc ; ie = g%iec ; js = g%jsc ; je = g%jec
00864
00865   if ((cs%grid%isc /= g%isc) .or. (cs%grid%iec /= g%iec) .or. &
00866       (cs%grid%jsc /= g%jsc) .or. (cs%grid%jec /= g%jec)) &
00867     call mom_error(fatal,"add_shelf_pressure: Incompatible ocean and ice shelf grids.")
00868
00869   do j=js,je ; do i=is,ie
00870     press_ice = (cs%ISS%area_shelf_h(i,j) * g%IareaT(i,j)) * (cs%g_Earth * cs%ISS%mass_shelf(i,j))
00871     if (associated(fluxes%p_surf)) then
00872       if (.not.fluxes%accumulate_p_surf) fluxes%p_surf(i,j) = 0.0
00873       fluxes%p_surf(i,j) = fluxes%p_surf(i,j) + press_ice
00874     endif
00875     if (associated(fluxes%p_surf_full)) then
```

```
00876       if (.not.fluxes%accumulate_p_surf) fluxes%p_surf_full(i,j) = 0.0
00877       fluxes%p_surf_full(i,j) = fluxes%p_surf_full(i,j) + press_ice
00878     endif
00879   enddo ; enddo
00880
```

42.4.3.4 change_thickness_using_melt()

```
subroutine mom_ice_shelf::change_thickness_using_melt (
             type(ice_shelf_state), intent(inout) ISS,
             type(ocean_grid_type), intent(inout) G,
             type(unit_scale_type), intent(in) US,
             real, intent(in) time_step,
             type(forcing), intent(inout) fluxes,
             real, intent(in) density_ice,
             logical, intent(in), optional debug )  [private]
```

Changes the thickness (mass) of the ice shelf based on sub-ice-shelf melting.

Parameters

in,out	*g*	The ocean's grid structure.
in,out	*iss*	A structure with elements that describe the ice-shelf state
in	*us*	A dimensional unit scaling type
in	*time_step*	The time step for this update [T ~> s].
in,out	*fluxes*	structure containing pointers to any possible thermodynamic or mass-flux forcing fields.
in	*density_ice*	The density of ice-shelf ice [R ~> kg m-3].
in	*debug*	If present and true, write chksums

Definition at line 728 of file MOM_ice_shelf.F90.

Referenced by shelf_calc_flux().

```
00728   type(ocean_grid_type), intent(inout) :: g  !< The ocean's grid structure.
00729   type(ice_shelf_state), intent(inout) :: iss !< A structure with elements that describe
00730                                                !! the ice-shelf state
00731   type(unit_scale_type), intent(in)    :: us   !< A dimensional unit scaling type
00732   real,                  intent(in)    :: time_step !< The time step for this update [T ~> s].
00733   type(forcing),         intent(inout) :: fluxes !< structure containing pointers to any possible
00734                                                  !! thermodynamic or mass-flux forcing fields.
00735   real,                  intent(in)    :: density_ice !< The density of ice-shelf ice [R ~> kg m-3].
00736   logical,     optional, intent(in)    :: debug !< If present and true, write chksums
00737
00738   ! locals
00739   real :: i_rho_ice ! Ice specific volume [R-1 ~> m3 kg-1]
00740   integer :: i, j
00741
00742   i_rho_ice = 1.0 / density_ice
00743
00744
00745   do j=g%jsc,g%jec ; do i=g%isc,g%iec
00746     if ((iss%hmask(i,j) == 1) .or. (iss%hmask(i,j) == 2)) then
00747       ! first, zero out fluxes applied during previous time step
00748       if (associated(fluxes%lprec)) fluxes%lprec(i,j) = 0.0
00749       if (associated(fluxes%sens)) fluxes%sens(i,j) = 0.0
00750       if (associated(fluxes%frac_shelf_h)) fluxes%frac_shelf_h(i,j) = 0.0
00751       if (associated(fluxes%salt_flux)) fluxes%salt_flux(i,j) = 0.0
00752
00753       if (iss%water_flux(i,j) * time_step / density_ice < iss%h_shelf(i,j)) then
00754         iss%h_shelf(i,j) = iss%h_shelf(i,j) - iss%water_flux(i,j) * time_step / density_ice
00755       else
```

```
00756         ! the ice is about to melt away, so set thickness, area, and mask to zero
00757         ! NOTE: this is not mass conservative should maybe scale salt & heat flux for this cell
00758         iss%h_shelf(i,j) = 0.0
00759         iss%hmask(i,j) = 0.0
00760         iss%area_shelf_h(i,j) = 0.0
00761       endif
00762       iss%mass_shelf(i,j) = iss%h_shelf(i,j) * density_ice
00763     endif
00764   enddo ; enddo
00765
00766   call pass_var(iss%area_shelf_h, g%domain)
00767   call pass_var(iss%h_shelf, g%domain)
00768   call pass_var(iss%hmask, g%domain)
00769   call pass_var(iss%mass_shelf, g%domain)
00770
```

42.4.3.5 ice_shelf_end()

```
subroutine, public mom_ice_shelf::ice_shelf_end (
              type(ice_shelf_cs), pointer CS )
```

Deallocates all memory associated with this module.

Parameters

cs	A pointer to the ice shelf control structure

Definition at line 1790 of file MOM_ice_shelf.F90.

```
01790   type(ice_shelf_cs), pointer   :: cs !< A pointer to the ice shelf control structure
01791
01792   if (.not.associated(cs)) return
01793
01794   call ice_shelf_state_end(cs%ISS)
01795
01796   if (cs%active_shelf_dynamics) call ice_shelf_dyn_end(cs%dCS)
01797
01798   deallocate(cs)
01799
```

42.4.3.6 ice_shelf_save_restart()

```
subroutine, public mom_ice_shelf::ice_shelf_save_restart (
              type(ice_shelf_cs), pointer CS,
              type(time_type), intent(in) Time,
              character(len=*), intent(in), optional directory,
              logical, intent(in), optional time_stamped,
              character(len=*), intent(in), optional filename_suffix )
```

Save the ice shelf restart file.

Parameters

	cs	ice shelf control structure
in	*time*	model time at this call
in	*directory*	An optional directory into which to write these restart files.
in	*time_stamped*	f true, the restart file names include a unique time stamp. The default is false.
in	*filename_suffix*	An optional suffix (e.g., a time-stamp) to append to the restart file names.

Definition at line 1767 of file MOM_ice_shelf.F90.

```
01767   type(ice_shelf_cs),         pointer    :: cs !< ice shelf control structure
01768   type(time_type),            intent(in) :: time !< model time at this call
01769   character(len=*), optional, intent(in) :: directory !< An optional directory into which to write
01770                                                 !! these restart files.
01771   logical,          optional, intent(in) :: time_stamped !< f true, the restart file names include
01772                                                 !! a unique time stamp.  The default is false.
01773   character(len=*), optional, intent(in) :: filename_suffix !< An optional suffix (e.g., a
01774                                                 !! time-stamp) to append to the restart file names.
01775   ! local variables
01776   type(ocean_grid_type), pointer :: g => null()
01777   character(len=200) :: restart_dir
01778
01779   g => cs%grid
01780
01781   if (present(directory)) then ; restart_dir = directory
01782   else ; restart_dir = cs%restart_output_dir ; endif
01783
01784   call save_restart(restart_dir, time, cs%grid, cs%restart_CSp, time_stamped)
01785
```

42.4.3.7 initialize_ice_shelf()

```
subroutine, public mom_ice_shelf::initialize_ice_shelf (
             type(param_file_type), intent(in) param_file,
             type(ocean_grid_type), pointer ocn_grid,
             type(time_type), intent(inout) Time,
             type(ice_shelf_cs), pointer CS,
             type(diag_ctrl), intent(in), target diag,
             type(mech_forcing), intent(inout), optional forces,
             type(forcing), intent(inout), optional fluxes,
             type(time_type), intent(in), optional Time_in,
             logical, intent(in), optional solo_ice_sheet_in )
```

Initializes shelf model data, parameters and diagnostics.

Parameters

in	*param_file*	A structure to parse for run-time parameters
	ocn_grid	The calling ocean model's horizontal grid structure
in,out	*time*	The clock that that will indicate the model time
	cs	A pointer to the ice shelf control structure
in	*diag*	A structure that is used to regulate the diagnostic output.
in,out	*fluxes*	A structure containing pointers to any possible thermodynamic or mass-flux forcing fields.
in,out	*forces*	A structure with the driving mechanical forces
in	*time_in*	The time at initialization.
in	*solo_ice_sheet_in*	If present, this indicates whether a solo ice-sheet driver.

Definition at line 1084 of file MOM_ice_shelf.F90.

References add_shelf_forces(), add_shelf_pressure(), mom_eos::eos_init(), mom_unit_scaling::fix_restart_unit_scaling(), id_clock_pass, id_clock_shelf, initialize_shelf_mass(), and mom_unit_scaling::unit_scaling_init().

```
01084   type(param_file_type),        intent(in)    :: param_file !< A structure to parse for run-time parameters
01085   type(ocean_grid_type),        pointer       :: ocn_grid   !< The calling ocean model's horizontal grid
      structure
01086   type(time_type),              intent(inout) :: time !< The clock that that will indicate the model time
01087   type(ice_shelf_cs),           pointer       :: cs   !< A pointer to the ice shelf control structure
01088   type(diag_ctrl),    target,   intent(in)    :: diag !< A structure that is used to regulate the
      diagnostic output.
01089   type(forcing),      optional, intent(inout) :: fluxes !< A structure containing pointers to any possible
01090                                                    !! thermodynamic or mass-flux forcing fields.
01091   type(mech_forcing), optional, intent(inout) :: forces !< A structure with the driving mechanical forces
01092   type(time_type),    optional, intent(in)    :: time_in !< The time at initialization.
01093   logical,            optional, intent(in)    :: solo_ice_sheet_in !< If present, this indicates whether
01094                                                    !! a solo ice-sheet driver.
01095
01096   type(ocean_grid_type), pointer :: g  => null(), og  => null() ! Pointers to grids for convenience.
01097   type(unit_scale_type), pointer :: us => null() ! Pointer to a structure containing
01098                                                  ! various unit conversion factors
01099   type(ice_shelf_state), pointer :: iss => null() !< A structure with elements that describe
01100                                          !! the ice-shelf state
01101   type(directories)  :: dirs
01102   type(dyn_horgrid_type), pointer :: dg => null()
01103   real    :: z_rescale  ! A rescaling factor for heights from the representation in
01104                         ! a restart file to the internal representation in this run.
01105   real    :: rz_rescale ! A rescaling factor for mass loads from the representation in
01106                         ! a restart file to the internal representation in this run.
01107   real    :: l_rescale  ! A rescaling factor for horizontal lengths from the representation in
01108                         ! a restart file to the internal representation in this run.
01109   real :: meltrate_conversion ! The conversion factor to use for in the melt rate diagnostic.
01110   real :: dz_ocean_min_float ! The minimum ocean thickness above which the ice shelf is considered
01111                         ! to be floating when CONST_SEA_LEVEL = True [Z ~> m].
01112   real :: cdrag, drag_bg_vel
01113   logical :: new_sim, save_ic, var_force
01114   !This include declares and sets the variable "version".
01115 # include "version_variable.h"
01116   character(len=200) :: config
01117   character(len=200) :: ic_file,filename,inputdir
01118   character(len=40)  :: mdl = "MOM_ice_shelf"  ! This module's name.
01119   integer :: i, j, is, ie, js, je, isd, ied, jsd, jed, isdq, iedq, jsdq, jedq
01120   integer :: wd_halos(2)
01121   logical :: read_tideamp, shelf_mass_is_dynamic, debug
01122   character(len=240) :: tideamp_file
01123   real    :: utide  ! A tidal velocity [L T-1 ~> m s-1]
01124   real    :: col_thick_melt_thresh ! An ocean column thickness below which iceshelf melting
01125                       ! does not occur [Z ~> m]
01126   if (associated(cs)) then
01127     call mom_error(fatal, "MOM_ice_shelf.F90, initialize_ice_shelf: "// &
01128                           "called with an associated control structure.")
01129     return
01130   endif
01131   allocate(cs)
01132
01133   !   Go through all of the infrastructure initialization calls, since this is
01134   ! being treated as an independent component that just happens to use the
01135   ! MOM's grid and infrastructure.
01136   call get_mom_input(dirs=dirs)
01137
01138   ! Determining the internal unit scaling factors for this run.
01139   call unit_scaling_init(param_file, cs%US)
01140
01141   ! Set up the ice-shelf domain and grid
01142   wd_halos(:)=0
01143   call mom_domains_init(cs%grid%domain, param_file, min_halo=wd_halos, symmetric=grid_sym_)
01144   ! call diag_mediator_init(CS%grid,param_file,CS%diag)
01145   ! this needs to be fixed - will probably break when not using coupled driver 0
01146   call mom_grid_init(cs%grid, param_file, cs%US)
01147
01148   call create_dyn_horgrid(dg, cs%grid%HI)
01149   call clone_mom_domain(cs%grid%Domain, dg%Domain)
01150
01151   call set_grid_metrics(dg, param_file, cs%US)
01152   ! call set_diag_mediator_grid(CS%grid, CS%diag)
01153
01154   ! The ocean grid possibly uses different symmetry.
01155   if (associated(ocn_grid)) then ; cs%ocn_grid => ocn_grid
01156   else ; cs%ocn_grid => cs%grid ; endif
01157
01158   ! Convenience pointers
01159   g => cs%grid
01160   og => cs%ocn_grid
01161   us => cs%US
01162
01163   if (is_root_pe()) then
01164     write(0,*) 'OG: ', og%isd, og%isc, og%iec, og%ied, og%jsd, og%jsc, og%jsd, og%jed
01165     write(0,*) 'IG: ', g%isd, g%isc, g%iec, g%ied, g%jsd, g%jsc, g%jsd, g%jed
01166   endif
01167
01168   cs%diag => diag
```

```
01169
01170   ! Are we being called from the solo ice-sheet driver? When called by the ocean
01171   ! model solo_ice_sheet_in is not preset.
01172   cs%solo_ice_sheet = .false.
01173   if (present(solo_ice_sheet_in)) cs%solo_ice_sheet = solo_ice_sheet_in
01174
01175   if (present(time_in)) time = time_in
01176
01177   is = g%isc ; ie = g%iec ; js = g%jsc ; je = g%jec
01178   isd = g%isd ; jsd = g%jsd ; ied = g%ied ; jed = g%jed
01179   isdq = g%IsdB ; iedq = g%IedB ; jsdq = g%JsdB ; jedq = g%JedB
01180
01181   cs%override_shelf_movement = .false. ; cs%active_shelf_dynamics = .false.
01182
01183   call log_version(param_file, mdl, version, "")
01184   call get_param(param_file, mdl, "DEBUG", debug, default=.false.)
01185   call get_param(param_file, mdl, "DEBUG_IS", cs%debug, &
01186                  "If true, write verbose debugging messages for the ice shelf.", &
01187                  default=debug)
01188   call get_param(param_file, mdl, "DYNAMIC_SHELF_MASS", shelf_mass_is_dynamic, &
01189                  "If true, the ice sheet mass can evolve with time.", &
01190                  default=.false.)
01191   if (shelf_mass_is_dynamic) then
01192     call get_param(param_file, mdl, "OVERRIDE_SHELF_MOVEMENT", cs%override_shelf_movement, &
01193                  "If true, user provided code specifies the ice-shelf "//&
01194                  "movement instead of the dynamic ice model.", default=.false.)
01195     cs%active_shelf_dynamics = .not.cs%override_shelf_movement
01196     call get_param(param_file, mdl, "GROUNDING_LINE_INTERPOLATE", cs%GL_regularize, &
01197                  "If true, regularize the floatation condition at the "//&
01198                  "grounding line as in Goldberg Holland Schoof 2009.", default=.false.)
01199     call get_param(param_file, mdl, "GROUNDING_LINE_COUPLE", cs%GL_couple, &
01200                  "If true, let the floatation condition be determined by "//&
01201                  "ocean column thickness. This means that update_OD_ffrac "//&
01202                  "will be called.  GL_REGULARIZE and GL_COUPLE are exclusive.", &
01203                  default=.false., do_not_log=cs%GL_regularize)
01204     if (cs%GL_regularize) cs%GL_couple = .false.
01205   endif
01206
01207   call get_param(param_file, mdl, "SHELF_THERMO", cs%isthermo, &
01208                  "If true, use a thermodynamically interactive ice shelf.", &
01209                  default=.false.)
01210   call get_param(param_file, mdl, "LATENT_HEAT_FUSION", cs%Lat_fusion, &
01211                  "The latent heat of fusion.", units="J/kg", default=hlf, scale=us%J_kg_to_Q)
01212   call get_param(param_file, mdl, "SHELF_THREE_EQN", cs%threeeq, &
01213                  "If true, use the three equation expression of "//&
01214                  "consistency to calculate the fluxes at the ice-ocean "//&
01215                  "interface.", default=.true.)
01216   call get_param(param_file, mdl, "SHELF_INSULATOR", cs%insulator, &
01217                  "If true, the ice shelf is a perfect insulatior "//&
01218                  "(no conduction).", default=.false.)
01219   call get_param(param_file, mdl, "MELTING_CUTOFF_DEPTH", cs%cutoff_depth, &
01220                  "Depth above which the melt is set to zero (it must be >= 0) "//&
01221                  "Default value won't affect the solution.", units="m", default=0.0, scale=us%m_to_Z)
01222   if (cs%cutoff_depth < 0.) &
01223     call mom_error(warning,"Initialize_ice_shelf: MELTING_CUTOFF_DEPTH must be >= 0.")
01224
01225   call get_param(param_file, mdl, "CONST_SEA_LEVEL", cs%constant_sea_level, &
01226                  "If true, apply evaporative, heat and salt fluxes in "//&
01227                  "the sponge region. This will avoid a large increase "//&
01228                  "in sea level. This option is needed for some of the "//&
01229                  "ISOMIP+ experiments (Ocean3 and Ocean4). "//&
01230                  "IMPORTANT: it is not currently possible to do "//&
01231                  "prefect restarts using this flag.", default=.false.)
01232   call get_param(param_file, mdl, "MIN_OCEAN_FLOAT_THICK", dz_ocean_min_float, &
01233                  "The minimum ocean thickness above which the ice shelf is considered to be "//&
01234                  "floating when CONST_SEA_LEVEL = True.", &
01235                  default=0.1, units="m", scale=us%m_to_Z, do_not_log=.not.cs%constant_sea_level)
01236
01237   call get_param(param_file, mdl, "ISOMIP_S_SUR_SPONGE", cs%S0, &
01238                  "Surface salinity in the restoring region.", &
01239                 default=33.8, units='ppt', do_not_log=.true.)
01240
01241   call get_param(param_file, mdl, "ISOMIP_T_SUR_SPONGE", cs%T0, &
01242                 "Surface temperature in the restoring region.", &
01243                 default=-1.9, units='degC', do_not_log=.true.)
01244
01245   call get_param(param_file, mdl, "SHELF_3EQ_GAMMA", cs%const_gamma, &
01246                  "If true, user specifies a constant nondimensional heat-transfer coefficient "//&
01247                  "(GAMMA_T_3EQ), from which the default salt-transfer coefficient is set "//&
01248                  "as GAMMA_T_3EQ/35. This is used with SHELF_THREE_EQN.", default=.false.)
01249   if (cs%threeeq) then
01250     call get_param(param_file, mdl, "SHELF_S_ROOT", cs%find_salt_root, &
01251                  "If SHELF_S_ROOT = True, salinity at the ice/ocean interface (Sbdry) "//&
01252                  "is computed from a quadratic equation. Otherwise, the previous "//&
01253                  "interactive method to estimate Sbdry is used.", default=.false.)
01254   else
01255     call get_param(param_file, mdl, "SHELF_2EQ_GAMMA_T", cs%gamma_t, &
```

```
01256                 "If SHELF_THREE_EQN is false, this the fixed turbulent "//&
01257                 "exchange velocity at the ice-ocean interface.", &
01258                 units="m s-1", scale=us%m_to_Z*us%T_to_s, fail_if_missing=.true.)
01259   endif
01260   if (cs%const_gamma .or. cs%find_salt_root) then
01261     call get_param(param_file, mdl, "SHELF_3EQ_GAMMA_T", cs%Gamma_T_3EQ, &
01262                 "Nondimensional heat-transfer coefficient.", &
01263                  units="nondim", default=2.2e-2)
01264     call get_param(param_file, mdl, "SHELF_3EQ_GAMMA_S", cs%Gamma_S_3EQ, &
01265                 "Nondimensional salt-transfer coefficient.", &
01266                 default=cs%Gamma_T_3EQ/35.0, units="nondim")
01267   endif
01268
01269   call get_param(param_file, mdl, "ICE_SHELF_MASS_FROM_FILE", &
01270                 cs%mass_from_file, "Read the mass of the "//&
01271                 "ice shelf (every time step) from a file.", default=.false.)
01272
01273   if (cs%find_salt_root) then ! read liquidus coeffs.
01274      call get_param(param_file, mdl, "TFREEZE_S0_P0", cs%TFr_0_0, &
01275                 "this is the freezing potential temperature at "//&
01276                 "S=0, P=0.", units="degC", default=0.0, do_not_log=.true.)
01277     call get_param(param_file, mdl, "DTFREEZE_DS", cs%dTFr_dS, &
01278                 "this is the derivative of the freezing potential temperature with salinity.", &
01279                 units="degC psu-1", default=-0.054, do_not_log=.true.)
01280     call get_param(param_file, mdl, "DTFREEZE_DP", cs%dTFr_dp, &
01281                 "this is the derivative of the freezing potential temperature with pressure.", &
01282                 units="degC Pa-1", default=0.0, scale=us%RL2_T2_to_Pa, do_not_log=.true.)
01283   endif
01284
01285   call get_param(param_file, mdl, "G_EARTH", cs%g_Earth, &
01286                 "The gravitational acceleration of the Earth.", &
01287                 units="m s-2", default = 9.80, scale=us%m_s_to_L_T**2*us%Z_to_m)
01288   call get_param(param_file, mdl, "C_P", cs%Cp, &
01289                 "The heat capacity of sea water, approximated as a constant. "//&
01290                 "The default value is from the TEOS-10 definition of conservative temperature.", &
01291                 units="J kg-1 K-1", default=3991.86795711963, scale=us%J_kg_to_Q)
01292   call get_param(param_file, mdl, "RHO_0", cs%Rho_ocn, &
01293                 "The mean ocean density used with BOUSSINESQ true to "//&
01294                 "calculate accelerations and the mass for conservation "//&
01295                 "properties, or with BOUSSINSEQ false to convert some "//&
01296                 "parameters from vertical units of m to kg m-2.", &
01297                 units="kg m-3", default=1035.0, scale=us%kg_m3_to_R)
01298   call get_param(param_file, mdl, "C_P_ICE", cs%Cp_ice, &
01299                 "The heat capacity of ice.", units="J kg-1 K-1", scale=us%J_kg_to_Q, &
01300                 default=2.10e3)
01301   if (cs%constant_sea_level) cs%min_ocean_mass_float = dz_ocean_min_float*cs%Rho_ocn
01302
01303   call get_param(param_file, mdl, "ICE_SHELF_FLUX_FACTOR", cs%flux_factor, &
01304                 "Non-dimensional factor applied to shelf thermodynamic "//&
01305                 "fluxes.", units="none", default=1.0)
01306
01307   call get_param(param_file, mdl, "KV_ICE", cs%kv_ice, &
01308                 "The viscosity of the ice.", &
01309                 units="m2 s-1", default=1.0e10, scale=us%Z_to_L**2*us%m_to_L**2*us%T_to_s)
01310   call get_param(param_file, mdl, "KV_MOLECULAR", cs%kv_molec, &
01311                 "The molecular kinimatic viscosity of sea water at the "//&
01312                 "freezing temperature.", units="m2 s-1", default=1.95e-6, scale=us%m2_s_to_Z2_T)
01313   call get_param(param_file, mdl, "ICE_SHELF_SALINITY", cs%Salin_ice, &
01314                 "The salinity of the ice inside the ice shelf.", units="psu", &
01315                 default=0.0)
01316   call get_param(param_file, mdl, "ICE_SHELF_TEMPERATURE", cs%Temp_ice, &
01317                 "The temperature at the center of the ice shelf.", &
01318                 units = "degC", default=-15.0)
01319   call get_param(param_file, mdl, "KD_SALT_MOLECULAR", cs%kd_molec_salt, &
01320                 "The molecular diffusivity of salt in sea water at the "//&
01321                 "freezing point.", units="m2 s-1", default=8.02e-10, scale=us%m2_s_to_Z2_T)
01322   call get_param(param_file, mdl, "KD_TEMP_MOLECULAR", cs%kd_molec_temp, &
01323                 "The molecular diffusivity of heat in sea water at the "//&
01324                 "freezing point.", units="m2 s-1", default=1.41e-7, scale=us%m2_s_to_Z2_T)
01325   call get_param(param_file, mdl, "DT_FORCING", cs%time_step, &
01326                 "The time step for changing forcing, coupling with other "//&
01327                 "components, or potentially writing certain diagnostics. "//&
01328                 "The default value is given by DT.", units="s", default=0.0)
01329
01330   call get_param(param_file, mdl, "COL_THICK_MELT_THRESHOLD", col_thick_melt_thresh, &
01331                 "The minimum ocean column thickness where melting is allowed.", &
01332                 units="m", scale=us%m_to_Z, default=0.0)
01333   cs%col_mass_melt_threshold =  cs%Rho_ocn * col_thick_melt_thresh
01334
01335   call get_param(param_file, mdl, "READ_TIDEAMP", read_tideamp, &
01336                 "If true, read a file (given by TIDEAMP_FILE) containing "//&
01337                 "the tidal amplitude with INT_TIDE_DISSIPATION.", default=.false.)
01338
01339   call safe_alloc_ptr(cs%utide,isd,ied,jsd,jed)   ; cs%utide(:,:) = 0.0
01340
01341   if (read_tideamp) then
01342     call get_param(param_file, mdl, "TIDEAMP_FILE", tideamp_file, &
```

```
01343                 "The path to the file containing the spatially varying "//&
01344                 "tidal amplitudes.", &
01345                 default="tideamp.nc")
01346   call get_param(param_file, mdl, "INPUTDIR", inputdir, default=".")
01347   inputdir = slasher(inputdir)
01348   tideamp_file = trim(inputdir) // trim(tideamp_file)
01349   call mom_read_data(tideamp_file, 'tideamp', cs%utide, g%domain, timelevel=1, scale=us%m_s_to_L_T)
01350 else
01351   call get_param(param_file, mdl, "UTIDE", utide, &
01352                 "The constant tidal amplitude used with INT_TIDE_DISSIPATION.", &
01353                 units="m s-1", default=0.0 , scale=us%m_s_to_L_T)
01354   cs%utide(:,:) = utide
01355 endif
01356
01357 call eos_init(param_file, cs%eqn_of_state)
01358
01359 !! new parameters that need to be in MOM_input
01360
01361 if (cs%active_shelf_dynamics) then
01362
01363   call get_param(param_file, mdl, "DENSITY_ICE", cs%density_ice, &
01364                 "A typical density of ice.", units="kg m-3", default=917.0, scale=us%kg_m3_to_R)
01365
01366   call get_param(param_file, mdl, "INPUT_FLUX_ICE_SHELF", cs%input_flux, &
01367                 "volume flux at upstream boundary", units="m2 s-1", default=0.)
01368   call get_param(param_file, mdl, "INPUT_THICK_ICE_SHELF", cs%input_thickness, &
01369                 "flux thickness at upstream boundary", units="m", default=1000.)
01370 else
01371   ! This is here because of inconsistent defaults.  I don't know why.  RWH
01372   call get_param(param_file, mdl, "DENSITY_ICE", cs%density_ice, &
01373                 "A typical density of ice.", units="kg m-3", default=900.0, scale=us%kg_m3_to_R)
01374 endif
01375 call get_param(param_file, mdl, "MIN_THICKNESS_SIMPLE_CALVE", &
01376               cs%min_thickness_simple_calve, &
01377                "Min thickness rule for the very simple calving law",&
01378                units="m", default=0.0, scale=us%m_to_Z)
01379
01380 call get_param(param_file, mdl, "USTAR_SHELF_BG", cs%ustar_bg, &
01381                 "The minimum value of ustar under ice shelves.", &
01382                 units="m s-1", default=0.0, scale=us%m_to_Z*us%T_to_s)
01383 call get_param(param_file, mdl, "CDRAG_SHELF", cdrag, &
01384      "CDRAG is the drag coefficient relating the magnitude of "//&
01385      "the velocity field to the surface stress.", units="nondim", &
01386      default=0.003)
01387 cs%cdrag = cdrag
01388 if (cs%ustar_bg <= 0.0) then
01389   call get_param(param_file, mdl, "DRAG_BG_VEL_SHELF", drag_bg_vel, &
01390                 "DRAG_BG_VEL is either the assumed bottom velocity (with "//&
01391                 "LINEAR_DRAG) or an unresolved  velocity that is "//&
01392                 "combined with the resolved velocity to estimate the "//&
01393                 "velocity magnitude.", units="m s-1", default=0.0, scale=us%m_to_Z*us%T_to_s)
01394   if (cs%cdrag*drag_bg_vel > 0.0) cs%ustar_bg = sqrt(cs%cdrag)*drag_bg_vel
01395 endif
01396
01397 ! Allocate and initialize state variables to default values
01398 call ice_shelf_state_init(cs%ISS, cs%grid)
01399 iss => cs%ISS
01400
01401 ! Allocate the arrays for passing ice-shelf data through the forcing type.
01402 if (.not. cs%solo_ice_sheet) then
01403   call mom_mesg("MOM_ice_shelf.F90, initialize_ice_shelf: allocating fluxes.")
01404    ! GMM: the following assures that water/heat fluxes are just allocated
01405    ! when SHELF_THERMO = True. These fluxes are necessary if one wants to
01406    ! use either ENERGETICS_SFC_PBL (ALE mode) or BULKMIXEDLAYER (layer mode).
01407   if (present(fluxes)) &
01408     call allocate_forcing_type(cs%ocn_grid, fluxes, ustar=.true., shelf=.true., &
01409                                press=.true., water=cs%isthermo, heat=cs%isthermo)
01410   if (present(forces)) &
01411     call allocate_mech_forcing(cs%ocn_grid, forces, ustar=.true., shelf=.true., press=.true.)
01412 else
01413   call mom_mesg("MOM_ice_shelf.F90, initialize_ice_shelf: allocating fluxes in solo mode.")
01414   if (present(fluxes)) &
01415     call allocate_forcing_type(g, fluxes, ustar=.true., shelf=.true., press=.true.)
01416   if (present(forces)) &
01417     call allocate_mech_forcing(g, forces, ustar=.true., shelf=.true., press=.true.)
01418 endif
01419
01420 ! Set up the bottom depth, G%D either analytically or from file
01421 call mom_initialize_topography(dg%bathyT, g%max_depth, dg, param_file)
01422 call rescale_dyn_horgrid_bathymetry(dg, us%Z_to_m)
01423
01424 ! Set up the Coriolis parameter, G%f, usually analytically.
01425 call mom_initialize_rotation(dg%CoriolisBu, dg, param_file, us)
01426 ! This copies grid elements, including bathyT and CoriolisBu from dG to CS%grid.
01427 call copy_dyngrid_to_mom_grid(dg, cs%grid, us)
01428
01429 call destroy_dyn_horgrid(dg)
```

```
01430
01431   ! Set up the restarts.
01432   call restart_init(param_file, cs%restart_CSp, "Shelf.res")
01433   call register_restart_field(iss%mass_shelf, "shelf_mass", .true., cs%restart_CSp, &
01434                               "Ice shelf mass", "kg m-2")
01435   call register_restart_field(iss%area_shelf_h, "shelf_area", .true., cs%restart_CSp, &
01436                               "Ice shelf area in cell", "m2")
01437   call register_restart_field(iss%h_shelf, "h_shelf", .true., cs%restart_CSp, &
01438                               "ice sheet/shelf thickness", "m")
01439   call register_restart_field(us%m_to_Z_restart, "m_to_Z", .false., cs%restart_CSp, &
01440                               "Height unit conversion factor", "Z meter-1")
01441   call register_restart_field(us%m_to_L_restart, "m_to_L", .false., cs%restart_CSp, &
01442                               "Length unit conversion factor", "L meter-1")
01443   call register_restart_field(us%kg_m3_to_R_restart, "kg_m3_to_R", .false., cs%restart_CSp, &
01444                               "Density unit conversion factor", "R m3 kg-1")
01445   if (cs%active_shelf_dynamics) then
01446     call register_restart_field(iss%hmask, "h_mask", .true., cs%restart_CSp, &
01447                                 "ice sheet/shelf thickness mask" ,"none")
01448   endif
01449
01450   if (cs%active_shelf_dynamics) then
01451     ! Allocate CS%dCS and specify additional restarts for ice shelf dynamics
01452     call register_ice_shelf_dyn_restarts(g, param_file, cs%dCS, cs%restart_CSp)
01453   endif
01454
01455   !GMM - I think we do not need to save ustar_shelf and iceshelf_melt in the restart file
01456   !if (.not. CS%solo_ice_sheet) then
01457   !  call register_restart_field(fluxes%ustar_shelf, "ustar_shelf", .false., CS%restart_CSp, &
01458   !                              "Friction velocity under ice shelves", "m s-1")
01459   !endif
01460
01461   cs%restart_output_dir = dirs%restart_output_dir
01462
01463   new_sim = .false.
01464   if ((dirs%input_filename(1:1) == 'n') .and. &
01465       (len_trim(dirs%input_filename) == 1)) new_sim = .true.
01466
01467   if (cs%override_shelf_movement .and. cs%mass_from_file) then
01468
01469     ! initialize the ids for reading shelf mass from a netCDF
01470     call initialize_shelf_mass(g, param_file, cs, iss)
01471
01472     if (new_sim) then
01473       ! new simulation, initialize ice thickness as in the static case
01474       call initialize_ice_thickness(iss%h_shelf, iss%area_shelf_h, iss%hmask, g, us, param_file)
01475
01476     ! next make sure mass is consistent with thickness
01477       do j=g%jsd,g%jed ; do i=g%isd,g%ied
01478         if ((iss%hmask(i,j) == 1) .or. (iss%hmask(i,j) == 2)) then
01479           iss%mass_shelf(i,j) = iss%h_shelf(i,j)*cs%density_ice
01480         endif
01481       enddo ; enddo
01482
01483       if (cs%min_thickness_simple_calve > 0.0) &
01484         call ice_shelf_min_thickness_calve(g, iss%h_shelf, iss%area_shelf_h, iss%hmask, &
01485                                            cs%min_thickness_simple_calve)
01486     endif
01487   endif
01488
01489   if (cs%active_shelf_dynamics) then
01490     ! the only reason to initialize boundary conds is if the shelf is dynamic - MJH
01491
01492     ! call initialize_ice_shelf_boundary ( CS%u_face_mask_bdry, CS%v_face_mask_bdry, &
01493     !                                      CS%u_flux_bdry_val, CS%v_flux_bdry_val, &
01494     !                                      CS%u_bdry_val, CS%v_bdry_val, CS%h_bdry_val, &
01495     !                                      ISS%hmask, G, param_file)
01496
01497   endif
01498
01499   if (new_sim .and. (.not. (cs%override_shelf_movement .and. cs%mass_from_file))) then
01500
01501     ! This model is initialized internally or from a file.
01502     call initialize_ice_thickness(iss%h_shelf, iss%area_shelf_h, iss%hmask, g, us, param_file)
01503
01504     ! next make sure mass is consistent with thickness
01505     do j=g%jsd,g%jed ; do i=g%isd,g%ied
01506       if ((iss%hmask(i,j) == 1) .or. (iss%hmask(i,j) == 2)) then
01507         iss%mass_shelf(i,j) = iss%h_shelf(i,j)*cs%density_ice
01508       endif
01509     enddo ; enddo
01510
01511   ! else ! Previous block for new_sim=.T., this block restores the state.
01512   elseif (.not.new_sim) then
01513     ! This line calls a subroutine that reads the initial conditions from a restart file.
01514     call mom_mesg("MOM_ice_shelf.F90, initialize_ice_shelf: Restoring ice shelf from file.")
01515     call restore_state(dirs%input_filename, dirs%restart_input_dir, time, &
01516                        g, cs%restart_CSp)
```

```
01517
01518     if ((us%m_to_Z_restart /= 0.0) .and. (us%m_to_Z_restart /= us%m_to_Z)) then
01519       z_rescale = us%m_to_Z / us%m_to_Z_restart
01520       do j=g%jsc,g%jec ; do i=g%isc,g%iec
01521         iss%h_shelf(i,j) = z_rescale * iss%h_shelf(i,j)
01522       enddo ; enddo
01523     endif
01524
01525     if ((us%m_to_Z_restart*us%kg_m3_to_R_restart /= 0.0) .and. &
01526         (us%m_to_Z*us%kg_m3_to_R /= us%m_to_Z_restart * us%kg_m3_to_R_restart)) then
01527       rz_rescale = us%m_to_Z*us%kg_m3_to_R / (us%m_to_Z_restart * us%kg_m3_to_R_restart)
01528       do j=g%jsc,g%jec ; do i=g%isc,g%iec
01529         iss%mass_shelf(i,j) = rz_rescale * iss%mass_shelf(i,j)
01530       enddo ; enddo
01531     endif
01532
01533     if ((us%m_to_L_restart /= 0.0) .and. (us%m_to_L_restart /= us%m_to_L)) then
01534       l_rescale = us%m_to_L / us%m_to_L_restart
01535       do j=g%jsc,g%jec ; do i=g%isc,g%iec
01536         iss%area_shelf_h(i,j) = l_rescale**2 * iss%area_shelf_h(i,j)
01537       enddo ; enddo
01538     endif
01539
01540   endif ! .not. new_sim
01541
01542   cs%Time = time
01543
01544   call cpu_clock_begin(id_clock_pass)
01545   call pass_var(iss%area_shelf_h, g%domain)
01546   call pass_var(iss%h_shelf, g%domain)
01547   call pass_var(iss%mass_shelf, g%domain)
01548   call pass_var(iss%hmask, g%domain)
01549   call pass_var(g%bathyT, g%domain)
01550   call cpu_clock_end(id_clock_pass)
01551
01552   do j=jsd,jed ; do i=isd,ied
01553     if (iss%area_shelf_h(i,j) > g%areaT(i,j)) then
01554       call mom_error(warning,"Initialize_ice_shelf: area_shelf_h exceeds G%areaT.")
01555       iss%area_shelf_h(i,j) = g%areaT(i,j)
01556     endif
01557   enddo ; enddo
01558   if (present(fluxes)) then ; do j=jsd,jed ; do i=isd,ied
01559     if (g%areaT(i,j) > 0.0) fluxes%frac_shelf_h(i,j) = iss%area_shelf_h(i,j) / g%areaT(i,j)
01560   enddo ; enddo ; endif
01561
01562   if (cs%debug) then
01563     call hchksum(fluxes%frac_shelf_h, "IS init: frac_shelf_h", g%HI, haloshift=0)
01564   endif
01565
01566   if (present(forces)) &
01567     call add_shelf_forces(g, us, cs, forces, do_shelf_area=.not.cs%solo_ice_sheet)
01568
01569   if (present(fluxes)) call add_shelf_pressure(g, us, cs, fluxes)
01570
01571   if (cs%active_shelf_dynamics .and. .not.cs%isthermo) then
01572     iss%water_flux(:,:) = 0.0
01573   endif
01574
01575   if (shelf_mass_is_dynamic) &
01576     call initialize_ice_shelf_dyn(param_file, time, iss, cs%dCS, g, us, diag, new_sim, solo_ice_sheet_in)
01577
01578   call fix_restart_unit_scaling(us)
01579
01580   call get_param(param_file, mdl, "SAVE_INITIAL_CONDS", save_ic, &
01581                  "If true, save the ice shelf initial conditions.", &
01582                  default=.false.)
01583   if (save_ic) call get_param(param_file, mdl, "SHELF_IC_OUTPUT_FILE", ic_file,&
01584                  "The name-root of the output file for the ice shelf "//&
01585                  "initial conditions.", default="MOM_Shelf_IC")
01586
01587   if (save_ic .and. .not.((dirs%input_filename(1:1) == 'r') .and. &
01588                           (len_trim(dirs%input_filename) == 1))) then
01589     call save_restart(dirs%output_directory, cs%Time, g, &
01590                       cs%restart_CSp, filename=ic_file)
01591   endif
01592
01593
01594   cs%id_area_shelf_h = register_diag_field('ocean_model', 'area_shelf_h', cs%diag%axesT1, cs%Time, &
01595      'Ice Shelf Area in cell', 'meter-2', conversion=us%L_to_m**2)
01596   cs%id_shelf_mass = register_diag_field('ocean_model', 'shelf_mass', cs%diag%axesT1, cs%Time, &
01597      'mass of shelf', 'kg/m^2', conversion=us%RZ_to_kg_m2)
01598   cs%id_h_shelf = register_diag_field('ocean_model', 'h_shelf', cs%diag%axesT1, cs%Time, &
01599        'ice shelf thickness', 'm', conversion=us%Z_to_m)
01600   cs%id_mass_flux = register_diag_field('ocean_model', 'mass_flux', cs%diag%axesT1,&
01601      cs%Time, 'Total mass flux of freshwater across the ice-ocean interface.', &
01602      'kg/s', conversion=us%RZ_T_to_kg_m2s*us%L_to_m**2)
01603
```

```
01604   if (cs%const_gamma) then ! use ISOMIP+ eq. with rho_fw = 1000. kg m-3
01605     meltrate_conversion = 86400.0*365.0*us%Z_to_m*us%s_to_T / (1000.0*us%kg_m3_to_R)
01606   else ! use original eq.
01607     meltrate_conversion = 86400.0*365.0*us%Z_to_m*us%s_to_T / cs%density_ice
01608   endif
01609   cs%id_melt = register_diag_field('ocean_model', 'melt', cs%diag%axesT1, cs%Time, &
01610      'Ice Shelf Melt Rate', 'm yr-1', conversion= meltrate_conversion)
01611   cs%id_thermal_driving = register_diag_field('ocean_model', 'thermal_driving', cs%diag%axesT1, cs%Time, &
01612      'pot. temp. in the boundary layer minus freezing pot. temp. at the ice-ocean interface.', 'Celsius')
01613   cs%id_haline_driving = register_diag_field('ocean_model', 'haline_driving', cs%diag%axesT1, cs%Time, &
01614      'salinity in the boundary layer minus salinity at the ice-ocean interface.', 'psu')
01615   cs%id_Sbdry = register_diag_field('ocean_model', 'sbdry', cs%diag%axesT1, cs%Time, &
01616      'salinity at the ice-ocean interface.', 'psu')
01617   cs%id_u_ml = register_diag_field('ocean_model', 'u_ml', cs%diag%axesCu1, cs%Time, &
01618      'Eastward vel. in the boundary layer (used to compute ustar)', 'm s-1', conversion=us%L_T_to_m_s)
01619   cs%id_v_ml = register_diag_field('ocean_model', 'v_ml', cs%diag%axesCv1, cs%Time, &
01620      'Northward vel. in the boundary layer (used to compute ustar)', 'm s-1', conversion=us%L_T_to_m_s)
01621   cs%id_exch_vel_s = register_diag_field('ocean_model', 'exch_vel_s', cs%diag%axesT1, cs%Time, &
01622      'Sub-shelf salinity exchange velocity', 'm s-1', conversion=us%Z_to_m*us%s_to_T)
01623   cs%id_exch_vel_t = register_diag_field('ocean_model', 'exch_vel_t', cs%diag%axesT1, cs%Time, &
01624      'Sub-shelf thermal exchange velocity', 'm s-1' , conversion=us%Z_to_m*us%s_to_T)
01625   cs%id_tfreeze = register_diag_field('ocean_model', 'tfreeze', cs%diag%axesT1, cs%Time, &
01626      'In Situ Freezing point at ice shelf interface', 'degC')
01627   cs%id_tfl_shelf = register_diag_field('ocean_model', 'tflux_shelf', cs%diag%axesT1, cs%Time, &
01628      'Heat conduction into ice shelf', 'W m-2', conversion=-us%QRZ_T_to_W_m2)
01629   cs%id_ustar_shelf = register_diag_field('ocean_model', 'ustar_shelf', cs%diag%axesT1, cs%Time, &
01630      'Fric vel under shelf', 'm/s', conversion=us%Z_to_m*us%s_to_T)
01631   if (cs%active_shelf_dynamics) then
01632     cs%id_h_mask = register_diag_field('ocean_model', 'h_mask', cs%diag%axesT1, cs%Time, &
01633        'ice shelf thickness mask', 'none')
01634   endif
01635
01636   id_clock_shelf = cpu_clock_id('Ice shelf', grain=clock_component)
01637   id_clock_pass = cpu_clock_id(' Ice shelf halo updates', grain=clock_routine)
01638
```

42.4.3.8 initialize_shelf_mass()

```
subroutine mom_ice_shelf::initialize_shelf_mass (
             type(ocean_grid_type), intent(in) G,
             type(param_file_type), intent(in) param_file,
             type(ice_shelf_cs), pointer CS,
             type(ice_shelf_state), intent(inout) ISS,
             logical, intent(in), optional new_sim )  [private]
```

Initializes shelf mass based on three options (file, zero and user)

Parameters

in	*g*	The ocean's grid structure.
in	*param_file*	A structure to parse for run-time parameters
	cs	A pointer to the ice shelf control structure
in,out	*iss*	The ice shelf state type that is being updated
in	*new_sim*	If present and false, this run is being restarted

Definition at line 1643 of file MOM_ice_shelf.F90.

Referenced by initialize_ice_shelf().

```
01643
01644   type(ocean_grid_type), intent(in) :: g   !< The ocean's grid structure.
01645   type(param_file_type), intent(in) :: param_file !< A structure to parse for run-time parameters
01646   type(ice_shelf_cs),    pointer    :: cs !< A pointer to the ice shelf control structure
```

```
01647   type(ice_shelf_state), intent(inout) :: iss !< The ice shelf state type that is being updated
01648   logical,     optional, intent(in) :: new_sim !< If present and false, this run is being restarted
01649
01650   integer :: i, j, is, ie, js, je
01651   logical :: read_shelf_area, new_sim_2
01652   character(len=240) :: config, inputdir, shelf_file, filename
01653   character(len=120) :: shelf_mass_var  ! The name of shelf mass in the file.
01654   character(len=120) :: shelf_area_var ! The name of shelf area in the file.
01655   character(len=40)  :: mdl = "MOM_ice_shelf"
01656   is = g%isc ; ie = g%iec ; js = g%jsc ; je = g%jec
01657
01658   new_sim_2 = .true. ; if (present(new_sim)) new_sim_2 = new_sim
01659
01660   call get_param(param_file, mdl, "ICE_SHELF_CONFIG", config, &
01661                  "A string that specifies how the ice shelf is "//&
01662                  "initialized. Valid options include:\n"//&
01663                  " \tfile\t Read from a file.\n"//&
01664                  " \tzero\t Set shelf mass to 0 everywhere.\n"//&
01665                  " \tUSER\t Call USER_initialize_shelf_mass.\n", &
01666                  fail_if_missing=.true.)
01667
01668   select case ( trim(config) )
01669     case ("file")
01670
01671       call time_interp_external_init()
01672
01673       call get_param(param_file, mdl, "INPUTDIR", inputdir, default=".")
01674       inputdir = slasher(inputdir)
01675
01676       call get_param(param_file, mdl, "SHELF_FILE", shelf_file, &
01677               "If DYNAMIC_SHELF_MASS = True, OVERRIDE_SHELF_MOVEMENT = True "//&
01678               "and ICE_SHELF_MASS_FROM_FILE = True, this is the file from "//&
01679               "which to read the shelf mass and area.", &
01680                default="shelf_mass.nc")
01681       call get_param(param_file, mdl, "SHELF_MASS_VAR", shelf_mass_var, &
01682                  "The variable in SHELF_FILE with the shelf mass.", &
01683                  default="shelf_mass")
01684       call get_param(param_file, mdl, "READ_SHELF_AREA", read_shelf_area, &
01685                  "If true, also read the area covered by ice-shelf from SHELF_FILE.", &
01686                  default=.false.)
01687
01688       filename = trim(slasher(inputdir))//trim(shelf_file)
01689       call log_param(param_file, mdl, "INPUTDIR/SHELF_FILE", filename)
01690
01691       cs%id_read_mass = init_external_field(filename, shelf_mass_var, &
01692                             domain=g%Domain%mpp_domain, verbose=cs%debug)
01693
01694       if (read_shelf_area) then
01695          call get_param(param_file, mdl, "SHELF_AREA_VAR", shelf_area_var, &
01696                   "The variable in SHELF_FILE with the shelf area.", &
01697                   default="shelf_area")
01698
01699          cs%id_read_area = init_external_field(filename,shelf_area_var, &
01700                               domain=g%Domain%mpp_domain)
01701       endif
01702
01703       if (.not.file_exists(filename, g%Domain)) call mom_error(fatal, &
01704            " initialize_shelf_mass: Unable to open "//trim(filename))
01705
01706     case ("zero")
01707       do j=js,je ; do i=is,ie
01708         iss%mass_shelf(i,j) = 0.0
01709         iss%area_shelf_h(i,j) = 0.0
01710       enddo ; enddo
01711
01712     case ("USER")
01713       call user_initialize_shelf_mass(iss%mass_shelf, iss%area_shelf_h, &
01714                    iss%h_shelf, iss%hmask, g, cs%US, cs%user_CS, param_file, new_sim_2)
01715
01716     case default ;  call mom_error(fatal,"initialize_ice_shelf: "// &
01717       "Unrecognized ice shelf setup "//trim(config))
01718   end select
01719
```

42.4.3.9 shelf_calc_flux()

```
subroutine, public mom_ice_shelf::shelf_calc_flux (
             type(surface), intent(inout) sfc_state,
```

```
type(forcing), intent(inout) fluxes,
type(time_type), intent(in) Time,
real, intent(in) time_step,
type(ice_shelf_cs), pointer CS,
type(mech_forcing), intent(inout), optional forces )
```

Calculates fluxes between the ocean and ice-shelf using the three-equations formulation (optional to use just two equations). See ICE_SHELF equations.

Parameters

in,out	*sfc_state*	A structure containing fields that describe the surface state of the ocean. The intent is only inout to allow for halo updates.
in,out	*fluxes*	structure containing pointers to any possible thermodynamic or mass-flux forcing fields.
in	*time*	Start time of the fluxes.
in	*time_step*	Length of time over which these fluxes will be applied [s].
	cs	A pointer to the control structure returned by a previous call to initialize_ice_shelf.
in,out	*forces*	A structure with the driving mechanical forces

Definition at line 195 of file MOM_ice_shelf.F90.

References add_shelf_flux(), add_shelf_forces(), change_thickness_using_melt(), id_clock_pass, id_clock_shelf, and update_shelf_mass().

```
00195   type(surface),         intent(inout) :: sfc_state !< A structure containing fields that
00196                                           !! describe the surface state of the ocean.  The
00197                                           !! intent is only inout to allow for halo updates.
00198   type(forcing),         intent(inout) :: fluxes !< structure containing pointers to any possible
00199                                           !! thermodynamic or mass-flux forcing fields.
00200   type(time_type),       intent(in)    :: time  !< Start time of the fluxes.
00201   real,                  intent(in)    :: time_step !< Length of time over which these fluxes
00202                                           !! will be applied [s].
00203   type(ice_shelf_cs),    pointer       :: cs    !< A pointer to the control structure returned
00204                                           !! by a previous call to initialize_ice_shelf.
00205   type(mech_forcing), optional, intent(inout) :: forces !< A structure with the driving mechanical forces
00206
00207   ! Local variables
00208   type(ocean_grid_type), pointer :: g => null()  !< The grid structure used by the ice shelf.
00209   type(unit_scale_type), pointer :: us => null() !< Pointer to a structure containing
00210                                                  !! various unit conversion factors
00211   type(ice_shelf_state), pointer :: iss => null() !< A structure with elements that describe
00212                                                   !! the ice-shelf state
00213
00214   real, dimension(SZI_(CS%grid)) :: &
00215     rhoml, &   !< Ocean mixed layer density [R ~> kg m-3].
00216     dr0_dt, &  !< Partial derivative of the mixed layer density
00217                !< with temperature [R degC-1 ~> kg m-3 degC-1].
00218     dr0_ds, &  !< Partial derivative of the mixed layer density
00219                !< with salinity [R ppt-1 ~> kg m-3 ppt-1].
00220     p_int      !< The pressure at the ice-ocean interface [R L2 T-2 ~> Pa].
00221
00222   real, dimension(SZI_(CS%grid),SZJ_(CS%grid)) :: &
00223     exch_vel_t, &  !< Sub-shelf thermal exchange velocity [Z T-1 ~> m s-1]
00224     exch_vel_s     !< Sub-shelf salt exchange velocity [Z T-1 ~> m s-1]
00225
00226   real, dimension(SZDI_(CS%grid),SZDJ_(CS%grid)) :: &
00227     mass_flux  !< Total mass flux of freshwater across the ice-ocean interface. [R Z L2 T-1 ~> kg/s]
00228   real, dimension(SZDI_(CS%grid),SZDJ_(CS%grid)) :: &
00229     haline_driving !< (SSS - S_boundary) ice-ocean
00230                !! interface, positive for melting and negative for freezing.
00231                !! This is computed as part of the ISOMIP diagnostics.
00232   real, parameter :: vk    = 0.40 !< Von Karman's constant - dimensionless
00233   real :: zeta_n = 0.052 !> The fraction of the boundary layer over which the
00234                !! viscosity is linearly increasing [nondim]. (Was 1/8. Why?)
00235   real, parameter :: rc    = 0.20     ! critical flux Richardson number.
00236   real :: i_zeta_n !< The inverse of ZETA_N [nondim].
00237   real :: i_lf     !< The inverse of the latent heat of fusion [Q-1 ~> kg J-1].
00238   real :: i_vk     !< The inverse of the Von Karman constant [nondim].
00239   real :: pr, sc   !< The Prandtl number and Schmidt number [nondim].
00240
```

```
00241   ! 3 equations formulation variables
00242   real, dimension(SZDI_(CS%grid),SZDJ_(CS%grid)) :: &
00243     sbdry     !< Salinities in the ocean at the interface with the ice shelf [ppt].
00244   real :: sbdry_it
00245   real :: sbdry1, sbdry2
00246   real :: s_a, s_b, s_c  ! Variables used to find salt roots
00247   real :: ds_it    !< The interface salinity change during an iteration [ppt].
00248   real :: hbl_neut !< The neutral boundary layer thickness [Z ~> m].
00249   real :: hbl_neut_h_molec !< The ratio of the neutral boundary layer thickness
00250                   !! to the molecular boundary layer thickness [nondim].
00251   real :: wt_flux !< The downward vertical flux of heat just inside the ocean [degC Z T-1 ~> degC m s-1].
00252   real :: wb_flux !< The downward vertical flux of buoyancy just inside the ocean [Z2 T-3 ~> m2 s-3].
00253   real :: db_ds   !< The derivative of buoyancy with salinity [Z T-2 ppt-1 ~> m s-2 ppt-1].
00254   real :: db_dt   !< The derivative of buoyancy with temperature [Z T-2 degC-1 ~> m s-2 degC-1].
00255   real :: i_n_star ! [nondim]
00256   real :: n_star_term ! A term in the expression for nstar [T3 Z-2 ~> s3 m-2]
00257   real :: absf     ! The absolute value of the Coriolis parameter [T-1 ~> s-1]
00258   real :: dins_dwb !< The partial derivative of I_n_star with wB_flux, in [T3 Z-2 ~> s3 m-2]
00259   real :: dt_ustar ! The difference between the the freezing point and the ocean boundary layer
00260                    ! temperature times the friction velocity [degC Z T-1 ~> degC m s-1]
00261   real :: ds_ustar ! The difference between the salinity at the ice-ocean interface and the ocean
00262                    ! boundary layer salinity times the friction velocity [ppt Z T-1 ~> ppt m s-1]
00263   real :: ustar_h  ! The friction velocity in the water below the ice shelf [Z T-1 ~> m s-1]
00264   real :: gam_turb ! [nondim]
00265   real :: gam_mol_t, gam_mol_s ! Relative coefficients of molecular diffusivites [nondim]
00266   real :: rhocp     ! A typical ocean density times the heat capacity of water [Q R ~> J m-3]
00267   real :: ln_neut
00268   real :: mass_exch ! A mass exchange rate [R Z T-1 ~> kg m-2 s-1]
00269   real :: sb_min, sb_max
00270   real :: ds_min, ds_max
00271   ! Variables used in iterating for wB_flux.
00272   real :: wb_flux_new, ddwb_dwb_in
00273   real :: i_gam_t, i_gam_s
00274   real :: dg_dwb   ! The derivative of Gam_turb with wB [T3 Z-2 ~> s3 m-2]
00275   real :: taux2, tauy2 ! The squared surface stresses [R2 L2 Z2 T-4 ~> Pa2].
00276   real :: u2_av, v2_av ! The ice-area weighted average squared ocean velocities [L2 T-2 ~> m2 s-2]
00277   real :: asu1, asu2   ! Ocean areas covered by ice shelves at neighboring u-
00278   real :: asv1, asv2   ! and v-points [L2 ~> m2].
00279   real :: i_au, i_av   ! The Adcroft reciprocals of the ice shelf areas at adjacent points [L-2 ~> m-2]
00280   real :: irho0        ! The inverse of the mean density times a unit conversion factor [R-1 L Z-1 ~> m3
      kg-1]
00281   logical :: sb_min_set, sb_max_set
00282   logical :: update_ice_vel ! If true, it is time to update the ice shelf velocities.
00283   logical :: coupled_gl     ! If true, the grouding line position is determined based on
00284                             ! coupled ice-ocean dynamics.
00285
00286   real, parameter :: c2_3 = 2.0/3.0
00287   character(len=160) :: mesg  ! The text of an error message
00288   integer, dimension(2) :: eosdom ! The i-computational domain for the equation of state
00289   integer :: i, j, is, ie, js, je, ied, jed, it1, it3
00290
00291   if (.not. associated(cs)) call mom_error(fatal, "shelf_calc_flux: "// &
00292         "initialize_ice_shelf must be called before shelf_calc_flux.")
00293   call cpu_clock_begin(id_clock_shelf)
00294
00295   g => cs%grid ; us => cs%US
00296   iss => cs%ISS
00297
00298   ! useful parameters
00299   is = g%isc ; ie = g%iec ; js = g%jsc ; je = g%jec ; ied = g%ied ; jed = g%jed
00300   i_zeta_n = 1.0 / zeta_n
00301   i_lf = 1.0 / cs%Lat_fusion
00302   sc = cs%kv_molec/cs%kd_molec_salt
00303   pr = cs%kv_molec/cs%kd_molec_temp
00304   i_vk = 1.0/vk
00305   rhocp = cs%Rho_ocn * cs%Cp
00306
00307   !first calculate molecular component
00308   gam_mol_t = 12.5 * (pr**c2_3) - 6.0
00309   gam_mol_s = 12.5 * (sc**c2_3) - 6.0
00310
00311   ! GMM, zero some fields of the ice shelf structure (ice_shelf_CS)
00312   ! these fields are already set to zero during initialization
00313   ! However, they seem to be changed somewhere and, for diagnostic
00314   ! reasons, it is better to set them to zero again.
00315   exch_vel_t(:,:) = 0.0 ; exch_vel_s(:,:) = 0.0
00316   iss%tflux_shelf(:,:) = 0.0 ; iss%water_flux(:,:) = 0.0
00317   iss%salt_flux(:,:) = 0.0 ; iss%tflux_ocn(:,:) = 0.0 ; iss%tfreeze(:,:) = 0.0
00318   ! define Sbdry to avoid Run-Time Check Failure, when melt is not computed.
00319   haline_driving(:,:) = 0.0
00320   sbdry(:,:) = sfc_state%sss(:,:)
00321
00322   !update time
00323   cs%Time = time
00324
00325   if (cs%override_shelf_movement) then
00326     cs%time_step = time_step
```

```
00327     ! update shelf mass
00328     if (cs%mass_from_file) call update_shelf_mass(g, us, cs, iss, time)
00329   endif
00330
00331   if (cs%debug) then
00332     call hchksum(fluxes%frac_shelf_h, "frac_shelf_h before apply melting", g%HI, haloshift=0)
00333     call hchksum(sfc_state%sst, "sst before apply melting", g%HI, haloshift=0)
00334     call hchksum(sfc_state%sss, "sss before apply melting", g%HI, haloshift=0)
00335     call hchksum(sfc_state%u, "u_ml before apply melting", g%HI, haloshift=0, scale=us%L_T_to_m_s)
00336     call hchksum(sfc_state%v, "v_ml before apply melting", g%HI, haloshift=0, scale=us%L_T_to_m_s)
00337     call hchksum(sfc_state%ocean_mass, "ocean_mass before apply melting", g%HI, haloshift=0, &
00338                  scale=us%RZ_to_kg_m2)
00339   endif
00340
00341   ! Calculate the friction velocity under ice shelves, using taux_shelf and tauy_shelf if possible.
00342   if (allocated(sfc_state%taux_shelf) .and. allocated(sfc_state%tauy_shelf)) then
00343     call pass_vector(sfc_state%taux_shelf, sfc_state%tauy_shelf, g%domain, to_all, cgrid_ne)
00344   endif
00345   irho0 = us%Z_to_L / cs%Rho_ocn
00346   do j=js,je ; do i=is,ie ; if (fluxes%frac_shelf_h(i,j) > 0.0) then
00347     taux2 = 0.0 ; tauy2 = 0.0 ; u2_av = 0.0 ; v2_av = 0.0
00348     asu1 = (iss%area_shelf_h(i-1,j) + iss%area_shelf_h(i,j))
00349     asu2 = (iss%area_shelf_h(i,j) + iss%area_shelf_h(i+1,j))
00350     asv1 = (iss%area_shelf_h(i,j-1) + iss%area_shelf_h(i,j))
00351     asv2 = (iss%area_shelf_h(i,j) + iss%area_shelf_h(i,j+1))
00352     i_au = 0.0 ; if (asu1 + asu2 > 0.0) i_au = 1.0 / (asu1 + asu2)
00353     i_av = 0.0 ; if (asv1 + asv2 > 0.0) i_av = 1.0 / (asv1 + asv2)
00354     if (allocated(sfc_state%taux_shelf) .and. allocated(sfc_state%tauy_shelf)) then
00355       taux2 = (asu1 * sfc_state%taux_shelf(i-1,j)**2 + asu2 * sfc_state%taux_shelf(i,j)**2  ) * i_au
00356       tauy2 = (asv1 * sfc_state%tauy_shelf(i,j-1)**2 + asv2 * sfc_state%tauy_shelf(i,j)**2  ) * i_av
00357     endif
00358     u2_av = (asu1 * sfc_state%u(i-1,j)**2 + asu2 * sfc_state%u(i,j)**2) * i_au
00359     v2_av = (asv1 * sfc_state%v(i,j-1)**2 + asu2 * sfc_state%v(i,j)**2) * i_av
00360
00361     if (taux2 + tauy2 > 0.0) then
00362       fluxes%ustar_shelf(i,j) = max(cs%ustar_bg, us%L_to_Z * &
00363           sqrt(irho0 * sqrt(taux2 + tauy2) + cs%cdrag*cs%utide(i,j)**2))
00364     else   ! Take care of the cases when taux_shelf is not set or not allocated.
00365       fluxes%ustar_shelf(i,j) = max(cs%ustar_bg, us%L_TO_Z * &
00366           sqrt(cs%cdrag*((u2_av + v2_av) + cs%utide(i,j)**2)))
00367     endif
00368   else ! There is no shelf here.
00369     fluxes%ustar_shelf(i,j) = 0.0
00370   endif ; enddo ; enddo
00371
00372   eosdom(:) = eos_domain(g%HI)
00373   do j=js,je
00374     ! Find the pressure at the ice-ocean interface, averaged only over the
00375     ! part of the cell covered by ice shelf.
00376     do i=is,ie ; p_int(i) = cs%g_Earth * iss%mass_shelf(i,j) ; enddo
00377
00378     ! Calculate insitu densities and expansion coefficients
00379     call calculate_density(sfc_state%sst(:,j), sfc_state%sss(:,j), p_int, rhoml(:), &
00380                            cs%eqn_of_state, eosdom)
00381     call calculate_density_derivs(sfc_state%sst(:,j), sfc_state%sss(:,j), p_int, dr0_dt, dr0_ds, &
00382                            cs%eqn_of_state, eosdom)
00383
00384     do i=is,ie
00385       if ((sfc_state%ocean_mass(i,j) > cs%col_mass_melt_threshold) .and. &
00386           (iss%area_shelf_h(i,j) > 0.0) .and. cs%isthermo) then
00387
00388         if (cs%threeeq) then
00389           !   Iteratively determine a self-consistent set of fluxes, with the ocean
00390           ! salinity just below the ice-shelf as the variable that is being
00391           ! iterated for.
00392
00393           ustar_h = fluxes%ustar_shelf(i,j)
00394
00395           ! Estimate the neutral ocean boundary layer thickness as the minimum of the
00396           ! reported ocean mixed layer thickness and the neutral Ekman depth.
00397           absf = 0.25*((abs(g%CoriolisBu(i,j)) + abs(g%CoriolisBu(i-1,j-1))) + &
00398                        (abs(g%CoriolisBu(i,j-1)) + abs(g%CoriolisBu(i-1,j))))
00399           if (absf*sfc_state%Hml(i,j) <= vk*ustar_h) then ; hbl_neut = sfc_state%Hml(i,j)
00400           else ; hbl_neut = (vk*ustar_h) / absf ; endif
00401           hbl_neut_h_molec = zeta_n * ((hbl_neut * ustar_h) / (5.0 * cs%kv_molec))
00402
00403           ! Determine the mixed layer buoyancy flux, wB_flux.
00404           db_ds = (us%L_to_Z**2*cs%g_Earth / rhoml(i)) * dr0_ds(i)
00405           db_dt = (us%L_to_Z**2*cs%g_Earth / rhoml(i)) * dr0_dt(i)
00406           ln_neut = 0.0 ; if (hbl_neut_h_molec > 1.0) ln_neut = log(hbl_neut_h_molec)
00407
00408           if (cs%find_salt_root) then
00409             ! Solve for the skin salinity using the linearized liquidus parameters and
00410             ! balancing the turbulent fresh water flux in the near-boundary layer with
00411             ! the net fresh water or salt added by melting:
00412             ! (Cp/Lat_fusion)*Gamma_T_3Eq*(TFr_skin-T_ocn) = Gamma_S_3Eq*(S_skin-S_ocn)/S_skin
00413
```

```
00414             ! S_a is always < 0.0 with a realistic expression for the freezing point.
00415             s_a = cs%dTFr_dS * cs%Gamma_T_3EQ * cs%Cp
00416             s_b = cs%Gamma_T_3EQ*cs%Cp*(cs%TFr_0_0 + cs%dTFr_dp*p_int(i) - sfc_state%sst(i,j)) - &
00417                   cs%Lat_fusion * cs%Gamma_S_3EQ    ! S_b Can take either sign, but is usually negative.
00418             s_c = cs%Lat_fusion * cs%Gamma_S_3EQ * sfc_state%sss(i,j) ! Always >= 0
00419
00420             if (s_c == 0.0) then  ! The solution for fresh water.
00421               sbdry(i,j) = 0.0
00422             elseif (s_a < 0.0) then ! This is the usual ocean case
00423               if (s_b < 0.0) then ! This is almost always the case
00424                 sbdry(i,j) = 2.0*s_c / (-s_b + sqrt(s_b*s_b - 4.*s_a*s_c))
00425               else
00426                 sbdry(i,j) = (s_b + sqrt(s_b*s_b - 4.*s_a*s_c)) / (-2.*s_a)
00427               endif
00428             elseif ((s_a == 0.0) .and. (s_b < 0.0)) then ! It should be the case that S_b < 0.
00429               sbdry(i,j) = -s_c / s_b
00430             else
00431               call mom_error(fatal, "Impossible conditions found in 3-equation skin salinity calculation.")
00432             endif
00433
00434             ! Safety check
00435             if (sbdry(i,j) < 0.) then
00436               write(mesg,*) 'sfc_state%sss(i,j) = ',sfc_state%sss(i,j), 'S_a, S_b, S_c', s_a, s_b, s_c
00437               call mom_error(warning, mesg, .true.)
00438               write(mesg,*) 'I,J,Sbdry1,Sbdry2',i,j,sbdry1,sbdry2
00439               call mom_error(warning, mesg, .true.)
00440               call mom_error(fatal, "shelf_calc_flux: Negative salinity (Sbdry).")
00441             endif
00442           else
00443             ! Guess sss as the iteration starting point for the boundary salinity.
00444             sbdry(i,j) = sfc_state%sss(i,j) ; sb_max_set = .false.
00445             sb_min_set = .false.
00446           endif !find_salt_root
00447
00448           do it1 = 1,20
00449             ! Determine the potential temperature at the ice-ocean interface.
00450             call calculate_tfreeze(sbdry(i,j), p_int(i), iss%tfreeze(i,j), cs%eqn_of_state, &
00451                                    pres_scale=us%RL2_T2_to_Pa)
00452
00453             dt_ustar = (iss%tfreeze(i,j) - sfc_state%sst(i,j)) * ustar_h
00454             ds_ustar = (sbdry(i,j) - sfc_state%sss(i,j)) * ustar_h
00455
00456             ! First, determine the buoyancy flux assuming no effects of stability
00457             ! on the turbulence.  Following H & J '99, this limit also applies
00458             ! when the buoyancy flux is destabilizing.
00459
00460             if (cs%const_gamma) then ! if using a constant gamma_T
00461               ! note the different form, here I_Gam_T is NOT 1/Gam_T!
00462               i_gam_t = cs%Gamma_T_3EQ
00463               i_gam_s = cs%Gamma_S_3EQ
00464             else
00465               gam_turb = i_vk * (ln_neut + (0.5 * i_zeta_n - 1.0))
00466               i_gam_t = 1.0 / (gam_mol_t + gam_turb)
00467               i_gam_s = 1.0 / (gam_mol_s + gam_turb)
00468             endif
00469
00470             wt_flux = dt_ustar * i_gam_t
00471             wb_flux = db_ds * (ds_ustar * i_gam_s) + db_dt * wt_flux
00472
00473             if (wb_flux < 0.0) then
00474               ! The buoyancy flux is stabilizing and will reduce the tubulent
00475               ! fluxes, and iteration is required.
00476               n_star_term = (zeta_n/rc) * (hbl_neut * vk) / (ustar_h)**3
00477               do it3 = 1,30
00478                ! n_star <= 1.0 is the ratio of working boundary layer thickness
00479                ! to the neutral thickness.
00480                ! hBL = n_star*hBL_neut ; hSub = 1/8*n_star*hBL
00481
00482                 i_n_star = sqrt(1.0 - n_star_term * wb_flux)
00483                 dins_dwb = 0.5 * n_star_term / i_n_star
00484                 if (hbl_neut_h_molec > i_n_star**2) then
00485                   gam_turb = i_vk * ((ln_neut - 2.0*log(i_n_star)) + &
00486                                      (0.5*i_zeta_n*i_n_star - 1.0))
00487                   dg_dwb =  i_vk * ( -2.0 / i_n_star + (0.5 * i_zeta_n)) * dins_dwb
00488                 else
00489                   !   The layer dominated by molecular viscosity is smaller than
00490                   ! the assumed boundary layer.  This should be rare!
00491                   gam_turb = i_vk * (0.5 * i_zeta_n*i_n_star - 1.0)
00492                   dg_dwb = i_vk * (0.5 * i_zeta_n) * dins_dwb
00493                 endif
00494
00495                 if (cs%const_gamma) then ! if using a constant gamma_T
00496                   ! note the different form, here I_Gam_T is NOT 1/Gam_T!
00497                   i_gam_t = cs%Gamma_T_3EQ
00498                   i_gam_s = cs%Gamma_S_3EQ
00499                 else
00500                   i_gam_t = 1.0 / (gam_mol_t + gam_turb)
```

```
00501                 i_gam_s = 1.0 / (gam_mol_s + gam_turb)
00502               endif
00503
00504               wt_flux = dt_ustar * i_gam_t
00505               wb_flux_new = db_ds * (ds_ustar * i_gam_s) + db_dt * wt_flux
00506
00507               ! Find the root where wB_flux_new = wB_flux.  Make the 1.0e-4 below into a parameter?
00508               if (abs(wb_flux_new - wb_flux) < 1.0e-4*(abs(wb_flux_new) + abs(wb_flux))) exit
00509
00510               ddwb_dwb_in = dg_dwb * (db_ds * (ds_ustar * i_gam_s**2) + &
00511                                       db_dt * (dt_ustar * i_gam_t**2)) - 1.0
00512               ! This is Newton's method without any bounds.  Should bounds be needed?
00513               wb_flux_new = wb_flux - (wb_flux_new - wb_flux) / ddwb_dwb_in
00514             enddo !it3
00515           endif
00516
00517           iss%tflux_ocn(i,j)  = rhocp * wt_flux
00518           exch_vel_t(i,j) = ustar_h * i_gam_t
00519           exch_vel_s(i,j) = ustar_h * i_gam_s
00520
00521           ! Calculate the heat flux inside the ice shelf.
00522           ! Vertical adv/diff as in H+J 1999, eqns (26) & approx from (31).
00523           !   Q_ice = density_ice * CS%Cp_ice * K_ice * dT/dz (at interface)
00524           ! vertical adv/diff as in H+J 1999, eqs (31) & (26)...
00525           !   dT/dz ~= min( (lprec/(density_ice*K_ice))*(CS%Temp_Ice-T_freeze) , 0.0 )
00526           ! If this approximation is not made, iterations are required... See H+J Fig 3.
00527
00528           if (iss%tflux_ocn(i,j) >= 0.0) then
00529             ! Freezing occurs due to downward ocean heat flux, so zero iout ce heat flux.
00530             iss%water_flux(i,j) = -i_lf * iss%tflux_ocn(i,j)
00531             iss%tflux_shelf(i,j) = 0.0
00532           else
00533             if (cs%insulator) then
00534               !no conduction/perfect insulator
00535               iss%tflux_shelf(i,j) = 0.0
00536               iss%water_flux(i,j) = i_lf * (iss%tflux_shelf(i,j) - iss%tflux_ocn(i,j))
00537
00538             else
00539               ! With melting, from H&J 1999, eqs (31) & (26)...
00540               !   Q_ice ~= Cp_ice * (CS%Temp_Ice-T_freeze) * lprec
00541               !   RhoLF*lprec = Q_ice - ISS%tflux_ocn(i,j)
00542               !   lprec = -(ISS%tflux_ocn(i,j)) / (CS%Lat_fusion + Cp_ice * (T_freeze-CS%Temp_Ice))
00543               iss%water_flux(i,j) = -iss%tflux_ocn(i,j) / &
00544                    (cs%Lat_fusion + cs%Cp_ice * (iss%tfreeze(i,j) - cs%Temp_Ice))
00545
00546               iss%tflux_shelf(i,j) = iss%tflux_ocn(i,j) + cs%Lat_fusion*iss%water_flux(i,j)
00547             endif
00548
00549           endif
00550           !other options: dTi/dz linear through shelf, with draft in [Z ~> m], KTI in [Z2 T-1 ~> m2 s-1]
00551           !    dTi_dz = (CS%Temp_Ice - ISS%tfreeze(i,j)) / draft(i,j)
00552           !    ISS%tflux_shelf(i,j) = Rho_Ice * CS%Cp_ice * KTI * dTi_dz
00553
00554
00555           if (cs%find_salt_root) then
00556             exit ! no need to do interaction, so exit loop
00557           else
00558
00559             mass_exch = exch_vel_s(i,j) * cs%Rho_ocn
00560             sbdry_it = (sfc_state%sss(i,j) * mass_exch + cs%Salin_ice * iss%water_flux(i,j)) / &
00561                        (mass_exch + iss%water_flux(i,j))
00562             ds_it = sbdry_it - sbdry(i,j)
00563             if (abs(ds_it) < 1.0e-4*(0.5*(sfc_state%sss(i,j) + sbdry(i,j) + 1.0e-10))) exit
00564
00565
00566             if (ds_it < 0.0) then ! Sbdry is now the upper bound.
00567               if (sb_max_set .and. (sbdry(i,j) > sb_max)) &
00568                 call mom_error(fatal,"shelf_calc_flux: Irregular iteration for Sbdry (max).")
00569               sb_max = sbdry(i,j) ; ds_max = ds_it ; sb_max_set = .true.
00570             else ! Sbdry is now the lower bound.
00571               if (sb_min_set .and. (sbdry(i,j) < sb_min)) &
00572                  call mom_error(fatal, "shelf_calc_flux: Irregular iteration for Sbdry (min).")
00573                sb_min = sbdry(i,j) ; ds_min = ds_it ; sb_min_set = .true.
00574             endif ! dS_it < 0.0
00575
00576             if (sb_min_set .and. sb_max_set) then
00577               ! Use the false position method for the next iteration.
00578               sbdry(i,j) = sb_min + (sb_max-sb_min) * (ds_min / (ds_min - ds_max))
00579             else
00580               sbdry(i,j) = sbdry_it
00581             endif ! Sb_min_set
00582
00583             sbdry(i,j) = sbdry_it
00584           endif ! CS%find_salt_root
00585
00586         enddo !it1
00587         ! Check for non-convergence and/or non-boundedness?
```

```
00588
00589         else
00590           !   In the 2-equation form, the mixed layer turbulent exchange velocity
00591           ! is specified and large enough that the ocean salinity at the interface
00592           ! is about the same as the boundary layer salinity.
00593
00594           call calculate_tfreeze(sfc_state%sss(i,j), p_int(i), iss%tfreeze(i,j), cs%eqn_of_state, &
00595                                  pres_scale=us%RL2_T2_to_Pa)
00596
00597           exch_vel_t(i,j) = cs%gamma_t
00598           iss%tflux_ocn(i,j) = rhocp * exch_vel_t(i,j) * (iss%tfreeze(i,j) - sfc_state%sst(i,j))
00599           iss%tflux_shelf(i,j) = 0.0
00600           iss%water_flux(i,j) = -i_lf * iss%tflux_ocn(i,j)
00601           sbdry(i,j) = 0.0
00602         endif
00603       elseif (iss%area_shelf_h(i,j) > 0.0) then ! This is an ice-sheet, not a floating shelf.
00604         iss%tflux_ocn(i,j) = 0.0
00605       else ! There is no ice shelf or sheet here.
00606         iss%tflux_ocn(i,j) = 0.0
00607       endif
00608
00609 !      haline_driving(i,j) = sfc_state%sss(i,j) - Sbdry(i,j)
00610
00611     enddo ! i-loop
00612   enddo ! j-loop
00613
00614
00615   do j=js,je ; do i=is,ie
00616     ! ISS%water_flux = net liquid water into the ocean [R Z T-1 ~> kg m-2 s-1]
00617     fluxes%iceshelf_melt(i,j) = iss%water_flux(i,j) * cs%flux_factor
00618
00619     if ((sfc_state%ocean_mass(i,j) > cs%col_mass_melt_threshold) .and. &
00620         (iss%area_shelf_h(i,j) > 0.0) .and.  (cs%isthermo)) then
00621
00622       ! Set melt to zero above a cutoff pressure (CS%Rho_ocn*CS%cutoff_depth*CS%g_Earth).
00623       ! This is needed for the ISOMIP test case.
00624       if (iss%mass_shelf(i,j) < cs%Rho_ocn*cs%cutoff_depth) then
00625         iss%water_flux(i,j) = 0.0
00626         fluxes%iceshelf_melt(i,j) = 0.0
00627       endif
00628       ! Compute haline driving, which is one of the diags. used in ISOMIP
00629       haline_driving(i,j) = (iss%water_flux(i,j) * sbdry(i,j)) / (cs%Rho_ocn * exch_vel_s(i,j))
00630
00631       !!!!!!!!!!!!!!!!!!!!!!!!!!!!Safety checks !!!!!!!!!!!!!!!!!!!!!!!!
00632       !1)Check if haline_driving computed above is consistent with
00633       ! haline_driving = sfc_state%sss - Sbdry
00634       !if (fluxes%iceshelf_melt(i,j) /= 0.0) then
00635       !   if (haline_driving(i,j) /= (sfc_state%sss(i,j) - Sbdry(i,j))) then
00636       !     write(mesg,*) 'at i,j=',i,j,' haline_driving, sss-Sbdry',haline_driving(i,j), &
00637       !                     (sfc_state%sss(i,j) - Sbdry(i,j))
00638       !     call MOM_error(FATAL, &
00639       !            "shelf_calc_flux: Inconsistency in melt and haline_driving"//trim(mesg))
00640       !   endif
00641       !endif
00642
00643       ! 2) check if |melt| > 0 when ustar_shelf = 0.
00644       ! this should never happen
00645       if ((abs(fluxes%iceshelf_melt(i,j))>0.0) .and. (fluxes%ustar_shelf(i,j) == 0.0)) then
00646         write(mesg,*) "|melt| = ",fluxes%iceshelf_melt(i,j)," > 0 and ustar_shelf = 0. at i,j", i, j
00647         call mom_error(fatal, "shelf_calc_flux: "//trim(mesg))
00648       endif
00649       !!!!!!!!!!!!!!!!!!!!!!!!!!!!End of safety checks !!!!!!!!!!!!!!!!!!
00650     elseif (iss%area_shelf_h(i,j) > 0.0) then
00651       ! This is grounded ice, that could be modified to melt if a geothermal heat flux were used.
00652       haline_driving(i,j) = 0.0
00653       iss%water_flux(i,j) = 0.0
00654       fluxes%iceshelf_melt(i,j) = 0.0
00655     endif ! area_shelf_h
00656
00657     ! mass flux [R Z L2 T-1 ~> kg s-1], part of ISOMIP diags.
00658     mass_flux(i,j) = iss%water_flux(i,j) * iss%area_shelf_h(i,j)
00659   enddo ; enddo ! i- and j-loops
00660
00661   if (cs%active_shelf_dynamics .or. cs%override_shelf_movement) then
00662     call cpu_clock_begin(id_clock_pass)
00663     call pass_var(iss%area_shelf_h, g%domain, complete=.false.)
00664     call pass_var(iss%mass_shelf, g%domain)
00665     call cpu_clock_end(id_clock_pass)
00666   endif
00667
00668   ! Melting has been computed, now is time to update thickness and mass
00669   if ( cs%override_shelf_movement .and. (.not.cs%mass_from_file)) then
00670     call change_thickness_using_melt(iss, g, us, us%s_to_T*time_step, fluxes, cs%density_ice, cs%debug)
00671
00672     if (cs%debug) then
00673       call hchksum(iss%h_shelf, "h_shelf after change thickness using melt", g%HI, haloshift=0, scale=us
      %Z_to_m)
```

```
00674       call hchksum(iss%mass_shelf, "mass_shelf after change thickness using melt", g%HI, haloshift=0, &
00675                    scale=us%RZ_to_kg_m2)
00676     endif
00677   endif
00678
00679   if (cs%debug) call mom_forcing_chksum("Before add shelf flux", fluxes, g, cs%US, haloshift=0)
00680
00681   call add_shelf_flux(g, us, cs, sfc_state, fluxes)
00682
00683   ! now the thermodynamic data is passed on... time to update the ice dynamic quantities
00684
00685   if (cs%active_shelf_dynamics) then
00686     update_ice_vel = .false.
00687     coupled_gl = (cs%GL_couple .and. .not.cs%solo_ice_sheet)
00688
00689     ! advect the ice shelf, and advance the front. Calving will be in here somewhere as well..
00690     ! when we decide on how to do it
00691     call update_ice_shelf(cs%dCS, iss, g, us, us%s_to_T*time_step, time, &
00692                           sfc_state%ocean_mass, coupled_gl)
00693
00694   endif
00695
00696   call enable_averaging(time_step,time,cs%diag)
00697   if (cs%id_shelf_mass > 0) call post_data(cs%id_shelf_mass, iss%mass_shelf, cs%diag)
00698   if (cs%id_area_shelf_h > 0) call post_data(cs%id_area_shelf_h, iss%area_shelf_h, cs%diag)
00699   if (cs%id_ustar_shelf > 0) call post_data(cs%id_ustar_shelf, fluxes%ustar_shelf, cs%diag)
00700   if (cs%id_melt > 0) call post_data(cs%id_melt, fluxes%iceshelf_melt, cs%diag)
00701   if (cs%id_thermal_driving > 0) call post_data(cs%id_thermal_driving, (sfc_state%sst-iss%tfreeze), cs%diag
      )
00702   if (cs%id_Sbdry > 0) call post_data(cs%id_Sbdry, sbdry, cs%diag)
00703   if (cs%id_haline_driving > 0) call post_data(cs%id_haline_driving, haline_driving, cs%diag)
00704   if (cs%id_mass_flux > 0) call post_data(cs%id_mass_flux, mass_flux, cs%diag)
00705   if (cs%id_u_ml > 0) call post_data(cs%id_u_ml, sfc_state%u, cs%diag)
00706   if (cs%id_v_ml > 0) call post_data(cs%id_v_ml, sfc_state%v, cs%diag)
00707   if (cs%id_tfreeze > 0) call post_data(cs%id_tfreeze, iss%tfreeze, cs%diag)
00708   if (cs%id_tfl_shelf > 0) call post_data(cs%id_tfl_shelf, iss%tflux_shelf, cs%diag)
00709   if (cs%id_exch_vel_t > 0) call post_data(cs%id_exch_vel_t, exch_vel_t, cs%diag)
00710   if (cs%id_exch_vel_s > 0) call post_data(cs%id_exch_vel_s, exch_vel_s, cs%diag)
00711   if (cs%id_h_shelf > 0) call post_data(cs%id_h_shelf, iss%h_shelf, cs%diag)
00712   if (cs%id_h_mask > 0) call post_data(cs%id_h_mask,iss%hmask,cs%diag)
00713   call disable_averaging(cs%diag)
00714
00715   if (present(forces)) then
00716     call add_shelf_forces(g, us, cs, forces, do_shelf_area=(cs%active_shelf_dynamics .or. &
00717                                                             cs%override_shelf_movement))
00718   endif
00719
00720   call cpu_clock_end(id_clock_shelf)
00721
00722   if (cs%debug) call mom_forcing_chksum("End of shelf calc flux", fluxes, g, cs%US, haloshift=0)
00723
```

42.4.3.10 solo_step_ice_shelf()

```
subroutine, public mom_ice_shelf::solo_step_ice_shelf (
             type(ice_shelf_cs), pointer CS,
             type(time_type), intent(in) time_interval,
             integer, intent(inout) nsteps,
             type(time_type), intent(inout) Time,
             real, intent(in), optional min_time_step_in )
```

This routine is for stepping a stand-alone ice shelf model without an ocean.

Parameters

	cs	A pointer to the ice shelf control structure
in	*time_interval*	The time interval for this update [s].
in,out	*nsteps*	The running number of ice shelf steps.
in,out	*time*	The current model time
in	*min_time_step_in*	The minimum permitted time step [T ~> s].

Definition at line 1804 of file MOM_ice_shelf.F90.

```
01804   type(ice_shelf_cs), pointer    :: cs      !< A pointer to the ice shelf control structure
01805   type(time_type), intent(in)    :: time_interval !< The time interval for this update [s].
01806   integer,         intent(inout) :: nsteps  !< The running number of ice shelf steps.
01807   type(time_type), intent(inout) :: time    !< The current model time
01808   real,  optional, intent(in)    :: min_time_step_in !< The minimum permitted time step [T ~> s].
01809
01810   type(ocean_grid_type), pointer :: g => null()  ! A pointer to the ocean's grid structure
01811   type(unit_scale_type), pointer :: us => null() ! Pointer to a structure containing
01812                                                  ! various unit conversion factors
01813   type(ice_shelf_state), pointer :: iss => null() !< A structure with elements that describe
01814                                      !! the ice-shelf state
01815   real :: remaining_time    ! The remaining time in this call [T ~> s]
01816   real :: time_step         ! The internal time step during this call [T ~> s]
01817   real :: min_time_step     ! The minimal required timestep that would indicate a fatal problem [T ~> s]
01818   character(len=240) :: mesg
01819   logical :: update_ice_vel ! If true, it is time to update the ice shelf velocities.
01820   logical :: coupled_gl     ! If true the grouding line position is determined based on
01821                             ! coupled ice-ocean dynamics.
01822   integer :: is, iec, js, jec, i, j
01823
01824   g => cs%grid
01825   us => cs%US
01826   iss => cs%ISS
01827   is = g%isc ; iec = g%iec ; js = g%jsc ; jec = g%jec
01828
01829   remaining_time = us%s_to_T*time_type_to_real(time_interval)
01830
01831   if (present (min_time_step_in)) then
01832     min_time_step = min_time_step_in
01833   else
01834     min_time_step = 1000.0*us%s_to_T ! At 1 km resolution this would imply ice is moving at ~1 meter per
      second
01835   endif
01836
01837   write (mesg,*) "TIME in ice shelf call, yrs: ", time_type_to_real(time)/(365. * 86400.)
01838   call mom_mesg("solo_step_ice_shelf: "//mesg, 5)
01839
01840   do while (remaining_time > 0.0)
01841     nsteps = nsteps+1
01842
01843     ! If time_interval is not too long, this is unnecessary.
01844     time_step = min(ice_time_step_cfl(cs%dCS, iss, g), remaining_time)
01845
01846     write (mesg,*) "Ice model timestep = ", us%T_to_s*time_step, " seconds"
01847     if ((time_step < min_time_step) .and. (time_step < remaining_time))  then
01848       call mom_error(fatal, "MOM_ice_shelf:solo_step_ice_shelf: abnormally small timestep "//mesg)
01849     else
01850       call mom_mesg("solo_step_ice_shelf: "//mesg, 5)
01851     endif
01852
01853     remaining_time = remaining_time - time_step
01854
01855     ! If the last mini-timestep is a day or less, we cannot expect velocities to change by much.
01856     ! Do not update the velocities if the last step is very short.
01857     update_ice_vel = ((time_step > min_time_step) .or. (remaining_time > 0.0))
01858     coupled_gl = .false.
01859
01860     call update_ice_shelf(cs%dCS, iss, g, us, time_step, time, must_update_vel=update_ice_vel)
01861
01862     call enable_averages(time_step, time, cs%diag)
01863     if (cs%id_area_shelf_h > 0) call post_data(cs%id_area_shelf_h, iss%area_shelf_h, cs%diag)
01864     if (cs%id_h_shelf > 0) call post_data(cs%id_h_shelf, iss%h_shelf, cs%diag)
01865     if (cs%id_h_mask > 0) call post_data(cs%id_h_mask, iss%hmask, cs%diag)
01866     call disable_averaging(cs%diag)
01867
01868   enddo
01869
```

42.4.3.11 update_shelf_mass()

```
subroutine mom_ice_shelf::update_shelf_mass (
            type(ocean_grid_type), intent(inout) G,
            type(unit_scale_type), intent(in) US,
```

```
type(ice_shelf_cs), intent(in) CS,
type(ice_shelf_state), intent(inout) ISS,
type(time_type), intent(in) Time )  [private]
```

Updates the ice shelf mass using data from a file.

Parameters

in,out	*g*	The ocean's grid structure.
in	*us*	A dimensional unit scaling type
in	*cs*	A pointer to the ice shelf control structure
in,out	*iss*	The ice shelf state type that is being updated
in	*time*	The current model time

Definition at line 1724 of file MOM_ice_shelf.F90.

Referenced by shelf_calc_flux().

```
01724   type(ocean_grid_type), intent(inout) :: g   !< The ocean's grid structure.
01725   type(unit_scale_type), intent(in)    :: us   !< A dimensional unit scaling type
01726   type(ice_shelf_cs),    intent(in)    :: cs  !< A pointer to the ice shelf control structure
01727   type(ice_shelf_state), intent(inout) :: iss !< The ice shelf state type that is being updated
01728   type(time_type),       intent(in)    :: time !< The current model time
01729
01730   ! local variables
01731   integer :: i, j, is, ie, js, je
01732   is = g%isc ; ie = g%iec ; js = g%jsc ; je = g%jec
01733
01734   call time_interp_external(cs%id_read_mass, time, iss%mass_shelf)
01735   ! This should only be done if time_interp_external did an update.
01736   do j=js,je ; do i=is,ie
01737     iss%mass_shelf(i,j) = us%kg_m3_to_R*us%m_to_Z * iss%mass_shelf(i,j) ! Rescale after time_interp
01738   enddo ; enddo
01739
01740   do j=js,je ; do i=is,ie
01741     iss%area_shelf_h(i,j) = 0.0
01742     iss%hmask(i,j) = 0.
01743     if (iss%mass_shelf(i,j) > 0.0) then
01744       iss%area_shelf_h(i,j) = g%areaT(i,j)
01745       iss%h_shelf(i,j) = iss%mass_shelf(i,j) / cs%density_ice
01746       iss%hmask(i,j) = 1.
01747     endif
01748   enddo ; enddo
01749
01750   !call USER_update_shelf_mass(ISS%mass_shelf, ISS%area_shelf_h, ISS%h_shelf, &
01751   !                            ISS%hmask, CS%grid, CS%user_CS, Time, .true.)
01752
01753   if (cs%min_thickness_simple_calve > 0.0) then
01754     call ice_shelf_min_thickness_calve(g, iss%h_shelf, iss%area_shelf_h, iss%hmask, &
01755                                        cs%min_thickness_simple_calve, halo=0)
01756   endif
01757
01758   call pass_var(iss%area_shelf_h, g%domain)
01759   call pass_var(iss%h_shelf, g%domain)
01760   call pass_var(iss%hmask, g%domain)
01761   call pass_var(iss%mass_shelf, g%domain)
01762
```

42.4.4 Variable Documentation

42.4.4.1 id_clock_pass

```
integer mom_ice_shelf::id_clock_pass  [private]
```

CPU Clock for group pass calls.

Definition at line 187 of file MOM_ice_shelf.F90.

Referenced by initialize_ice_shelf(), and shelf_calc_flux().

```
00187 integer :: id_clock_pass !< CPU Clock for group pass calls
```

42.4.4.2 id_clock_shelf

```
integer mom_ice_shelf::id_clock_shelf
```

CPU Clock for the ice shelf code.

Definition at line 186 of file MOM_ice_shelf.F90.

Referenced by initialize_ice_shelf(), and shelf_calc_flux().

```
00186 integer :: id_clock_shelf !< CPU Clock for the ice shelf code
```

42.5 mom_meke Module Reference

Implements the Mesoscale Eddy Kinetic Energy framework with topographic beta effect included in computing beta in Rhines scale.

Data Types

- type meke_cs

  Control structure that contains MEKE parameters and diagnostics handles.

Functions/Subroutines

- subroutine, public step_forward_meke (MEKE, h, SN_u, SN_v, visc, dt, G, GV, US, CS, hu, hv)

  Integrates forward-in-time the MEKE eddy energy equation. See MEKE equations.
- subroutine meke_equilibrium (CS, MEKE, G, GV, US, SN_u, SN_v, drag_rate_visc, I_mass)

  Calculates the equilibrium solutino where the source depends only on MEKE diffusivity and there is no lateral diffusion of MEKE. Results is in MEKEMEKE.
- subroutine meke_equilibrium_restoring (CS, G, US, SN_u, SN_v)
- subroutine meke_lengthscales (CS, MEKE, G, GV, US, SN_u, SN_v, EKE, bottomFac2, barotrFac2, Lmix↩ Scale)

  Calculates the eddy mixing length scale and γ_b and γ_t functions that are ratios of either bottom or barotropic eddy energy to the column eddy energy, respectively. See MEKE equations.
- subroutine meke_lengthscales_0d (CS, US, area, beta, depth, Rd_dx, SN, EKE, bottomFac2, barotrFac2, LmixScale, Lrhines, Leady)

  Calculates the eddy mixing length scale and γ_b and γ_t functions that are ratios of either bottom or barotropic eddy energy to the column eddy energy, respectively. See MEKE equations.
- logical function, public meke_init (Time, G, US, param_file, diag, CS, MEKE, restart_CS)

  Initializes the MOM_MEKE module and reads parameters. Returns True if module is to be used, otherwise returns False.
- subroutine, public meke_alloc_register_restart (HI, param_file, MEKE, restart_CS)

  Allocates memory and register restart fields for the MOM_MEKE module.
- subroutine, public meke_end (MEKE, CS)

  Deallocates any variables allocated in MEKE_init or MEKE_alloc_register_restart.

42.5.1 Detailed Description

Implements the Mesoscale Eddy Kinetic Energy framework with topographic beta effect included in computing beta in Rhines scale.

42.5.2 The Mesoscale Eddy Kinetic Energy (MEKE) framework

The MEKE framework accounts for the mean potential energy removed by the first order closures used to parameterize mesoscale eddies. It requires closure at the second order, namely dissipation and transport of eddy energy.

Monitoring the sub-grid scale eddy energy budget provides a means to predict a sub-grid eddy-velocity scale which can be used in the lower order closures.

42.5.2.1 MEKE equations

The eddy kinetic energy equation is:

$$\partial_{\tilde{t}} E = \overbrace{\dot{E}_b + \gamma_\eta \dot{E}_\eta + \gamma_v \dot{E}_v}^{\text{sources}} - \overbrace{(\lambda + C_d |U_d| \gamma_b^2) E}^{\text{local dissipation}} + \overbrace{\nabla \cdot ((\kappa_E + \gamma_M \kappa_M) \nabla E - \kappa_4 \nabla^3 E)}^{\text{smoothing}}$$

where E is the eddy kinetic energy (variable `MEKE`) with units of m^{2}s^{-2}, and $\tilde{t} = at$ is a scaled time. The non-dimensional factor $a \geq 1$ is used to accelerate towards equilibrium.

The MEKE equation is two-dimensional and obtained by depth averaging the the three-dimensional eddy energy equation. In the following expressions $\langle \phi \rangle = \frac{1}{H} \int_{-D}^{\eta} \phi \, dz$ maps three dimensional terms into the two-dimensional quantities needed.

42.5.2.1.1 MEKE source terms

The source term $\dot{E}_b$ is a constant background source of energy intended to avoid the limit $E \to 0$.

The "GM" source term

$$\dot{E}_\eta = -\left\langle \overline{w'b'} \right\rangle = \left\langle \kappa_h N^2 S^2 \right\rangle \approx \left\langle \kappa_h g\prime |\nabla_\sigma \eta|^2 \right\rangle$$

equals the mean potential energy removed by the Gent-McWilliams closure, and is excluded/included in the MEKE budget by the efficiency parameter $\gamma_\eta \in [0, 1]$.

The "frictional" source term

$$\dot{E}_v = \langle \partial_i u_j \tau_{ij} \rangle$$

equals the mean kinetic energy removed by lateral viscous fluxes, and is excluded/included in the MEKE budget by the efficiency parameter $\gamma_v \in [0, 1]$.

42.5.2.1.2 MEKE dissipation terms

The local dissipation of E is parameterized through a linear damping, λ, and bottom drag, $C_d |U_d| \gamma_b^2$. The γ_b accounts for the weak projection of the column-mean eddy velocty to the bottom. In other words, the bottom velocity is estimated as $\gamma_b U_e$. The bottom drag coefficient, C_d is the same as that used in the bottom friction in the mean model equations.

The bottom drag velocity scale, U_d, has contributions from the resolved state and E:

$$U_d = \sqrt{U_b^2 + |u|^2_{z=-D} + |\gamma_b U_e|^2}.$$

where the eddy velocity scale, U_e, is given by:

$$U_e = \sqrt{2E}.$$

U_b is a constant background bottom velocity scale and is typically not used (i.e. set to zero).

Following Jansen et al., 2015, the projection of eddy energy on to the bottom is given by the ratio of bottom energy to column mean energy:

$$\gamma_b^2 = \frac{E_b}{E} = \gamma_{d0} + \left(1 + c_b \frac{L_d}{L_f}\right)^{-\frac{4}{5}},$$
$$\gamma_b^2 \leftarrow \max\left(\gamma_b^2, \gamma_{min}^2\right).$$

42.5.2.2 MEKE smoothing terms

E is laterally diffused by a diffusivity $\kappa_E + \gamma_M \kappa_M$ where κ_E is a constant diffusivity and the term $\gamma_M \kappa_M$ is a "self diffusion" using the diffusivity calculated in the section Diffusivity derived from MEKE. κ_4 is a constant bi-harmonic diffusivity.

42.5.2.3 Diffusivity derived from MEKE

The predicted eddy velocity scale, U_e, can be combined with a mixing length scale to form a diffusivity. The primary use of a MEKE derived diffusivity is for use in thickness diffusion (module mom_thickness_diffuse) and optionally in along isopycnal mixing of tracers (module mom_tracer_hor_diff). The original form used (enabled with MEKE_O↩ LD_LSCALE=True):

$$\kappa_M = \gamma_\kappa \sqrt{\gamma_t^2 U_e^2 A_\Delta}$$

where A_Δ is the area of the grid cell. Following Jansen et al., 2015, we now use

$$\kappa_M = \gamma_\kappa l_M \sqrt{\gamma_t^2 U_e^2}$$

where $\gamma_\kappa \in [0, 1]$ is a non-dimensional factor and, following Jansen et al., 2015, γ_t^2 is the ratio of barotropic eddy energy to column mean eddy energy given by

$$\gamma_t^2 = \frac{E_t}{E} = \left(1 + c_t \frac{L_d}{L_f}\right)^{-\frac{1}{4}},$$
$$\gamma_t^2 \leftarrow \max\left(\gamma_t^2, \gamma_{min}^2\right).$$

The length-scale is a configurable combination of multiple length scales:

$$l_M = \left(\frac{\alpha_d}{L_d} + \frac{\alpha_f}{L_f} + \frac{\alpha_R}{L_R} + \frac{\alpha_e}{L_e} + \frac{\alpha_\Delta}{L_\Delta} + \frac{\delta[L_c]}{L_c}\right)^{-1}$$

where

$$\begin{aligned} L_d &= \sqrt{\frac{c_g^2}{f^2 + 2\beta c_g}} \sim \frac{c_g}{f} \\ L_R &= \sqrt{\frac{U_e}{\beta^*}} \\ L_e &= \frac{U_e}{|S|N} \\ L_f &= \frac{H}{c_d} \\ L_\Delta &= \sqrt{A_\Delta}. \end{aligned}$$

L_c is a constant and $\delta[L_c]$ is the impulse function so that the term $\frac{\delta[L_c]}{L_c}$ evaluates to $\frac{1}{L_c}$ when L_c is non-zero but is dropped if $L_c = 0$.

β^* is the effective β that combines both the planetary vorticity gradient (i.e. $\beta = \nabla f$) and the topographic β effect, with the latter weighed by a weighting constant, c_β, that varies from 0 to 1, so that $c_\beta = 0$ means the topographic β effect is ignored, while $c_\beta = 1$ means it is fully considered. The new β^* therefore takes the form of

$$\beta^* = \sqrt{(\partial_x f - c_\beta \frac{f}{D} \partial_x D)^2 + (\partial_y f - c_\beta \frac{f}{D} \partial_y D)^2}$$

where D is water column depth at T points.

42.5.2.4 Viscosity derived from MEKE

As for κ_M, the predicted eddy velocity scale can be used to form a harmonic eddy viscosity,

$$\kappa_u = \gamma_u \sqrt{U_e^2 A_\Delta}$$

as well as a biharmonic eddy viscosity,

$$\kappa_4 = \gamma_4 \sqrt{U_e^2 A_\Delta^3}$$

42.5.2.5 Limit cases for local source-dissipative balance

Note that in steady-state (or when $a >> 1$) and there is no diffusion of E then

$$\overline{E} \approx \frac{\dot{E}_b + \gamma_\eta \dot{E}_\eta + \gamma_v \dot{E}_v}{\lambda + C_d |U_d| \gamma_b^2}.$$

In the linear drag limit, where $U_e << \min(U_b, |u|_{z=-D}, C_d^{-1}\lambda)$, the equilibrium becomes $\overline{E} \approx \frac{\dot{E}_b + \gamma_\eta \dot{E}_\eta + \gamma_v \dot{E}_v}{\lambda + C_d \sqrt{U_b^2 + |u|^2_{z=-D}}}$.

In the nonlinear drag limit, where $U_e >> \max(U_b, |u|_{z=-D}, C_d^{-1}\lambda)$, the equilibrium becomes $\overline{E} \approx \left(\frac{\dot{E}_b + \gamma_\eta \dot{E}_\eta + \gamma_v \dot{E}_v}{\sqrt{2} C_d \gamma_b^3}\right)^{\frac{2}{3}}$.

42.5.2.5.1 MEKE module parameters

Symbol	Module parameter
-	USE_MEKE
a	MEKE_DTSCALE
$\dot{E}_b$	MEKE_BGSRC
γ_η	MEKE_GMCOEFF
γ_v	MEKE_FrCOEFF
λ	MEKE_DAMPING
U_b	MEKE_USCALE
γ_{d0}	MEKE_CD_SCALE
c_b	MEKE_CB
c_t	MEKE_CT
κ_E	MEKE_KH
κ_4	MEKE_K4
γ_κ	MEKE_KHCOEFF
γ_M	MEKE_KHMEKE_FAC
γ_u	MEKE_VISCOSITY_COEFF_KU
γ_4	MEKE_VISCOSITY_COEFF_AU
γ^2_{min}	MEKE_MIN_GAMMA2
α_d	MEKE_ALPHA_DEFORM
α_f	MEKE_ALPHA_FRICT
α_R	MEKE_ALPHA_RHINES
α_e	MEKE_ALPHA_EADY
α_Δ	MEKE_ALPHA_GRID
L_c	MEKE_FIXED_MIXING_LENGTH
c_β	MEKE_TOPOGRAPHIC_BETA
-	MEKE_KHTH_FAC

Symbol	Model parameter
C_d	CDRAG

42.5.2.6 References

Jansen, M. F., A. J. Adcroft, R. Hallberg, and I. M. Held, 2015: Parameterization of eddy fluxes based on a mesoscale energy budget. Ocean Modelling, 92, 28–41, http://doi.org/10.1016/j.ocemod.2015.↩05.007 .

Marshall, D. P., and A. J. Adcroft, 2010: Parameterization of ocean eddies: Potential vorticity mixing, energetics and Arnold first stability theorem. Ocean Modelling, 32, 188–204, http://doi.org/10.1016/j.ocemod.↩2010.02.001 .

42.5.3 Function/Subroutine Documentation

42.5.3.1 meke_alloc_register_restart()

```
subroutine, public mom_meke::meke_alloc_register_restart (
              type(hor_index_type), intent(in) HI,
              type(param_file_type), intent(in) param_file,
              type(meke_type), pointer MEKE,
              type(mom_restart_cs), pointer restart_CS )
```

Allocates memory and register restart fields for the MOM_MEKE module.

Parameters

in	*hi*	Horizontal index structure
in	*param_file*	Parameter file parser structure.
	meke	A structure with MEKE-related fields.
	restart_cs	Restart control structure for MOM_MEKE.

Definition at line 1347 of file MOM_MEKE.F90.

```
01347 ! Arguments
01348   type(hor_index_type),  intent(in)     :: hi         !< Horizontal index structure
01349   type(param_file_type), intent(in)     :: param_file !< Parameter file parser structure.
01350   type(meke_type),       pointer        :: meke       !< A structure with MEKE-related fields.
01351   type(mom_restart_cs),  pointer        :: restart_cs !< Restart control structure for MOM_MEKE.
01352 ! Local variables
01353   type(vardesc) :: vd
01354   real :: meke_gmcoeff, meke_frcoeff, meke_gmecoeff, meke_khcoeff, meke_visccoeff_ku, meke_visccoeff_au
01355   logical :: use_kh_in_meke
01356   logical :: usemeke
01357   integer :: isd, ied, jsd, jed
01358
01359 ! Determine whether this module will be used
01360   usemeke = .false.; call read_param(param_file,"USE_MEKE",usemeke)
01361
01362 ! Read these parameters to determine what should be in the restarts
01363   meke_gmcoeff =-1.; call read_param(param_file,"MEKE_GMCOEFF",meke_gmcoeff)
01364   meke_frcoeff =-1.; call read_param(param_file,"MEKE_FRCOEFF",meke_frcoeff)
01365   meke_gmecoeff =-1.; call read_param(param_file,"MEKE_GMECOEFF",meke_gmecoeff)
01366   meke_khcoeff =1.; call read_param(param_file,"MEKE_KHCOEFF",meke_khcoeff)
```

```
01367   meke_visccoeff_ku =0.; call read_param(param_file,"MEKE_VISCOSITY_COEFF_KU",meke_visccoeff_ku)
01368   meke_visccoeff_au =0.; call read_param(param_file,"MEKE_VISCOSITY_COEFF_AU",meke_visccoeff_au)
01369   use_kh_in_meke = .false.; call read_param(param_file,"USE_KH_IN_MEKE", use_kh_in_meke)
01370 ! Allocate control structure
01371   if (associated(meke)) then
01372     call mom_error(warning, "MEKE_alloc_register_restart called with an associated "// &
01373                             "MEKE type.")
01374     return
01375   else; allocate(meke); endif
01376
01377   if (.not. usemeke) return
01378
01379 ! Allocate memory
01380   call mom_mesg("MEKE_alloc_register_restart: allocating and registering", 5)
01381   isd = hi%isd ; ied = hi%ied ; jsd = hi%jsd ; jed = hi%jed
01382   allocate(meke%MEKE(isd:ied,jsd:jed)) ; meke%MEKE(:,:) = 0.0
01383   vd = var_desc("MEKE", "m2 s-2", hor_grid='h', z_grid='1', &
01384            longname="Mesoscale Eddy Kinetic Energy")
01385   call register_restart_field(meke%MEKE, vd, .false., restart_cs)
01386   if (meke_gmcoeff>=0.) then
01387     allocate(meke%GM_src(isd:ied,jsd:jed)) ; meke%GM_src(:,:) = 0.0
01388   endif
01389   if (meke_frcoeff>=0. .or. meke_gmecoeff>=0.)  then
01390     allocate(meke%mom_src(isd:ied,jsd:jed)) ; meke%mom_src(:,:) = 0.0
01391   endif
01392   if (meke_gmecoeff>=0.) then
01393     allocate(meke%GME_snk(isd:ied,jsd:jed)) ; meke%GME_snk(:,:) = 0.0
01394   endif
01395   if (meke_khcoeff>=0.) then
01396     allocate(meke%Kh(isd:ied,jsd:jed)) ; meke%Kh(:,:) = 0.0
01397     vd = var_desc("MEKE_Kh", "m2 s-1", hor_grid='h', z_grid='1', &
01398              longname="Lateral diffusivity from Mesoscale Eddy Kinetic Energy")
01399     call register_restart_field(meke%Kh, vd, .false., restart_cs)
01400   endif
01401   allocate(meke%Rd_dx_h(isd:ied,jsd:jed)) ; meke%Rd_dx_h(:,:) = 0.0
01402   if (meke_visccoeff_ku/=0.) then
01403     allocate(meke%Ku(isd:ied,jsd:jed)) ; meke%Ku(:,:) = 0.0
01404     vd = var_desc("MEKE_Ku", "m2 s-1", hor_grid='h', z_grid='1', &
01405              longname="Lateral viscosity from Mesoscale Eddy Kinetic Energy")
01406     call register_restart_field(meke%Ku, vd, .false., restart_cs)
01407   endif
01408   if (use_kh_in_meke) then
01409     allocate(meke%Kh_diff(isd:ied,jsd:jed)) ; meke%Kh_diff(:,:) = 0.0
01410     vd = var_desc("MEKE_Kh_diff", "m2 s-1",hor_grid='h',z_grid='1', &
01411              longname="Copy of thickness diffusivity for diffusing MEKE")
01412     call register_restart_field(meke%Kh_diff, vd, .false., restart_cs)
01413   endif
01414
01415   if (meke_visccoeff_au/=0.) then
01416     allocate(meke%Au(isd:ied,jsd:jed)) ; meke%Au(:,:) = 0.0
01417     vd = var_desc("MEKE_Au", "m4 s-1", hor_grid='h', z_grid='1', &
01418              longname="Lateral biharmonic viscosity from Mesoscale Eddy Kinetic Energy")
01419     call register_restart_field(meke%Au, vd, .false., restart_cs)
01420   endif
01421
```

42.5.3.2 meke_end()

```
subroutine, public mom_meke::meke_end (
              type(meke_type), pointer MEKE,
              type(meke_cs), pointer CS )
```

Deallocates any variables allocated in MEKE_init or MEKE_alloc_register_restart.

Parameters

meke	A structure with MEKE-related fields.
cs	The control structure for MOM_MEKE.

Definition at line 1427 of file MOM_MEKE.F90.

```
01427   type(meke_type), pointer :: meke !< A structure with MEKE-related fields.
01428   type(meke_cs),   pointer :: cs   !< The control structure for MOM_MEKE.
01429
01430   if (associated(cs)) deallocate(cs)
01431
01432   if (.not.associated(meke)) return
01433
01434   if (associated(meke%MEKE)) deallocate(meke%MEKE)
01435   if (associated(meke%GM_src)) deallocate(meke%GM_src)
01436   if (associated(meke%mom_src)) deallocate(meke%mom_src)
01437   if (associated(meke%GME_snk)) deallocate(meke%GME_snk)
01438   if (associated(meke%Kh)) deallocate(meke%Kh)
01439   if (associated(meke%Kh_diff)) deallocate(meke%Kh_diff)
01440   if (associated(meke%Ku)) deallocate(meke%Ku)
01441   if (associated(meke%Au)) deallocate(meke%Au)
01442   deallocate(meke)
01443
```

42.5.3.3 meke_equilibrium()

```
subroutine mom_meke::meke_equilibrium (
             type(meke_cs), pointer CS,
             type(meke_type), pointer MEKE,
             type(ocean_grid_type), intent(inout) G,
             type(verticalgrid_type), intent(in) GV,
             type(unit_scale_type), intent(in) US,
             real, dimension( g %isdb:  g %iedb, g %jsd:  g %jed), intent(in) SN_u,
             real, dimension( g %isd:  g %ied, g %jsdb:  g %jedb), intent(in) SN_v,
             real, dimension( g %isd:  g %ied, g %jsd:  g %jed), intent(in) drag_rate_visc,
             real, dimension( g %isd:  g %ied, g %jsd:  g %jed), intent(in) I_mass )  [private]
```

Calculates the equilibrium solutino where the source depends only on MEKE diffusivity and there is no lateral diffusion of MEKE. Results is in MEKEMEKE.

Parameters

in,out	*g*	Ocean grid.
in	*gv*	Ocean vertical grid structure.
in	*us*	A dimensional unit scaling type
	cs	MEKE control structure.
	meke	A structure with MEKE data.
in	*sn_u*	Eady growth rate at u-points [T-1 ~> s-1].
in	*sn_v*	Eady growth rate at v-points [T-1 ~> s-1].
in	*drag_rate_visc*	Mean flow velocity contribution to the MEKE drag rate [L T-1 ~> m s-1]
in	*i_mass*	Inverse of column mass [R-1 Z-1 ~> m2 kg-1].

Definition at line 644 of file MOM_MEKE.F90.

References meke_lengthscales_0d().

Referenced by step_forward_meke().

```
00644   type(ocean_grid_type),           intent(inout) :: g    !< Ocean grid.
00645   type(verticalgrid_type),         intent(in)    :: gv   !< Ocean vertical grid structure.
00646   type(unit_scale_type),           intent(in)    :: us   !< A dimensional unit scaling type
00647   type(meke_cs),                   pointer       :: cs   !< MEKE control structure.
00648   type(meke_type),                 pointer       :: meke !< A structure with MEKE data.
```

```
00649   real, dimension(SZIB_(G),SZJ_(G)), intent(in)    :: sn_u !< Eady growth rate at u-points [T-1 ~> s-1].
00650   real, dimension(SZI_(G),SZJB_(G)), intent(in)    :: sn_v !< Eady growth rate at v-points [T-1 ~> s-1].
00651   real, dimension(SZI_(G),SZJ_(G)),  intent(in)    :: drag_rate_visc !< Mean flow velocity contribution
00652                                                         !! to the MEKE drag rate [L T-1 ~> m s-1]
00653   real, dimension(SZI_(G),SZJ_(G)),  intent(in)    :: i_mass  !< Inverse of column mass [R-1 Z-1 ~> m2
       kg-1].
00654   ! Local variables
00655   real :: beta ! Combined topograpic and planetary vorticity gradient [T-1 L-1 ~> s-1 m-1]
00656   real :: sn   ! The local Eady growth rate [T-1 ~> s-1]
00657   real :: bottomfac2, barotrfac2    ! Vertical structure factors [nondim]
00658   real :: lmixscale, lrhines, leady ! Various mixing length scales [L ~> m]
00659   real :: i_h, khcoeff
00660   real :: kh    ! A lateral diffusivity [L2 T-1 ~> m2 s-1]
00661   real :: ubg2  ! Background (tidal?) velocity squared [L2 T-2 ~> m2 s-2]
00662   real :: cd2
00663   real :: drag_rate ! The MEKE spindown timescale due to bottom drag [T-1 ~> s-1].
00664   real :: src   ! The sum of MEKE sources [L2 T-3 ~> W kg-1]
00665   real :: ldamping  ! The MEKE damping rate [T-1 ~> s-1].
00666   real :: eke, ekemin, ekemax, ekeerr ! [L2 T-2 ~> m2 s-2]
00667   real :: resid, resmin, resmax ! Residuals [L2 T-3 ~> W kg-1]
00668   real :: fath    ! Coriolis parameter at h points; to compute topographic beta [T-1 ~> s-1]
00669   real :: beta_topo_x, beta_topo_y    ! Topographic PV gradients in x and y [T-1 L-1 ~> s-1 m-1]
00670   integer :: i, j, is, ie, js, je, n1, n2
00671   real :: tolerance ! Width of EKE bracket [L2 T-2 ~> m2 s-2].
00672   logical :: usesecant, debugiteration
00673
00674   real :: lgrid, ldeform, lfrict
00675
00676   is = g%isc ; ie = g%iec ; js = g%jsc ; je = g%jec
00677
00678   debugiteration = .false.
00679   khcoeff = cs%MEKE_KhCoeff
00680   ubg2 = cs%MEKE_Uscale**2
00681   cd2 = cs%cdrag**2
00682   tolerance = 1.0e-12*us%m_s_to_L_T**2
00683
00684 !$OMP do
00685   do j=js,je ; do i=is,ie
00686     ! SN = 0.25*max( (SN_u(I,j) + SN_u(I-1,j)) + (SN_v(i,J) + SN_v(i,J-1)), 0.)
00687     ! This avoids extremes values in equilibrium solution due to bad values in SN_u, SN_v
00688     sn = min(sn_u(i,j), sn_u(i-1,j), sn_v(i,j), sn_v(i,j-1))
00689
00690     if (cs%MEKE_equilibrium_alt) then
00691       meke%MEKE(i,j) = (cs%MEKE_GEOMETRIC_alpha * sn * us%Z_to_m*g%bathyT(i,j))**2 / cd2
00692     else
00693       fath = 0.25*((g%CoriolisBu(i,j) + g%CoriolisBu(i-1,j-1)) + &
00694                    (g%CoriolisBu(i-1,j) + g%CoriolisBu(i,j-1))) ! Coriolis parameter at h points
00695
00696       ! Since zero-bathymetry cells are masked, this avoids calculations on land
00697       if (cs%MEKE_topographic_beta == 0. .or. g%bathyT(i,j) == 0.) then
00698         beta_topo_x = 0. ; beta_topo_y = 0.
00699       else
00700         !### Consider different combinations of these estimates of topographic beta, and the use
00701         !    of the water column thickness instead of the bathymetric depth.
00702         beta_topo_x = -cs%MEKE_topographic_beta * fath * 0.5 * ( &
00703                       (g%bathyT(i+1,j)-g%bathyT(i,j)) * g%IdxCu(i,j)  &
00704                     / max(g%bathyT(i+1,j),g%bathyT(i,j), gv%H_subroundoff) &
00705                   +       (g%bathyT(i,j)-g%bathyT(i-1,j)) * g%IdxCu(i-1,j) &
00706                     / max(g%bathyT(i,j),g%bathyT(i-1,j), gv%H_subroundoff) )
00707         beta_topo_y = -cs%MEKE_topographic_beta * fath * 0.5 * ( &
00708                       (g%bathyT(i,j+1)-g%bathyT(i,j)) * g%IdyCv(i,j)  &
00709                     / max(g%bathyT(i,j+1),g%bathyT(i,j), gv%H_subroundoff) + &
00710                       (g%bathyT(i,j)-g%bathyT(i,j-1)) * g%IdyCv(i,j-1) &
00711                     / max(g%bathyT(i,j),g%bathyT(i,j-1), gv%H_subroundoff) )
00712       endif
00713       beta =  sqrt((g%dF_dx(i,j) + beta_topo_x)**2 + &
00714                    (g%dF_dy(i,j) + beta_topo_y)**2 )
00715
00716       i_h = us%L_to_Z*gv%Rho0 * i_mass(i,j)
00717
00718       if (khcoeff*sn*i_h>0.) then
00719         ! Solve resid(E) = 0, where resid = Kh(E) * (SN)^2 - damp_rate(E) E
00720         ekemin = 0.   ! Use the trivial root as the left bracket
00721         resmin = 0.   ! Need to detect direction of left residual
00722         ekemax = 0.01*us%m_s_to_L_T**2 ! First guess at right bracket
00723         usesecant = .false. ! Start using a bisection method
00724
00725         ! First find right bracket for which resid<0
00726         resid = 1.0*us%m_to_L**2*us%T_to_s**3 ; n1 = 0
00727         do while (resid>0.)
00728           n1 = n1 + 1
00729           eke = ekemax
00730           call meke_lengthscales_0d(cs, us, g%areaT(i,j), beta, g%bathyT(i,j), &
00731                                     meke%Rd_dx_h(i,j), sn, eke, &
00732                                     bottomfac2, barotrfac2, lmixscale, lrhines, leady)
00733           ! TODO: Should include resolution function in Kh
00734           kh = (khcoeff * sqrt(2.*barotrfac2*eke) * lmixscale)
```

```
00735           src = kh * (sn * sn)
00736           drag_rate = i_h * sqrt(drag_rate_visc(i,j)**2 + cd2 * ( 2.0*bottomfac2*eke + ubg2 ) )
00737           ldamping = cs%MEKE_damping + drag_rate * bottomfac2
00738           resid = src - ldamping * eke
00739           ! if (debugIteration) then
00740           !   write(0,*) n1, 'EKE=',EKE,'resid=',resid
00741           !   write(0,*) 'EKEmin=',EKEmin,'ResMin=',ResMin
00742           !   write(0,*) 'src=',src,'ldamping=',ldamping
00743           !   write(0,*) 'gamma-b=',bottomFac2,'gamma-t=',barotrFac2
00744           !   write(0,*) 'drag_visc=',drag_rate_visc(i,j),'Ubg2=',Ubg2
00745           ! endif
00746           if (resid>0.) then    ! EKE is to the left of the root
00747             ekemin = eke        ! so we move the left bracket here
00748             ekemax = 10. * eke  ! and guess again for the right bracket
00749             if (resid<resmin) usesecant = .true.
00750             resmin = resid
00751             if (ekemax > 2.e17*us%m_s_to_L_T**2) then
00752               if (debugiteration) stop 'Something has gone very wrong'
00753               debugiteration = .true.
00754               resid = 1. ; n1 = 0
00755               ekemin = 0. ; resmin = 0.
00756               ekemax = 0.01*us%m_s_to_L_T**2
00757               usesecant = .false.
00758             endif
00759           endif
00760         enddo ! while(resid>0.) searching for right bracket
00761         resmax = resid
00762
00763         ! Bisect the bracket
00764         n2 = 0 ; ekeerr = ekemax - ekemin
00765         do while (ekeerr > tolerance)
00766           n2 = n2 + 1
00767           if (usesecant) then
00768             eke = ekemin + (ekemax - ekemin) * (resmin / (resmin - resmax))
00769           else
00770             eke = 0.5 * (ekemin + ekemax)
00771           endif
00772           ekeerr = min( eke-ekemin, ekemax-eke )
00773           ! TODO: Should include resolution function in Kh
00774           kh = (khcoeff * sqrt(2.*barotrfac2*eke) * lmixscale)
00775           src = kh * (sn * sn)
00776           drag_rate = i_h * sqrt( drag_rate_visc(i,j)**2 + cd2 * ( 2.0*bottomfac2*eke + ubg2 ) )
00777           ldamping = cs%MEKE_damping + drag_rate * bottomfac2
00778           resid = src - ldamping * eke
00779           if (usesecant .and. resid>resmin) usesecant = .false.
00780           if (resid>0.) then              ! EKE is to the left of the root
00781             ekemin = eke                  ! so we move the left bracket here
00782             if (resid<resmin) usesecant = .true.
00783             resmin = resid                ! Save this for the secant method
00784           elseif (resid<0.) then          ! EKE is to the right of the root
00785             ekemax = eke                  ! so we move the right bracket here
00786             resmax = resid                ! Save this for the secant method
00787           else
00788             exit                          ! resid=0 => EKE is exactly at the root
00789           endif
00790           if (n2>200) stop 'Failing to converge?'
00791         enddo ! while(EKEmax-EKEmin>tolerance)
00792
00793       else
00794         eke = 0.
00795       endif
00796       meke%MEKE(i,j) = eke
00797     endif
00798   enddo ; enddo
00799
```

42.5.3.4 meke_equilibrium_restoring()

```
subroutine mom_meke::meke_equilibrium_restoring (
             type(meke_cs), pointer CS,
             type(ocean_grid_type), intent(inout) G,
             type(unit_scale_type), intent(in) US,
             real, dimension( g %isdb:  g %iedb, g %jsd:  g %jed), intent(in) SN_u,
             real, dimension( g %isd:  g %ied, g %jsdb:  g %jedb), intent(in) SN_v )  [private]
```

Parameters

in,out	*g*	Ocean grid.
in	*us*	A dimensional unit scaling type.
	cs	MEKE control structure.
in	*sn_u*	Eady growth rate at u-points [T-1 ~> s-1].
in	*sn_v*	Eady growth rate at v-points [T-1 ~> s-1].

Definition at line 806 of file MOM_MEKE.F90.

Referenced by step_forward_meke().

```
00806   type(ocean_grid_type),              intent(inout) :: g    !< Ocean grid.
00807   type(unit_scale_type),              intent(in)    :: us   !< A dimensional unit scaling type.
00808   type(meke_cs),                      pointer       :: cs   !< MEKE control structure.
00809   real, dimension(SZIB_(G),SZJ_(G)), intent(in)    :: sn_u !< Eady growth rate at u-points [T-1 ~> s-1].
00810   real, dimension(SZI_(G),SZJB_(G)), intent(in)    :: sn_v !< Eady growth rate at v-points [T-1 ~> s-1].
00811   ! Local variables
00812   real :: sn                      ! The local Eady growth rate [T-1 ~> s-1]
00813   integer :: i, j, is, ie, js, je ! local indices
00814   real :: cd2                     ! bottom drag
00815
00816   is = g%isc ; ie = g%iec ; js = g%jsc ; je = g%jec
00817   cd2 = cs%cdrag**2
00818
00819   if (.not. associated(cs%equilibrium_value)) allocate(cs%equilibrium_value(szi_(g),szj_(g)))
00820   cs%equilibrium_value(:,:) = 0.0
00821
00822 !$OMP do
00823   do j=js,je ; do i=is,ie
00824     ! SN = 0.25*max( (SN_u(I,j) + SN_u(I-1,j)) + (SN_v(i,J) + SN_v(i,J-1)), 0.)
00825     ! This avoids extremes values in equilibrium solution due to bad values in SN_u, SN_v
00826     sn = min(sn_u(i,j), sn_u(i-1,j), sn_v(i,j), sn_v(i,j-1))
00827     cs%equilibrium_value(i,j) = (cs%MEKE_GEOMETRIC_alpha * sn * us%Z_to_m*g%bathyT(i,j))**2 / cd2
00828   enddo ; enddo
00829
00830   if (cs%id_MEKE_equilibrium>0) call post_data(cs%id_MEKE_equilibrium, cs%equilibrium_value, cs%diag)
00831
```

42.5.3.5 meke_init()

```
logical function, public mom_meke::meke_init (
              type(time_type), intent(in) Time,
              type(ocean_grid_type), intent(inout) G,
              type(unit_scale_type), intent(in) US,
              type(param_file_type), intent(in) param_file,
              type(diag_ctrl), intent(inout), target diag,
              type(meke_cs), pointer CS,
              type(meke_type), pointer MEKE,
              type(mom_restart_cs), pointer restart_CS )
```

Initializes the MOM_MEKE module and reads parameters. Returns True if module is to be used, otherwise returns False.

Parameters

in	*time*	The current model time.
in,out	*g*	The ocean's grid structure.

Parameters

in	*us*	A dimensional unit scaling type
in	*param_file*	Parameter file parser structure.
in,out	*diag*	Diagnostics structure.
	cs	MEKE control structure.
	meke	MEKE-related fields.
	restart_cs	Restart control structure for MOM_MEKE.

Definition at line 984 of file MOM_MEKE.F90.

```
00984   type(time_type),         intent(in)    :: time       !< The current model time.
00985   type(ocean_grid_type),   intent(inout) :: g          !< The ocean's grid structure.
00986   type(unit_scale_type),   intent(in)    :: us         !< A dimensional unit scaling type
00987   type(param_file_type),   intent(in)    :: param_file !< Parameter file parser structure.
00988   type(diag_ctrl), target, intent(inout) :: diag       !< Diagnostics structure.
00989   type(meke_cs),           pointer       :: cs         !< MEKE control structure.
00990   type(meke_type),         pointer       :: meke       !< MEKE-related fields.
00991   type(mom_restart_cs),    pointer       :: restart_cs !< Restart control structure for MOM_MEKE.
00992
00993   ! Local variables
00994   real    :: i_t_rescale   ! A rescaling factor for time from the internal representation in this
00995                            ! run to the representation in a restart file.
00996   real    :: l_rescale     ! A rescaling factor for length from the internal representation in this
00997                            ! run to the representation in a restart file.
00998   real    :: meke_restoring_timescale ! The timescale used to nudge MEKE toward its equilibrium value.
00999   real :: cdrag            ! The default bottom drag coefficient [nondim].
01000   integer :: i, j, is, ie, js, je, isd, ied, jsd, jed
01001   logical :: laplacian, biharmonic, usevarmix, coldstart
01002   ! This include declares and sets the variable "version".
01003 # include "version_variable.h"
01004   character(len=40)  :: mdl = "MOM_MEKE" ! This module's name.
01005
01006   is = g%isc ; ie = g%iec ; js = g%jsc ; je = g%jec
01007   isd = g%isd ; ied = g%ied ; jsd = g%jsd ; jed = g%jed
01008
01009   ! Determine whether this module will be used
01010   call get_param(param_file, mdl, "USE_MEKE", meke_init, default=.false., do_not_log=.true.)
01011   call log_version(param_file, mdl, version, "", all_default=.not.meke_init)
01012   call get_param(param_file, mdl, "USE_MEKE", meke_init, &
01013                  "If true, turns on the MEKE scheme which calculates "// &
01014                  "a sub-grid mesoscale eddy kinetic energy budget.", &
01015                  default=.false.)
01016   if (.not. meke_init) return
01017
01018   if (.not. associated(meke)) then
01019     ! The MEKE structure should have been allocated in MEKE_alloc_register_restart()
01020     call mom_error(warning, "MEKE_init called with NO associated "// &
01021                             "MEKE-type structure.")
01022     return
01023   endif
01024   if (associated(cs)) then
01025     call mom_error(warning, &
01026       "MEKE_init called with an associated control structure.")
01027     return
01028   else ; allocate(cs) ; endif
01029
01030   call mom_mesg("MEKE_init: reading parameters ", 5)
01031
01032   ! Read all relevant parameters and write them to the model log.
01033   call get_param(param_file, mdl, "MEKE_DAMPING", cs%MEKE_damping, &
01034                  "The local depth-independent MEKE dissipation rate.", &
01035                  units="s-1", default=0.0, scale=us%T_to_s)
01036   call get_param(param_file, mdl, "MEKE_CD_SCALE", cs%MEKE_Cd_scale, &
01037                  "The ratio of the bottom eddy velocity to the column mean "//&
01038                  "eddy velocity, i.e. sqrt(2*MEKE). This should be less than 1 "//&
01039                  "to account for the surface intensification of MEKE.", &
01040                  units="nondim", default=0.)
01041   call get_param(param_file, mdl, "MEKE_CB", cs%MEKE_Cb, &
01042                  "A coefficient in the expression for the ratio of bottom projected "//&
01043                  "eddy energy and mean column energy (see Jansen et al. 2015).",&
01044                  units="nondim", default=25.)
01045   call get_param(param_file, mdl, "MEKE_MIN_GAMMA2", cs%MEKE_min_gamma, &
01046                  "The minimum allowed value of gamma_b^2.",&
01047                  units="nondim", default=0.0001)
01048   call get_param(param_file, mdl, "MEKE_CT", cs%MEKE_Ct, &
01049                  "A coefficient in the expression for the ratio of barotropic "//&
01050                  "eddy energy and mean column energy (see Jansen et al. 2015).",&
```

```
01051                 units="nondim", default=50.)
01052   call get_param(param_file, mdl, "MEKE_GMCOEFF", cs%MEKE_GMcoeff, &
01053                 "The efficiency of the conversion of potential energy "//&
01054                 "into MEKE by the thickness mixing parameterization. "//&
01055                 "If MEKE_GMCOEFF is negative, this conversion is not "//&
01056                 "used or calculated.", units="nondim", default=-1.0)
01057   call get_param(param_file, mdl, "MEKE_GEOMETRIC", cs%MEKE_GEOMETRIC, &
01058                 "If MEKE_GEOMETRIC is true, uses the GM coefficient formulation "//&
01059                 "from the GEOMETRIC framework (Marshall et al., 2012).", default=.false.)
01060   call get_param(param_file, mdl, "MEKE_GEOMETRIC_ALPHA", cs%MEKE_GEOMETRIC_alpha, &
01061                 "The nondimensional coefficient governing the efficiency of the GEOMETRIC \n"//&
01062                 "thickness diffusion.", units="nondim", default=0.05)
01063   call get_param(param_file, mdl, "MEKE_EQUILIBRIUM_ALT", cs%MEKE_equilibrium_alt, &
01064                 "If true, use an alternative formula for computing the (equilibrium)"//&
01065                 "initial value of MEKE.", default=.false.)
01066   call get_param(param_file, mdl, "MEKE_EQUILIBRIUM_RESTORING", cs%MEKE_equilibrium_restoring, &
01067                 "If true, restore MEKE back to its equilibrium value, which is calculated at "//&
01068                 "each time step.", default=.false.)
01069   if (cs%MEKE_equilibrium_restoring) then
01070     call get_param(param_file, mdl, "MEKE_RESTORING_TIMESCALE", meke_restoring_timescale, &
01071                   "The timescale used to nudge MEKE toward its equilibrium value.", units="s", &
01072                   default=1e6, scale=us%T_to_s)
01073     cs%MEKE_restoring_rate = 1.0 / meke_restoring_timescale
01074   endif
01075
01076   call get_param(param_file, mdl, "MEKE_FRCOEFF", cs%MEKE_FrCoeff, &
01077                 "The efficiency of the conversion of mean energy into "//&
01078                 "MEKE.  If MEKE_FRCOEFF is negative, this conversion "//&
01079                 "is not used or calculated.", units="nondim", default=-1.0)
01080   call get_param(param_file, mdl, "MEKE_GMECOEFF", cs%MEKE_GMECoeff, &
01081                 "The efficiency of the conversion of MEKE into mean energy "//&
01082                 "by GME.  If MEKE_GMECOEFF is negative, this conversion "//&
01083                 "is not used or calculated.", units="nondim", default=-1.0)
01084   call get_param(param_file, mdl, "MEKE_BGSRC", cs%MEKE_BGsrc, &
01085                 "A background energy source for MEKE.", units="W kg-1", &
01086                 default=0.0, scale=us%m_to_L**2*us%T_to_s**3)
01087   call get_param(param_file, mdl, "MEKE_KH", cs%MEKE_Kh, &
01088                 "A background lateral diffusivity of MEKE. "//&
01089                 "Use a negative value to not apply lateral diffusion to MEKE.", &
01090                 units="m2 s-1", default=-1.0, scale=us%m_to_L**2*us%T_to_s)
01091   call get_param(param_file, mdl, "MEKE_K4", cs%MEKE_K4, &
01092                 "A lateral bi-harmonic diffusivity of MEKE. "//&
01093                 "Use a negative value to not apply bi-harmonic diffusion to MEKE.", &
01094                 units="m4 s-1", default=-1.0, scale=us%m_to_L**4*us%T_to_s)
01095   call get_param(param_file, mdl, "MEKE_DTSCALE", cs%MEKE_dtScale, &
01096                 "A scaling factor to accelerate the time evolution of MEKE.", &
01097                 units="nondim", default=1.0)
01098   call get_param(param_file, mdl, "MEKE_KHCOEFF", cs%MEKE_KhCoeff, &
01099                 "A scaling factor in the expression for eddy diffusivity "//&
01100                 "which is otherwise proportional to the MEKE velocity- "//&
01101                 "scale times an eddy mixing-length. This factor "//&
01102                 "must be >0 for MEKE to contribute to the thickness/ "//&
01103                 "and tracer diffusivity in the rest of the model.", &
01104                 units="nondim", default=1.0)
01105   call get_param(param_file, mdl, "MEKE_USCALE", cs%MEKE_Uscale, &
01106                 "The background velocity that is combined with MEKE to "//&
01107                 "calculate the bottom drag.", units="m s-1", default=0.0, scale=us%m_s_to_L_T)
01108   call get_param(param_file, mdl, "MEKE_GM_SRC_ALT", cs%GM_src_alt, &
01109                 "If true, use the GM energy conversion form S^2*N^2*kappa rather "//&
01110                 "than the streamfunction for the MEKE GM source term.", default=.false.)
01111   call get_param(param_file, mdl, "MEKE_VISC_DRAG", cs%visc_drag, &
01112                 "If true, use the vertvisc_type to calculate the bottom "//&
01113                 "drag acting on MEKE.", default=.true.)
01114   call get_param(param_file, mdl, "MEKE_KHTH_FAC", meke%KhTh_fac, &
01115                 "A factor that maps MEKE%Kh to KhTh.", units="nondim", &
01116                 default=0.0)
01117   call get_param(param_file, mdl, "MEKE_KHTR_FAC", meke%KhTr_fac, &
01118                 "A factor that maps MEKE%Kh to KhTr.", units="nondim", &
01119                 default=0.0)
01120   call get_param(param_file, mdl, "MEKE_KHMEKE_FAC", cs%KhMEKE_Fac, &
01121                 "A factor that maps MEKE%Kh to Kh for MEKE itself.", &
01122                 units="nondim", default=0.0)
01123   call get_param(param_file, mdl, "MEKE_OLD_LSCALE", cs%use_old_lscale, &
01124                 "If true, use the old formula for length scale which is "//&
01125                 "a function of grid spacing and deformation radius.",  &
01126                 default=.false.)
01127   call get_param(param_file, mdl, "MEKE_MIN_LSCALE", cs%use_min_lscale, &
01128                 "If true, use a strict minimum of provided length scales "//&
01129                 "rather than harmonic mean.",  &
01130                 default=.false.)
01131   call get_param(param_file, mdl, "MEKE_RD_MAX_SCALE", cs%Rd_as_max_scale, &
01132                 "If true, the length scale used by MEKE is the minimum of "//&
01133                 "the deformation radius or grid-spacing. Only used if "//&
01134                 "MEKE_OLD_LSCALE=True", units="nondim", default=.false.)
01135   call get_param(param_file, mdl, "MEKE_VISCOSITY_COEFF_KU", cs%viscosity_coeff_Ku, &
01136                 "If non-zero, is the scaling coefficient in the expression for"//&
01137                 "viscosity used to parameterize harmonic lateral momentum mixing by"//&
```

```
01138                 "unresolved eddies represented by MEKE. Can be negative to"//&
01139                 "represent backscatter from the unresolved eddies.", &
01140                 units="nondim", default=0.0)
01141   call get_param(param_file, mdl, "MEKE_VISCOSITY_COEFF_AU", cs%viscosity_coeff_Au, &
01142                 "If non-zero, is the scaling coefficient in the expression for"//&
01143                 "viscosity used to parameterize biharmonic lateral momentum mixing by"//&
01144                 "unresolved eddies represented by MEKE. Can be negative to"//&
01145                 "represent backscatter from the unresolved eddies.", &
01146                 units="nondim", default=0.0)
01147   call get_param(param_file, mdl, "MEKE_FIXED_MIXING_LENGTH", cs%Lfixed, &
01148                 "If positive, is a fixed length contribution to the expression "//&
01149                 "for mixing length used in MEKE-derived diffusivity.", &
01150                 units="m", default=0.0, scale=us%m_to_L)
01151   call get_param(param_file, mdl, "MEKE_ALPHA_DEFORM", cs%aDeform, &
01152                 "If positive, is a coefficient weighting the deformation scale "//&
01153                 "in the expression for mixing length used in MEKE-derived diffusivity.", &
01154                 units="nondim", default=0.0)
01155   call get_param(param_file, mdl, "MEKE_ALPHA_RHINES", cs%aRhines, &
01156                 "If positive, is a coefficient weighting the Rhines scale "//&
01157                 "in the expression for mixing length used in MEKE-derived diffusivity.", &
01158                 units="nondim", default=0.0)
01159   call get_param(param_file, mdl, "MEKE_ALPHA_EADY", cs%aEady, &
01160                 "If positive, is a coefficient weighting the Eady length scale "//&
01161                 "in the expression for mixing length used in MEKE-derived diffusivity.", &
01162                 units="nondim", default=0.0)
01163   call get_param(param_file, mdl, "MEKE_ALPHA_FRICT", cs%aFrict, &
01164                 "If positive, is a coefficient weighting the frictional arrest scale "//&
01165                 "in the expression for mixing length used in MEKE-derived diffusivity.", &
01166                 units="nondim", default=0.0)
01167   call get_param(param_file, mdl, "MEKE_ALPHA_GRID", cs%aGrid, &
01168                 "If positive, is a coefficient weighting the grid-spacing as a scale "//&
01169                 "in the expression for mixing length used in MEKE-derived diffusivity.", &
01170                 units="nondim", default=0.0)
01171   call get_param(param_file, mdl, "MEKE_COLD_START", coldstart, &
01172                 "If true, initialize EKE to zero. Otherwise a local equilibrium solution "//&
01173                 "is used as an initial condition for EKE.", default=.false.)
01174   call get_param(param_file, mdl, "MEKE_BACKSCAT_RO_C", meke%backscatter_Ro_c, &
01175                 "The coefficient in the Rossby number function for scaling the biharmonic "//&
01176                 "frictional energy source. Setting to non-zero enables the Rossby number function.", &
01177                 units="nondim", default=0.0)
01178   call get_param(param_file, mdl, "MEKE_BACKSCAT_RO_POW", meke%backscatter_Ro_pow, &
01179                 "The power in the Rossby number function for scaling the biharmonic "//&
01180                 "frictional energy source.", units="nondim", default=0.0)
01181   call get_param(param_file, mdl, "MEKE_ADVECTION_FACTOR", cs%MEKE_advection_factor, &
01182                 "A scale factor in front of advection of eddy energy. Zero turns advection off. "//&
01183                 "Using unity would be normal but other values could accommodate a mismatch "//&
01184                 "between the advecting barotropic flow and the vertical structure of MEKE.", &
01185                 units="nondim", default=0.0)
01186   call get_param(param_file, mdl, "MEKE_TOPOGRAPHIC_BETA", cs%MEKE_topographic_beta, &
01187                 "A scale factor to determine how much topographic beta is weighed in " //&
01188                 "computing beta in the expression of Rhines scale. Use 1 if full "//&
01189                 "topographic beta effect is considered; use 0 if it's completely ignored.", &
01190                 units="nondim", default=0.0)
01191
01192   ! Nonlocal module parameters
01193   call get_param(param_file, mdl, "CDRAG", cdrag, &
01194                 "CDRAG is the drag coefficient relating the magnitude of "//&
01195                 "the velocity field to the bottom stress.", units="nondim", &
01196                 default=0.003)
01197   call get_param(param_file, mdl, "MEKE_CDRAG", cs%cdrag, &
01198                 "Drag coefficient relating the magnitude of the velocity "//&
01199                 "field to the bottom stress in MEKE.", units="nondim", &
01200                 default=cdrag)
01201   call get_param(param_file, mdl, "LAPLACIAN", laplacian, default=.false., do_not_log=.true.)
01202   call get_param(param_file, mdl, "BIHARMONIC", biharmonic, default=.false., do_not_log=.true.)
01203
01204   if (cs%viscosity_coeff_Ku/=0. .and. .not. laplacian) call mom_error(fatal, &
01205                 "LAPLACIAN must be true if MEKE_VISCOSITY_COEFF_KU is true.")
01206
01207   if (cs%viscosity_coeff_Au/=0. .and. .not. biharmonic) call mom_error(fatal, &
01208                 "BIHARMONIC must be true if MEKE_VISCOSITY_COEFF_AU is true.")
01209
01210   call get_param(param_file, mdl, "DEBUG", cs%debug, default=.false., do_not_log=.true.)
01211
01212   ! Identify if any lateral diffusive processes are active
01213   cs%kh_flux_enabled = .false.
01214   if ((cs%MEKE_KH >= 0.0)  .or. (cs%KhMEKE_FAC > 0.0) .or. (cs%MEKE_advection_factor > 0.0)) &
01215     cs%kh_flux_enabled = .true.
01216
01217 ! Register fields for output from this module.
01218   cs%diag => diag
01219   cs%id_MEKE = register_diag_field('ocean_model', 'MEKE', diag%axesT1, time, &
01220      'Mesoscale Eddy Kinetic Energy', 'm2 s-2', conversion=us%L_T_to_m_s**2)
01221   if (.not. associated(meke%MEKE)) cs%id_MEKE = -1
01222   cs%id_Kh = register_diag_field('ocean_model', 'MEKE_KH', diag%axesT1, time, &
01223      'MEKE derived diffusivity', 'm2 s-1', conversion=us%L_to_m**2*us%s_to_T)
01224   if (.not. associated(meke%Kh)) cs%id_Kh = -1
```

```
01225   cs%id_Ku = register_diag_field('ocean_model', 'MEKE_KU', diag%axesT1, time, &
01226      'MEKE derived lateral viscosity', 'm2 s-1', conversion=us%L_to_m**2*us%s_to_T)
01227   if (.not. associated(meke%Ku)) cs%id_Ku = -1
01228   cs%id_Au = register_diag_field('ocean_model', 'MEKE_AU', diag%axesT1, time, &
01229      'MEKE derived lateral biharmonic viscosity', 'm4 s-1', conversion=us%L_to_m**4*us%s_to_T)
01230   if (.not. associated(meke%Au)) cs%id_Au = -1
01231   cs%id_Ue = register_diag_field('ocean_model', 'MEKE_Ue', diag%axesT1, time, &
01232      'MEKE derived eddy-velocity scale', 'm s-1', conversion=us%L_T_to_m_s)
01233   if (.not. associated(meke%MEKE)) cs%id_Ue = -1
01234   cs%id_Ub = register_diag_field('ocean_model', 'MEKE_Ub', diag%axesT1, time, &
01235      'MEKE derived bottom eddy-velocity scale', 'm s-1', conversion=us%L_T_to_m_s)
01236   if (.not. associated(meke%MEKE)) cs%id_Ub = -1
01237   cs%id_Ut = register_diag_field('ocean_model', 'MEKE_Ut', diag%axesT1, time, &
01238      'MEKE derived barotropic eddy-velocity scale', 'm s-1', conversion=us%L_T_to_m_s)
01239   if (.not. associated(meke%MEKE)) cs%id_Ut = -1
01240   cs%id_src = register_diag_field('ocean_model', 'MEKE_src', diag%axesT1, time, &
01241      'MEKE energy source', 'm2 s-3', conversion=(us%L_T_to_m_s**2)*us%s_to_T)
01242   cs%id_decay = register_diag_field('ocean_model', 'MEKE_decay', diag%axesT1, time, &
01243      'MEKE decay rate', 's-1', conversion=us%s_to_T)
01244   cs%id_GM_src = register_diag_field('ocean_model', 'MEKE_GM_src', diag%axesT1, time, &
01245      'MEKE energy available from thickness mixing', &
01246      'W m-2', conversion=us%RZ3_T3_to_W_m2*us%L_to_Z**2)
01247   if (.not. associated(meke%GM_src)) cs%id_GM_src = -1
01248   cs%id_mom_src = register_diag_field('ocean_model', 'MEKE_mom_src',diag%axesT1, time, &
01249      'MEKE energy available from momentum', &
01250      'W m-2', conversion=us%RZ3_T3_to_W_m2*us%L_to_Z**2)
01251   if (.not. associated(meke%mom_src)) cs%id_mom_src = -1
01252   cs%id_GME_snk = register_diag_field('ocean_model', 'MEKE_GME_snk',diag%axesT1, time, &
01253      'MEKE energy lost to GME backscatter', &
01254      'W m-2', conversion=us%RZ3_T3_to_W_m2*us%L_to_Z**2)
01255   if (.not. associated(meke%GME_snk)) cs%id_GME_snk = -1
01256   cs%id_Le = register_diag_field('ocean_model', 'MEKE_Le', diag%axesT1, time, &
01257      'Eddy mixing length used in the MEKE derived eddy diffusivity', 'm', conversion=us%L_to_m)
01258   cs%id_Lrhines = register_diag_field('ocean_model', 'MEKE_Lrhines', diag%axesT1, time, &
01259      'Rhines length scale used in the MEKE derived eddy diffusivity', 'm', conversion=us%L_to_m)
01260   cs%id_Leady = register_diag_field('ocean_model', 'MEKE_Leady', diag%axesT1, time, &
01261      'Eady length scale used in the MEKE derived eddy diffusivity', 'm', conversion=us%L_to_m)
01262   cs%id_gamma_b = register_diag_field('ocean_model', 'MEKE_gamma_b', diag%axesT1, time, &
01263      'Ratio of bottom-projected eddy velocity to column-mean eddy velocity', 'nondim')
01264   cs%id_gamma_t = register_diag_field('ocean_model', 'MEKE_gamma_t', diag%axesT1, time, &
01265      'Ratio of barotropic eddy velocity to column-mean eddy velocity', 'nondim')
01266
01267   if (cs%kh_flux_enabled) then
01268     cs%id_KhMEKE_u = register_diag_field('ocean_model', 'KHMEKE_u', diag%axesCu1, time, &
01269      'Zonal diffusivity of MEKE', 'm2 s-1', conversion=us%L_to_m**2*us%s_to_T)
01270     cs%id_KhMEKE_v = register_diag_field('ocean_model', 'KHMEKE_v', diag%axesCv1, time, &
01271      'Meridional diffusivity of MEKE', 'm2 s-1', conversion=us%L_to_m**2*us%s_to_T)
01272   endif
01273
01274   if (cs%MEKE_equilibrium_restoring) then
01275     cs%id_MEKE_equilibrium = register_diag_field('ocean_model', 'MEKE_equilibrium', diag%axesT1, time, &
01276      'Equilibrated Mesoscale Eddy Kinetic Energy', 'm2 s-2', conversion=us%L_T_to_m_s**2)
01277   endif
01278
01279   cs%id_clock_pass = cpu_clock_id('(Ocean continuity halo updates)', grain=clock_routine)
01280
01281   ! Detect whether this instance of MEKE_init() is at the beginning of a run
01282   ! or after a restart. If at the beginning, we will initialize MEKE to a local
01283   ! equilibrium.
01284   cs%initialize = .not.query_initialized(meke%MEKE, "MEKE", restart_cs)
01285   if (coldstart) cs%initialize = .false.
01286   if (cs%initialize) call mom_error(warning, &
01287                        "MEKE_init: Initializing MEKE with a local equilibrium balance.")
01288
01289   ! Account for possible changes in dimensional scaling for variables that have been
01290   ! read from a restart file.
01291   i_t_rescale = 1.0
01292   if ((us%s_to_T_restart /= 0.0) .and. (us%s_to_T_restart /= us%s_to_T)) &
01293     i_t_rescale = us%s_to_T_restart / us%s_to_T
01294   l_rescale = 1.0
01295   if ((us%m_to_L_restart /= 0.0) .and. (us%m_to_L_restart /= us%m_to_L)) &
01296     l_rescale = us%m_to_L / us%m_to_L_restart
01297
01298   if (l_rescale*i_t_rescale /= 1.0) then
01299     if (associated(meke%MEKE)) then ; if (query_initialized(meke%MEKE, "MEKE_MEKE", restart_cs)) then
01300       do j=js,je ; do i=is,ie
01301         meke%MEKE(i,j) = l_rescale*i_t_rescale * meke%MEKE(i,j)
01302       enddo ; enddo
01303     endif ; endif
01304   endif
01305   if (l_rescale**2*i_t_rescale /= 1.0) then
01306     if (associated(meke%Kh)) then ; if (query_initialized(meke%Kh, "MEKE_Kh", restart_cs)) then
01307       do j=js,je ; do i=is,ie
01308         meke%Kh(i,j) = l_rescale**2*i_t_rescale * meke%Kh(i,j)
01309       enddo ; enddo
01310     endif ; endif
01311     if (associated(meke%Ku)) then ; if (query_initialized(meke%Ku, "MEKE_Ku", restart_cs)) then
```

```
01312       do j=js,je ; do i=is,ie
01313         meke%Ku(i,j) = l_rescale**2*i_t_rescale * meke%Ku(i,j)
01314       enddo ; enddo
01315     endif ; endif
01316     if (associated(meke%Kh_diff)) then ; if (query_initialized(meke%Kh, "MEKE_Kh_diff", restart_cs)) then
01317       do j=js,je ; do i=is,ie
01318         meke%Kh_diff(i,j) = l_rescale**2*i_t_rescale * meke%Kh_diff(i,j)
01319       enddo ; enddo
01320     endif ; endif
01321   endif
01322   if (l_rescale**4*i_t_rescale /= 1.0) then
01323     if (associated(meke%Au)) then ; if (query_initialized(meke%Au, "MEKE_Au", restart_cs)) then
01324       do j=js,je ; do i=is,ie
01325         meke%Au(i,j) = l_rescale**4*i_t_rescale * meke%Au(i,j)
01326       enddo ; enddo
01327     endif ; endif
01328   endif
01329
01330   ! Set up group passes.  In the case of a restart, these fields need a halo update now.
01331   if (associated(meke%MEKE)) then
01332     call create_group_pass(cs%pass_MEKE, meke%MEKE, g%Domain)
01333     if (associated(meke%Kh_diff)) call create_group_pass(cs%pass_MEKE, meke%Kh_diff, g%Domain)
01334     if (.not.cs%initialize) call do_group_pass(cs%pass_MEKE, g%Domain)
01335   endif
01336   if (associated(meke%Kh)) call create_group_pass(cs%pass_Kh, meke%Kh, g%Domain)
01337   if (associated(meke%Ku)) call create_group_pass(cs%pass_Kh, meke%Ku, g%Domain)
01338   if (associated(meke%Au)) call create_group_pass(cs%pass_Kh, meke%Au, g%Domain)
01339
01340   if (associated(meke%Kh) .or. associated(meke%Ku) .or. associated(meke%Au)) &
01341     call do_group_pass(cs%pass_Kh, g%Domain)
01342
```

42.5.3.6 meke_lengthscales()

```
subroutine mom_meke::meke_lengthscales (
             type(meke_cs), pointer CS,
             type(meke_type), pointer MEKE,
             type(ocean_grid_type), intent(inout) G,
             type(verticalgrid_type), intent(in) GV,
             type(unit_scale_type), intent(in) US,
             real, dimension( g %isdb:  g %iedb, g %jsd:  g %jed), intent(in) SN_u,
             real, dimension( g %isd:  g %ied, g %jsdb:  g %jedb), intent(in) SN_v,
             real, dimension( g %isd:  g %ied, g %jsd:  g %jed), intent(in) EKE,
             real, dimension( g %isd:  g %ied, g %jsd:  g %jed), intent(out) bottomFac2,
             real, dimension( g %isd:  g %ied, g %jsd:  g %jed), intent(out) barotrFac2,
             real, dimension( g %isd:  g %ied, g %jsd:  g %jed), intent(out) LmixScale )  [private]
```

Calculates the eddy mixing length scale and γ_b and γ_t functions that are ratios of either bottom or barotropic eddy energy to the column eddy energy, respectively. See MEKE equations.

Parameters

	cs	MEKE control structure.
	meke	MEKE data.
in,out	*g*	Ocean grid.
in	*gv*	Ocean vertical grid structure.
in	*us*	A dimensional unit scaling type
in	*sn_u*	Eady growth rate at u-points [T-1 ~> s-1].
in	*sn_v*	Eady growth rate at v-points [T-1 ~> s-1].
in	*eke*	Eddy kinetic energy [L2 T-2 ~> m2 s-2].
out	*bottomfac2*	gamma_b^2
out	*barotrfac2*	gamma_t^2
out	*lmixscale*	Eddy mixing length [L ~> m].

Definition at line 839 of file MOM_MEKE.F90.

References meke_lengthscales_0d().

Referenced by step_forward_meke().

```
00839   type(meke_cs),                    pointer       :: cs   !< MEKE control structure.
00840   type(meke_type),                  pointer       :: meke !< MEKE data.
00841   type(ocean_grid_type),            intent(inout) :: g    !< Ocean grid.
00842   type(verticalgrid_type),          intent(in)    :: gv   !< Ocean vertical grid structure.
00843   type(unit_scale_type),            intent(in)    :: us   !< A dimensional unit scaling type
00844   real, dimension(SZIB_(G),SZJ_(G)), intent(in)    :: sn_u !< Eady growth rate at u-points [T-1 ~> s-1].
00845   real, dimension(SZI_(G),SZJB_(G)), intent(in)    :: sn_v !< Eady growth rate at v-points [T-1 ~> s-1].
00846   real, dimension(SZI_(G),SZJ_(G)),  intent(in)    :: eke  !< Eddy kinetic energy [L2 T-2 ~> m2 s-2].
00847   real, dimension(SZI_(G),SZJ_(G)),  intent(out)   :: bottomfac2 !< gamma_b^2
00848   real, dimension(SZI_(G),SZJ_(G)),  intent(out)   :: barotrfac2 !< gamma_t^2
00849   real, dimension(SZI_(G),SZJ_(G)),  intent(out)   :: lmixscale !< Eddy mixing length [L ~> m].
00850   ! Local variables
00851   real, dimension(SZI_(G),SZJ_(G)) :: lrhines, leady  ! Possible mixing length scales [L ~> m]
00852   real :: beta ! Combined topograpic and planetary vorticity gradient [T-1 L-1 ~> s-1 m-1]
00853   real :: sn   ! The local Eady growth rate [T-1 ~> s-1]
00854   real :: fath ! Coriolis parameter at h points [T-1 ~> s-1]
00855   real :: beta_topo_x, beta_topo_y  ! Topographic PV gradients in x and y [T-1 L-1 ~> s-1 m-1]
00856   integer :: i, j, is, ie, js, je
00857
00858   is = g%isc ; ie = g%iec ; js = g%jsc ; je = g%jec
00859
00860 !$OMP do
00861   do j=js,je ; do i=is,ie
00862     if (.not.cs%use_old_lscale) then
00863       if (cs%aEady > 0.) then
00864         sn = 0.25 * ( (sn_u(i,j) + sn_u(i-1,j)) + (sn_v(i,j) + sn_v(i,j-1)) )
00865       else
00866         sn = 0.
00867       endif
00868       fath = 0.25* ( ( g%CoriolisBu(i,j) + g%CoriolisBu(i-1,j-1) ) + &
00869                      ( g%CoriolisBu(i-1,j) + g%CoriolisBu(i,j-1) ) )  ! Coriolis parameter at h points
00870
00871       ! If bathyT is zero, then a division by zero FPE will be raised.  In this
00872       ! case, we apply Adcroft's rule of reciprocals and set the term to zero.
00873       ! Since zero-bathymetry cells are masked, this should not affect values.
00874       if (cs%MEKE_topographic_beta == 0. .or. g%bathyT(i,j) == 0.0) then
00875         beta_topo_x = 0. ; beta_topo_y = 0.
00876       else
00877         !### Consider different combinations of these estimates of topographic beta, and the use
00878         !    of the water column thickness instead of the bathymetric depth.
00879         beta_topo_x = -cs%MEKE_topographic_beta * fath * 0.5 * ( &
00880                       (g%bathyT(i+1,j)-g%bathyT(i,j)) * g%IdxCu(i,j)  &
00881                  / max(g%bathyT(i+1,j),g%bathyT(i,j), gv%H_subroundoff) &
00882                +       (g%bathyT(i,j)-g%bathyT(i-1,j)) * g%IdxCu(i-1,j) &
00883                  / max(g%bathyT(i,j),g%bathyT(i-1,j), gv%H_subroundoff) )
00884         beta_topo_y = -cs%MEKE_topographic_beta * fath * 0.5 * ( &
00885                       (g%bathyT(i,j+1)-g%bathyT(i,j)) * g%IdyCv(i,j)  &
00886                  / max(g%bathyT(i,j+1),g%bathyT(i,j), gv%H_subroundoff) + &
00887                       (g%bathyT(i,j)-g%bathyT(i,j-1)) * g%IdyCv(i,j-1) &
00888                  / max(g%bathyT(i,j),g%bathyT(i,j-1), gv%H_subroundoff) )
00889       endif
00890       beta =  sqrt((g%dF_dx(i,j) + beta_topo_x)**2 + &
00891                    (g%dF_dy(i,j) + beta_topo_y)**2 )
00892
00893     else
00894       beta = 0.
00895     endif
00896     ! Returns bottomFac2, barotrFac2 and LmixScale
00897     call meke_lengthscales_0d(cs, us, g%areaT(i,j), beta, g%bathyT(i,j),  &
00898                               meke%Rd_dx_h(i,j), sn, meke%MEKE(i,j), &
00899                               bottomfac2(i,j), barotrfac2(i,j), lmixscale(i,j), &
00900                               lrhines(i,j), leady(i,j))
00901   enddo ; enddo
00902   if (cs%id_Lrhines>0) call post_data(cs%id_LRhines, lrhines, cs%diag)
00903   if (cs%id_Leady>0) call post_data(cs%id_LEady, leady, cs%diag)
00904
```

42.5.3.7 meke_lengthscales_0d()

```
subroutine mom_meke::meke_lengthscales_0d (
            type(meke_cs), pointer CS,
            type(unit_scale_type), intent(in) US,
            real, intent(in) area,
            real, intent(in) beta,
            real, intent(in) depth,
            real, intent(in) Rd_dx,
            real, intent(in) SN,
            real, intent(in) EKE,
            real, intent(out) bottomFac2,
            real, intent(out) barotrFac2,
            real, intent(out) LmixScale,
            real, intent(out) Lrhines,
            real, intent(out) Leady )  [private]
```

Calculates the eddy mixing length scale and γ_b and γ_t functions that are ratios of either bottom or barotropic eddy energy to the column eddy energy, respectively. See MEKE equations.

Parameters

	cs	MEKE control structure.
in	*us*	A dimensional unit scaling type
in	*area*	Grid cell area [L2 ~> m2]
in	*beta*	Planetary beta = |grad F| [T-1 L-1 ~> s-1 m-1]
in	*depth*	Ocean depth [Z ~> m]
in	*rd_dx*	Resolution Ld/dx [nondim].
in	*sn*	Eady growth rate [T-1 ~> s-1].
in	*eke*	Eddy kinetic energy [L2 T-2 ~> m2 s-2].
out	*bottomfac2*	gamma_b^2
out	*barotrfac2*	gamma_t^2
out	*lmixscale*	Eddy mixing length [L ~> m].
out	*lrhines*	Rhines length scale [L ~> m].
out	*leady*	Eady length scale [L ~> m].

Definition at line 912 of file MOM_MEKE.F90.

Referenced by meke_equilibrium(), and meke_lengthscales().

```
00912   type(meke_cs), pointer       :: cs         !< MEKE control structure.
00913   type(unit_scale_type), intent(in) :: us    !< A dimensional unit scaling type
00914   real,          intent(in)    :: area       !< Grid cell area [L2 ~> m2]
00915   real,          intent(in)    :: beta       !< Planetary beta = |grad F| [T-1 L-1 ~> s-1 m-1]
00916   real,          intent(in)    :: depth      !< Ocean depth [Z ~> m]
00917   real,          intent(in)    :: rd_dx      !< Resolution Ld/dx [nondim].
00918   real,          intent(in)    :: sn         !< Eady growth rate [T-1 ~> s-1].
00919   real,          intent(in)    :: eke        !< Eddy kinetic energy [L2 T-2 ~> m2 s-2].
00920 !  real,          intent(in)    :: Z_to_L     !< A conversion factor from depth units (Z) to
00921 !                                             !! the units for lateral distances (L).
00922   real,          intent(out)   :: bottomfac2 !< gamma_b^2
00923   real,          intent(out)   :: barotrfac2 !< gamma_t^2
00924   real,          intent(out)   :: lmixscale  !< Eddy mixing length [L ~> m].
00925   real,          intent(out)   :: lrhines    !< Rhines length scale [L ~> m].
00926   real,          intent(out)   :: leady      !< Eady length scale [L ~> m].
00927   ! Local variables
00928   real :: lgrid, ldeform, lfrict ! Length scales [L ~> m]
00929   real :: ue  ! An eddy velocity [L T-1 ~> m s-1]
00930
00931   ! Length scale for MEKE derived diffusivity
```

```
00932   lgrid = sqrt(area)                ! Grid scale
00933   ldeform = lgrid * rd_dx           ! Deformation scale
00934   lfrict = (us%Z_to_L * depth) / cs%cdrag  ! Frictional arrest scale
00935   ! gamma_b^2 is the ratio of bottom eddy energy to mean column eddy energy
00936   ! used in calculating bottom drag
00937   bottomfac2 = cs%MEKE_CD_SCALE**2
00938   if (lfrict*cs%MEKE_Cb>0.) bottomfac2 = bottomfac2 + 1./( 1. + cs%MEKE_Cb*(ldeform/lfrict) )**0.8
00939   bottomfac2 = max(bottomfac2, cs%MEKE_min_gamma)
00940   ! gamma_t^2 is the ratio of barotropic eddy energy to mean column eddy energy
00941   ! used in the velocity scale for diffusivity
00942   barotrfac2 = 1.
00943   if (lfrict*cs%MEKE_Ct>0.) barotrfac2 = 1. / ( 1. + cs%MEKE_Ct*(ldeform/lfrict) )**0.25
00944   barotrfac2 = max(barotrfac2, cs%MEKE_min_gamma)
00945   if (cs%use_old_lscale) then
00946     if (cs%Rd_as_max_scale) then
00947       lmixscale = min(ldeform, lgrid) ! The smaller of Ld or dx
00948     else
00949       lmixscale = lgrid
00950     endif
00951   else
00952     ue = sqrt( 2.0 * max( 0., barotrfac2*eke ) ) ! Barotropic eddy flow scale
00953     lrhines = sqrt( ue / max( beta, 1.e-30*us%T_to_s*us%L_to_m ) )       ! Rhines scale
00954     if (cs%aEady > 0.) then
00955       leady = ue / max( sn, 1.e-15*us%T_to_s ) ! Bound Eady time-scale < 1e15 seconds
00956     else
00957       leady = 0.
00958     endif
00959     if (cs%use_min_lscale) then
00960       lmixscale = 1.e7
00961       if (cs%aDeform*ldeform > 0.) lmixscale = min(lmixscale,cs%aDeform*ldeform)
00962       if (cs%aFrict *lfrict  > 0.) lmixscale = min(lmixscale,cs%aFrict *lfrict)
00963       if (cs%aRhines*lrhines > 0.) lmixscale = min(lmixscale,cs%aRhines*lrhines)
00964       if (cs%aEady  *leady   > 0.) lmixscale = min(lmixscale,cs%aEady  *leady)
00965       if (cs%aGrid  *lgrid   > 0.) lmixscale = min(lmixscale,cs%aGrid  *lgrid)
00966       if (cs%Lfixed          > 0.) lmixscale = min(lmixscale,cs%Lfixed)
00967     else
00968       lmixscale = 0.
00969       if (cs%aDeform*ldeform > 0.) lmixscale = lmixscale + 1./(cs%aDeform*ldeform)
00970       if (cs%aFrict *lfrict  > 0.) lmixscale = lmixscale + 1./(cs%aFrict *lfrict)
00971       if (cs%aRhines*lrhines > 0.) lmixscale = lmixscale + 1./(cs%aRhines*lrhines)
00972       if (cs%aEady  *leady   > 0.) lmixscale = lmixscale + 1./(cs%aEady  *leady)
00973       if (cs%aGrid  *lgrid   > 0.) lmixscale = lmixscale + 1./(cs%aGrid  *lgrid)
00974       if (cs%Lfixed          > 0.) lmixscale = lmixscale + 1./cs%Lfixed
00975       if (lmixscale > 0.) lmixscale = 1. / lmixscale
00976     endif
00977   endif
00978
```

42.5.3.8 step_forward_meke()

```
subroutine, public mom_meke::step_forward_meke (
             type(meke_type), pointer MEKE,
             real, dimension( g %isd:  g %ied, g %jsd:  g %jed, g %ke), intent(in) h,
             real, dimension( g %isdb:  g %iedb, g %jsd:  g %jed), intent(in) SN_u,
             real, dimension( g %isd:  g %ied, g %jsdb:  g %jedb), intent(in) SN_v,
             type(vertvisc_type), intent(in) visc,
             real, intent(in) dt,
             type(ocean_grid_type), intent(inout) G,
             type(verticalgrid_type), intent(in) GV,
             type(unit_scale_type), intent(in) US,
             type(meke_cs), pointer CS,
             real, dimension( g %isdb:  g %iedb, g %jsd:  g %jed, g %ke), intent(in) hu,
             real, dimension( g %isd:  g %ied, g %jsdb:  g %jedb, g %ke), intent(in) hv )
```

Integrates forward-in-time the MEKE eddy energy equation. See MEKE equations.

Parameters

	meke	MEKE data.
in,out	*g*	Ocean grid.
in	*gv*	Ocean vertical grid structure.
in	*us*	A dimensional unit scaling type
in	*h*	Layer thickness [H ~> m or kg m-2].
in	*sn↩ _u*	Eady growth rate at u-points [T-1 ~> s-1].
in	*sn↩ _v*	Eady growth rate at v-points [T-1 ~> s-1].
in	*visc*	The vertical viscosity type.
in	*dt*	Model(baroclinic) time-step [T ~> s].
	cs	MEKE control structure.
in	*hu*	Accumlated zonal mass flux [H L2 ~> m3 or kg].
in	*hv*	Accumlated meridional mass flux [H L2 ~> m3 or kg]

Definition at line 112 of file MOM_MEKE.F90.

References meke_equilibrium(), meke_equilibrium_restoring(), and meke_lengthscales().

```
00112   type(meke_type),                          pointer       :: meke !< MEKE data.
00113   type(ocean_grid_type),                    intent(inout) :: g    !< Ocean grid.
00114   type(verticalgrid_type),                  intent(in)    :: gv   !< Ocean vertical grid structure.
00115   type(unit_scale_type),                    intent(in)    :: us   !< A dimensional unit scaling type
00116   real, dimension(SZI_(G),SZJ_(G),SZK_(G)), intent(in)    :: h    !< Layer thickness [H ~> m or kg m-2].
00117   real, dimension(SZIB_(G),SZJ_(G)),        intent(in)    :: sn_u !< Eady growth rate at u-points [T-1 ~>
      s-1].
00118   real, dimension(SZI_(G),SZJB_(G)),        intent(in)    :: sn_v !< Eady growth rate at v-points [T-1 ~>
      s-1].
00119   type(vertvisc_type),                      intent(in)    :: visc !< The vertical viscosity type.
00120   real,                                     intent(in)    :: dt   !< Model(baroclinic) time-step [T ~> s].
00121   type(meke_cs),                            pointer       :: cs   !< MEKE control structure.
00122   real, dimension(SZIB_(G),SZJ_(G),SZK_(G)), intent(in)   :: hu   !< Accumlated zonal mass flux [H L2 ~> m3
      or kg].
00123   real, dimension(SZI_(G),SZJB_(G),SZK_(G)), intent(in)   :: hv   !< Accumlated meridional mass flux [H L2
      ~> m3 or kg]
00124
00125   ! Local variables
00126   real, dimension(SZI_(G),SZJ_(G)) :: &
00127     mass, &          ! The total mass of the water column [R Z ~> kg m-2].
00128     i_mass, &        ! The inverse of mass [R-1 Z-1 ~> m2 kg-1].
00129     src, &           ! The sum of all MEKE sources [L2 T-3 ~> W kg-1] (= m2 s-3).
00130     meke_decay, &    ! A diagnostic of the MEKE decay timescale [T-1 ~> s-1].
00131     drag_rate_visc, & ! Near-bottom velocity contribution to bottom dratg [L T-1 ~> m s-1]
00132     drag_rate, &     ! The MEKE spindown timescale due to bottom drag [T-1 ~> s-1].
00133     drag_rate_j15, &  ! The MEKE spindown timescale due to bottom drag with the Jansen 2015 scheme.
00134                      ! Unfortunately, as written the units seem inconsistent. [T-1 ~> s-1].
00135     del2meke, &      ! Laplacian of MEKE, used for bi-harmonic diffusion [T-2 ~> s-2].
00136     del4meke, &      ! Time-integrated MEKE tendency arising from the biharmonic of MEKE [L2 T-2 ~> m2 s-2].
00137     lmixscale, &     ! Eddy mixing length [L ~> m].
00138     barotrfac2, &    ! Ratio of EKE_barotropic / EKE [nondim]
00139     bottomfac2, &    ! Ratio of EKE_bottom / EKE [nondim]
00140     tmp              ! Temporary variable for diagnostic computation
00141
00142   real, dimension(SZIB_(G),SZJ_(G)) :: &
00143     meke_uflux, &    ! The zonal advective and diffusive flux of MEKE with units of [R Z L4 T-3 ~> kg m-2
      s-3].
00144                      ! In one place, MEKE_uflux is used as temporary work space with units of [L2 T-2 ~> m2
      s-2].
00145     kh_u, &          ! The zonal diffusivity that is actually used [L2 T-1 ~> m2 s-1].
00146     barohu, &        ! Depth integrated accumulated zonal mass flux [R Z L2 ~> kg].
00147     drag_vel_u       ! A (vertical) viscosity associated with bottom drag at
00148                      ! u-points [Z T-1 ~> m s-1].
00149   real, dimension(SZI_(G),SZJB_(G)) :: &
00150     meke_vflux, &    ! The meridional advective and diffusive flux of MEKE with units of [R Z L4 T-3 ~> kg
      m-2 s-3].
00151                      ! In one place, MEKE_vflux is used as temporary work space with units of [L2 T-2 ~> m2
      s-2].
00152     kh_v, &          ! The meridional diffusivity that is actually used [L2 T-1 ~> m2 s-1].
00153     barohv, &        ! Depth integrated accumulated meridional mass flux [R Z L2 ~> kg].
```

```
00154     drag_vel_v      ! A (vertical) viscosity associated with bottom drag at
00155                     ! v-points [Z T-1 ~> m s-1].
00156   real :: kh_here   ! The local horizontal viscosity [L2 T-1 ~> m2 s-1]
00157   real :: inv_kh_max ! The inverse of the local horizontal viscosity [T L-2 ~> s m-2]
00158   real :: k4_here   ! The local horizontal biharmonic viscosity [L4 T-1 ~> m4 s-1]
00159   real :: inv_k4_max ! The inverse of the local horizontal biharmonic viscosity [T L-4 ~> s m-4]
00160   real :: cdrag2
00161   real :: advfac    ! The product of the advection scaling factor and 1/dt [T-1 ~> s-1]
00162   real :: mass_neglect ! A negligible mass [R Z ~> kg m-2].
00163   real :: ldamping  ! The MEKE damping rate [T-1 ~> s-1].
00164   real :: rho0      ! A density used to convert mass to distance [R ~> kg m-3].
00165   real :: sdt       ! dt to use locally [T ~> s] (could be scaled to accelerate)
00166   real :: sdt_damp  ! dt for damping [T ~> s] (sdt could be split).
00167   logical :: use_drag_rate ! Flag to indicate drag_rate is finite
00168   integer :: i, j, k, is, ie, js, je, isq, ieq, jsq, jeq, nz
00169
00170   is = g%isc ; ie = g%iec ; js = g%jsc ; je = g%jec ; nz = g%ke
00171   isq = g%IscB ; ieq = g%IecB ; jsq = g%JscB ; jeq = g%JecB
00172
00173   if (.not.associated(cs)) call mom_error(fatal, &
00174         "MOM_MEKE: Module must be initialized before it is used.")
00175   if (.not.associated(meke)) call mom_error(fatal, &
00176         "MOM_MEKE: MEKE must be initialized before it is used.")
00177
00178   if ((cs%MEKE_damping > 0.0) .or. (cs%MEKE_Cd_scale > 0.0) .or. (cs%MEKE_Cb>0.) &
00179       .or. cs%visc_drag) then
00180     use_drag_rate = .true.
00181   else
00182     use_drag_rate = .false.
00183   endif
00184
00185   ! Only integrate the MEKE equations if MEKE is required.
00186   if (.not.associated(meke%MEKE)) then
00187 !   call MOM_error(FATAL, "MOM_MEKE: MEKE%MEKE is not associated!")
00188     return
00189   endif
00190
00191     if (cs%debug) then
00192       if (associated(meke%mom_src)) &
00193         call hchksum(meke%mom_src, 'MEKE mom_src', g%HI, scale=us%RZ3_T3_to_W_m2*us%L_to_Z**2)
00194       if (associated(meke%GME_snk)) &
00195         call hchksum(meke%GME_snk, 'MEKE GME_snk', g%HI, scale=us%RZ3_T3_to_W_m2*us%L_to_Z**2)
00196       if (associated(meke%GM_src)) &
00197         call hchksum(meke%GM_src, 'MEKE GM_src', g%HI, scale=us%RZ3_T3_to_W_m2*us%L_to_Z**2)
00198       if (associated(meke%MEKE)) call hchksum(meke%MEKE, 'MEKE MEKE', g%HI, scale=us%L_T_to_m_s**2)
00199       call uvchksum("MEKE SN_[uv]", sn_u, sn_v, g%HI, scale=us%s_to_T, &
00200                     scalar_pair=.true.)
00201       call uvchksum("MEKE h[uv]", hu, hv, g%HI, haloshift=1, &
00202                     scale=gv%H_to_m*(us%L_to_m**2))
00203     endif
00204
00205     sdt = dt*cs%MEKE_dtScale ! Scaled dt to use for time-stepping
00206     rho0 = gv%Rho0
00207     mass_neglect = gv%H_to_RZ * gv%H_subroundoff
00208     cdrag2 = cs%cdrag**2
00209
00210     ! With a depth-dependent (and possibly strong) damping, it seems
00211     ! advisable to use Strang splitting between the damping and diffusion.
00212     sdt_damp = sdt ; if (cs%MEKE_KH >= 0.0 .or. cs%MEKE_K4 >= 0.) sdt_damp = 0.5*sdt
00213
00214     ! Calculate depth integrated mass exchange if doing advection [R Z L2 ~> kg]
00215     if (cs%MEKE_advection_factor>0.) then
00216       do j=js,je ; do i=is-1,ie
00217         barohu(i,j) = 0.
00218       enddo ; enddo
00219       do k=1,nz
00220         do j=js,je ; do i=is-1,ie
00221           barohu(i,j) = hu(i,j,k) * gv%H_to_RZ
00222         enddo ; enddo
00223       enddo
00224       do j=js-1,je ; do i=is,ie
00225         barohv(i,j) = 0.
00226       enddo ; enddo
00227       do k=1,nz
00228         do j=js-1,je ; do i=is,ie
00229           barohv(i,j) = hv(i,j,k) * gv%H_to_RZ
00230         enddo ; enddo
00231       enddo
00232     endif
00233
00234     if (cs%MEKE_Cd_scale == 0.0 .and. .not. cs%visc_drag) then
00235       !$OMP parallel do default(shared) private(ldamping)
00236       do j=js,je ; do i=is,ie
00237         drag_rate(i,j) = 0. ; drag_rate_j15(i,j) = 0.
00238       enddo ; enddo
00239     endif
00240
```

```
00241     ! Calculate drag_rate_visc(i,j) which accounts for the model bottom mean flow
00242     if (cs%visc_drag) then
00243       !$OMP parallel do default(shared)
00244       do j=js,je ; do i=is-1,ie
00245         drag_vel_u(i,j) = 0.0
00246         if ((g%mask2dCu(i,j) > 0.0) .and. (visc%bbl_thick_u(i,j) > 0.0)) &
00247           drag_vel_u(i,j) = visc%Kv_bbl_u(i,j) / visc%bbl_thick_u(i,j)
00248       enddo ; enddo
00249       !$OMP parallel do default(shared)
00250       do j=js-1,je ; do i=is,ie
00251         drag_vel_v(i,j) = 0.0
00252         if ((g%mask2dCv(i,j) > 0.0) .and. (visc%bbl_thick_v(i,j) > 0.0)) &
00253           drag_vel_v(i,j) = visc%Kv_bbl_v(i,j) / visc%bbl_thick_v(i,j)
00254       enddo ; enddo
00255
00256       !$OMP parallel do default(shared)
00257       do j=js,je ; do i=is,ie
00258         drag_rate_visc(i,j) = (0.25*g%IareaT(i,j) * us%Z_to_L * &
00259                 ((g%areaCu(i-1,j)*drag_vel_u(i-1,j) + &
00260                   g%areaCu(i,j)*drag_vel_u(i,j)) + &
00261                  (g%areaCv(i,j-1)*drag_vel_v(i,j-1) + &
00262                   g%areaCv(i,j)*drag_vel_v(i,j)) ) )
00263       enddo ; enddo
00264     else
00265       !$OMP parallel do default(shared)
00266       do j=js,je ; do i=is,ie
00267         drag_rate_visc(i,j) = 0.
00268       enddo ; enddo
00269     endif
00270
00271     !$OMP parallel do default(shared)
00272     do j=js-1,je+1
00273       do i=is-1,ie+1 ; mass(i,j) = 0.0 ; enddo
00274       do k=1,nz ; do i=is-1,ie+1
00275         mass(i,j) = mass(i,j) + g%mask2dT(i,j) * (gv%H_to_RZ * h(i,j,k)) ! [R Z ~> kg m-2]
00276       enddo ; enddo
00277       do i=is-1,ie+1
00278         i_mass(i,j) = 0.0
00279         if (mass(i,j) > 0.0) i_mass(i,j) = 1.0 / mass(i,j) ! [R-1 Z-1 ~> m2 kg-1]
00280       enddo
00281     enddo
00282
00283     if (cs%initialize) then
00284       call meke_equilibrium(cs, meke, g, gv, us, sn_u, sn_v, drag_rate_visc, i_mass)
00285       cs%initialize = .false.
00286     endif
00287
00288     ! Calculates bottomFac2, barotrFac2 and LmixScale
00289     call meke_lengthscales(cs, meke, g, gv, us, sn_u, sn_v, meke%MEKE, bottomfac2, barotrfac2, lmixscale)
00290     if (cs%debug) then
00291       if (cs%visc_drag) &
00292         call uvchksum("MEKE drag_vel_[uv]", drag_vel_u, drag_vel_v, g%HI, &
00293                       scale=us%Z_to_m*us%s_to_T, scalar_pair=.true.)
00294       call hchksum(mass, 'MEKE mass',g%HI,haloshift=1, scale=us%RZ_to_kg_m2)
00295       call hchksum(drag_rate_visc, 'MEKE drag_rate_visc', g%HI, scale=us%L_T_to_m_s)
00296       call hchksum(bottomfac2, 'MEKE bottomFac2', g%HI)
00297       call hchksum(barotrfac2, 'MEKE barotrFac2', g%HI)
00298       call hchksum(lmixscale, 'MEKE LmixScale', g%HI,scale=us%L_to_m)
00299     endif
00300
00301     ! Aggregate sources of MEKE (background, frictional and GM)
00302     !$OMP parallel do default(shared)
00303     do j=js,je ; do i=is,ie
00304       src(i,j) = cs%MEKE_BGsrc
00305     enddo ; enddo
00306
00307     if (associated(meke%mom_src)) then
00308       !$OMP parallel do default(shared)
00309       do j=js,je ; do i=is,ie
00310         src(i,j) = src(i,j) - cs%MEKE_FrCoeff*i_mass(i,j)*meke%mom_src(i,j)
00311       enddo ; enddo
00312     endif
00313
00314     if (associated(meke%GME_snk)) then
00315       !$OMP parallel do default(shared)
00316       do j=js,je ; do i=is,ie
00317         src(i,j) = src(i,j) - cs%MEKE_GMECoeff*i_mass(i,j)*meke%GME_snk(i,j)
00318       enddo ; enddo
00319     endif
00320
00321     if (associated(meke%GM_src)) then
00322       if (cs%GM_src_alt) then
00323         !$OMP parallel do default(shared)
00324         do j=js,je ; do i=is,ie
00325           src(i,j) = src(i,j) - cs%MEKE_GMcoeff*meke%GM_src(i,j) / &
00326                      (gv%Rho0 * max(1.0*us%m_to_Z, g%bathyT(i,j)))
00327         enddo ; enddo
```

```
00328       else
00329         !$OMP parallel do default(shared)
00330         do j=js,je ; do i=is,ie
00331           src(i,j) = src(i,j) - cs%MEKE_GMcoeff*i_mass(i,j)*meke%GM_src(i,j)
00332         enddo ; enddo
00333       endif
00334     endif
00335
00336     if (cs%MEKE_equilibrium_restoring) then
00337       call meke_equilibrium_restoring(cs, g, us, sn_u, sn_v)
00338       do j=js,je ; do i=is,ie
00339         src(i,j) = src(i,j) - cs%MEKE_restoring_rate*(meke%MEKE(i,j) - cs%equilibrium_value(i,j))
00340       enddo ; enddo
00341     endif
00342
00343     ! Increase EKE by a full time-steps worth of source
00344     !$OMP parallel do default(shared)
00345     do j=js,je ; do i=is,ie
00346       meke%MEKE(i,j) = (meke%MEKE(i,j) + sdt*src(i,j))*g%mask2dT(i,j)
00347     enddo ; enddo
00348
00349     if (use_drag_rate) then
00350       ! Calculate a viscous drag rate (includes BBL contributions from mean flow and eddies)
00351       !$OMP parallel do default(shared)
00352       do j=js,je ; do i=is,ie
00353         drag_rate(i,j) = (us%L_to_Z*rho0 * i_mass(i,j)) * sqrt( drag_rate_visc(i,j)**2 + &
00354                   cdrag2 * ( max(0.0, 2.0*bottomfac2(i,j)*meke%MEKE(i,j)) + cs%MEKE_Uscale**2 ) )
00355       enddo ; enddo
00356     endif
00357
00358     ! First stage of Strang splitting
00359     !$OMP parallel do default(shared)
00360     do j=js,je ; do i=is,ie
00361       ldamping = cs%MEKE_damping + drag_rate(i,j) * bottomfac2(i,j)
00362       if (meke%MEKE(i,j) < 0.) ldamping = 0.
00363       ! notice that the above line ensures a damping only if MEKE is positive,
00364       ! while leaving MEKE unchanged if it is negative
00365       meke%MEKE(i,j) =  meke%MEKE(i,j) / (1.0 + sdt_damp*ldamping)
00366       meke_decay(i,j) = ldamping*g%mask2dT(i,j)
00367     enddo ; enddo
00368
00369     if (cs%kh_flux_enabled .or. cs%MEKE_K4 >= 0.0) then
00370       ! Update MEKE in the halos for lateral or bi-harmonic diffusion
00371       call cpu_clock_begin(cs%id_clock_pass)
00372       call do_group_pass(cs%pass_MEKE, g%Domain)
00373       call cpu_clock_end(cs%id_clock_pass)
00374     endif
00375
00376     if (cs%MEKE_K4 >= 0.0) then
00377       ! Calculate Laplacian of MEKE using MEKE_uflux and MEKE_vflux as temporary work space.
00378       !$OMP parallel do default(shared)
00379       do j=js-1,je+1 ; do i=is-2,ie+1
00380         ! MEKE_uflux is used here as workspace with units of [L2 T-2 ~> m2 s-2].
00381         meke_uflux(i,j) = ((g%dy_Cu(i,j)*g%IdxCu(i,j)) * g%mask2dCu(i,j)) * &
00382             (meke%MEKE(i+1,j) - meke%MEKE(i,j))
00383       ! This would have units of [R Z L2 T-2 ~> kg s-2]
00384       ! MEKE_uflux(I,j) = ((G%dy_Cu(I,j)*G%IdxCu(I,j)) * &
00385       !     ((2.0*mass(i,j)*mass(i+1,j)) / ((mass(i,j)+mass(i+1,j)) + mass_neglect)) ) * &
00386       !     (MEKE%MEKE(i+1,j) - MEKE%MEKE(i,j))
00387       enddo ; enddo
00388       !$OMP parallel do default(shared)
00389       do j=js-2,je+1 ; do i=is-1,ie+1
00390         ! MEKE_vflux is used here as workspace with units of [L2 T-2 ~> m2 s-2].
00391         meke_vflux(i,j) = ((g%dx_Cv(i,j)*g%IdyCv(i,j)) * g%mask2dCv(i,j)) * &
00392             (meke%MEKE(i,j+1) - meke%MEKE(i,j))
00393       ! This would have units of [R Z L2 T-2 ~> kg s-2]
00394       ! MEKE_vflux(i,J) = ((G%dx_Cv(i,J)*G%IdyCv(i,J)) * &
00395       !     ((2.0*mass(i,j)*mass(i,j+1)) / ((mass(i,j)+mass(i,j+1)) + mass_neglect)) ) * &
00396       !     (MEKE%MEKE(i,j+1) - MEKE%MEKE(i,j))
00397       enddo ; enddo
00398
00399       !$OMP parallel do default(shared)
00400       do j=js-1,je+1 ; do i=is-1,ie+1 ! del2MEKE has units [T-2 ~> s-2].
00401         del2meke(i,j) = g%IareaT(i,j) * &
00402             ((meke_uflux(i,j) - meke_uflux(i-1,j)) + (meke_vflux(i,j) - meke_vflux(i,j-1)))
00403       enddo ; enddo
00404
00405       ! Bi-harmonic diffusion of MEKE
00406       !$OMP parallel do default(shared) private(K4_here,Inv_K4_max)
00407       do j=js,je ; do i=is-1,ie
00408         k4_here = cs%MEKE_K4 ! [L4 T-1 ~> m4 s-1]
00409         ! Limit Kh to avoid CFL violations.
00410         inv_k4_max = 64.0 * sdt * ((g%dy_Cu(i,j)*g%IdxCu(i,j)) * &
00411                       max(g%IareaT(i,j), g%IareaT(i+1,j)))**2
00412         if (k4_here*inv_k4_max > 0.3) k4_here = 0.3 / inv_k4_max
00413
00414         ! Here the units of MEKE_uflux are [R Z L4 T-3 ~> kg m2 s-3].
```

```
00415         meke_uflux(i,j) = ((k4_here * (g%dy_Cu(i,j)*g%IdxCu(i,j))) * &
00416             ((2.0*mass(i,j)*mass(i+1,j)) / ((mass(i,j)+mass(i+1,j)) + mass_neglect)) ) * &
00417             (del2meke(i+1,j) - del2meke(i,j))
00418       enddo ; enddo
00419       !$OMP parallel do default(shared) private(K4_here,Inv_K4_max)
00420       do j=js-1,je ; do i=is,ie
00421         k4_here = cs%MEKE_K4 ! [L4 T-1 ~> m4 s-1]
00422         inv_k4_max = 64.0 * sdt * ((g%dx_Cv(i,j)*g%IdyCv(i,j)) * max(g%IareaT(i,j), g%IareaT(i,j+1)))**2
00423         if (k4_here*inv_k4_max > 0.3) k4_here = 0.3 / inv_k4_max
00424
00425         ! Here the units of MEKE_vflux are [R Z L4 T-3 ~> kg m2 s-3].
00426         meke_vflux(i,j) = ((k4_here * (g%dx_Cv(i,j)*g%IdyCv(i,j))) * &
00427             ((2.0*mass(i,j)*mass(i,j+1)) / ((mass(i,j)+mass(i,j+1)) + mass_neglect)) ) * &
00428             (del2meke(i,j+1) - del2meke(i,j))
00429       enddo ; enddo
00430       ! Store change in MEKE arising from the bi-harmonic in del4MEKE [L2 T-2 ~> m2 s-2].
00431       !$OMP parallel do default(shared)
00432       do j=js,je ; do i=is,ie
00433         del4meke(i,j) = (sdt*(g%IareaT(i,j)*i_mass(i,j))) * &
00434             ((meke_uflux(i-1,j) - meke_uflux(i,j)) + &
00435              (meke_vflux(i,j-1) - meke_vflux(i,j)))
00436       enddo ; enddo
00437     endif !
00438
00439     if (cs%kh_flux_enabled) then
00440       ! Lateral diffusion of MEKE
00441       kh_here = max(0., cs%MEKE_Kh)
00442       !$OMP parallel do default(shared) firstprivate(Kh_here) private(Inv_Kh_max)
00443       do j=js,je ; do i=is-1,ie
00444         ! Limit Kh to avoid CFL violations.
00445         if (associated(meke%Kh)) &
00446           kh_here = max(0., cs%MEKE_Kh) + &
00447               cs%KhMEKE_Fac*0.5*(meke%Kh(i,j)+meke%Kh(i+1,j))
00448         if (associated(meke%Kh_diff)) &
00449           kh_here = max(0.,cs%MEKE_Kh) + &
00450               cs%KhMEKE_Fac*0.5*(meke%Kh_diff(i,j)+meke%Kh_diff(i+1,j))
00451         inv_kh_max = 2.0*sdt * ((g%dy_Cu(i,j)*g%IdxCu(i,j)) * &
00452                       max(g%IareaT(i,j),g%IareaT(i+1,j)))
00453         if (kh_here*inv_kh_max > 0.25) kh_here = 0.25 / inv_kh_max
00454         kh_u(i,j) = kh_here
00455
00456         ! Here the units of MEKE_uflux and MEKE_vflux are [R Z L4 T-3 ~> kg m2 s-3].
00457         meke_uflux(i,j) = ((kh_here * (g%dy_Cu(i,j)*g%IdxCu(i,j))) * &
00458             ((2.0*mass(i,j)*mass(i+1,j)) / ((mass(i,j)+mass(i+1,j)) + mass_neglect)) ) * &
00459             (meke%MEKE(i,j) - meke%MEKE(i+1,j))
00460       enddo ; enddo
00461       !$OMP parallel do default(shared) firstprivate(Kh_here) private(Inv_Kh_max)
00462       do j=js-1,je ; do i=is,ie
00463         if (associated(meke%Kh)) &
00464           kh_here = max(0.,cs%MEKE_Kh) + cs%KhMEKE_Fac * 0.5*(meke%Kh(i,j)+meke%Kh(i,j+1))
00465         if (associated(meke%Kh_diff)) &
00466           kh_here = max(0.,cs%MEKE_Kh) + cs%KhMEKE_Fac * 0.5*(meke%Kh_diff(i,j)+meke%Kh_diff(i,j+1))
00467         inv_kh_max = 2.0*sdt * ((g%dx_Cv(i,j)*g%IdyCv(i,j)) * max(g%IareaT(i,j),g%IareaT(i,j+1)))
00468         if (kh_here*inv_kh_max > 0.25) kh_here = 0.25 / inv_kh_max
00469         kh_v(i,j) = kh_here
00470
00471         ! Here the units of MEKE_uflux and MEKE_vflux are [R Z L4 T-3 ~> kg m2 s-3].
00472         meke_vflux(i,j) = ((kh_here * (g%dx_Cv(i,j)*g%IdyCv(i,j))) * &
00473             ((2.0*mass(i,j)*mass(i,j+1)) / ((mass(i,j)+mass(i,j+1)) + mass_neglect)) ) * &
00474             (meke%MEKE(i,j) - meke%MEKE(i,j+1))
00475       enddo ; enddo
00476       if (cs%MEKE_advection_factor>0.) then
00477         advfac = cs%MEKE_advection_factor / sdt ! [T-1 ~> s-1]
00478         !$OMP parallel do default(shared)
00479         do j=js,je ; do i=is-1,ie
00480           ! Here the units of the quantities added to MEKE_uflux are [R Z L4 T-3 ~> kg m2 s-3].
00481           if (barohu(i,j)>0.) then
00482             meke_uflux(i,j) = meke_uflux(i,j) + barohu(i,j)*meke%MEKE(i,j)*advfac
00483           elseif (barohu(i,j)<0.) then
00484             meke_uflux(i,j) = meke_uflux(i,j) + barohu(i,j)*meke%MEKE(i+1,j)*advfac
00485           endif
00486         enddo ; enddo
00487         !$OMP parallel do default(shared)
00488         do j=js-1,je ; do i=is,ie
00489           ! Here the units of the quantities added to MEKE_vflux are [R Z L4 T-3 ~> kg m2 s-3].
00490           if (barohv(i,j)>0.) then
00491             meke_vflux(i,j) = meke_vflux(i,j) + barohv(i,j)*meke%MEKE(i,j)*advfac
00492           elseif (barohv(i,j)<0.) then
00493             meke_vflux(i,j) = meke_vflux(i,j) + barohv(i,j)*meke%MEKE(i,j+1)*advfac
00494           endif
00495         enddo ; enddo
00496       endif
00497
00498       !$OMP parallel do default(shared)
00499       do j=js,je ; do i=is,ie
00500         meke%MEKE(i,j) = meke%MEKE(i,j) + (sdt*(g%IareaT(i,j)*i_mass(i,j))) * &
00501             ((meke_uflux(i-1,j) - meke_uflux(i,j)) + &
```

```
00502              (meke_vflux(i,j-1) - meke_vflux(i,j)))
00503       enddo ; enddo
00504     endif ! MEKE_KH>0
00505
00506     ! Add on bi-harmonic tendency
00507     if (cs%MEKE_K4 >= 0.0) then
00508       !$OMP parallel do default(shared)
00509       do j=js,je ; do i=is,ie
00510         meke%MEKE(i,j) = meke%MEKE(i,j) + del4meke(i,j)
00511       enddo ; enddo
00512     endif
00513
00514     ! Second stage of Strang splitting
00515     if (cs%MEKE_KH >= 0.0 .or. cs%MEKE_K4 >= 0.0) then
00516       if (sdt>sdt_damp) then
00517         ! Recalculate the drag rate, since MEKE has changed.
00518         if (use_drag_rate) then
00519           !$OMP parallel do default(shared)
00520           do j=js,je ; do i=is,ie
00521             drag_rate(i,j) = (us%L_to_Z*rho0 * i_mass(i,j)) * sqrt( drag_rate_visc(i,j)**2 + &
00522                    cdrag2 * ( max(0.0, 2.0*bottomfac2(i,j)*meke%MEKE(i,j)) + cs%MEKE_Uscale**2 ) )
00523           enddo ; enddo
00524           !$OMP parallel do default(shared)
00525           do j=js,je ; do i=is,ie
00526             ldamping = cs%MEKE_damping + drag_rate(i,j) * bottomfac2(i,j)
00527             if (meke%MEKE(i,j) < 0.) ldamping = 0.
00528             ! notice that the above line ensures a damping only if MEKE is positive,
00529             ! while leaving MEKE unchanged if it is negative
00530             meke%MEKE(i,j) =  meke%MEKE(i,j) / (1.0 + sdt_damp*ldamping)
00531             meke_decay(i,j) = ldamping*g%mask2dT(i,j)
00532           enddo ; enddo
00533         endif
00534       endif
00535     endif ! MEKE_KH>=0
00536
00537 !   do j=js,je ; do i=is,ie
00538 !     MEKE%MEKE(i,j) =  MAX(MEKE%MEKE(i,j),0.0)
00539 !   enddo ; enddo
00540
00541     call cpu_clock_begin(cs%id_clock_pass)
00542     call do_group_pass(cs%pass_MEKE, g%Domain)
00543     call cpu_clock_end(cs%id_clock_pass)
00544
00545     ! Calculate diffusivity for main model to use
00546     if (cs%MEKE_KhCoeff>0.) then
00547       if (.not.cs%MEKE_GEOMETRIC) then
00548         if (cs%use_old_lscale) then
00549           if (cs%Rd_as_max_scale) then
00550             !$OMP parallel do default(shared)
00551             do j=js,je ; do i=is,ie
00552               meke%Kh(i,j) = (cs%MEKE_KhCoeff * &
00553                           sqrt(2.*max(0.,barotrfac2(i,j)*meke%MEKE(i,j))*g%areaT(i,j)) ) * &
00554                           min(meke%Rd_dx_h(i,j), 1.0)
00555             enddo ; enddo
00556           else
00557             !$OMP parallel do default(shared)
00558             do j=js,je ; do i=is,ie
00559               meke%Kh(i,j) = cs%MEKE_KhCoeff * &
00560                   sqrt(2.*max(0., barotrfac2(i,j)*meke%MEKE(i,j))*g%areaT(i,j))
00561             enddo ; enddo
00562           endif
00563         else
00564           !$OMP parallel do default(shared)
00565           do j=js,je ; do i=is,ie
00566             meke%Kh(i,j) = cs%MEKE_KhCoeff * &
00567                 sqrt(2.*max(0., barotrfac2(i,j)*meke%MEKE(i,j))) * lmixscale(i,j)
00568           enddo ; enddo
00569         endif
00570       endif
00571     endif
00572
00573     ! Calculate viscosity for the main model to use
00574     if (cs%viscosity_coeff_Ku /=0.) then
00575       do j=js,je ; do i=is,ie
00576         meke%Ku(i,j) = cs%viscosity_coeff_Ku * sqrt(2.*max(0.,meke%MEKE(i,j))) * lmixscale(i,j)
00577       enddo ; enddo
00578     endif
00579
00580     if (cs%viscosity_coeff_Au /=0.) then
00581       do j=js,je ; do i=is,ie
00582         meke%Au(i,j) = cs%viscosity_coeff_Au * sqrt(2.*max(0.,meke%MEKE(i,j))) * lmixscale(i,j)**3
00583       enddo ; enddo
00584     endif
00585
00586     if (associated(meke%Kh) .or. associated(meke%Ku) .or. associated(meke%Au)) then
00587       call cpu_clock_begin(cs%id_clock_pass)
00588       call do_group_pass(cs%pass_Kh, g%Domain)
```

```
00589       call cpu_clock_end(cs%id_clock_pass)
00590     endif
00591
00592     ! Offer fields for averaging.
00593     if (any([cs%id_Ue, cs%id_Ub, cs%id_Ut] > 0)) &
00594       tmp(:,:) = 0.
00595     if (cs%id_MEKE>0) call post_data(cs%id_MEKE, meke%MEKE, cs%diag)
00596     if (cs%id_Ue>0) then
00597       do j=js,je ; do i=is,ie
00598         tmp(i,j) = sqrt(max(0., 2. * meke%MEKE(i,j)))
00599       enddo ; enddo
00600       call post_data(cs%id_Ue, tmp, cs%diag)
00601     endif
00602     if (cs%id_Ub>0) then
00603       do j=js,je ; do i=is,ie
00604         tmp(i,j) = sqrt(max(0., 2. * meke%MEKE(i,j) * bottomfac2(i,j)))
00605       enddo ; enddo
00606       call post_data(cs%id_Ub, tmp, cs%diag)
00607     endif
00608     if (cs%id_Ut>0) then
00609       do j=js,je ; do i=is,ie
00610         tmp(i,j) = sqrt(max(0., 2. * meke%MEKE(i,j) * barotrfac2(i,j)))
00611       enddo ; enddo
00612       call post_data(cs%id_Ut, tmp, cs%diag)
00613     endif
00614     if (cs%id_Kh>0) call post_data(cs%id_Kh, meke%Kh, cs%diag)
00615     if (cs%id_Ku>0) call post_data(cs%id_Ku, meke%Ku, cs%diag)
00616     if (cs%id_Au>0) call post_data(cs%id_Au, meke%Au, cs%diag)
00617     if (cs%id_KhMEKE_u>0) call post_data(cs%id_KhMEKE_u, kh_u, cs%diag)
00618     if (cs%id_KhMEKE_v>0) call post_data(cs%id_KhMEKE_v, kh_v, cs%diag)
00619     if (cs%id_src>0) call post_data(cs%id_src, src, cs%diag)
00620     if (cs%id_decay>0) call post_data(cs%id_decay, meke_decay, cs%diag)
00621     if (cs%id_GM_src>0) call post_data(cs%id_GM_src, meke%GM_src, cs%diag)
00622     if (cs%id_mom_src>0) call post_data(cs%id_mom_src, meke%mom_src, cs%diag)
00623     if (cs%id_GME_snk>0) call post_data(cs%id_GME_snk, meke%GME_snk, cs%diag)
00624     if (cs%id_Le>0) call post_data(cs%id_Le, lmixscale, cs%diag)
00625     if (cs%id_gamma_b>0) then
00626       do j=js,je ; do i=is,ie
00627         bottomfac2(i,j) = sqrt(bottomfac2(i,j))
00628       enddo ; enddo
00629       call post_data(cs%id_gamma_b, bottomfac2, cs%diag)
00630     endif
00631     if (cs%id_gamma_t>0) then
00632       do j=js,je ; do i=is,ie
00633         barotrfac2(i,j) = sqrt(barotrfac2(i,j))
00634       enddo ; enddo
00635       call post_data(cs%id_gamma_t, barotrfac2, cs%diag)
00636     endif
00637
```

42.6 mom_unit_scaling Module Reference

Provides a transparent unit rescaling type to facilitate dimensional consistency testing.

Data Types

- type unit_scale_type

  Describes various unit conversion factors.

Functions/Subroutines

- subroutine, public unit_scaling_init (param_file, US)

  Allocates and initializes the ocean model unit scaling type.
- subroutine, public fix_restart_unit_scaling (US)

  Set the unit scaling factors for output to restart files to the unit scaling factors for this run.
- subroutine, public unit_scaling_end (US)

  Deallocates a unit scaling structure.

42.6.1 Detailed Description

Provides a transparent unit rescaling type to facilitate dimensional consistency testing.

42.6.2 Function/Subroutine Documentation

42.6.2.1 fix_restart_unit_scaling()

```
subroutine, public mom_unit_scaling::fix_restart_unit_scaling (
             type(unit_scale_type), intent(inout) US )
```

Set the unit scaling factors for output to restart files to the unit scaling factors for this run.

Parameters

in,out	*us*	A dimensional unit scaling type

Definition at line 173 of file MOM_unit_scaling.F90.

Referenced by mom_ice_shelf::initialize_ice_shelf().

```
00173   type(unit_scale_type), intent(inout) :: us !< A dimensional unit scaling type
00174
00175   us%m_to_Z_restart = us%m_to_Z
00176   us%m_to_L_restart = us%m_to_L
00177   us%s_to_T_restart = us%s_to_T
00178   us%kg_m3_to_R_restart = us%kg_m3_to_R
00179   us%J_kg_to_Q_restart = us%J_kg_to_Q
00180
```

42.6.2.2 unit_scaling_end()

```
subroutine, public mom_unit_scaling::unit_scaling_end (
             type(unit_scale_type), pointer US )
```

Deallocates a unit scaling structure.

Parameters

us	A dimensional unit scaling type

Definition at line 185 of file MOM_unit_scaling.F90.

```
00185   type(unit_scale_type), pointer :: us !< A dimensional unit scaling type
00186
00187   deallocate( us )
00188
```

42.6.2.3 unit_scaling_init()

```
subroutine, public mom_unit_scaling::unit_scaling_init (
             type(param_file_type), intent(in), optional param_file,
             type(unit_scale_type), optional, pointer US )
```

Allocates and initializes the ocean model unit scaling type.

Parameters

in	*param_file*	Parameter file handle/type
	us	A dimensional unit scaling type

Definition at line 57 of file MOM_unit_scaling.F90.

Referenced by mom_ice_shelf::initialize_ice_shelf().

```
00057   type(param_file_type), optional, intent(in) :: param_file !< Parameter file handle/type
00058   type(unit_scale_type), optional, pointer    :: us         !< A dimensional unit scaling type
00059
00060   ! This routine initializes a unit_scale_type structure (US).
00061
00062   ! Local variables
00063   integer :: z_power, l_power, t_power, r_power, q_power
00064   real    :: z_rescale_factor, l_rescale_factor, t_rescale_factor, r_rescale_factor, q_rescale_factor
00065   ! This include declares and sets the variable "version".
00066 # include "version_variable.h"
00067   character(len=16) :: mdl = "MOM_unit_scaling"
00068
00069   if (.not.present(us)) return
00070
00071   if (associated(us)) call mom_error(fatal, &
00072      'unit_scaling_init: called with an associated US pointer.')
00073   allocate(us)
00074
00075   if (present(param_file)) then
00076     ! Read all relevant parameters and write them to the model log.
00077     call log_version(param_file, mdl, version, &
00078                 "Parameters for doing unit scaling of variables.", debugging=.true.)
00079     call get_param(param_file, mdl, "Z_RESCALE_POWER", z_power, &
00080                 "An integer power of 2 that is used to rescale the model's "//&
00081                 "internal units of depths and heights.  Valid values range from -300 to 300.", &
00082                 units="nondim", default=0, debuggingparam=.true.)
00083     call get_param(param_file, mdl, "L_RESCALE_POWER", l_power, &
00084                 "An integer power of 2 that is used to rescale the model's "//&
00085                 "internal units of lateral distances.  Valid values range from -300 to 300.", &
00086                 units="nondim", default=0, debuggingparam=.true.)
00087     call get_param(param_file, mdl, "T_RESCALE_POWER", t_power, &
00088                 "An integer power of 2 that is used to rescale the model's "//&
00089                 "internal units of time.  Valid values range from -300 to 300.", &
00090                 units="nondim", default=0, debuggingparam=.true.)
00091     call get_param(param_file, mdl, "R_RESCALE_POWER", r_power, &
00092                 "An integer power of 2 that is used to rescale the model's "//&
00093                 "internal units of density.  Valid values range from -300 to 300.", &
00094                 units="nondim", default=0, debuggingparam=.true.)
00095     call get_param(param_file, mdl, "Q_RESCALE_POWER", q_power, &
00096                 "An integer power of 2 that is used to rescale the model's "//&
00097                   "internal units of heat content.  Valid values range from -300 to 300.", &
00098                   units="nondim", default=0, debuggingparam=.true.)
00099   else
00100     z_power = 0 ; l_power = 0 ; t_power = 0 ; r_power = 0 ; q_power = 0
00101   endif
00102
00103   if (abs(z_power) > 300) call mom_error(fatal, "unit_scaling_init: "//&
00104                 "Z_RESCALE_POWER is outside of the valid range of -300 to 300.")
00105   if (abs(l_power) > 300) call mom_error(fatal, "unit_scaling_init: "//&
00106                 "L_RESCALE_POWER is outside of the valid range of -300 to 300.")
00107   if (abs(t_power) > 300) call mom_error(fatal, "unit_scaling_init: "//&
00108                 "T_RESCALE_POWER is outside of the valid range of -300 to 300.")
00109   if (abs(r_power) > 300) call mom_error(fatal, "unit_scaling_init: "//&
```

```
00110                 "R_RESCALE_POWER is outside of the valid range of -300 to 300.")
00111   if (abs(q_power) > 300) call mom_error(fatal, "unit_scaling_init: "//&
00112                 "Q_RESCALE_POWER is outside of the valid range of -300 to 300.")
00113
00114   z_rescale_factor = 1.0
00115   if (z_power /= 0) z_rescale_factor = 2.0**z_power
00116   us%Z_to_m = 1.0 * z_rescale_factor
00117   us%m_to_Z = 1.0 / z_rescale_factor
00118
00119   l_rescale_factor = 1.0
00120   if (l_power /= 0) l_rescale_factor = 2.0**l_power
00121   us%L_to_m = 1.0 * l_rescale_factor
00122   us%m_to_L = 1.0 / l_rescale_factor
00123
00124   t_rescale_factor = 1.0
00125   if (t_power /= 0) t_rescale_factor = 2.0**t_power
00126   us%T_to_s = 1.0 * t_rescale_factor
00127   us%s_to_T = 1.0 / t_rescale_factor
00128
00129   r_rescale_factor = 1.0
00130   if (r_power /= 0) r_rescale_factor = 2.0**r_power
00131   us%R_to_kg_m3 = 1.0 * r_rescale_factor
00132   us%kg_m3_to_R = 1.0 / r_rescale_factor
00133
00134   q_rescale_factor = 1.0
00135   if (q_power /= 0) q_rescale_factor = 2.0**q_power
00136   us%Q_to_J_kg = 1.0 * q_rescale_factor
00137   us%J_kg_to_Q = 1.0 / q_rescale_factor
00138
00139   ! These are useful combinations of the fundamental scale conversion factors set above.
00140   us%Z_to_L = us%Z_to_m * us%m_to_L
00141   us%L_to_Z = us%L_to_m * us%m_to_Z
00142   ! Horizontal velocities:
00143   us%L_T_to_m_s = us%L_to_m * us%s_to_T
00144   us%m_s_to_L_T = us%m_to_L * us%T_to_s
00145   ! Horizontal accelerations:
00146   us%L_T2_to_m_s2 = us%L_to_m * us%s_to_T**2
00147     ! It does not look like US%m_s2_to_L_T2 would be used, so it does not exist.
00148   ! Vertical diffusivities and viscosities:
00149   us%Z2_T_to_m2_s = us%Z_to_m**2 * us%s_to_T
00150   us%m2_s_to_Z2_T = us%m_to_Z**2 * us%T_to_s
00151   ! Column mass loads:
00152   us%RZ_to_kg_m2  = us%R_to_kg_m3 * us%Z_to_m
00153     ! It does not seem like US%kg_m2_to_RZ would be used enough in MOM6 to justify its existence.
00154   ! Vertical mass fluxes:
00155   us%kg_m2s_to_RZ_T = us%kg_m3_to_R * us%m_to_Z * us%T_to_s
00156   us%RZ_T_to_kg_m2s = us%R_to_kg_m3 * us%Z_to_m * us%s_to_T
00157   ! Turbulent kinetic energy vertical fluxes:
00158   us%RZ3_T3_to_W_m2 = us%R_to_kg_m3 * us%Z_to_m**3 * us%s_to_T**3
00159   us%W_m2_to_RZ3_T3 = us%kg_m3_to_R * us%m_to_Z**3 * us%T_to_s**3
00160   ! Vertical heat fluxes:
00161   us%W_m2_to_QRZ_T = us%J_kg_to_Q * us%kg_m3_to_R * us%m_to_Z * us%T_to_s
00162   us%QRZ_T_to_W_m2 = us%Q_to_J_kg * us%R_to_kg_m3 * us%Z_to_m * us%s_to_T
00163   ! Pressures:
00164   us%RL2_T2_to_Pa = us%R_to_kg_m3 * us%L_T_to_m_s**2
00165     ! It does not seem like US%Pa_to_RL2_T2 would be used enough in MOM6 to justify its existence.
00166   ! US%Pa_to_RL2_T2 = US%kg_m3_to_R * US%m_s_to_L_T**2
00167
```

Chapter 43

Data Type Documentation

43.1 mom_ale::ale_cs Type Reference

ALE control structure.

Public Attributes

- logical remap_uv_using_old_alg

  If true, uses the old "remapping via a delta z" method. If False, uses the new method that remaps between grids described by h.
- real regrid_time_scale

  The time-scale used in blending between the current (old) grid and the target (new) grid [T ~> s].
- type(regridding_cs) regridcs

  Regridding parameters and work arrays.
- type(remapping_cs) remapcs

  Remapping parameters and work arrays.
- integer nk

  Used only for queries, not directly by this module.
- logical remap_after_initialization

  Indicates whether to regrid/remap after initializing the state.
- logical answers_2018

  If true, use the order of arithmetic and expressions for remapping that recover the answers from the end of 2018. Otherwise, use more robust and accurate forms of mathematically equivalent expressions.
- logical show_call_tree

  For debugging.
- type(diag_ctrl), pointer diag

  structure to regulate output
- integer, dimension(:), allocatable id_tracer_remap_tendency

  diagnostic id
- integer, dimension(:), allocatable id_htracer_remap_tendency

  diagnostic id
- integer, dimension(:), allocatable id_htracer_remap_tendency_2d

  diagnostic id
- logical, dimension(:), allocatable do_tendency_diag

  flag for doing diagnostics

- integer id_dzregrid = -1

  diagnostic id
- integer id_u_preale = -1

  diagnostic id for zonal velocity before ALE.
- integer id_v_preale = -1

  diagnostic id for meridional velocity before ALE.
- integer id_h_preale = -1

  diagnostic id for layer thicknesses before ALE.
- integer id_t_preale = -1

  diagnostic id for temperatures before ALE.
- integer id_s_preale = -1

  diagnostic id for salinities before ALE.
- integer id_e_preale = -1

  diagnostic id for interface heights before ALE.
- integer id_vert_remap_h = -1

  diagnostic id for layer thicknesses used for remapping
- integer id_vert_remap_h_tendency = -1

  diagnostic id for layer thickness tendency due to ALE

43.1.1 Detailed Description

ALE control structure.

Definition at line 63 of file MOM_ALE.F90.

43.1.2 Member Data Documentation

43.1.2.1 answers_2018

```
logical mom_ale::ale_cs::answers_2018
```

If true, use the order of arithmetic and expressions for remapping that recover the answers from the end of 2018. Otherwise, use more robust and accurate forms of mathematically equivalent expressions.

Definition at line 78 of file MOM_ALE.F90.

```
00078   logical :: answers_2018   !< If true, use the order of arithmetic and expressions for remapping
```

43.1.2.2 diag

```
type(diag_ctrl), pointer mom_ale::ale_cs::diag
```

structure to regulate output

Definition at line 85 of file MOM_ALE.F90.

```
00085   type(diag_ctrl), pointer          :: diag                       !< structure to regulate output
```

43.1.2.3 do_tendency_diag

```
logical, dimension(:), allocatable mom_ale::ale_cs::do_tendency_diag
```

flag for doing diagnostics

Definition at line 89 of file MOM_ALE.F90.

```
00089   logical, dimension(:), allocatable :: do_tendency_diag             !< flag for doing diagnostics
```

43.1.2.4 id_dzregrid

```
integer mom_ale::ale_cs::id_dzregrid = -1
```

diagnostic id

Definition at line 90 of file MOM_ALE.F90.

```
00090   integer                           :: id_dzregrid = -1             !< diagnostic id
```

43.1.2.5 id_e_preale

```
integer mom_ale::ale_cs::id_e_preale = -1
```

diagnostic id for interface heights before ALE.

Definition at line 98 of file MOM_ALE.F90.

```
00098   integer :: id_e_preale = -1 !< diagnostic id for interface heights before ALE.
```

43.1.2.6 id_h_preale

```
integer mom_ale::ale_cs::id_h_preale = -1
```

diagnostic id for layer thicknesses before ALE.

Definition at line 95 of file MOM_ALE.F90.

```
00095   integer :: id_h_preale = -1 !< diagnostic id for layer thicknesses before ALE.
```

43.1.2.7 id_htracer_remap_tendency

```
integer, dimension(:), allocatable mom_ale::ale_cs::id_htracer_remap_tendency
```

diagnostic id

Definition at line 87 of file MOM_ALE.F90.

```
00087   integer, dimension(:), allocatable :: id_htracer_remap_tendency     !< diagnostic id
```

43.1.2.8 id_htracer_remap_tendency_2d

```
integer, dimension(:), allocatable mom_ale::ale_cs::id_htracer_remap_tendency_2d
```

diagnostic id

Definition at line 88 of file MOM_ALE.F90.

```
00088   integer, dimension(:), allocatable :: id_htracer_remap_tendency_2d  !< diagnostic id
```

43.1.2.9 id_s_preale

```
integer mom_ale::ale_cs::id_s_preale = -1
```

diagnostic id for salinities before ALE.

Definition at line 97 of file MOM_ALE.F90.

```
00097   integer :: id_s_preale = -1 !< diagnostic id for salinities before ALE.
```

43.1.2.10 id_t_preale

```
integer mom_ale::ale_cs::id_t_preale = -1
```

diagnostic id for temperatures before ALE.

Definition at line 96 of file MOM_ALE.F90.

```
00096   integer :: id_t_preale = -1 !< diagnostic id for temperatures before ALE.
```

43.1.2.11 id_tracer_remap_tendency

```
integer, dimension(:), allocatable mom_ale::ale_cs::id_tracer_remap_tendency
```

diagnostic id

Definition at line 86 of file MOM_ALE.F90.

```
00086   integer, dimension(:), allocatable :: id_tracer_remap_tendency      !< diagnostic id
```

43.1.2.12 id_u_preale

```
integer mom_ale::ale_cs::id_u_preale = -1
```

diagnostic id for zonal velocity before ALE.

Definition at line 93 of file MOM_ALE.F90.

```
00093   integer :: id_u_preale = -1 !< diagnostic id for zonal velocity before ALE.
```

43.1.2.13 id_v_preale

```
integer mom_ale::ale_cs::id_v_preale = -1
```

diagnostic id for meridional velocity before ALE.

Definition at line 94 of file MOM_ALE.F90.

```
00094   integer :: id_v_preale = -1 !< diagnostic id for meridional velocity before ALE.
```

43.1.2.14 id_vert_remap_h

```
integer mom_ale::ale_cs::id_vert_remap_h = -1
```

diagnostic id for layer thicknesses used for remapping

Definition at line 99 of file MOM_ALE.F90.

```
00099   integer :: id_vert_remap_h = -1      !< diagnostic id for layer thicknesses used for remapping
```

43.1.2.15 id_vert_remap_h_tendency

```
integer mom_ale::ale_cs::id_vert_remap_h_tendency = -1
```

diagnostic id for layer thickness tendency due to ALE

Definition at line 100 of file MOM_ALE.F90.

```
00100   integer :: id_vert_remap_h_tendency = -1 !< diagnostic id for layer thickness tendency due to ALE
```

43.1.2.16 nk

```
integer mom_ale::ale_cs::nk
```

Used only for queries, not directly by this module.

Definition at line 74 of file MOM_ALE.F90.

```
00074   integer :: nk             !< Used only for queries, not directly by this module
```

43.1.2.17 regrid_time_scale

```
real mom_ale::ale_cs::regrid_time_scale
```

The time-scale used in blending between the current (old) grid and the target (new) grid [T ~> s].

Definition at line 68 of file MOM_ALE.F90.

```
00068   real :: regrid_time_scale !< The time-scale used in blending between the current (old) grid
```

43.1.2.18 regridcs

```
type(regridding_cs) mom_ale::ale_cs::regridcs
```

Regridding parameters and work arrays.

Definition at line 71 of file MOM_ALE.F90.

```
00071   type(regridding_cs) :: regridcs !< Regridding parameters and work arrays
```

43.1.2.19 remap_after_initialization

```
logical mom_ale::ale_cs::remap_after_initialization
```

Indicates whether to regrid/remap after initializing the state.

Definition at line 76 of file MOM_ALE.F90.

```
00076   logical :: remap_after_initialization !< Indicates whether to regrid/remap after initializing the state.
```

43.1.2.20 remap_uv_using_old_alg

```
logical mom_ale::ale_cs::remap_uv_using_old_alg
```

If true, uses the old "remapping via a delta z" method. If False, uses the new method that remaps between grids described by h.

Definition at line 64 of file MOM_ALE.F90.

```
00064   logical :: remap_uv_using_old_alg !< If true, uses the old "remapping via a delta z"
```

43.1.2.21 remapcs

```
type(remapping_cs) mom_ale::ale_cs::remapcs
```

Remapping parameters and work arrays.

Definition at line 72 of file MOM_ALE.F90.

```
00072   type(remapping_cs)  :: remapcs  !< Remapping parameters and work arrays
```

43.1.2.22 show_call_tree

```
logical mom_ale::ale_cs::show_call_tree
```

For debugging.

Definition at line 82 of file MOM_ALE.F90.

```
00082   logical :: show_call_tree !< For debugging
```

The documentation for this type was generated from the following file:

- /home/cermak/src/MOM6/src/ALE/MOM_ALE.F90

43.2 mom_eos::calculate_compress Interface Reference

Calculates the compressibility of water from T, S, and P.

Private Member Functions

- subroutine calculate_compress_scalar (T, S, pressure, rho, drho_dp, EOS)

  Calculate density and compressibility for a scalar. This just promotes the scalar to an array with a singleton dimension and calls calculate_compress_array. If US is present, the units of the inputs and outputs are rescaled.
- subroutine calculate_compress_array (T, S, press, rho, drho_dp, start, npts, EOS)

  Calls the appropriate subroutine to calculate the density and compressibility for 1-D array inputs. If US is present, the units of the inputs and outputs are rescaled.

43.2.1 Detailed Description

Calculates the compressibility of water from T, S, and P.

Definition at line 103 of file MOM_EOS.F90.

43.2.2 Member Function/Subroutine Documentation

43.2.2.1 calculate_compress_array()

```
subroutine mom_eos::calculate_compress::calculate_compress_array (
             real, dimension(:), intent(in) T,
             real, dimension(:), intent(in) S,
             real, dimension(:), intent(in) press,
             real, dimension(:), intent(inout) rho,
             real, dimension(:), intent(inout) drho_dp,
             integer, intent(in) start,
             integer, intent(in) npts,
             type(eos_type), pointer EOS )  [private]
```

Calls the appropriate subroutine to calculate the density and compressibility for 1-D array inputs. If US is present, the units of the inputs and outputs are rescaled.

Parameters

in	*t*	Potential temperature referenced to the surface [degC]
in	*s*	Salinity [PSU]
in	*press*	Pressure [Pa] or [R L2 T-2 ~> Pa]
in,out	*rho*	In situ density [kg m-3] or [R ~> kg m-3]
in,out	*drho_dp*	The partial derivative of density with pressure (also the inverse of the square of sound speed) [s2 m-2] or [T2 L-2]
in	*start*	Starting index within the array
in	*npts*	The number of values to calculate
	eos	Equation of state structure

Definition at line 1089 of file MOM_EOS.F90.

```
01089   real, dimension(:), intent(in)  :: t        !< Potential temperature referenced to the surface [degC]
01090   real, dimension(:), intent(in)  :: s        !< Salinity [PSU]
01091   real, dimension(:), intent(in)  :: press    !< Pressure [Pa] or [R L2 T-2 ~> Pa]
01092   real, dimension(:), intent(inout) :: rho      !< In situ density [kg m-3] or [R ~> kg m-3]
01093   real, dimension(:), intent(inout) :: drho_dp  !< The partial derivative of density with pressure
01094                                                 !! (also the inverse of the square of sound speed)
01095                                                 !! [s2 m-2] or [T2 L-2]
01096   integer,            intent(in)  :: start    !< Starting index within the array
01097   integer,            intent(in)  :: npts     !< The number of values to calculate
01098   type(eos_type),     pointer     :: eos      !< Equation of state structure
01099
01100   ! Local variables
01101   real, dimension(size(press)) :: pressure  ! Pressure converted to [Pa]
01102   integer :: i, is, ie
01103
01104   if (.not.associated(eos)) call mom_error(fatal, &
01105     "calculate_compress called with an unassociated EOS_type EOS.")
01106
01107   is = start ; ie = is + npts - 1
01108   do i=is,ie ; pressure(i) = eos%RL2_T2_to_Pa * press(i) ; enddo
01109
01110   select case (eos%form_of_EOS)
01111     case (eos_linear)
01112       call calculate_compress_linear(t, s, pressure, rho, drho_dp, start, npts, &
01113                                      eos%Rho_T0_S0, eos%dRho_dT, eos%dRho_dS)
01114     case (eos_unesco)
01115       call calculate_compress_unesco(t, s, pressure, rho, drho_dp, start, npts)
01116     case (eos_wright)
01117       call calculate_compress_wright(t, s, pressure, rho, drho_dp, start, npts)
01118     case (eos_teos10)
01119       call calculate_compress_teos10(t, s, pressure, rho, drho_dp, start, npts)
01120     case (eos_nemo)
01121       call calculate_compress_nemo(t, s, pressure, rho, drho_dp, start, npts)
01122     case default
01123       call mom_error(fatal, "calculate_compress: EOS%form_of_EOS is not valid.")
01124   end select
01125
01126   if (eos%kg_m3_to_R /= 1.0) then ; do i=is,ie
01127     rho(i) = eos%kg_m3_to_R * rho(i)
01128   enddo ; endif
01129   if (eos%L_T_to_m_s /= 1.0) then ; do i=is,ie
01130     drho_dp(i) = eos%L_T_to_m_s**2 * drho_dp(i)
01131   enddo ; endif
01132
```

43.2.2.2 calculate_compress_scalar()

```
subroutine mom_eos::calculate_compress::calculate_compress_scalar (
             real, intent(in) T,
             real, intent(in) S,
             real, intent(in) pressure,
```

```
real, intent(out) rho,
real, intent(out) drho_dp,
type(eos_type), pointer EOS )  [private]
```

Calculate density and compressibility for a scalar. This just promotes the scalar to an array with a singleton dimension and calls calculate_compress_array. If US is present, the units of the inputs and outputs are rescaled.

Parameters

in	*t*	Potential temperature referenced to the surface [degC]
in	*s*	Salinity [ppt]
in	*pressure*	Pressure [Pa] or [R L2 T-2 ~> Pa]
out	*rho*	In situ density [kg m-3] or [R ~> kg m-3]
out	*drho_dp*	The partial derivative of density with pressure (also the inverse of the square of sound speed) [s2 m-2] or [T2 L-2]
	eos	Equation of state structure

Definition at line 1139 of file MOM_EOS.F90.

```
01139   real, intent(in)         :: t        !< Potential temperature referenced to the surface [degC]
01140   real, intent(in)         :: s        !< Salinity [ppt]
01141   real, intent(in)         :: pressure !< Pressure [Pa] or [R L2 T-2 ~> Pa]
01142   real, intent(out)        :: rho      !< In situ density [kg m-3] or [R ~> kg m-3]
01143   real, intent(out)        :: drho_dp  !< The partial derivative of density with pressure (also the
01144                                        !! inverse of the square of sound speed) [s2 m-2] or [T2 L-2]
01145   type(eos_type), pointer :: eos       !< Equation of state structure
01146
01147   ! Local variables
01148   real, dimension(1) :: ta, sa, pa, rhoa, drho_dpa
01149
01150   if (.not.associated(eos)) call mom_error(fatal, &
01151     "calculate_compress called with an unassociated EOS_type EOS.")
01152   ta(1) = t ; sa(1) = s; pa(1) = pressure
01153
01154   call calculate_compress_array(ta, sa, pa, rhoa, drho_dpa, 1, 1, eos)
01155   rho = rhoa(1) ; drho_dp = drho_dpa(1)
01156
```

The documentation for this interface was generated from the following file:

- /home/cermak/src/MOM6/src/equation_of_state/MOM_EOS.F90

43.3 mom_eos::calculate_density Interface Reference

Calculates density of sea water from T, S and P.

Private Member Functions

- subroutine calculate_density_scalar (T, S, pressure, rho, EOS, rho_ref, scale)

  Calls the appropriate subroutine to calculate density of sea water for scalar inputs. If rho_ref is present, the anomaly with respect to rho_ref is returned. The pressure and density can be rescaled with the US. If both the US and scale arguments are present the density scaling uses the product of the two scaling factors.
- subroutine calculate_density_array (T, S, pressure, rho, start, npts, EOS, rho_ref, scale)

  Calls the appropriate subroutine to calculate the density of sea water for 1-D array inputs. If rho_ref is present, the anomaly with respect to rho_ref is returned.

- subroutine calculate_density_1d (T, S, pressure, rho, EOS, dom, rho_ref, scale)

  Calls the appropriate subroutine to calculate the density of sea water for 1-D array inputs, potentially limiting the domain of indices that are worked on. If rho_ref is present, the anomaly with respect to rho_ref is returned.
- subroutine calculate_stanley_density_scalar (T, S, pressure, Tvar, TScov, Svar, rho, EOS, rho_ref, scale)

  Calls the appropriate subroutine to calculate density of sea water for scalar inputs including the variance of T, S and covariance of T-S. The calculation uses only the second order correction in a series as discussed in Stanley et al., 2020. If rho_ref is present, the anomaly with respect to rho_ref is returned. The density can be rescaled using rho_ref.
- subroutine calculate_stanley_density_array (T, S, pressure, Tvar, TScov, Svar, rho, start, npts, EOS, rho_ref, scale)

  Calls the appropriate subroutine to calculate the density of sea water for 1-D array inputs including the variance of T, S and covariance of T-S. The calculation uses only the second order correction in a series as discussed in Stanley et al., 2020. If rho_ref is present, the anomaly with respect to rho_ref is returned.
- subroutine calculate_stanley_density_1d (T, S, pressure, Tvar, TScov, Svar, rho, EOS, dom, rho_ref, scale)

  Calls the appropriate subroutine to calculate the density of sea water for 1-D array inputs including the variance of T, S and covariance of T-S, potentially limiting the domain of indices that are worked on. The calculation uses only the second order correction in a series as discussed in Stanley et al., 2020. If rho_ref is present, the anomaly with respect to rho_ref is returned.

43.3.1 Detailed Description

Calculates density of sea water from T, S and P.

Definition at line 68 of file MOM_EOS.F90.

43.3.2 Member Function/Subroutine Documentation

43.3.2.1 calculate_density_1d()

```
subroutine mom_eos::calculate_density::calculate_density_1d (
             real, dimension(:), intent(in) T,
             real, dimension(:), intent(in) S,
             real, dimension(:), intent(in) pressure,
             real, dimension(:), intent(inout) rho,
             type(eos_type), pointer EOS,
             integer, dimension(2), intent(in), optional dom,
             real, intent(in), optional rho_ref,
             real, intent(in), optional scale )  [private]
```

Calls the appropriate subroutine to calculate the density of sea water for 1-D array inputs, potentially limiting the domain of indices that are worked on. If rho_ref is present, the anomaly with respect to rho_ref is returned.

Parameters

in	*t*	Potential temperature referenced to the surface [degC]
in	*s*	Salinity [ppt]
in	*pressure*	Pressure [R L2 T-2 ~> Pa]
in,out	*rho*	Density (in-situ if pressure is local) [R ~> kg m-3]
	eos	Equation of state structure
in	*dom*	The domain of indices to work on, taking into account that arrays start at 1.
in	*rho_ref*	A reference density [kg m-3]
in	*scale*	A multiplicative factor by which to scale density in combination with scaling given by US [various]

Definition at line 360 of file MOM_EOS.F90.

```
00360   real, dimension(:),    intent(in)    :: t        !< Potential temperature referenced to the surface
       [degC]
00361   real, dimension(:),    intent(in)    :: s        !< Salinity [ppt]
00362   real, dimension(:),    intent(in)    :: pressure !< Pressure [R L2 T-2 ~> Pa]
00363   real, dimension(:),    intent(inout) :: rho      !< Density (in-situ if pressure is local) [R ~> kg m-3]
00364   type(eos_type),        pointer       :: eos      !< Equation of state structure
00365   integer, dimension(2), optional, intent(in) :: dom   !< The domain of indices to work on, taking
00366                                                        !! into account that arrays start at 1.
00367   real,                  optional, intent(in) :: rho_ref !< A reference density [kg m-3]
00368   real,                  optional, intent(in) :: scale !< A multiplicative factor by which to scale density
00369                                                        !! in combination with scaling given by US [various]
00370   ! Local variables
00371   real :: p_scale   ! A factor to convert pressure to units of Pa [Pa T2 R-1 L-2 ~> 1]
00372   real :: rho_scale ! A factor to convert density from kg m-3 to the desired units [R m3 kg-1 ~> 1]
00373   real :: rho_unscale ! A factor to convert density from R to kg m-3 [kg m-3 R-1 ~> 1]
00374   real :: rho_reference ! rho_ref converted to [kg m-3]
00375   real, dimension(size(rho)) :: pres  ! Pressure converted to [Pa]
00376   integer :: i, is, ie, npts
00377
00378   if (.not.associated(eos)) call mom_error(fatal, &
00379     "calculate_density_1d called with an unassociated EOS_type EOS.")
00380
00381   if (present(dom)) then
00382     is = dom(1) ; ie = dom(2) ; npts = 1 + ie - is
00383   else
00384     is = 1 ; ie = size(rho) ; npts = 1 + ie - is
00385   endif
00386
00387   p_scale = eos%RL2_T2_to_Pa
00388   rho_unscale = eos%R_to_kg_m3
00389
00390   if ((p_scale == 1.0) .and. (rho_unscale == 1.0)) then
00391     call calculate_density_array(t, s, pressure, rho, is, npts, eos, rho_ref=rho_ref)
00392   elseif (present(rho_ref)) then ! This is the same as above, but with some extra work to rescale
       variables.
00393     do i=is,ie ; pres(i) = p_scale * pressure(i) ; enddo
00394     rho_reference = rho_unscale*rho_ref
00395     call calculate_density_array(t, s, pres, rho, is, npts, eos, rho_ref=rho_reference)
00396   else  ! There is rescaling of variables, but rho_ref is not present. Passing a 0 value of rho_ref
00397         ! changes answers at roundoff for some equations of state, like Wright and UNESCO.
00398     do i=is,ie ; pres(i) = p_scale * pressure(i) ; enddo
00399     call calculate_density_array(t, s, pres, rho, is, npts, eos)
00400   endif
00401
00402   rho_scale = eos%kg_m3_to_R
00403   if (present(scale)) rho_scale = rho_scale * scale
00404   if (rho_scale /= 1.0) then ; do i=is,ie
00405     rho(i) = rho_scale * rho(i)
00406   enddo ; endif
00407
```

43.3.2.2 calculate_density_array()

```
subroutine mom_eos::calculate_density::calculate_density_array (
             real, dimension(:), intent(in) T,
             real, dimension(:), intent(in) S,
             real, dimension(:), intent(in) pressure,
             real, dimension(:), intent(inout) rho,
             integer, intent(in) start,
             integer, intent(in) npts,
             type(eos_type), pointer EOS,
             real, intent(in), optional rho_ref,
             real, intent(in), optional scale )  [private]
```

Calls the appropriate subroutine to calculate the density of sea water for 1-D array inputs. If rho_ref is present, the anomaly with respect to rho_ref is returned.

Parameters

in	*t*	Potential temperature referenced to the surface [degC]
in	*s*	Salinity [ppt]
in	*pressure*	Pressure [Pa] or [R L2 T-2 ~> Pa]
in,out	*rho*	Density (in-situ if pressure is local) [kg m-3] or [R ~> kg m-3]
in	*start*	Start index for computation
in	*npts*	Number of point to compute
	eos	Equation of state structure
in	*rho_ref*	A reference density [kg m-3]
in	*scale*	A multiplicative factor by which to scale density in combination with scaling given by US [various]

Definition at line 263 of file MOM_EOS.F90.

```
00263   real, dimension(:), intent(in)    :: t        !< Potential temperature referenced to the surface [degC]
00264   real, dimension(:), intent(in)    :: s        !< Salinity [ppt]
00265   real, dimension(:), intent(in)    :: pressure !< Pressure [Pa] or [R L2 T-2 ~> Pa]
00266   real, dimension(:), intent(inout) :: rho      !< Density (in-situ if pressure is local) [kg m-3] or [R ~>
      kg m-3]
00267   integer,            intent(in)    :: start    !< Start index for computation
00268   integer,            intent(in)    :: npts     !< Number of point to compute
00269   type(eos_type),     pointer       :: eos      !< Equation of state structure
00270   real,                  optional, intent(in) :: rho_ref  !< A reference density [kg m-3]
00271   real,                  optional, intent(in) :: scale    !< A multiplicative factor by which to scale
      density
00272                                                 !! in combination with scaling given by US [various]
00273   integer :: j
00274
00275   if (.not.associated(eos)) call mom_error(fatal, &
00276     "calculate_density_array called with an unassociated EOS_type EOS.")
00277
00278   select case (eos%form_of_EOS)
00279     case (eos_linear)
00280       call calculate_density_linear(t, s, pressure, rho, start, npts, &
00281                                     eos%Rho_T0_S0, eos%dRho_dT, eos%dRho_dS, rho_ref)
00282     case (eos_unesco)
00283       call calculate_density_unesco(t, s, pressure, rho, start, npts, rho_ref)
00284     case (eos_wright)
00285       call calculate_density_wright(t, s, pressure, rho, start, npts, rho_ref)
00286     case (eos_teos10)
00287       call calculate_density_teos10(t, s, pressure, rho, start, npts, rho_ref)
00288     case (eos_nemo)
00289     call calculate_density_nemo(t, s, pressure, rho, start, npts, rho_ref)
00290     case default
00291       call mom_error(fatal, "calculate_density_array: EOS%form_of_EOS is not valid.")
00292   end select
00293
00294   if (present(scale)) then ; if (scale /= 1.0) then ; do j=start,start+npts-1
00295     rho(j) = scale * rho(j)
00296   enddo ; endif ; endif
00297
```

43.3.2.3 calculate_density_scalar()

```
subroutine mom_eos::calculate_density::calculate_density_scalar (
             real, intent(in) T,
             real, intent(in) S,
             real, intent(in) pressure,
             real, intent(out) rho,
             type(eos_type), pointer EOS,
             real, intent(in), optional rho_ref,
             real, intent(in), optional scale )  [private]
```

Calls the appropriate subroutine to calculate density of sea water for scalar inputs. If rho_ref is present, the anomaly with respect to rho_ref is returned. The pressure and density can be rescaled with the US. If both the US and scale arguments are present the density scaling uses the product of the two scaling factors.

Parameters

in	*t*	Potential temperature referenced to the surface [degC]
in	*s*	Salinity [ppt]
in	*pressure*	Pressure [Pa] or [R L2 T-2 ~> Pa]
out	*rho*	Density (in-situ if pressure is local) [kg m-3] or [R ~> kg m-3]
	eos	Equation of state structure
in	*rho_ref*	A reference density [kg m-3]
in	*scale*	A multiplicative factor by which to scale density in combination with scaling given by US [various]

Definition at line 166 of file MOM_EOS.F90.

```
00166   real,           intent(in)  :: t        !< Potential temperature referenced to the surface [degC]
00167   real,           intent(in)  :: s        !< Salinity [ppt]
00168   real,           intent(in)  :: pressure !< Pressure [Pa] or [R L2 T-2 ~> Pa]
00169   real,           intent(out) :: rho      !< Density (in-situ if pressure is local) [kg m-3] or [R ~> kg
      m-3]
00170   type(eos_type), pointer     :: eos      !< Equation of state structure
00171   real, optional, intent(in)  :: rho_ref  !< A reference density [kg m-3]
00172   real, optional, intent(in)  :: scale    !< A multiplicative factor by which to scale density in
00173                                           !! combination with scaling given by US [various]
00174
00175   real :: rho_scale ! A factor to convert density from kg m-3 to the desired units [R m3 kg-1 ~> 1]
00176   real :: p_scale   ! A factor to convert pressure to units of Pa [Pa T2 R-1 L-2 ~> 1]
00177
00178   if (.not.associated(eos)) call mom_error(fatal, &
00179     "calculate_density_scalar called with an unassociated EOS_type EOS.")
00180
00181   p_scale = eos%RL2_T2_to_Pa
00182
00183   select case (eos%form_of_EOS)
00184     case (eos_linear)
00185       call calculate_density_linear(t, s, p_scale*pressure, rho, &
00186                                     eos%Rho_T0_S0, eos%dRho_dT, eos%dRho_dS, rho_ref)
00187     case (eos_unesco)
00188       call calculate_density_unesco(t, s, p_scale*pressure, rho, rho_ref)
00189     case (eos_wright)
00190       call calculate_density_wright(t, s, p_scale*pressure, rho, rho_ref)
00191     case (eos_teos10)
00192       call calculate_density_teos10(t, s, p_scale*pressure, rho, rho_ref)
00193     case (eos_nemo)
00194       call calculate_density_nemo(t, s, p_scale*pressure, rho, rho_ref)
00195     case default
00196       call mom_error(fatal, "calculate_density_scalar: EOS is not valid.")
00197   end select
00198
00199   rho_scale = eos%kg_m3_to_R
00200   if (present(scale)) rho_scale = rho_scale * scale
00201   rho = rho_scale * rho
00202
```

43.3.2.4 calculate_stanley_density_1d()

```
subroutine mom_eos::calculate_density::calculate_stanley_density_1d (
             real, dimension(:), intent(in) T,
             real, dimension(:), intent(in) S,
             real, dimension(:), intent(in) pressure,
             real, dimension(:), intent(in) Tvar,
             real, dimension(:), intent(in) TScov,
             real, dimension(:), intent(in) Svar,
             real, dimension(:), intent(inout) rho,
             type(eos_type), pointer EOS,
```

```
integer, dimension(2), intent(in), optional dom,
real, intent(in), optional rho_ref,
real, intent(in), optional scale )  [private]
```

Calls the appropriate subroutine to calculate the density of sea water for 1-D array inputs including the variance of T, S and covariance of T-S, potentially limiting the domain of indices that are worked on. The calculation uses only the second order correction in a series as discussed in Stanley et al., 2020. If rho_ref is present, the anomaly with respect to rho_ref is returned.

Parameters

in	*t*	Potential temperature referenced to the surface [degC]
in	*s*	Salinity [ppt]
in	*pressure*	Pressure [R L2 T-2 ~> Pa]
in	*tvar*	Variance of potential temperature [degC2]
in	*tscov*	Covariance of potential temperature and salinity [degC ppt]
in	*svar*	Variance of salinity [ppt2]
in,out	*rho*	Density (in-situ if pressure is local) [R ~> kg m-3]
	eos	Equation of state structure
in	*dom*	The domain of indices to work on, taking into account that arrays start at 1.
in	*rho_ref*	A reference density [kg m-3]
in	*scale*	A multiplicative factor by which to scale density in combination with scaling given by US [various]

Definition at line 417 of file MOM_EOS.F90.

```
00417   real, dimension(:),    intent(in)    :: t        !< Potential temperature referenced to the surface
      [degC]
00418   real, dimension(:),    intent(in)    :: s        !< Salinity [ppt]
00419   real, dimension(:),    intent(in)    :: pressure !< Pressure [R L2 T-2 ~> Pa]
00420   real, dimension(:),    intent(in)    :: tvar     !< Variance of potential temperature [degC2]
00421   real, dimension(:),    intent(in)    :: tscov    !< Covariance of potential temperature and salinity
      [degC ppt]
00422   real, dimension(:),    intent(in)    :: svar     !< Variance of salinity [ppt2]
00423   real, dimension(:),    intent(inout) :: rho      !< Density (in-situ if pressure is local) [R ~> kg m-3]
00424   type(eos_type),        pointer       :: eos      !< Equation of state structure
00425   integer, dimension(2), optional, intent(in) :: dom   !< The domain of indices to work on, taking
00426                                                        !! into account that arrays start at 1.
00427   real,                  optional, intent(in) :: rho_ref !< A reference density [kg m-3]
00428   real,                  optional, intent(in) :: scale !< A multiplicative factor by which to scale density
00429                                                        !! in combination with scaling given by US [various]
00430   ! Local variables
00431   real :: p_scale   ! A factor to convert pressure to units of Pa [Pa T2 R-1 L-2 ~> 1]
00432   real :: rho_scale ! A factor to convert density from kg m-3 to the desired units [R m3 kg-1 ~> 1]
00433   real, dimension(size(rho)) :: pres  ! Pressure converted to [Pa]
00434   real, dimension(size(T)) :: d2rdtt, d2rdst, d2rdss, d2rdsp, d2rdtp ! Second derivatives of density wrt
      T,S,p
00435   integer :: i, is, ie, npts
00436
00437   if (.not.associated(eos)) call mom_error(fatal, &
00438     "calculate_density_1d called with an unassociated EOS_type EOS.")
00439
00440   if (present(dom)) then
00441     is = dom(1) ; ie = dom(2) ; npts = 1 + ie - is
00442   else
00443     is = 1 ; ie = size(rho) ; npts = 1 + ie - is
00444   endif
00445
00446   p_scale = eos%RL2_T2_to_Pa
00447   do i=is,ie
00448     pres(i) = p_scale * pressure(i)
00449   enddo
00450
00451   select case (eos%form_of_EOS)
00452     case (eos_linear)
00453       call calculate_density_linear(t, s, pres, rho, 1, npts, &
00454                                     eos%Rho_T0_S0, eos%dRho_dT, eos%dRho_dS, rho_ref)
00455       call calculate_density_second_derivs_linear(t, s, pres, d2rdss, d2rdst, &
```

```
00456                                                 d2rdtt, d2rdsp, d2rdtp, 1, npts)
00457     case (eos_wright)
00458       call calculate_density_wright(t, s, pres, rho, 1, npts, rho_ref)
00459       call calculate_density_second_derivs_wright(t, s, pres, d2rdss, d2rdst, &
00460                                                 d2rdtt, d2rdsp, d2rdtp, 1, npts)
00461     case (eos_teos10)
00462       call calculate_density_teos10(t, s, pres, rho, 1, npts, rho_ref)
00463       call calculate_density_second_derivs_teos10(t, s, pres, d2rdss, d2rdst, &
00464                                                 d2rdtt, d2rdsp, d2rdtp, 1, npts)
00465     case default
00466       call mom_error(fatal, "calculate_stanley_density_scalar: EOS is not valid.")
00467   end select
00468
00469   ! Equation 25 of Stanley et al., 2020.
00470   do i=is,ie
00471     rho(i) = rho(i) &
00472              + ( 0.5 * d2rdtt(i) * tvar(i) + ( d2rdst(i) * tscov(i) + 0.5 * d2rdss(i) * svar(i) ) )
00473   enddo
00474
00475   rho_scale = eos%kg_m3_to_R
00476   if (present(scale)) rho_scale = rho_scale * scale
00477   if (rho_scale /= 1.0) then ; do i=is,ie
00478     rho(i) = rho_scale * rho(i)
00479   enddo ; endif
00480
```

43.3.2.5 calculate_stanley_density_array()

```
subroutine mom_eos::calculate_density::calculate_stanley_density_array (
             real, dimension(:), intent(in) T,
             real, dimension(:), intent(in) S,
             real, dimension(:), intent(in) pressure,
             real, dimension(:), intent(in) Tvar,
             real, dimension(:), intent(in) TScov,
             real, dimension(:), intent(in) Svar,
             real, dimension(:), intent(inout) rho,
             integer, intent(in) start,
             integer, intent(in) npts,
             type(eos_type), pointer EOS,
             real, intent(in), optional rho_ref,
             real, intent(in), optional scale )  [private]
```

Calls the appropriate subroutine to calculate the density of sea water for 1-D array inputs including the variance of T, S and covariance of T-S. The calculation uses only the second order correction in a series as discussed in Stanley et al., 2020. If rho_ref is present, the anomaly with respect to rho_ref is returned.

Parameters

in	*t*	Potential temperature referenced to the surface [degC]
in	*s*	Salinity [ppt]
in	*pressure*	Pressure [Pa]
in	*tvar*	Variance of potential temperature referenced to the surface [degC2]
in	*tscov*	Covariance of potential temperature and salinity [degC ppt]
in	*svar*	Variance of salinity [ppt2]
in,out	*rho*	Density (in-situ if pressure is local) [kg m-3]
in	*start*	Start index for computation
in	*npts*	Number of point to compute
	eos	Equation of state structure
in	*rho_ref*	A reference density [kg m-3].
in	*scale*	A multiplicative factor by which to scale density from kg m-3 to the desired units [R m3 kg-1]

Definition at line 306 of file MOM_EOS.F90.

```
00306   real, dimension(:), intent(in)    :: t        !< Potential temperature referenced to the surface [degC]
00307   real, dimension(:), intent(in)    :: s        !< Salinity [ppt]
00308   real, dimension(:), intent(in)    :: pressure !< Pressure [Pa]
00309   real, dimension(:), intent(in)    :: tvar     !< Variance of potential temperature referenced to the
      surface [degC2]
00310   real, dimension(:), intent(in)    :: tscov    !< Covariance of potential temperature and salinity [degC
      ppt]
00311   real, dimension(:), intent(in)    :: svar     !< Variance of salinity [ppt2]
00312   real, dimension(:), intent(inout) :: rho      !< Density (in-situ if pressure is local) [kg m-3]
00313   integer,            intent(in)    :: start    !< Start index for computation
00314   integer,            intent(in)    :: npts     !< Number of point to compute
00315   type(eos_type),     pointer       :: eos      !< Equation of state structure
00316   real,     optional, intent(in)    :: rho_ref  !< A reference density [kg m-3].
00317   real,     optional, intent(in)    :: scale    !< A multiplicative factor by which to scale density
00318                                                 !! from kg m-3 to the desired units [R m3 kg-1]
00319   ! Local variables
00320   real, dimension(size(T)) :: d2rdtt, d2rdst, d2rdss, d2rdsp, d2rdtp ! Second derivatives of density wrt
      T,S,p
00321   integer :: j
00322
00323   if (.not.associated(eos)) call mom_error(fatal, &
00324     "calculate_density_array called with an unassociated EOS_type EOS.")
00325
00326   select case (eos%form_of_EOS)
00327     case (eos_linear)
00328       call calculate_density_linear(t, s, pressure, rho, start, npts, &
00329                                     eos%Rho_T0_S0, eos%dRho_dT, eos%dRho_dS, rho_ref)
00330       call calculate_density_second_derivs_linear(t, s, pressure, d2rdss, d2rdst, &
00331                                                   d2rdtt, d2rdsp, d2rdtp, start, npts)
00332     case (eos_wright)
00333       call calculate_density_wright(t, s, pressure, rho, start, npts, rho_ref)
00334       call calculate_density_second_derivs_wright(t, s, pressure, d2rdss, d2rdst, &
00335                                                   d2rdtt, d2rdsp, d2rdtp, start, npts)
00336     case (eos_teos10)
00337       call calculate_density_teos10(t, s, pressure, rho, start, npts, rho_ref)
00338       call calculate_density_second_derivs_teos10(t, s, pressure, d2rdss, d2rdst, &
00339                                                   d2rdtt, d2rdsp, d2rdtp, start, npts)
00340     case default
00341       call mom_error(fatal, "calculate_stanley_density_array: EOS%form_of_EOS is not valid.")
00342   end select
00343
00344   ! Equation 25 of Stanley et al., 2020.
00345   do j=start,start+npts-1
00346     rho(j) = rho(j) &
00347              + ( 0.5 * d2rdtt(j) * tvar(j) + ( d2rdst(j) * tscov(j) + 0.5 * d2rdss(j) * svar(j) ) )
00348   enddo
00349
00350   if (present(scale)) then ; if (scale /= 1.0) then ; do j=start,start+npts-1
00351     rho(j) = scale * rho(j)
00352   enddo ; endif ; endif
00353
```

43.3.2.6 calculate_stanley_density_scalar()

```
subroutine mom_eos::calculate_density::calculate_stanley_density_scalar (
              real, intent(in) T,
              real, intent(in) S,
              real, intent(in) pressure,
              real, intent(in) Tvar,
              real, intent(in) TScov,
              real, intent(in) Svar,
              real, intent(out) rho,
              type(eos_type), pointer EOS,
              real, intent(in), optional rho_ref,
              real, intent(in), optional scale )  [private]
```

Calls the appropriate subroutine to calculate density of sea water for scalar inputs including the variance of T, S and covariance of T-S. The calculation uses only the second order correction in a series as discussed in Stanley et al., 2020. If rho_ref is present, the anomaly with respect to rho_ref is returned. The density can be rescaled using rho_ref.

Parameters

in	*t*	Potential temperature referenced to the surface [degC]
in	*s*	Salinity [ppt]
in	*tvar*	Variance of potential temperature referenced to the surface [degC2]
in	*tscov*	Covariance of potential temperature and salinity [degC ppt]
in	*svar*	Variance of salinity [ppt2]
in	*pressure*	Pressure [Pa]
out	*rho*	Density (in-situ if pressure is local) [kg m-3] or [R ~> kg m-3]
	eos	Equation of state structure
in	*rho_ref*	A reference density [kg m-3].
in	*scale*	A multiplicative factor by which to scale density from kg m-3 to the desired units [R m3 kg-1]

Definition at line 212 of file MOM_EOS.F90.

```
00212   real,           intent(in)  :: t        !< Potential temperature referenced to the surface [degC]
00213   real,           intent(in)  :: s        !< Salinity [ppt]
00214   real,           intent(in)  :: tvar     !< Variance of potential temperature referenced to the surface
      [degC2]
00215   real,           intent(in)  :: tscov    !< Covariance of potential temperature and salinity [degC ppt]
00216   real,           intent(in)  :: svar     !< Variance of salinity [ppt2]
00217   real,           intent(in)  :: pressure !< Pressure [Pa]
00218   real,           intent(out) :: rho      !< Density (in-situ if pressure is local) [kg m-3] or [R ~> kg
      m-3]
00219   type(eos_type), pointer     :: eos      !< Equation of state structure
00220   real, optional, intent(in)  :: rho_ref  !< A reference density [kg m-3].
00221   real, optional, intent(in)  :: scale    !< A multiplicative factor by which to scale density
00222                                           !! from kg m-3 to the desired units [R m3 kg-1]
00223   ! Local variables
00224   real :: d2rdtt, d2rdst, d2rdss, d2rdsp, d2rdtp ! Second derivatives of density wrt T,S,p
00225   real :: rho_scale ! A factor to convert density from kg m-3 to the desired units [R m3 kg-1 ~> 1]
00226   real :: p_scale   ! A factor to convert pressure to units of Pa [Pa T2 R-1 L-2 ~> 1]
00227
00228   if (.not.associated(eos)) call mom_error(fatal, &
00229     "calculate_stanley_density_scalar called with an unassociated EOS_type EOS.")
00230
00231   p_scale = eos%RL2_T2_to_Pa
00232
00233   select case (eos%form_of_EOS)
00234     case (eos_linear)
00235       call calculate_density_linear(t, s, p_scale*pressure, rho, &
00236                                     eos%Rho_T0_S0, eos%dRho_dT, eos%dRho_dS, rho_ref)
00237       call calculate_density_second_derivs_linear(t, s, pressure, d2rdss, d2rdst, &
00238                                                   d2rdtt, d2rdsp, d2rdtp)
00239     case (eos_wright)
00240       call calculate_density_wright(t, s, p_scale*pressure, rho, rho_ref)
00241       call calculate_density_second_derivs_wright(t, s, pressure, d2rdss, d2rdst, &
00242                                                   d2rdtt, d2rdsp, d2rdtp)
00243     case (eos_teos10)
00244       call calculate_density_teos10(t, s, p_scale*pressure, rho, rho_ref)
00245       call calculate_density_second_derivs_teos10(t, s, pressure, d2rdss, d2rdst, &
00246                                                   d2rdtt, d2rdsp, d2rdtp)
00247     case default
00248       call mom_error(fatal, "calculate_stanley_density_scalar: EOS is not valid.")
00249   end select
00250
00251   ! Equation 25 of Stanley et al., 2020.
00252   rho = rho + ( 0.5 * d2rdtt * tvar + ( d2rdst * tscov + 0.5 * d2rdss * svar ) )
00253
00254   rho_scale = eos%kg_m3_to_R
00255   if (present(scale)) rho_scale = rho_scale * scale
00256   rho = rho_scale * rho
00257
```

The documentation for this interface was generated from the following file:

- /home/cermak/src/MOM6/src/equation_of_state/MOM_EOS.F90

43.4 mom_eos::calculate_density_derivs Interface Reference

Calculate the derivatives of density with temperature and salinity from T, S, and P.

Private Member Functions

- subroutine calculate_density_derivs_scalar (T, S, pressure, drho_dT, drho_dS, EOS, scale)

  Calls the appropriate subroutines to calculate density derivatives by promoting a scalar to a one-element array.
- subroutine calculate_density_derivs_array (T, S, pressure, drho_dT, drho_dS, start, npts, EOS, scale)

  Calls the appropriate subroutine to calculate density derivatives for 1-D array inputs.
- subroutine calculate_density_derivs_1d (T, S, pressure, drho_dT, drho_dS, EOS, dom, scale)

  Calls the appropriate subroutine to calculate density derivatives for 1-D array inputs.

43.4.1 Detailed Description

Calculate the derivatives of density with temperature and salinity from T, S, and P.

Definition at line 81 of file MOM_EOS.F90.

43.4.2 Member Function/Subroutine Documentation

43.4.2.1 calculate_density_derivs_1d()

```
subroutine mom_eos::calculate_density_derivs::calculate_density_derivs_1d (
             real, dimension(:), intent(in) T,
             real, dimension(:), intent(in) S,
             real, dimension(:), intent(in) pressure,
             real, dimension(:), intent(inout) drho_dT,
             real, dimension(:), intent(inout) drho_dS,
             type(eos_type), pointer EOS,
             integer, dimension(2), intent(in), optional dom,
             real, intent(in), optional scale )  [private]
```

Calls the appropriate subroutine to calculate density derivatives for 1-D array inputs.

Parameters

in	*t*	Potential temperature referenced to the surface [degC]
in	*s*	Salinity [ppt]
in	*pressure*	Pressure [R L2 T-2 ~> Pa]
in,out	*drho_dt*	The partial derivative of density with potential temperature [R degC-1 ~> kg m-3 degC-1]
in,out	*drho_ds*	The partial derivative of density with salinity [R degC-1 ~> kg m-3 ppt-1]
	eos	Equation of state structure
in	*dom*	The domain of indices to work on, taking into account that arrays start at 1.
in	*scale*	A multiplicative factor by which to scale density in combination with scaling given by US [various]

Definition at line 751 of file MOM_EOS.F90.

```
00751   real, dimension(:),    intent(in)    :: t        !< Potential temperature referenced to the surface
       [degC]
00752   real, dimension(:),    intent(in)    :: s        !< Salinity [ppt]
00753   real, dimension(:),    intent(in)    :: pressure !< Pressure [R L2 T-2 ~> Pa]
00754   real, dimension(:),    intent(inout) :: drho_dt  !< The partial derivative of density with potential
00755                                                      !! temperature [R degC-1 ~> kg m-3 degC-1]
00756   real, dimension(:),    intent(inout) :: drho_ds  !< The partial derivative of density with salinity
00757                                                      !! [R degC-1 ~> kg m-3 ppt-1]
00758   type(eos_type),        pointer       :: eos      !< Equation of state structure
00759   integer, dimension(2), optional, intent(in) :: dom   !< The domain of indices to work on, taking
00760                                                        !! into account that arrays start at 1.
00761   real,                  optional, intent(in) :: scale !< A multiplicative factor by which to scale density
00762                                                        !! in combination with scaling given by US [various]
00763   ! Local variables
00764   real, dimension(size(drho_dT)) :: pres  ! Pressure converted to [Pa]
00765   real :: rho_scale ! A factor to convert density from kg m-3 to the desired units [R m3 kg-1 ~> 1]
00766   real :: p_scale   ! A factor to convert pressure to units of Pa [Pa T2 R-1 L-2 ~> 1]
00767   integer :: i, is, ie, npts
00768
00769   if (.not.associated(eos)) call mom_error(fatal, &
00770     "calculate_density_derivs called with an unassociated EOS_type EOS.")
00771
00772   if (present(dom)) then
00773     is = dom(1) ; ie = dom(2) ; npts = 1 + ie - is
00774   else
00775     is = 1 ; ie = size(drho_dt) ; npts = 1 + ie - is
00776   endif
00777
00778   p_scale = eos%RL2_T2_to_Pa
00779
00780   if (p_scale == 1.0) then
00781     call calculate_density_derivs_array(t, s, pressure, drho_dt, drho_ds, is, npts, eos)
00782   else
00783     do i=is,ie ; pres(i) = p_scale * pressure(i) ; enddo
00784     call calculate_density_derivs_array(t, s, pres, drho_dt, drho_ds, is, npts, eos)
00785   endif
00786
00787   rho_scale = eos%kg_m3_to_R
00788   if (present(scale)) rho_scale = rho_scale * scale
00789   if (rho_scale /= 1.0) then ; do i=is,ie
00790     drho_dt(i) = rho_scale * drho_dt(i)
00791     drho_ds(i) = rho_scale * drho_ds(i)
00792   enddo ; endif
00793
```

43.4.2.2 calculate_density_derivs_array()

```
subroutine mom_eos::calculate_density_derivs::calculate_density_derivs_array (
             real, dimension(:), intent(in) T,
             real, dimension(:), intent(in) S,
             real, dimension(:), intent(in) pressure,
             real, dimension(:), intent(inout) drho_dT,
             real, dimension(:), intent(inout) drho_dS,
             integer, intent(in) start,
             integer, intent(in) npts,
             type(eos_type), pointer EOS,
             real, intent(in), optional scale )  [private]
```

Calls the appropriate subroutine to calculate density derivatives for 1-D array inputs.

Parameters

in	*t*	Potential temperature referenced to the surface [degC]
in	*s*	Salinity [ppt]
in	*pressure*	Pressure [Pa] or [R L2 T-2 ~> Pa]

Parameters

in,out	*drho_dt*	The partial derivative of density with potential temperature [kg m-3 degC-1] or [R degC-1 ~> kg m-3 degC-1]
in,out	*drho_ds*	The partial derivative of density with salinity, in [kg m-3 ppt-1] or [R degC-1 ~> kg m-3 ppt-1]
in	*start*	Starting index within the array
in	*npts*	The number of values to calculate
	eos	Equation of state structure
in	*scale*	A multiplicative factor by which to scale density in combination with scaling given by US [various]

Definition at line 706 of file MOM_EOS.F90.

```
00706   real, dimension(:), intent(in)    :: t        !< Potential temperature referenced to the surface [degC]
00707   real, dimension(:), intent(in)    :: s        !< Salinity [ppt]
00708   real, dimension(:), intent(in)    :: pressure !< Pressure [Pa] or [R L2 T-2 ~> Pa]
00709   real, dimension(:), intent(inout) :: drho_dt  !< The partial derivative of density with potential
00710                                                 !! temperature [kg m-3 degC-1] or [R degC-1 ~> kg m-3
      degC-1]
00711   real, dimension(:), intent(inout) :: drho_ds  !< The partial derivative of density with salinity,
00712                                                 !! in [kg m-3 ppt-1] or [R degC-1 ~> kg m-3 ppt-1]
00713   integer,            intent(in)    :: start    !< Starting index within the array
00714   integer,            intent(in)    :: npts     !< The number of values to calculate
00715   type(eos_type),     pointer       :: eos      !< Equation of state structure
00716   real,     optional, intent(in)    :: scale !< A multiplicative factor by which to scale density
00717                                                 !! in combination with scaling given by US [various]
00718
00719   ! Local variables
00720   integer :: j
00721
00722   if (.not.associated(eos)) call mom_error(fatal, &
00723     "calculate_density_derivs called with an unassociated EOS_type EOS.")
00724
00725   select case (eos%form_of_EOS)
00726     case (eos_linear)
00727       call calculate_density_derivs_linear(t, s, pressure, drho_dt, drho_ds, eos%Rho_T0_S0, &
00728                                            eos%dRho_dT, eos%dRho_dS, start, npts)
00729     case (eos_unesco)
00730       call calculate_density_derivs_unesco(t, s, pressure, drho_dt, drho_ds, start, npts)
00731     case (eos_wright)
00732       call calculate_density_derivs_wright(t, s, pressure, drho_dt, drho_ds, start, npts)
00733     case (eos_teos10)
00734       call calculate_density_derivs_teos10(t, s, pressure, drho_dt, drho_ds, start, npts)
00735     case (eos_nemo)
00736       call calculate_density_derivs_nemo(t, s, pressure, drho_dt, drho_ds, start, npts)
00737     case default
00738       call mom_error(fatal, "calculate_density_derivs_array: EOS%form_of_EOS is not valid.")
00739   end select
00740
00741   if (present(scale)) then ; if (scale /= 1.0) then ; do j=start,start+npts-1
00742     drho_dt(j) = scale * drho_dt(j)
00743     drho_ds(j) = scale * drho_ds(j)
00744   enddo ; endif ; endif
00745
```

43.4.2.3 calculate_density_derivs_scalar()

```
subroutine mom_eos::calculate_density_derivs::calculate_density_derivs_scalar (
              real, intent(in) T,
              real, intent(in) S,
              real, intent(in) pressure,
              real, intent(out) drho_dT,
              real, intent(out) drho_dS,
              type(eos_type), pointer EOS,
              real, intent(in), optional scale )  [private]
```

Calls the appropriate subroutines to calculate density derivatives by promoting a scalar to a one-element array.

Parameters

in	*t*	Potential temperature referenced to the surface [degC]
in	*s*	Salinity [ppt]
in	*pressure*	Pressure [Pa] or [R L2 T-2 ~> Pa]
out	*drho_dt*	The partial derivative of density with potential temperature [kg m-3 degC-1] or [R degC-1 ~> kg m-3 degC-1]
out	*drho_ds*	The partial derivative of density with salinity, in [kg m-3 ppt-1] or [R ppt-1 ~> kg m-3 ppt-1]
	eos	Equation of state structure
in	*scale*	A multiplicative factor by which to scale density in combination with scaling given by US [various]

Definition at line 800 of file MOM_EOS.F90.

```
00800   real,           intent(in)  :: t !< Potential temperature referenced to the surface [degC]
00801   real,           intent(in)  :: s !< Salinity [ppt]
00802   real,           intent(in)  :: pressure !< Pressure [Pa] or [R L2 T-2 ~> Pa]
00803   real,           intent(out) :: drho_dt !< The partial derivative of density with potential
00804                                  !! temperature [kg m-3 degC-1] or [R degC-1 ~> kg m-3 degC-1]
00805   real,           intent(out) :: drho_ds !< The partial derivative of density with salinity,
00806                                  !! in [kg m-3 ppt-1] or [R ppt-1 ~> kg m-3 ppt-1]
00807   type(eos_type), pointer     :: eos     !< Equation of state structure
00808   real, optional, intent(in)  :: scale   !< A multiplicative factor by which to scale density
00809                                  !! in combination with scaling given by US [various]
00810   ! Local variables
00811   real :: rho_scale ! A factor to convert density from kg m-3 to the desired units [R m3 kg-1 ~> 1]
00812   real :: p_scale   ! A factor to convert pressure to units of Pa [Pa T2 R-1 L-2 ~> 1]
00813   integer :: j
00814
00815   if (.not.associated(eos)) call mom_error(fatal, &
00816     "calculate_density_derivs called with an unassociated EOS_type EOS.")
00817
00818   p_scale = eos%RL2_T2_to_Pa
00819
00820   select case (eos%form_of_EOS)
00821     case (eos_linear)
00822       call calculate_density_derivs_linear(t, s, p_scale*pressure, drho_dt, drho_ds, &
00823                                            eos%Rho_T0_S0, eos%dRho_dT, eos%dRho_dS)
00824     case (eos_wright)
00825       call calculate_density_derivs_wright(t, s, p_scale*pressure, drho_dt, drho_ds)
00826     case (eos_teos10)
00827       call calculate_density_derivs_teos10(t, s, p_scale*pressure, drho_dt, drho_ds)
00828     case default
00829       call mom_error(fatal, "calculate_density_derivs_scalar: EOS%form_of_EOS is not valid.")
00830   end select
00831
00832   rho_scale = eos%kg_m3_to_R
00833   if (present(scale)) rho_scale = rho_scale * scale
00834   if (rho_scale /= 1.0) then
00835     drho_dt = rho_scale * drho_dt
00836     drho_ds = rho_scale * drho_ds
00837   endif
00838
```

The documentation for this interface was generated from the following file:

- /home/cermak/src/MOM6/src/equation_of_state/MOM_EOS.F90

43.5 mom_eos::calculate_density_second_derivs Interface Reference

Calculates the second derivatives of density with various combinations of temperature, salinity, and pressure from T, S and P.

Private Member Functions

- subroutine calculate_density_second_derivs_scalar (T, S, pressure, drho_dS_dS, drho_dS_dT, drho_dT_dT, drho_dS_dP, drho_dT_dP, EOS, scale)

  Calls the appropriate subroutine to calculate density second derivatives for scalar nputs.
- subroutine calculate_density_second_derivs_array (T, S, pressure, drho_dS_dS, drho_dS_dT, drho_dT_dT, drho_dS_dP, drho_dT_dP, start, npts, EOS, scale)

  Calls the appropriate subroutine to calculate density second derivatives for 1-D array inputs.

43.5.1 Detailed Description

Calculates the second derivatives of density with various combinations of temperature, salinity, and pressure from T, S and P.

Definition at line 93 of file MOM_EOS.F90.

43.5.2 Member Function/Subroutine Documentation

43.5.2.1 calculate_density_second_derivs_array()

```
subroutine mom_eos::calculate_density_second_derivs::calculate_density_second_derivs_array (
             real, dimension(:), intent(in) T,
             real, dimension(:), intent(in) S,
             real, dimension(:), intent(in) pressure,
             real, dimension(:), intent(inout) drho_dS_dS,
             real, dimension(:), intent(inout) drho_dS_dT,
             real, dimension(:), intent(inout) drho_dT_dT,
             real, dimension(:), intent(inout) drho_dS_dP,
             real, dimension(:), intent(inout) drho_dT_dP,
             integer, intent(in) start,
             integer, intent(in) npts,
             type(eos_type), pointer EOS,
             real, intent(in), optional scale )  [private]
```

Calls the appropriate subroutine to calculate density second derivatives for 1-D array inputs.

Parameters

in	*t*	Potential temperature referenced to the surface [degC]
in	*s*	Salinity [ppt]
in	*pressure*	Pressure [Pa] or [R L2 T-2 ~> Pa]
in,out	*drho_ds_ds*	Partial derivative of beta with respect to S [kg m-3 ppt-2] or [R ppt-2 ~> kg m-3 ppt-2]
in,out	*drho_ds_dt*	Partial derivative of beta with respect to T [kg m-3 ppt-1 degC-1] or [R ppt-1 degC-1 ~> kg m-3 ppt-1 degC-1]
in,out	*drho_dt_dt*	Partial derivative of alpha with respect to T [kg m-3 degC-2] or [R degC-2 ~> kg m-3 degC-2]
in,out	*drho_ds_dp*	Partial derivative of beta with respect to pressure [kg m-3 ppt-1 Pa-1] or [R ppt-1 Pa-1 ~> kg m-3 ppt-1 Pa-1]

Parameters

in,out	*drho_dt_dp*	Partial derivative of alpha with respect to pressure [kg m-3 degC-1 Pa-1] or [R degC-1 Pa-1 ~> kg m-3 degC-1 Pa-1]
in	*start*	Starting index within the array
in	*npts*	The number of values to calculate
	eos	Equation of state structure
in	*scale*	A multiplicative factor by which to scale density in combination with scaling given by US [various]

Definition at line 844 of file MOM_EOS.F90.

```
00844   real, dimension(:), intent(in)  :: t !< Potential temperature referenced to the surface [degC]
00845   real, dimension(:), intent(in)  :: s !< Salinity [ppt]
00846   real, dimension(:), intent(in)  :: pressure   !< Pressure [Pa] or [R L2 T-2 ~> Pa]
00847   real, dimension(:), intent(inout) :: drho_ds_ds !< Partial derivative of beta with respect to S
00848                                                   !!  [kg m-3 ppt-2] or [R ppt-2 ~> kg m-3 ppt-2]
00849   real, dimension(:), intent(inout) :: drho_ds_dt !< Partial derivative of beta with respect to T
00850                                                   !! [kg m-3 ppt-1 degC-1] or [R ppt-1 degC-1 ~> kg m-3
      ppt-1 degC-1]
00851   real, dimension(:), intent(inout) :: drho_dt_dt !< Partial derivative of alpha with respect to T
00852                                                   !! [kg m-3 degC-2] or [R degC-2 ~> kg m-3 degC-2]
00853   real, dimension(:), intent(inout) :: drho_ds_dp !< Partial derivative of beta with respect to pressure
00854                                                   !! [kg m-3 ppt-1 Pa-1] or [R ppt-1 Pa-1 ~> kg m-3 ppt-1
      Pa-1]
00855   real, dimension(:), intent(inout) :: drho_dt_dp !< Partial derivative of alpha with respect to pressure
00856                                                   !! [kg m-3 degC-1 Pa-1] or [R degC-1 Pa-1 ~> kg m-3
      degC-1 Pa-1]
00857   integer,            intent(in)  :: start !< Starting index within the array
00858   integer,            intent(in)  :: npts  !< The number of values to calculate
00859   type(eos_type),     pointer     :: eos   !< Equation of state structure
00860   real,                  optional, intent(in) :: scale !< A multiplicative factor by which to scale density
00861                                                   !! in combination with scaling given by US [various]
00862   ! Local variables
00863   real, dimension(size(pressure)) :: pres  ! Pressure converted to [Pa]
00864   real :: rho_scale ! A factor to convert density from kg m-3 to the desired units [R m3 kg-1 ~> 1]
00865   real :: p_scale   ! A factor to convert pressure to units of Pa [Pa T2 R-1 L-2 ~> 1]
00866   real :: i_p_scale ! The inverse of the factor to convert pressure to units of Pa [R L2 T-2 Pa-1 ~> 1]
00867   integer :: j
00868
00869   if (.not.associated(eos)) call mom_error(fatal, &
00870     "calculate_density_derivs called with an unassociated EOS_type EOS.")
00871
00872   p_scale = eos%RL2_T2_to_Pa
00873
00874   if (p_scale == 1.0) then
00875     select case (eos%form_of_EOS)
00876       case (eos_linear)
00877         call calculate_density_second_derivs_linear(t, s, pressure, drho_ds_ds, drho_ds_dt, &
00878                                                     drho_dt_dt, drho_ds_dp, drho_dt_dp, start, npts)
00879       case (eos_wright)
00880         call calculate_density_second_derivs_wright(t, s, pressure, drho_ds_ds, drho_ds_dt, &
00881                                                     drho_dt_dt, drho_ds_dp, drho_dt_dp, start, npts)
00882       case (eos_teos10)
00883         call calculate_density_second_derivs_teos10(t, s, pressure, drho_ds_ds, drho_ds_dt, &
00884                                                     drho_dt_dt, drho_ds_dp, drho_dt_dp, start, npts)
00885       case default
00886         call mom_error(fatal, "calculate_density_derivs: EOS%form_of_EOS is not valid.")
00887     end select
00888   else
00889     do j=start,start+npts-1 ; pres(j) = p_scale * pressure(j) ; enddo
00890     select case (eos%form_of_EOS)
00891       case (eos_linear)
00892         call calculate_density_second_derivs_linear(t, s, pres, drho_ds_ds, drho_ds_dt, &
00893                                                     drho_dt_dt, drho_ds_dp, drho_dt_dp, start, npts)
00894       case (eos_wright)
00895         call calculate_density_second_derivs_wright(t, s, pres, drho_ds_ds, drho_ds_dt, &
00896                                                     drho_dt_dt, drho_ds_dp, drho_dt_dp, start, npts)
00897       case (eos_teos10)
00898         call calculate_density_second_derivs_teos10(t, s, pres, drho_ds_ds, drho_ds_dt, &
00899                                                     drho_dt_dt, drho_ds_dp, drho_dt_dp, start, npts)
00900       case default
00901         call mom_error(fatal, "calculate_density_derivs: EOS%form_of_EOS is not valid.")
00902     end select
00903   endif
00904
00905   rho_scale = eos%kg_m3_to_R
```

```
00906   if (present(scale)) rho_scale = rho_scale * scale
00907   if (rho_scale /= 1.0) then ; do j=start,start+npts-1
00908     drho_ds_ds(j) = rho_scale * drho_ds_ds(j)
00909     drho_ds_dt(j) = rho_scale * drho_ds_dt(j)
00910     drho_dt_dt(j) = rho_scale * drho_dt_dt(j)
00911     drho_ds_dp(j) = rho_scale * drho_ds_dp(j)
00912     drho_dt_dp(j) = rho_scale * drho_dt_dp(j)
00913   enddo ; endif
00914
00915   if (p_scale /= 1.0) then
00916     i_p_scale = 1.0 / p_scale
00917     do j=start,start+npts-1
00918       drho_ds_dp(j) = i_p_scale * drho_ds_dp(j)
00919       drho_dt_dp(j) = i_p_scale * drho_dt_dp(j)
00920     enddo
00921   endif
00922
```

43.5.2.2 calculate_density_second_derivs_scalar()

```
subroutine mom_eos::calculate_density_second_derivs::calculate_density_second_derivs_scalar (
             real, intent(in) T,
             real, intent(in) S,
             real, intent(in) pressure,
             real, intent(out) drho_dS_dS,
             real, intent(out) drho_dS_dT,
             real, intent(out) drho_dT_dT,
             real, intent(out) drho_dS_dP,
             real, intent(out) drho_dT_dP,
             type(eos_type), pointer EOS,
             real, intent(in), optional scale )  [private]
```

Calls the appropriate subroutine to calculate density second derivatives for scalar nputs.

Parameters

in	*t*	Potential temperature referenced to the surface [degC]
in	*s*	Salinity [ppt]
in	*pressure*	Pressure [Pa] or [R L2 T-2 ~> Pa]
out	*drho_ds_ds*	Partial derivative of beta with respect to S [kg m-3 ppt-2] or [R ppt-2 ~> kg m-3 ppt-2]
out	*drho_ds_dt*	Partial derivative of beta with respect to T [kg m-3 ppt-1 degC-1] or [R ppt-1 degC-1 ~> kg m-3 ppt-1 degC-1]
out	*drho_dt_dt*	Partial derivative of alpha with respect to T [kg m-3 degC-2] or [R degC-2 ~> kg m-3 degC-2]
out	*drho_ds_dp*	Partial derivative of beta with respect to pressure [kg m-3 ppt-1 Pa-1] or [R ppt-1 Pa-1 ~> kg m-3 ppt-1 Pa-1]
out	*drho_dt_dp*	Partial derivative of alpha with respect to pressure [kg m-3 degC-1 Pa-1] or [R degC-1 Pa-1 ~> kg m-3 degC-1 Pa-1]
	eos	Equation of state structure
in	*scale*	A multiplicative factor by which to scale density in combination with scaling given by US [various]

Definition at line 928 of file MOM_EOS.F90.

```
00928   real, intent(in)  :: t !< Potential temperature referenced to the surface [degC]
```

```
00929   real, intent(in)  :: s !< Salinity [ppt]
00930   real, intent(in)  :: pressure   !< Pressure [Pa] or [R L2 T-2 ~> Pa]
00931   real, intent(out) :: drho_ds_ds !< Partial derivative of beta with respect to S
00932                                   !! [kg m-3 ppt-2] or [R ppt-2 ~> kg m-3 ppt-2]
00933   real, intent(out) :: drho_ds_dt !< Partial derivative of beta with respect to T
00934                                   !! [kg m-3 ppt-1 degC-1] or [R ppt-1 degC-1 ~> kg m-3 ppt-1 degC-1]
00935   real, intent(out) :: drho_dt_dt !< Partial derivative of alpha with respect to T
00936                                   !! [kg m-3 degC-2] or [R degC-2 ~> kg m-3 degC-2]
00937   real, intent(out) :: drho_ds_dp !< Partial derivative of beta with respect to pressure
00938                                   !! [kg m-3 ppt-1 Pa-1] or [R ppt-1 Pa-1 ~> kg m-3 ppt-1 Pa-1]
00939   real, intent(out) :: drho_dt_dp !< Partial derivative of alpha with respect to pressure
00940                                   !! [kg m-3 degC-1 Pa-1] or [R degC-1 Pa-1 ~> kg m-3 degC-1 Pa-1]
00941   type(eos_type), pointer    :: eos !< Equation of state structure
00942   real, optional, intent(in) :: scale !< A multiplicative factor by which to scale density
00943                                   !! in combination with scaling given by US [various]
00944   ! Local variables
00945   real :: rho_scale ! A factor to convert density from kg m-3 to the desired units [R m3 kg-1 ~> 1]
00946   real :: p_scale   ! A factor to convert pressure to units of Pa [Pa T2 R-1 L-2 ~> 1]
00947   real :: i_p_scale ! The inverse of the factor to convert pressure to units of Pa [R L2 T-2 Pa-1 ~> 1]
00948
00949   if (.not.associated(eos)) call mom_error(fatal, &
00950     "calculate_density_derivs called with an unassociated EOS_type EOS.")
00951
00952   p_scale = eos%RL2_T2_to_Pa
00953
00954   select case (eos%form_of_EOS)
00955     case (eos_linear)
00956       call calculate_density_second_derivs_linear(t, s, p_scale*pressure, drho_ds_ds, drho_ds_dt, &
00957                                                   drho_dt_dt, drho_ds_dp, drho_dt_dp)
00958     case (eos_wright)
00959       call calculate_density_second_derivs_wright(t, s, p_scale*pressure, drho_ds_ds, drho_ds_dt, &
00960                                                   drho_dt_dt, drho_ds_dp, drho_dt_dp)
00961     case (eos_teos10)
00962       call calculate_density_second_derivs_teos10(t, s, p_scale*pressure, drho_ds_ds, drho_ds_dt, &
00963                                                   drho_dt_dt, drho_ds_dp, drho_dt_dp)
00964     case default
00965       call mom_error(fatal, "calculate_density_derivs: EOS%form_of_EOS is not valid.")
00966   end select
00967
00968   rho_scale = eos%kg_m3_to_R
00969   if (present(scale)) rho_scale = rho_scale * scale
00970   if (rho_scale /= 1.0) then
00971     drho_ds_ds = rho_scale * drho_ds_ds
00972     drho_ds_dt = rho_scale * drho_ds_dt
00973     drho_dt_dt = rho_scale * drho_dt_dt
00974     drho_ds_dp = rho_scale * drho_ds_dp
00975     drho_dt_dp = rho_scale * drho_dt_dp
00976   endif
00977
00978   if (p_scale /= 1.0) then
00979     i_p_scale = 1.0 / p_scale
00980     drho_ds_dp = i_p_scale * drho_ds_dp
00981     drho_dt_dp = i_p_scale * drho_dt_dp
00982   endif
00983
```

The documentation for this interface was generated from the following file:

- /home/cermak/src/MOM6/src/equation_of_state/MOM_EOS.F90

43.6 mom_eos::calculate_spec_vol Interface Reference

Calculates specific volume of sea water from T, S and P.

Private Member Functions

- subroutine calc_spec_vol_scalar (T, S, pressure, specvol, EOS, spv_ref, scale)

  Calls the appropriate subroutine to calculate specific volume of sea water for scalar inputs.
- subroutine calculate_spec_vol_array (T, S, pressure, specvol, start, npts, EOS, spv_ref, scale)

  Calls the appropriate subroutine to calculate the specific volume of sea water for 1-D array inputs.
- subroutine calc_spec_vol_1d (T, S, pressure, specvol, EOS, dom, spv_ref, scale)

  Calls the appropriate subroutine to calculate the specific volume of sea water for 1-D array inputs, potentially limiting the domain of indices that are worked on.

43.6.1 Detailed Description

Calculates specific volume of sea water from T, S and P.

Definition at line 75 of file MOM_EOS.F90.

43.6.2 Member Function/Subroutine Documentation

43.6.2.1 calc_spec_vol_1d()

```
subroutine mom_eos::calculate_spec_vol::calc_spec_vol_1d (
             real, dimension(:), intent(in) T,
             real, dimension(:), intent(in) S,
             real, dimension(:), intent(in) pressure,
             real, dimension(:), intent(inout) specvol,
             type(eos_type), pointer EOS,
             integer, dimension(2), intent(in), optional dom,
             real, intent(in), optional spv_ref,
             real, intent(in), optional scale )  [private]
```

Calls the appropriate subroutine to calculate the specific volume of sea water for 1-D array inputs, potentially limiting the domain of indices that are worked on.

Parameters

in	*t*	Potential temperature referenced to the surface [degC]
in	*s*	Salinity [ppt]
in	*pressure*	Pressure [R L2 T-2 ~> Pa]
in,out	*specvol*	In situ specific volume [R-1 ~> m3 kg-1]
	eos	Equation of state structure
in	*dom*	The domain of indices to work on, taking into account that arrays start at 1.
in	*spv_ref*	A reference specific volume [R-1 ~> m3 kg-1]
in	*scale*	A multiplicative factor by which to scale output specific volume in combination with scaling given by US [various]

Definition at line 571 of file MOM_EOS.F90.

```
00571   real, dimension(:),    intent(in)    :: t        !< Potential temperature referenced to the surface
       [degC]
00572   real, dimension(:),    intent(in)    :: s        !< Salinity [ppt]
00573   real, dimension(:),    intent(in)    :: pressure !< Pressure [R L2 T-2 ~> Pa]
00574   real, dimension(:),    intent(inout) :: specvol  !< In situ specific volume [R-1 ~> m3 kg-1]
00575   type(eos_type),        pointer       :: eos      !< Equation of state structure
00576   integer, dimension(2), optional, intent(in) :: dom   !< The domain of indices to work on, taking
00577                                                        !! into account that arrays start at 1.
00578   real,                  optional, intent(in) :: spv_ref !< A reference specific volume [R-1 ~> m3 kg-1]
00579   real,                  optional, intent(in) :: scale !< A multiplicative factor by which to scale
00580                                                        !! output specific volume in combination with
00581                                                        !! scaling given by US [various]
00582   ! Local variables
00583   real, dimension(size(specvol)) :: pres  ! Pressure converted to [Pa]
00584   real :: p_scale   ! A factor to convert pressure to units of Pa [Pa T2 R-1 L-2 ~> 1]
00585   real :: spv_unscale ! A factor to convert specific volume from R-1 to m3 kg-1 [m3 kg-1 R ~> 1]
00586   real :: spv_scale ! A factor to convert specific volume from m3 kg-1 to the desired units [kg m-3 R-1 ~>
```

```
      1]
00587   real :: spv_reference ! spv_ref converted to [m3 kg-1]
00588   integer :: i, is, ie, npts
00589
00590   if (.not.associated(eos)) call mom_error(fatal, &
00591     "calc_spec_vol_1d called with an unassociated EOS_type EOS.")
00592
00593   if (present(dom)) then
00594     is = dom(1) ; ie = dom(2) ; npts = 1 + ie - is
00595   else
00596     is = 1 ; ie = size(specvol) ; npts = 1 + ie - is
00597   endif
00598
00599   p_scale = eos%RL2_T2_to_Pa
00600   spv_unscale = eos%kg_m3_to_R
00601
00602   if ((p_scale == 1.0) .and. (spv_unscale == 1.0)) then
00603     call calculate_spec_vol_array(t, s, pressure, specvol, is, npts, eos, spv_ref)
00604   elseif (present(spv_ref)) then ! This is the same as above, but with some extra work to rescale
      variables.
00605     do i=is,ie ; pres(i) = p_scale * pressure(i) ; enddo
00606     spv_reference = spv_unscale*spv_ref
00607     call calculate_spec_vol_array(t, s, pres, specvol, is, npts, eos, spv_reference)
00608   else  ! There is rescaling of variables, but spv_ref is not present. Passing a 0 value of spv_ref
00609         ! changes answers at roundoff for some equations of state, like Wright and UNESCO.
00610     do i=is,ie ; pres(i) = p_scale * pressure(i) ; enddo
00611     call calculate_spec_vol_array(t, s, pres, specvol, is, npts, eos)
00612   endif
00613
00614   spv_scale = eos%R_to_kg_m3
00615   if (present(scale)) spv_scale = spv_scale * scale
00616   if (spv_scale /= 1.0) then ; do i=is,ie
00617     specvol(i) = spv_scale * specvol(i)
00618   enddo ; endif
00619
```

43.6.2.2 calc_spec_vol_scalar()

```
subroutine mom_eos::calculate_spec_vol::calc_spec_vol_scalar (
             real, intent(in) T,
             real, intent(in) S,
             real, intent(in) pressure,
             real, intent(out) specvol,
             type(eos_type), pointer EOS,
             real, intent(in), optional spv_ref,
             real, intent(in), optional scale )  [private]
```

Calls the appropriate subroutine to calculate specific volume of sea water for scalar inputs.

Parameters

in	*t*	Potential temperature referenced to the surface [degC]
in	*s*	Salinity [ppt]
in	*pressure*	Pressure [Pa] or [R L2 T-2 ~> Pa]
out	*specvol*	In situ? specific volume [m3 kg-1] or [R-1 ~> m3 kg-1]
	eos	Equation of state structure
in	*spv_ref*	A reference specific volume [m3 kg-1] or [R-1 m3 kg-1]
in	*scale*	A multiplicative factor by which to scale specific volume in combination with scaling given by US [various]

Definition at line 533 of file MOM_EOS.F90.

```
00533   real,           intent(in)  :: t        !< Potential temperature referenced to the surface [degC]
```

```
00534   real,           intent(in)  :: s        !< Salinity [ppt]
00535   real,           intent(in)  :: pressure !< Pressure [Pa] or [R L2 T-2 ~> Pa]
00536   real,           intent(out) :: specvol  !< In situ? specific volume [m3 kg-1] or [R-1 ~> m3 kg-1]
00537   type(eos_type), pointer     :: eos      !< Equation of state structure
00538   real, optional, intent(in)  :: spv_ref  !< A reference specific volume [m3 kg-1] or [R-1 m3 kg-1]
00539   real, optional, intent(in)  :: scale    !< A multiplicative factor by which to scale specific
00540                                           !! volume in combination with scaling given by US [various]
00541
00542   real, dimension(1) :: ta, sa, pres, spv  ! Rescaled single element array versions of the arguments.
00543   real :: spv_reference ! spv_ref converted to [m3 kg-1]
00544   real :: spv_scale ! A factor to convert specific volume from m3 kg-1 to the desired units [kg R-1 m-3 ~>
      1]
00545
00546   if (.not.associated(eos)) call mom_error(fatal, &
00547     "calc_spec_vol_scalar called with an unassociated EOS_type EOS.")
00548
00549   pres(1) = eos%RL2_T2_to_Pa*pressure
00550   ta(1) = t ; sa(1) = s
00551
00552   if (present(spv_ref)) then
00553     spv_reference = eos%kg_m3_to_R*spv_ref
00554     call calculate_spec_vol_array(ta, sa, pres, spv, 1, 1, eos, spv_reference)
00555   else
00556     call calculate_spec_vol_array(ta, sa, pres, spv, 1, 1, eos)
00557   endif
00558   specvol = spv(1)
00559
00560   spv_scale = eos%R_to_kg_m3
00561   if (present(scale)) spv_scale = spv_scale * scale
00562   if (spv_scale /= 1.0) then
00563     specvol = spv_scale * specvol
00564   endif
00565
```

43.6.2.3 calculate_spec_vol_array()

```
subroutine mom_eos::calculate_spec_vol::calculate_spec_vol_array (
             real, dimension(:), intent(in) T,
             real, dimension(:), intent(in) S,
             real, dimension(:), intent(in) pressure,
             real, dimension(:), intent(inout) specvol,
             integer, intent(in) start,
             integer, intent(in) npts,
             type(eos_type), pointer EOS,
             real, intent(in), optional spv_ref,
             real, intent(in), optional scale )  [private]
```

Calls the appropriate subroutine to calculate the specific volume of sea water for 1-D array inputs.

Parameters

in	*t*	potential temperature relative to the surface [degC]
in	*s*	salinity [ppt]
in	*pressure*	pressure [Pa]
in,out	*specvol*	in situ specific volume [kg m-3]
in	*start*	the starting point in the arrays.
in	*npts*	the number of values to calculate.
	eos	Equation of state structure
in	*spv_ref*	A reference specific volume [m3 kg-1]
in	*scale*	A multiplicative factor by which to scale specific volume in combination with scaling given by US [various]

Definition at line 486 of file MOM_EOS.F90.

```
00486   real, dimension(:), intent(in)    :: t        !< potential temperature relative to the surface [degC]
00487   real, dimension(:), intent(in)    :: s        !< salinity [ppt]
00488   real, dimension(:), intent(in)    :: pressure !< pressure [Pa]
00489   real, dimension(:), intent(inout) :: specvol  !< in situ specific volume [kg m-3]
00490   integer,            intent(in)    :: start    !< the starting point in the arrays.
00491   integer,            intent(in)    :: npts     !< the number of values to calculate.
00492   type(eos_type),     pointer       :: eos      !< Equation of state structure
00493   real,     optional, intent(in)    :: spv_ref  !< A reference specific volume [m3 kg-1]
00494   real,     optional, intent(in)    :: scale    !< A multiplicative factor by which to scale specific
00495                                                 !! volume in combination with scaling given by US [various]
00496
00497   real, dimension(size(specvol))  :: rho   ! Density [kg m-3]
00498   integer :: j
00499
00500   if (.not.associated(eos)) call mom_error(fatal, &
00501     "calculate_spec_vol_array called with an unassociated EOS_type EOS.")
00502
00503   select case (eos%form_of_EOS)
00504     case (eos_linear)
00505       call calculate_spec_vol_linear(t, s, pressure, specvol, start, npts, &
00506                 eos%rho_T0_S0, eos%drho_dT, eos%drho_dS, spv_ref)
00507     case (eos_unesco)
00508       call calculate_spec_vol_unesco(t, s, pressure, specvol, start, npts, spv_ref)
00509     case (eos_wright)
00510       call calculate_spec_vol_wright(t, s, pressure, specvol, start, npts, spv_ref)
00511     case (eos_teos10)
00512       call calculate_spec_vol_teos10(t, s, pressure, specvol, start, npts, spv_ref)
00513     case (eos_nemo)
00514       call calculate_density_nemo(t, s, pressure, rho, start, npts)
00515       if (present(spv_ref)) then
00516         specvol(:) = 1.0 / rho(:) - spv_ref
00517       else
00518         specvol(:) = 1.0 / rho(:)
00519       endif
00520     case default
00521       call mom_error(fatal, "calculate_spec_vol_array: EOS%form_of_EOS is not valid.")
00522   end select
00523
00524   if (present(scale)) then ; if (scale /= 1.0) then ; do j=start,start+npts-1
00525     specvol(j) = scale * specvol(j)
00526   enddo ; endif ; endif
00527
```

The documentation for this interface was generated from the following file:

- /home/cermak/src/MOM6/src/equation_of_state/MOM_EOS.F90

43.7 mom_eos::calculate_specific_vol_derivs Interface Reference

Calculate the derivatives of specific volume with temperature and salinity from T, S, and P.

Private Member Functions

- subroutine calculate_spec_vol_derivs_array (T, S, pressure, dSV_dT, dSV_dS, start, npts, EOS)

  Calls the appropriate subroutine to calculate specific volume derivatives for an array.
- subroutine calc_spec_vol_derivs_1d (T, S, pressure, dSV_dT, dSV_dS, EOS, dom, scale)

  Calls the appropriate subroutine to calculate specific volume derivatives for 1-d array inputs, potentially limiting the domain of indices that are worked on.

43.7.1 Detailed Description

Calculate the derivatives of specific volume with temperature and salinity from T, S, and P.

Definition at line 87 of file MOM_EOS.F90.

43.7.2 Member Function/Subroutine Documentation

43.7.2.1 calc_spec_vol_derivs_1d()

```
subroutine mom_eos::calculate_specific_vol_derivs::calc_spec_vol_derivs_1d (
            real, dimension(:), intent(in) T,
            real, dimension(:), intent(in) S,
            real, dimension(:), intent(in) pressure,
            real, dimension(:), intent(inout) dSV_dT,
            real, dimension(:), intent(inout) dSV_dS,
            type(eos_type), pointer EOS,
            integer, dimension(2), intent(in), optional dom,
            real, intent(in), optional scale )  [private]
```

Calls the appropriate subroutine to calculate specific volume derivatives for 1-d array inputs, potentially limiting the domain of indices that are worked on.

Parameters

in	*t*	Potential temperature referenced to the surface [degC]
in	*s*	Salinity [ppt]
in	*pressure*	Pressure [R L2 T-2 ~> Pa]
in,out	*dsv_dt*	The partial derivative of specific volume with potential temperature [R-1 degC-1 ~> m3 kg-1 degC-1]
in,out	*dsv_ds*	The partial derivative of specific volume with salinity [R-1 ppt-1 ~> m3 kg-1 ppt-1]
	eos	Equation of state structure
in	*dom*	The domain of indices to work on, taking into account that arrays start at 1.
in	*scale*	A multiplicative factor by which to scale specific volume in combination with scaling given by US [various]

Definition at line 1040 of file MOM_EOS.F90.

```
01040   real, dimension(:), intent(in)    :: t        !< Potential temperature referenced to the surface [degC]
01041   real, dimension(:), intent(in)    :: s        !< Salinity [ppt]
01042   real, dimension(:), intent(in)    :: pressure !< Pressure [R L2 T-2 ~> Pa]
01043   real, dimension(:), intent(inout) :: dsv_dt   !< The partial derivative of specific volume with potential
01044                                                 !! temperature [R-1 degC-1 ~> m3 kg-1 degC-1]
01045   real, dimension(:), intent(inout) :: dsv_ds   !< The partial derivative of specific volume with salinity
01046                                                 !! [R-1 ppt-1 ~> m3 kg-1 ppt-1]
01047   type(eos_type),     pointer       :: eos      !< Equation of state structure
01048   integer, dimension(2), optional, intent(in) :: dom   !< The domain of indices to work on, taking
01049                                                        !! into account that arrays start at 1.
01050   real,                  optional, intent(in) :: scale !< A multiplicative factor by which to scale
       specific
01051                                                 !! volume in combination with scaling given by US [various]
01052
01053   ! Local variables
01054   real, dimension(size(dSV_dT)) :: press   ! Pressure converted to [Pa]
01055   real :: spv_scale ! A factor to convert specific volume from m3 kg-1 to the desired units [kg R-1 m-3 ~>
       1]
01056   real :: p_scale   ! A factor to convert pressure to units of Pa [Pa T2 R-1 L-2 ~> 1]
01057   integer :: i, is, ie, npts
01058
01059   if (.not.associated(eos)) call mom_error(fatal, &
01060     "calculate_spec_vol_derivs_1d called with an unassociated EOS_type EOS.")
01061
01062   if (present(dom)) then
01063     is = dom(1) ; ie = dom(2) ; npts = 1 + ie - is
01064   else
```

```
01065     is = 1 ; ie = size(dsv_dt) ; npts = 1 + ie - is
01066   endif
01067   p_scale = eos%RL2_T2_to_Pa
01068
01069   if (p_scale == 1.0) then
01070     call calculate_spec_vol_derivs_array(t, s, pressure, dsv_dt, dsv_ds, is, npts, eos)
01071   else
01072     do i=is,ie ; press(i) = p_scale * pressure(i) ; enddo
01073     call calculate_spec_vol_derivs_array(t, s, press, dsv_dt, dsv_ds, is, npts, eos)
01074   endif
01075
01076   spv_scale = eos%R_to_kg_m3
01077   if (present(scale)) spv_scale = spv_scale * scale
01078   if (spv_scale /= 1.0) then ; do i=is,ie
01079     dsv_dt(i) = spv_scale * dsv_dt(i)
01080     dsv_ds(i) = spv_scale * dsv_ds(i)
01081   enddo ; endif
01082
```

43.7.2.2 calculate_spec_vol_derivs_array()

```
subroutine mom_eos::calculate_specific_vol_derivs::calculate_spec_vol_derivs_array (
             real, dimension(:), intent(in) T,
             real, dimension(:), intent(in) S,
             real, dimension(:), intent(in) pressure,
             real, dimension(:), intent(inout) dSV_dT,
             real, dimension(:), intent(inout) dSV_dS,
             integer, intent(in) start,
             integer, intent(in) npts,
             type(eos_type), pointer EOS )  [private]
```

Calls the appropriate subroutine to calculate specific volume derivatives for an array.

Parameters

in	*t*	Potential temperature referenced to the surface [degC]
in	*s*	Salinity [ppt]
in	*pressure*	Pressure [Pa]
in,out	*dsv_dt*	The partial derivative of specific volume with potential temperature [m3 kg-1 degC-1]
in,out	*dsv_ds*	The partial derivative of specific volume with salinity [m3 kg-1 ppt-1]
in	*start*	Starting index within the array
in	*npts*	The number of values to calculate
	eos	Equation of state structure

Definition at line 988 of file MOM_EOS.F90.

```
00988   real, dimension(:), intent(in)  :: t !< Potential temperature referenced to the surface [degC]
00989   real, dimension(:), intent(in)  :: s !< Salinity [ppt]
00990   real, dimension(:), intent(in)  :: pressure !< Pressure [Pa]
00991   real, dimension(:), intent(inout) :: dsv_dt !< The partial derivative of specific volume with potential
00992                                                !! temperature [m3 kg-1 degC-1]
00993   real, dimension(:), intent(inout) :: dsv_ds !< The partial derivative of specific volume with salinity
00994                                                !! [m3 kg-1 ppt-1]
00995   integer,            intent(in)  :: start  !< Starting index within the array
00996   integer,            intent(in)  :: npts   !< The number of values to calculate
00997   type(eos_type),     pointer     :: eos    !< Equation of state structure
00998
00999   ! Local variables
01000   real, dimension(size(T)) :: press   ! Pressure converted to [Pa]
01001   real, dimension(size(T)) :: rho     ! In situ density [kg m-3]
```

```
01002   real, dimension(size(T)) :: drho_dt ! Derivative of density with temperature [kg m-3 degC-1]
01003   real, dimension(size(T)) :: drho_ds ! Derivative of density with salinity [kg m-3 ppt-1]
01004   integer :: j
01005
01006   if (.not.associated(eos)) call mom_error(fatal, &
01007     "calculate_spec_vol_derivs_array called with an unassociated EOS_type EOS.")
01008
01009   select case (eos%form_of_EOS)
01010     case (eos_linear)
01011       call calculate_specvol_derivs_linear(t, s, pressure, dsv_dt, dsv_ds, start, &
01012                                            npts, eos%Rho_T0_S0, eos%dRho_dT, eos%dRho_dS)
01013     case (eos_unesco)
01014       call calculate_density_unesco(t, s, pressure, rho, start, npts)
01015       call calculate_density_derivs_unesco(t, s, pressure, drho_dt, drho_ds, start, npts)
01016       do j=start,start+npts-1
01017         dsv_dt(j) = -drho_dt(j)/(rho(j)**2)
01018         dsv_ds(j) = -drho_ds(j)/(rho(j)**2)
01019       enddo
01020     case (eos_wright)
01021       call calculate_specvol_derivs_wright(t, s, pressure, dsv_dt, dsv_ds, start, npts)
01022     case (eos_teos10)
01023       call calculate_specvol_derivs_teos10(t, s, pressure, dsv_dt, dsv_ds, start, npts)
01024     case (eos_nemo)
01025       call calculate_density_nemo(t, s, pressure, rho, start, npts)
01026       call calculate_density_derivs_nemo(t, s, pressure, drho_dt, drho_ds, start, npts)
01027       do j=start,start+npts-1
01028         dsv_dt(j) = -drho_dt(j)/(rho(j)**2)
01029         dsv_ds(j) = -drho_ds(j)/(rho(j)**2)
01030       enddo
01031     case default
01032       call mom_error(fatal, "calculate_spec_vol_derivs_array: EOS%form_of_EOS is not valid.")
01033   end select
01034
```

The documentation for this interface was generated from the following file:

- /home/cermak/src/MOM6/src/equation_of_state/MOM_EOS.F90

43.8 mom_eos::calculate_tfreeze Interface Reference

Calculates the freezing point of sea water from T, S and P.

Private Member Functions

- subroutine calculate_tfreeze_scalar (S, pressure, T_fr, EOS, pres_scale)

  Calls the appropriate subroutine to calculate the freezing point for scalar inputs.
- subroutine calculate_tfreeze_array (S, pressure, T_fr, start, npts, EOS, pres_scale)

  Calls the appropriate subroutine to calculate the freezing point for a 1-D array.

43.8.1 Detailed Description

Calculates the freezing point of sea water from T, S and P.

Definition at line 98 of file MOM_EOS.F90.

43.8.2 Member Function/Subroutine Documentation

43.8.2.1 calculate_tfreeze_array()

```
subroutine mom_eos::calculate_tfreeze::calculate_tfreeze_array (
             real, dimension(:), intent(in) S,
             real, dimension(:), intent(in) pressure,
             real, dimension(:), intent(inout) T_fr,
             integer, intent(in) start,
             integer, intent(in) npts,
             type(eos_type), pointer EOS,
             real, intent(in), optional pres_scale )  [private]
```

Calls the appropriate subroutine to calculate the freezing point for a 1-D array.

Parameters

in	*s*	Salinity [ppt]
in	*pressure*	Pressure [Pa] or [other]
in,out	*t_fr*	Freezing point potential temperature referenced to the surface [degC]
in	*start*	Starting index within the array
in	*npts*	The number of values to calculate
	eos	Equation of state structure
in	*pres_scale*	A multiplicative factor to convert pressure into Pa.

Definition at line 656 of file MOM_EOS.F90.

```
00656   real, dimension(:), intent(in)    :: s        !< Salinity [ppt]
00657   real, dimension(:), intent(in)    :: pressure !< Pressure [Pa] or [other]
00658   real, dimension(:), intent(inout) :: t_fr     !< Freezing point potential temperature referenced
00659                                                 !! to the surface [degC]
00660   integer,            intent(in)    :: start    !< Starting index within the array
00661   integer,            intent(in)    :: npts     !< The number of values to calculate
00662   type(eos_type),     pointer       :: eos      !< Equation of state structure
00663   real,     optional, intent(in)    :: pres_scale !< A multiplicative factor to convert pressure into Pa.
00664
00665   ! Local variables
00666   real, dimension(size(pressure)) :: pres  ! Pressure converted to [Pa]
00667   real :: p_scale ! A factor to convert pressure to units of Pa.
00668   integer :: j
00669
00670   if (.not.associated(eos)) call mom_error(fatal, &
00671     "calculate_TFreeze_scalar called with an unassociated EOS_type EOS.")
00672
00673   p_scale = 1.0 ; if (present(pres_scale)) p_scale = pres_scale
00674
00675   if (p_scale == 1.0) then
00676     select case (eos%form_of_TFreeze)
00677       case (tfreeze_linear)
00678         call calculate_tfreeze_linear(s, pressure, t_fr, start, npts, &
00679                                       eos%TFr_S0_P0, eos%dTFr_dS, eos%dTFr_dp)
00680       case (tfreeze_millero)
00681         call calculate_tfreeze_millero(s, pressure, t_fr, start, npts)
00682       case (tfreeze_teos10)
00683         call calculate_tfreeze_teos10(s, pressure, t_fr, start, npts)
00684       case default
00685         call mom_error(fatal, "calculate_TFreeze_scalar: form_of_TFreeze is not valid.")
00686     end select
00687   else
00688     do j=start,start+npts-1 ; pres(j) = p_scale * pressure(j) ; enddo
00689     select case (eos%form_of_TFreeze)
00690       case (tfreeze_linear)
00691         call calculate_tfreeze_linear(s, pres, t_fr, start, npts, &
00692                                       eos%TFr_S0_P0, eos%dTFr_dS, eos%dTFr_dp)
00693       case (tfreeze_millero)
00694         call calculate_tfreeze_millero(s, pres, t_fr, start, npts)
00695       case (tfreeze_teos10)
00696         call calculate_tfreeze_teos10(s, pres, t_fr, start, npts)
00697       case default
00698         call mom_error(fatal, "calculate_TFreeze_scalar: form_of_TFreeze is not valid.")
00699     end select
00700   endif
00701
```

43.8.2.2 calculate_tfreeze_scalar()

```
subroutine mom_eos::calculate_tfreeze::calculate_tfreeze_scalar (
             real, intent(in) S,
             real, intent(in) pressure,
             real, intent(out) T_fr,
             type(eos_type), pointer EOS,
             real, intent(in), optional pres_scale )  [private]
```

Calls the appropriate subroutine to calculate the freezing point for scalar inputs.

Parameters

in	*s*	Salinity [ppt]
in	*pressure*	Pressure [Pa] or [other]
out	*t_fr*	Freezing point potential temperature referenced to the surface [degC]
	eos	Equation of state structure
in	*pres_scale*	A multiplicative factor to convert pressure into Pa

Definition at line 625 of file MOM_EOS.F90.

```
00625   real,           intent(in)  :: s !< Salinity [ppt]
00626   real,           intent(in)  :: pressure !< Pressure [Pa] or [other]
00627   real,           intent(out) :: t_fr !< Freezing point potential temperature referenced
00628                                      !! to the surface [degC]
00629   type(eos_type), pointer     :: eos !< Equation of state structure
00630   real, optional, intent(in)  :: pres_scale !< A multiplicative factor to convert pressure into Pa
00631
00632   ! Local variables
00633   real :: p_scale ! A factor to convert pressure to units of Pa.
00634
00635   if (.not.associated(eos)) call mom_error(fatal, &
00636     "calculate_TFreeze_scalar called with an unassociated EOS_type EOS.")
00637
00638   p_scale = 1.0 ; if (present(pres_scale)) p_scale = pres_scale
00639
00640   select case (eos%form_of_TFreeze)
00641     case (tfreeze_linear)
00642       call calculate_tfreeze_linear(s, p_scale*pressure, t_fr, eos%TFr_S0_P0, &
00643                                     eos%dTFr_dS, eos%dTFr_dp)
00644     case (tfreeze_millero)
00645       call calculate_tfreeze_millero(s, p_scale*pressure, t_fr)
00646     case (tfreeze_teos10)
00647       call calculate_tfreeze_teos10(s, p_scale*pressure, t_fr)
00648     case default
00649       call mom_error(fatal, "calculate_TFreeze_scalar: form_of_TFreeze is not valid.")
00650   end select
00651
```

The documentation for this interface was generated from the following file:

- /home/cermak/src/MOM6/src/equation_of_state/MOM_EOS.F90

43.9 mom_eos::eos_type Type Reference

A control structure for the equation of state.

Public Attributes

- integer form_of_eos = 0
  The equation of state to use.
- integer form_of_tfreeze = 0
  The expression for the potential temperature of the freezing point.
- logical eos_quadrature
  If true, always use the generic (quadrature) code for the integrals of density.
- logical compressible = .true.
  If true, in situ density is a function of pressure.
- real rho_t0_s0
  The density at T=0, S=0 [kg m-3].
- real drho_dt
  The partial derivative of density with temperature [kg m-3 degC-1].
- real drho_ds
  The partial derivative of density with salinity [kg m-3 ppt-1].
- real tfr_s0_p0
  The freezing potential temperature at S=0, P=0 [degC].
- real dtfr_ds
  The derivative of freezing point with salinity [degC ppt-1].
- real dtfr_dp
  The derivative of freezing point with pressure [degC Pa-1].
- real m_to_z = 1.
  A constant that translates distances in meters to the units of depth.
- real kg_m3_to_r = 1.
  A constant that translates kilograms per meter cubed to the units of density.
- real r_to_kg_m3 = 1.
  A constant that translates the units of density to kilograms per meter cubed.
- real rl2_t2_to_pa = 1.
  Convert pressures from R L2 T-2 to Pa.
- real l_t_to_m_s = 1.
  Convert lateral velocities from L T-1 to m s-1.

43.9.1 Detailed Description

A control structure for the equation of state.

Definition at line 108 of file MOM_EOS.F90.

43.9.2 Member Data Documentation

43.9.2.1 compressible

```
logical mom_eos::eos_type::compressible = .true.
```

If true, in situ density is a function of pressure.

Definition at line 114 of file MOM_EOS.F90.

```
00114   logical :: compressible = .true. !< If true, in situ density is a function of pressure.
```

43.9.2.2 drho_ds

```
real mom_eos::eos_type::drho_ds
```

The partial derivative of density with salinity [kg m-3 ppt-1].

Definition at line 118 of file MOM_EOS.F90.

```
00118   real :: drho_ds   !< The partial derivative of density with salinity [kg m-3 ppt-1]
```

43.9.2.3 drho_dt

```
real mom_eos::eos_type::drho_dt
```

The partial derivative of density with temperature [kg m-3 degC-1].

Definition at line 117 of file MOM_EOS.F90.

```
00117   real :: drho_dt   !< The partial derivative of density with temperature [kg m-3 degC-1]
```

43.9.2.4 dtfr_dp

```
real mom_eos::eos_type::dtfr_dp
```

The derivative of freezing point with pressure [degC Pa-1].

Definition at line 123 of file MOM_EOS.F90.

```
00123   real :: dtfr_dp   !< The derivative of freezing point with pressure [degC Pa-1]
```

43.9.2.5 dtfr_ds

```
real mom_eos::eos_type::dtfr_ds
```

The derivative of freezing point with salinity [degC ppt-1].

Definition at line 122 of file MOM_EOS.F90.

```
00122   real :: dtfr_ds   !< The derivative of freezing point with salinity [degC ppt-1]
```

43.9.2.6 eos_quadrature

```
logical mom_eos::eos_type::eos_quadrature
```

If true, always use the generic (quadrature) code for the integrals of density.

Definition at line 112 of file MOM_EOS.F90.

```
00112   logical :: eos_quadrature  !< If true, always use the generic (quadrature)
```

43.9.2.7 form_of_eos

```
integer mom_eos::eos_type::form_of_eos = 0
```

The equation of state to use.

Definition at line 109 of file MOM_EOS.F90.

```
00109   integer :: form_of_eos = 0 !< The equation of state to use.
```

43.9.2.8 form_of_tfreeze

```
integer mom_eos::eos_type::form_of_tfreeze = 0
```

The expression for the potential temperature of the freezing point.

Definition at line 110 of file MOM_EOS.F90.

```
00110   integer :: form_of_tfreeze = 0 !< The expression for the potential temperature
```

43.9.2.9 kg_m3_to_r

```
real mom_eos::eos_type::kg_m3_to_r = 1.
```

A constant that translates kilograms per meter cubed to the units of density.

Definition at line 128 of file MOM_EOS.F90.

```
00128   real :: kg_m3_to_r = 1.  !< A constant that translates kilograms per meter cubed to the units of density.
```

43.9.2.10 l_t_to_m_s

```
real mom_eos::eos_type::l_t_to_m_s = 1.
```

Convert lateral velocities from L T-1 to m s-1.

Definition at line 131 of file MOM_EOS.F90.

```
00131   real :: l_t_to_m_s = 1.  !< Convert lateral velocities from L T-1 to m s-1.
```

43.9.2.11 m_to_z

```
real mom_eos::eos_type::m_to_z = 1.
```

A constant that translates distances in meters to the units of depth.

Definition at line 127 of file MOM_EOS.F90.

```
00127   real :: m_to_z = 1.      !< A constant that translates distances in meters to the units of depth.
```

43.9.2.12 r_to_kg_m3

```
real mom_eos::eos_type::r_to_kg_m3 = 1.
```

A constant that translates the units of density to kilograms per meter cubed.

Definition at line 129 of file MOM_EOS.F90.

```
00129   real :: r_to_kg_m3 = 1.  !< A constant that translates the units of density to kilograms per meter cubed.
```

43.9.2.13 rho_t0_s0

```
real mom_eos::eos_type::rho_t0_s0
```

The density at T=0, S=0 [kg m-3].

Definition at line 116 of file MOM_EOS.F90.

```
00116   real :: rho_t0_s0 !< The density at T=0, S=0 [kg m-3]
```

43.9.2.14 rl2_t2_to_pa

```
real mom_eos::eos_type::rl2_t2_to_pa = 1.
```

Convert pressures from R L2 T-2 to Pa.

Definition at line 130 of file MOM_EOS.F90.

```
00130   real :: rl2_t2_to_pa = 1.!< Convert pressures from R L2 T-2 to Pa.
```

43.9.2.15 tfr_s0_p0

```
real mom_eos::eos_type::tfr_s0_p0
```

The freezing potential temperature at S=0, P=0 [degC].

Definition at line 121 of file MOM_EOS.F90.

```
00121   real :: tfr_s0_p0 !< The freezing potential temperature at S=0, P=0 [degC]
```

The documentation for this type was generated from the following file:

- /home/cermak/src/MOM6/src/equation_of_state/MOM_EOS.F90

43.10 mom_ice_shelf::ice_shelf_cs Type Reference

Control structure that contains ice shelf parameters and diagnostics handles.

Public Attributes

- type(mom_restart_cs), pointer restart_csp => NULL()
  A pointer to the restart control structure for the ice shelves.
- type(ocean_grid_type) grid
  Grid for the ice-shelf model.
- type(unit_scale_type), pointer us => NULL()
  A structure containing various unit conversion factors.
- type(ocean_grid_type), pointer ocn_grid => NULL()
  A pointer to the ocean model grid The rest is private.
- real flux_factor = 1.0
  A factor that can be used to turn off ice shelf melting (flux_factor = 0) [nondim].
- character(len=128) restart_output_dir = ' '
  The directory in which to write restart files.
- type(ice_shelf_state), pointer iss => NULL()
  A structure with elements that describe the ice-shelf state.
- type(ice_shelf_dyn_cs), pointer dcs => NULL()

The control structure for the ice-shelf dynamics.

- real, dimension(:,:), pointer utide => NULL()

  An unresolved tidal velocity [L T-1 ~> m s-1].
- real ustar_bg

  A minimum value for ustar under ice shelves [Z T-1 ~> m s-1].
- real cdrag

  drag coefficient under ice shelves [nondim].
- real g_earth

  The gravitational acceleration [L2 Z-1 T-2 ~> m s-2].
- real cp

  The heat capacity of sea water [Q degC-1 ~> J kg-1 degC-1].
- real rho_ocn

  A reference ocean density [R ~> kg m-3].
- real cp_ice

  The heat capacity of fresh ice [Q degC-1 ~> J kg-1 degC-1].
- real gamma_t

  The (fixed) turbulent exchange velocity in the 2-equation formulation [Z T-1 ~> m s-1].
- real salin_ice

  The salinity of shelf ice [ppt].
- real temp_ice

  The core temperature of shelf ice [degC].
- real kv_ice

  The viscosity of ice [L4 Z-2 T-1 ~> m2 s-1].
- real density_ice

  A typical density of ice [R ~> kg m-3].
- real kv_molec

  The molecular kinematic viscosity of sea water [Z2 T-1 ~> m2 s-1].
- real kd_molec_salt

  The molecular diffusivity of salt [Z2 T-1 ~> m2 s-1].
- real kd_molec_temp

  The molecular diffusivity of heat [Z2 T-1 ~> m2 s-1].
- real lat_fusion

  The latent heat of fusion [Q ~> J kg-1].
- real gamma_t_3eq

  Nondimensional heat-transfer coefficient, used in the 3Eq. formulation.
- real gamma_s_3eq

  Nondimensional salt-transfer coefficient, used in the 3Eq. formulation This number should be specified by the user.
- real col_mass_melt_threshold

  An ocean column mass below the iceshelf below which melting does not occur [R Z ~> kg m-2].
- logical mass_from_file

  Read the ice shelf mass from a file every dt.
- real time_step

  this is the shortest timestep that the ice shelf sees, and is equal to the forcing timestep (it is passed in when the shelf is initialized - so need to reorganize MOM driver. it will be the prognistic timestep ... maybe.
- logical solo_ice_sheet

  whether the ice model is running without being coupled to the ocean
- logical gl_regularize

  whether to regularize the floatation condition at the grounding line a la Goldberg Holland Schoof 2009
- logical gl_couple

  whether to let the floatation condition be determined by ocean column thickness means update_OD_ffrac will be called (note: GL_regularize and GL_couple should be exclusive)

- logical calve_to_mask

  If true, calve any ice that passes outside of a masked area.
- real min_thickness_simple_calve

  min. ice shelf thickness criteria for calving [Z ~> m].
- real t0

  temperature at ocean surface in the restoring region [degC]
- real s0

  Salinity at ocean surface in the restoring region [ppt].
- real input_flux

  Ice volume flux at an upstream open boundary [m3 s-1].
- real input_thickness

  Ice thickness at an upstream open boundary [m].
- type(time_type) time

  The component's time.
- type(eos_type), pointer eqn_of_state => NULL()

  Type that indicates the equation of state to use.
- logical active_shelf_dynamics

  True if the ice shelf mass changes as a result the dynamic ice-shelf model.
- logical override_shelf_movement

  If true, user code specifies the shelf movement instead of using the dynamic ice-shelf mode.
- logical isthermo

  True if the ice shelf can exchange heat and mass with the underlying ocean.
- logical threeeq

  If true, the 3 equation consistency equations are used to calculate the flux at the ocean-ice interface.
- logical insulator

  If true, ice shelf is a perfect insulator.
- logical const_gamma

  If true, gamma_T is specified by the user.
- logical constant_sea_level

  if true, apply an evaporative, heat and salt fluxes. It will avoid large increase in sea level.
- real min_ocean_mass_float

  The minimum ocean mass per unit area before the ice shelf is considered to float when constant_sea_level is used [R Z ~> kg m-2].
- real cutoff_depth

  Depth above which melt is set to zero (>= 0) [Z ~> m].
- logical find_salt_root

  If true, if true find Sbdry using a quadratic eq.
- real tfr_0_0

  The freezing point at 0 pressure and 0 salinity [degC].
- real dtfr_ds

  Partial derivative of freezing temperature with salinity [degC ppt-1].
- real dtfr_dp

  Partial derivative of freezing temperature with pressure [degC T2 R-1 L-2 ~> degC Pa-1].
- integer id_read_mass

  An integer handle used in time interpolation of the ice shelf mass read from a file.
- integer id_read_area

  An integer handle used in time interpolation of the ice shelf mass read from a file.
- type(diag_ctrl), pointer diag => NULL()

  A structure that is used to control diagnostic output.
- type(user_ice_shelf_cs), pointer user_cs => NULL()

  A pointer to the control structure for user-supplied modifications to the ice shelf code.

- logical debug

  If true, write verbose checksums for debugging purposes and use reproducible sums.

- integer id_melt = -1

  Diagnostic handles.
- integer id_exch_vel_s = -1

  Diagnostic handles.
- integer id_exch_vel_t = -1

  Diagnostic handles.
- integer id_tfreeze = -1

  Diagnostic handles.
- integer id_tfl_shelf = -1

  Diagnostic handles.
- integer id_thermal_driving = -1

  Diagnostic handles.
- integer id_haline_driving = -1

  Diagnostic handles.
- integer id_u_ml = -1

  Diagnostic handles.
- integer id_v_ml = -1

  Diagnostic handles.
- integer id_sbdry = -1

  Diagnostic handles.
- integer id_h_shelf = -1

  Diagnostic handles.
- integer id_h_mask = -1

  Diagnostic handles.
- integer id_surf_elev = -1

  Diagnostic handles.
- integer id_bathym = -1

  Diagnostic handles.
- integer id_area_shelf_h = -1

  Diagnostic handles.
- integer id_ustar_shelf = -1

  Diagnostic handles.
- integer id_shelf_mass = -1

  Diagnostic handles.
- integer id_mass_flux = -1

  Diagnostic handles.

43.10.1 Detailed Description

Control structure that contains ice shelf parameters and diagnostics handles.

Definition at line 72 of file MOM_ice_shelf.F90.

43.10.2 Member Data Documentation

43.10.2.1 active_shelf_dynamics

logical mom_ice_shelf::ice_shelf_cs::active_shelf_dynamics

True if the ice shelf mass changes as a result the dynamic ice-shelf model.

Definition at line 140 of file MOM_ice_shelf.F90.

```
00140   logical :: active_shelf_dynamics       !< True if the ice shelf mass changes as a result
```

43.10.2.2 calve_to_mask

logical mom_ice_shelf::ice_shelf_cs::calve_to_mask

If true, calve any ice that passes outside of a masked area.

Definition at line 130 of file MOM_ice_shelf.F90.

```
00130   logical :: calve_to_mask  !< If true, calve any ice that passes outside of a masked area
```

43.10.2.3 cdrag

real mom_ice_shelf::ice_shelf_cs::cdrag

drag coefficient under ice shelves [nondim].

Definition at line 93 of file MOM_ice_shelf.F90.

```
00093   real :: cdrag        !< drag coefficient under ice shelves [nondim].
```

43.10.2.4 col_mass_melt_threshold

real mom_ice_shelf::ice_shelf_cs::col_mass_melt_threshold

An ocean column mass below the iceshelf below which melting does not occur [R Z ~> kg m-2].

Definition at line 111 of file MOM_ice_shelf.F90.

```
00111   real :: col_mass_melt_threshold !< An ocean column mass below the iceshelf below which melting
```

43.10.2.5 const_gamma

```
logical mom_ice_shelf::ice_shelf_cs::const_gamma
```

If true, gamma_T is specified by the user.

Definition at line 150 of file MOM_ice_shelf.F90.

```
00150   logical :: const_gamma                !< If true, gamma_T is specified by the user.
```

43.10.2.6 constant_sea_level

```
logical mom_ice_shelf::ice_shelf_cs::constant_sea_level
```

if true, apply an evaporative, heat and salt fluxes. It will avoid large increase in sea level.

Definition at line 151 of file MOM_ice_shelf.F90.

```
00151   logical :: constant_sea_level         !< if true, apply an evaporative, heat and salt
```

43.10.2.7 cp

```
real mom_ice_shelf::ice_shelf_cs::cp
```

The heat capacity of sea water [Q degC-1 ~> J kg-1 degC-1].

Definition at line 95 of file MOM_ice_shelf.F90.

```
00095   real :: cp           !< The heat capacity of sea water [Q degC-1 ~> J kg-1 degC-1].
```

43.10.2.8 cp_ice

```
real mom_ice_shelf::ice_shelf_cs::cp_ice
```

The heat capacity of fresh ice [Q degC-1 ~> J kg-1 degC-1].

Definition at line 97 of file MOM_ice_shelf.F90.

```
00097   real :: cp_ice       !< The heat capacity of fresh ice [Q degC-1 ~> J kg-1 degC-1].
```

43.10.2.9 cutoff_depth

```
real mom_ice_shelf::ice_shelf_cs::cutoff_depth
```

Depth above which melt is set to zero (>= 0) [Z ~> m].

Definition at line 156 of file MOM_ice_shelf.F90.

```
00156   real    :: cutoff_depth               !< Depth above which melt is set to zero (>= 0) [Z ~> m].
```

43.10.2.10 dcs

```
type(ice_shelf_dyn_cs), pointer mom_ice_shelf::ice_shelf_cs::dcs => NULL()
```

The control structure for the ice-shelf dynamics.

Definition at line 87 of file MOM_ice_shelf.F90.

```
00087   type(ice_shelf_dyn_cs), pointer :: dcs => null() !< The control structure for the ice-shelf dynamics.
```

43.10.2.11 debug

```
logical mom_ice_shelf::ice_shelf_cs::debug
```

If true, write verbose checksums for debugging purposes and use reproducible sums.

Definition at line 182 of file MOM_ice_shelf.F90.

```
00182   logical :: debug                !< If true, write verbose checksums for debugging purposes
```

43.10.2.12 density_ice

```
real mom_ice_shelf::ice_shelf_cs::density_ice
```

A typical density of ice [R ~> kg m-3].

Definition at line 103 of file MOM_ice_shelf.F90.

```
00103   real :: density_ice  !< A typical density of ice [R ~> kg m-3].
```

43.10.2.13 diag

```
type(diag_ctrl), pointer mom_ice_shelf::ice_shelf_cs::diag => NULL()
```

A structure that is used to control diagnostic output.

Definition at line 178 of file MOM_ice_shelf.F90.

```
00178   type(diag_ctrl), pointer :: diag => null() !< A structure that is used to control diagnostic output.
```

43.10.2.14 dtfr_dp

```
real mom_ice_shelf::ice_shelf_cs::dtfr_dp
```

Partial derivative of freezing temperature with pressure [degC T2 R-1 L-2 ~> degC Pa-1].

Definition at line 160 of file MOM_ice_shelf.F90.

```
00160   real    :: dtfr_dp                  !< Partial derivative of freezing temperature with
```

43.10.2.15 dtfr_ds

```
real mom_ice_shelf::ice_shelf_cs::dtfr_ds
```

Partial derivative of freezing temperature with salinity [degC ppt-1].

Definition at line 159 of file MOM_ice_shelf.F90.

```
00159   real    :: dtfr_ds                  !< Partial derivative of freezing temperature with salinity [degC
      ppt-1]
```

43.10.2.16 eqn_of_state

```
type(eos_type), pointer mom_ice_shelf::ice_shelf_cs::eqn_of_state => NULL()
```

Type that indicates the equation of state to use.

Definition at line 138 of file MOM_ice_shelf.F90.

```
00138   type(eos_type), pointer :: eqn_of_state => null() !< Type that indicates the
```

43.10.2.17 find_salt_root

```
logical mom_ice_shelf::ice_shelf_cs::find_salt_root
```

If true, if true find Sbdry using a quadratic eq.

Definition at line 157 of file MOM_ice_shelf.F90.

```
00157   logical :: find_salt_root             !< If true, if true find Sbdry using a quadratic eq.
```

43.10.2.18 flux_factor

```
real mom_ice_shelf::ice_shelf_cs::flux_factor = 1.0
```

A factor that can be used to turn off ice shelf melting (flux_factor = 0) [nondim].

Definition at line 82 of file MOM_ice_shelf.F90.

```
00082   real ::   flux_factor = 1.0           !< A factor that can be used to turn off ice shelf
```

43.10.2.19 g_earth

```
real mom_ice_shelf::ice_shelf_cs::g_earth
```

The gravitational acceleration [L2 Z-1 T-2 ~> m s-2].

Definition at line 94 of file MOM_ice_shelf.F90.

```
00094   real :: g_earth      !< The gravitational acceleration [L2 Z-1 T-2 ~> m s-2]
```

43.10.2.20 gamma_s_3eq

```
real mom_ice_shelf::ice_shelf_cs::gamma_s_3eq
```

Nondimensional salt-transfer coefficient, used in the 3Eq. formulation This number should be specified by the user.

Definition at line 109 of file MOM_ice_shelf.F90.

```
00109   real :: gamma_s_3eq  !<  Nondimensional salt-transfer coefficient, used in the 3Eq. formulation
```

43.10.2.21 gamma_t

```
real mom_ice_shelf::ice_shelf_cs::gamma_t
```

The (fixed) turbulent exchange velocity in the 2-equation formulation [Z T-1 ~> m s-1].

Definition at line 98 of file MOM_ice_shelf.F90.

```
00098   real :: gamma_t      !< The (fixed) turbulent exchange velocity in the
```

43.10.2.22 gamma_t_3eq

```
real mom_ice_shelf::ice_shelf_cs::gamma_t_3eq
```

Nondimensional heat-transfer coefficient, used in the 3Eq. formulation.

Definition at line 108 of file MOM_ice_shelf.F90.

```
00108   real :: gamma_t_3eq  !<  Nondimensional heat-transfer coefficient, used in the 3Eq. formulation
```

43.10.2.23 gl_couple

```
logical mom_ice_shelf::ice_shelf_cs::gl_couple
```

whether to let the floatation condition be determined by ocean column thickness means update_OD_ffrac will be called (note: GL_regularize and GL_couple should be exclusive)

Definition at line 126 of file MOM_ice_shelf.F90.

```
00126   logical :: gl_couple      !< whether to let the floatation condition be
```

43.10.2.24 gl_regularize

```
logical mom_ice_shelf::ice_shelf_cs::gl_regularize
```

whether to regularize the floatation condition at the grounding line a la Goldberg Holland Schoof 2009

Definition at line 124 of file MOM_ice_shelf.F90.

```
00124   logical :: gl_regularize  !< whether to regularize the floatation condition
```

43.10.2.25 grid

```
type(ocean_grid_type) mom_ice_shelf::ice_shelf_cs::grid
```

Grid for the ice-shelf model.

Definition at line 76 of file MOM_ice_shelf.F90.

```
00076   type(ocean_grid_type) :: grid          !< Grid for the ice-shelf model
```

43.10.2.26 id_area_shelf_h

```
integer mom_ice_shelf::ice_shelf_cs::id_area_shelf_h = -1
```

Diagnostic handles.

Definition at line 163 of file MOM_ice_shelf.F90.

43.10.2.27 id_bathym

```
integer mom_ice_shelf::ice_shelf_cs::id_bathym = -1
```

Diagnostic handles.

Definition at line 163 of file MOM_ice_shelf.F90.

43.10.2.28 id_exch_vel_s

```
integer mom_ice_shelf::ice_shelf_cs::id_exch_vel_s = -1
```

Diagnostic handles.

Definition at line 163 of file MOM_ice_shelf.F90.

43.10.2.29 id_exch_vel_t

```
integer mom_ice_shelf::ice_shelf_cs::id_exch_vel_t = -1
```

Diagnostic handles.

Definition at line 163 of file MOM_ice_shelf.F90.

43.10.2.30 id_h_mask

```
integer mom_ice_shelf::ice_shelf_cs::id_h_mask = -1
```

Diagnostic handles.

Definition at line 163 of file MOM_ice_shelf.F90.

43.10.2.31 id_h_shelf

```
integer mom_ice_shelf::ice_shelf_cs::id_h_shelf = -1
```

Diagnostic handles.

Definition at line 163 of file MOM_ice_shelf.F90.

43.10.2.32 id_haline_driving

```
integer mom_ice_shelf::ice_shelf_cs::id_haline_driving = -1
```

Diagnostic handles.

Definition at line 163 of file MOM_ice_shelf.F90.

43.10.2.33 id_mass_flux

```
integer mom_ice_shelf::ice_shelf_cs::id_mass_flux = -1
```

Diagnostic handles.

Definition at line 163 of file MOM_ice_shelf.F90.

43.10.2.34 id_melt

```
integer mom_ice_shelf::ice_shelf_cs::id_melt = -1
```

Diagnostic handles.

Definition at line 163 of file MOM_ice_shelf.F90.

```
00163   integer :: id_melt = -1, id_exch_vel_s = -1, id_exch_vel_t = -1, &
00164              id_tfreeze = -1, id_tfl_shelf = -1, &
00165              id_thermal_driving = -1, id_haline_driving = -1, &
00166              id_u_ml = -1, id_v_ml = -1, id_sbdry = -1, &
00167              id_h_shelf = -1, id_h_mask = -1, &
00168              id_surf_elev = -1, id_bathym = -1, &
00169              id_area_shelf_h = -1, &
00170              id_ustar_shelf = -1, id_shelf_mass = -1, id_mass_flux = -1
```

43.10.2.35 id_read_area

```
integer mom_ice_shelf::ice_shelf_cs::id_read_area
```

An integer handle used in time interpolation of the ice shelf mass read from a file.

Definition at line 175 of file MOM_ice_shelf.F90.

```
00175   integer :: id_read_area !< An integer handle used in time interpolation of
```

43.10.2.36 id_read_mass

```
integer mom_ice_shelf::ice_shelf_cs::id_read_mass
```

An integer handle used in time interpolation of the ice shelf mass read from a file.

Definition at line 173 of file MOM_ice_shelf.F90.

```
00173   integer :: id_read_mass !< An integer handle used in time interpolation of
```

43.10.2.37 id_sbdry

```
integer mom_ice_shelf::ice_shelf_cs::id_sbdry = -1
```

Diagnostic handles.

Definition at line 163 of file MOM_ice_shelf.F90.

43.10.2.38 id_shelf_mass

```
integer mom_ice_shelf::ice_shelf_cs::id_shelf_mass = -1
```

Diagnostic handles.

Definition at line 163 of file MOM_ice_shelf.F90.

43.10.2.39 id_surf_elev

```
integer mom_ice_shelf::ice_shelf_cs::id_surf_elev = -1
```

Diagnostic handles.

Definition at line 163 of file MOM_ice_shelf.F90.

43.10.2.40 id_tfl_shelf

```
integer mom_ice_shelf::ice_shelf_cs::id_tfl_shelf = -1
```

Diagnostic handles.

Definition at line 163 of file MOM_ice_shelf.F90.

43.10.2.41 id_tfreeze

```
integer mom_ice_shelf::ice_shelf_cs::id_tfreeze = -1
```

Diagnostic handles.

Definition at line 163 of file MOM_ice_shelf.F90.

43.10.2.42 id_thermal_driving

```
integer mom_ice_shelf::ice_shelf_cs::id_thermal_driving = -1
```

Diagnostic handles.

Definition at line 163 of file MOM_ice_shelf.F90.

43.10.2.43 id_u_ml

```
integer mom_ice_shelf::ice_shelf_cs::id_u_ml = -1
```

Diagnostic handles.

Definition at line 163 of file MOM_ice_shelf.F90.

43.10.2.44 id_ustar_shelf

```
integer mom_ice_shelf::ice_shelf_cs::id_ustar_shelf = -1
```

Diagnostic handles.

Definition at line 163 of file MOM_ice_shelf.F90.

43.10.2.45 id_v_ml

```
integer mom_ice_shelf::ice_shelf_cs::id_v_ml = -1
```

Diagnostic handles.

Definition at line 163 of file MOM_ice_shelf.F90.

43.10.2.46 input_flux

```
real mom_ice_shelf::ice_shelf_cs::input_flux
```

Ice volume flux at an upstream open boundary [m3 s-1].

Definition at line 134 of file MOM_ice_shelf.F90.

```
00134   real :: input_flux        !< Ice volume flux at an upstream open boundary [m3 s-1].
```

43.10.2.47 input_thickness

```
real mom_ice_shelf::ice_shelf_cs::input_thickness
```

Ice thickness at an upstream open boundary [m].

Definition at line 135 of file MOM_ice_shelf.F90.

```
00135   real :: input_thickness   !< Ice thickness at an upstream open boundary [m].
```

43.10.2.48 insulator

```
logical mom_ice_shelf::ice_shelf_cs::insulator
```

If true, ice shelf is a perfect insulator.

Definition at line 149 of file MOM_ice_shelf.F90.

```
00149   logical :: insulator                  !< If true, ice shelf is a perfect insulator
```

43.10.2.49 iss

```
type(ice_shelf_state), pointer mom_ice_shelf::ice_shelf_cs::iss => NULL()
```

A structure with elements that describe the ice-shelf state.

Definition at line 85 of file MOM_ice_shelf.F90.

```
00085   type(ice_shelf_state), pointer :: iss => null() !< A structure with elements that describe
```

43.10.2.50 isthermo

```
logical mom_ice_shelf::ice_shelf_cs::isthermo
```

True if the ice shelf can exchange heat and mass with the underlying ocean.

Definition at line 144 of file MOM_ice_shelf.F90.

```
00144   logical :: isthermo                   !< True if the ice shelf can exchange heat and
```

43.10.2.51 kd_molec_salt

```
real mom_ice_shelf::ice_shelf_cs::kd_molec_salt
```

The molecular diffusivity of salt [Z2 T-1 ~> m2 s-1].

Definition at line 105 of file MOM_ice_shelf.F90.

```
00105   real :: kd_molec_salt!< The molecular diffusivity of salt [Z2 T-1 ~> m2 s-1].
```

43.10.2.52 kd_molec_temp

```
real mom_ice_shelf::ice_shelf_cs::kd_molec_temp
```

The molecular diffusivity of heat [Z2 T-1 ~> m2 s-1].

Definition at line 106 of file MOM_ice_shelf.F90.

```
00106   real :: kd_molec_temp!< The molecular diffusivity of heat [Z2 T-1 ~> m2 s-1].
```

43.10.2.53 kv_ice

```
real mom_ice_shelf::ice_shelf_cs::kv_ice
```

The viscosity of ice [L4 Z-2 T-1 ~> m2 s-1].

Definition at line 102 of file MOM_ice_shelf.F90.

```
00102   real :: kv_ice       !< The viscosity of ice [L4 Z-2 T-1 ~> m2 s-1].
```

43.10.2.54 kv_molec

```
real mom_ice_shelf::ice_shelf_cs::kv_molec
```

The molecular kinematic viscosity of sea water [Z2 T-1 ~> m2 s-1].

Definition at line 104 of file MOM_ice_shelf.F90.

```
00104   real :: kv_molec     !< The molecular kinematic viscosity of sea water [Z2 T-1 ~> m2 s-1].
```

43.10.2.55 lat_fusion

```
real mom_ice_shelf::ice_shelf_cs::lat_fusion
```

The latent heat of fusion [Q ~> J kg-1].

Definition at line 107 of file MOM_ice_shelf.F90.

```
00107   real :: lat_fusion   !< The latent heat of fusion [Q ~> J kg-1].
```

43.10.2.56 mass_from_file

```
logical mom_ice_shelf::ice_shelf_cs::mass_from_file
```

Read the ice shelf mass from a file every dt.

Definition at line 113 of file MOM_ice_shelf.F90.

```
00113   logical :: mass_from_file !< Read the ice shelf mass from a file every dt
```

43.10.2.57 min_ocean_mass_float

```
real mom_ice_shelf::ice_shelf_cs::min_ocean_mass_float
```

The minimum ocean mass per unit area before the ice shelf is considered to float when constant_sea_level is used [R Z ~> kg m-2].

Definition at line 153 of file MOM_ice_shelf.F90.

```
00153   real    :: min_ocean_mass_float       !< The minimum ocean mass per unit area before the ice
```

43.10.2.58 min_thickness_simple_calve

```
real mom_ice_shelf::ice_shelf_cs::min_thickness_simple_calve
```

min. ice shelf thickness criteria for calving [Z ~> m].

Definition at line 131 of file MOM_ice_shelf.F90.

```
00131   real :: min_thickness_simple_calve !< min. ice shelf thickness criteria for calving [Z ~> m].
```

43.10.2.59 ocn_grid

```
type(ocean_grid_type), pointer mom_ice_shelf::ice_shelf_cs::ocn_grid => NULL()
```

A pointer to the ocean model grid The rest is private.

Definition at line 80 of file MOM_ice_shelf.F90.

```
00080   type(ocean_grid_type), pointer :: ocn_grid => null() !< A pointer to the ocean model grid
```

43.10.2.60 override_shelf_movement

```
logical mom_ice_shelf::ice_shelf_cs::override_shelf_movement
```

If true, user code specifies the shelf movement instead of using the dynamic ice-shelf mode.

Definition at line 142 of file MOM_ice_shelf.F90.

```
00142   logical :: override_shelf_movement     !< If true, user code specifies the shelf movement
```

43.10.2.61 restart_csp

```
type(mom_restart_cs), pointer mom_ice_shelf::ice_shelf_cs::restart_csp => NULL()
```

A pointer to the restart control structure for the ice shelves.

Definition at line 74 of file MOM_ice_shelf.F90.

```
00074   type(mom_restart_cs), pointer :: restart_csp => null() !< A pointer to the restart control
```

43.10.2.62 restart_output_dir

```
character(len=128) mom_ice_shelf::ice_shelf_cs::restart_output_dir = ' '
```

The directory in which to write restart files.

Definition at line 84 of file MOM_ice_shelf.F90.

```
00084   character(len=128) :: restart_output_dir = ' ' !< The directory in which to write restart files
```

43.10.2.63 rho_ocn

```
real mom_ice_shelf::ice_shelf_cs::rho_ocn
```

A reference ocean density [R ~> kg m-3].

Definition at line 96 of file MOM_ice_shelf.F90.

```
00096   real :: rho_ocn      !< A reference ocean density [R ~> kg m-3].
```

43.10.2.64 s0

```
real mom_ice_shelf::ice_shelf_cs::s0
```

Salinity at ocean surface in the restoring region [ppt].

Definition at line 133 of file MOM_ice_shelf.F90.

```
00133   real :: s0               !< Salinity at ocean surface in the restoring region [ppt].
```

43.10.2.65 salin_ice

```
real mom_ice_shelf::ice_shelf_cs::salin_ice
```

The salinity of shelf ice [ppt].

Definition at line 100 of file MOM_ice_shelf.F90.

```
00100   real :: salin_ice    !< The salinity of shelf ice [ppt].
```

43.10.2.66 solo_ice_sheet

```
logical mom_ice_shelf::ice_shelf_cs::solo_ice_sheet
```

whether the ice model is running without being coupled to the ocean

Definition at line 122 of file MOM_ice_shelf.F90.

```
00122   logical :: solo_ice_sheet !< whether the ice model is running without being
```

43.10.2.67 t0

```
real mom_ice_shelf::ice_shelf_cs::t0
```

temperature at ocean surface in the restoring region [degC]

Definition at line 132 of file MOM_ice_shelf.F90.

```
00132   real :: t0               !< temperature at ocean surface in the restoring region [degC]
```

43.10.2.68 temp_ice

```
real mom_ice_shelf::ice_shelf_cs::temp_ice
```

The core temperature of shelf ice [degC].

Definition at line 101 of file MOM_ice_shelf.F90.

```
00101   real :: temp_ice     !< The core temperature of shelf ice [degC].
```

43.10.2.69 tfr_0_0

```
real mom_ice_shelf::ice_shelf_cs::tfr_0_0
```

The freezing point at 0 pressure and 0 salinity [degC].

Definition at line 158 of file MOM_ice_shelf.F90.

```
00158   real    :: tfr_0_0                   !< The freezing point at 0 pressure and 0 salinity [degC]
```

43.10.2.70 threeeq

```
logical mom_ice_shelf::ice_shelf_cs::threeeq
```

If true, the 3 equation consistency equations are used to calculate the flux at the ocean-ice interface.

Definition at line 146 of file MOM_ice_shelf.F90.

```
00146   logical :: threeeq                   !< If true, the 3 equation consistency equations are
```

43.10.2.71 time

```
type(time_type) mom_ice_shelf::ice_shelf_cs::time
```

The component's time.

Definition at line 137 of file MOM_ice_shelf.F90.

```
00137   type(time_type) :: time              !< The component's time.
```

43.10.2.72 time_step

```
real mom_ice_shelf::ice_shelf_cs::time_step
```

this is the shortest timestep that the ice shelf sees, and is equal to the forcing timestep (it is passed in when the shelf is initialized - so need to reorganize MOM driver. it will be the prognistic timestep ... maybe.

Definition at line 117 of file MOM_ice_shelf.F90.

```
00117   real :: time_step    !< this is the shortest timestep that the ice shelf sees, and
```

43.10.2.73 us

```
type(unit_scale_type), pointer mom_ice_shelf::ice_shelf_cs::us => NULL()
```

A structure containing various unit conversion factors.

Definition at line 77 of file MOM_ice_shelf.F90.

```
00077   type(unit_scale_type), pointer :: &
00078     us => null()       !< A structure containing various unit conversion factors
```

43.10.2.74 user_cs

```
type(user_ice_shelf_cs), pointer mom_ice_shelf::ice_shelf_cs::user_cs => NULL()
```

A pointer to the control structure for user-supplied modifications to the ice shelf code.

Definition at line 179 of file MOM_ice_shelf.F90.

```
00179   type(user_ice_shelf_cs), pointer :: user_cs => null() !< A pointer to the control structure for
```

43.10.2.75 ustar_bg

```
real mom_ice_shelf::ice_shelf_cs::ustar_bg
```

A minimum value for ustar under ice shelves [Z T-1 ~> m s-1].

Definition at line 92 of file MOM_ice_shelf.F90.

```
00092   real :: ustar_bg     !< A minimum value for ustar under ice shelves [Z T-1 ~> m s-1].
```

43.10.2.76 utide

```
real, dimension(:,:), pointer mom_ice_shelf::ice_shelf_cs::utide => NULL()
```

An unresolved tidal velocity [L T-1 ~> m s-1].

Definition at line 89 of file MOM_ice_shelf.F90.

```
00089   real, pointer, dimension(:,:) :: &
00090     utide   => null()  !< An unresolved tidal velocity [L T-1 ~> m s-1]
```

The documentation for this type was generated from the following file:

- /home/cermak/src/MOM6/src/ice_shelf/MOM_ice_shelf.F90

43.11 mom_meke::meke_cs Type Reference

Control structure that contains MEKE parameters and diagnostics handles.

Public Attributes

- real, dimension(:,:), pointer equilibrium_value => NULL()

  The equilbrium value of MEKE to be calculated at each time step [L2 T-2 ~> m2 s-2].
- real meke_frcoeff

  Efficiency of conversion of ME into MEKE [nondim].
- real meke_gmcoeff

  Efficiency of conversion of PE into MEKE [nondim].
- real meke_gmecoeff

  Efficiency of conversion of MEKE into ME by GME [nondim].
- real meke_damping

  Local depth-independent MEKE dissipation rate [T-1 ~> s-1].
- real meke_cd_scale

  *The ratio of the bottom eddy velocity to the column mean eddy velocity, i.e. sqrt(2*MEKE). This should be less than 1 to account for the surface intensification of MEKE.*
- real meke_cb

  Coefficient in the γ_{bot} expression [nondim].
- real meke_min_gamma

  Minimum value of gamma_b^2 allowed [nondim].
- real meke_ct

  Coefficient in the γ_{bt} expression [nondim].
- logical visc_drag

  If true use the vertvisc_type to calculate bottom drag.
- logical meke_geometric

  If true, uses the GM coefficient formulation from the GEOMETRIC framework (Marshall et al., 2012)
- real meke_geometric_alpha

  The nondimensional coefficient governing the efficiency of the GEOMETRIC thickness diffusion.
- logical meke_equilibrium_alt

  If true, use an alternative calculation for the equilibrium value of MEKE.
- logical meke_equilibrium_restoring

If true, restore MEKE back to its equilibrium value, which is calculated at each time step.

- logical gm_src_alt

  *If true, use the GM energy conversion form S^2*N^2*kappa rather than the streamfunction for the MEKE GM source term.*
- logical rd_as_max_scale

  If true the length scale can not exceed the first baroclinic deformation radius.
- logical use_old_lscale

  Use the old formula for mixing length scale.
- logical use_min_lscale

  Use simple minimum for mixing length scale.
- real cdrag

  The bottom drag coefficient for MEKE [nondim].
- real meke_bgsrc

  Background energy source for MEKE [L2 T-3 ~> W kg-1] (= m2 s-3).
- real meke_dtscale

  Scale factor to accelerate time-stepping [nondim].
- real meke_khcoeff

  Scaling factor to convert MEKE into Kh [nondim].
- real meke_uscale

  MEKE velocity scale for bottom drag [L T-1 ~> m s-1].
- real meke_kh

  Background lateral diffusion of MEKE [L2 T-1 ~> m2 s-1].
- real meke_k4

  Background bi-harmonic diffusivity (of MEKE) [L4 T-1 ~> m4 s-1].
- real khmeke_fac

  A factor relating MEKEKh to the diffusivity used for MEKE itself [nondim].
- real viscosity_coeff_ku

  The scaling coefficient in the expression for viscosity used to parameterize lateral harmonic momentum mixing by unresolved eddies represented by MEKE.
- real viscosity_coeff_au

  The scaling coefficient in the expression for viscosity used to parameterize lateral biharmonic momentum mixing by unresolved eddies represented by MEKE.
- real lfixed

  Fixed mixing length scale [L ~> m].
- real adeform

  Weighting towards deformation scale of mixing length [nondim].
- real arhines

  Weighting towards Rhines scale of mixing length [nondim].
- real africt

  Weighting towards frictional arrest scale of mixing length [nondim].
- real aeady

  Weighting towards Eady scale of mixing length [nondim].
- real agrid

  Weighting towards grid scale of mixing length [nondim].
- real meke_advection_factor

  A scaling in front of the advection of MEKE [nondim].
- real meke_topographic_beta

  Weight for how much topographic beta is considered when computing beta in Rhines scale [nondim].
- real meke_restoring_rate

  Inverse of the timescale used to nudge MEKE toward its equilibrium value [s-1].
- logical kh_flux_enabled

If true, lateral diffusive MEKE flux is enabled.

- logical initialize

  If True, invokes a steady state solver to calculate MEKE.

- logical debug

  If true, write out checksums of data for debugging.

- type(diag_ctrl), pointer diag => NULL()

  A type that regulates diagnostics output.

- integer id_clock_pass

  Clock for group pass calls.

- type(group_pass_type) pass_meke

  Group halo pass handle for MEKEMEKE and maybe MEKEKh_diff.

- type(group_pass_type) pass_kh

  Group halo pass handle for MEKEKh, MEKEKu, and/or MEKEAu.

- integer id_meke = -1

  Diagnostic handles.

- integer id_ue = -1

  Diagnostic handles.

- integer id_kh = -1

  Diagnostic handles.

- integer id_src = -1

  Diagnostic handles.

- integer id_ub = -1

  Diagnostic handles.

- integer id_ut = -1

  Diagnostic handles.

- integer id_gm_src = -1

  Diagnostic handles.

- integer id_mom_src = -1

  Diagnostic handles.

- integer id_gme_snk = -1

  Diagnostic handles.

- integer id_decay = -1

  Diagnostic handles.

- integer id_khmeke_u = -1

  Diagnostic handles.

- integer id_khmeke_v = -1

  Diagnostic handles.

- integer id_ku = -1

  Diagnostic handles.

- integer id_au = -1

  Diagnostic handles.

- integer id_le = -1

  Diagnostic handles.

- integer id_gamma_b = -1

  Diagnostic handles.

- integer id_gamma_t = -1

  Diagnostic handles.

- integer id_lrhines = -1

  Diagnostic handles.
- integer id_leady = -1

  Diagnostic handles.
- integer id_meke_equilibrium = -1

  Diagnostic handles.

43.11.1 Detailed Description

Control structure that contains MEKE parameters and diagnostics handles.

Definition at line 31 of file MOM_MEKE.F90.

43.11.2 Member Data Documentation

43.11.2.1 adeform

```
real mom_meke::meke_cs::adeform
```

Weighting towards deformation scale of mixing length [nondim].

Definition at line 76 of file MOM_MEKE.F90.

```
00076   real :: adeform       !< Weighting towards deformation scale of mixing length [nondim]
```

43.11.2.2 aeady

```
real mom_meke::meke_cs::aeady
```

Weighting towards Eady scale of mixing length [nondim].

Definition at line 79 of file MOM_MEKE.F90.

```
00079   real :: aeady         !< Weighting towards Eady scale of mixing length [nondim]
```

43.11.2.3 africt

```
real mom_meke::meke_cs::africt
```

Weighting towards frictional arrest scale of mixing length [nondim].

Definition at line 78 of file MOM_MEKE.F90.

```
00078   real :: africt        !< Weighting towards frictional arrest scale of mixing length [nondim]
```

43.11.2.4 agrid

```
real mom_meke::meke_cs::agrid
```

Weighting towards grid scale of mixing length [nondim].

Definition at line 80 of file MOM_MEKE.F90.

```
00080   real :: agrid         !< Weighting towards grid scale of mixing length [nondim]
```

43.11.2.5 arhines

```
real mom_meke::meke_cs::arhines
```

Weighting towards Rhines scale of mixing length [nondim].

Definition at line 77 of file MOM_MEKE.F90.

```
00077   real :: arhines       !< Weighting towards Rhines scale of mixing length [nondim]
```

43.11.2.6 cdrag

```
real mom_meke::meke_cs::cdrag
```

The bottom drag coefficient for MEKE [nondim].

Definition at line 60 of file MOM_MEKE.F90.

```
00060   real :: cdrag         !< The bottom drag coefficient for MEKE [nondim].
```

43.11.2.7 debug

```
logical mom_meke::meke_cs::debug
```

If true, write out checksums of data for debugging.

Definition at line 88 of file MOM_MEKE.F90.

```
00088   logical :: debug      !< If true, write out checksums of data for debugging
```

43.11.2.8 diag

```
type(diag_ctrl), pointer mom_meke::meke_cs::diag => NULL()
```

A type that regulates diagnostics output.

Definition at line 90 of file MOM_MEKE.F90.

```
00090   type(diag_ctrl), pointer :: diag => null() !< A type that regulates diagnostics output
```

43.11.2.9 equilibrium_value

```
real, dimension(:,:), pointer mom_meke::meke_cs::equilibrium_value => NULL()
```

The equilbrium value of MEKE to be calculated at each time step [L2 T-2 ~> m2 s-2].

Definition at line 33 of file MOM_MEKE.F90.

```
00033   real, dimension(:,:), pointer :: equilibrium_value => null() !< The equilbrium value
```

43.11.2.10 gm_src_alt

```
logical mom_meke::meke_cs::gm_src_alt
```

If true, use the GM energy conversion form S^2*N^2*kappa rather than the streamfunction for the MEKE GM source term.

Definition at line 54 of file MOM_MEKE.F90.

```
00054   logical :: gm_src_alt !< If true, use the GM energy conversion form S^2*N^2*kappa rather
```

43.11.2.11 id_au

```
integer mom_meke::meke_cs::id_au = -1
```

Diagnostic handles.

Definition at line 95 of file MOM_MEKE.F90.

43.11.2.12 id_clock_pass

```
integer mom_meke::meke_cs::id_clock_pass
```

Clock for group pass calls.

Definition at line 102 of file MOM_MEKE.F90.

```
00102   integer :: id_clock_pass !< Clock for group pass calls
```

43.11.2.13 id_decay

```
integer mom_meke::meke_cs::id_decay = -1
```

Diagnostic handles.

Definition at line 94 of file MOM_MEKE.F90.

43.11.2.14 id_gamma_b

```
integer mom_meke::meke_cs::id_gamma_b = -1
```

Diagnostic handles.

Definition at line 96 of file MOM_MEKE.F90.

43.11.2.15 id_gamma_t

```
integer mom_meke::meke_cs::id_gamma_t = -1
```

Diagnostic handles.

Definition at line 96 of file MOM_MEKE.F90.

43.11.2.16 id_gm_src

```
integer mom_meke::meke_cs::id_gm_src = -1
```

Diagnostic handles.

Definition at line 94 of file MOM_MEKE.F90.

```
00094   integer :: id_gm_src = -1, id_mom_src = -1, id_gme_snk = -1, id_decay = -1
```

43.11.2.17 id_gme_snk

```
integer mom_meke::meke_cs::id_gme_snk = -1
```

Diagnostic handles.

Definition at line 94 of file MOM_MEKE.F90.

43.11.2.18 id_kh

```
integer mom_meke::meke_cs::id_kh = -1
```

Diagnostic handles.

Definition at line 92 of file MOM_MEKE.F90.

43.11.2.19 id_khmeke_u

```
integer mom_meke::meke_cs::id_khmeke_u = -1
```

Diagnostic handles.

Definition at line 95 of file MOM_MEKE.F90.

```
00095   integer :: id_khmeke_u = -1, id_khmeke_v = -1, id_ku = -1, id_au = -1
```

43.11.2.20 id_khmeke_v

```
integer mom_meke::meke_cs::id_khmeke_v = -1
```

Diagnostic handles.

Definition at line 95 of file MOM_MEKE.F90.

43.11.2.21 id_ku

```
integer mom_meke::meke_cs::id_ku = -1
```

Diagnostic handles.

Definition at line 95 of file MOM_MEKE.F90.

43.11.2.22 id_le

```
integer mom_meke::meke_cs::id_le = -1
```

Diagnostic handles.

Definition at line 96 of file MOM_MEKE.F90.

```
00096   integer :: id_le = -1, id_gamma_b = -1, id_gamma_t = -1
```

43.11.2.23 id_leady

```
integer mom_meke::meke_cs::id_leady = -1
```

Diagnostic handles.

Definition at line 97 of file MOM_MEKE.F90.

43.11.2.24 id_lrhines

```
integer mom_meke::meke_cs::id_lrhines = -1
```

Diagnostic handles.

Definition at line 97 of file MOM_MEKE.F90.

```
00097   integer :: id_lrhines = -1, id_leady = -1
```

43.11.2.25 id_meke

```
integer mom_meke::meke_cs::id_meke = -1
```

Diagnostic handles.

Definition at line 92 of file MOM_MEKE.F90.

```
00092   integer :: id_meke = -1, id_ue = -1, id_kh = -1, id_src = -1
```

43.11.2.26 id_meke_equilibrium

```
integer mom_meke::meke_cs::id_meke_equilibrium = -1
```

Diagnostic handles.

Definition at line 98 of file MOM_MEKE.F90.

```
00098   integer :: id_meke_equilibrium = -1
```

43.11.2.27 id_mom_src

```
integer mom_meke::meke_cs::id_mom_src = -1
```

Diagnostic handles.

Definition at line 94 of file MOM_MEKE.F90.

43.11.2.28 id_src

```
integer mom_meke::meke_cs::id_src = -1
```

Diagnostic handles.

Definition at line 92 of file MOM_MEKE.F90.

43.11.2.29 id_ub

```
integer mom_meke::meke_cs::id_ub = -1
```

Diagnostic handles.

Definition at line 93 of file MOM_MEKE.F90.

```
00093   integer :: id_ub = -1, id_ut = -1
```

43.11.2.30 id_ue

```
integer mom_meke::meke_cs::id_ue = -1
```

Diagnostic handles.

Definition at line 92 of file MOM_MEKE.F90.

43.11.2.31 id_ut

```
integer mom_meke::meke_cs::id_ut = -1
```

Diagnostic handles.

Definition at line 93 of file MOM_MEKE.F90.

43.11.2.32 initialize

```
logical mom_meke::meke_cs::initialize
```

If True, invokes a steady state solver to calculate MEKE.

Definition at line 87 of file MOM_MEKE.F90.

```
00087   logical :: initialize !< If True, invokes a steady state solver to calculate MEKE.
```

43.11.2.33 kh_flux_enabled

```
logical mom_meke::meke_cs::kh_flux_enabled
```

If true, lateral diffusive MEKE flux is enabled.

Definition at line 86 of file MOM_MEKE.F90.

```
00086   logical :: kh_flux_enabled !< If true, lateral diffusive MEKE flux is enabled.
```

43.11.2.34 khmeke_fac

```
real mom_meke::meke_cs::khmeke_fac
```

A factor relating MEKEKh to the diffusivity used for MEKE itself [nondim].

Definition at line 67 of file MOM_MEKE.F90.

```
00067   real :: khmeke_fac    !< A factor relating MEKE%Kh to the diffusivity used for
```

43.11.2.35 lfixed

```
real mom_meke::meke_cs::lfixed
```

Fixed mixing length scale [L ~> m].

Definition at line 75 of file MOM_MEKE.F90.

```
00075   real :: lfixed        !< Fixed mixing length scale [L ~> m].
```

43.11.2.36 meke_advection_factor

```
real mom_meke::meke_cs::meke_advection_factor
```

A scaling in front of the advection of MEKE [nondim].

Definition at line 81 of file MOM_MEKE.F90.

```
00081   real :: meke_advection_factor !< A scaling in front of the advection of MEKE [nondim]
```

43.11.2.37 meke_bgsrc

```
real mom_meke::meke_cs::meke_bgsrc
```

Background energy source for MEKE [L2 T-3 ~> W kg-1] (= m2 s-3).

Definition at line 61 of file MOM_MEKE.F90.

```
00061   real :: meke_bgsrc    !< Background energy source for MEKE [L2 T-3 ~> W kg-1] (= m2 s-3).
```

43.11.2.38 meke_cb

```
real mom_meke::meke_cs::meke_cb
```

Coefficient in the γ_{bot} expression [nondim].

Definition at line 42 of file MOM_MEKE.F90.

```
00042   real :: meke_cb       !< Coefficient in the \f$\gamma_{bot}\f$ expression [nondim]
```

43.11.2.39 meke_cd_scale

```
real mom_meke::meke_cs::meke_cd_scale
```

The ratio of the bottom eddy velocity to the column mean eddy velocity, i.e. sqrt(2∗MEKE). This should be less than 1 to account for the surface intensification of MEKE.

Definition at line 39 of file MOM_MEKE.F90.

```
00039   real :: meke_cd_scale !< The ratio of the bottom eddy velocity to the column mean
```

43.11.2.40 meke_ct

```
real mom_meke::meke_cs::meke_ct
```

Coefficient in the γ_{bt} expression [nondim].

Definition at line 44 of file MOM_MEKE.F90.

```
00044   real :: meke_ct       !< Coefficient in the \f$\gamma_{bt}\f$ expression [nondim]
```

43.11.2.41 meke_damping

```
real mom_meke::meke_cs::meke_damping
```

Local depth-independent MEKE dissipation rate [T-1 ~> s-1].

Definition at line 38 of file MOM_MEKE.F90.

```
00038   real :: meke_damping  !< Local depth-independent MEKE dissipation rate [T-1 ~> s-1].
```

43.11.2.42 meke_dtscale

```
real mom_meke::meke_cs::meke_dtscale
```

Scale factor to accelerate time-stepping [nondim].

Definition at line 62 of file MOM_MEKE.F90.

```
00062   real :: meke_dtscale  !< Scale factor to accelerate time-stepping [nondim]
```

43.11.2.43 meke_equilibrium_alt

```
logical mom_meke::meke_cs::meke_equilibrium_alt
```

If true, use an alternative calculation for the equilibrium value of MEKE.

Definition at line 50 of file MOM_MEKE.F90.

```
00050   logical :: meke_equilibrium_alt !< If true, use an alternative calculation for the
```

43.11.2.44 meke_equilibrium_restoring

```
logical mom_meke::meke_cs::meke_equilibrium_restoring
```

If true, restore MEKE back to its equilibrium value, which is calculated at each time step.

Definition at line 52 of file MOM_MEKE.F90.

```
00052   logical :: meke_equilibrium_restoring !< If true, restore MEKE back to its equilibrium value,
```

43.11.2.45 meke_frcoeff

```
real mom_meke::meke_cs::meke_frcoeff
```

Efficiency of conversion of ME into MEKE [nondim].

Definition at line 35 of file MOM_MEKE.F90.

```
00035   real :: meke_frcoeff  !< Efficiency of conversion of ME into MEKE [nondim]
```

43.11.2.46 meke_geometric

```
logical mom_meke::meke_cs::meke_geometric
```

If true, uses the GM coefficient formulation from the GEOMETRIC framework (Marshall et al., 2012)

Definition at line 46 of file MOM_MEKE.F90.

```
00046   logical :: meke_geometric !< If true, uses the GM coefficient formulation from the GEOMETRIC
```

43.11.2.47 meke_geometric_alpha

```
real mom_meke::meke_cs::meke_geometric_alpha
```

The nondimensional coefficient governing the efficiency of the GEOMETRIC thickness diffusion.

Definition at line 48 of file MOM_MEKE.F90.

```
00048   real    :: meke_geometric_alpha !< The nondimensional coefficient governing the efficiency of the
```

43.11.2.48 meke_gmcoeff

```
real mom_meke::meke_cs::meke_gmcoeff
```

Efficiency of conversion of PE into MEKE [nondim].

Definition at line 36 of file MOM_MEKE.F90.

```
00036   real :: meke_gmcoeff  !< Efficiency of conversion of PE into MEKE [nondim]
```

43.11.2.49 meke_gmecoeff

```
real mom_meke::meke_cs::meke_gmecoeff
```

Efficiency of conversion of MEKE into ME by GME [nondim].

Definition at line 37 of file MOM_MEKE.F90.

```
00037   real :: meke_gmecoeff !< Efficiency of conversion of MEKE into ME by GME [nondim]
```

43.11.2.50 meke_k4

```
real mom_meke::meke_cs::meke_k4
```

Background bi-harmonic diffusivity (of MEKE) [L4 T-1 ~> m4 s-1].

Definition at line 66 of file MOM_MEKE.F90.

```
00066   real :: meke_k4       !< Background bi-harmonic diffusivity (of MEKE) [L4 T-1 ~> m4 s-1]
```

43.11.2.51 meke_kh

```
real mom_meke::meke_cs::meke_kh
```

Background lateral diffusion of MEKE [L2 T-1 ~> m2 s-1].

Definition at line 65 of file MOM_MEKE.F90.

```
00065   real :: meke_kh       !< Background lateral diffusion of MEKE [L2 T-1 ~> m2 s-1]
```

43.11.2.52 meke_khcoeff

```
real mom_meke::meke_cs::meke_khcoeff
```

Scaling factor to convert MEKE into Kh [nondim].

Definition at line 63 of file MOM_MEKE.F90.

```
00063   real :: meke_khcoeff  !< Scaling factor to convert MEKE into Kh [nondim]
```

43.11.2.53 meke_min_gamma

```
real mom_meke::meke_cs::meke_min_gamma
```

Minimum value of gamma_b^2 allowed [nondim].

Definition at line 43 of file MOM_MEKE.F90.

```
00043   real :: meke_min_gamma!< Minimum value of gamma_b^2 allowed [nondim]
```

43.11.2.54 meke_restoring_rate

```
real mom_meke::meke_cs::meke_restoring_rate
```

Inverse of the timescale used to nudge MEKE toward its equilibrium value [s-1].

Definition at line 84 of file MOM_MEKE.F90.

```
00084   real :: meke_restoring_rate !< Inverse of the timescale used to nudge MEKE toward its equilibrium value
       [s-1].
```

43.11.2.55 meke_topographic_beta

```
real mom_meke::meke_cs::meke_topographic_beta
```

Weight for how much topographic beta is considered when computing beta in Rhines scale [nondim].

Definition at line 82 of file MOM_MEKE.F90.

```
00082   real :: meke_topographic_beta !< Weight for how much topographic beta is considered
```

43.11.2.56 meke_uscale

```
real mom_meke::meke_cs::meke_uscale
```

MEKE velocity scale for bottom drag [L T-1 ~> m s-1].

Definition at line 64 of file MOM_MEKE.F90.

```
00064   real :: meke_uscale   !< MEKE velocity scale for bottom drag [L T-1 ~> m s-1]
```

43.11.2.57 pass_kh

```
type(group_pass_type) mom_meke::meke_cs::pass_kh
```

Group halo pass handle for MEKEKh, MEKEKu, and/or MEKEAu.

Definition at line 104 of file MOM_MEKE.F90.

```
00104   type(group_pass_type) :: pass_kh   !< Group halo pass handle for MEKE%Kh, MEKE%Ku, and/or MEKE%Au
```

43.11.2.58 pass_meke

```
type(group_pass_type) mom_meke::meke_cs::pass_meke
```

Group halo pass handle for MEKEMEKE and maybe MEKEKh_diff.

Definition at line 103 of file MOM_MEKE.F90.

```
00103   type(group_pass_type) :: pass_meke !< Group halo pass handle for MEKE%MEKE and maybe MEKE%Kh_diff
```

43.11.2.59 rd_as_max_scale

```
logical mom_meke::meke_cs::rd_as_max_scale
```

If true the length scale can not exceed the first baroclinic deformation radius.

Definition at line 56 of file MOM_MEKE.F90.

```
00056   logical :: rd_as_max_scale !< If true the length scale can not exceed the
```

43.11.2.60 use_min_lscale

```
logical mom_meke::meke_cs::use_min_lscale
```

Use simple minimum for mixing length scale.

Definition at line 59 of file MOM_MEKE.F90.

```
00059   logical :: use_min_lscale !< Use simple minimum for mixing length scale.
```

43.11.2.61 use_old_lscale

```
logical mom_meke::meke_cs::use_old_lscale
```

Use the old formula for mixing length scale.

Definition at line 58 of file MOM_MEKE.F90.

```
00058   logical :: use_old_lscale !< Use the old formula for mixing length scale.
```

43.11.2.62 visc_drag

```
logical mom_meke::meke_cs::visc_drag
```

If true use the vertvisc_type to calculate bottom drag.

Definition at line 45 of file MOM_MEKE.F90.

```
00045   logical :: visc_drag  !< If true use the vertvisc_type to calculate bottom drag.
```

43.11.2.63 viscosity_coeff_au

```
real mom_meke::meke_cs::viscosity_coeff_au
```

The scaling coefficient in the expression for viscosity used to parameterize lateral biharmonic momentum mixing by unresolved eddies represented by MEKE.

Definition at line 72 of file MOM_MEKE.F90.

```
00072   real :: viscosity_coeff_au !< The scaling coefficient in the expression for
```

43.11.2.64 viscosity_coeff_ku

```
real mom_meke::meke_cs::viscosity_coeff_ku
```

The scaling coefficient in the expression for viscosity used to parameterize lateral harmonic momentum mixing by unresolved eddies represented by MEKE.

Definition at line 69 of file MOM_MEKE.F90.

```
00069   real :: viscosity_coeff_ku !< The scaling coefficient in the expression for
```

The documentation for this type was generated from the following file:

- /home/cermak/src/MOM6/src/parameterizations/lateral/MOM_MEKE.F90

43.12 mom::mom_control_struct Type Reference

Control structure for the MOM module, including the variables that describe the state of the ocean.

Public Attributes

- real, dimension(:, :, :), allocatable h

  layer thickness [H ~> m or kg m-2]
- real, dimension(:, :, :), allocatable t

  potential temperature [degC]
- real, dimension(:, :, :), allocatable s

  salinity [ppt]
- real, dimension(:, :, :), allocatable u

  zonal velocity component [L T-1 ~> m s-1]
- real, dimension(:, :, :), allocatable uh

  *uh = u * h * dy at u grid points [H L2 T-1 ~> m3 s-1 or kg s-1]*
- real, dimension(:, :, :), allocatable uhtr

  accumulated zonal thickness fluxes to advect tracers [H L2 ~> m3 or kg]
- real, dimension(:, :, :), allocatable v

  meridional velocity [L T-1 ~> m s-1]
- real, dimension(:, :, :), allocatable vh

  *vh = v * h * dx at v grid points [H L2 T-1 ~> m3 s-1 or kg s-1]*
- real, dimension(:, :, :), allocatable vhtr

  accumulated meridional thickness fluxes to advect tracers [H L2 ~> m3 or kg]
- real, dimension(:, :), allocatable ssh_rint

  A running time integral of the sea surface height [T m ~> s m].
- real, dimension(:, :), allocatable ave_ssh_ibc

  time-averaged (over a forcing time step) sea surface height with a correction for the inverse barometer [m]
- real, dimension(:, :), allocatable eta_av_bc

  free surface height or column mass time averaged over the last baroclinic dynamics time step [H ~> m or kg m-2]
- real, dimension(:,:), pointer hml => NULL()

  active mixed layer depth [Z ~> m]
- real time_in_cycle

  The running time of the current time-stepping cycle in calls that step the dynamics, and also the length of the time integral of ssh_rint [T ~> s].
- real time_in_thermo_cycle

  The running time of the current time-stepping cycle in calls that step the thermodynamics [T ~> s].
- type(ocean_grid_type) g_in

  Input grid metric.
- type(ocean_grid_type), pointer g => NULL()

  Model grid metric.
- logical rotate_index = .false.

  True if index map is rotated.
- type(verticalgrid_type), pointer gv => NULL()

  structure containing vertical grid info
- type(unit_scale_type), pointer us => NULL()

  structure containing various unit conversion factors
- type(thermo_var_ptrs) tv

  structure containing pointers to available thermodynamic fields
- real t_dyn_rel_adv

The time of the dynamics relative to tracer advection and lateral mixing [T ~> s], or equivalently the elapsed time since advectively updating the tracers. t_dyn_rel_adv is invariably positive and may span multiple coupling timesteps.

- real t_dyn_rel_thermo

  The time of the dynamics relative to diabatic processes and remapping [T ~> s]. t_dyn_rel_thermo can be negative or positive depending on whether the diabatic processes are applied before or after the dynamics and may span multiple coupling timesteps.

- real t_dyn_rel_diag

  The time of the diagnostics relative to diabatic processes and remapping [T ~> s]. t_dyn_rel_diag is always positive, since the diagnostics must lag.

- logical preadv_h_stored = .false.

  If true, the thicknesses from before the advective cycle have been stored for use in diagnostics.

- type(diag_ctrl) diag

  structure to regulate diagnostic output timing

- type(vertvisc_type) visc

  structure containing vertical viscosities, bottom drag viscosities, and related fields

- type(meke_type), pointer meke => NULL()

  structure containing fields related to the Mesoscale Eddy Kinetic Energy

- logical adiabatic

  If true, there are no diapycnal mass fluxes, and no calls to routines to calculate or apply diapycnal fluxes.

- logical diabatic_first

  If true, apply diabatic and thermodynamic processes before time stepping the dynamics.

- logical use_ale_algorithm

  If true, use the ALE algorithm rather than layered isopycnal/stacked shallow water mode. This logical is set by calling the function useRegridding() from the MOM_regridding module.

- logical offline_tracer_mode = .false.

  If true, step_offline() is called instead of step_MOM(). This is intended for running MOM6 in offline tracer mode.

- type(time_type), pointer time

  pointer to the ocean clock

- real dt

  (baroclinic) dynamics time step [T ~> s]

- real dt_therm

  thermodynamics time step [T ~> s]

- logical thermo_spans_coupling

  If true, thermodynamic and tracer time steps can span multiple coupled time steps.

- integer nstep_tot = 0

  The total number of dynamic timesteps tcaaken so far in this run segment.

- logical count_calls = .false.

  If true, count the calls to step_MOM, rather than the number of dynamics steps in nstep_tot.

- logical debug

  If true, write verbose checksums for debugging purposes.

- integer ntrunc

  number u,v truncations since last call to write_energy

- integer cont_stencil

  The stencil for thickness from the continuity solver.

- logical do_dynamics

  If false, does not call step_MOM_dyn_∗. This is an undocumented run-time flag that is fragile.

- logical split

  If true, use the split time stepping scheme.

- logical use_rk2

  If true, use RK2 instead of RK3 in unsplit mode (i.e., no split between barotropic and baroclinic).

- logical thickness_diffuse

  If true, diffuse interface height w/ a diffusivity KHTH.

- logical thickness_diffuse_first

  If true, diffuse thickness before dynamics.
- logical mixedlayer_restrat

  If true, use submesoscale mixed layer restratifying scheme.
- logical usemeke

  If true, call the MEKE parameterization.
- logical usewaves

  If true, update Stokes drift.
- logical use_p_surf_in_eos

  If true, always include the surface pressure contributions in equation of state calculations.
- real dtbt_reset_period

  The time interval between dynamic recalculation of the barotropic time step [s]. If this is negative dtbt is never calculated, and if it is 0, dtbt is calculated every step.
- type(time_type) dtbt_reset_interval

  A time_time representation of dtbt_reset_period.
- type(time_type) dtbt_reset_time

  The next time DTBT should be calculated.
- real, dimension(:,:,:), pointer h_pre_dyn => NULL()

  The thickness before the transports [H ~> m or kg m-2].
- real, dimension(:,:,:), pointer t_pre_dyn => NULL()

  Temperature before the transports [degC].
- real, dimension(:,:,:), pointer s_pre_dyn => NULL()

  Salinity before the transports [ppt].
- type(accel_diag_ptrs) adp

  structure containing pointers to accelerations, for derived diagnostics (e.g., energy budgets)
- type(cont_diag_ptrs) cdp

  structure containing pointers to continuity equation terms, for derived diagnostics (e.g., energy budgets)
- real, dimension(:,:,:), pointer u_prev => NULL()

  previous value of u stored for diagnostics [L T-1 ~> m s-1]
- real, dimension(:,:,:), pointer v_prev => NULL()

  previous value of v stored for diagnostics [L T-1 ~> m s-1]
- logical interp_p_surf

  If true, linearly interpolate surface pressure over the coupling time step, using specified value at the end of the coupling step. False by default.
- logical p_surf_prev_set

  If true, p_surf_prev has been properly set from a previous time-step or the ocean restart file. This is only valid when interp_p_surf is true.
- real, dimension(:,:), pointer p_surf_prev => NULL()

  surface pressure [R L2 T-2 ~> Pa] at end previous call to step_MOM
- real, dimension(:,:), pointer p_surf_begin => NULL()

  surface pressure [R L2 T-2 ~> Pa] at start of step_MOM_dyn_...
- real, dimension(:,:), pointer p_surf_end => NULL()

  surface pressure [R L2 T-2 ~> Pa] at end of step_MOM_dyn_...
- logical write_ic

  If true, then the initial conditions will be written to file.
- character(len=120) ic_file

  A file into which the initial conditions are written in a new run if SAVE_INITIAL_CONDS is true.
- logical calc_rho_for_sea_lev

  If true, calculate rho to convert pressure to sea level.
- real hmix

Diagnostic mixed layer thickness over which to average surface tracer properties when a bulk mixed layer is not used [Z ~> m], or a negative value if a bulk mixed layer is being used.

- real hfrz

  If HFrz > 0, the nominal depth over which melt potential is computed [Z ~> m]. The actual depth over which melt potential is computed is min(HFrz, OBLD), where OBLD is the boundary layer depth. If HFrz <= 0 (default), melt potential will not be computed.

- real hmix_uv

  Depth scale over which to average surface flow to feedback to the coupler/driver [Z ~> m] when bulk mixed layer is not used, or a negative value if a bulk mixed layer is being used.

- logical check_bad_sfc_vals

  If true, scan surface state for ridiculous values.

- real bad_val_ssh_max

  Maximum SSH before triggering bad value message [Z ~> m].

- real bad_val_sst_max

  Maximum SST before triggering bad value message [degC].

- real bad_val_sst_min

  Minimum SST before triggering bad value message [degC].

- real bad_val_sss_max

  Maximum SSS before triggering bad value message [ppt].

- real bad_val_col_thick

  Minimum column thickness before triggering bad value message [Z ~> m].

- logical answers_2018

  If true, use expressions for the surface properties that recover the answers from the end of 2018. Otherwise, use more appropriate expressions that differ at roundoff for non-Boussinsq cases.

- type(mom_diag_ids) ids

  Handles used for diagnostics.

- type(transport_diag_ids) transport_ids

  Handles used for transport diagnostics.

- type(surface_diag_ids) sfc_ids

  Handles used for surface diagnostics.

- type(diag_grid_storage) diag_pre_sync

  The grid (thicknesses) before remapping.

- type(diag_grid_storage) diag_pre_dyn

  The grid (thicknesses) before dynamics.

- type(mom_dyn_unsplit_cs), pointer dyn_unsplit_csp => NULL()

  Pointer to the control structure used for the unsplit dynamics.

- type(mom_dyn_unsplit_rk2_cs), pointer dyn_unsplit_rk2_csp => NULL()

  Pointer to the control structure used for the unsplit RK2 dynamics.

- type(mom_dyn_split_rk2_cs), pointer dyn_split_rk2_csp => NULL()

  Pointer to the control structure used for the mode-split RK2 dynamics.

- type(thickness_diffuse_cs), pointer thickness_diffuse_csp => NULL()

  Pointer to the control structure used for the isopycnal height diffusive transport. This is also common referred to as Gent-McWilliams diffusion.

- type(mixedlayer_restrat_cs), pointer mixedlayer_restrat_csp => NULL()

  Pointer to the control structure used for the mixed layer restratification.

- type(set_visc_cs), pointer set_visc_csp => NULL()

  Pointer to the control structure used to set viscosities.

- type(diabatic_cs), pointer diabatic_csp => NULL()

  Pointer to the control structure for the diabatic driver.

- type(meke_cs), pointer meke_csp => NULL()

  Pointer to the control structure for the MEKE updates.

- type(varmix_cs), pointer varmix => NULL()

Pointer to the control structure for the variable mixing module.

- type(barotropic_cs), pointer barotropic_csp => NULL()

  Pointer to the control structure for the barotropic module.
- type(tracer_registry_type), pointer tracer_reg => NULL()

  Pointer to the MOM tracer registry.
- type(tracer_advect_cs), pointer tracer_adv_csp => NULL()

  Pointer to the MOM tracer advection control structure.
- type(tracer_hor_diff_cs), pointer tracer_diff_csp => NULL()

  Pointer to the MOM along-isopycnal tracer diffusion control structure.
- type(tracer_flow_control_cs), pointer tracer_flow_csp => NULL()

  Pointer to the control structure that orchestrates the calling of tracer packages.
- type(update_obc_cs), pointer update_obc_csp => NULL()

  Pointer to the control structure for updating open boundary condition properties.
- type(ocean_obc_type), pointer obc => NULL()

  Pointer to the MOM open boundary condition type.
- type(sponge_cs), pointer sponge_csp => NULL()

  Pointer to the layered-mode sponge control structure.
- type(ale_sponge_cs), pointer ale_sponge_csp => NULL()

  Pointer to the ALE-mode sponge control structure.
- type(ale_cs), pointer ale_csp => NULL()

  Pointer to the Arbitrary Lagrangian Eulerian (ALE) vertical coordinate control structure.
- type(sum_output_cs), pointer sum_output_csp => NULL()

  Pointer to the globally summed output control structure.
- type(diagnostics_cs), pointer diagnostics_csp => NULL()

  Pointer to the MOM diagnostics control structure.
- type(offline_transport_cs), pointer offline_csp => NULL()

  Pointer to the offline tracer transport control structure.
- logical ensemble_ocean

  if true, this run is part of a larger ensemble for the purpose of data assimilation or statistical analysis.
- type(oda_cs), pointer odacs => NULL()

  a pointer to the control structure for handling ensemble model state vectors and data assimilation increments and priors

43.12.1 Detailed Description

Control structure for the MOM module, including the variables that describe the state of the ocean.

Definition at line 165 of file MOM.F90.

43.12.2 Member Data Documentation

43.12.2.1 adiabatic

```
logical mom::mom_control_struct::adiabatic
```

If true, there are no diapycnal mass fluxes, and no calls to routines to calculate or apply diapycnal fluxes.

Definition at line 220 of file MOM.F90.

```
00220   logical :: adiabatic !< If true, there are no diapycnal mass fluxes, and no calls
```

43.12.2.2 adp

```
type(accel_diag_ptrs) mom::mom_control_struct::adp
```

structure containing pointers to accelerations, for derived diagnostics (e.g., energy budgets)

Definition at line 268 of file MOM.F90.

```
00268   type(accel_diag_ptrs) :: adp  !< structure containing pointers to accelerations,
```

43.12.2.3 ale_csp

```
type(ale_cs), pointer mom::mom_control_struct::ale_csp => NULL()
```

Pointer to the Arbitrary Lagrangian Eulerian (ALE) vertical coordinate control structure.

Definition at line 364 of file MOM.F90.

```
00364   type(ale_cs),                 pointer :: ale_csp => null()
```

43.12.2.4 ale_sponge_csp

```
type(ale_sponge_cs), pointer mom::mom_control_struct::ale_sponge_csp => NULL()
```

Pointer to the ALE-mode sponge control structure.

Definition at line 362 of file MOM.F90.

```
00362   type(ale_sponge_cs),          pointer :: ale_sponge_csp => null()
```

43.12.2.5 answers_2018

```
logical mom::mom_control_struct::answers_2018
```

If true, use expressions for the surface properties that recover the answers from the end of 2018. Otherwise, use more appropriate expressions that differ at roundoff for non-Boussinsq cases.

Definition at line 313 of file MOM.F90.

```
00313   logical :: answers_2018       !< If true, use expressions for the surface properties that recover
```

43.12.2.6 ave_ssh_ibc

```
real, dimension( :  , :  ), allocatable mom::mom_control_struct::ave_ssh_ibc
```

time-averaged (over a forcing time step) sea surface height with a correction for the inverse barometer [m]

Definition at line 180 of file MOM.F90.

```
00180   real allocable_, dimension(NIMEM_,NJMEM_) :: ave_ssh_ibc
```

43.12.2.7 bad_val_col_thick

```
real mom::mom_control_struct::bad_val_col_thick
```

Minimum column thickness before triggering bad value message [Z ~> m].

Definition at line 312 of file MOM.F90.

```
00312   real    :: bad_val_col_thick  !< Minimum column thickness before triggering bad value message [Z ~> m]
```

43.12.2.8 bad_val_ssh_max

```
real mom::mom_control_struct::bad_val_ssh_max
```

Maximum SSH before triggering bad value message [Z ~> m].

Definition at line 308 of file MOM.F90.

```
00308   real    :: bad_val_ssh_max    !< Maximum SSH before triggering bad value message [Z ~> m]
```

43.12.2.9 bad_val_sss_max

```
real mom::mom_control_struct::bad_val_sss_max
```

Maximum SSS before triggering bad value message [ppt].

Definition at line 311 of file MOM.F90.

```
00311   real    :: bad_val_sss_max    !< Maximum SSS before triggering bad value message [ppt]
```

43.12.2.10 bad_val_sst_max

```
real mom::mom_control_struct::bad_val_sst_max
```

Maximum SST before triggering bad value message [degC].

Definition at line 309 of file MOM.F90.

```
00309   real    :: bad_val_sst_max    !< Maximum SST before triggering bad value message [degC]
```

43.12.2.11 bad_val_sst_min

```
real mom::mom_control_struct::bad_val_sst_min
```

Minimum SST before triggering bad value message [degC].

Definition at line 310 of file MOM.F90.

```
00310   real    :: bad_val_sst_min    !< Minimum SST before triggering bad value message [degC]
```

43.12.2.12 barotropic_csp

```
type(barotropic_cs), pointer mom::mom_control_struct::barotropic_csp => NULL()
```

Pointer to the control structure for the barotropic module.

Definition at line 344 of file MOM.F90.

```
00344   type(barotropic_cs),          pointer :: barotropic_csp => null()
```

43.12.2.13 calc_rho_for_sea_lev

```
logical mom::mom_control_struct::calc_rho_for_sea_lev
```

If true, calculate rho to convert pressure to sea level.

Definition at line 292 of file MOM.F90.

```
00292   logical :: calc_rho_for_sea_lev !< If true, calculate rho to convert pressure to sea level
```

43.12.2.14 cdp

```
type(cont_diag_ptrs) mom::mom_control_struct::cdp
```

structure containing pointers to continuity equation terms, for derived diagnostics (e.g., energy budgets)

Definition at line 270 of file MOM.F90.

```
00270   type(cont_diag_ptrs)  :: cdp  !< structure containing pointers to continuity equation
```

43.12.2.15 check_bad_sfc_vals

```
logical mom::mom_control_struct::check_bad_sfc_vals
```

If true, scan surface state for ridiculous values.

Definition at line 307 of file MOM.F90.

```
00307   logical :: check_bad_sfc_vals !< If true, scan surface state for ridiculous values.
```

43.12.2.16 cont_stencil

```
integer mom::mom_control_struct::cont_stencil
```

The stencil for thickness from the continuity solver.

Definition at line 243 of file MOM.F90.

```
00243   integer :: cont_stencil           !< The stencil for thickness from the continuity solver.
```

43.12.2.17 count_calls

```
logical mom::mom_control_struct::count_calls = .false.
```

If true, count the calls to step_MOM, rather than the number of dynamics steps in nstep_tot.

Definition at line 238 of file MOM.F90.

```
00238   logical :: count_calls = .false.   !< If true, count the calls to step_MOM, rather than the
```

43.12.2.18 debug

```
logical mom::mom_control_struct::debug
```

If true, write verbose checksums for debugging purposes.

Definition at line 240 of file MOM.F90.

```
00240   logical :: debug                  !< If true, write verbose checksums for debugging purposes.
```

43.12.2.19 diabatic_csp

```
type(diabatic_cs), pointer mom::mom_control_struct::diabatic_csp => NULL()
```

Pointer to the control structure for the diabatic driver.

Definition at line 338 of file MOM.F90.

```
00338   type(diabatic_cs),            pointer :: diabatic_csp => null()
```

43.12.2.20 diabatic_first

```
logical mom::mom_control_struct::diabatic_first
```

If true, apply diabatic and thermodynamic processes before time stepping the dynamics.

Definition at line 222 of file MOM.F90.

```
00222   logical :: diabatic_first !< If true, apply diabatic and thermodynamic processes before time
```

43.12.2.21 diag

```
type(diag_ctrl) mom::mom_control_struct::diag
```

structure to regulate diagnostic output timing

Definition at line 215 of file MOM.F90.

```
00215   type(diag_ctrl)     :: diag !< structure to regulate diagnostic output timing
```

43.12.2.22 diag_pre_dyn

```
type(diag_grid_storage) mom::mom_control_struct::diag_pre_dyn
```

The grid (thicknesses) before dynamics.

Definition at line 321 of file MOM.F90.

```
00321   type(diag_grid_storage)  :: diag_pre_dyn  !< The grid (thicknesses) before dynamics
```

43.12.2.23 diag_pre_sync

```
type(diag_grid_storage) mom::mom_control_struct::diag_pre_sync
```

The grid (thicknesses) before remapping.

Definition at line 320 of file MOM.F90.

```
00320   type(diag_grid_storage)  :: diag_pre_sync !< The grid (thicknesses) before remapping
```

43.12.2.24 diagnostics_csp

```
type(diagnostics_cs), pointer mom::mom_control_struct::diagnostics_csp => NULL()
```

Pointer to the MOM diagnostics control structure.

Definition at line 370 of file MOM.F90.

```
00370   type(diagnostics_cs),          pointer :: diagnostics_csp => null()
```

43.12.2.25 do_dynamics

```
logical mom::mom_control_struct::do_dynamics
```

If false, does not call step_MOM_dyn_*. This is an undocumented run-time flag that is fragile.

Definition at line 245 of file MOM.F90.

```
00245   logical :: do_dynamics            !< If false, does not call step_MOM_dyn_*. This is an
```

43.12.2.26 dt

```
real mom::mom_control_struct::dt
```

(baroclinic) dynamics time step [T ~> s]

Definition at line 232 of file MOM.F90.

```
00232   real    :: dt                    !< (baroclinic) dynamics time step [T ~> s]
```

43.12.2.27 dt_therm

```
real mom::mom_control_struct::dt_therm
```

thermodynamics time step [T ~> s]

Definition at line 233 of file MOM.F90.

```
00233   real    :: dt_therm              !< thermodynamics time step [T ~> s]
```

43.12.2.28 dtbt_reset_interval

```
type(time_type) mom::mom_control_struct::dtbt_reset_interval
```

A time_time representation of dtbt_reset_period.

Definition at line 260 of file MOM.F90.

```
00260   type(time_type) :: dtbt_reset_interval !< A time_time representation of dtbt_reset_period.
```

43.12.2.29 dtbt_reset_period

```
real mom::mom_control_struct::dtbt_reset_period
```

The time interval between dynamic recalculation of the barotropic time step [s]. If this is negative dtbt is never calculated, and if it is 0, dtbt is calculated every step.

Definition at line 257 of file MOM.F90.

```
00257   real :: dtbt_reset_period        !< The time interval between dynamic recalculation of the
```

43.12.2.30 dtbt_reset_time

```
type(time_type) mom::mom_control_struct::dtbt_reset_time
```

The next time DTBT should be calculated.

Definition at line 261 of file MOM.F90.

```
00261   type(time_type) :: dtbt_reset_time !< The next time DTBT should be calculated.
```

43.12.2.31 dyn_split_rk2_csp

```
type(mom_dyn_split_rk2_cs), pointer mom::mom_control_struct::dyn_split_rk2_csp => NULL()
```

Pointer to the control structure used for the mode-split RK2 dynamics.

Definition at line 329 of file MOM.F90.

```
00329   type(mom_dyn_split_rk2_cs),    pointer :: dyn_split_rk2_csp => null()
```

43.12.2.32 dyn_unsplit_csp

```
type(mom_dyn_unsplit_cs), pointer mom::mom_control_struct::dyn_unsplit_csp => NULL()
```

Pointer to the control structure used for the unsplit dynamics.

Definition at line 325 of file MOM.F90.

```
00325   type(mom_dyn_unsplit_cs),      pointer :: dyn_unsplit_csp => null()
```

43.12.2.33 dyn_unsplit_rk2_csp

```
type(mom_dyn_unsplit_rk2_cs), pointer mom::mom_control_struct::dyn_unsplit_rk2_csp => NULL()
```

Pointer to the control structure used for the unsplit RK2 dynamics.

Definition at line 327 of file MOM.F90.

```
00327   type(mom_dyn_unsplit_rk2_cs),  pointer :: dyn_unsplit_rk2_csp => null()
```

43.12.2.34 ensemble_ocean

```
logical mom::mom_control_struct::ensemble_ocean
```

if true, this run is part of a larger ensemble for the purpose of data assimilation or statistical analysis.

Definition at line 375 of file MOM.F90.

```
00375   logical              :: ensemble_ocean !< if true, this run is part of a
```

43.12.2.35 eta_av_bc

```
real, dimension( :  , :  ), allocatable mom::mom_control_struct::eta_av_bc
```

free surface height or column mass time averaged over the last baroclinic dynamics time step [H ~> m or kg m-2]

Definition at line 183 of file MOM.F90.

```
00183   real allocable_, dimension(NIMEM_,NJMEM_) :: eta_av_bc
```

43.12.2.36 g

```
type(ocean_grid_type), pointer mom::mom_control_struct::g => NULL()
```

Model grid metric.

Definition at line 195 of file MOM.F90.

```
00195   type(ocean_grid_type), pointer :: g => null()   !< Model grid metric
```

43.12.2.37 g_in

```
type(ocean_grid_type) mom::mom_control_struct::g_in
```

Input grid metric.

Definition at line 194 of file MOM.F90.

```
00194   type(ocean_grid_type) :: g_in                   !< Input grid metric
```

43.12.2.38 gv

```
type(verticalgrid_type), pointer mom::mom_control_struct::gv => NULL()
```

structure containing vertical grid info

Definition at line 198 of file MOM.F90.

```
00198   type(verticalgrid_type), pointer :: &
00199     gv => null()    !< structure containing vertical grid info
```

43.12.2.39 h

```
real, dimension( :  , :  , :  ), allocatable mom::mom_control_struct::h
```

layer thickness [H ~> m or kg m-2]

Definition at line 166 of file MOM.F90.

```
00166   real allocable_, dimension(NIMEM_,NJMEM_,NKMEM_) :: &
00167     h, &            !< layer thickness [H ~> m or kg m-2]
00168     t, &            !< potential temperature [degC]
00169     s               !< salinity [ppt]
```

43.12.2.40 h_pre_dyn

```
real, dimension(:,:,:), pointer mom::mom_control_struct::h_pre_dyn => NULL()
```

The thickness before the transports [H ~> m or kg m-2].

Definition at line 264 of file MOM.F90.

```
00264   real, dimension(:,:,:), pointer :: &
00265     h_pre_dyn => null(), &      !< The thickness before the transports [H ~> m or kg m-2].
00266     t_pre_dyn => null(), &      !< Temperature before the transports [degC].
00267     s_pre_dyn => null()         !< Salinity before the transports [ppt].
```

43.12.2.41 hfrz

```
real mom::mom_control_struct::hfrz
```

If HFrz > 0, the nominal depth over which melt potential is computed [Z ~> m]. The actual depth over which melt potential is computed is min(HFrz, OBLD), where OBLD is the boundary layer depth. If HFrz <= 0 (default), melt potential will not be computed.

Definition at line 299 of file MOM.F90.

```
00299   real :: hfrz                !< If HFrz > 0, the nominal depth over which melt potential is
```

43.12.2.42 hmix

```
real mom::mom_control_struct::hmix
```

Diagnostic mixed layer thickness over which to average surface tracer properties when a bulk mixed layer is not used [Z ~> m], or a negative value if a bulk mixed layer is being used.

Definition at line 295 of file MOM.F90.

```
00295   real :: hmix                !< Diagnostic mixed layer thickness over which to
```

43.12.2.43 hmix_uv

```
real mom::mom_control_struct::hmix_uv
```

Depth scale over which to average surface flow to feedback to the coupler/driver [Z ~> m] when bulk mixed layer is not used, or a negative value if a bulk mixed layer is being used.

Definition at line 303 of file MOM.F90.

```
00303   real :: hmix_uv             !< Depth scale over which to average surface flow to
```

43.12.2.44 hml

```
real, dimension(:,:), pointer mom::mom_control_struct::hml => NULL()
```

active mixed layer depth [Z ~> m]

Definition at line 186 of file MOM.F90.

```
00186   real, dimension(:,:), pointer :: &
00187     hml => null()   !< active mixed layer depth [Z ~> m]
```

43.12.2.45 ic_file

```
character(len=120) mom::mom_control_struct::ic_file
```

A file into which the initial conditions are written in a new run if SAVE_INITIAL_CONDS is true.

Definition at line 289 of file MOM.F90.

```
00289   character(len=120) :: ic_file !< A file into which the initial conditions are
```

43.12.2.46 ids

```
type(mom_diag_ids) mom::mom_control_struct::ids
```

Handles used for diagnostics.

Definition at line 317 of file MOM.F90.

```
00317   type(mom_diag_ids)       :: ids      !<  Handles used for diagnostics.
```

43.12.2.47 interp_p_surf

```
logical mom::mom_control_struct::interp_p_surf
```

If true, linearly interpolate surface pressure over the coupling time step, using specified value at the end of the coupling step. False by default.

Definition at line 276 of file MOM.F90.

```
00276   logical :: interp_p_surf      !< If true, linearly interpolate surface pressure
```

43.12.2.48 meke

```
type(meke_type), pointer mom::mom_control_struct::meke => NULL()
```

structure containing fields related to the Mesoscale Eddy Kinetic Energy

Definition at line 218 of file MOM.F90.

```
00218   type(meke_type), pointer :: meke => null() !<  structure containing fields
```

43.12.2.49 meke_csp

```
type(meke_cs), pointer mom::mom_control_struct::meke_csp => NULL()
```

Pointer to the control structure for the MEKE updates.

Definition at line 340 of file MOM.F90.

```
00340   type(meke_cs),                pointer :: meke_csp => null()
```

43.12.2.50 mixedlayer_restrat

```
logical mom::mom_control_struct::mixedlayer_restrat
```

If true, use submesoscale mixed layer restratifying scheme.

Definition at line 252 of file MOM.F90.

```
00252   logical :: mixedlayer_restrat      !< If true, use submesoscale mixed layer restratifying scheme.
```

43.12.2.51 mixedlayer_restrat_csp

```
type(mixedlayer_restrat_cs), pointer mom::mom_control_struct::mixedlayer_restrat_csp => NULL()
```

Pointer to the control structure used for the mixed layer restratification.

Definition at line 334 of file MOM.F90.

```
00334   type(mixedlayer_restrat_cs),   pointer :: mixedlayer_restrat_csp => null()
```

43.12.2.52 nstep_tot

```
integer mom::mom_control_struct::nstep_tot = 0
```

The total number of dynamic timesteps tcaaken so far in this run segment.

Definition at line 236 of file MOM.F90.

```
00236   integer :: nstep_tot = 0           !< The total number of dynamic timesteps tcaaken
```

43.12.2.53 ntrunc

```
integer mom::mom_control_struct::ntrunc
```

number u,v truncations since last call to write_energy

Definition at line 241 of file MOM.F90.

```
00241   integer :: ntrunc                  !< number u,v truncations since last call to write_energy
```

43.12.2.54 obc

```
type(ocean_obc_type), pointer mom::mom_control_struct::obc => NULL()
```

Pointer to the MOM open boundary condition type.

Definition at line 358 of file MOM.F90.

```
00358   type(ocean_obc_type),          pointer :: obc => null()
```

43.12.2.55 odacs

```
type(oda_cs), pointer mom::mom_control_struct::odacs => NULL()
```

a pointer to the control structure for handling ensemble model state vectors and data assimilation increments and priors

Definition at line 378 of file MOM.F90.

```
00378   type(oda_cs), pointer :: odacs => null() !< a pointer to the control structure for handling
```

43.12.2.56 offline_csp

```
type(offline_transport_cs), pointer mom::mom_control_struct::offline_csp => NULL()
```

Pointer to the offline tracer transport control structure.

Definition at line 372 of file MOM.F90.

```
00372   type(offline_transport_cs),    pointer :: offline_csp => null()
```

43.12.2.57 offline_tracer_mode

```
logical mom::mom_control_struct::offline_tracer_mode = .false.
```

If true, step_offline() is called instead of step_MOM(). This is intended for running MOM6 in offline tracer mode.

Definition at line 227 of file MOM.F90.

```
00227   logical :: offline_tracer_mode = .false.
```

43.12.2.58 p_surf_begin

```
real, dimension(:,:), pointer mom::mom_control_struct::p_surf_begin => NULL()
```

surface pressure [R L2 T-2 ~> Pa] at start of step_MOM_dyn_...

Definition at line 282 of file MOM.F90.

43.12.2.59 p_surf_end

```
real, dimension(:,:), pointer mom::mom_control_struct::p_surf_end => NULL()
```

surface pressure [R L2 T-2 ~> Pa] at end of step_MOM_dyn_...

Definition at line 282 of file MOM.F90.

43.12.2.60 p_surf_prev

```
real, dimension(:,:), pointer mom::mom_control_struct::p_surf_prev => NULL()
```

surface pressure [R L2 T-2 ~> Pa] at end previous call to step_MOM

Definition at line 282 of file MOM.F90.

```
00282   real, dimension(:,:), pointer :: &
00283     p_surf_prev  => null(), &   !< surface pressure [R L2 T-2 ~> Pa] at end  previous call to step_MOM
00284     p_surf_begin => null(), &   !< surface pressure [R L2 T-2 ~> Pa] at start of step_MOM_dyn_...
00285     p_surf_end   => null()      !< surface pressure [R L2 T-2 ~> Pa] at end   of step_MOM_dyn_...
```

43.12.2.61 p_surf_prev_set

```
logical mom::mom_control_struct::p_surf_prev_set
```

If true, p_surf_prev has been properly set from a previous time-step or the ocean restart file. This is only valid when interp_p_surf is true.

Definition at line 279 of file MOM.F90.

```
00279   logical :: p_surf_prev_set    !< If true, p_surf_prev has been properly set from
```

43.12.2.62 preadv_h_stored

```
logical mom::mom_control_struct::preadv_h_stored = .false.
```

If true, the thicknesses from before the advective cycle have been stored for use in diagnostics.

Definition at line 212 of file MOM.F90.

```
00212   logical :: preadv_h_stored = .false. !< If true, the thicknesses from before the advective cycle
```

43.12.2.63 rotate_index

```
logical mom::mom_control_struct::rotate_index = .false.
```

True if index map is rotated.

Definition at line 196 of file MOM.F90.

```
00196   logical :: rotate_index = .false.   !< True if index map is rotated
```

43.12.2.64 s

```
real, dimension( :  , :  , :  ), allocatable mom::mom_control_struct::s
```

salinity [ppt]

Definition at line 166 of file MOM.F90.

43.12.2.65 s_pre_dyn

```
real, dimension(:,:,:), pointer mom::mom_control_struct::s_pre_dyn => NULL()
```

Salinity before the transports [ppt].

Definition at line 264 of file MOM.F90.

43.12.2.66 set_visc_csp

```
type(set_visc_cs), pointer mom::mom_control_struct::set_visc_csp => NULL()
```

Pointer to the control structure used to set viscosities.

Definition at line 336 of file MOM.F90.

```
00336   type(set_visc_cs),             pointer :: set_visc_csp => null()
```

43.12.2.67 sfc_ids

```
type(surface_diag_ids) mom::mom_control_struct::sfc_ids
```

Handles used for surface diagnostics.

Definition at line 319 of file MOM.F90.

```
00319   type(surface_diag_ids)   :: sfc_ids  !< Handles used for surface diagnostics.
```

43.12.2.68 split

```
logical mom::mom_control_struct::split
```

If true, use the split time stepping scheme.

Definition at line 247 of file MOM.F90.

```
00247   logical :: split                 !< If true, use the split time stepping scheme.
```

43.12.2.69 sponge_csp

```
type(sponge_cs), pointer mom::mom_control_struct::sponge_csp => NULL()
```

Pointer to the layered-mode sponge control structure.

Definition at line 360 of file MOM.F90.

```
00360   type(sponge_cs),               pointer :: sponge_csp => null()
```

43.12.2.70 ssh_rint

```
real, dimension( :  , :  ), allocatable mom::mom_control_struct::ssh_rint
```

A running time integral of the sea surface height [T m ~> s m].

Definition at line 178 of file MOM.F90.

```
00178   real allocable_, dimension(NIMEM_,NJMEM_) :: ssh_rint
```

43.12.2.71 sum_output_csp

```
type(sum_output_cs), pointer mom::mom_control_struct::sum_output_csp => NULL()
```

Pointer to the globally summed output control structure.

Definition at line 368 of file MOM.F90.

```
00368   type(sum_output_cs),           pointer :: sum_output_csp => null()
```

43.12.2.72 t

```
real, dimension( :  , :  , :  ), allocatable mom::mom_control_struct::t
```

potential temperature [degC]

Definition at line 166 of file MOM.F90.

43.12.2.73 t_dyn_rel_adv

```
real mom::mom_control_struct::t_dyn_rel_adv
```

The time of the dynamics relative to tracer advection and lateral mixing [T ~> s], or equivalently the elapsed time since advectively updating the tracers. t_dyn_rel_adv is invariably positive and may span multiple coupling timesteps.

Definition at line 203 of file MOM.F90.

```
00203   real :: t_dyn_rel_adv !< The time of the dynamics relative to tracer advection and lateral mixing
```

43.12.2.74 t_dyn_rel_diag

```
real mom::mom_control_struct::t_dyn_rel_diag
```

The time of the diagnostics relative to diabatic processes and remapping [T ~> s]. t_dyn_rel_diag is always positive, since the diagnostics must lag.

Definition at line 210 of file MOM.F90.

```
00210   real :: t_dyn_rel_diag !< The time of the diagnostics relative to diabatic processes and remapping
```

43.12.2.75 t_dyn_rel_thermo

```
real mom::mom_control_struct::t_dyn_rel_thermo
```

The time of the dynamics relative to diabatic processes and remapping [T ~> s]. t_dyn_rel_thermo can be negative or positive depending on whether the diabatic processes are applied before or after the dynamics and may span multiple coupling timesteps.

Definition at line 206 of file MOM.F90.

```
00206   real :: t_dyn_rel_thermo  !< The time of the dynamics relative to diabatic  processes and remapping
```

43.12.2.76 t_pre_dyn

```
real, dimension(:,:,:), pointer mom::mom_control_struct::t_pre_dyn => NULL()
```

Temperature before the transports [degC].

Definition at line 264 of file MOM.F90.

43.12.2.77 thermo_spans_coupling

```
logical mom::mom_control_struct::thermo_spans_coupling
```

If true, thermodynamic and tracer time steps can span multiple coupled time steps.

Definition at line 234 of file MOM.F90.

```
00234   logical :: thermo_spans_coupling   !< If true, thermodynamic and tracer time
```

43.12.2.78 thickness_diffuse

```
logical mom::mom_control_struct::thickness_diffuse
```

If true, diffuse interface height w/ a diffusivity KHTH.

Definition at line 250 of file MOM.F90.

```
00250   logical :: thickness_diffuse       !< If true, diffuse interface height w/ a diffusivity KHTH.
```

43.12.2.79 thickness_diffuse_csp

```
type(thickness_diffuse_cs), pointer mom::mom_control_struct::thickness_diffuse_csp => NULL()
```

Pointer to the control structure used for the isopycnal height diffusive transport. This is also common referred to as Gent-McWilliams diffusion.

Definition at line 331 of file MOM.F90.

```
00331   type(thickness_diffuse_cs),    pointer :: thickness_diffuse_csp => null()
```

43.12.2.80 thickness_diffuse_first

```
logical mom::mom_control_struct::thickness_diffuse_first
```

If true, diffuse thickness before dynamics.

Definition at line 251 of file MOM.F90.

```
00251   logical :: thickness_diffuse_first !< If true, diffuse thickness before dynamics.
```

43.12.2.81 time

```
type(time_type), pointer mom::mom_control_struct::time
```

pointer to the ocean clock

Definition at line 231 of file MOM.F90.

```
00231   type(time_type), pointer :: time   !< pointer to the ocean clock
```

43.12.2.82 time_in_cycle

```
real mom::mom_control_struct::time_in_cycle
```

The running time of the current time-stepping cycle in calls that step the dynamics, and also the length of the time integral of ssh_rint [T ~> s].

Definition at line 188 of file MOM.F90.

```
00188   real :: time_in_cycle !< The running time of the current time-stepping cycle
```

43.12.2.83 time_in_thermo_cycle

```
real mom::mom_control_struct::time_in_thermo_cycle
```

The running time of the current time-stepping cycle in calls that step the thermodynamics [T ~> s].

Definition at line 191 of file MOM.F90.

```
00191   real :: time_in_thermo_cycle !< The running time of the current time-stepping
```

43.12.2.84 tracer_adv_csp

```
type(tracer_advect_cs), pointer mom::mom_control_struct::tracer_adv_csp => NULL()
```

Pointer to the MOM tracer advection control structure.

Definition at line 348 of file MOM.F90.

```
00348   type(tracer_advect_cs),        pointer :: tracer_adv_csp => null()
```

43.12.2.85 tracer_diff_csp

```
type(tracer_hor_diff_cs), pointer mom::mom_control_struct::tracer_diff_csp => NULL()
```

Pointer to the MOM along-isopycnal tracer diffusion control structure.

Definition at line 350 of file MOM.F90.

```
00350   type(tracer_hor_diff_cs),      pointer :: tracer_diff_csp => null()
```

43.12.2.86 tracer_flow_csp

```
type(tracer_flow_control_cs), pointer mom::mom_control_struct::tracer_flow_csp => NULL()
```

Pointer to the control structure that orchestrates the calling of tracer packages.

Definition at line 352 of file MOM.F90.

```
00352   type(tracer_flow_control_cs),  pointer :: tracer_flow_csp => null()
```

43.12.2.87 tracer_reg

```
type(tracer_registry_type), pointer mom::mom_control_struct::tracer_reg => NULL()
```

Pointer to the MOM tracer registry.

Definition at line 346 of file MOM.F90.

```
00346   type(tracer_registry_type),    pointer :: tracer_reg => null()
```

43.12.2.88 transport_ids

```
type(transport_diag_ids) mom::mom_control_struct::transport_ids
```

Handles used for transport diagnostics.

Definition at line 318 of file MOM.F90.

```
00318   type(transport_diag_ids) :: transport_ids  !< Handles used for transport diagnostics.
```

43.12.2.89 tv

```
type(thermo_var_ptrs) mom::mom_control_struct::tv
```

structure containing pointers to available thermodynamic fields

Definition at line 202 of file MOM.F90.

```
00202   type(thermo_var_ptrs) :: tv !< structure containing pointers to available thermodynamic fields
```

43.12.2.90 u

```
real, dimension( :  , :  , :  ), allocatable mom::mom_control_struct::u
```

zonal velocity component [L T-1 ~> m s-1]

Definition at line 170 of file MOM.F90.

```
00170   real allocable_, dimension(NIMEMB_PTR_,NJMEM_,NKMEM_) :: &
00171     u,  &           !< zonal velocity component [L T-1 ~> m s-1]
00172     uh, &           !< uh = u * h * dy at u grid points [H L2 T-1 ~> m3 s-1 or kg s-1]
00173     uhtr            !< accumulated zonal thickness fluxes to advect tracers [H L2 ~> m3 or kg]
```

43.12.2.91 u_prev

```
real, dimension(:,:,:), pointer mom::mom_control_struct::u_prev => NULL()
```

previous value of u stored for diagnostics [L T-1 ~> m s-1]

Definition at line 272 of file MOM.F90.

```
00272   real, dimension(:,:,:), pointer :: &
00273     u_prev => null(), &         !< previous value of u stored for diagnostics [L T-1 ~> m s-1]
00274     v_prev => null()            !< previous value of v stored for diagnostics [L T-1 ~> m s-1]
```

43.12.2.92 uh

```
real, dimension( :  , :  , :  ), allocatable mom::mom_control_struct::uh
```

uh = u * h * dy at u grid points [H L2 T-1 ~> m3 s-1 or kg s-1]

Definition at line 170 of file MOM.F90.

43.12.2.93 uhtr

```
real, dimension( :  , :  , :  ), allocatable mom::mom_control_struct::uhtr
```

accumulated zonal thickness fluxes to advect tracers [H L2 ~> m3 or kg]

Definition at line 170 of file MOM.F90.

43.12.2.94 update_obc_csp

```
type(update_obc_cs), pointer mom::mom_control_struct::update_obc_csp => NULL()
```

Pointer to the control structure for updating open boundary condition properties.

Definition at line 356 of file MOM.F90.

```
00356   type(update_obc_cs),           pointer :: update_obc_csp => null()
```

43.12.2.95 us

```
type(unit_scale_type), pointer mom::mom_control_struct::us => NULL()
```

structure containing various unit conversion factors

Definition at line 200 of file MOM.F90.

```
00200   type(unit_scale_type), pointer :: &
00201     us => null()    !< structure containing various unit conversion factors
```

43.12.2.96 use_ale_algorithm

```
logical mom::mom_control_struct::use_ale_algorithm
```

If true, use the ALE algorithm rather than layered isopycnal/stacked shallow water mode. This logical is set by calling the function useRegridding() from the MOM_regridding module.

Definition at line 224 of file MOM.F90.

```
00224   logical :: use_ale_algorithm  !< If true, use the ALE algorithm rather than layered
```

43.12.2.97 use_p_surf_in_eos

```
logical mom::mom_control_struct::use_p_surf_in_eos
```

If true, always include the surface pressure contributions in equation of state calculations.

Definition at line 255 of file MOM.F90.

```
00255   logical :: use_p_surf_in_eos       !< If true, always include the surface pressure contributions
```

43.12.2.98 use_rk2

```
logical mom::mom_control_struct::use_rk2
```

If true, use RK2 instead of RK3 in unsplit mode (i.e., no split between barotropic and baroclinic).

Definition at line 248 of file MOM.F90.

```
00248   logical :: use_rk2                !< If true, use RK2 instead of RK3 in unsplit mode
```

43.12.2.99 usemeke

```
logical mom::mom_control_struct::usemeke
```

If true, call the MEKE parameterization.

Definition at line 253 of file MOM.F90.

```
00253   logical :: usemeke                !< If true, call the MEKE parameterization.
```

43.12.2.100 usewaves

```
logical mom::mom_control_struct::usewaves
```

If true, update Stokes drift.

Definition at line 254 of file MOM.F90.

```
00254   logical :: usewaves               !< If true, update Stokes drift
```

43.12.2.101 v

```
real, dimension( :  , :  , :  ), allocatable mom::mom_control_struct::v
```

meridional velocity [L T-1 ~> m s-1]

Definition at line 174 of file MOM.F90.

```
00174   real allocable_, dimension(NIMEM_,NJMEMB_PTR_,NKMEM_) :: &
00175     v,  &           !< meridional velocity [L T-1 ~> m s-1]
00176     vh, &           !< vh = v * h * dx at v grid points [H L2 T-1 ~> m3 s-1 or kg s-1]
00177     vhtr            !< accumulated meridional thickness fluxes to advect tracers [H L2 ~> m3 or kg]
```

43.12.2.102 v_prev

```
real, dimension(:,:,:), pointer mom::mom_control_struct::v_prev => NULL()
```

previous value of v stored for diagnostics [L T-1 ~> m s-1]

Definition at line 272 of file MOM.F90.

43.12.2.103 varmix

```
type(varmix_cs), pointer mom::mom_control_struct::varmix => NULL()
```

Pointer to the control structure for the variable mixing module.

Definition at line 342 of file MOM.F90.

```
00342   type(varmix_cs),              pointer :: varmix => null()
```

43.12.2.104 vh

```
real, dimension( :  , :  , :  ), allocatable mom::mom_control_struct::vh
```

vh = v * h * dx at v grid points [H L2 T-1 ~> m3 s-1 or kg s-1]

Definition at line 174 of file MOM.F90.

43.12.2.105 vhtr

```
real, dimension( :  , :  , :  ), allocatable mom::mom_control_struct::vhtr
```

accumulated meridional thickness fluxes to advect tracers [H L2 ~> m3 or kg]

Definition at line 174 of file MOM.F90.

43.12.2.106 visc

```
type(vertvisc_type) mom::mom_control_struct::visc
```

structure containing vertical viscosities, bottom drag viscosities, and related fields

Definition at line 216 of file MOM.F90.

```
00216   type(vertvisc_type) :: visc !< structure containing vertical viscosities,
```

43.12.2.107 write_ic

```
logical mom::mom_control_struct::write_ic
```

If true, then the initial conditions will be written to file.

Definition at line 288 of file MOM.F90.

```
00288   logical :: write_ic          !< If true, then the initial conditions will be written to file
```

The documentation for this type was generated from the following file:

- /home/cermak/src/MOM6/src/core/MOM.F90

43.13 mom::mom_diag_ids Type Reference

A structure with diagnostic IDs of the state variables.

Private Attributes

- integer id_ssh_inst = -1
  2-d state field diagnotic ID

- integer id_u = -1
  3-d state field diagnostic IDs
- integer id_v = -1
  3-d state field diagnostic IDs
- integer id_h = -1
  3-d state field diagnostic IDs

43.13.1 Detailed Description

A structure with diagnostic IDs of the state variables.

Definition at line 155 of file MOM.F90.

43.13.2 Member Data Documentation

43.13.2.1 id_h

```
integer mom::mom_diag_ids::id_h = -1  [private]
```

3-d state field diagnostic IDs

Definition at line 157 of file MOM.F90.

43.13.2.2 id_ssh_inst

```
integer mom::mom_diag_ids::id_ssh_inst = -1  [private]
```

2-d state field diagnotic ID

Definition at line 160 of file MOM.F90.

```
00160   integer :: id_ssh_inst = -1
```

43.13.2.3 id_u

```
integer mom::mom_diag_ids::id_u = -1  [private]
```

3-d state field diagnostic IDs

Definition at line 157 of file MOM.F90.

```
00157   integer :: id_u  = -1, id_v  = -1, id_h  = -1
```

43.13.2.4 id_v

```
integer mom::mom_diag_ids::id_v = -1  [private]
```

3-d state field diagnostic IDs

Definition at line 157 of file MOM.F90.

The documentation for this type was generated from the following file:

- /home/cermak/src/MOM6/src/core/MOM.F90

43.14 mom_unit_scaling::unit_scale_type Type Reference

Describes various unit conversion factors.

Public Attributes

- real m_to_z

  A constant that translates distances in meters to the units of depth.
- real z_to_m

  A constant that translates distances in the units of depth to meters.
- real m_to_l

  A constant that translates lengths in meters to the units of horizontal lengths.
- real l_to_m

  A constant that translates lengths in the units of horizontal lengths to meters.
- real s_to_t

  A constant that translates time intervals in seconds to the units of time.
- real t_to_s

  A constant that translates the units of time to seconds.
- real r_to_kg_m3

  A constant that translates the units of density to kilograms per meter cubed.
- real kg_m3_to_r

  A constant that translates kilograms per meter cubed to the units of density.
- real q_to_j_kg

  A constant that translates the units of enthalpy to Joules per kilogram.
- real j_kg_to_q

  A constant that translates Joules per kilogram to the units of enthalpy.
- real z_to_l

  Convert vertical distances to lateral lengths.
- real l_to_z

  Convert lateral lengths to vertical distances.
- real l_t_to_m_s

  Convert lateral velocities from L T-1 to m s-1.
- real m_s_to_l_t

  Convert lateral velocities from m s-1 to L T-1.
- real l_t2_to_m_s2

  Convert lateral accelerations from L T-2 to m s-2.
- real z2_t_to_m2_s

  Convert vertical diffusivities from Z2 T-1 to m2 s-1.
- real m2_s_to_z2_t

  Convert vertical diffusivities from m2 s-1 to Z2 T-1.
- real w_m2_to_qrz_t

  Convert heat fluxes from W m-2 to Q R Z T-1.
- real qrz_t_to_w_m2

  Convert heat fluxes from Q R Z T-1 to W m-2.
- real rz_to_kg_m2

  Convert mass loads from R Z to kg m-2.
- real kg_m2s_to_rz_t

  Convert mass fluxes from kg m-2 s-1 to R Z T-1.
- real rz_t_to_kg_m2s

Convert mass fluxes from R Z T-1 to kg m-2 s-1.

- real rz3_t3_to_w_m2

  Convert turbulent kinetic energy fluxes from R Z3 T-3 to W m-2.
- real w_m2_to_rz3_t3

  Convert turbulent kinetic energy fluxes from W m-2 to R Z3 T-3.
- real rl2_t2_to_pa

  Convert pressures from R L2 T-2 to Pa.
- real m_to_z_restart = 0.0

  A copy of the m_to_Z that is used in restart files.
- real m_to_l_restart = 0.0

  A copy of the m_to_L that is used in restart files.
- real s_to_t_restart = 0.0

  A copy of the s_to_T that is used in restart files.
- real kg_m3_to_r_restart = 0.0

  A copy of the kg_m3_to_R that is used in restart files.
- real j_kg_to_q_restart = 0.0

  A copy of the J_kg_to_Q that is used in restart files.

43.14.1 Detailed Description

Describes various unit conversion factors.

Definition at line 14 of file MOM_unit_scaling.F90.

43.14.2 Member Data Documentation

43.14.2.1 j_kg_to_q

```
real mom_unit_scaling::unit_scale_type::j_kg_to_q
```

A constant that translates Joules per kilogram to the units of enthalpy.

Definition at line 24 of file MOM_unit_scaling.F90.

```
00024   real :: j_kg_to_q  !< A constant that translates Joules per kilogram to the units of enthalpy.
```

43.14.2.2 j_kg_to_q_restart

```
real mom_unit_scaling::unit_scale_type::j_kg_to_q_restart = 0.0
```

A copy of the J_kg_to_Q that is used in restart files.

Definition at line 50 of file MOM_unit_scaling.F90.

```
00050   real :: j_kg_to_q_restart = 0.0 !< A copy of the J_kg_to_Q that is used in restart files.
```

43.14.2.3 kg_m2s_to_rz_t

```
real mom_unit_scaling::unit_scale_type::kg_m2s_to_rz_t
```

Convert mass fluxes from kg m-2 s-1 to R Z T-1.

Definition at line 38 of file MOM_unit_scaling.F90.

```
00038   real :: kg_m2s_to_rz_t  !< Convert mass fluxes from kg m-2 s-1 to R Z T-1.
```

43.14.2.4 kg_m3_to_r

```
real mom_unit_scaling::unit_scale_type::kg_m3_to_r
```

A constant that translates kilograms per meter cubed to the units of density.

Definition at line 22 of file MOM_unit_scaling.F90.

```
00022   real :: kg_m3_to_r !< A constant that translates kilograms per meter cubed to the units of density.
```

43.14.2.5 kg_m3_to_r_restart

```
real mom_unit_scaling::unit_scale_type::kg_m3_to_r_restart = 0.0
```

A copy of the kg_m3_to_R that is used in restart files.

Definition at line 49 of file MOM_unit_scaling.F90.

```
00049   real :: kg_m3_to_r_restart = 0.0 !< A copy of the kg_m3_to_R that is used in restart files.
```

43.14.2.6 l_t2_to_m_s2

```
real mom_unit_scaling::unit_scale_type::l_t2_to_m_s2
```

Convert lateral accelerations from L T-2 to m s-2.

Definition at line 31 of file MOM_unit_scaling.F90.

```
00031   real :: l_t2_to_m_s2    !< Convert lateral accelerations from L T-2 to m s-2.
```

43.14.2.7 l_t_to_m_s

```
real mom_unit_scaling::unit_scale_type::l_t_to_m_s
```

Convert lateral velocities from L T-1 to m s-1.

Definition at line 29 of file MOM_unit_scaling.F90.

```
00029   real :: l_t_to_m_s      !< Convert lateral velocities from L T-1 to m s-1.
```

43.14.2.8 l_to_m

```
real mom_unit_scaling::unit_scale_type::l_to_m
```

A constant that translates lengths in the units of horizontal lengths to meters.

Definition at line 18 of file MOM_unit_scaling.F90.

```
00018   real :: l_to_m !< A constant that translates lengths in the units of horizontal lengths to meters.
```

43.14.2.9 l_to_z

```
real mom_unit_scaling::unit_scale_type::l_to_z
```

Convert lateral lengths to vertical distances.

Definition at line 28 of file MOM_unit_scaling.F90.

```
00028   real :: l_to_z          !< Convert lateral lengths to vertical distances
```

43.14.2.10 m2_s_to_z2_t

```
real mom_unit_scaling::unit_scale_type::m2_s_to_z2_t
```

Convert vertical diffusivities from m2 s-1 to Z2 T-1.

Definition at line 33 of file MOM_unit_scaling.F90.

```
00033   real :: m2_s_to_z2_t    !< Convert vertical diffusivities from m2 s-1 to Z2 T-1.
```

43.14.2.11 m_s_to_l_t

```
real mom_unit_scaling::unit_scale_type::m_s_to_l_t
```

Convert lateral velocities from m s-1 to L T-1.

Definition at line 30 of file MOM_unit_scaling.F90.

```
00030   real :: m_s_to_l_t      !< Convert lateral velocities from m s-1 to L T-1.
```

43.14.2.12 m_to_l

```
real mom_unit_scaling::unit_scale_type::m_to_l
```

A constant that translates lengths in meters to the units of horizontal lengths.

Definition at line 17 of file MOM_unit_scaling.F90.

```
00017   real :: m_to_l !< A constant that translates lengths in meters to the units of horizontal lengths.
```

43.14.2.13 m_to_l_restart

```
real mom_unit_scaling::unit_scale_type::m_to_l_restart = 0.0
```

A copy of the m_to_L that is used in restart files.

Definition at line 47 of file MOM_unit_scaling.F90.

```
00047   real :: m_to_l_restart = 0.0 !< A copy of the m_to_L that is used in restart files.
```

43.14.2.14 m_to_z

```
real mom_unit_scaling::unit_scale_type::m_to_z
```

A constant that translates distances in meters to the units of depth.

Definition at line 15 of file MOM_unit_scaling.F90.

```
00015   real :: m_to_z !< A constant that translates distances in meters to the units of depth.
```

43.14.2.15 m_to_z_restart

```
real mom_unit_scaling::unit_scale_type::m_to_z_restart = 0.0
```

A copy of the m_to_Z that is used in restart files.

Definition at line 46 of file MOM_unit_scaling.F90.

```
00046   real :: m_to_z_restart = 0.0 !< A copy of the m_to_Z that is used in restart files.
```

43.14.2.16 q_to_j_kg

```
real mom_unit_scaling::unit_scale_type::q_to_j_kg
```

A constant that translates the units of enthalpy to Joules per kilogram.

Definition at line 23 of file MOM_unit_scaling.F90.

```
00023   real :: q_to_j_kg  !< A constant that translates the units of enthalpy to Joules per kilogram.
```

43.14.2.17 qrz_t_to_w_m2

```
real mom_unit_scaling::unit_scale_type::qrz_t_to_w_m2
```

Convert heat fluxes from Q R Z T-1 to W m-2.

Definition at line 35 of file MOM_unit_scaling.F90.

```
00035   real :: qrz_t_to_w_m2   !< Convert heat fluxes from Q R Z T-1 to W m-2.
```

43.14.2.18 r_to_kg_m3

```
real mom_unit_scaling::unit_scale_type::r_to_kg_m3
```

A constant that translates the units of density to kilograms per meter cubed.

Definition at line 21 of file MOM_unit_scaling.F90.

```
00021   real :: r_to_kg_m3 !< A constant that translates the units of density to kilograms per meter cubed.
```

43.14.2.19 rl2_t2_to_pa

```
real mom_unit_scaling::unit_scale_type::rl2_t2_to_pa
```

Convert pressures from R L2 T-2 to Pa.

Definition at line 42 of file MOM_unit_scaling.F90.

```
00042   real :: rl2_t2_to_pa    !< Convert pressures from R L2 T-2 to Pa.
```

43.14.2.20 rz3_t3_to_w_m2

```
real mom_unit_scaling::unit_scale_type::rz3_t3_to_w_m2
```

Convert turbulent kinetic energy fluxes from R Z3 T-3 to W m-2.

Definition at line 40 of file MOM_unit_scaling.F90.

```
00040   real :: rz3_t3_to_w_m2  !< Convert turbulent kinetic energy fluxes from R Z3 T-3 to W m-2.
```

43.14.2.21 rz_t_to_kg_m2s

```
real mom_unit_scaling::unit_scale_type::rz_t_to_kg_m2s
```

Convert mass fluxes from R Z T-1 to kg m-2 s-1.

Definition at line 39 of file MOM_unit_scaling.F90.

```
00039   real :: rz_t_to_kg_m2s  !< Convert mass fluxes from R Z T-1 to kg m-2 s-1.
```

43.14.2.22 rz_to_kg_m2

```
real mom_unit_scaling::unit_scale_type::rz_to_kg_m2
```

Convert mass loads from R Z to kg m-2.

Definition at line 37 of file MOM_unit_scaling.F90.

```
00037   real :: rz_to_kg_m2     !< Convert mass loads from R Z to kg m-2.
```

43.14.2.23 s_to_t

```
real mom_unit_scaling::unit_scale_type::s_to_t
```

A constant that translates time intervals in seconds to the units of time.

Definition at line 19 of file MOM_unit_scaling.F90.

```
00019   real :: s_to_t !< A constant that translates time intervals in seconds to the units of time.
```

43.14.2.24 s_to_t_restart

```
real mom_unit_scaling::unit_scale_type::s_to_t_restart = 0.0
```

A copy of the s_to_T that is used in restart files.

Definition at line 48 of file MOM_unit_scaling.F90.

```
00048   real :: s_to_t_restart = 0.0 !< A copy of the s_to_T that is used in restart files.
```

43.14.2.25 t_to_s

```
real mom_unit_scaling::unit_scale_type::t_to_s
```

A constant that translates the units of time to seconds.

Definition at line 20 of file MOM_unit_scaling.F90.

```
00020   real :: t_to_s !< A constant that translates the units of time to seconds.
```

43.14.2.26 w_m2_to_qrz_t

```
real mom_unit_scaling::unit_scale_type::w_m2_to_qrz_t
```

Convert heat fluxes from W m-2 to Q R Z T-1.

Definition at line 34 of file MOM_unit_scaling.F90.

```
00034   real :: w_m2_to_qrz_t   !< Convert heat fluxes from W m-2 to Q R Z T-1.
```

43.14.2.27 w_m2_to_rz3_t3

```
real mom_unit_scaling::unit_scale_type::w_m2_to_rz3_t3
```

Convert turbulent kinetic energy fluxes from W m-2 to R Z3 T-3.

Definition at line 41 of file MOM_unit_scaling.F90.

```
00041   real :: w_m2_to_rz3_t3  !< Convert turbulent kinetic energy fluxes from W m-2 to R Z3 T-3.
```

43.14.2.28 z2_t_to_m2_s

```
real mom_unit_scaling::unit_scale_type::z2_t_to_m2_s
```

Convert vertical diffusivities from Z2 T-1 to m2 s-1.

Definition at line 32 of file MOM_unit_scaling.F90.

```
00032   real :: z2_t_to_m2_s    !< Convert vertical diffusivities from Z2 T-1 to m2 s-1.
```

43.14.2.29 z_to_l

```
real mom_unit_scaling::unit_scale_type::z_to_l
```

Convert vertical distances to lateral lengths.

Definition at line 27 of file MOM_unit_scaling.F90.

```
00027   real :: z_to_l          !< Convert vertical distances to lateral lengths
```

43.14.2.30 z_to_m

```
real mom_unit_scaling::unit_scale_type::z_to_m
```

A constant that translates distances in the units of depth to meters.

Definition at line 16 of file MOM_unit_scaling.F90.

```
00016   real :: z_to_m !< A constant that translates distances in the units of depth to meters.
```

The documentation for this type was generated from the following file:

- /home/cermak/src/MOM6/src/framework/MOM_unit_scaling.F90

Chapter 44

File Documentation

44.1 /home/cermak/src/MOM6/src/ALE/_ALE.dox File Reference

44.2 /home/cermak/src/MOM6/src/ALE/_ALE_timestep.dox File Reference

44.3 /home/cermak/src/MOM6/src/ALE/MOM_ALE.F90 File Reference

```
#include <MOM_memory.h>
```

Data Types

- type mom_ale::ale_cs

  ALE control structure.

Modules

- module mom_ale

  This module contains the main regridding routines.

Functions/Subroutines

- subroutine, public mom_ale::ale_init (param_file, GV, US, max_depth, CS)

  This routine is typically called (from initialize_MOM in file MOM.F90) before the main time integration loop to initialize the regridding stuff. We read the MOM_input file to register the values of different regridding/remapping parameters.
- subroutine, public mom_ale::ale_register_diags (Time, G, GV, US, diag, CS)

  Initialize diagnostics for the ALE module.
- subroutine, public mom_ale::adjustgridforintegrity (CS, G, GV, h)

  Crudely adjust (initial) grid for integrity. This routine is typically called (from initialize_MOM in file MOM.F90) before the main time integration loop to initialize the regridding stuff. We read the MOM_input file to register the values of different regridding/remapping parameters.
- subroutine, public mom_ale::ale_end (CS)

End of regridding (memory deallocation). This routine is typically called (from MOM_end in file MOM.F90) after the main time integration loop to deallocate the regridding stuff.

- subroutine, public mom_ale::ale_main (G, GV, US, h, u, v, tv, Reg, CS, OBC, dt, frac_shelf_h)

  Takes care of (1) building a new grid and (2) remapping all variables between the old grid and the new grid. The creation of the new grid can be based on z coordinates, target interface densities, sigma coordinates or any arbitrary coordinate system.

- subroutine, public mom_ale::ale_main_offline (G, GV, h, tv, Reg, CS, OBC, dt)

  Takes care of (1) building a new grid and (2) remapping all variables between the old grid and the new grid. The creation of the new grid can be based on z coordinates, target interface densities, sigma coordinates or any arbitrary coordinate system.

- subroutine, public mom_ale::ale_offline_inputs (CS, G, GV, h, tv, Reg, uhtr, vhtr, Kd, debug, OBC)

  Regrid/remap stored fields used for offline tracer integrations. These input fields are assumed to have the same layer thicknesses at the end of the last offline interval (which should be a Zstar grid). This routine builds a grid on the runtime specified vertical coordinate.

- subroutine, public mom_ale::ale_offline_tracer_final (G, GV, h, tv, h_target, Reg, CS, OBC)

  Remaps all tracers from h onto h_target. This is intended to be called when tracers are done offline. In the case where transports don't quite conserve, we still want to make sure that layer thicknesses offline do not drift too far away from the online model.

- subroutine mom_ale::check_grid (G, GV, h, threshold)

  Check grid for negative thicknesses.

- subroutine, public mom_ale::ale_build_grid (G, GV, regridCS, remapCS, h, tv, debug, frac_shelf_h)

  Generates new grid.

- subroutine, public mom_ale::ale_regrid_accelerated (CS, G, GV, h, tv, n, u, v, OBC, Reg, dt, dzRegrid, initial)

  For a state-based coordinate, accelerate the process of regridding by repeatedly applying the grid calculation algorithm.

- subroutine mom_ale::remap_all_state_vars (CS_remapping, CS_ALE, G, GV, h_old, h_new, Reg, OBC, dx↩ Interface, u, v, debug, dt)

  This routine takes care of remapping all variable between the old and the new grids. When velocity components need to be remapped, thicknesses at velocity points are taken to be arithmetic averages of tracer thicknesses. This routine is called during initialization of the model at time=0, to remap initiali conditions to the model grid. It is also called during a time step to update the state.

- subroutine, public mom_ale::ale_remap_scalar (CS, G, GV, nk_src, h_src, s_src, h_dst, s_dst, all_cells, old↩ _remap, answers_2018)

  Remaps a single scalar between grids described by thicknesses h_src and h_dst. h_dst must be dimensioned as a model array with GVke layers while h_src can have an arbitrary number of layers specified by nk_src.

- subroutine, public mom_ale::ts_plm_edge_values (CS, S_t, S_b, T_t, T_b, G, GV, tv, h, bdry_extrap)

  Calculate edge values (top and bottom of layer) for T and S consistent with a PLM reconstruction in the vertical direction. Boundary reconstructions are PCM unless bdry_extrap is true.

- subroutine, public mom_ale::ale_plm_edge_values (CS, G, GV, h, Q, bdry_extrap, Q_t, Q_b)

  Calculate edge values (top and bottom of layer) 3d scalar array. Boundary reconstructions are PCM unless bdry_↩ extrap is true.

- subroutine, public mom_ale::ts_ppm_edge_values (CS, S_t, S_b, T_t, T_b, G, GV, tv, h, bdry_extrap)

  Calculate edge values (top and bottom of layer) for T and S consistent with a PPM reconstruction in the vertical direction. Boundary reconstructions are PCM unless bdry_extrap is true.

- subroutine, public mom_ale::ale_initregridding (GV, US, max_depth, param_file, mdl, regridCS)

  Initializes regridding for the main ALE algorithm.

- real function, dimension(cs%nk+1), public mom_ale::ale_getcoordinate (CS)

  Query the target coordinate interfaces positions.

- character(len=20) function, public mom_ale::ale_getcoordinateunits (CS)

  Query the target coordinate units.

- logical function, public mom_ale::ale_remap_init_conds (CS)

  Returns true if initial conditions should be regridded and remapped.

- subroutine, public mom_ale::ale_update_regrid_weights (dt, CS)

  Updates the weights for time filtering the new grid generated in regridding.

- subroutine, public mom_ale::ale_updateverticalgridtype (CS, GV)

  Update the vertical grid type with ALE information. This subroutine sets information in the verticalGrid_type to be consistent with the use of ALE mode.

- subroutine, public mom_ale::ale_writecoordinatefile (CS, GV, directory)

  Write the vertical coordinate information into a file. This subroutine writes out a file containing any available data related to the vertical grid used by the MOM ocean model when in ALE mode.

- subroutine, public mom_ale::ale_initthicknesstocoord (CS, G, GV, h)

  Set h to coordinate values for fixed coordinate systems.

44.4 MOM_ALE.F90

```
00001 !> This module contains the main regridding routines.
00002 !!
00003 !! Regridding comprises two steps:
00004 !! 1. Interpolation and creation of a new grid based on target interface
00005 !!    densities (or any other criterion).
00006 !! 2. Remapping of quantities between old grid and new grid.
00007 !!
00008 !! Original module written by Laurent White, 2008.06.09
00009 module mom_ale
00010
00011 ! This file is part of MOM6. See LICENSE.md for the license.
00012
00013 use mom_debugging,         only : check_column_integrals
00014 use mom_diag_mediator,     only : register_diag_field, post_data, diag_ctrl
00015 use mom_diag_mediator,     only : time_type, diag_update_remap_grids
00016 use mom_diag_vkernels,     only : interpolate_column, reintegrate_column
00017 use mom_domains,           only : create_group_pass, do_group_pass, group_pass_type
00018 use mom_eos,               only : calculate_density
00019 use mom_domains,           only : create_group_pass, do_group_pass, group_pass_type
00020 use mom_error_handler,     only : mom_error, fatal, warning
00021 use mom_error_handler,     only : calltree_showquery
00022 use mom_error_handler,     only : calltree_enter, calltree_leave, calltree_waypoint
00023 use mom_file_parser,       only : get_param, param_file_type, log_param
00024 use mom_io,                only : vardesc, var_desc, fieldtype, single_file
00025 use mom_io,                only : create_file, write_field, close_file
00026 use mom_interface_heights,only : find_eta
00027 use mom_open_boundary,     only : ocean_obc_type, obc_direction_e, obc_direction_w
00028 use mom_open_boundary,     only : obc_direction_n, obc_direction_s
00029 use mom_regridding,        only : initialize_regridding, regridding_main, end_regridding
00030 use mom_regridding,        only : uniformresolution
00031 use mom_regridding,        only : inflate_vanished_layers_old
00032 use mom_regridding,        only : set_target_densities_from_gv, set_target_densities
00033 use mom_regridding,        only : regriddingcoordinatemodedoc, default_coordinate_mode
00034 use mom_regridding,        only : regriddinginterpschemedoc, regriddingdefaultinterpscheme
00035 use mom_regridding,        only : regriddingdefaultboundaryextrapolation
00036 use mom_regridding,        only : regriddingdefaultminthickness
00037 use mom_regridding,        only : regridding_cs, set_regrid_params
00038 use mom_regridding,        only : getcoordinateinterfaces, getcoordinateresolution
00039 use mom_regridding,        only : getcoordinateunits, getcoordinateshortname
00040 use mom_regridding,        only : getstaticthickness
00041 use mom_remapping,         only : initialize_remapping, end_remapping
00042 use mom_remapping,         only : remapping_core_h, remapping_core_w
00043 use mom_remapping,         only : remappingschemesdoc, remappingdefaultscheme
00044 use mom_remapping,         only : remapping_cs, dzfromh1h2
00045 use mom_string_functions, only : uppercase, extractword, extract_integer
00046 use mom_tracer_registry,   only : tracer_registry_type, tracer_type, mom_tracer_chkinv
00047 use mom_unit_scaling,      only : unit_scale_type
00048 use mom_variables,         only : ocean_grid_type, thermo_var_ptrs
00049 use mom_verticalgrid,      only : get_thickness_units, verticalgrid_type
00050
00051 !use regrid_consts,        only : coordinateMode, DEFAULT_COORDINATE_MODE
00052 use regrid_consts,         only : coordinateunits, coordinatemode, state_dependent
00053 use regrid_edge_values,    only : edge_values_implicit_h4
00054 use plm_functions,         only : plm_reconstruction, plm_boundary_extrapolation
00055 use plm_functions,         only : plm_extrapolate_slope, plm_monotonized_slope, plm_slope_wa
00056 use ppm_functions,         only : ppm_reconstruction, ppm_boundary_extrapolation
00057
00058 implicit none ; private
00059 #include <MOM_memory.h>
00060
00061
00062 !> ALE control structure
00063 type, public :: ale_cs ; private
00064   logical :: remap_uv_using_old_alg !< If true, uses the old "remapping via a delta z"
00065                                     !! method. If False, uses the new method that
00066                                     !! remaps between grids described by h.
```

```
00067
00068   real :: regrid_time_scale !< The time-scale used in blending between the current (old) grid
00069                             !! and the target (new) grid [T ~> s]
00070
00071   type(regridding_cs) :: regridcs !< Regridding parameters and work arrays
00072   type(remapping_cs)  :: remapcs  !< Remapping parameters and work arrays
00073
00074   integer :: nk             !< Used only for queries, not directly by this module
00075
00076   logical :: remap_after_initialization !< Indicates whether to regrid/remap after initializing the state.
00077
00078   logical :: answers_2018   !< If true, use the order of arithmetic and expressions for remapping
00079                             !! that recover the answers from the end of 2018.  Otherwise, use more
00080                             !! robust and accurate forms of mathematically equivalent expressions.
00081
00082   logical :: show_call_tree !< For debugging
00083
00084   ! for diagnostics
00085   type(diag_ctrl), pointer           :: diag                          !< structure to regulate output
00086   integer, dimension(:), allocatable :: id_tracer_remap_tendency      !< diagnostic id
00087   integer, dimension(:), allocatable :: id_htracer_remap_tendency     !< diagnostic id
00088   integer, dimension(:), allocatable :: id_htracer_remap_tendency_2d  !< diagnostic id
00089   logical, dimension(:), allocatable :: do_tendency_diag              !< flag for doing diagnostics
00090   integer                            :: id_dzregrid = -1              !< diagnostic id
00091
00092   ! diagnostic for fields prior to applying ALE remapping
00093   integer :: id_u_preale = -1 !< diagnostic id for zonal velocity before ALE.
00094   integer :: id_v_preale = -1 !< diagnostic id for meridional velocity before ALE.
00095   integer :: id_h_preale = -1 !< diagnostic id for layer thicknesses before ALE.
00096   integer :: id_t_preale = -1 !< diagnostic id for temperatures before ALE.
00097   integer :: id_s_preale = -1 !< diagnostic id for salinities before ALE.
00098   integer :: id_e_preale = -1 !< diagnostic id for interface heights before ALE.
00099   integer :: id_vert_remap_h = -1      !< diagnostic id for layer thicknesses used for remapping
00100   integer :: id_vert_remap_h_tendency = -1 !< diagnostic id for layer thickness tendency due to ALE
00101
00102 end type
00103
00104 ! Publicly available functions
00105 public ale_init
00106 public ale_end
00107 public ale_main
00108 public ale_main_offline
00109 public ale_offline_inputs
00110 public ale_offline_tracer_final
00111 public ale_build_grid
00112 public ale_regrid_accelerated
00113 public ale_remap_scalar
00114 public ale_plm_edge_values
00115 public ts_plm_edge_values
00116 public ts_ppm_edge_values
00117 public adjustgridforintegrity
00118 public ale_initregridding
00119 public ale_getcoordinate
00120 public ale_getcoordinateunits
00121 public ale_writecoordinatefile
00122 public ale_updateverticalgridtype
00123 public ale_initthicknesstocoord
00124 public ale_update_regrid_weights
00125 public ale_remap_init_conds
00126 public ale_register_diags
00127
00128 ! A note on unit descriptions in comments: MOM6 uses units that can be rescaled for dimensional
00129 ! consistency testing. These are noted in comments with units like Z, H, L, and T, along with
00130 ! their mks counterparts with notation like "a velocity [Z T-1 ~> m s-1]".  If the units
00131 ! vary with the Boussinesq approximation, the Boussinesq variant is given first.
00132
00133 contains
00134
00135 !> This routine is typically called (from initialize_MOM in file MOM.F90)
00136 !! before the main time integration loop to initialize the regridding stuff.
00137 !! We read the MOM_input file to register the values of different
00138 !! regridding/remapping parameters.
00139 subroutine ale_init( param_file, GV, US, max_depth, CS)
00140   type(param_file_type),   intent(in) :: param_file !< Parameter file
00141   type(verticalgrid_type), intent(in) :: GV         !< Ocean vertical grid structure
00142   type(unit_scale_type),   intent(in) :: US         !< A dimensional unit scaling type
00143   real,                    intent(in) :: max_depth  !< The maximum depth of the ocean [Z ~> m].
00144   type(ale_cs),            pointer    :: CS         !< Module control structure
00145
00146   ! Local variables
00147   real, dimension(:), allocatable :: dz
00148   character(len=40)               :: mdl = "MOM_ALE" ! This module's name.
00149   character(len=80)               :: string ! Temporary strings
00150   real                            :: filter_shallow_depth, filter_deep_depth
00151   logical       :: default_2018_answers
00152   logical                         :: check_reconstruction
00153   logical                         :: check_remapping
```

```
00154   logical                          :: force_bounds_in_subcell
00155   logical                          :: local_logical
00156   logical                          :: remap_boundary_extrap
00157
00158   if (associated(cs)) then
00159     call mom_error(warning, "ALE_init called with an associated "// &
00160                             "control structure.")
00161     return
00162   endif
00163   allocate(cs)
00164
00165   cs%show_call_tree = calltree_showquery()
00166   if (cs%show_call_tree) call calltree_enter("ALE_init(), MOM_ALE.F90")
00167
00168   call get_param(param_file, mdl, "REMAP_UV_USING_OLD_ALG", cs%remap_uv_using_old_alg, &
00169                  "If true, uses the old remapping-via-a-delta-z method for "//&
00170                  "remapping u and v. If false, uses the new method that remaps "//&
00171                  "between grids described by an old and new thickness.", &
00172                  default=.false.)
00173
00174   ! Initialize and configure regridding
00175   call ale_initregridding(gv, us, max_depth, param_file, mdl, cs%regridCS)
00176
00177   ! Initialize and configure remapping
00178   call get_param(param_file, mdl, "REMAPPING_SCHEME", string, &
00179                  "This sets the reconstruction scheme used "//&
00180                  "for vertical remapping for all variables. "//&
00181                  "It can be one of the following schemes: "//&
00182                  trim(remappingschemesdoc), default=remappingdefaultscheme)
00183   call get_param(param_file, mdl, "FATAL_CHECK_RECONSTRUCTIONS", check_reconstruction, &
00184                  "If true, cell-by-cell reconstructions are checked for "//&
00185                  "consistency and if non-monotonicity or an inconsistency is "//&
00186                  "detected then a FATAL error is issued.", default=.false.)
00187   call get_param(param_file, mdl, "FATAL_CHECK_REMAPPING", check_remapping, &
00188                  "If true, the results of remapping are checked for "//&
00189                  "conservation and new extrema and if an inconsistency is "//&
00190                  "detected then a FATAL error is issued.", default=.false.)
00191   call get_param(param_file, mdl, "REMAP_BOUND_INTERMEDIATE_VALUES", force_bounds_in_subcell, &
00192                  "If true, the values on the intermediate grid used for remapping "//&
00193                  "are forced to be bounded, which might not be the case due to "//&
00194                  "round off.", default=.false.)
00195   call get_param(param_file, mdl, "REMAP_BOUNDARY_EXTRAP", remap_boundary_extrap, &
00196                  "If true, values at the interfaces of boundary cells are "//&
00197                  "extrapolated instead of piecewise constant", default=.false.)
00198   call get_param(param_file, mdl, "DEFAULT_2018_ANSWERS", default_2018_answers, &
00199                  "This sets the default value for the various _2018_ANSWERS parameters.", &
00200                  default=.false.)
00201   call get_param(param_file, mdl, "REMAPPING_2018_ANSWERS", cs%answers_2018, &
00202                  "If true, use the order of arithmetic and expressions that recover the "//&
00203                  "answers from the end of 2018.  Otherwise, use updated and more robust "//&
00204                  "forms of the same expressions.", default=default_2018_answers)
00205   call initialize_remapping( cs%remapCS, string, &
00206                              boundary_extrapolation=remap_boundary_extrap, &
00207                              check_reconstruction=check_reconstruction, &
00208                              check_remapping=check_remapping, &
00209                              force_bounds_in_subcell=force_bounds_in_subcell, &
00210                              answers_2018=cs%answers_2018)
00211
00212   call get_param(param_file, mdl, "REMAP_AFTER_INITIALIZATION", cs%remap_after_initialization, &
00213                  "If true, applies regridding and remapping immediately after "//&
00214                  "initialization so that the state is ALE consistent. This is a "//&
00215                  "legacy step and should not be needed if the initialization is "//&
00216                  "consistent with the coordinate mode.", default=.true.)
00217
00218   call get_param(param_file, mdl, "REGRID_TIME_SCALE", cs%regrid_time_scale, &
00219                  "The time-scale used in blending between the current (old) grid "//&
00220                  "and the target (new) grid. A short time-scale favors the target "//&
00221                  "grid (0. or anything less than DT_THERM) has no memory of the old "//&
00222                  "grid. A very long time-scale makes the model more Lagrangian.", &
00223                  units="s", default=0., scale=us%s_to_T)
00224   call get_param(param_file, mdl, "REGRID_FILTER_SHALLOW_DEPTH", filter_shallow_depth, &
00225                  "The depth above which no time-filtering is applied. Above this depth "//&
00226                  "final grid exactly matches the target (new) grid.", &
00227                  units="m", default=0., scale=gv%m_to_H)
00228   call get_param(param_file, mdl, "REGRID_FILTER_DEEP_DEPTH", filter_deep_depth, &
00229                  "The depth below which full time-filtering is applied with time-scale "//&
00230                  "REGRID_TIME_SCALE. Between depths REGRID_FILTER_SHALLOW_DEPTH and "//&
00231                  "REGRID_FILTER_SHALLOW_DEPTH the filter weights adopt a cubic profile.", &
00232                  units="m", default=0., scale=gv%m_to_H)
00233   call set_regrid_params(cs%regridCS, depth_of_time_filter_shallow=filter_shallow_depth, &
00234                          depth_of_time_filter_deep=filter_deep_depth)
00235   call get_param(param_file, mdl, "REGRID_USE_OLD_DIRECTION", local_logical, &
00236                  "If true, the regridding ntegrates upwards from the bottom for "//&
00237                  "interface positions, much as the main model does. If false "//&
00238                  "regridding integrates downward, consistant with the remapping "//&
00239                  "code.", default=.true., do_not_log=.true.)
00240   call set_regrid_params(cs%regridCS, integrate_downward_for_e=.not.local_logical)
```

```
00241
00242   ! Keep a record of values for subsequent queries
00243   cs%nk = gv%ke
00244
00245   if (cs%show_call_tree) call calltree_leave("ALE_init()")
00246 end subroutine ale_init
00247
00248 !> Initialize diagnostics for the ALE module.
00249 subroutine ale_register_diags(Time, G, GV, US, diag, CS)
00250   type(time_type),target,     intent(in)  :: Time  !< Time structure
00251   type(ocean_grid_type),      intent(in)  :: G     !< Grid structure
00252   type(unit_scale_type),      intent(in)  :: US    !< A dimensional unit scaling type
00253   type(verticalgrid_type),    intent(in)  :: GV    !< Ocean vertical grid structure
00254   type(diag_ctrl), target,    intent(in)  :: diag  !< Diagnostics control structure
00255   type(ale_cs), pointer                   :: CS    !< Module control structure
00256
00257   cs%diag => diag
00258
00259   ! These diagnostics of the state variables before ALE are useful for
00260   ! debugging the ALE code.
00261   cs%id_u_preale = register_diag_field('ocean_model', 'u_preale', diag%axesCuL, time, &
00262       'Zonal velocity before remapping', 'm s-1', conversion=us%L_T_to_m_s)
00263   cs%id_v_preale = register_diag_field('ocean_model', 'v_preale', diag%axesCvL, time, &
00264       'Meridional velocity before remapping', 'm s-1', conversion=us%L_T_to_m_s)
00265   cs%id_h_preale = register_diag_field('ocean_model', 'h_preale', diag%axesTL, time, &
00266       'Layer Thickness before remapping', get_thickness_units(gv), &
00267       conversion=gv%H_to_MKS, v_extensive=.true.)
00268   cs%id_T_preale = register_diag_field('ocean_model', 'T_preale', diag%axesTL, time, &
00269       'Temperature before remapping', 'degC')
00270   cs%id_S_preale = register_diag_field('ocean_model', 'S_preale', diag%axesTL, time, &
00271       'Salinity before remapping', 'PSU')
00272   cs%id_e_preale = register_diag_field('ocean_model', 'e_preale', diag%axesTi, time, &
00273       'Interface Heights before remapping', 'm', conversion=us%Z_to_m)
00274
00275   cs%id_dzRegrid = register_diag_field('ocean_model','dzRegrid',diag%axesTi,time, &
00276       'Change in interface height due to ALE regridding', 'm', &
00277       conversion=gv%H_to_m)
00278   cs%id_vert_remap_h = register_diag_field('ocean_model', 'vert_remap_h', &
00279       diag%axestl, time, 'layer thicknesses after ALE regridding and remapping', 'm', &
00280       conversion=gv%H_to_m, v_extensive=.true.)
00281   cs%id_vert_remap_h_tendency = register_diag_field('ocean_model','vert_remap_h_tendency',diag%axestl,time,
      &
00282       'Layer thicknesses tendency due to ALE regridding and remapping', 'm', &
00283       conversion=gv%H_to_m*us%s_to_T, v_extensive = .true.)
00284
00285 end subroutine ale_register_diags
00286
00287 !> Crudely adjust (initial) grid for integrity.
00288 !! This routine is typically called (from initialize_MOM in file MOM.F90)
00289 !! before the main time integration loop to initialize the regridding stuff.
00290 !! We read the MOM_input file to register the values of different
00291 !! regridding/remapping parameters.
00292 subroutine adjustgridforintegrity( CS, G, GV, h )
00293   type(ale_cs),                              pointer       :: CS  !< Regridding parameters and
      options
00294   type(ocean_grid_type),                     intent(in)    :: G   !< Ocean grid informations
00295   type(verticalgrid_type),                   intent(in)    :: GV  !< Ocean vertical grid structure
00296   real, dimension(SZI_(G),SZJ_(G),SZK_(GV)), intent(inout) :: h   !< Current 3D grid thickness that
00297                                                                     !! are to be adjusted [H ~> m or kg-2]
00298   call inflate_vanished_layers_old( cs%regridCS, g, gv, h(:,:,:) )
00299
00300 end subroutine adjustgridforintegrity
00301
00302
00303 !> End of regridding (memory deallocation).
00304 !! This routine is typically called (from MOM_end in file MOM.F90)
00305 !! after the main time integration loop to deallocate the regridding stuff.
00306 subroutine ale_end(CS)
00307   type(ale_cs), pointer :: CS  !< module control structure
00308
00309   ! Deallocate memory used for the regridding
00310   call end_remapping( cs%remapCS )
00311   call end_regridding( cs%regridCS )
00312
00313   deallocate(cs)
00314
00315 end subroutine ale_end
00316
00317 !> Takes care of (1) building a new grid and (2) remapping all variables between
00318 !! the old grid and the new grid. The creation of the new grid can be based
00319 !! on z coordinates, target interface densities, sigma coordinates or any
00320 !! arbitrary coordinate system.
00321 subroutine ale_main( G, GV, US, h, u, v, tv, Reg, CS, OBC, dt, frac_shelf_h)
00322   type(ocean_grid_type),                     intent(in)    :: G   !< Ocean grid informations
00323   type(verticalgrid_type),                   intent(in)    :: GV  !< Ocean vertical grid structure
00324   type(unit_scale_type),                     intent(in)    :: US  !< A dimensional unit
      scaling type
```

```
00325   real, dimension(SZI_(G),SZJ_(G),SZK_(GV)),  intent(inout) :: h   !< Current 3D grid obtained after the
00326                                                                     !! last time step [H ~> m or kg m-2]
00327   real, dimension(SZIB_(G),SZJ_(G),SZK_(GV)), intent(inout) :: u   !< Zonal velocity field [L T-1 ~> m s-1]
00328   real, dimension(SZI_(G),SZJB_(G),SZK_(GV)), intent(inout) :: v   !< Meridional velocity field [L T-1 ~> m
      s-1]
00329   type(thermo_var_ptrs),                       intent(inout) :: tv  !< Thermodynamic variable structure
00330   type(tracer_registry_type),                  pointer       :: Reg !< Tracer registry structure
00331   type(ale_cs),                                pointer       :: CS  !< Regridding parameters and
      options
00332   type(ocean_obc_type),                        pointer       :: OBC !< Open boundary structure
00333   real,                              optional, intent(in)    :: dt  !< Time step between calls to ALE_main
      [T ~> s]
00334   real, dimension(:,:),              optional, pointer       :: frac_shelf_h !< Fractional ice shelf
      coverage
00335   ! Local variables
00336   real, dimension(SZI_(G),SZJ_(G),SZK_(GV)+1) :: dzRegrid ! The change in grid interface positions
00337   real, dimension(SZI_(G),SZJ_(G),SZK_(GV)+1) :: eta_preale
00338   real, dimension(SZI_(G),SZJ_(G),SZK_(GV)) :: h_new ! New 3D grid obtained after last time step [H ~> m or
      kg-2]
00339   integer :: nk, i, j, k, isc, iec, jsc, jec
00340   logical :: ice_shelf
00341
00342   nk = gv%ke; isc = g%isc; iec = g%iec; jsc = g%jsc; jec = g%jec
00343
00344   ice_shelf = .false.
00345   if (present(frac_shelf_h)) then
00346     if (associated(frac_shelf_h)) ice_shelf = .true.
00347   endif
00348
00349   if (cs%show_call_tree) call calltree_enter("ALE_main(), MOM_ALE.F90")
00350
00351   ! These diagnostics of the state before ALE is applied are mostly used for debugging.
00352   if (cs%id_u_preale > 0) call post_data(cs%id_u_preale, u,    cs%diag)
00353   if (cs%id_v_preale > 0) call post_data(cs%id_v_preale, v,    cs%diag)
00354   if (cs%id_h_preale > 0) call post_data(cs%id_h_preale, h,    cs%diag)
00355   if (cs%id_T_preale > 0) call post_data(cs%id_T_preale, tv%T, cs%diag)
00356   if (cs%id_S_preale > 0) call post_data(cs%id_S_preale, tv%S, cs%diag)
00357   if (cs%id_e_preale > 0) then
00358     call find_eta(h, tv, g, gv, us, eta_preale)
00359     call post_data(cs%id_e_preale, eta_preale, cs%diag)
00360   endif
00361
00362   if (present(dt)) then
00363     call ale_update_regrid_weights( dt, cs )
00364   endif
00365   dzregrid(:,:,:) = 0.0
00366
00367   ! Build new grid. The new grid is stored in h_new. The old grid is h.
00368   ! Both are needed for the subsequent remapping of variables.
00369   if (ice_shelf) then
00370      call regridding_main( cs%remapCS, cs%regridCS, g, gv, h, tv, h_new, dzregrid, frac_shelf_h)
00371   else
00372      call regridding_main( cs%remapCS, cs%regridCS, g, gv, h, tv, h_new, dzregrid)
00373   endif
00374
00375   call check_grid( g, gv, h, 0. )
00376
00377   if (cs%show_call_tree) call calltree_waypoint("new grid generated (ALE_main)")
00378
00379   ! The presence of dt is used for expediency to distinguish whether ALE_main is being called during init
00380   ! or in the main loop. Tendency diagnostics in remap_all_state_vars also rely on this logic.
00381   if (present(dt)) then
00382     call diag_update_remap_grids(cs%diag)
00383   endif
00384   ! Remap all variables from old grid h onto new grid h_new
00385   call remap_all_state_vars( cs%remapCS, cs, g, gv, h, h_new, reg, obc, -dzregrid, &
00386                              u, v, cs%show_call_tree, dt )
00387
00388   if (cs%show_call_tree) call calltree_waypoint("state remapped (ALE_main)")
00389
00390   ! Override old grid with new one. The new grid 'h_new' is built in
00391   ! one of the 'build_...' routines above.
00392   !$OMP parallel do default(shared)
00393   do k = 1,nk ; do j = jsc-1,jec+1 ; do i = isc-1,iec+1
00394     h(i,j,k) = h_new(i,j,k)
00395   enddo ; enddo ; enddo
00396
00397   if (cs%show_call_tree) call calltree_leave("ALE_main()")
00398
00399   if (cs%id_dzRegrid>0 .and. present(dt)) call post_data(cs%id_dzRegrid, dzregrid, cs%diag)
00400
00401
00402 end subroutine ale_main
00403
00404 !> Takes care of (1) building a new grid and (2) remapping all variables between
00405 !! the old grid and the new grid. The creation of the new grid can be based
00406 !! on z coordinates, target interface densities, sigma coordinates or any
```

```
00407 !! arbitrary coordinate system.
00408 subroutine ale_main_offline( G, GV, h, tv, Reg, CS, OBC, dt)
00409   type(ocean_grid_type),                      intent(in)    :: G   !< Ocean grid informations
00410   type(verticalgrid_type),                    intent(in)    :: GV  !< Ocean vertical grid structure
00411   real, dimension(SZI_(G),SZJ_(G),SZK_(GV)),  intent(inout) :: h   !< Current 3D grid obtained after the
00412                                                                      !! last time step [H ~> m or kg-2]
00413   type(thermo_var_ptrs),                      intent(inout) :: tv  !< Thermodynamic variable structure
00414   type(tracer_registry_type),                 pointer       :: Reg !< Tracer registry structure
00415   type(ale_cs),                               pointer       :: CS  !< Regridding parameters and
      options
00416   type(ocean_obc_type),                       pointer       :: OBC !< Open boundary structure
00417   real,                             optional, intent(in)    :: dt  !< Time step between calls to ALE_main
      [T ~> s]
00418   ! Local variables
00419   real, dimension(SZI_(G), SZJ_(G), SZK_(GV)+1) :: dzRegrid ! The change in grid interface positions
00420   real, dimension(SZI_(G),SZJ_(G),SZK_(GV)) :: h_new ! New 3D grid obtained after last time step [H ~> m or
      kg-2]
00421   integer :: nk, i, j, k, isc, iec, jsc, jec
00422
00423   nk = gv%ke; isc = g%isc; iec = g%iec; jsc = g%jsc; jec = g%jec
00424
00425   if (cs%show_call_tree) call calltree_enter("ALE_main_offline(), MOM_ALE.F90")
00426
00427   if (present(dt)) then
00428     call ale_update_regrid_weights( dt, cs )
00429   endif
00430   dzregrid(:,:,:) = 0.0
00431
00432   ! Build new grid. The new grid is stored in h_new. The old grid is h.
00433   ! Both are needed for the subsequent remapping of variables.
00434   call regridding_main( cs%remapCS, cs%regridCS, g, gv, h, tv, h_new, dzregrid )
00435
00436   call check_grid( g, gv, h, 0. )
00437
00438   if (cs%show_call_tree) call calltree_waypoint("new grid generated (ALE_main)")
00439
00440   ! Remap all variables from old grid h onto new grid h_new
00441
00442   call remap_all_state_vars(cs%remapCS, cs, g, gv, h, h_new, reg, obc, &
00443                             debug=cs%show_call_tree, dt=dt )
00444
00445   if (cs%show_call_tree) call calltree_waypoint("state remapped (ALE_main)")
00446
00447   ! Override old grid with new one. The new grid 'h_new' is built in
00448   ! one of the 'build_...' routines above.
00449   !$OMP parallel do default(shared)
00450   do k = 1,nk ; do j = jsc-1,jec+1 ; do i = isc-1,iec+1
00451     h(i,j,k) = h_new(i,j,k)
00452   enddo ; enddo ; enddo
00453
00454   if (cs%show_call_tree) call calltree_leave("ALE_main()")
00455   if (cs%id_dzRegrid>0 .and. present(dt)) call post_data(cs%id_dzRegrid, dzregrid, cs%diag)
00456
00457 end subroutine ale_main_offline
00458
00459 !> Regrid/remap stored fields used for offline tracer integrations. These input fields are assumed to have
00460 !! the same layer thicknesses at the end of the last offline interval (which should be a Zstar grid). This
00461 !! routine builds a grid on the runtime specified vertical coordinate
00462 subroutine ale_offline_inputs(CS, G, GV, h, tv, Reg, uhtr, vhtr, Kd, debug, OBC)
00463   type(ale_cs),                                pointer       :: CS    !< Regridding parameters and
      options
00464   type(ocean_grid_type),                       intent(in   ) :: G     !< Ocean grid informations
00465   type(verticalgrid_type),                     intent(in   ) :: GV    !< Ocean vertical grid structure
00466   real, dimension(SZI_(G),SZJ_(G),SZK_(GV)),   intent(inout) :: h     !< Layer thicknesses
00467   type(thermo_var_ptrs),                       intent(inout) :: tv    !< Thermodynamic variable structure
00468   type(tracer_registry_type),                  pointer       :: Reg   !< Tracer registry structure
00469   real, dimension(SZIB_(G),SZJ_(G),SZK_(GV)),  intent(inout) :: uhtr  !< Zonal mass fluxes
00470   real, dimension(SZI_(G),SZJB_(G),SZK_(GV)),  intent(inout) :: vhtr  !< Meridional mass fluxes
00471   real, dimension(SZI_(G),SZJ_(G),SZK_(GV)+1), intent(inout) :: Kd    !< Input diffusivites
00472   logical,                                     intent(in   ) :: debug !< If true, then turn checksums
00473   type(ocean_obc_type),                        pointer       :: OBC   !< Open boundary structure
00474   ! Local variables
00475   integer :: nk, i, j, k, isc, iec, jsc, jec
00476   real, dimension(SZI_(G), SZJ_(G), SZK_(GV))   :: h_new    ! Layer thicknesses after regridding
00477   real, dimension(SZI_(G), SZJ_(G), SZK_(GV)+1) :: dzRegrid ! The change in grid interface positions
00478   real, dimension(SZK_(GV)) :: h_src
00479   real, dimension(SZK_(GV)) :: h_dest, uh_dest
00480   real, dimension(SZK_(GV)) :: temp_vec
00481
00482   nk = gv%ke; isc = g%isc; iec = g%iec; jsc = g%jsc; jec = g%jec
00483   dzregrid(:,:,:) = 0.0
00484   h_new(:,:,:) = 0.0
00485
00486   if (debug) call mom_tracer_chkinv("Before ALE_offline_inputs", g, h, reg%Tr, reg%ntr)
00487
00488   ! Build new grid from the Zstar state onto the requested vertical coordinate. The new grid is stored
00489   ! in h_new. The old grid is h. Both are needed for the subsequent remapping of variables. Convective
```

```
00490   ! adjustment right now is not used because it is unclear what to do with vanished layers
00491   call regridding_main( cs%remapCS, cs%regridCS, g, gv, h, tv, h_new, dzregrid, conv_adjust = .false. )
00492   call check_grid( g, gv, h_new, 0. )
00493   if (cs%show_call_tree) call calltree_waypoint("new grid generated (ALE_offline_inputs)")
00494
00495   ! Remap all variables from old grid h onto new grid h_new
00496   call remap_all_state_vars( cs%remapCS, cs, g, gv, h, h_new, reg, obc, debug=cs
      %show_call_tree )
00497   if (cs%show_call_tree) call calltree_waypoint("state remapped (ALE_inputs)")
00498
00499   ! Reintegrate mass transports from Zstar to the offline vertical coordinate
00500   do j=jsc,jec ; do i=g%iscB,g%iecB
00501     if (g%mask2dCu(i,j)>0.) then
00502       h_src(:) = 0.5 * (h(i,j,:) + h(i+1,j,:))
00503       h_dest(:) = 0.5 * (h_new(i,j,:) + h_new(i+1,j,:))
00504       call reintegrate_column(nk, h_src, uhtr(i,j,:), nk, h_dest, 0., temp_vec)
00505       uhtr(i,j,:) = temp_vec
00506     endif
00507   enddo ; enddo
00508   do j=g%jscB,g%jecB ; do i=isc,iec
00509     if (g%mask2dCv(i,j)>0.) then
00510       h_src(:) = 0.5 * (h(i,j,:) + h(i,j+1,:))
00511       h_dest(:) = 0.5 * (h_new(i,j,:) + h_new(i,j+1,:))
00512       call reintegrate_column(nk, h_src, vhtr(i,j,:), nk, h_dest, 0., temp_vec)
00513       vhtr(i,j,:) = temp_vec
00514     endif
00515   enddo ; enddo
00516
00517   do j = jsc,jec ; do i=isc,iec
00518     if (g%mask2dT(i,j)>0.) then
00519       if (check_column_integrals(nk, h_src, nk, h_dest)) then
00520         call mom_error(fatal, "ALE_offline_inputs: Kd interpolation columns do not match")
00521       endif
00522       call interpolate_column(nk, h(i,j,:), kd(i,j,:), nk, h_new(i,j,:), 0., kd(i,j,:))
00523     endif
00524   enddo ; enddo
00525
00526   call ale_remap_scalar(cs%remapCS, g, gv, nk, h, tv%T, h_new, tv%T, answers_2018=cs
      %answers_2018)
00527   call ale_remap_scalar(cs%remapCS, g, gv, nk, h, tv%S, h_new, tv%S, answers_2018=cs
      %answers_2018)
00528
00529   if (debug) call mom_tracer_chkinv("After ALE_offline_inputs", g, h_new, reg%Tr, reg%ntr)
00530
00531   ! Copy over the new layer thicknesses
00532   do k = 1,nk  ; do j = jsc-1,jec+1 ; do i = isc-1,iec+1
00533       h(i,j,k) = h_new(i,j,k)
00534   enddo ; enddo ; enddo
00535
00536   if (cs%show_call_tree) call calltree_leave("ALE_offline_inputs()")
00537 end subroutine ale_offline_inputs
00538
00539
00540 !> Remaps all tracers from h onto h_target. This is intended to be called when tracers
00541 !! are done offline. In the case where transports don't quite conserve, we still want to
00542 !! make sure that layer thicknesses offline do not drift too far away from the online model
00543 subroutine ale_offline_tracer_final( G, GV, h, tv, h_target, Reg, CS, OBC)
00544   type(ocean_grid_type),                      intent(in)    :: G   !< Ocean grid informations
00545   type(verticalgrid_type),                    intent(in)    :: GV  !< Ocean vertical grid structure
00546   real, dimension(SZI_(G),SZJ_(G),SZK_(GV)),  intent(inout) :: h   !< Current 3D grid obtained after the
00547                                                                      !! last time step [H ~> m or kg-2]
00548   type(thermo_var_ptrs),                      intent(inout) :: tv  !< Thermodynamic variable structure
00549   real, dimension(SZI_(G),SZJ_(G),SZK_(GV)),  intent(inout) :: h_target !< Current 3D grid obtained after
00550                                                                          !! last time step  [H ~> m or kg-2]
00551   type(tracer_registry_type),                 pointer       :: Reg !< Tracer registry structure
00552   type(ale_cs),                               pointer       :: CS  !< Regridding parameters and
       options
00553   type(ocean_obc_type),                       pointer       :: OBC !< Open boundary structure
00554   ! Local variables
00555
00556   real, dimension(SZI_(G), SZJ_(G), SZK_(GV)+1) :: dzRegrid !< The change in grid interface positions
00557   real, dimension(SZI_(G), SZJ_(G), SZK_(GV))   :: h_new    !< Regridded target thicknesses
00558   integer :: nk, i, j, k, isc, iec, jsc, jec
00559
00560   nk = gv%ke; isc = g%isc; iec = g%iec; jsc = g%jsc; jec = g%jec
00561
00562   if (cs%show_call_tree) call calltree_enter("ALE_offline_tracer_final(), MOM_ALE.F90")
00563   ! Need to make sure that h_target is consistent with the current offline ALE confiuration
00564   call regridding_main( cs%remapCS, cs%regridCS, g, gv, h_target, tv, h_new, dzregrid )
00565   call check_grid( g, gv, h_target, 0. )
00566
00567
00568   if (cs%show_call_tree) call calltree_waypoint("Source and target grids checked
       (ALE_offline_tracer_final)")
00569
00570   ! Remap all variables from old grid h onto new grid h_new
00571
```

```
00572   call remap_all_state_vars( cs%remapCS, cs, g, gv, h, h_new, reg, obc, debug=cs
      %show_call_tree )
00573
00574   if (cs%show_call_tree) call calltree_waypoint("state remapped (ALE_offline_tracer_final)")
00575
00576   ! Override old grid with new one. The new grid 'h_new' is built in
00577   ! one of the 'build_...' routines above.
00578   !$OMP parallel do default(shared)
00579   do k = 1,nk
00580     do j = jsc-1,jec+1 ; do i = isc-1,iec+1
00581       h(i,j,k) = h_new(i,j,k)
00582     enddo ; enddo
00583   enddo
00584   if (cs%show_call_tree) call calltree_leave("ALE_offline_tracer_final()")
00585 end subroutine ale_offline_tracer_final
00586
00587 !> Check grid for negative thicknesses
00588 subroutine check_grid( G, GV, h, threshold )
00589   type(ocean_grid_type),                    intent(in) :: G  !< Ocean grid structure
00590   type(verticalgrid_type),                  intent(in) :: GV !< Ocean vertical grid structure
00591   real, dimension(SZI_(G),SZJ_(G),SZK_(GV)), intent(in) :: h  !< Current 3D grid obtained after the
00592                                                                !! last time step [H ~> m or kg m-2]
00593   real,                                     intent(in) :: threshold !< Value below which to flag issues,
00594                                                                !! [H ~> m or kg m-2]
00595   ! Local variables
00596   integer :: i, j
00597
00598   do j = g%jsc,g%jec ; do i = g%isc,g%iec
00599     if (g%mask2dT(i,j)>0.) then
00600       if (minval(h(i,j,:)) < threshold) then
00601         write(0,*) 'check_grid: i,j=',i,j,'h(i,j,:)=',h(i,j,:)
00602         if (threshold <= 0.) then
00603           call mom_error(fatal,"MOM_ALE, check_grid: negative thickness encountered.")
00604         else
00605           call mom_error(fatal,"MOM_ALE, check_grid: too tiny thickness encountered.")
00606         endif
00607       endif
00608     endif
00609   enddo ; enddo
00610
00611
00612 end subroutine check_grid
00613
00614 !> Generates new grid
00615 subroutine ale_build_grid( G, GV, regridCS, remapCS, h, tv, debug, frac_shelf_h )
00616   type(ocean_grid_type),                    intent(in)    :: G        !< Ocean grid structure
00617   type(verticalgrid_type),                  intent(in)    :: GV       !< Ocean vertical grid structure
00618   type(regridding_cs),                      intent(in)    :: regridCS !< Regridding parameters and options
00619   type(remapping_cs),                       intent(in)    :: remapCS  !< Remapping parameters and options
00620   type(thermo_var_ptrs),                    intent(inout) :: tv       !< Thermodynamical variable structure
00621   real, dimension(SZI_(G),SZJ_(G), SZK_(GV)), intent(inout) :: h      !< Current 3D grid obtained after the
00622                                                                !! last time step [H ~> m or kg-2]
00623   logical,                        optional, intent(in)    :: debug    !< If true, show the call tree
00624   real, dimension(:,:),           optional, pointer       :: frac_shelf_h !< Fractional ice shelf coverage
00625   ! Local variables
00626   integer :: nk, i, j, k
00627   real, dimension(SZI_(G), SZJ_(G), SZK_(GV)+1) :: dzRegrid ! The change in grid interface positions
00628   real, dimension(SZI_(G), SZJ_(G), SZK_(GV)) :: h_new ! The new grid thicknesses
00629   logical :: show_call_tree, use_ice_shelf
00630
00631   show_call_tree = .false.
00632   if (present(debug)) show_call_tree = debug
00633   if (show_call_tree) call calltree_enter("ALE_build_grid(), MOM_ALE.F90")
00634   use_ice_shelf = .false.
00635   if (present(frac_shelf_h)) then
00636     if (associated(frac_shelf_h)) use_ice_shelf = .true.
00637   endif
00638
00639   ! Build new grid. The new grid is stored in h_new. The old grid is h.
00640   ! Both are needed for the subsequent remapping of variables.
00641   if (use_ice_shelf) then
00642     call regridding_main( remapcs, regridcs, g, gv, h, tv, h_new, dzregrid, frac_shelf_h )
00643   else
00644     call regridding_main( remapcs, regridcs, g, gv, h, tv, h_new, dzregrid )
00645   endif
00646
00647   ! Override old grid with new one. The new grid 'h_new' is built in
00648   ! one of the 'build_...' routines above.
00649 !$OMP parallel do default(none) shared(G,h,h_new)
00650   do j = g%jsc,g%jec ; do i = g%isc,g%iec
00651     if (g%mask2dT(i,j)>0.) h(i,j,:) = h_new(i,j,:)
00652   enddo ; enddo
00653
00654   if (show_call_tree) call calltree_leave("ALE_build_grid()")
00655 end subroutine ale_build_grid
00656
00657 !> For a state-based coordinate, accelerate the process of regridding by
```

```
00658 !! repeatedly applying the grid calculation algorithm
00659 subroutine ale_regrid_accelerated(CS, G, GV, h, tv, n, u, v, OBC, Reg, dt, dzRegrid,
       initial)
00660   type(ale_cs),            pointer       :: CS     !< ALE control structure
00661   type(ocean_grid_type),   intent(inout) :: G      !< Ocean grid
00662   type(verticalgrid_type), intent(in)    :: GV     !< Vertical grid
00663   real, dimension(SZI_(G),SZJ_(G),SZK_(GV)), &
00664                            intent(inout) :: h      !< Original thicknesses [H ~> m or kg-2]
00665   type(thermo_var_ptrs),   intent(inout) :: tv     !< Thermo vars (T/S/EOS)
00666   integer,                 intent(in)    :: n      !< Number of times to regrid
00667   real, dimension(SZIB_(G),SZJ_(G),SZK_(GV)), &
00668                            intent(inout) :: u      !< Zonal velocity [L T-1 ~> m s-1]
00669   real, dimension(SZI_(G),SZJB_(G),SZK_(GV)), &
00670                            intent(inout) :: v      !< Meridional velocity [L T-1 ~> m s-1]
00671   type(ocean_obc_type),    pointer       :: OBC    !< Open boundary structure
00672   type(tracer_registry_type), &
00673                  optional, pointer       :: Reg    !< Tracer registry to remap onto new grid
00674   real,          optional, intent(in)    :: dt     !< Model timestep to provide a timescale for regridding
       [T ~> s]
00675   real, dimension(SZI_(G),SZJ_(G),SZK_(GV)+1), &
00676                  optional, intent(inout) :: dzRegrid !< Final change in interface positions
00677   logical,       optional, intent(in)    :: initial !< Whether we're being called from an initialization
00678                                                     !! routine (and expect diagnostics to work)
00679
00680   ! Local variables
00681   integer :: i, j, k, nz
00682   type(thermo_var_ptrs) :: tv_local ! local/intermediate temp/salt
00683   type(group_pass_type) :: pass_T_S_h ! group pass if the coordinate has a stencil
00684   real, dimension(SZI_(G),SZJ_(G),SZK_(GV))         :: h_loc, h_orig ! A working copy of layer thicknesses
00685   real, dimension(SZI_(G),SZJ_(G),SZK_(GV)), target :: T, S ! local temporary state
00686   ! we have to keep track of the total dzInterface if for some reason
00687   ! we're using the old remapping algorithm for u/v
00688   real, dimension(SZI_(G),SZJ_(G),SZK_(GV)+1) :: dzInterface, dzIntTotal
00689
00690   nz = gv%ke
00691
00692   ! initial total interface displacement due to successive regridding
00693   dzinttotal(:,:,:) = 0.
00694
00695   call create_group_pass(pass_t_s_h, t, g%domain)
00696   call create_group_pass(pass_t_s_h, s, g%domain)
00697   call create_group_pass(pass_t_s_h, h_loc, g%domain)
00698
00699   ! copy original temp/salt and set our local tv_pointers to them
00700   tv_local = tv
00701   t(:,:,:) = tv%T(:,:,:)
00702   s(:,:,:) = tv%S(:,:,:)
00703   tv_local%T => t
00704   tv_local%S => s
00705
00706   ! get local copy of thickness and save original state for remapping
00707   h_loc(:,:,:) = h(:,:,:)
00708   h_orig(:,:,:) = h(:,:,:)
00709
00710   ! Apply timescale to regridding (for e.g. filtered_grid_motion)
00711   if (present(dt)) &
00712     call ale_update_regrid_weights(dt, cs)
00713
00714   do k = 1, n
00715     call do_group_pass(pass_t_s_h, g%domain)
00716
00717     ! generate new grid
00718     call regridding_main(cs%remapCS, cs%regridCS, g, gv, h_loc, tv_local, h, dzinterface)
00719     dzinttotal(:,:,:) = dzinttotal(:,:,:) + dzinterface(:,:,:)
00720
00721     ! remap from original grid onto new grid
00722     do j = g%jsc-1,g%jec+1 ; do i = g%isc-1,g%iec+1
00723       call remapping_core_h(cs%remapCS, nz, h_orig(i,j,:), tv%S(i,j,:), nz, h(i,j,:), tv_local%S(i,j,:))
00724       call remapping_core_h(cs%remapCS, nz, h_orig(i,j,:), tv%T(i,j,:), nz, h(i,j,:), tv_local%T(i,j,:))
00725     enddo ; enddo
00726
00727     ! starting grid for next iteration
00728     h_loc(:,:,:) = h(:,:,:)
00729   enddo
00730
00731   ! remap all state variables (including those that weren't needed for regridding)
00732   call remap_all_state_vars(cs%remapCS, cs, g, gv, h_orig, h, reg, obc, dzinttotal, u,
      v)
00733
00734   ! save total dzregrid for diags if needed?
00735   if (present(dzregrid)) dzregrid(:,:,:) = dzinttotal(:,:,:)
00736 end subroutine ale_regrid_accelerated
00737
00738 !> This routine takes care of remapping all variable between the old and the
00739 !! new grids. When velocity components need to be remapped, thicknesses at
00740 !! velocity points are taken to be arithmetic averages of tracer thicknesses.
00741 !! This routine is called during initialization of the model at time=0, to
```

```
00742 !! remap initiali conditions to the model grid.  It is also called during a
00743 !! time step to update the state.
00744 subroutine remap_all_state_vars(CS_remapping, CS_ALE, G, GV, h_old, h_new, Reg, OBC, &
00745                                 dxInterface, u, v, debug, dt)
00746   type(remapping_cs),                         intent(in)    :: CS_remapping !< Remapping control structure
00747   type(ale_cs),                               intent(in)    :: CS_ALE       !< ALE control structure
00748   type(ocean_grid_type),                      intent(in)    :: G            !< Ocean grid structure
00749   type(verticalgrid_type),                    intent(in)    :: GV           !< Ocean vertical grid structure
00750   real, dimension(SZI_(G),SZJ_(G),SZK_(GV)), intent(in)    :: h_old        !< Thickness of source grid
00751                                                                             !! [H ~> m or kg-2]
00752   real, dimension(SZI_(G),SZJ_(G),SZK_(GV)), intent(in)    :: h_new        !< Thickness of destination grid
00753                                                                             !! [H ~> m or kg-2]
00754   type(tracer_registry_type),                 pointer       :: Reg          !< Tracer registry structure
00755   type(ocean_obc_type),                       pointer       :: OBC          !< Open boundary structure
00756   real, dimension(SZI_(G),SZJ_(G),SZK_(GV)+1), &
00757                                               optional, intent(in)    :: dxInterface  !< Change in interface position
00758                                                                             !! [H ~> m or kg-2]
00759   real, dimension(SZIB_(G),SZJ_(G),SZK_(GV)), &
00760                                               optional, intent(inout) :: u      !< Zonal velocity [L T-1 ~> m s-1]
00761   real, dimension(SZI_(G),SZJB_(G),SZK_(GV)), &
00762                                               optional, intent(inout) :: v      !< Meridional velocity [L T-1 ~> m
      s-1]
00763   logical,                                    optional, intent(in)    :: debug  !< If true, show the call tree
00764   real,                                       optional, intent(in)    :: dt     !< time step for diagnostics [T ~> s]
00765   ! Local variables
00766   integer                                               :: i, j, k, m
00767   integer                                               :: nz, ntr
00768   real, dimension(GV%ke+1)                              :: dx
00769   real, dimension(GV%ke)                                :: h1, u_column
00770   real, dimension(SZI_(G), SZJ_(G), SZK_(GV))           :: work_conc
00771   real, dimension(SZI_(G), SZJ_(G), SZK_(GV))           :: work_cont
00772   real, dimension(SZI_(G), SZJ_(G))                     :: work_2d
00773   real                                                  :: Idt ! The inverse of the timestep [T-1 ~> s-1]
00774   real                                                  :: ppt2mks
00775   real, dimension(GV%ke)                                :: h2
00776   real :: h_neglect, h_neglect_edge
00777   logical                                               :: show_call_tree
00778   type(tracer_type), pointer                            :: Tr => null()
00779
00780   show_call_tree = .false.
00781   if (present(debug)) show_call_tree = debug
00782   if (show_call_tree) call calltree_enter("remap_all_state_vars(), MOM_ALE.F90")
00783
00784   ! If remap_uv_using_old_alg is .true. and u or v is requested, then we must have dxInterface. Otherwise,
00785   ! u and v can be remapped without dxInterface
00786   if ( .not. present(dxinterface) .and. (cs_ale%remap_uv_using_old_alg .and. (present(u) .or. present(v)))
      ) then
00787     call mom_error(fatal, "remap_all_state_vars: dxInterface must be present if using old algorithm "// &
00788                           "and u/v are to be remapped")
00789   endif
00790
00791   if (.not.cs_ale%answers_2018) then
00792     h_neglect = gv%H_subroundoff ; h_neglect_edge = gv%H_subroundoff
00793   elseif (gv%Boussinesq) then
00794     h_neglect = gv%m_to_H*1.0e-30 ; h_neglect_edge = gv%m_to_H*1.0e-10
00795   else
00796     h_neglect = gv%kg_m2_to_H*1.0e-30 ; h_neglect_edge = gv%kg_m2_to_H*1.0e-10
00797   endif
00798
00799   nz      = gv%ke
00800   ppt2mks = 0.001
00801
00802   ntr = 0 ; if (associated(reg)) ntr = reg%ntr
00803
00804   if (present(dt)) then
00805     idt = 1.0/dt
00806     work_conc(:,:,:) = 0.0
00807     work_cont(:,:,:) = 0.0
00808   endif
00809
00810   ! Remap tracer
00811   if (ntr>0) then
00812     if (show_call_tree) call calltree_waypoint("remapping tracers (remap_all_state_vars)")
00813     !$OMP parallel do default(shared) private(h1,h2,u_column,Tr)
00814     do m=1,ntr ! For each tracer
00815       tr => reg%Tr(m)
00816       do j = g%jsc,g%jec ; do i = g%isc,g%iec ; if (g%mask2dT(i,j)>0.) then
00817         ! Build the start and final grids
00818         h1(:) = h_old(i,j,:)
00819         h2(:) = h_new(i,j,:)
00820         call remapping_core_h(cs_remapping, nz, h1, tr%t(i,j,:), nz, h2, &
00821                               u_column, h_neglect, h_neglect_edge)
00822
00823         ! Intermediate steps for tendency of tracer concentration and tracer content.
00824         if (present(dt)) then
00825           if (tr%id_remap_conc > 0) then
00826             do k=1,gv%ke
```

```
00827               work_conc(i,j,k) = (u_column(k) - tr%t(i,j,k)) * idt
00828             enddo
00829           endif
00830           if (tr%id_remap_cont > 0 .or. tr%id_remap_cont_2d > 0) then
00831             do k=1,gv%ke
00832               work_cont(i,j,k) = (u_column(k)*h2(k) - tr%t(i,j,k)*h1(k)) * idt
00833             enddo
00834           endif
00835         endif
00836         ! update tracer concentration
00837         tr%t(i,j,:) = u_column(:)
00838       endif ; enddo ; enddo
00839
00840       ! tendency diagnostics.
00841       if (present(dt)) then
00842         if (tr%id_remap_conc > 0) then
00843           call post_data(tr%id_remap_conc, work_conc, cs_ale%diag)
00844         endif
00845         if (tr%id_remap_cont > 0) then
00846           call post_data(tr%id_remap_cont, work_cont, cs_ale%diag)
00847         endif
00848         if (tr%id_remap_cont_2d > 0) then
00849           do j = g%jsc,g%jec ; do i = g%isc,g%iec
00850             work_2d(i,j) = 0.0
00851             do k = 1,gv%ke
00852               work_2d(i,j) = work_2d(i,j) + work_cont(i,j,k)
00853             enddo
00854           enddo ; enddo
00855           call post_data(tr%id_remap_cont_2d, work_2d, cs_ale%diag)
00856         endif
00857       endif
00858     enddo ! m=1,ntr
00859
00860   endif   ! endif for ntr > 0
00861
00862   if (show_call_tree) call calltree_waypoint("tracers remapped (remap_all_state_vars)")
00863
00864   ! Remap u velocity component
00865   if ( present(u) ) then
00866     !$OMP parallel do default(shared) private(h1,h2,dx,u_column)
00867     do j = g%jsc,g%jec ; do i = g%iscB,g%iecB ; if (g%mask2dCu(i,j)>0.) then
00868       ! Build the start and final grids
00869       h1(:) = 0.5 * ( h_old(i,j,:) + h_old(i+1,j,:) )
00870       if (cs_ale%remap_uv_using_old_alg) then
00871         dx(:) = 0.5 * ( dxinterface(i,j,:) + dxinterface(i+1,j,:) )
00872         do k = 1, nz
00873           h2(k) = max( 0., h1(k) + ( dx(k+1) - dx(k) ) )
00874         enddo
00875       else
00876         h2(:) = 0.5 * ( h_new(i,j,:) + h_new(i+1,j,:) )
00877       endif
00878       if (associated(obc)) then
00879         if (obc%segnum_u(i,j) .ne. 0) then
00880           if (obc%segment(obc%segnum_u(i,j))%direction == obc_direction_e) then
00881             h1(:) = h_old(i,j,:)
00882             h2(:) = h_new(i,j,:)
00883           else ! (OBC%segment(n)%direction == OBC_DIRECTION_W)
00884             h1(:) = h_old(i+1,j,:)
00885             h2(:) = h_new(i+1,j,:)
00886           endif
00887         endif
00888       endif
00889       call remapping_core_h(cs_remapping, nz, h1, u(i,j,:), nz, h2, &
00890                             u_column, h_neglect, h_neglect_edge)
00891       u(i,j,:) = u_column(:)
00892     endif ; enddo ; enddo
00893   endif
00894
00895   if (show_call_tree) call calltree_waypoint("u remapped (remap_all_state_vars)")
00896
00897   ! Remap v velocity component
00898   if ( present(v) ) then
00899     !$OMP parallel do default(shared) private(h1,h2,dx,u_column)
00900     do j = g%jscB,g%jecB ; do i = g%isc,g%iec ; if (g%mask2dCv(i,j)>0.) then
00901       ! Build the start and final grids
00902       h1(:) = 0.5 * ( h_old(i,j,:) + h_old(i,j+1,:) )
00903       if (cs_ale%remap_uv_using_old_alg) then
00904         dx(:) = 0.5 * ( dxinterface(i,j,:) + dxinterface(i,j+1,:) )
00905         do k = 1, nz
00906           h2(k) = max( 0., h1(k) + ( dx(k+1) - dx(k) ) )
00907         enddo
00908       else
00909         h2(:) = 0.5 * ( h_new(i,j,:) + h_new(i,j+1,:) )
00910       endif
00911       if (associated(obc)) then
00912         if (obc%segnum_v(i,j) .ne. 0) then
00913           if (obc%segment(obc%segnum_v(i,j))%direction == obc_direction_n) then
```

```
00914             h1(:) = h_old(i,j,:)
00915             h2(:) = h_new(i,j,:)
00916           else ! (OBC%segment(n)%direction == OBC_DIRECTION_S)
00917             h1(:) = h_old(i,j+1,:)
00918             h2(:) = h_new(i,j+1,:)
00919           endif
00920         endif
00921       endif
00922       call remapping_core_h(cs_remapping, nz, h1, v(i,j,:), nz, h2, &
00923                             u_column, h_neglect, h_neglect_edge)
00924       v(i,j,:) = u_column(:)
00925     endif ; enddo ; enddo
00926   endif
00927
00928   if (cs_ale%id_vert_remap_h > 0) call post_data(cs_ale%id_vert_remap_h, h_old, cs_ale%diag)
00929   if ((cs_ale%id_vert_remap_h_tendency > 0) .and. present(dt)) then
00930     do k = 1, nz ; do j = g%jsc,g%jec ; do i = g%isc,g%iec
00931       work_cont(i,j,k) = (h_new(i,j,k) - h_old(i,j,k))*idt
00932     enddo ; enddo ; enddo
00933     call post_data(cs_ale%id_vert_remap_h_tendency, work_cont, cs_ale%diag)
00934   endif
00935   if (show_call_tree) call calltree_waypoint("v remapped (remap_all_state_vars)")
00936   if (show_call_tree) call calltree_leave("remap_all_state_vars()")
00937
00938 end subroutine remap_all_state_vars
00939
00940
00941 !> Remaps a single scalar between grids described by thicknesses h_src and h_dst.
00942 !! h_dst must be dimensioned as a model array with GV%ke layers while h_src can
00943 !! have an arbitrary number of layers specified by nk_src.
00944 subroutine ale_remap_scalar(CS, G, GV, nk_src, h_src, s_src, h_dst, s_dst, all_cells,
      old_remap, answers_2018 )
00945   type(remapping_cs),                     intent(in)    :: CS        !< Remapping control structure
00946   type(ocean_grid_type),                  intent(in)    :: G         !< Ocean grid structure
00947   type(verticalgrid_type),                intent(in)    :: GV        !< Ocean vertical grid structure
00948   integer,                                intent(in)    :: nk_src    !< Number of levels on source grid
00949   real, dimension(SZI_(G),SZJ_(G),nk_src), intent(in)    :: h_src     !< Level thickness of source grid
00950                                                                        !! [H ~> m or kg-2]
00951   real, dimension(SZI_(G),SZJ_(G),nk_src), intent(in)    :: s_src     !< Scalar on source grid
00952   real, dimension(SZI_(G),SZJ_(G),SZK_(GV)),intent(in)   :: h_dst     !< Level thickness of destination
      grid
00953                                                                        !! [H ~> m or kg-2]
00954   real, dimension(SZI_(G),SZJ_(G),SZK_(GV)),intent(inout) :: s_dst    !< Scalar on destination grid
00955   logical, optional,                      intent(in)    :: all_cells !< If false, only reconstruct for
00956                                                                        !! non-vanished cells. Use all
      vanished
00957                                                                        !! layers otherwise (default).
00958   logical, optional,                      intent(in)    :: old_remap !< If true, use the old
      "remapping_core_w"
00959                                                                        !! method, otherwise use
      "remapping_core_h".
00960   logical,                       optional, intent(in)    :: answers_2018 !< If true, use the order of
      arithmetic
00961                                                                        !! and expressions that recover the
      answers for
00962                                                                        !! remapping from the end of 2018.
      Otherwise,
00963                                                                        !! use more robust forms of the same
      expressions.
00964   ! Local variables
00965   integer :: i, j, k, n_points
00966   real :: dx(gv%ke+1)
00967   real :: h_neglect, h_neglect_edge
00968   logical :: ignore_vanished_layers, use_remapping_core_w, use_2018_remap
00969
00970   ignore_vanished_layers = .false.
00971   if (present(all_cells)) ignore_vanished_layers = .not. all_cells
00972   use_remapping_core_w = .false.
00973   if (present(old_remap)) use_remapping_core_w = old_remap
00974   n_points = nk_src
00975   use_2018_remap = .true. ; if (present(answers_2018)) use_2018_remap = answers_2018
00976
00977   if (.not.use_2018_remap) then
00978     h_neglect = gv%H_subroundoff ; h_neglect_edge = gv%H_subroundoff
00979   elseif (gv%Boussinesq) then
00980     h_neglect = gv%m_to_H*1.0e-30 ; h_neglect_edge = gv%m_to_H*1.0e-10
00981   else
00982     h_neglect = gv%kg_m2_to_H*1.0e-30 ; h_neglect_edge = gv%kg_m2_to_H*1.0e-10
00983   endif
00984
00985   !$OMP parallel do default(shared) firstprivate(n_points,dx)
00986   do j = g%jsc,g%jec ; do i = g%isc,g%iec
00987     if (g%mask2dT(i,j) > 0.) then
00988       if (ignore_vanished_layers) then
00989         n_points = 0
00990         do k = 1, nk_src
00991           if (h_src(i,j,k)>0.) n_points = n_points + 1
```

```
00992         enddo
00993         s_dst(i,j,:) = 0.
00994       endif
00995       if (use_remapping_core_w) then
00996         call dzfromh1h2( n_points, h_src(i,j,1:n_points), gv%ke, h_dst(i,j,:), dx )
00997         call remapping_core_w(cs, n_points, h_src(i,j,1:n_points), s_src(i,j,1:n_points), &
00998                               gv%ke, dx, s_dst(i,j,:), h_neglect, h_neglect_edge)
00999       else
01000         call remapping_core_h(cs, n_points, h_src(i,j,1:n_points), s_src(i,j,1:n_points), &
01001                               gv%ke, h_dst(i,j,:), s_dst(i,j,:), h_neglect, h_neglect_edge)
01002       endif
01003     else
01004       s_dst(i,j,:) = 0.
01005     endif
01006   enddo ; enddo
01007
01008 end subroutine ale_remap_scalar
01009
01010
01011 !> Calculate edge values (top and bottom of layer) for T and S consistent with a PLM reconstruction
01012 !! in the vertical direction. Boundary reconstructions are PCM unless bdry_extrap is true.
01013 subroutine ts_plm_edge_values( CS, S_t, S_b, T_t, T_b, G, GV, tv, h, bdry_extrap )
01014   type(ocean_grid_type),   intent(in)    :: G    !< ocean grid structure
01015   type(verticalgrid_type), intent(in)    :: GV   !< Ocean vertical grid structure
01016   type(ale_cs),            intent(inout) :: CS   !< module control structure
01017   real, dimension(SZI_(G),SZJ_(G),SZK_(GV)), &
01018                            intent(inout) :: S_t  !< Salinity at the top edge of each layer
01019   real, dimension(SZI_(G),SZJ_(G),SZK_(GV)), &
01020                            intent(inout) :: S_b  !< Salinity at the bottom edge of each layer
01021   real, dimension(SZI_(G),SZJ_(G),SZK_(GV)), &
01022                            intent(inout) :: T_t  !< Temperature at the top edge of each layer
01023   real, dimension(SZI_(G),SZJ_(G),SZK_(GV)), &
01024                            intent(inout) :: T_b  !< Temperature at the bottom edge of each layer
01025   type(thermo_var_ptrs),   intent(in)    :: tv   !< thermodynamics structure
01026   real, dimension(SZI_(G),SZJ_(G),SZK_(GV)), &
01027                            intent(in)    :: h    !< layer thickness [H ~> m or kg m-2]
01028   logical,                 intent(in)    :: bdry_extrap !< If true, use high-order boundary
01029                                                 !! extrapolation within boundary cells
01030
01031   call ale_plm_edge_values( cs, g, gv, h, tv%S, bdry_extrap, s_t, s_b )
01032   call ale_plm_edge_values( cs, g, gv, h, tv%T, bdry_extrap, t_t, t_b )
01033
01034 end subroutine ts_plm_edge_values
01035
01036 !> Calculate edge values (top and bottom of layer) 3d scalar array.
01037 !! Boundary reconstructions are PCM unless bdry_extrap is true.
01038 subroutine ale_plm_edge_values( CS, G, GV, h, Q, bdry_extrap, Q_t, Q_b )
01039   type(ale_cs),            intent(in)    :: CS   !< module control structure
01040   type(ocean_grid_type),   intent(in)    :: G    !< ocean grid structure
01041   type(verticalgrid_type), intent(in)    :: GV   !< Ocean vertical grid structure
01042   real, dimension(SZI_(G),SZJ_(G),SZK_(GV)), &
01043                            intent(in)    :: h    !< layer thickness [H ~> m or kg m-2]
01044   real, dimension(SZI_(G),SZJ_(G),SZK_(GV)), &
01045                            intent(in)    :: Q    !< 3d scalar array
01046   logical,                 intent(in)    :: bdry_extrap !< If true, use high-order boundary
01047                                                 !! extrapolation within boundary cells
01048   real, dimension(SZI_(G),SZJ_(G),SZK_(GV)), &
01049                            intent(inout) :: Q_t  !< Scalar at the top edge of each layer
01050   real, dimension(SZI_(G),SZJ_(G),SZK_(GV)), &
01051                            intent(inout) :: Q_b  !< Scalar at the bottom edge of each layer
01052   ! Local variables
01053   integer :: i, j, k
01054   real :: slp(gv%ke)
01055   real :: mslp
01056   real :: h_neglect
01057
01058   if (.not.cs%answers_2018) then
01059     h_neglect = gv%H_subroundoff
01060   elseif (gv%Boussinesq) then
01061     h_neglect = gv%m_to_H*1.0e-30
01062   else
01063     h_neglect = gv%kg_m2_to_H*1.0e-30
01064   endif
01065
01066   !$OMP parallel do default(shared) private(slp,mslp)
01067   do j = g%jsc-1,g%jec+1 ; do i = g%isc-1,g%iec+1
01068     slp(1) = 0.
01069     do k = 2, gv%ke-1
01070       slp(k) = plm_slope_wa(h(i,j,k-1), h(i,j,k), h(i,j,k+1), h_neglect, q(i,j,k-1), q(i,j,k), q(i,j,k+1))
01071     enddo
01072     slp(gv%ke) = 0.
01073
01074     do k = 2, gv%ke-1
01075       mslp = plm_monotonized_slope(q(i,j,k-1), q(i,j,k), q(i,j,k+1), slp(k-1), slp(k), slp(k+1))
01076       q_t(i,j,k) = q(i,j,k) - 0.5 * mslp
01077       q_b(i,j,k) = q(i,j,k) + 0.5 * mslp
01078     enddo
```

```
01079     if (bdry_extrap) then
01080       mslp = - plm_extrapolate_slope(h(i,j,2), h(i,j,1), h_neglect, q(i,j,2), q(i,j,1))
01081       q_t(i,j,1) = q(i,j,1) - 0.5 * mslp
01082       q_b(i,j,1) = q(i,j,1) + 0.5 * mslp
01083       mslp = plm_extrapolate_slope(h(i,j,gv%ke-1), h(i,j,gv%ke), h_neglect, q(i,j,gv%ke-1), q(i,j,gv%ke))
01084       q_t(i,j,gv%ke) = q(i,j,gv%ke) - 0.5 * mslp
01085       q_b(i,j,gv%ke) = q(i,j,gv%ke) + 0.5 * mslp
01086     else
01087       q_t(i,j,1) = q(i,j,1)
01088       q_b(i,j,1) = q(i,j,1)
01089       q_t(i,j,gv%ke) = q(i,j,gv%ke)
01090       q_b(i,j,gv%ke) = q(i,j,gv%ke)
01091     endif
01092
01093   enddo ; enddo
01094
01095 end subroutine ale_plm_edge_values
01096
01097 !> Calculate edge values (top and bottom of layer) for T and S consistent with a PPM reconstruction
01098 !! in the vertical direction. Boundary reconstructions are PCM unless bdry_extrap is true.
01099 subroutine ts_ppm_edge_values( CS, S_t, S_b, T_t, T_b, G, GV, tv, h, bdry_extrap )
01100   type(ocean_grid_type),   intent(in)    :: G    !< ocean grid structure
01101   type(verticalgrid_type), intent(in)    :: GV   !< Ocean vertical grid structure
01102   type(ale_cs),            intent(inout) :: CS   !< module control structure
01103   real, dimension(SZI_(G),SZJ_(G),SZK_(GV)), &
01104                            intent(inout) :: S_t  !< Salinity at the top edge of each layer
01105   real, dimension(SZI_(G),SZJ_(G),SZK_(GV)), &
01106                            intent(inout) :: S_b  !< Salinity at the bottom edge of each layer
01107   real, dimension(SZI_(G),SZJ_(G),SZK_(GV)), &
01108                            intent(inout) :: T_t  !< Temperature at the top edge of each layer
01109   real, dimension(SZI_(G),SZJ_(G),SZK_(GV)), &
01110                            intent(inout) :: T_b  !< Temperature at the bottom edge of each layer
01111   type(thermo_var_ptrs),   intent(in)    :: tv   !< thermodynamics structure
01112   real, dimension(SZI_(G),SZJ_(G),SZK_(GV)), &
01113                            intent(in)    :: h    !< layer thicknesses [H ~> m or kg m-2]
01114   logical,                 intent(in)    :: bdry_extrap !< If true, use high-order boundary
01115                                                 !! extrapolation within boundary cells
01116
01117   ! Local variables
01118   integer :: i, j, k
01119   real    :: hTmp(gv%ke) ! A 1-d copy of h [H ~> m or kg m-2]
01120   real    :: tmp(gv%ke)  ! A 1-d copy of a column of temperature [degC] or salinity [ppt]
01121   real, dimension(CS%nk,2) :: &
01122       ppol_E            ! Edge value of polynomial in [degC] or [ppt]
01123   real, dimension(CS%nk,3) :: &
01124       ppol_coefs        ! Coefficients of polynomial, all in [degC] or [ppt]
01125   real :: h_neglect, h_neglect_edge ! Tiny thicknesses [H ~> m or kg m-2]
01126
01127   if (.not.cs%answers_2018) then
01128     h_neglect = gv%H_subroundoff ; h_neglect_edge = gv%H_subroundoff
01129   elseif (gv%Boussinesq) then
01130     h_neglect = gv%m_to_H*1.0e-30 ; h_neglect_edge = gv%m_to_H*1.0e-10
01131   else
01132     h_neglect = gv%kg_m2_to_H*1.0e-30 ; h_neglect_edge = gv%kg_m2_to_H*1.0e-10
01133   endif
01134
01135   ! Determine reconstruction within each column
01136   !$OMP parallel do default(shared) private(hTmp,tmp,ppol_E,ppol_coefs)
01137   do j = g%jsc-1,g%jec+1 ; do i = g%isc-1,g%iec+1
01138
01139     ! Build current grid
01140     htmp(:) = h(i,j,:)
01141     tmp(:) = tv%S(i,j,:)
01142
01143     ! Reconstruct salinity profile
01144     ppol_e(:,:) = 0.0
01145     ppol_coefs(:,:) = 0.0
01146     call edge_values_implicit_h4( gv%ke, htmp, tmp, ppol_e, h_neglect=h_neglect_edge, &
01147                                   answers_2018=cs%answers_2018 )
01148     call ppm_reconstruction( gv%ke, htmp, tmp, ppol_e, ppol_coefs, h_neglect, &
01149                                   answers_2018=cs%answers_2018 )
01150     if (bdry_extrap) &
01151       call ppm_boundary_extrapolation( gv%ke, htmp, tmp, ppol_e, ppol_coefs, h_neglect )
01152
01153     do k = 1,gv%ke
01154       s_t(i,j,k) = ppol_e(k,1)
01155       s_b(i,j,k) = ppol_e(k,2)
01156     enddo
01157
01158     ! Reconstruct temperature profile
01159     ppol_e(:,:) = 0.0
01160     ppol_coefs(:,:) = 0.0
01161     tmp(:) = tv%T(i,j,:)
01162     if (cs%answers_2018) then
01163       call edge_values_implicit_h4( gv%ke, htmp, tmp, ppol_e, h_neglect=1.0e-10*gv%m_to_H, &
01164                                     answers_2018=cs%answers_2018 )
01165     else
```

```
01166       call edge_values_implicit_h4( gv%ke, htmp, tmp, ppol_e, h_neglect=gv%H_subroundoff, &
01167                                     answers_2018=cs%answers_2018 )
01168     endif
01169     call ppm_reconstruction( gv%ke, htmp, tmp, ppol_e, ppol_coefs, h_neglect, &
01170                                     answers_2018=cs%answers_2018 )
01171     if (bdry_extrap) &
01172       call ppm_boundary_extrapolation(gv%ke, htmp, tmp, ppol_e, ppol_coefs, h_neglect )
01173
01174     do k = 1,gv%ke
01175       t_t(i,j,k) = ppol_e(k,1)
01176       t_b(i,j,k) = ppol_e(k,2)
01177     enddo
01178
01179   enddo ; enddo
01180
01181 end subroutine ts_ppm_edge_values
01182
01183
01184 !> Initializes regridding for the main ALE algorithm
01185 subroutine ale_initregridding(GV, US, max_depth, param_file, mdl, regridCS)
01186   type(verticalgrid_type), intent(in)  :: GV         !< Ocean vertical grid structure
01187   type(unit_scale_type),   intent(in)  :: US         !< A dimensional unit scaling type
01188   real,                    intent(in)  :: max_depth  !< The maximum depth of the ocean [Z ~> m].
01189   type(param_file_type),   intent(in)  :: param_file !< parameter file
01190   character(len=*),        intent(in)  :: mdl        !< Name of calling module
01191   type(regridding_cs),     intent(out) :: regridCS   !< Regridding parameters and work arrays
01192   ! Local variables
01193   character(len=30) :: coord_mode
01194
01195   call get_param(param_file, mdl, "REGRIDDING_COORDINATE_MODE", coord_mode, &
01196                  "Coordinate mode for vertical regridding. "//&
01197                  "Choose among the following possibilities: "//&
01198                  trim(regriddingcoordinatemodedoc), &
01199                  default=default_coordinate_mode, fail_if_missing=.true.)
01200
01201   call initialize_regridding(regridcs, gv, us, max_depth, param_file, mdl, coord_mode, '', '')
01202
01203 end subroutine ale_initregridding
01204
01205 !> Query the target coordinate interfaces positions
01206 function ale_getcoordinate( CS )
01207   type(ale_cs), pointer    :: CS                  !< module control structure
01208
01209   real, dimension(CS%nk+1) :: ALE_getCoordinate
01210   ale_getcoordinate(:) = getcoordinateinterfaces( cs%regridCS, undo_scaling=.true. )
01211
01212 end function ale_getcoordinate
01213
01214
01215 !> Query the target coordinate units
01216 function ale_getcoordinateunits( CS )
01217   type(ale_cs), pointer :: CS   !< module control structure
01218
01219   character(len=20)     :: ALE_getCoordinateUnits
01220
01221   ale_getcoordinateunits = getcoordinateunits( cs%regridCS )
01222
01223 end function ale_getcoordinateunits
01224
01225
01226 !> Returns true if initial conditions should be regridded and remapped
01227 logical function ale_remap_init_conds( CS )
01228   type(ale_cs), pointer :: CS   !< module control structure
01229
01230   ale_remap_init_conds = .false.
01231   if (associated(cs)) ale_remap_init_conds = cs%remap_after_initialization
01232 end function ale_remap_init_conds
01233
01234 !> Updates the weights for time filtering the new grid generated in regridding
01235 subroutine ale_update_regrid_weights( dt, CS )
01236   real,         intent(in) :: dt !< Time-step used between ALE calls [T ~> s]
01237   type(ale_cs), pointer    :: CS !< ALE control structure
01238   ! Local variables
01239   real :: w  ! An implicit weighting estimate.
01240
01241   if (associated(cs)) then
01242     w = 0.0
01243     if (cs%regrid_time_scale > 0.0) then
01244       w = cs%regrid_time_scale / (cs%regrid_time_scale + dt)
01245     endif
01246     call set_regrid_params(cs%regridCS, old_grid_weight=w)
01247   endif
01248
01249 end subroutine ale_update_regrid_weights
01250
01251 !> Update the vertical grid type with ALE information.
01252 !! This subroutine sets information in the verticalGrid_type to be
```

```
01253 !! consistent with the use of ALE mode.
01254 subroutine ale_updateverticalgridtype(CS, GV)
01255   type(ale_cs),           pointer :: CS  !< ALE control structure
01256   type(verticalgrid_type), pointer :: GV  !< vertical grid information
01257
01258   integer :: nk
01259
01260   nk = gv%ke
01261   gv%sInterface(1:nk+1) = getcoordinateinterfaces( cs%regridCS, undo_scaling=.true. )
01262   gv%sLayer(1:nk) = 0.5*( gv%sInterface(1:nk) + gv%sInterface(2:nk+1) )
01263   gv%zAxisUnits = getcoordinateunits( cs%regridCS )
01264   gv%zAxisLongName = getcoordinateshortname( cs%regridCS )
01265   gv%direction = -1 ! Because of ferret in z* mode. Need method to set
01266                     ! as function of coordinae mode.
01267
01268 end subroutine ale_updateverticalgridtype
01269
01270
01271 !> Write the vertical coordinate information into a file.
01272 !! This subroutine writes out a file containing any available data related
01273 !! to the vertical grid used by the MOM ocean model when in ALE mode.
01274 subroutine ale_writecoordinatefile( CS, GV, directory )
01275   type(ale_cs),           pointer     :: CS         !< module control structure
01276   type(verticalgrid_type), intent(in)  :: GV         !< ocean vertical grid structure
01277   character(len=*),        intent(in)  :: directory  !< directory for writing grid info
01278
01279   character(len=240) :: filepath
01280   type(vardesc)      :: vars(2)
01281   type(fieldtype)    :: fields(2)
01282   integer            :: unit
01283   real               :: ds(gv%ke), dsi(gv%ke+1)
01284
01285   filepath    = trim(directory) // trim("Vertical_coordinate")
01286   ds(:)       = getcoordinateresolution( cs%regridCS, undo_scaling=.true. )
01287   dsi(1)      = 0.5*ds(1)
01288   dsi(2:gv%ke) = 0.5*( ds(1:gv%ke-1) + ds(2:gv%ke) )
01289   dsi(gv%ke+1) = 0.5*ds(gv%ke)
01290
01291   vars(1) = var_desc('ds', getcoordinateunits( cs%regridCS ), &
01292                      'Layer Coordinate Thickness','1','L','1')
01293   vars(2) = var_desc('ds_interface', getcoordinateunits( cs%regridCS ), &
01294                      'Layer Center Coordinate Separation','1','i','1')
01295
01296   call create_file(unit, trim(filepath), vars, 2, fields, single_file, gv=gv)
01297   call write_field(unit, fields(1), ds)
01298   call write_field(unit, fields(2), dsi)
01299   call close_file(unit)
01300
01301 end subroutine ale_writecoordinatefile
01302
01303
01304 !> Set h to coordinate values for fixed coordinate systems
01305 subroutine ale_initthicknesstocoord( CS, G, GV, h )
01306   type(ale_cs), intent(inout)                             :: CS  !< module control structure
01307   type(ocean_grid_type), intent(in)                       :: G   !< module grid structure
01308   type(verticalgrid_type), intent(in)                     :: GV  !< Ocean vertical grid structure
01309   real, dimension(SZI_(G),SZJ_(G),SZK_(GV)), intent(out) :: h   !< layer thickness [H ~> m or kg m-2]
01310
01311   ! Local variables
01312   integer :: i, j, k
01313
01314   do j = g%jsd,g%jed ; do i = g%isd,g%ied
01315     h(i,j,:) = gv%Z_to_H * getstaticthickness( cs%regridCS, 0., g%bathyT(i,j) )
01316   enddo ; enddo
01317
01318 end subroutine ale_initthicknesstocoord
01319
01320 end module mom_ale
```

44.5 /home/cermak/src/MOM6/src/core/_Baroclinic_Momentum.dox File Reference

44.6 /home/cermak/src/MOM6/src/core/_Barotropic_Baroclinic_Coupling.dox File Reference

44.7 /home/cermak/src/MOM6/src/core/_Barotropic_Momentum.dox File Reference

44.8 /home/cermak/src/MOM6/src/core/_Discrete_Coriolis.dox File Reference

44.9 /home/cermak/src/MOM6/src/core/_Discrete_grids.dox File Reference

44.10 /home/cermak/src/MOM6/src/core/_Discrete_OBC.dox File Reference

44.11 /home/cermak/src/MOM6/src/core/_Discrete_PG.dox File Reference

44.12 /home/cermak/src/MOM6/src/core/_Energetic_consistancy.dox File Reference

44.13 /home/cermak/src/MOM6/src/core/_Finite_difference.dox File Reference

44.14 /home/cermak/src/MOM6/src/core/_General_coordinate.dox File Reference

44.15 /home/cermak/src/MOM6/src/core/_Governing.dox File Reference

44.16 /home/cermak/src/MOM6/src/core/_Notation.dox File Reference

44.17 /home/cermak/src/MOM6/src/core/_PPM.dox File Reference

44.18 /home/cermak/src/MOM6/src/core/_Sea_ice.dox File Reference

44.19 /home/cermak/src/MOM6/src/core/_Solar_radiation.dox File Reference

44.20 /home/cermak/src/MOM6/src/core/_Specifics.dox File Reference

44.21 /home/cermak/src/MOM6/src/core/_Time_Filter.dox File Reference

44.22 /home/cermak/src/MOM6/src/core/MOM.F90 File Reference

```
#include <MOM_memory.h>
```

Data Types

- type mom::mom_diag_ids

  A structure with diagnostic IDs of the state variables.
- type mom::mom_control_struct

  Control structure for the MOM module, including the variables that describe the state of the ocean.

Modules

- module mom

  The central module of the MOM6 ocean model.

Functions/Subroutines

- subroutine, public mom::step_mom (forces_in, fluxes_in, sfc_state, Time_start, time_int_in, CS, Waves, do_dynamics, do_thermodynamics, start_cycle, end_cycle, cycle_length, reset_therm)

  This subroutine orchestrates the time stepping of MOM. The adiabatic dynamics are stepped by calls to one of the step_MOM_dyn_...routines. The action of lateral processes on tracers occur in calls to advect_tracer and tracer_hordiff. Vertical mixing and possibly remapping occur inside of diabatic.
- subroutine mom::step_mom_dynamics (forces, p_surf_begin, p_surf_end, dt, dt_thermo, bbl_time_int, CS, Time_local, Waves)

  Time step the ocean dynamics, including the momentum and continuity equations.
- subroutine mom::step_mom_tracer_dyn (CS, G, GV, US, h, Time_local)

  step_MOM_tracer_dyn does tracer advection and lateral diffusion, bringing the tracers up to date with the changes in state due to the dynamics. Surface sources and sinks and remapping are handled via step_MOM_thermo.
- subroutine mom::step_mom_thermo (CS, G, GV, US, u, v, h, tv, fluxes, dtdia, Time_end_thermo, update_BBL, Waves)

  MOM_step_thermo orchestrates the thermodynamic time stepping and vertical remapping, via calls to diabatic (or adiabatic) and ALE_main.
- subroutine, public mom::step_offline (forces, fluxes, sfc_state, Time_start, time_interval, CS)

  step_offline is the main driver for running tracers offline in MOM6. This has been primarily developed with ALE configurations in mind. Some work has been done in isopycnal configuration, but the work is very preliminary. Some more detail about this capability along with some of the subroutines called here can be found in tracers/MOM_offline_control.F90
- subroutine, public mom::initialize_mom (Time, Time_init, param_file, dirs, CS, restart_CSp, Time_in, offline_tracer_mode, input_restart_file, diag_ptr, count_calls, tracer_flow_CSp)

  Initialize MOM, including memory allocation, setting up parameters and diagnostics, initializing the ocean state variables, and initializing subsidiary modules.
- subroutine, public mom::finish_mom_initialization (Time, dirs, CS, restart_CSp)

  Finishes initializing MOM and writes out the initial conditions.
- subroutine mom::register_diags (Time, G, GV, US, IDs, diag)

  Register certain diagnostics.
- subroutine mom::mom_timing_init (CS)

  Set up CPU clock IDs for timing various subroutines.
- subroutine mom::set_restart_fields (GV, US, param_file, CS, restart_CSp)

  Set the fields that are needed for bitwise identical restarting the time stepping scheme. In addition to those specified here directly, there may be fields related to the forcing or to the barotropic solver that are needed; these are specified in sub- routines that are called from this one.
- subroutine mom::adjust_ssh_for_p_atm (tv, G, GV, US, ssh, p_atm, use_EOS)

  Apply a correction to the sea surface height to compensate for the atmospheric pressure (the inverse barometer).
- subroutine, public mom::extract_surface_state (CS, sfc_state_in)

  Set the surface (return) properties of the ocean model by setting the appropriate fields in sfc_state. Unused fields are set to NULL or are unallocated.
- subroutine mom::rotate_initial_state (u_in, v_in, h_in, T_in, S_in, use_temperature, turns, u, v, h, T, S)

  Rotate initialization fields from input to rotated arrays.
- logical function, public mom::mom_state_is_synchronized (CS, adv_dyn)

  Return true if all phases of step_MOM are at the same point in time.
- subroutine, public mom::get_mom_state_elements (CS, G, GV, US, C_p, C_p_scaled, use_temp)

  This subroutine offers access to values or pointers to other types from within the MOM_control_struct, allowing the MOM_control_struct to be opaque.
- subroutine, public mom::get_ocean_stocks (CS, mass, heat, salt, on_PE_only)

  Find the global integrals of various quantities.
- subroutine, public mom::mom_end (CS)

  End of ocean model, including memory deallocation.

Variables

- integer mom::id_clock_ocean

  CPU time clock IDs.
- integer mom::id_clock_dynamics

  CPU time clock IDs.
- integer mom::id_clock_thermo

  CPU time clock IDs.
- integer mom::id_clock_tracer

  CPU time clock IDs.
- integer mom::id_clock_diabatic

  CPU time clock IDs.
- integer mom::id_clock_adiabatic

  CPU time clock IDs.
- integer mom::id_clock_continuity

  CPU time clock IDs.
- integer mom::id_clock_thick_diff

  CPU time clock IDs.
- integer mom::id_clock_bbl_visc

  CPU time clock IDs.
- integer mom::id_clock_ml_restrat

  CPU time clock IDs.
- integer mom::id_clock_diagnostics

  CPU time clock IDs.
- integer mom::id_clock_z_diag

  CPU time clock IDs.
- integer mom::id_clock_init

  CPU time clock IDs.
- integer mom::id_clock_mom_init

  CPU time clock IDs.
- integer mom::id_clock_pass

  CPU time clock IDs.
- integer mom::id_clock_pass_init

  CPU time clock IDs.
- integer mom::id_clock_ale

  CPU time clock IDs.
- integer mom::id_clock_other

  CPU time clock IDs.
- integer mom::id_clock_offline_tracer

  CPU time clock IDs.
- integer mom::id_clock_unit_tests

  CPU time clock IDs.

44.23 MOM.F90

```
00001 !> The central module of the MOM6 ocean model
00002 module mom
00003
00004 ! This file is part of MOM6. See LICENSE.md for the license.
00005
00006 ! Infrastructure modules
00007 use mom_array_transform,      only : rotate_array, rotate_vector
00008 use mom_debugging,            only : mom_debugging_init, hchksum, uvchksum
00009 use mom_debugging,            only : check_redundant
00010 use mom_checksum_packages,    only : mom_thermo_chksum, mom_state_chksum
00011 use mom_checksum_packages,    only : mom_accel_chksum, mom_surface_chksum
00012 use mom_coms,                 only : num_pes
00013 use mom_cpu_clock,            only : cpu_clock_id, cpu_clock_begin, cpu_clock_end
00014 use mom_cpu_clock,            only : clock_component, clock_subcomponent
00015 use mom_cpu_clock,            only : clock_module_driver, clock_module, clock_routine
00016 use mom_diag_mediator,        only : diag_mediator_init, enable_averaging, enable_averages
00017 use mom_diag_mediator,        only : diag_mediator_infrastructure_init
00018 use mom_diag_mediator,        only : diag_set_state_ptrs, diag_update_remap_grids
00019 use mom_diag_mediator,        only : disable_averaging, post_data, safe_alloc_ptr
00020 use mom_diag_mediator,        only : register_diag_field, register_cell_measure
00021 use mom_diag_mediator,        only : set_axes_info, diag_ctrl, diag_masks_set
00022 use mom_diag_mediator,        only : set_masks_for_axes
00023 use mom_diag_mediator,        only : diag_grid_storage, diag_grid_storage_init
00024 use mom_diag_mediator,        only : diag_save_grids, diag_restore_grids
00025 use mom_diag_mediator,        only : diag_copy_storage_to_diag, diag_copy_diag_to_storage
00026 use mom_domains,              only : mom_domains_init, clone_mom_domain
00027 use mom_domains,              only : sum_across_pes, pass_var, pass_vector
00028 use mom_domains,              only : to_north, to_east, to_south, to_west
00029 use mom_domains,              only : to_all, omit_corners, cgrid_ne, scalar_pair
00030 use mom_domains,              only : create_group_pass, do_group_pass, group_pass_type
00031 use mom_domains,              only : start_group_pass, complete_group_pass, omit_corners
00032 use mom_error_handler,        only : mom_error, mom_mesg, fatal, warning, is_root_pe
00033 use mom_error_handler,        only : mom_set_verbosity, calltree_showquery
00034 use mom_error_handler,        only : calltree_enter, calltree_leave, calltree_waypoint
00035 use mom_file_parser,          only : read_param, get_param, log_version, param_file_type
00036 use mom_forcing_type,         only : forcing, mech_forcing
00037 use mom_forcing_type,         only : mom_forcing_chksum, mom_mech_forcing_chksum
00038 use mom_get_input,            only : get_mom_input, directories
00039 use mom_io,                   only : mom_io_init, vardesc, var_desc
00040 use mom_io,                   only : slasher, file_exists, mom_read_data
00041 use mom_obsolete_params,      only : find_obsolete_params
00042 use mom_restart,              only : register_restart_field, register_restart_pair
00043 use mom_restart,              only : query_initialized, save_restart
00044 use mom_restart,              only : restart_init, is_new_run, mom_restart_cs
00045 use mom_spatial_means,        only : global_mass_integral
00046 use mom_time_manager,         only : time_type, real_to_time, time_type_to_real, operator(+)
00047 use mom_time_manager,         only : operator(-), operator(>), operator(*), operator(/)
00048 use mom_time_manager,         only : operator(>=), operator(==), increment_date
00049 use mom_unit_tests,           only : unit_tests
00050 use coupler_types_mod,        only : coupler_type_send_data, coupler_1d_bc_type, coupler_type_spawn
00051
00052 ! MOM core modules
00053 use mom_ale,                  only : ale_init, ale_end,
      ale_main, ale_cs, adjustgridforintegrity
00054 use mom_ale,                  only : ale_getcoordinate,
      ale_getcoordinateunits, ale_writecoordinatefile
00055 use mom_ale,                  only : ale_updateverticalgridtype,
      ale_remap_init_conds, ale_register_diags
00056 use mom_ale_sponge,           only : rotate_ale_sponge, update_ale_sponge_field
00057 use mom_barotropic,           only : barotropic_cs
00058 use mom_boundary_update,      only : call_obc_register, obc_register_end, update_obc_cs
00059 use mom_coord_initialization, only : mom_initialize_coord
00060 use mom_diabatic_driver,      only : diabatic, diabatic_driver_init, diabatic_cs, extract_diabatic_member
00061 use mom_diabatic_driver,      only : adiabatic, adiabatic_driver_init, diabatic_driver_end
00062 use mom_diagnostics,          only : calculate_diagnostic_fields, mom_diagnostics_init
00063 use mom_diagnostics,          only : register_transport_diags, post_transport_diagnostics
00064 use mom_diagnostics,          only : register_surface_diags, write_static_fields
00065 use mom_diagnostics,          only : post_surface_dyn_diags, post_surface_thermo_diags
00066 use mom_diagnostics,          only : diagnostics_cs, surface_diag_ids, transport_diag_ids
00067 use mom_dynamics_unsplit,     only : step_mom_dyn_unsplit, register_restarts_dyn_unsplit
00068 use mom_dynamics_unsplit,     only : initialize_dyn_unsplit, end_dyn_unsplit
00069 use mom_dynamics_unsplit,     only : mom_dyn_unsplit_cs
00070 use mom_dynamics_split_rk2,   only : step_mom_dyn_split_rk2, register_restarts_dyn_split_rk2
00071 use mom_dynamics_split_rk2,   only : initialize_dyn_split_rk2, end_dyn_split_rk2
00072 use mom_dynamics_split_rk2,   only : mom_dyn_split_rk2_cs
00073 use mom_dynamics_unsplit_rk2, only : step_mom_dyn_unsplit_rk2, register_restarts_dyn_unsplit_rk2
00074 use mom_dynamics_unsplit_rk2, only : initialize_dyn_unsplit_rk2, end_dyn_unsplit_rk2
00075 use mom_dynamics_unsplit_rk2, only : mom_dyn_unsplit_rk2_cs
00076 use mom_dyn_horgrid,          only : dyn_horgrid_type, create_dyn_horgrid, destroy_dyn_horgrid
00077 use mom_eos,                  only : eos_init, calculate_density,
      calculate_tfreeze, eos_domain
00078 use mom_fixed_initialization, only : mom_initialize_fixed
00079 use mom_forcing_type,         only : allocate_forcing_type, allocate_mech_forcing
00080 use mom_forcing_type,         only : deallocate_mech_forcing, deallocate_forcing_type
```

```
00081 use mom_forcing_type,          only : rotate_forcing, rotate_mech_forcing
00082 use mom_grid,                 only : ocean_grid_type, mom_grid_init, mom_grid_end
00083 use mom_grid,                 only : set_first_direction, rescale_grid_bathymetry
00084 use mom_hor_index,            only : hor_index_type, hor_index_init
00085 use mom_hor_index,            only : rotate_hor_index
00086 use mom_interface_heights,    only : find_eta
00087 use mom_lateral_mixing_coeffs, only : calc_slope_functions, varmix_init
00088 use mom_lateral_mixing_coeffs, only : calc_resoln_function, calc_depth_function, varmix_cs
00089 use mom_meke,                 only : meke_init,
      meke_alloc_register_restart, step_forward_meke,
      meke_cs
00090 use mom_meke_types,           only : meke_type
00091 use mom_mixed_layer_restrat,  only : mixedlayer_restrat, mixedlayer_restrat_init, mixedlayer_restrat_cs
00092 use mom_mixed_layer_restrat,  only : mixedlayer_restrat_register_restarts
00093 use mom_obsolete_diagnostics, only : register_obsolete_diagnostics
00094 use mom_open_boundary,        only : ocean_obc_type, obc_registry_type
00095 use mom_open_boundary,        only : register_temp_salt_segments
00096 use mom_open_boundary,        only : open_boundary_register_restarts
00097 use mom_open_boundary,        only : update_segment_tracer_reservoirs
00098 use mom_open_boundary,        only : rotate_obc_config, rotate_obc_init
00099 use mom_set_visc,             only : set_viscous_bbl, set_viscous_ml, set_visc_init
00100 use mom_set_visc,             only : set_visc_register_restarts, set_visc_cs
00101 use mom_sponge,               only : init_sponge_diags, sponge_cs
00102 use mom_state_initialization, only : mom_initialize_state
00103 use mom_sum_output,           only : write_energy, accumulate_net_input
00104 use mom_sum_output,           only : mom_sum_output_init, sum_output_cs
00105 use mom_ale_sponge,           only : init_ale_sponge_diags, ale_sponge_cs
00106 use mom_thickness_diffuse,    only : thickness_diffuse, thickness_diffuse_init, thickness_diffuse_cs
00107 use mom_tracer_advect,        only : advect_tracer, tracer_advect_init
00108 use mom_tracer_advect,        only : tracer_advect_end, tracer_advect_cs
00109 use mom_tracer_hor_diff,      only : tracer_hordiff, tracer_hor_diff_init
00110 use mom_tracer_hor_diff,      only : tracer_hor_diff_end, tracer_hor_diff_cs
00111 use mom_tracer_registry,      only : tracer_registry_type, register_tracer, tracer_registry_init
00112 use mom_tracer_registry,      only : register_tracer_diagnostics, post_tracer_diagnostics_at_sync
00113 use mom_tracer_registry,      only : post_tracer_transport_diagnostics
00114 use mom_tracer_registry,      only : preale_tracer_diagnostics, postale_tracer_diagnostics
00115 use mom_tracer_registry,      only : lock_tracer_registry, tracer_registry_end
00116 use mom_tracer_flow_control,  only : call_tracer_register, tracer_flow_control_cs
00117 use mom_tracer_flow_control,  only : tracer_flow_control_init, call_tracer_surface_state
00118 use mom_tracer_flow_control,  only : tracer_flow_control_end
00119 use mom_transcribe_grid,      only : copy_dyngrid_to_mom_grid, copy_mom_grid_to_dyngrid
00120 use mom_transcribe_grid,      only : rotate_dyngrid
00121 use mom_unit_scaling,         only : unit_scale_type,
      unit_scaling_init
00122 use mom_unit_scaling,         only : unit_scaling_end,
      fix_restart_unit_scaling
00123 use mom_variables,            only : surface, allocate_surface_state, deallocate_surface_state
00124 use mom_variables,            only : thermo_var_ptrs, vertvisc_type
00125 use mom_variables,            only : accel_diag_ptrs, cont_diag_ptrs, ocean_internal_state
00126 use mom_variables,            only : rotate_surface_state
00127 use mom_verticalgrid,         only : verticalgrid_type, verticalgridinit, verticalgridend
00128 use mom_verticalgrid,         only : fix_restart_scaling
00129 use mom_verticalgrid,         only : get_thickness_units, get_flux_units, get_tr_flux_units
00130 use mom_wave_interface,       only : wave_parameters_cs, waves_end
00131 use mom_wave_interface,       only : update_stokes_drift
00132
00133 ! ODA modules
00134 use mom_oda_driver_mod,       only : oda_cs, oda, init_oda, oda_end
00135 use mom_oda_driver_mod,       only : set_prior_tracer, set_analysis_time, apply_oda_tracer_increments
00136 ! Offline modules
00137 use mom_offline_main,         only : offline_transport_cs, offline_transport_init, update_offline_fields
00138 use mom_offline_main,         only : insert_offline_main, extract_offline_main,
      post_offline_convergence_diags
00139 use mom_offline_main,         only : register_diags_offline_transport, offline_advection_ale
00140 use mom_offline_main,         only : offline_redistribute_residual, offline_diabatic_ale
00141 use mom_offline_main,         only : offline_fw_fluxes_into_ocean, offline_fw_fluxes_out_ocean
00142 use mom_offline_main,         only : offline_advection_layer, offline_transport_end
00143 use mom_ale,                  only : ale_offline_tracer_final,
      ale_main_offline
00144
00145 implicit none ; private
00146
00147 #include <MOM_memory.h>
00148
00149 ! A note on unit descriptions in comments: MOM6 uses units that can be rescaled for dimensional
00150 ! consistency testing. These are noted in comments with units like Z, H, L, and T, along with
00151 ! their mks counterparts with notation like "a velocity [Z T-1 ~> m s-1]".  If the units
00152 ! vary with the Boussinesq approximation, the Boussinesq variant is given first.
00153
00154 !> A structure with diagnostic IDs of the state variables
00155 type mom_diag_ids
00156   !>@{ 3-d state field diagnostic IDs
00157   integer :: id_u  = -1, id_v  = -1, id_h  = -1
00158   !>@}
00159   !> 2-d state field diagnotic ID
00160   integer :: id_ssh_inst = -1
00161 end type mom_diag_ids
```

```
00162
00163 !> Control structure for the MOM module, including the variables that describe
00164 !! the state of the ocean.
00165 type, public :: mom_control_struct ; private
00166   real allocable_, dimension(NIMEM_,NJMEM_,NKMEM_) :: &
00167     h, &            !< layer thickness [H ~> m or kg m-2]
00168     t, &            !< potential temperature [degC]
00169     s               !< salinity [ppt]
00170   real allocable_, dimension(NIMEMB_PTR_,NJMEM_,NKMEM_) :: &
00171     u,  &           !< zonal velocity component [L T-1 ~> m s-1]
00172     uh, &           !< uh = u * h * dy at u grid points [H L2 T-1 ~> m3 s-1 or kg s-1]
00173     uhtr            !< accumulated zonal thickness fluxes to advect tracers [H L2 ~> m3 or kg]
00174   real allocable_, dimension(NIMEM_,NJMEMB_PTR_,NKMEM_) :: &
00175     v,  &           !< meridional velocity [L T-1 ~> m s-1]
00176     vh, &           !< vh = v * h * dx at v grid points [H L2 T-1 ~> m3 s-1 or kg s-1]
00177     vhtr            !< accumulated meridional thickness fluxes to advect tracers [H L2 ~> m3 or kg]
00178   real allocable_, dimension(NIMEM_,NJMEM_) :: ssh_rint
00179                     !< A running time integral of the sea surface height [T m ~> s m].
00180   real allocable_, dimension(NIMEM_,NJMEM_) :: ave_ssh_ibc
00181                     !< time-averaged (over a forcing time step) sea surface height
00182                     !! with a correction for the inverse barometer [m]
00183   real allocable_, dimension(NIMEM_,NJMEM_) :: eta_av_bc
00184                     !< free surface height or column mass time averaged over the last
00185                     !! baroclinic dynamics time step [H ~> m or kg m-2]
00186   real, dimension(:,:), pointer :: &
00187     hml => null()   !< active mixed layer depth [Z ~> m]
00188   real :: time_in_cycle !< The running time of the current time-stepping cycle
00189                     !! in calls that step the dynamics, and also the length of
00190                     !! the time integral of ssh_rint [T ~> s].
00191   real :: time_in_thermo_cycle !< The running time of the current time-stepping
00192                     !! cycle in calls that step the thermodynamics [T ~> s].
00193
00194   type(ocean_grid_type) :: g_in                   !< Input grid metric
00195   type(ocean_grid_type), pointer :: g => null()   !< Model grid metric
00196   logical :: rotate_index = .false.   !< True if index map is rotated
00197
00198   type(verticalgrid_type), pointer :: &
00199     gv => null()    !< structure containing vertical grid info
00200   type(unit_scale_type), pointer :: &
00201     us => null()    !< structure containing various unit conversion factors
00202   type(thermo_var_ptrs) :: tv !< structure containing pointers to available thermodynamic fields
00203   real :: t_dyn_rel_adv !< The time of the dynamics relative to tracer advection and lateral mixing
00204                     !! [T ~> s], or equivalently the elapsed time since advectively updating the
00205                     !! tracers.  t_dyn_rel_adv is invariably positive and may span multiple coupling
      timesteps.
00206   real :: t_dyn_rel_thermo  !< The time of the dynamics relative to diabatic  processes and remapping
00207                     !! [T ~> s].  t_dyn_rel_thermo can be negative or positive depending on whether
00208                     !! the diabatic processes are applied before or after the dynamics and may span
00209                     !! multiple coupling timesteps.
00210   real :: t_dyn_rel_diag !< The time of the diagnostics relative to diabatic processes and remapping
00211                     !!  [T ~> s].  t_dyn_rel_diag is always positive, since the diagnostics must lag.
00212   logical :: preadv_h_stored = .false. !< If true, the thicknesses from before the advective cycle
00213                     !! have been stored for use in diagnostics.
00214
00215   type(diag_ctrl)     :: diag !< structure to regulate diagnostic output timing
00216   type(vertvisc_type) :: visc !< structure containing vertical viscosities,
00217                     !! bottom drag viscosities, and related fields
00218   type(meke_type), pointer :: meke => null() !<  structure containing fields
00219                     !! related to the Mesoscale Eddy Kinetic Energy
00220   logical :: adiabatic !< If true, there are no diapycnal mass fluxes, and no calls
00221                     !! to routines to calculate or apply diapycnal fluxes.
00222   logical :: diabatic_first !< If true, apply diabatic and thermodynamic processes before time
00223                     !! stepping the dynamics.
00224   logical :: use_ale_algorithm  !< If true, use the ALE algorithm rather than layered
00225                     !! isopycnal/stacked shallow water mode. This logical is set by calling the
00226                     !! function useRegridding() from the MOM_regridding module.
00227   logical :: offline_tracer_mode = .false.
00228                     !< If true, step_offline() is called instead of step_MOM().
00229                     !! This is intended for running MOM6 in offline tracer mode
00230
00231   type(time_type), pointer :: time   !< pointer to the ocean clock
00232   real    :: dt                      !< (baroclinic) dynamics time step [T ~> s]
00233   real    :: dt_therm                !< thermodynamics time step [T ~> s]
00234   logical :: thermo_spans_coupling   !< If true, thermodynamic and tracer time
00235                                      !! steps can span multiple coupled time steps.
00236   integer :: nstep_tot = 0           !< The total number of dynamic timesteps tcaaken
00237                                      !! so far in this run segment
00238   logical :: count_calls = .false.   !< If true, count the calls to step_MOM, rather than the
00239                                      !! number of dynamics steps in nstep_tot
00240   logical :: debug                   !< If true, write verbose checksums for debugging purposes.
00241   integer :: ntrunc                  !< number u,v truncations since last call to write_energy
00242
00243   integer :: cont_stencil            !< The stencil for thickness from the continuity solver.
00244   ! These elements are used to control the dynamics updates.
00245   logical :: do_dynamics             !< If false, does not call step_MOM_dyn_*. This is an
00246                                      !! undocumented run-time flag that is fragile.
00247   logical :: split                   !< If true, use the split time stepping scheme.
```

```
00248   logical :: use_rk2                !< If true, use RK2 instead of RK3 in unsplit mode
00249                                     !! (i.e., no split between barotropic and baroclinic).
00250   logical :: thickness_diffuse      !< If true, diffuse interface height w/ a diffusivity KHTH.
00251   logical :: thickness_diffuse_first !< If true, diffuse thickness before dynamics.
00252   logical :: mixedlayer_restrat     !< If true, use submesoscale mixed layer restratifying scheme.
00253   logical :: usemeke                !< If true, call the MEKE parameterization.
00254   logical :: usewaves               !< If true, update Stokes drift
00255   logical :: use_p_surf_in_eos      !< If true, always include the surface pressure contributions
00256                                     !! in equation of state calculations.
00257   real :: dtbt_reset_period         !< The time interval between dynamic recalculation of the
00258                                     !! barotropic time step [s]. If this is negative dtbt is never
00259                                     !! calculated, and if it is 0, dtbt is calculated every step.
00260   type(time_type) :: dtbt_reset_interval !< A time_time representation of dtbt_reset_period.
00261   type(time_type) :: dtbt_reset_time !< The next time DTBT should be calculated.
00262
00263
00264   real, dimension(:,:,:), pointer :: &
00265     h_pre_dyn => null(), &      !< The thickness before the transports [H ~> m or kg m-2].
00266     t_pre_dyn => null(), &      !< Temperature before the transports [degC].
00267     s_pre_dyn => null()         !< Salinity before the transports [ppt].
00268   type(accel_diag_ptrs) :: adp  !< structure containing pointers to accelerations,
00269                                 !! for derived diagnostics (e.g., energy budgets)
00270   type(cont_diag_ptrs)  :: cdp  !< structure containing pointers to continuity equation
00271                                 !! terms, for derived diagnostics (e.g., energy budgets)
00272   real, dimension(:,:,:), pointer :: &
00273     u_prev => null(), &         !< previous value of u stored for diagnostics [L T-1 ~> m s-1]
00274     v_prev => null()            !< previous value of v stored for diagnostics [L T-1 ~> m s-1]
00275
00276   logical :: interp_p_surf      !< If true, linearly interpolate surface pressure
00277                                 !! over the coupling time step, using specified value
00278                                 !! at the end of the coupling step. False by default.
00279   logical :: p_surf_prev_set    !< If true, p_surf_prev has been properly set from
00280                                 !! a previous time-step or the ocean restart file.
00281                                 !! This is only valid when interp_p_surf is true.
00282   real, dimension(:,:), pointer :: &
00283     p_surf_prev  => null(), &   !< surface pressure [R L2 T-2 ~> Pa] at end  previous call to step_MOM
00284     p_surf_begin => null(), &   !< surface pressure [R L2 T-2 ~> Pa] at start of step_MOM_dyn_...
00285     p_surf_end   => null()      !< surface pressure [R L2 T-2 ~> Pa] at end   of step_MOM_dyn_...
00286
00287   ! Variables needed to reach between start and finish phases of initialization
00288   logical :: write_ic           !< If true, then the initial conditions will be written to file
00289   character(len=120) :: ic_file !< A file into which the initial conditions are
00290                                 !! written in a new run if SAVE_INITIAL_CONDS is true.
00291
00292   logical :: calc_rho_for_sea_lev !< If true, calculate rho to convert pressure to sea level
00293
00294   ! These elements are used to control the calculation and error checking of the surface state
00295   real :: hmix                  !< Diagnostic mixed layer thickness over which to
00296                                 !! average surface tracer properties when a bulk
00297                                 !! mixed layer is not used [Z ~> m], or a negative value
00298                                 !! if a bulk mixed layer is being used.
00299   real :: hfrz                  !< If HFrz > 0, the nominal depth over which melt potential is
00300                                 !! computed [Z ~> m]. The actual depth over which melt potential is
00301                                 !! computed is min(HFrz, OBLD), where OBLD is the boundary layer depth.
00302                                 !! If HFrz <= 0 (default), melt potential will not be computed.
00303   real :: hmix_uv               !< Depth scale over which to average surface flow to
00304                                 !! feedback to the coupler/driver [Z ~> m] when
00305                                 !! bulk mixed layer is not used, or a negative value
00306                                 !! if a bulk mixed layer is being used.
00307   logical :: check_bad_sfc_vals !< If true, scan surface state for ridiculous values.
00308   real    :: bad_val_ssh_max    !< Maximum SSH before triggering bad value message [Z ~> m]
00309   real    :: bad_val_sst_max    !< Maximum SST before triggering bad value message [degC]
00310   real    :: bad_val_sst_min    !< Minimum SST before triggering bad value message [degC]
00311   real    :: bad_val_sss_max    !< Maximum SSS before triggering bad value message [ppt]
00312   real    :: bad_val_col_thick  !< Minimum column thickness before triggering bad value message [Z ~> m]
00313   logical :: answers_2018       !< If true, use expressions for the surface properties that recover
00314                                 !! the answers from the end of 2018. Otherwise, use more appropriate
00315                                 !! expressions that differ at roundoff for non-Boussinsq cases.
00316
00317   type(mom_diag_ids)       :: ids      !<  Handles used for diagnostics.
00318   type(transport_diag_ids) :: transport_ids  !< Handles used for transport diagnostics.
00319   type(surface_diag_ids)   :: sfc_ids  !< Handles used for surface diagnostics.
00320   type(diag_grid_storage)  :: diag_pre_sync !< The grid (thicknesses) before remapping
00321   type(diag_grid_storage)  :: diag_pre_dyn  !< The grid (thicknesses) before dynamics
00322
00323   ! The remainder of this type provides pointers to child module control structures.
00324
00325   type(mom_dyn_unsplit_cs),      pointer :: dyn_unsplit_csp => null()
00326     !< Pointer to the control structure used for the unsplit dynamics
00327   type(mom_dyn_unsplit_rk2_cs),  pointer :: dyn_unsplit_rk2_csp => null()
00328     !< Pointer to the control structure used for the unsplit RK2 dynamics
00329   type(mom_dyn_split_rk2_cs),    pointer :: dyn_split_rk2_csp => null()
00330     !< Pointer to the control structure used for the mode-split RK2 dynamics
00331   type(thickness_diffuse_cs),    pointer :: thickness_diffuse_csp => null()
00332     !< Pointer to the control structure used for the isopycnal height diffusive transport.
00333     !! This is also common referred to as Gent-McWilliams diffusion
00334   type(mixedlayer_restrat_cs),   pointer :: mixedlayer_restrat_csp => null()
```

```
00335     !< Pointer to the control structure used for the mixed layer restratification
00336   type(set_visc_cs),             pointer :: set_visc_csp => null()
00337     !< Pointer to the control structure used to set viscosities
00338   type(diabatic_cs),             pointer :: diabatic_csp => null()
00339     !< Pointer to the control structure for the diabatic driver
00340   type(meke_cs),                 pointer :: meke_csp => null()
00341     !< Pointer to the control structure for the MEKE updates
00342   type(varmix_cs),               pointer :: varmix => null()
00343     !< Pointer to the control structure for the variable mixing module
00344   type(barotropic_cs),           pointer :: barotropic_csp => null()
00345     !< Pointer to the control structure for the barotropic module
00346   type(tracer_registry_type),    pointer :: tracer_reg => null()
00347     !< Pointer to the MOM tracer registry
00348   type(tracer_advect_cs),        pointer :: tracer_adv_csp => null()
00349     !< Pointer to the MOM tracer advection control structure
00350   type(tracer_hor_diff_cs),      pointer :: tracer_diff_csp => null()
00351     !< Pointer to the MOM along-isopycnal tracer diffusion control structure
00352   type(tracer_flow_control_cs),  pointer :: tracer_flow_csp => null()
00353     !< Pointer to the control structure that orchestrates the calling of tracer packages
00354     ! Although update_OBC_CS is not used directly outside of initialization, other modules
00355     ! set pointers to this type, so it should be kept for the duration of the run.
00356   type(update_obc_cs),           pointer :: update_obc_csp => null()
00357     !< Pointer to the control structure for updating open boundary condition properties
00358   type(ocean_obc_type),          pointer :: obc => null()
00359     !< Pointer to the MOM open boundary condition type
00360   type(sponge_cs),               pointer :: sponge_csp => null()
00361     !< Pointer to the layered-mode sponge control structure
00362   type(ale_sponge_cs),           pointer :: ale_sponge_csp => null()
00363     !< Pointer to the ALE-mode sponge control structure
00364   type(ale_cs),                  pointer :: ale_csp => null()
00365     !< Pointer to the Arbitrary Lagrangian Eulerian (ALE) vertical coordinate control structure
00366
00367   ! Pointers to control structures used for diagnostics
00368   type(sum_output_cs),           pointer :: sum_output_csp => null()
00369     !< Pointer to the globally summed output control structure
00370   type(diagnostics_cs),          pointer :: diagnostics_csp => null()
00371     !< Pointer to the MOM diagnostics control structure
00372   type(offline_transport_cs),    pointer :: offline_csp => null()
00373     !< Pointer to the offline tracer transport control structure
00374
00375   logical               :: ensemble_ocean !< if true, this run is part of a
00376                                           !! larger ensemble for the purpose of data assimilation
00377                                           !! or statistical analysis.
00378   type(oda_cs), pointer :: odacs => null() !< a pointer to the control structure for handling
00379                                           !! ensemble model state vectors and data assimilation
00380                                           !! increments and priors
00381 end type mom_control_struct
00382
00383 public initialize_mom, finish_mom_initialization,
      mom_end
00384 public step_mom, step_offline
00385 public extract_surface_state, get_ocean_stocks
00386 public get_mom_state_elements, mom_state_is_synchronized
00387 public allocate_surface_state, deallocate_surface_state
00388
00389 !>@{ CPU time clock IDs
00390 integer :: id_clock_ocean
00391 integer :: id_clock_dynamics
00392 integer :: id_clock_thermo
00393 integer :: id_clock_tracer
00394 integer :: id_clock_diabatic
00395 integer :: id_clock_adiabatic
00396 integer :: id_clock_continuity  ! also in dynamics s/r
00397 integer :: id_clock_thick_diff
00398 integer :: id_clock_bbl_visc
00399 integer :: id_clock_ml_restrat
00400 integer :: id_clock_diagnostics
00401 integer :: id_clock_z_diag
00402 integer :: id_clock_init
00403 integer :: id_clock_mom_init
00404 integer :: id_clock_pass       ! also in dynamics d/r
00405 integer :: id_clock_pass_init  ! also in dynamics d/r
00406 integer :: id_clock_ale
00407 integer :: id_clock_other
00408 integer :: id_clock_offline_tracer
00409 integer :: id_clock_unit_tests
00410 !>@}
00411
00412 contains
00413
00414 !> This subroutine orchestrates the time stepping of MOM.  The adiabatic
00415 !! dynamics are stepped by calls to one of the step_MOM_dyn_...routines.
00416 !! The action of lateral processes on tracers occur in calls to
00417 !! advect_tracer and tracer_hordiff.  Vertical mixing and possibly remapping
00418 !! occur inside of diabatic.
00419 subroutine step_mom(forces_in, fluxes_in, sfc_state, Time_start, time_int_in, CS, &
00420                     Waves, do_dynamics, do_thermodynamics, start_cycle, &
```

```
00421                    end_cycle, cycle_length, reset_therm)
00422   type(mech_forcing), target, intent(inout) :: forces_in !< A structure with the driving mechanical forces
00423   type(forcing), target, intent(inout) :: fluxes_in  !< A structure with pointers to themodynamic,
00424                                                      !! tracer and mass exchange forcing fields
00425   type(surface), target, intent(inout) :: sfc_state  !< surface ocean state
00426   type(time_type),    intent(in)    :: Time_start    !< starting time of a segment, as a time type
00427   real,               intent(in)    :: time_int_in   !< time interval covered by this run segment [s].
00428   type(mom_control_struct), pointer :: CS            !< control structure from
      initialize_MOM
00429   type(wave_parameters_cs), &
00430             optional, pointer       :: Waves         !< An optional pointer to a wave property CS
00431   logical,  optional, intent(in)    :: do_dynamics   !< Present and false, do not do updates due
00432                                                      !! to the dynamics.
00433   logical,  optional, intent(in)    :: do_thermodynamics  !< Present and false, do not do updates due
00434                                                      !! to the thermodynamics or remapping.
00435   logical,  optional, intent(in)    :: start_cycle   !< This indicates whether this call is to be
00436                                                      !! treated as the first call to step_MOM in a
00437                                                      !! time-stepping cycle; missing is like true.
00438   logical,  optional, intent(in)    :: end_cycle     !< This indicates whether this call is to be
00439                                                      !! treated as the last call to step_MOM in a
00440                                                      !! time-stepping cycle; missing is like true.
00441   real,     optional, intent(in)    :: cycle_length  !< The amount of time in a coupled time
00442                                                      !! stepping cycle [s].
00443   logical,  optional, intent(in)    :: reset_therm   !< This indicates whether the running sums of
00444                                                      !! thermodynamic quantities should be reset.
00445                                                      !! If missing, this is like start_cycle.
00446
00447   ! local variables
00448   type(ocean_grid_type),   pointer :: G => null()  ! pointer to a structure containing
00449                                                    ! metrics and related information
00450   type(ocean_grid_type),   pointer :: G_in => null()  ! Input grid metric
00451   type(verticalgrid_type), pointer :: GV => null() ! Pointer to the vertical grid structure
00452   type(unit_scale_type),   pointer :: US => null() ! Pointer to a structure containing
00453                                                    ! various unit conversion factors
00454   integer       :: ntstep ! time steps between tracer updates or diabatic forcing
00455   integer       :: n_max  ! number of steps to take in this call
00456
00457   integer :: i, j, k, is, ie, js, je, Isq, Ieq, Jsq, Jeq, nz, n
00458   integer :: isd, ied, jsd, jed, IsdB, IedB, JsdB, JedB
00459
00460   real :: time_interval   ! time interval covered by this run segment [T ~> s].
00461   real :: dt              ! baroclinic time step [T ~> s]
00462   real :: dtdia           ! time step for diabatic processes [T ~> s]
00463   real :: dt_therm        ! a limited and quantized version of CS%dt_therm [T ~> s]
00464   real :: dt_therm_here   ! a further limited value of dt_therm [T ~> s]
00465
00466   real :: wt_end, wt_beg
00467   real :: bbl_time_int    ! The amount of time over which the calculated BBL
00468                           ! properties will apply, for use in diagnostics, or 0
00469                           ! if it is not to be calculated anew [T ~> s].
00470   real :: rel_time = 0.0  ! relative time since start of this call [T ~> s].
00471
00472   logical :: calc_dtbt                ! Indicates whether the dynamically adjusted
00473                                       ! barotropic time step needs to be updated.
00474   logical :: do_advection             ! If true, it is time to advect tracers.
00475   logical :: do_calc_bbl              ! If true, calculate the boundary layer properties.
00476   logical :: thermo_does_span_coupling ! If true, thermodynamic forcing spans
00477                                       ! multiple dynamic timesteps.
00478   logical :: do_dyn     ! If true, dynamics are updated with this call.
00479   logical :: do_thermo  ! If true, thermodynamics and remapping may be applied with this call.
00480   logical :: cycle_start ! If true, do calculations that are only done at the start of
00481                          ! a stepping cycle (whatever that may mean).
00482   logical :: cycle_end  ! If true, do calculations and diagnostics that are only done at
00483                         ! the end of a stepping cycle (whatever that may mean).
00484   logical :: therm_reset ! If true, reset running sums of thermodynamic quantities.
00485   real :: cycle_time    ! The length of the coupled time-stepping cycle [T ~> s].
00486   real, dimension(SZI_(CS%G),SZJ_(CS%G)) :: &
00487     ssh         ! sea surface height, which may be based on eta_av [m]
00488
00489   real, dimension(:,:,:), pointer :: &
00490     u => null(), & ! u : zonal velocity component [L T-1 ~> m s-1]
00491     v => null(), & ! v : meridional velocity component [L T-1 ~> m s-1]
00492     h => null()    ! h : layer thickness [H ~> m or kg m-2]
00493   real, dimension(:,:), pointer :: &
00494     p_surf => null() ! A pointer to the ocean surface pressure [R L2 T-2 ~> Pa].
00495   real :: I_wt_ssh  ! The inverse of the time weights [T-1 ~> s-1]
00496
00497   type(time_type) :: Time_local, end_time_thermo, Time_temp
00498   type(group_pass_type) :: pass_tau_ustar_psurf
00499   logical :: showCallTree
00500
00501   ! External forcing fields on the model index map
00502   type(mech_forcing), pointer :: forces     ! Mechanical forcing
00503   type(forcing), pointer :: fluxes          ! Boundary fluxes
00504   type(surface), pointer :: sfc_state_diag  ! Surface boundary fields
00505   integer :: turns  ! Number of quarter turns from input to model indexing
00506
```

```
00507   g => cs%G ; g_in => cs%G_in ; gv => cs%GV ; us => cs%US
00508   is   = g%isc  ; ie   = g%iec  ; js   = g%jsc  ; je   = g%jec ; nz = g%ke
00509   isq  = g%IscB ; ieq  = g%IecB ; jsq  = g%JscB ; jeq  = g%JecB
00510   isd  = g%isd  ; ied  = g%ied  ; jsd  = g%jsd  ; jed  = g%jed
00511   isdb = g%IsdB ; iedb = g%IedB ; jsdb = g%JsdB ; jedb = g%JedB
00512   u => cs%u ; v => cs%v ; h => cs%h
00513
00514   time_interval = us%s_to_T*time_int_in
00515   do_dyn = .true. ; if (present(do_dynamics)) do_dyn = do_dynamics
00516   do_thermo = .true. ; if (present(do_thermodynamics)) do_thermo = do_thermodynamics
00517   if (.not.(do_dyn .or. do_thermo)) call mom_error(fatal,"Step_MOM: "//&
00518     "Both do_dynamics and do_thermodynamics are false, which makes no sense.")
00519   cycle_start = .true. ; if (present(start_cycle)) cycle_start = start_cycle
00520   cycle_end = .true. ; if (present(end_cycle)) cycle_end = end_cycle
00521   cycle_time = time_interval ; if (present(cycle_length)) cycle_time = us%s_to_T*cycle_length
00522   therm_reset = cycle_start ; if (present(reset_therm)) therm_reset = reset_therm
00523
00524   call cpu_clock_begin(id_clock_ocean)
00525   call cpu_clock_begin(id_clock_other)
00526
00527   if (cs%debug) then
00528     call mom_state_chksum("Beginning of step_MOM ", u, v, h, cs%uh, cs%vh, g, gv, us)
00529   endif
00530
00531   showcalltree = calltree_showquery()
00532   if (showcalltree) call calltree_enter("step_MOM(), MOM.F90")
00533
00534   ! Rotate the forces from G_in to G
00535   if (cs%rotate_index) then
00536     turns = g%HI%turns
00537     allocate(forces)
00538     call allocate_mech_forcing(forces_in, g, forces)
00539     call rotate_mech_forcing(forces_in, turns, forces)
00540
00541     allocate(fluxes)
00542     call allocate_forcing_type(fluxes_in, g, fluxes)
00543     call rotate_forcing(fluxes_in, fluxes, turns)
00544   else
00545     forces => forces_in
00546     fluxes => fluxes_in
00547   endif
00548
00549   ! First determine the time step that is consistent with this call and an
00550   ! integer fraction of time_interval.
00551   if (do_dyn) then
00552     n_max = 1
00553     if (time_interval > cs%dt) n_max = ceiling(time_interval/cs%dt - 0.001)
00554     dt = time_interval / real(n_max)
00555     thermo_does_span_coupling = (cs%thermo_spans_coupling .and. &
00556                                  (cs%dt_therm > 1.5*cycle_time))
00557     if (thermo_does_span_coupling) then
00558       ! Set dt_therm to be an integer multiple of the coupling time step.
00559       dt_therm = cycle_time * floor(cs%dt_therm / cycle_time + 0.001)
00560       ntstep = floor(dt_therm/dt + 0.001)
00561     elseif (.not.do_thermo) then
00562       dt_therm = cs%dt_therm
00563       if (present(cycle_length)) dt_therm = min(cs%dt_therm, us%s_to_T*cycle_length)
00564       ! ntstep is not used.
00565     else
00566       ntstep = max(1, min(n_max, floor(cs%dt_therm/dt + 0.001)))
00567       dt_therm = dt*ntstep
00568     endif
00569
00570     if (associated(forces%p_surf)) p_surf => forces%p_surf
00571     if (.not.associated(forces%p_surf)) cs%interp_p_surf = .false.
00572     cs%tv%p_surf => null()
00573     if (cs%use_p_surf_in_EOS .and. associated(forces%p_surf)) cs%tv%p_surf => forces%p_surf
00574
00575     !---------- Initiate group halo pass of the forcing fields
00576     call cpu_clock_begin(id_clock_pass)
00577     call create_group_pass(pass_tau_ustar_psurf, forces%taux, forces%tauy, g%Domain)
00578     if (associated(forces%ustar)) &
00579       call create_group_pass(pass_tau_ustar_psurf, forces%ustar, g%Domain)
00580     if (associated(forces%p_surf)) &
00581       call create_group_pass(pass_tau_ustar_psurf, forces%p_surf, g%Domain)
00582     if (g%nonblocking_updates) then
00583       call start_group_pass(pass_tau_ustar_psurf, g%Domain)
00584     else
00585       call do_group_pass(pass_tau_ustar_psurf, g%Domain)
00586     endif
00587     call cpu_clock_end(id_clock_pass)
00588   else
00589     ! This step only updates the thermodynamics so setting timesteps is simpler.
00590     n_max = 1
00591     if ((time_interval > cs%dt_therm) .and. (cs%dt_therm > 0.0)) &
00592       n_max = ceiling(time_interval/cs%dt_therm - 0.001)
00593
```

```
00594     dt = time_interval / real(n_max)
00595     dt_therm = dt ; ntstep = 1
00596     if (associated(fluxes%p_surf)) p_surf => fluxes%p_surf
00597     cs%tv%p_surf => null()
00598     if (associated(fluxes%p_surf)) then
00599       if (cs%use_p_surf_in_EOS) cs%tv%p_surf => fluxes%p_surf
00600     endif
00601     if (cs%UseWaves) call pass_var(fluxes%ustar, g%Domain, clock=id_clock_pass)
00602   endif
00603
00604   if (therm_reset) then
00605     cs%time_in_thermo_cycle = 0.0
00606     if (associated(cs%tv%frazil))        cs%tv%frazil(:,:)        = 0.0
00607     if (associated(cs%tv%salt_deficit))  cs%tv%salt_deficit(:,:)  = 0.0
00608     if (associated(cs%tv%TempxPmE))      cs%tv%TempxPmE(:,:)      = 0.0
00609     if (associated(cs%tv%internal_heat)) cs%tv%internal_heat(:,:) = 0.0
00610   endif
00611
00612   if (cycle_start) then
00613     cs%time_in_cycle = 0.0
00614     do j=js,je ; do i=is,ie ; cs%ssh_rint(i,j) = 0.0 ; enddo ; enddo
00615
00616     if (associated(cs%VarMix)) then
00617       call enable_averages(cycle_time, time_start + real_to_time(us%T_to_s*cycle_time), cs%diag)
00618       call calc_resoln_function(h, cs%tv, g, gv, us, cs%VarMix)
00619       call calc_depth_function(g, cs%VarMix)
00620       call disable_averaging(cs%diag)
00621     endif
00622   endif
00623
00624   if (do_dyn) then
00625     if (g%nonblocking_updates) &
00626       call complete_group_pass(pass_tau_ustar_psurf, g%Domain, clock=
      id_clock_pass)
00627
00628     if (cs%interp_p_surf) then
00629       if (.not.associated(cs%p_surf_end))   allocate(cs%p_surf_end(isd:ied,jsd:jed))
00630       if (.not.associated(cs%p_surf_begin)) allocate(cs%p_surf_begin(isd:ied,jsd:jed))
00631       if (.not.cs%p_surf_prev_set) then
00632         do j=jsd,jed ; do i=isd,ied
00633           cs%p_surf_prev(i,j) = forces%p_surf(i,j)
00634         enddo ; enddo
00635         cs%p_surf_prev_set = .true.
00636       endif
00637     else
00638       cs%p_surf_end  => forces%p_surf
00639     endif
00640
00641     if (cs%UseWaves) then
00642       ! Update wave information, which is presently kept static over each call to step_mom
00643       call enable_averages(time_interval, time_start + real_to_time(us%T_to_s*time_interval), cs%diag)
00644       call update_stokes_drift(g, gv, us, waves, h, forces%ustar)
00645       call disable_averaging(cs%diag)
00646     endif
00647   else ! not do_dyn.
00648     if (cs%UseWaves) & ! Diagnostics are not enabled in this call.
00649       call update_stokes_drift(g, gv, us, waves, h, fluxes%ustar)
00650   endif
00651
00652   if (cs%debug) then
00653     if (cycle_start) &
00654       call mom_state_chksum("Before steps ", u, v, h, cs%uh, cs%vh, g, gv, us)
00655     if (cycle_start) call check_redundant("Before steps ", u, v, g)
00656     if (do_dyn) call mom_mech_forcing_chksum("Before steps", forces, g, us, haloshift=0)
00657     if (do_dyn) call check_redundant("Before steps ", forces%taux, forces%tauy, g)
00658   endif
00659   call cpu_clock_end(id_clock_other)
00660
00661   rel_time = 0.0
00662   do n=1,n_max
00663     rel_time = rel_time + dt ! The relative time at the end of the step.
00664     ! Set the universally visible time to the middle of the time step.
00665     cs%Time = time_start + real_to_time(us%T_to_s*(rel_time - 0.5*dt))
00666     ! Set the local time to the end of the time step.
00667     time_local = time_start + real_to_time(us%T_to_s*rel_time)
00668
00669     if (showcalltree) call calltree_enter("DT cycles (step_MOM) n=",n)
00670
00671     ! Update the vertically extensive diagnostic grids so that they are
00672     ! referenced to the beginning timestep
00673     call diag_update_remap_grids(cs%diag, update_intensive = .false., update_extensive = .true. )
00674
00675     !===========================================================================
00676     ! This is the first place where the diabatic processes and remapping could occur.
00677     if (cs%diabatic_first .and. (cs%t_dyn_rel_adv==0.0) .and. do_thermo) then ! do thermodynamics.
00678
00679       if (.not.do_dyn) then
```

```
00680         dtdia = dt
00681       elseif (thermo_does_span_coupling) then
00682         dtdia = dt_therm
00683         if ((fluxes%dt_buoy_accum > 0.0) .and. (dtdia > time_interval) .and. &
00684             (abs(fluxes%dt_buoy_accum - dtdia) > 1e-6*dtdia)) then
00685           call mom_error(fatal, "step_MOM: Mismatch between long thermodynamic "//&
00686             "timestep and time over which buoyancy fluxes have been accumulated.")
00687         endif
00688         call mom_error(fatal, "MOM is not yet set up to have restarts that work "//&
00689           "with THERMO_SPANS_COUPLING and DIABATIC_FIRST.")
00690       else
00691         dtdia = dt*min(ntstep,n_max-(n-1))
00692       endif
00693
00694       end_time_thermo = time_local
00695       if (dtdia > dt) then
00696         ! If necessary, temporarily reset CS%Time to the center of the period covered
00697         ! by the call to step_MOM_thermo, noting that they begin at the same time.
00698         cs%Time = cs%Time + real_to_time(0.5*us%T_to_s*(dtdia-dt))
00699         ! The end-time of the diagnostic interval needs to be set ahead if there
00700         ! are multiple dynamic time steps worth of thermodynamics applied here.
00701         end_time_thermo = Time_local + real_to_time(US%T_to_s*(dtdia-dt))
00702       endif
00703
00704       ! Apply diabatic forcing, do mixing, and regrid.
00705       call step_mom_thermo(cs, g, gv, us, u, v, h, cs%tv, fluxes, dtdia, &
00706                            end_time_thermo, .true., Waves=Waves)
00707       cs%time_in_thermo_cycle = cs%time_in_thermo_cycle + dtdia
00708
00709       ! The diabatic processes are now ahead of the dynamics by dtdia.
00710       cs%t_dyn_rel_thermo = -dtdia
00711       if (showcalltree) call calltree_waypoint("finished diabatic_first (step_MOM)")
00712
00713       if (dtdia > dt) & ! Reset CS%Time to its previous value.
00714         cs%Time = time_start + real_to_time(us%T_to_s*(rel_time - 0.5*dt))
00715     endif ! end of block "(CS%diabatic_first .and. (CS%t_dyn_rel_adv==0.0))"
00716
00717     if (do_dyn) then
00718       ! Store pre-dynamics thicknesses for proper diagnostic remapping for transports or
00719       ! advective tendencies.  If there are more than one dynamics steps per advective
00720       ! step (i.e DT_THERM > DT), this needs to be stored at the first dynamics call.
00721       if (.not.cs%preadv_h_stored .and. (cs%t_dyn_rel_adv == 0.)) then
00722         call diag_copy_diag_to_storage(cs%diag_pre_dyn, h, cs%diag)
00723         cs%preadv_h_stored = .true.
00724       endif
00725
00726       ! The pre-dynamics velocities might be stored for debugging truncations.
00727       if (associated(cs%u_prev) .and. associated(cs%v_prev)) then
00728         do k=1,nz ; do j=jsd,jed ; do i=isdb,iedb
00729           cs%u_prev(i,j,k) = u(i,j,k)
00730         enddo ; enddo ; enddo
00731         do k=1,nz ; do j=jsdb,jedb ; do i=isd,ied
00732           cs%v_prev(i,j,k) = v(i,j,k)
00733         enddo ; enddo ; enddo
00734       endif
00735
00736       dt_therm_here = dt_therm
00737       if (do_thermo .and. do_dyn .and. .not.thermo_does_span_coupling) &
00738         dt_therm_here = dt*min(ntstep, n_max-n+1)
00739
00740       ! Indicate whether the bottom boundary layer properties need to be
00741       ! recalculated, and if so for how long an interval they are valid.
00742       bbl_time_int = 0.0
00743       if (do_thermo) then
00744         if ((cs%t_dyn_rel_adv == 0.0) .or. (n==1)) &
00745           bbl_time_int = max(dt, min(dt_therm - cs%t_dyn_rel_adv, dt*(1+n_max-n)) )
00746       else
00747         if ((cs%t_dyn_rel_adv == 0.0) .or. ((n==1) .and. cycle_start)) &
00748           bbl_time_int = min(dt_therm, cycle_time)
00749       endif
00750
00751       if (cs%interp_p_surf) then
00752         wt_end = real(n) / real(n_max)
00753         wt_beg = real(n-1) / real(n_max)
00754         do j=jsd,jed ; do i=isd,ied
00755           cs%p_surf_end(i,j) = wt_end * forces%p_surf(i,j) + &
00756                                (1.0-wt_end) * cs%p_surf_prev(i,j)
00757           cs%p_surf_begin(i,j) = wt_beg * forces%p_surf(i,j) + &
00758                                  (1.0-wt_beg) * cs%p_surf_prev(i,j)
00759         enddo ; enddo
00760       endif
00761
00762       call step_mom_dynamics(forces, cs%p_surf_begin, cs%p_surf_end, dt, &
00763                              dt_therm_here, bbl_time_int, cs, &
00764                              time_local, waves=waves)
00765
00766       !===========================================================================
```

```
00767       ! This is the start of the tracer advection part of the algorithm.
00768
00769       if (thermo_does_span_coupling .or. .not.do_thermo) then
00770         do_advection = (cs%t_dyn_rel_adv + 0.5*dt > dt_therm)
00771       else
00772         do_advection = ((mod(n,ntstep) == 0) .or. (n==n_max))
00773       endif
00774
00775       if (do_advection) then ! Do advective transport and lateral tracer mixing.
00776         call step_mom_tracer_dyn(cs, g, gv, us, h, time_local)
00777         if (cs%diabatic_first .and. abs(cs%t_dyn_rel_thermo) > 1e-6*dt) call mom_error(fatal, &
00778                 "step_MOM: Mismatch between the dynamics and diabatic times "//&
00779                 "with DIABATIC_FIRST.")
00780       endif
00781     endif ! end of (do_dyn)
00782
00783     !============================================================================
00784     ! This is the second place where the diabatic processes and remapping could occur.
00785     if ((cs%t_dyn_rel_adv==0.0) .and. do_thermo .and. (.not.cs%diabatic_first)) then
00786
00787       dtdia = cs%t_dyn_rel_thermo
00788       ! If the MOM6 dynamic and thermodynamic time stepping is being orchestrated
00789       ! by the coupler, the value of diabatic_first does not matter.
00790       if ((cs%t_dyn_rel_thermo==0.0) .and. .not.do_dyn) dtdia = dt
00791
00792       if (cs%thermo_spans_coupling .and. (cs%dt_therm > 1.5*cycle_time) .and. &
00793           (abs(dt_therm - dtdia) > 1e-6*dt_therm)) then
00794         call mom_error(fatal, "step_MOM: Mismatch between dt_therm and dtdia "//&
00795                        "before call to diabatic.")
00796       endif
00797
00798       ! If necessary, temporarily reset CS%Time to the center of the period covered
00799       ! by the call to step_MOM_thermo, noting that they end at the same time.
00800       if (dtdia > dt) cs%Time = cs%Time - real_to_time(0.5*us%T_to_s*(dtdia-dt))
00801
00802       ! Apply diabatic forcing, do mixing, and regrid.
00803       call step_mom_thermo(cs, g, gv, us, u, v, h, cs%tv, fluxes, dtdia, &
00804                            time_local, .false., waves=waves)
00805       cs%time_in_thermo_cycle = cs%time_in_thermo_cycle + dtdia
00806
00807       if ((cs%t_dyn_rel_thermo==0.0) .and. .not.do_dyn) then
00808         ! The diabatic processes are now ahead of the dynamics by dtdia.
00809         cs%t_dyn_rel_thermo = -dtdia
00810       else ! The diabatic processes and the dynamics are synchronized.
00811         cs%t_dyn_rel_thermo = 0.0
00812       endif
00813
00814       if (dtdia > dt) & ! Reset CS%Time to its previous value.
00815         cs%Time = time_start + real_to_time(us%T_to_s*(rel_time - 0.5*dt))
00816     endif
00817
00818     if (do_dyn) then
00819       call cpu_clock_begin(id_clock_dynamics)
00820       ! Determining the time-average sea surface height is part of the algorithm.
00821       ! This may be eta_av if Boussinesq, or need to be diagnosed if not.
00822       cs%time_in_cycle = cs%time_in_cycle + dt
00823       call find_eta(h, cs%tv, g, gv, us, ssh, cs%eta_av_bc, eta_to_m=1.0)
00824       do j=js,je ; do i=is,ie
00825         cs%ssh_rint(i,j) = cs%ssh_rint(i,j) + dt*ssh(i,j)
00826       enddo ; enddo
00827       if (cs%IDs%id_ssh_inst > 0) call post_data(cs%IDs%id_ssh_inst, ssh, cs%diag)
00828       call cpu_clock_end(id_clock_dynamics)
00829     endif
00830
00831     !============================================================================
00832     ! Calculate diagnostics at the end of the time step if the state is self-consistent.
00833     if (mom_state_is_synchronized(cs)) then
00834     !### Perhaps this should be if (CS%t_dyn_rel_thermo == 0.0)
00835       call cpu_clock_begin(id_clock_other) ; call cpu_clock_begin(
      id_clock_diagnostics)
00836       ! Diagnostics that require the complete state to be up-to-date can be calculated.
00837
00838       call enable_averages(cs%t_dyn_rel_diag, time_local, cs%diag)
00839       call calculate_diagnostic_fields(u, v, h, cs%uh, cs%vh, cs%tv, cs%ADp,  &
00840                           cs%CDp, p_surf, cs%t_dyn_rel_diag, cs%diag_pre_sync,&
00841                           g, gv, us, cs%diagnostics_CSp)
00842       call post_tracer_diagnostics_at_sync(cs%Tracer_reg, h, cs%diag_pre_sync, cs%diag, g, gv, cs
      %t_dyn_rel_diag)
00843       call diag_copy_diag_to_storage(cs%diag_pre_sync, h, cs%diag)
00844       if (showcalltree) call calltree_waypoint("finished calculate_diagnostic_fields (step_MOM)")
00845       call disable_averaging(cs%diag)
00846       cs%t_dyn_rel_diag = 0.0
00847
00848       call cpu_clock_end(id_clock_diagnostics) ; call cpu_clock_end(
      id_clock_other)
00849     endif
00850
```

```
00851     if (do_dyn .and. .not.cs%count_calls) cs%nstep_tot = cs%nstep_tot + 1
00852     if (showcalltree) call calltree_leave("DT cycles (step_MOM)")
00853
00854   enddo ! complete the n loop
00855
00856   if (cs%count_calls .and. cycle_start) cs%nstep_tot = cs%nstep_tot + 1
00857
00858   call cpu_clock_begin(id_clock_other)
00859
00860   if (cs%time_in_cycle > 0.0) then
00861     i_wt_ssh = 1.0/cs%time_in_cycle
00862     do j=js,je ; do i=is,ie
00863       ssh(i,j) = cs%ssh_rint(i,j)*i_wt_ssh
00864       cs%ave_ssh_ibc(i,j) = ssh(i,j)
00865     enddo ; enddo
00866     if (do_dyn) then
00867       call adjust_ssh_for_p_atm(cs%tv, g, gv, us, cs%ave_ssh_ibc, forces%p_surf_SSH, &
00868                                 cs%calc_rho_for_sea_lev)
00869     elseif (do_thermo) then
00870       call adjust_ssh_for_p_atm(cs%tv, g, gv, us, cs%ave_ssh_ibc, fluxes%p_surf_SSH, &
00871                                 cs%calc_rho_for_sea_lev)
00872     endif
00873   endif
00874
00875   if (do_dyn .and. cs%interp_p_surf) then ; do j=jsd,jed ; do i=isd,ied
00876     cs%p_surf_prev(i,j) = forces%p_surf(i,j)
00877   enddo ; enddo ; endif
00878
00879   if (cs%ensemble_ocean) then
00880     ! update the time for the next analysis step if needed
00881     call set_analysis_time(cs%Time,cs%odaCS)
00882     ! store ensemble vector in odaCS
00883     call set_prior_tracer(cs%Time, g, gv, cs%h, cs%tv, cs%odaCS)
00884     ! call DA interface
00885     call oda(cs%Time,cs%odaCS)
00886   endif
00887
00888   if (showcalltree) call calltree_waypoint("calling extract_surface_state (step_MOM)")
00889   ! NOTE: sfc_state uses input indexing, since it is also used by drivers.
00890   call extract_surface_state(cs, sfc_state)
00891
00892   ! Do diagnostics that only occur at the end of a complete forcing step.
00893   if (cycle_end) then
00894     if (cs%rotate_index) then
00895       allocate(sfc_state_diag)
00896       call rotate_surface_state(sfc_state, g_in, sfc_state_diag, g, turns)
00897     else
00898       sfc_state_diag => sfc_state
00899     endif
00900
00901     call cpu_clock_begin(id_clock_diagnostics)
00902     if (cs%time_in_cycle > 0.0) then
00903       call enable_averages(cs%time_in_cycle, time_local, cs%diag)
00904       call post_surface_dyn_diags(cs%sfc_IDs, g, cs%diag, sfc_state_diag, ssh)
00905     endif
00906     if (cs%time_in_thermo_cycle > 0.0) then
00907       call enable_averages(cs%time_in_thermo_cycle, time_local, cs%diag)
00908       call post_surface_thermo_diags(cs%sfc_IDs, g, gv, us, cs%diag, cs%time_in_thermo_cycle, &
00909                                      sfc_state_diag, cs%tv, ssh, cs%ave_ssh_ibc)
00910     endif
00911     call disable_averaging(cs%diag)
00912     call cpu_clock_end(id_clock_diagnostics)
00913   endif
00914
00915   ! Accumulate the surface fluxes for assessing conservation
00916   if (do_thermo .and. fluxes%fluxes_used) &
00917     call accumulate_net_input(fluxes, sfc_state, cs%tv, fluxes%dt_buoy_accum, &
00918                               g, us, cs%sum_output_CSp)
00919
00920   if (mom_state_is_synchronized(cs)) &
00921     call write_energy(cs%u, cs%v, cs%h, cs%tv, time_local, cs%nstep_tot, &
00922                       g, gv, us, cs%sum_output_CSp, cs%tracer_flow_CSp, &
00923                       dt_forcing=real_to_time(us%T_to_s*time_interval) )
00924
00925   call cpu_clock_end(id_clock_other)
00926
00927   ! De-rotate fluxes and copy back to the input, since they can be changed.
00928   if (cs%rotate_index) then
00929     call rotate_forcing(fluxes, fluxes_in, -turns)
00930
00931     call deallocate_mech_forcing(forces)
00932     deallocate(forces)
00933
00934     call deallocate_forcing_type(fluxes)
00935     deallocate(fluxes)
00936   endif
00937
```

```
00938   if (showcalltree) call calltree_leave("step_MOM()")
00939   call cpu_clock_end(id_clock_ocean)
00940
00941 end subroutine step_mom
00942
00943 !> Time step the ocean dynamics, including the momentum and continuity equations
00944 subroutine step_mom_dynamics(forces, p_surf_begin, p_surf_end, dt, dt_thermo, &
00945                              bbl_time_int, CS, Time_local, Waves)
00946   type(mech_forcing), intent(in)    :: forces     !< A structure with the driving mechanical forces
00947   real, dimension(:,:), pointer     :: p_surf_begin !< A pointer (perhaps NULL) to the surface
00948                                                     !! pressure at the beginning of this dynamic
00949                                                     !! step, intent in [R L2 T-2 ~> Pa].
00950   real, dimension(:,:), pointer     :: p_surf_end !< A pointer (perhaps NULL) to the surface
00951                                                     !! pressure at the end of this dynamic step,
00952                                                     !! intent in [R L2 T-2 ~> Pa].
00953   real,                 intent(in)  :: dt         !< time interval covered by this call [T ~> s].
00954   real,                 intent(in)  :: dt_thermo  !< time interval covered by any updates that may
00955                                                     !! span multiple dynamics steps [T ~> s].
00956   real,                 intent(in)  :: bbl_time_int !< time interval over which updates to the
00957                                                     !! bottom boundary layer properties will apply [T ~> s],
00958                                                     !! or zero not to update the properties.
00959   type(mom_control_struct), pointer :: CS         !< control structure from
       initialize_MOM
00960   type(time_type),    intent(in)    :: Time_local !< End time of a segment, as a time type
00961   type(wave_parameters_cs), &
00962             optional, pointer       :: Waves      !< Container for wave related parameters; the
00963                                                     !! fields in Waves are intent in here.
00964
00965   ! local variables
00966   type(ocean_grid_type),   pointer :: G => null()  ! pointer to a structure containing
00967                                                    ! metrics and related information
00968   type(verticalgrid_type), pointer :: GV => null() ! Pointer to the vertical grid structure
00969   type(unit_scale_type),   pointer :: US => null() ! Pointer to a structure containing
00970                                                    ! various unit conversion factors
00971   type(mom_diag_ids), pointer :: IDs => null() ! A structure with the diagnostic IDs.
00972   real, dimension(:,:,:), pointer :: &
00973     u => null(), & ! u : zonal velocity component [L T-1 ~> m s-1]
00974     v => null(), & ! v : meridional velocity component [L T-1 ~> m s-1]
00975     h => null()    ! h : layer thickness [H ~> m or kg m-2]
00976
00977   logical :: calc_dtbt  ! Indicates whether the dynamically adjusted
00978                         ! barotropic time step needs to be updated.
00979   logical :: showCallTree
00980
00981   integer :: i, j, k, is, ie, js, je, Isq, Ieq, Jsq, Jeq, nz
00982   integer :: isd, ied, jsd, jed, IsdB, IedB, JsdB, JedB
00983
00984   g => cs%G ; gv => cs%GV ; us => cs%US ; ids => cs%IDs
00985   is   = g%isc  ; ie   = g%iec  ; js   = g%jsc  ; je   = g%jec ; nz = g%ke
00986   isq  = g%IscB ; ieq  = g%IecB ; jsq  = g%JscB ; jeq  = g%JecB
00987   isd  = g%isd  ; ied  = g%ied  ; jsd  = g%jsd  ; jed  = g%jed
00988   isdb = g%IsdB ; iedb = g%IedB ; jsdb = g%JsdB ; jedb = g%JedB
00989   u => cs%u ; v => cs%v ; h => cs%h
00990   showcalltree = calltree_showquery()
00991
00992   call cpu_clock_begin(id_clock_dynamics)
00993
00994   if ((cs%t_dyn_rel_adv == 0.0) .and. cs%thickness_diffuse .and. cs%thickness_diffuse_first) then
00995
00996     call enable_averages(dt_thermo, time_local+real_to_time(us%T_to_s*(dt_thermo-dt)), cs%diag)
00997     call cpu_clock_begin(id_clock_thick_diff)
00998     if (associated(cs%VarMix)) &
00999       call calc_slope_functions(h, cs%tv, dt, g, gv, us, cs%VarMix, obc=cs%OBC)
01000     call thickness_diffuse(h, cs%uhtr, cs%vhtr, cs%tv, dt_thermo, g, gv, us, &
01001                            cs%MEKE, cs%VarMix, cs%CDp, cs%thickness_diffuse_CSp)
01002     call cpu_clock_end(id_clock_thick_diff)
01003     call pass_var(h, g%Domain, clock=id_clock_pass, halo=max(2,cs%cont_stencil))
01004     call disable_averaging(cs%diag)
01005     if (showcalltree) call calltree_waypoint("finished thickness_diffuse_first (step_MOM)")
01006
01007     ! Whenever thickness changes let the diag manager know, target grids
01008     ! for vertical remapping may need to be regenerated.
01009     call diag_update_remap_grids(cs%diag)
01010   endif
01011
01012   ! The bottom boundary layer properties need to be recalculated.
01013   if (bbl_time_int > 0.0) then
01014     call enable_averages(bbl_time_int, &
01015               time_local + real_to_time(us%T_to_s*(bbl_time_int-dt)), cs%diag)
01016     ! Calculate the BBL properties and store them inside visc (u,h).
01017     call cpu_clock_begin(id_clock_bbl_visc)
01018     call set_viscous_bbl(cs%u, cs%v, cs%h, cs%tv, cs%visc, g, gv, us, &
01019                          cs%set_visc_CSp, symmetrize=.true.)
01020     call cpu_clock_end(id_clock_bbl_visc)
01021     if (showcalltree) call calltree_waypoint("done with set_viscous_BBL (step_MOM)")
01022     call disable_averaging(cs%diag)
01023   endif
```

```
01024
01025
01026   if (cs%do_dynamics .and. cs%split) then !--------------------------- start SPLIT
01027     ! This section uses a split time stepping scheme for the dynamic equations,
01028     ! basically the stacked shallow water equations with viscosity.
01029
01030     calc_dtbt = .false.
01031     if (cs%dtbt_reset_period == 0.0) calc_dtbt = .true.
01032     if (cs%dtbt_reset_period > 0.0) then
01033       if (time_local >= cs%dtbt_reset_time) then  !### Change >= to > here.
01034         calc_dtbt = .true.
01035         cs%dtbt_reset_time = cs%dtbt_reset_time + cs%dtbt_reset_interval
01036       endif
01037     endif
01038
01039     call step_mom_dyn_split_rk2(u, v, h, cs%tv, cs%visc, time_local, dt, forces, &
01040                 p_surf_begin, p_surf_end, cs%uh, cs%vh, cs%uhtr, cs%vhtr, &
01041                 cs%eta_av_bc, g, gv, us, cs%dyn_split_RK2_CSp, calc_dtbt, cs%VarMix, &
01042                 cs%MEKE, cs%thickness_diffuse_CSp, waves=waves)
01043     if (showcalltree) call calltree_waypoint("finished step_MOM_dyn_split (step_MOM)")
01044
01045   elseif (cs%do_dynamics) then ! ---------------------------------- not SPLIT
01046     !   This section uses an unsplit stepping scheme for the dynamic
01047     ! equations; basically the stacked shallow water equations with viscosity.
01048     ! Because the time step is limited by CFL restrictions on the external
01049     ! gravity waves, the unsplit is usually much less efficient that the split
01050     ! approaches. But because of its simplicity, the unsplit method is very
01051     ! useful for debugging purposes.
01052
01053     if (cs%use_RK2) then
01054       call step_mom_dyn_unsplit_rk2(u, v, h, cs%tv, cs%visc, time_local, dt, forces, &
01055                p_surf_begin, p_surf_end, cs%uh, cs%vh, cs%uhtr, cs%vhtr, &
01056                cs%eta_av_bc, g, gv, us, cs%dyn_unsplit_RK2_CSp, cs%VarMix, cs%MEKE)
01057     else
01058       call step_mom_dyn_unsplit(u, v, h, cs%tv, cs%visc, time_local, dt, forces, &
01059                p_surf_begin, p_surf_end, cs%uh, cs%vh, cs%uhtr, cs%vhtr, &
01060                cs%eta_av_bc, g, gv, us, cs%dyn_unsplit_CSp, cs%VarMix, cs%MEKE, waves=waves)
01061     endif
01062     if (showcalltree) call calltree_waypoint("finished step_MOM_dyn_unsplit (step_MOM)")
01063
01064   endif ! ------------------------------------------------- end SPLIT
01065
01066   if (cs%thickness_diffuse .and. .not.cs%thickness_diffuse_first) then
01067     call cpu_clock_begin(id_clock_thick_diff)
01068
01069     if (cs%debug) call hchksum(h,"Pre-thickness_diffuse h", g%HI, haloshift=0, scale=gv%H_to_m)
01070
01071     if (associated(cs%VarMix)) &
01072       call calc_slope_functions(h, cs%tv, dt, g, gv, us, cs%VarMix, obc=cs%OBC)
01073     call thickness_diffuse(h, cs%uhtr, cs%vhtr, cs%tv, dt, g, gv, us, &
01074                            cs%MEKE, cs%VarMix, cs%CDp, cs%thickness_diffuse_CSp)
01075
01076     if (cs%debug) call hchksum(h,"Post-thickness_diffuse h", g%HI, haloshift=1, scale=gv%H_to_m)
01077     call cpu_clock_end(id_clock_thick_diff)
01078     call pass_var(h, g%Domain, clock=id_clock_pass, halo=max(2,cs%cont_stencil))
01079     if (showcalltree) call calltree_waypoint("finished thickness_diffuse (step_MOM)")
01080   endif
01081
01082   ! apply the submesoscale mixed layer restratification parameterization
01083   if (cs%mixedlayer_restrat) then
01084     if (cs%debug) then
01085       call hchksum(h,"Pre-mixedlayer_restrat h", g%HI, haloshift=1, scale=gv%H_to_m)
01086       call uvchksum("Pre-mixedlayer_restrat uhtr", &
01087                     cs%uhtr, cs%vhtr, g%HI, haloshift=0, scale=gv%H_to_m*us%L_to_m**2)
01088     endif
01089     call cpu_clock_begin(id_clock_ml_restrat)
01090     call mixedlayer_restrat(h, cs%uhtr, cs%vhtr, cs%tv, forces, dt, cs%visc%MLD, &
01091                             cs%VarMix, g, gv, us, cs%mixedlayer_restrat_CSp)
01092     call cpu_clock_end(id_clock_ml_restrat)
01093     call pass_var(h, g%Domain, clock=id_clock_pass, halo=max(2,cs%cont_stencil))
01094     if (cs%debug) then
01095       call hchksum(h,"Post-mixedlayer_restrat h", g%HI, haloshift=1, scale=gv%H_to_m)
01096       call uvchksum("Post-mixedlayer_restrat [uv]htr", &
01097                     cs%uhtr, cs%vhtr, g%HI, haloshift=0, scale=gv%H_to_m*us%L_to_m**2)
01098     endif
01099   endif
01100
01101   ! Whenever thickness changes let the diag manager know, target grids
01102   ! for vertical remapping may need to be regenerated.
01103   call diag_update_remap_grids(cs%diag)
01104
01105   if (cs%useMEKE) call step_forward_meke(cs%MEKE, h, cs%VarMix%SN_u, cs%VarMix%SN_v, &
01106                                          cs%visc, dt, g, gv, us, cs%MEKE_CSp, cs%uhtr, cs%vhtr)
01107   call disable_averaging(cs%diag)
01108
01109   ! Advance the dynamics time by dt.
01110   cs%t_dyn_rel_adv = cs%t_dyn_rel_adv + dt
```

```
01111   cs%t_dyn_rel_thermo = cs%t_dyn_rel_thermo + dt
01112   if (abs(cs%t_dyn_rel_thermo) < 1e-6*dt) cs%t_dyn_rel_thermo = 0.0
01113   cs%t_dyn_rel_diag = cs%t_dyn_rel_diag + dt
01114
01115   call cpu_clock_end(id_clock_dynamics)
01116
01117   call cpu_clock_begin(id_clock_other) ; call cpu_clock_begin(
      id_clock_diagnostics)
01118   call enable_averages(dt, time_local, cs%diag)
01119   ! These diagnostics are available after every time dynamics step.
01120   if (ids%id_u > 0) call post_data(ids%id_u, u, cs%diag)
01121   if (ids%id_v > 0) call post_data(ids%id_v, v, cs%diag)
01122   if (ids%id_h > 0) call post_data(ids%id_h, h, cs%diag)
01123   call disable_averaging(cs%diag)
01124   call cpu_clock_end(id_clock_diagnostics) ; call cpu_clock_end(
      id_clock_other)
01125
01126 end subroutine step_mom_dynamics
01127
01128 !> step_MOM_tracer_dyn does tracer advection and lateral diffusion, bringing the
01129 !! tracers up to date with the changes in state due to the dynamics.  Surface
01130 !! sources and sinks and remapping are handled via step_MOM_thermo.
01131 subroutine step_mom_tracer_dyn(CS, G, GV, US, h, Time_local)
01132   type(mom_control_struct), intent(inout) :: CS     !< control structure
01133   type(ocean_grid_type),    intent(inout) :: G      !< ocean grid structure
01134   type(verticalgrid_type),  intent(in)    :: GV     !< ocean vertical grid structure
01135   type(unit_scale_type),    intent(in)    :: US     !< A dimensional unit scaling type
01136   real, dimension(SZI_(G),SZJ_(G),SZK_(G)),  &
01137                             intent(in)    :: h      !< layer thicknesses after the transports [H ~> m or kg
       m-2]
01138   type(time_type),          intent(in)    :: Time_local !< The model time at the end
01139                                                     !! of the time step.
01140   type(group_pass_type) :: pass_T_S
01141   integer :: halo_sz ! The size of a halo where data must be valid.
01142   logical :: showCallTree
01143   showcalltree = calltree_showquery()
01144
01145   if (cs%debug) then
01146     call cpu_clock_begin(id_clock_other)
01147     call hchksum(h,"Pre-advection h", g%HI, haloshift=1, scale=gv%H_to_m)
01148     call uvchksum("Pre-advection uhtr", cs%uhtr, cs%vhtr, g%HI, &
01149                   haloshift=0, scale=gv%H_to_m*us%L_to_m**2)
01150     if (associated(cs%tv%T)) call hchksum(cs%tv%T, "Pre-advection T", g%HI, haloshift=1)
01151     if (associated(cs%tv%S)) call hchksum(cs%tv%S, "Pre-advection S", g%HI, haloshift=1)
01152     if (associated(cs%tv%frazil)) call hchksum(cs%tv%frazil, "Pre-advection frazil", g%HI, haloshift=0, &
01153                                                scale=g%US%Q_to_J_kg*g%US%RZ_to_kg_m2)
01154     if (associated(cs%tv%salt_deficit)) call hchksum(cs%tv%salt_deficit, &
01155                    "Pre-advection salt deficit", g%HI, haloshift=0, scale=us%RZ_to_kg_m2)
01156   ! call MOM_thermo_chksum("Pre-advection ", CS%tv, G, US)
01157     call cpu_clock_end(id_clock_other)
01158   endif
01159
01160   call cpu_clock_begin(id_clock_thermo) ; call cpu_clock_begin(
      id_clock_tracer)
01161   call enable_averages(cs%t_dyn_rel_adv, time_local, cs%diag)
01162
01163   call advect_tracer(h, cs%uhtr, cs%vhtr, cs%OBC, cs%t_dyn_rel_adv, g, gv, us, &
01164                      cs%tracer_adv_CSp, cs%tracer_Reg)
01165   call tracer_hordiff(h, cs%t_dyn_rel_adv, cs%MEKE, cs%VarMix, g, gv, us, &
01166                       cs%tracer_diff_CSp, cs%tracer_Reg, cs%tv)
01167   if (showcalltree) call calltree_waypoint("finished tracer advection/diffusion (step_MOM)")
01168   call update_segment_tracer_reservoirs(g, gv, cs%uhtr, cs%vhtr, h, cs%OBC, &
01169                      cs%t_dyn_rel_adv, cs%tracer_Reg)
01170   call cpu_clock_end(id_clock_tracer) ; call cpu_clock_end(
      id_clock_thermo)
01171
01172   call cpu_clock_begin(id_clock_other) ; call cpu_clock_begin(
      id_clock_diagnostics)
01173   call post_transport_diagnostics(g, gv, us, cs%uhtr, cs%vhtr, h, cs%transport_IDs, &
01174            cs%diag_pre_dyn, cs%diag, cs%t_dyn_rel_adv, cs%tracer_reg)
01175   ! Rebuild the remap grids now that we've posted the fields which rely on thicknesses
01176   ! from before the dynamics calls
01177   call diag_update_remap_grids(cs%diag)
01178
01179   call disable_averaging(cs%diag)
01180   call cpu_clock_end(id_clock_diagnostics) ; call cpu_clock_end(
      id_clock_other)
01181
01182   ! Reset the accumulated transports to 0 and record that the dynamics
01183   ! and advective times now agree.
01184   call cpu_clock_begin(id_clock_thermo) ; call cpu_clock_begin(
      id_clock_tracer)
01185   cs%uhtr(:,:,:) = 0.0
01186   cs%vhtr(:,:,:) = 0.0
01187   cs%t_dyn_rel_adv = 0.0
01188   call cpu_clock_end(id_clock_tracer) ; call cpu_clock_end(
      id_clock_thermo)
```

```
01189
01190   if (associated(cs%tv%T)) then
01191     call extract_diabatic_member(cs%diabatic_CSp, diabatic_halo=halo_sz)
01192     if (halo_sz > 0) then
01193       call create_group_pass(pass_t_s, cs%tv%T, g%Domain, to_all, halo=halo_sz)
01194       call create_group_pass(pass_t_s, cs%tv%S, g%Domain, to_all, halo=halo_sz)
01195       call do_group_pass(pass_t_s, g%Domain, clock=id_clock_pass)
01196     elseif (cs%diabatic_first) then
01197       ! Temperature and salinity need halo updates because they will be used
01198       ! in the dynamics before they are changed again.
01199       call create_group_pass(pass_t_s, cs%tv%T, g%Domain, to_all+omit_corners, halo=1)
01200       call create_group_pass(pass_t_s, cs%tv%S, g%Domain, to_all+omit_corners, halo=1)
01201       call do_group_pass(pass_t_s, g%Domain, clock=id_clock_pass)
01202     endif
01203   endif
01204
01205   cs%preadv_h_stored = .false.
01206
01207 end subroutine step_mom_tracer_dyn
01208
01209 !> MOM_step_thermo orchestrates the thermodynamic time stepping and vertical
01210 !! remapping, via calls to diabatic (or adiabatic) and ALE_main.
01211 subroutine step_mom_thermo(CS, G, GV, US, u, v, h, tv, fluxes, dtdia, &
01212                            Time_end_thermo, update_BBL, Waves)
01213   type(mom_control_struct), intent(inout) :: CS     !< Master MOM control structure
01214   type(ocean_grid_type),    intent(inout) :: G      !< ocean grid structure
01215   type(verticalgrid_type),  intent(inout) :: GV     !< ocean vertical grid structure
01216   type(unit_scale_type),    intent(in)    :: US     !< A dimensional unit scaling type
01217   real, dimension(SZIB_(G),SZJ_(G),SZK_(G)), &
01218                             intent(inout) :: u      !< zonal velocity [L T-1 ~> m s-1]
01219   real, dimension(SZI_(G),SZJB_(G),SZK_(G)), &
01220                             intent(inout) :: v      !< meridional velocity [L T-1 ~> m s-1]
01221   real, dimension(SZI_(G),SZJ_(G),SZK_(G)),  &
01222                             intent(inout) :: h      !< layer thickness [H ~> m or kg m-2]
01223   type(thermo_var_ptrs),    intent(inout) :: tv     !< A structure pointing to various thermodynamic
      variables
01224   type(forcing),            intent(inout) :: fluxes !< pointers to forcing fields
01225   real,                     intent(in)    :: dtdia  !< The time interval over which to advance [T ~> s]
01226   type(time_type),          intent(in)    :: Time_end_thermo !< End of averaging interval for thermo diags
01227   logical,                  intent(in)    :: update_BBL !< If true, calculate the bottom boundary layer
      properties.
01228   type(wave_parameters_cs), &
01229                   optional, pointer       :: Waves  !< Container for wave related parameters
01230                                                     !! the fields in Waves are intent in here.
01231
01232   logical :: use_ice_shelf ! Needed for selecting the right ALE interface.
01233   logical :: showCallTree
01234   type(group_pass_type) :: pass_T_S, pass_T_S_h, pass_uv_T_S_h
01235   integer :: dynamics_stencil  ! The computational stencil for the calculations
01236                                ! in the dynamic core.
01237   integer :: halo_sz ! The size of a halo where data must be valid.
01238   integer :: i, j, k, is, ie, js, je, nz
01239
01240   is = g%isc ; ie = g%iec ; js = g%jsc ; je = g%jec ; nz = g%ke
01241   showcalltree = calltree_showquery()
01242   if (showcalltree) call calltree_enter("step_MOM_thermo(), MOM.F90")
01243
01244   use_ice_shelf = .false.
01245   if (associated(fluxes%frac_shelf_h)) use_ice_shelf = .true.
01246
01247   call enable_averages(dtdia, time_end_thermo, cs%diag)
01248
01249   if (associated(cs%odaCS)) then
01250     call apply_oda_tracer_increments(us%T_to_s*dtdia,g,tv,h,cs%odaCS)
01251   endif
01252
01253   if (associated(fluxes%p_surf) .or. associated(fluxes%p_surf_full)) then
01254     call extract_diabatic_member(cs%diabatic_CSp, diabatic_halo=halo_sz)
01255     if (halo_sz > 0) then
01256       if (associated(fluxes%p_surf_full)) &
01257         call pass_var(fluxes%p_surf_full, g%Domain, &
01258                       clock=id_clock_pass, halo=halo_sz, complete=.not.associated(fluxes
      %p_surf))
01259       call pass_var(fluxes%p_surf, g%Domain, clock=id_clock_pass, halo=halo_sz, complete=.true
      .)
01260     endif
01261   endif
01262
01263   if (update_bbl) then
01264     !   Calculate the BBL properties and store them inside visc (u,h).
01265     ! This is here so that CS%visc is updated before diabatic() when
01266     ! DIABATIC_FIRST=True. Otherwise diabatic() is called after the dynamics
01267     ! and set_viscous_BBL is called as a part of the dynamic stepping.
01268     call cpu_clock_begin(id_clock_bbl_visc)
01269     call set_viscous_bbl(u, v, h, tv, cs%visc, g, gv, us, cs%set_visc_CSp, symmetrize=.true.)
01270     call cpu_clock_end(id_clock_bbl_visc)
01271     if (showcalltree) call calltree_waypoint("done with set_viscous_BBL (step_MOM_thermo)")
```

```
01272   endif
01273
01274   call cpu_clock_begin(id_clock_thermo)
01275   if (.not.cs%adiabatic) then
01276     if (cs%debug) then
01277       call uvchksum("Pre-diabatic [uv]", u, v, g%HI, haloshift=2, scale=us%L_T_to_m_s)
01278       call hchksum(h,"Pre-diabatic h", g%HI, haloshift=1, scale=gv%H_to_m)
01279       call uvchksum("Pre-diabatic [uv]h", cs%uhtr, cs%vhtr, g%HI, &
01280                     haloshift=0, scale=gv%H_to_m*us%L_to_m**2)
01281     ! call MOM_state_chksum("Pre-diabatic ", u, v, h, CS%uhtr, CS%vhtr, G, GV, vel_scale=1.0)
01282       call mom_thermo_chksum("Pre-diabatic ", tv, g, us, haloshift=0)
01283       call check_redundant("Pre-diabatic ", u, v, g)
01284       call mom_forcing_chksum("Pre-diabatic", fluxes, g, us, haloshift=0)
01285     endif
01286
01287     call cpu_clock_begin(id_clock_diabatic)
01288
01289     call diabatic(u, v, h, tv, cs%Hml, fluxes, cs%visc, cs%ADp, cs%CDp, dtdia, &
01290                   time_end_thermo, g, gv, us, cs%diabatic_CSp, obc=cs%OBC, waves=waves)
01291     fluxes%fluxes_used = .true.
01292
01293     if (showcalltree) call calltree_waypoint("finished diabatic (step_MOM_thermo)")
01294
01295     ! Regridding/remapping is done here, at end of thermodynamics time step
01296     ! (that may comprise several dynamical time steps)
01297     ! The routine 'ALE_main' can be found in 'MOM_ALE.F90'.
01298     if ( cs%use_ALE_algorithm ) then
01299       call enable_averages(dtdia, time_end_thermo, cs%diag)
01300 !       call pass_vector(u, v, G%Domain)
01301       call cpu_clock_begin(id_clock_pass)
01302       if (associated(tv%T)) &
01303         call create_group_pass(pass_t_s_h, tv%T, g%Domain, to_all+omit_corners, halo=1)
01304       if (associated(tv%S)) &
01305         call create_group_pass(pass_t_s_h, tv%S, g%Domain, to_all+omit_corners, halo=1)
01306       call create_group_pass(pass_t_s_h, h, g%Domain, to_all+omit_corners, halo=1)
01307       call do_group_pass(pass_t_s_h, g%Domain)
01308       call cpu_clock_end(id_clock_pass)
01309
01310       call preale_tracer_diagnostics(cs%tracer_Reg, g, gv)
01311
01312       if (cs%debug) then
01313         call mom_state_chksum("Pre-ALE ", u, v, h, cs%uh, cs%vh, g, gv, us)
01314         call hchksum(tv%T,"Pre-ALE T", g%HI, haloshift=1)
01315         call hchksum(tv%S,"Pre-ALE S", g%HI, haloshift=1)
01316         call check_redundant("Pre-ALE ", u, v, g)
01317       endif
01318       call cpu_clock_begin(id_clock_ale)
01319       if (use_ice_shelf) then
01320         call ale_main(g, gv, us, h, u, v, tv, cs%tracer_Reg, cs%ALE_CSp, cs%OBC, &
01321                       dtdia, fluxes%frac_shelf_h)
01322       else
01323         call ale_main(g, gv, us, h, u, v, tv, cs%tracer_Reg, cs%ALE_CSp, cs%OBC, dtdia)
01324       endif
01325
01326       if (showcalltree) call calltree_waypoint("finished ALE_main (step_MOM_thermo)")
01327       call cpu_clock_end(id_clock_ale)
01328     endif   ! endif for the block "if ( CS%use_ALE_algorithm )"
01329
01330     dynamics_stencil = min(3, g%Domain%nihalo, g%Domain%njhalo)
01331     call create_group_pass(pass_uv_t_s_h, u, v, g%Domain, halo=dynamics_stencil)
01332     if (associated(tv%T)) &
01333       call create_group_pass(pass_uv_t_s_h, tv%T, g%Domain, halo=dynamics_stencil)
01334     if (associated(tv%S)) &
01335       call create_group_pass(pass_uv_t_s_h, tv%S, g%Domain, halo=dynamics_stencil)
01336     call create_group_pass(pass_uv_t_s_h, h, g%Domain, halo=dynamics_stencil)
01337     call do_group_pass(pass_uv_t_s_h, g%Domain, clock=id_clock_pass)
01338
01339     if (cs%debug .and. cs%use_ALE_algorithm) then
01340       call mom_state_chksum("Post-ALE ", u, v, h, cs%uh, cs%vh, g, gv, us)
01341       call hchksum(tv%T, "Post-ALE T", g%HI, haloshift=1)
01342       call hchksum(tv%S, "Post-ALE S", g%HI, haloshift=1)
01343       call check_redundant("Post-ALE ", u, v, g)
01344     endif
01345
01346     ! Whenever thickness changes let the diag manager know, target grids
01347     ! for vertical remapping may need to be regenerated. This needs to
01348     ! happen after the H update and before the next post_data.
01349     call diag_update_remap_grids(cs%diag)
01350
01351     !### Consider moving this up into the if ALE block.
01352     call postale_tracer_diagnostics(cs%tracer_Reg, g, gv, cs%diag, dtdia)
01353
01354     if (cs%debug) then
01355       call uvchksum("Post-diabatic u", u, v, g%HI, haloshift=2, scale=us%L_T_to_m_s)
01356       call hchksum(h, "Post-diabatic h", g%HI, haloshift=1, scale=gv%H_to_m)
01357       call uvchksum("Post-diabatic [uv]h", cs%uhtr, cs%vhtr, g%HI, &
01358                     haloshift=0, scale=gv%H_to_m*us%L_to_m**2)
```

```
01359     ! call MOM_state_chksum("Post-diabatic ", u, v, &
01360     !                       h, CS%uhtr, CS%vhtr, G, GV, haloshift=1)
01361       if (associated(tv%T)) call hchksum(tv%T, "Post-diabatic T", g%HI, haloshift=1)
01362       if (associated(tv%S)) call hchksum(tv%S, "Post-diabatic S", g%HI, haloshift=1)
01363       if (associated(tv%frazil)) call hchksum(tv%frazil, "Post-diabatic frazil", g%HI, haloshift=0, &
01364                                               scale=g%US%Q_to_J_kg*g%US%RZ_to_kg_m2)
01365       if (associated(tv%salt_deficit)) call hchksum(tv%salt_deficit, &
01366                                 "Post-diabatic salt deficit", g%HI, haloshift=0, scale=us%RZ_to_kg_m2)
01367     ! call MOM_thermo_chksum("Post-diabatic ", tv, G, US)
01368       call check_redundant("Post-diabatic ", u, v, g)
01369     endif
01370     call disable_averaging(cs%diag)
01371
01372     call cpu_clock_end(id_clock_diabatic)
01373   else   ! complement of "if (.not.CS%adiabatic)"
01374
01375     call cpu_clock_begin(id_clock_adiabatic)
01376     call adiabatic(h, tv, fluxes, dtdia, g, gv, us, cs%diabatic_CSp)
01377     fluxes%fluxes_used = .true.
01378     call cpu_clock_end(id_clock_adiabatic)
01379
01380     if (associated(tv%T)) then
01381       call create_group_pass(pass_t_s, tv%T, g%Domain, to_all+omit_corners, halo=1)
01382       call create_group_pass(pass_t_s, tv%S, g%Domain, to_all+omit_corners, halo=1)
01383       call do_group_pass(pass_t_s, g%Domain, clock=id_clock_pass)
01384       if (cs%debug) then
01385         if (associated(tv%T)) call hchksum(tv%T, "Post-diabatic T", g%HI, haloshift=1)
01386         if (associated(tv%S)) call hchksum(tv%S, "Post-diabatic S", g%HI, haloshift=1)
01387       endif
01388     endif
01389
01390   endif   ! endif for the block "if (.not.CS%adiabatic)"
01391   call cpu_clock_end(id_clock_thermo)
01392
01393   call disable_averaging(cs%diag)
01394
01395   if (showcalltree) call calltree_leave("step_MOM_thermo(), MOM.F90")
01396
01397 end subroutine step_mom_thermo
01398
01399
01400 !> step_offline is the main driver for running tracers offline in MOM6. This has been primarily
01401 !! developed with ALE configurations in mind. Some work has been done in isopycnal configuration, but
01402 !! the work is very preliminary. Some more detail about this capability along with some of the subroutines
01403 !! called here can be found in tracers/MOM_offline_control.F90
01404 subroutine step_offline(forces, fluxes, sfc_state, Time_start, time_interval, CS)
01405   type(mech_forcing), intent(in)    :: forces        !< A structure with the driving mechanical forces
01406   type(forcing),      intent(inout) :: fluxes        !< pointers to forcing fields
01407   type(surface),      intent(inout) :: sfc_state     !< surface ocean state
01408   type(time_type),    intent(in)    :: Time_start    !< starting time of a segment, as a time type
01409   real,               intent(in)    :: time_interval !< time interval
01410   type(mom_control_struct), pointer :: CS            !< control structure from
      initialize_MOM
01411
01412   ! Local pointers
01413   type(ocean_grid_type),      pointer :: G  => null() ! Pointer to a structure containing
01414                                                       ! metrics and related information
01415   type(verticalgrid_type),    pointer :: GV => null() ! Pointer to structure containing information
01416                                                       ! about the vertical grid
01417   type(unit_scale_type),      pointer :: US => null() ! Pointer to a structure containing
01418                                                       ! various unit conversion factors
01419
01420   logical :: first_iter    !< True if this is the first time step_offline has been called in a given
      interval
01421   logical :: last_iter     !< True if this is the last time step_tracer is to be called in an offline
      interval
01422   logical :: do_vertical   !< If enough time has elapsed, do the diabatic tracer sources/sinks
01423   logical :: adv_converged !< True if all the horizontal fluxes have been used
01424
01425   real :: dt_offline          ! The offline timestep for advection [T ~> s]
01426   real :: dt_offline_vertical ! The offline timestep for vertical fluxes and remapping [T ~> s]
01427   logical :: skip_diffusion
01428   integer :: id_eta_diff_end
01429
01430   type(time_type), pointer :: accumulated_time => null()
01431   type(time_type), pointer :: vertical_time => null()
01432   integer :: i,j,k
01433   integer :: is, ie, js, je, isd, ied, jsd, jed
01434
01435   ! 3D pointers
01436   real, dimension(:,:,:), pointer :: &
01437     uhtr => null(), vhtr => null(), &
01438     eatr => null(), ebtr => null(), &
01439     h_end => null()
01440
01441   ! 2D Array for diagnostics
01442   real, dimension(SZI_(CS%G),SZJ_(CS%G)) :: eta_pre, eta_end
```

```
01443   type(time_type) :: Time_end    ! End time of a segment, as a time type
01444
01445   ! Grid-related pointer assignments
01446   g => cs%G ; gv => cs%GV ; us => cs%US
01447
01448   is  = g%isc  ; ie  = g%iec  ; js  = g%jsc  ; je  = g%jec
01449   isd = g%isd  ; ied = g%ied  ; jsd = g%jsd  ; jed = g%jed
01450
01451   call cpu_clock_begin(id_clock_offline_tracer)
01452   call extract_offline_main(cs%offline_CSp, uhtr, vhtr, eatr, ebtr, h_end, accumulated_time, &
01453                             vertical_time, dt_offline, dt_offline_vertical, skip_diffusion)
01454   time_end = increment_date(time_start, seconds=floor(time_interval+0.001))
01455
01456   call enable_averaging(time_interval, time_end, cs%diag)
01457
01458   ! Check to see if this is the first iteration of the offline interval
01459   if (accumulated_time == real_to_time(0.0)) then
01460     first_iter = .true.
01461   else ! This is probably unnecessary but is used to guard against unwanted behavior
01462     first_iter = .false.
01463   endif
01464
01465   ! Check to see if vertical tracer functions should be done
01466   if (first_iter .or. (accumulated_time >= vertical_time)) then
01467     do_vertical = .true.
01468     vertical_time = accumulated_time + real_to_time(us%T_to_s*dt_offline_vertical)
01469   else
01470     do_vertical = .false.
01471   endif
01472
01473   ! Increment the amount of time elapsed since last read and check if it's time to roll around
01474   accumulated_time = accumulated_time + real_to_time(time_interval)
01475
01476   last_iter = (accumulated_time >= real_to_time(us%T_to_s*dt_offline))
01477
01478   if (cs%use_ALE_algorithm) then
01479     ! If this is the first iteration in the offline timestep, then we need to read in fields and
01480     ! perform the main advection.
01481     if (first_iter) then
01482       call mom_mesg("Reading in new offline fields")
01483       ! Read in new transport and other fields
01484       ! call update_transport_from_files(G, GV, CS%offline_CSp, h_end, eatr, ebtr, uhtr, vhtr, &
01485       !     CS%tv%T, CS%tv%S, fluxes, CS%use_ALE_algorithm)
01486       ! call update_transport_from_arrays(CS%offline_CSp)
01487       call update_offline_fields(cs%offline_CSp, cs%h, fluxes, cs%use_ALE_algorithm)
01488
01489       ! Apply any fluxes into the ocean
01490       call offline_fw_fluxes_into_ocean(g, gv, cs%offline_CSp, fluxes, cs%h)
01491
01492       if (.not.cs%diabatic_first) then
01493         call offline_advection_ale(fluxes, time_start, time_interval, cs%offline_CSp,
    id_clock_ale, &
01494             cs%h, uhtr, vhtr, converged=adv_converged)
01495
01496         ! Redistribute any remaining transport
01497         call offline_redistribute_residual(cs%offline_CSp, cs%h, uhtr, vhtr, adv_converged)
01498
01499         ! Perform offline diffusion if requested
01500         if (.not. skip_diffusion) then
01501           if (associated(cs%VarMix)) then
01502             call pass_var(cs%h, g%Domain)
01503             call calc_resoln_function(cs%h, cs%tv, g, gv, us, cs%VarMix)
01504             call calc_depth_function(g, cs%VarMix)
01505             call calc_slope_functions(cs%h, cs%tv, dt_offline, g, gv, us, cs%VarMix, obc=cs%OBC)
01506           endif
01507           call tracer_hordiff(cs%h, dt_offline, cs%MEKE, cs%VarMix, g, gv, us, &
01508               cs%tracer_diff_CSp, cs%tracer_Reg, cs%tv)
01509         endif
01510       endif
01511     endif
01512     ! The functions related to column physics of tracers is performed separately in ALE mode
01513     if (do_vertical) then
01514       call offline_diabatic_ale(fluxes, time_start, time_end, cs%offline_CSp, cs%h, eatr, ebtr)
01515     endif
01516
01517     ! Last thing that needs to be done is the final ALE remapping
01518     if (last_iter) then
01519       if (cs%diabatic_first) then
01520         call offline_advection_ale(fluxes, time_start, time_interval, cs%offline_CSp,
    id_clock_ale, &
01521             cs%h, uhtr, vhtr, converged=adv_converged)
01522
01523         ! Redistribute any remaining transport and perform the remaining advection
01524         call offline_redistribute_residual(cs%offline_CSp, cs%h, uhtr, vhtr, adv_converged)
01525                 ! Perform offline diffusion if requested
01526         if (.not. skip_diffusion) then
01527           if (associated(cs%VarMix)) then
```

```
01528             call pass_var(cs%h, g%Domain)
01529             call calc_resoln_function(cs%h, cs%tv, g, gv, us, cs%VarMix)
01530             call calc_depth_function(g, cs%VarMix)
01531             call calc_slope_functions(cs%h, cs%tv, dt_offline, g, gv, us, cs%VarMix, obc=cs%OBC)
01532           endif
01533           call tracer_hordiff(cs%h, dt_offline, cs%MEKE, cs%VarMix, g, gv, us, &
01534               cs%tracer_diff_CSp, cs%tracer_Reg, cs%tv)
01535         endif
01536       endif
01537
01538       call mom_mesg("Last iteration of offline interval")
01539
01540       ! Apply freshwater fluxes out of the ocean
01541       call offline_fw_fluxes_out_ocean(g, gv, cs%offline_CSp, fluxes, cs%h)
01542       ! These diagnostic can be used to identify which grid points did not converge within
01543       ! the specified number of advection sub iterations
01544       call post_offline_convergence_diags(cs%offline_CSp, cs%h, h_end, uhtr, vhtr)
01545
01546       ! Call ALE one last time to make sure that tracers are remapped onto the layer thicknesses
01547       ! stored from the forward run
01548       call cpu_clock_begin(id_clock_ale)
01549       call ale_offline_tracer_final( g, gv, cs%h, cs%tv, h_end, cs%tracer_Reg, cs%ALE_CSp, cs%OBC)
01550       call cpu_clock_end(id_clock_ale)
01551       call pass_var(cs%h, g%Domain)
01552     endif
01553   else ! NON-ALE MODE...NOT WELL TESTED
01554     call mom_error(warning, &
01555         "Offline tracer mode in non-ALE configuration has not been thoroughly tested")
01556     ! Note that for the layer mode case, the calls to tracer sources and sinks is embedded in
01557     ! main_offline_advection_layer. Warning: this may not be appropriate for tracers that
01558     ! exchange with the atmosphere
01559     if (abs(time_interval - us%T_to_s*dt_offline) > 1.0e-6) then
01560       call mom_error(fatal, &
01561           "For offline tracer mode in a non-ALE configuration, dt_offline must equal time_interval")
01562     endif
01563     call update_offline_fields(cs%offline_CSp, cs%h, fluxes, cs%use_ALE_algorithm)
01564     call offline_advection_layer(fluxes, time_start, time_interval, cs%offline_CSp, &
01565         cs%h, eatr, ebtr, uhtr, vhtr)
01566     ! Perform offline diffusion if requested
01567     if (.not. skip_diffusion) then
01568       call tracer_hordiff(h_end, dt_offline, cs%MEKE, cs%VarMix, g, gv, us, &
01569                           cs%tracer_diff_CSp, cs%tracer_Reg, cs%tv)
01570     endif
01571
01572     cs%h = h_end
01573
01574     call pass_var(cs%tv%T, g%Domain)
01575     call pass_var(cs%tv%S, g%Domain)
01576     call pass_var(cs%h, g%Domain)
01577
01578   endif
01579
01580   call adjust_ssh_for_p_atm(cs%tv, g, gv, us, cs%ave_ssh_ibc, forces%p_surf_SSH, &
01581                             cs%calc_rho_for_sea_lev)
01582   call extract_surface_state(cs, sfc_state)
01583
01584   call disable_averaging(cs%diag)
01585   call pass_var(cs%tv%T, g%Domain)
01586   call pass_var(cs%tv%S, g%Domain)
01587   call pass_var(cs%h, g%Domain)
01588
01589   fluxes%fluxes_used = .true.
01590
01591   if (last_iter) then
01592     accumulated_time = real_to_time(0.0)
01593   endif
01594
01595   call cpu_clock_end(id_clock_offline_tracer)
01596
01597 end subroutine step_offline
01598
01599 !> Initialize MOM, including memory allocation, setting up parameters and diagnostics,
01600 !! initializing the ocean state variables, and initializing subsidiary modules
01601 subroutine initialize_mom(Time, Time_init, param_file, dirs, CS, restart_CSp, &
01602                           Time_in, offline_tracer_mode, input_restart_file, diag_ptr, &
01603                           count_calls, tracer_flow_CSp)
01604   type(time_type), target,   intent(inout) :: Time        !< model time, set in this routine
01605   type(time_type),           intent(in)    :: Time_init   !< The start time for the coupled model's
      calendar
01606   type(param_file_type),     intent(out)   :: param_file  !< structure indicating parameter file to parse
01607   type(directories),         intent(out)   :: dirs        !< structure with directory paths
01608   type(mom_control_struct), pointer       :: CS          !< pointer set in this routine
      to MOM control structure
01609   type(mom_restart_cs),      pointer       :: restart_CSp !< pointer set in this routine to the
01610                                                           !! restart control structure that will
01611                                                           !! be used for MOM.
01612   type(time_type), optional, intent(in)    :: Time_in     !< time passed to MOM_initialize_state when
```

```
01613                                                       !! model is not being started from a restart file
01614   logical,          optional, intent(out)   :: offline_tracer_mode !< True is returned if tracers are being
      run offline
01615   character(len=*),optional, intent(in)    :: input_restart_file !< If present, name of restart file to
      read
01616   type(diag_ctrl), optional, pointer       :: diag_ptr    !< A pointer set in this routine to the
      diagnostic
01617                                                       !! regulatory structure
01618   type(tracer_flow_control_cs), &
01619                     optional, pointer       :: tracer_flow_CSp !< A pointer set in this routine to
01620                                                       !! the tracer flow control structure.
01621   logical,          optional, intent(in)    :: count_calls !< If true, nstep_tot counts the number of
01622                                                       !! calls to step_MOM instead of the number of
01623                                                       !! dynamics timesteps.
01624   ! local variables
01625   type(ocean_grid_type),  pointer :: G => null()    ! A pointer to the metric grid use for the run
01626   type(ocean_grid_type),  pointer :: G_in => null() ! Pointer to the input grid
01627   type(hor_index_type),   pointer :: HI => null()   ! A hor_index_type for array extents
01628   type(hor_index_type),   target  :: HI_in          ! HI on the input grid
01629   type(verticalgrid_type), pointer :: GV => null()
01630   type(dyn_horgrid_type), pointer :: dG => null()
01631   type(dyn_horgrid_type), pointer :: dG_in => null()
01632   type(diag_ctrl),        pointer :: diag => null()
01633   type(unit_scale_type),  pointer :: US => null()
01634   character(len=4), parameter :: vers_num = 'v2.0'
01635   integer :: turns   ! Number of grid quarter-turns
01636
01637   ! Initial state on the input index map
01638   real, allocatable, dimension(:,:,:) :: u_in, v_in, h_in
01639   real, allocatable, dimension(:,:,:), target :: T_in, S_in
01640   type(ocean_obc_type), pointer :: OBC_in => null()
01641   type(sponge_cs), pointer :: sponge_in_CSp => null()
01642   type(ale_sponge_cs), pointer :: ALE_sponge_in_CSp => null()
01643
01644   ! This include declares and sets the variable "version".
01645 # include "version_variable.h"
01646
01647   integer :: i, j, k, is, ie, js, je, isd, ied, jsd, jed, nz
01648   integer :: IsdB, IedB, JsdB, JedB
01649   real    :: dtbt        ! The barotropic timestep [s]
01650
01651   real, allocatable, dimension(:,:)   :: eta ! free surface height or column mass [H ~> m or kg m-2]
01652   real, allocatable, dimension(:,:)   :: area_shelf_h ! area occupied by ice shelf [L2 ~> m2]
01653   real, dimension(:,:), allocatable, target  :: frac_shelf_h ! fraction of total area occupied by ice shelf
      [nondim]
01654   real, dimension(:,:), pointer :: shelf_area => null()
01655   type(mom_restart_cs),  pointer      :: restart_CSp_tmp => null()
01656   type(group_pass_type) :: tmp_pass_uv_T_S_h, pass_uv_T_S_h
01657
01658   real    :: default_val       ! default value for a parameter
01659   logical :: write_geom_files  ! If true, write out the grid geometry files.
01660   logical :: ensemble_ocean    ! If true, perform an ensemble gather at the end of step_MOM
01661   logical :: new_sim
01662   logical :: use_geothermal    ! If true, apply geothermal heating.
01663   logical :: use_EOS           ! If true, density calculated from T & S using an equation of state.
01664   logical :: symmetric         ! If true, use symmetric memory allocation.
01665   logical :: save_IC           ! If true, save the initial conditions.
01666   logical :: do_unit_tests     ! If true, call unit tests.
01667   logical :: test_grid_copy = .false.
01668
01669   logical :: bulkmixedlayer    ! If true, a refined bulk mixed layer scheme is used
01670                                ! with nkml sublayers and nkbl buffer layer.
01671   logical :: use_temperature   ! If true, temp and saln used as state variables.
01672   logical :: use_frazil        ! If true, liquid seawater freezes if temp below freezing,
01673                                ! with accumulated heat deficit returned to surface ocean.
01674   logical :: bound_salinity    ! If true, salt is added to keep salinity above
01675                                ! a minimum value, and the deficit is reported.
01676   logical :: default_2018_answers ! The default setting for the various 2018_ANSWERS flags.
01677   logical :: use_conT_absS     ! If true, the prognostics T & S are conservative temperature
01678                                ! and absolute salinity. Care should be taken to convert them
01679                                ! to potential temperature and practical salinity before
01680                                ! exchanging them with the coupler and/or reporting T&S diagnostics.
01681   logical :: advect_TS         ! If false, then no horizontal advection of temperature
01682                                ! and salnity is performed
01683   logical :: use_ice_shelf     ! Needed for ALE
01684   logical :: global_indexing   ! If true use global horizontal index values instead
01685                                ! of having the data domain on each processor start at 1.
01686   logical :: bathy_at_vel      ! If true, also define bathymetric fields at the
01687                                ! the velocity points.
01688   logical :: calc_dtbt         ! Indicates whether the dynamically adjusted barotropic
01689                                ! time step needs to be updated before it is used.
01690   logical :: debug_truncations ! If true, turn on diagnostics useful for debugging truncations.
01691   integer :: first_direction   ! An integer that indicates which direction is to be
01692                                ! updated first in directionally split parts of the
01693                                ! calculation.  This can be altered during the course
01694                                ! of the run via calls to set_first_direction.
01695   integer :: nkml, nkbl, verbosity, write_geom
```

```
01696   integer :: dynamics_stencil  ! The computational stencil for the calculations
01697                                ! in the dynamic core.
01698   real :: conv2watt, conv2salt
01699   real :: RL2_T2_rescale, Z_rescale, QRZ_rescale ! Unit conversion factors
01700   character(len=48) :: flux_units, S_flux_units
01701
01702   type(vardesc) :: vd_T, vd_S  ! Structures describing temperature and salinity variables.
01703   type(time_type)                :: Start_time
01704   type(ocean_internal_state)     :: MOM_internal_state
01705   character(len=200) :: area_varname, ice_shelf_file, inputdir, filename
01706
01707   if (associated(cs)) then
01708     call mom_error(warning, "initialize_MOM called with an associated "// &
01709                             "control structure.")
01710     return
01711   endif
01712   allocate(cs)
01713
01714   cs%Time => time
01715
01716   id_clock_init = cpu_clock_id('Ocean Initialization', grain=clock_subcomponent)
01717   call cpu_clock_begin(id_clock_init)
01718
01719   start_time = time ; if (present(time_in)) start_time = time_in
01720
01721   ! Read paths and filenames from namelist and store in "dirs".
01722   ! Also open the parsed input parameter file(s) and setup param_file.
01723   call get_mom_input(param_file, dirs, default_input_filename=input_restart_file)
01724
01725   verbosity = 2 ; call read_param(param_file, "VERBOSITY", verbosity)
01726   call mom_set_verbosity(verbosity)
01727   call calltree_enter("initialize_MOM(), MOM.F90")
01728
01729   call find_obsolete_params(param_file)
01730
01731   ! Determining the internal unit scaling factors for this run.
01732   call unit_scaling_init(param_file, cs%US)
01733   us => cs%US
01734
01735   ! Read relevant parameters and write them to the model log.
01736   call log_version(param_file, "MOM", version, "", log_to_all=.true., layout=.true., debugging=.true.)
01737   call get_param(param_file, "MOM", "VERBOSITY", verbosity,  &
01738                  "Integer controlling level of messaging\n" // &
01739                  "\t0 = Only FATAL messages\n" // &
01740                  "\t2 = Only FATAL, WARNING, NOTE [default]\n" // &
01741                  "\t9 = All)", default=2, debuggingparam=.true.)
01742   call get_param(param_file, "MOM", "DO_UNIT_TESTS", do_unit_tests, &
01743                  "If True, exercises unit tests at model start up.", &
01744                  default=.false., debuggingparam=.true.)
01745   if (do_unit_tests) then
01746     id_clock_unit_tests = cpu_clock_id('(Ocean unit tests)', grain=clock_module)
01747     call cpu_clock_begin(id_clock_unit_tests)
01748     call unit_tests(verbosity)
01749     call cpu_clock_end(id_clock_unit_tests)
01750   endif
01751
01752   call get_param(param_file, "MOM", "SPLIT", cs%split, &
01753                  "Use the split time stepping if true.", default=.true.)
01754   if (cs%split) then
01755     cs%use_RK2 = .false.
01756   else
01757     call get_param(param_file, "MOM", "USE_RK2", cs%use_RK2, &
01758                  "If true, use RK2 instead of RK3 in the unsplit time stepping.", &
01759                  default=.false.)
01760   endif
01761
01762   call get_param(param_file, "MOM", "CALC_RHO_FOR_SEA_LEVEL", cs%calc_rho_for_sea_lev, &
01763                  "If true, the in-situ density is used to calculate the "//&
01764                  "effective sea level that is returned to the coupler. If false, "//&
01765                  "the Boussinesq parameter RHO_0 is used.", default=.false.)
01766   call get_param(param_file, "MOM", "ENABLE_THERMODYNAMICS", use_temperature, &
01767                  "If true, Temperature and salinity are used as state "//&
01768                  "variables.", default=.true.)
01769   call get_param(param_file, "MOM", "USE_EOS", use_eos, &
01770                  "If true,  density is calculated from temperature and "//&
01771                  "salinity with an equation of state.  If USE_EOS is "//&
01772                  "true, ENABLE_THERMODYNAMICS must be true as well.", &
01773                  default=use_temperature)
01774   call get_param(param_file, "MOM", "DIABATIC_FIRST", cs%diabatic_first, &
01775                  "If true, apply diabatic and thermodynamic processes, "//&
01776                  "including buoyancy forcing and mass gain or loss, "//&
01777                  "before stepping the dynamics forward.", default=.false.)
01778   call get_param(param_file, "MOM", "USE_CONTEMP_ABSSAL", use_cont_abss, &
01779                  "If true, the prognostics T&S are the conservative temperature "//&
01780                  "and absolute salinity. Care should be taken to convert them "//&
01781                  "to potential temperature and practical salinity before "//&
01782                  "exchanging them with the coupler and/or reporting T&S diagnostics.", &
```

```
01783                 default=.false.)
01784   cs%tv%T_is_conT = use_cont_abss ; cs%tv%S_is_absS = use_cont_abss
01785   call get_param(param_file, "MOM", "ADIABATIC", cs%adiabatic, &
01786                 "There are no diapycnal mass fluxes if ADIABATIC is "//&
01787                 "true. This assumes that KD = KDML = 0.0 and that "//&
01788                 "there is no buoyancy forcing, but makes the model "//&
01789                 "faster by eliminating subroutine calls.", default=.false.)
01790   call get_param(param_file, "MOM", "DO_DYNAMICS", cs%do_dynamics, &
01791                 "If False, skips the dynamics calls that update u & v, as well as "//&
01792                 "the gravity wave adjustment to h. This may be a fragile feature, "//&
01793                 "but can be useful during development", default=.true.)
01794   call get_param(param_file, "MOM", "ADVECT_TS", advect_ts, &
01795                 "If True, advect temperature and salinity horizontally "//&
01796                 "If False, T/S are registered for advection. "//&
01797                 "This is intended only to be used in offline tracer mode "//&
01798                 "and is by default false in that case.", &
01799                 do_not_log = .true., default=.true. )
01800   if (present(offline_tracer_mode)) then ! Only read this parameter in enabled modes
01801     call get_param(param_file, "MOM", "OFFLINE_TRACER_MODE", cs%offline_tracer_mode, &
01802                 "If true, barotropic and baroclinic dynamics, thermodynamics "//&
01803                 "are all bypassed with all the fields necessary to integrate "//&
01804                 "the tracer advection and diffusion equation are read in from "//&
01805                 "files stored from a previous integration of the prognostic model. "//&
01806                 "NOTE: This option only used in the ocean_solo_driver.", default=.false.)
01807     if (cs%offline_tracer_mode) then
01808       call get_param(param_file, "MOM", "ADVECT_TS", advect_ts, &
01809                   "If True, advect temperature and salinity horizontally "//&
01810                   "If False, T/S are registered for advection. "//&
01811                   "This is intended only to be used in offline tracer mode."//&
01812                   "and is by default false in that case", &
01813                   default=.false. )
01814     endif
01815   endif
01816   call get_param(param_file, "MOM", "USE_REGRIDDING", cs%use_ALE_algorithm, &
01817                 "If True, use the ALE algorithm (regridding/remapping). "//&
01818                 "If False, use the layered isopycnal algorithm.", default=.false. )
01819   call get_param(param_file, "MOM", "BULKMIXEDLAYER", bulkmixedlayer, &
01820                 "If true, use a Kraus-Turner-like bulk mixed layer "//&
01821                 "with transitional buffer layers.  Layers 1 through "//&
01822                 "NKML+NKBL have variable densities. There must be at "//&
01823                 "least NKML+NKBL+1 layers if BULKMIXEDLAYER is true. "//&
01824                 "BULKMIXEDLAYER can not be used with USE_REGRIDDING. "//&
01825                 "The default is influenced by ENABLE_THERMODYNAMICS.", &
01826                 default=use_temperature .and. .not.cs%use_ALE_algorithm)
01827   call get_param(param_file, "MOM", "THICKNESSDIFFUSE", cs%thickness_diffuse, &
01828                 "If true, interface heights are diffused with a "//&
01829                 "coefficient of KHTH.", default=.false.)
01830   call get_param(param_file, "MOM",  "THICKNESSDIFFUSE_FIRST", &
01831                                      cs%thickness_diffuse_first, &
01832                 "If true, do thickness diffusion before dynamics. "//&
01833                 "This is only used if THICKNESSDIFFUSE is true.", &
01834                 default=.false.)
01835   if (.not.cs%thickness_diffuse) cs%thickness_diffuse_first = .false.
01836   call get_param(param_file, "MOM", "BATHYMETRY_AT_VEL", bathy_at_vel, &
01837                 "If true, there are separate values for the basin depths "//&
01838                 "at velocity points.  Otherwise the effects of topography "//&
01839                 "are entirely determined from thickness points.", &
01840                 default=.false.)
01841   call get_param(param_file, "MOM", "USE_WAVES", cs%UseWaves, default=.false., &
01842                 do_not_log=.true.)
01843
01844   call get_param(param_file, "MOM", "DEBUG", cs%debug, &
01845                 "If true, write out verbose debugging data.", &
01846                 default=.false., debuggingparam=.true.)
01847   call get_param(param_file, "MOM", "DEBUG_TRUNCATIONS", debug_truncations, &
01848                 "If true, calculate all diagnostics that are useful for "//&
01849                 "debugging truncations.", default=.false., debuggingparam=.true.)
01850
01851   call get_param(param_file, "MOM", "DT", cs%dt, &
01852                 "The (baroclinic) dynamics time step.  The time-step that "//&
01853                 "is actually used will be an integer fraction of the "//&
01854                 "forcing time-step (DT_FORCING in ocean-only mode or the "//&
01855                 "coupling timestep in coupled mode.)", units="s", scale=us%s_to_T, &
01856                 fail_if_missing=.true.)
01857   call get_param(param_file, "MOM", "DT_THERM", cs%dt_therm, &
01858                 "The thermodynamic and tracer advection time step. "//&
01859                 "Ideally DT_THERM should be an integer multiple of DT "//&
01860                 "and less than the forcing or coupling time-step, unless "//&
01861                 "THERMO_SPANS_COUPLING is true, in which case DT_THERM "//&
01862                 "can be an integer multiple of the coupling timestep.  By "//&
01863                 "default DT_THERM is set to DT.", &
01864                 units="s", scale=us%s_to_T, default=us%T_to_s*cs%dt)
01865   call get_param(param_file, "MOM", "THERMO_SPANS_COUPLING", cs%thermo_spans_coupling, &
01866                 "If true, the MOM will take thermodynamic and tracer "//&
01867                 "timesteps that can be longer than the coupling timestep. "//&
01868                 "The actual thermodynamic timestep that is used in this "//&
01869                 "case is the largest integer multiple of the coupling "//&
```

```
01870                 "timestep that is less than or equal to DT_THERM.", default=.false.)
01871
01872   if (bulkmixedlayer) then
01873     cs%Hmix = -1.0 ; cs%Hmix_UV = -1.0
01874   else
01875     call get_param(param_file, "MOM", "HMIX_SFC_PROP", cs%Hmix, &
01876                 "If BULKMIXEDLAYER is false, HMIX_SFC_PROP is the depth "//&
01877                 "over which to average to find surface properties like "//&
01878                 "SST and SSS or density (but not surface velocities).", &
01879                 units="m", default=1.0, scale=us%m_to_Z)
01880     call get_param(param_file, "MOM", "HMIX_UV_SFC_PROP", cs%Hmix_UV, &
01881                 "If BULKMIXEDLAYER is false, HMIX_UV_SFC_PROP is the depth "//&
01882                 "over which to average to find surface flow properties, "//&
01883                 "SSU, SSV. A non-positive value indicates no averaging.", &
01884                 units="m", default=0.0, scale=us%m_to_Z)
01885   endif
01886   call get_param(param_file, "MOM", "HFREEZE", cs%HFrz, &
01887                 "If HFREEZE > 0, melt potential will be computed. The actual depth "//&
01888                 "over which melt potential is computed will be min(HFREEZE, OBLD), "//&
01889                 "where OBLD is the boundary layer depth. If HFREEZE <= 0 (default), "//&
01890                 "melt potential will not be computed.", units="m", default=-1.0, scale=us%m_to_Z)
01891   call get_param(param_file, "MOM", "INTERPOLATE_P_SURF", cs%interp_p_surf, &
01892                 "If true, linearly interpolate the surface pressure "//&
01893                 "over the coupling time step, using the specified value "//&
01894                 "at the end of the step.", default=.false.)
01895
01896   if (cs%split) then
01897     call get_param(param_file, "MOM", "DTBT", dtbt, default=-0.98)
01898     default_val = us%T_to_s*cs%dt_therm ; if (dtbt > 0.0) default_val = -1.0
01899     cs%dtbt_reset_period = -1.0
01900     call get_param(param_file, "MOM", "DTBT_RESET_PERIOD", cs%dtbt_reset_period, &
01901                 "The period between recalculations of DTBT (if DTBT <= 0). "//&
01902                 "If DTBT_RESET_PERIOD is negative, DTBT is set based "//&
01903                 "only on information available at initialization.  If 0, "//&
01904                 "DTBT will be set every dynamics time step. The default "//&
01905                 "is set by DT_THERM.  This is only used if SPLIT is true.", &
01906                 units="s", default=default_val, do_not_read=(dtbt > 0.0))
01907   endif
01908
01909   ! This is here in case these values are used inappropriately.
01910   use_frazil = .false. ; bound_salinity = .false.
01911   cs%tv%P_Ref = 2.0e7*us%kg_m3_to_R*us%m_s_to_L_T**2
01912   if (use_temperature) then
01913     call get_param(param_file, "MOM", "FRAZIL", use_frazil, &
01914                 "If true, water freezes if it gets too cold, and the "//&
01915                 "accumulated heat deficit is returned in the "//&
01916                 "surface state.  FRAZIL is only used if "//&
01917                 "ENABLE_THERMODYNAMICS is true.", default=.false.)
01918     call get_param(param_file, "MOM", "DO_GEOTHERMAL", use_geothermal, &
01919                 "If true, apply geothermal heating.", default=.false.)
01920     call get_param(param_file, "MOM", "BOUND_SALINITY", bound_salinity, &
01921                 "If true, limit salinity to being positive. (The sea-ice "//&
01922                 "model may ask for more salt than is available and "//&
01923                 "drive the salinity negative otherwise.)", default=.false.)
01924     call get_param(param_file, "MOM", "MIN_SALINITY", cs%tv%min_salinity, &
01925                 "The minimum value of salinity when BOUND_SALINITY=True.", &
01926                 units="PPT", default=0.0, do_not_log=.not.bound_salinity)
01927     call get_param(param_file, "MOM", "C_P", cs%tv%C_p, &
01928                 "The heat capacity of sea water, approximated as a "//&
01929                 "constant. This is only used if ENABLE_THERMODYNAMICS is "//&
01930                 "true. The default value is from the TEOS-10 definition "//&
01931                 "of conservative temperature.", units="J kg-1 K-1", &
01932                 default=3991.86795711963, scale=us%J_kg_to_Q)
01933     call get_param(param_file, "MOM", "USE_PSURF_IN_EOS", cs%use_p_surf_in_EOS, &
01934                 "If true, always include the surface pressure contributions "//&
01935                 "in equation of state calculations.", default=.true.)
01936   endif
01937   if (use_eos) call get_param(param_file, "MOM", "P_REF", cs%tv%P_Ref, &
01938                 "The pressure that is used for calculating the coordinate "//&
01939                 "density.  (1 Pa = 1e4 dbar, so 2e7 is commonly used.) "//&
01940                 "This is only used if USE_EOS and ENABLE_THERMODYNAMICS are true.", &
01941                 units="Pa", default=2.0e7, scale=us%kg_m3_to_R*us%m_s_to_L_T**2)
01942
01943   if (bulkmixedlayer) then
01944     call get_param(param_file, "MOM", "NKML", nkml, &
01945                 "The number of sublayers within the mixed layer if "//&
01946                 "BULKMIXEDLAYER is true.", units="nondim", default=2)
01947     call get_param(param_file, "MOM", "NKBL", nkbl, &
01948                 "The number of layers that are used as variable density "//&
01949                 "buffer layers if BULKMIXEDLAYER is true.", units="nondim", &
01950                 default=2)
01951   endif
01952
01953   call get_param(param_file, "MOM", "GLOBAL_INDEXING", global_indexing, &
01954                 "If true, use a global lateral indexing convention, so "//&
01955                 "that corresponding points on different processors have "//&
01956                 "the same index. This does not work with static memory.", &
```

```
01957                 default=.false., layoutparam=.true.)
01958 #ifdef STATIC_MEMORY_
01959   if (global_indexing) call mom_error(fatal, "initialize_MOM: "//&
01960         "GLOBAL_INDEXING can not be true with STATIC_MEMORY.")
01961 #endif
01962   call get_param(param_file, "MOM", "FIRST_DIRECTION", first_direction, &
01963                  "An integer that indicates which direction goes first "//&
01964                  "in parts of the code that use directionally split "//&
01965                  "updates, with even numbers (or 0) used for x- first "//&
01966                  "and odd numbers used for y-first.", default=0)
01967
01968   call get_param(param_file, "MOM", "CHECK_BAD_SURFACE_VALS", cs%check_bad_sfc_vals, &
01969                  "If true, check the surface state for ridiculous values.", &
01970                  default=.false.)
01971   if (cs%check_bad_sfc_vals) then
01972     call get_param(param_file, "MOM", "BAD_VAL_SSH_MAX", cs%bad_val_ssh_max, &
01973                  "The value of SSH above which a bad value message is "//&
01974                  "triggered, if CHECK_BAD_SURFACE_VALS is true.", &
01975                  units="m", default=20.0, scale=us%m_to_Z)
01976     call get_param(param_file, "MOM", "BAD_VAL_SSS_MAX", cs%bad_val_sss_max, &
01977                  "The value of SSS above which a bad value message is "//&
01978                  "triggered, if CHECK_BAD_SURFACE_VALS is true.", units="PPT", &
01979                  default=45.0)
01980     call get_param(param_file, "MOM", "BAD_VAL_SST_MAX", cs%bad_val_sst_max, &
01981                  "The value of SST above which a bad value message is "//&
01982                  "triggered, if CHECK_BAD_SURFACE_VALS is true.", &
01983                  units="deg C", default=45.0)
01984     call get_param(param_file, "MOM", "BAD_VAL_SST_MIN", cs%bad_val_sst_min, &
01985                  "The value of SST below which a bad value message is "//&
01986                  "triggered, if CHECK_BAD_SURFACE_VALS is true.", &
01987                  units="deg C", default=-2.1)
01988     call get_param(param_file, "MOM", "BAD_VAL_COLUMN_THICKNESS", cs%bad_val_col_thick, &
01989                  "The value of column thickness below which a bad value message is "//&
01990                  "triggered, if CHECK_BAD_SURFACE_VALS is true.", &
01991                  units="m", default=0.0, scale=us%m_to_Z)
01992   endif
01993   call get_param(param_file, "MOM", "DEFAULT_2018_ANSWERS", default_2018_answers, &
01994                  "This sets the default value for the various _2018_ANSWERS parameters.", &
01995                  default=.false.)
01996   call get_param(param_file, "MOM", "SURFACE_2018_ANSWERS", cs%answers_2018, &
01997                  "If true, use expressions for the surface properties that recover the answers "//&
01998                  "from the end of 2018. Otherwise, use more appropriate expressions that differ "//&
01999                  "at roundoff for non-Boussinesq cases.", default=default_2018_answers)
02000
02001   call get_param(param_file, "MOM", "SAVE_INITIAL_CONDS", save_ic, &
02002                  "If true, write the initial conditions to a file given "//&
02003                  "by IC_OUTPUT_FILE.", default=.false.)
02004   call get_param(param_file, "MOM", "IC_OUTPUT_FILE", cs%IC_file, &
02005                  "The file into which to write the initial conditions.", &
02006                  default="MOM_IC")
02007   call get_param(param_file, "MOM", "WRITE_GEOM", write_geom, &
02008                  "If =0, never write the geometry and vertical grid files. "//&
02009                  "If =1, write the geometry and vertical grid files only for "//&
02010                  "a new simulation. If =2, always write the geometry and "//&
02011                  "vertical grid files. Other values are invalid.", default=1)
02012   if (write_geom<0 .or. write_geom>2) call mom_error(fatal,"MOM: "//&
02013           "WRITE_GEOM must be equal to 0, 1 or 2.")
02014   write_geom_files = ((write_geom==2) .or. ((write_geom==1) .and. &
02015      ((dirs%input_filename(1:1)=='n') .and. (len_trim(dirs%input_filename)==1))))
02016 ! If the restart file type had been initialized, this could become:
02017 !  write_geom_files = ((write_geom==2) .or. &
02018 !                     ((write_geom==1) .and. is_new_run(restart_CSp)))
02019
02020   ! Check for inconsistent parameter settings.
02021   if (cs%use_ALE_algorithm .and. bulkmixedlayer) call mom_error(fatal, &
02022     "MOM: BULKMIXEDLAYER can not currently be used with the ALE algorithm.")
02023   if (cs%use_ALE_algorithm .and. .not.use_temperature) call mom_error(fatal, &
02024      "MOM: At this time, USE_EOS should be True when using the ALE algorithm.")
02025   if (cs%adiabatic .and. use_temperature) call mom_error(warning, &
02026     "MOM: ADIABATIC and ENABLE_THERMODYNAMICS both defined is usually unwise.")
02027   if (use_eos .and. .not.use_temperature) call mom_error(fatal, &
02028     "MOM: ENABLE_THERMODYNAMICS must be defined to use USE_EOS.")
02029   if (cs%adiabatic .and. bulkmixedlayer) call mom_error(fatal, &
02030     "MOM: ADIABATIC and BULKMIXEDLAYER can not both be defined.")
02031   if (bulkmixedlayer .and. .not.use_eos) call mom_error(fatal, &
02032       "initialize_MOM: A bulk mixed layer can only be used with T & S as "//&
02033       "state variables. Add USE_EOS = True to MOM_input.")
02034
02035   call get_param(param_file, 'MOM', "ICE_SHELF", use_ice_shelf, default=.false., do_not_log=.true.)
02036   if (use_ice_shelf) then
02037      inputdir = "." ;  call get_param(param_file, 'MOM', "INPUTDIR", inputdir)
02038      inputdir = slasher(inputdir)
02039      call get_param(param_file, 'MOM', "ICE_THICKNESS_FILE", ice_shelf_file, &
02040                     "The file from which the ice bathymetry and area are read.", &
02041                     fail_if_missing=.true.)
02042      call get_param(param_file, 'MOM', "ICE_AREA_VARNAME", area_varname, &
02043                     "The name of the area variable in ICE_THICKNESS_FILE.", &
```

```
02044                     fail_if_missing=.true.)
02045   endif
02046
02047
02048   cs%ensemble_ocean=.false.
02049   call get_param(param_file, "MOM", "ENSEMBLE_OCEAN", cs%ensemble_ocean, &
02050                  "If False, The model is being run in serial mode as a single realization. "//&
02051                  "If True, The current model realization is part of a larger ensemble "//&
02052                  "and at the end of step MOM, we will perform a gather of the ensemble "//&
02053                  "members for statistical evaluation and/or data assimilation.", default=.false.)
02054
02055   call calltree_waypoint("MOM parameters read (initialize_MOM)")
02056
02057   ! Grid rotation test
02058   call get_param(param_file, "MOM", "ROTATE_INDEX", cs%rotate_index, &
02059       "Enable rotation of the horizontal indices.", default=.false., &
02060       debuggingparam=.true.)
02061   if (cs%rotate_index) then
02062     ! TODO: Index rotation currently only works when index rotation does not
02063     !   change the MPI rank of each domain.  Resolving this will require a
02064     !   modification to FMS PE assignment.
02065     !   For now, we only permit single-core runs.
02066
02067     if (num_pes() /= 1) &
02068       call mom_error(fatal, "Index rotation is only supported on one PE.")
02069
02070     call get_param(param_file, "MOM", "INDEX_TURNS", turns, &
02071         "Number of counterclockwise quarter-turn index rotations.", &
02072         default=1, debuggingparam=.true.)
02073   endif
02074
02075   ! Set up the model domain and grids.
02076 #ifdef SYMMETRIC_MEMORY_
02077   symmetric = .true.
02078 #else
02079   symmetric = .false.
02080 #endif
02081   g_in => cs%G_in
02082 #ifdef STATIC_MEMORY_
02083   call mom_domains_init(g_in%domain, param_file, symmetric=symmetric, &
02084             static_memory=.true., nihalo=nihalo_, njhalo=njhalo_, &
02085             niglobal=niglobal_, njglobal=njglobal_, niproc=niproc_, &
02086             njproc=njproc_)
02087 #else
02088   call mom_domains_init(g_in%domain, param_file, symmetric=symmetric, &
02089                         domain_name="MOM_in")
02090 #endif
02091
02092   ! Copy input grid (G_in) domain to active grid G
02093   ! Swap axes for quarter and 3-quarter turns
02094   if (cs%rotate_index) then
02095     allocate(cs%G)
02096     call clone_mom_domain(g_in%Domain, cs%G%Domain, turns=turns)
02097     first_direction = modulo(first_direction + turns, 2)
02098   else
02099     cs%G => g_in
02100   endif
02101
02102   ! TODO: It is unlikey that test_grid_copy and rotate_index would work at the
02103   !   same time.  It may be possible to enable both but for now we prevent it.
02104   if (test_grid_copy .and. cs%rotate_index) &
02105     call mom_error(fatal, "Grid cannot be copied during index rotation.")
02106
02107   if (test_grid_copy) then ; allocate(g)
02108   else ; g => cs%G ; endif
02109
02110   call calltree_waypoint("domains initialized (initialize_MOM)")
02111
02112   call mom_debugging_init(param_file)
02113   call diag_mediator_infrastructure_init()
02114   call mom_io_init(param_file)
02115
02116   ! Create HI and dG on the input index map.
02117   call hor_index_init(g_in%Domain, hi_in, param_file, &
02118                       local_indexing=.not.global_indexing)
02119   call create_dyn_horgrid(dg_in, hi_in, bathymetry_at_vel=bathy_at_vel)
02120   call clone_mom_domain(g_in%Domain, dg_in%Domain)
02121
02122   ! Allocate initialize time-invariant MOM variables.
02123   call mom_initialize_fixed(dg_in, us, obc_in, param_file, write_geom_files, &
02124                             dirs%output_directory)
02125
02126   call calltree_waypoint("returned from MOM_initialize_fixed() (initialize_MOM)")
02127
02128   ! Determine HI and dG for the model index map.
02129   if (cs%rotate_index) then
02130     allocate(hi)
```

```
02131     call rotate_hor_index(hi_in, turns, hi)
02132     call create_dyn_horgrid(dg, hi, bathymetry_at_vel=bathy_at_vel)
02133     call clone_mom_domain(g%Domain, dg%Domain)
02134     call rotate_dyngrid(dg_in, dg, us, turns)
02135     if (associated(obc_in)) then
02136       ! TODO: General OBC index rotations is not yet supported.
02137       if (modulo(turns, 4) /= 1) &
02138         call mom_error(fatal, "OBC index rotation of 180 and 270 degrees is " &
02139           // "not yet unsupported.")
02140       allocate(cs%OBC)
02141       call rotate_obc_config(obc_in, dg_in, cs%OBC, dg, turns)
02142     endif
02143   else
02144     hi => hi_in
02145     dg => dg_in
02146     cs%OBC => obc_in
02147   endif
02148
02149   call verticalgridinit( param_file, cs%GV, us )
02150   gv => cs%GV
02151
02152   ! Allocate the auxiliary non-symmetric domain for debugging or I/O purposes.
02153   if (cs%debug .or. dg%symmetric) &
02154     call clone_mom_domain(dg%Domain, dg%Domain_aux, symmetric=.false.)
02155
02156   call calltree_waypoint("grids initialized (initialize_MOM)")
02157
02158   call mom_timing_init(cs)
02159
02160   if (associated(cs%OBC)) call call_obc_register(param_file, cs%update_OBC_CSp, cs%OBC)
02161
02162   call tracer_registry_init(param_file, cs%tracer_Reg)
02163
02164   ! Allocate and initialize space for the primary time-varying MOM variables.
02165   is   = dg%isc   ; ie   = dg%iec  ; js   = dg%jsc  ; je   = dg%jec ; nz = gv%ke
02166   isd  = dg%isd   ; ied  = dg%ied  ; jsd  = dg%jsd  ; jed  = dg%jed
02167   isdb = dg%IsdB  ; iedb = dg%IedB ; jsdb = dg%JsdB ; jedb = dg%JedB
02168   alloc_(cs%u(isdb:iedb,jsd:jed,nz))   ; cs%u(:,:,:) = 0.0
02169   alloc_(cs%v(isd:ied,jsdb:jedb,nz))   ; cs%v(:,:,:) = 0.0
02170   alloc_(cs%h(isd:ied,jsd:jed,nz))     ; cs%h(:,:,:) = gv%Angstrom_H
02171   alloc_(cs%uh(isdb:iedb,jsd:jed,nz))  ; cs%uh(:,:,:) = 0.0
02172   alloc_(cs%vh(isd:ied,jsdb:jedb,nz))  ; cs%vh(:,:,:) = 0.0
02173   if (use_temperature) then
02174     alloc_(cs%T(isd:ied,jsd:jed,nz))   ; cs%T(:,:,:) = 0.0
02175     alloc_(cs%S(isd:ied,jsd:jed,nz))   ; cs%S(:,:,:) = 0.0
02176     cs%tv%T => cs%T ; cs%tv%S => cs%S
02177     if (cs%tv%T_is_conT) then
02178       vd_t = var_desc(name="contemp", units="Celsius", longname="Conservative Temperature", &
02179                       cmor_field_name="thetao", cmor_longname="Sea Water Potential Temperature", &
02180                       conversion=us%Q_to_J_kg*cs%tv%C_p)
02181     else
02182       vd_t = var_desc(name="temp", units="degC", longname="Potential Temperature", &
02183                       cmor_field_name="thetao", cmor_longname="Sea Water Potential Temperature", &
02184                       conversion=us%Q_to_J_kg*cs%tv%C_p)
02185     endif
02186     if (cs%tv%S_is_absS) then
02187       vd_s = var_desc(name="abssalt",units="g kg-1",longname="Absolute Salinity", &
02188                       cmor_field_name="so", cmor_longname="Sea Water Salinity", &
02189                       conversion=0.001)
02190     else
02191       vd_s = var_desc(name="salt",units="psu",longname="Salinity", &
02192                       cmor_field_name="so", cmor_longname="Sea Water Salinity", &
02193                       conversion=0.001)
02194     endif
02195
02196     if (advect_ts) then
02197       s_flux_units = get_tr_flux_units(gv, "psu") ! Could change to "kg m-2 s-1"?
02198       conv2watt    = gv%H_to_kg_m2 * us%Q_to_J_kg*cs%tv%C_p
02199       if (gv%Boussinesq) then
02200         conv2salt = gv%H_to_m ! Could change to GV%H_to_kg_m2 * 0.001?
02201       else
02202         conv2salt = gv%H_to_kg_m2
02203       endif
02204       call register_tracer(cs%tv%T, cs%tracer_Reg, param_file, dg%HI, gv, &
02205                            tr_desc=vd_t, registry_diags=.true., flux_nameroot='T', &
02206                            flux_units='W', flux_longname='Heat', &
02207                            flux_scale=conv2watt, convergence_units='W m-2', &
02208                            convergence_scale=conv2watt, cmor_tendprefix="opottemp", diag_form=2)
02209       call register_tracer(cs%tv%S, cs%tracer_Reg, param_file, dg%HI, gv, &
02210                            tr_desc=vd_s, registry_diags=.true., flux_nameroot='S', &
02211                            flux_units=s_flux_units, flux_longname='Salt', &
02212                            flux_scale=conv2salt, convergence_units='kg m-2 s-1', &
02213                            convergence_scale=0.001*gv%H_to_kg_m2, cmor_tendprefix="osalt", diag_form=2)
02214     endif
02215     ! NOTE: register_temp_salt_segments includes allocation of tracer fields
02216     !   along segments.  Bit reproducibility requires that MOM_initialize_state
02217     !   be called on the input index map, so we must setup both OBC and OBC_in.
```

```
02218     !
02219     ! XXX: This call on OBC_in allocates the tracer fields on the unrotated
02220     !   grid, but also incorrectly stores a pointer to a tracer_type for the
02221     !   rotated registry (e.g. segment%tr_reg%Tr(n)%Tr) from CS%tracer_reg.
02222     !
02223     !   While incorrect and potentially dangerous, it does not seem that this
02224     !   pointer is used during initialization, so we leave it for now.
02225     if (cs%rotate_index .and. associated(obc_in)) &
02226       call register_temp_salt_segments(gv, obc_in, cs%tracer_Reg, param_file)
02227     if (associated(cs%OBC)) &
02228       call register_temp_salt_segments(gv, cs%OBC, cs%tracer_Reg, param_file)
02229   endif
02230   if (use_frazil) then
02231     allocate(cs%tv%frazil(isd:ied,jsd:jed)) ; cs%tv%frazil(:,:) = 0.0
02232   endif
02233   if (bound_salinity) then
02234     allocate(cs%tv%salt_deficit(isd:ied,jsd:jed)) ; cs%tv%salt_deficit(:,:) = 0.0
02235   endif
02236
02237   if (bulkmixedlayer .or. use_temperature) then
02238     allocate(cs%Hml(isd:ied,jsd:jed)) ; cs%Hml(:,:) = 0.0
02239   endif
02240
02241   if (bulkmixedlayer) then
02242     gv%nkml = nkml ; gv%nk_rho_varies = nkml + nkbl
02243   else
02244     gv%nkml = 0 ; gv%nk_rho_varies = 0
02245   endif
02246   if (cs%use_ALE_algorithm) then
02247     call get_param(param_file, "MOM", "NK_RHO_VARIES", gv%nk_rho_varies, default=0) ! Will default to nz
      later... -AJA
02248   endif
02249
02250   alloc_(cs%uhtr(isdb:iedb,jsd:jed,nz)) ; cs%uhtr(:,:,:) = 0.0
02251   alloc_(cs%vhtr(isd:ied,jsdb:jedb,nz)) ; cs%vhtr(:,:,:) = 0.0
02252   cs%t_dyn_rel_adv = 0.0 ; cs%t_dyn_rel_thermo = 0.0 ; cs%t_dyn_rel_diag = 0.0
02253
02254   if (debug_truncations) then
02255     allocate(cs%u_prev(isdb:iedb,jsd:jed,nz)) ; cs%u_prev(:,:,:) = 0.0
02256     allocate(cs%v_prev(isd:ied,jsdb:jedb,nz)) ; cs%v_prev(:,:,:) = 0.0
02257     mom_internal_state%u_prev => cs%u_prev
02258     mom_internal_state%v_prev => cs%v_prev
02259     call safe_alloc_ptr(cs%ADp%du_dt_visc,isdb,iedb,jsd,jed,nz)
02260     call safe_alloc_ptr(cs%ADp%dv_dt_visc,isd,ied,jsdb,jedb,nz)
02261     if (.not.cs%adiabatic) then
02262       call safe_alloc_ptr(cs%ADp%du_dt_dia,isdb,iedb,jsd,jed,nz)
02263       call safe_alloc_ptr(cs%ADp%dv_dt_dia,isd,ied,jsdb,jedb,nz)
02264     endif
02265   endif
02266
02267   mom_internal_state%u => cs%u ; mom_internal_state%v => cs%v
02268   mom_internal_state%h => cs%h
02269   mom_internal_state%uh => cs%uh ; mom_internal_state%vh => cs%vh
02270   if (use_temperature) then
02271     mom_internal_state%T => cs%T ; mom_internal_state%S => cs%S
02272   endif
02273
02274   cs%CDp%uh => cs%uh ; cs%CDp%vh => cs%vh
02275
02276   if (cs%interp_p_surf) then
02277     allocate(cs%p_surf_prev(isd:ied,jsd:jed)) ; cs%p_surf_prev(:,:) = 0.0
02278   endif
02279
02280   alloc_(cs%ssh_rint(isd:ied,jsd:jed)) ; cs%ssh_rint(:,:) = 0.0
02281   alloc_(cs%ave_ssh_ibc(isd:ied,jsd:jed)) ; cs%ave_ssh_ibc(:,:) = 0.0
02282   alloc_(cs%eta_av_bc(isd:ied,jsd:jed)) ; cs%eta_av_bc(:,:) = 0.0
02283   cs%time_in_cycle = 0.0 ; cs%time_in_thermo_cycle = 0.0
02284
02285   ! Use the Wright equation of state by default, unless otherwise specified
02286   ! Note: this line and the following block ought to be in a separate
02287   ! initialization routine for tv.
02288   if (use_eos) call eos_init(param_file, cs%tv%eqn_of_state, us)
02289   if (use_temperature) then
02290     allocate(cs%tv%TempxPmE(isd:ied,jsd:jed)) ; cs%tv%TempxPmE(:,:) = 0.0
02291     if (use_geothermal) then
02292       allocate(cs%tv%internal_heat(isd:ied,jsd:jed)) ; cs%tv%internal_heat(:,:) = 0.0
02293     endif
02294   endif
02295   call calltree_waypoint("state variables allocated (initialize_MOM)")
02296
02297   ! Set the fields that are needed for bitwise identical restarting
02298   ! the time stepping scheme.
02299   call restart_init(param_file, restart_csp)
02300   call set_restart_fields(gv, us, param_file, cs, restart_csp)
02301   if (cs%split) then
02302     call register_restarts_dyn_split_rk2(dg%HI, gv, param_file, &
02303              cs%dyn_split_RK2_CSp, restart_csp, cs%uh, cs%vh)
```

```
02304   elseif (cs%use_RK2) then
02305     call register_restarts_dyn_unsplit_rk2(dg%HI, gv, param_file, &
02306             cs%dyn_unsplit_RK2_CSp, restart_csp)
02307   else
02308     call register_restarts_dyn_unsplit(dg%HI, gv, param_file, &
02309             cs%dyn_unsplit_CSp, restart_csp)
02310   endif
02311
02312   ! This subroutine calls user-specified tracer registration routines.
02313   ! Additional calls can be added to MOM_tracer_flow_control.F90.
02314   call call_tracer_register(dg%HI, gv, us, param_file, cs%tracer_flow_CSp, &
02315                             cs%tracer_Reg, restart_csp)
02316
02317   call meke_alloc_register_restart(dg%HI, param_file, cs%MEKE, restart_csp)
02318   call set_visc_register_restarts(dg%HI, gv, param_file, cs%visc, restart_csp)
02319   call mixedlayer_restrat_register_restarts(dg%HI, param_file, &
02320            cs%mixedlayer_restrat_CSp, restart_csp)
02321
02322   if (associated(cs%OBC)) &
02323     call open_boundary_register_restarts(dg%HI, gv, cs%OBC, cs%tracer_Reg, &
02324                           param_file, restart_csp, use_temperature)
02325
02326   call calltree_waypoint("restart registration complete (initialize_MOM)")
02327
02328   ! Initialize dynamically evolving fields, perhaps from restart files.
02329   call cpu_clock_begin(id_clock_mom_init)
02330   call mom_initialize_coord(gv, us, param_file, write_geom_files, &
02331                             dirs%output_directory, cs%tv, dg%max_depth)
02332   call calltree_waypoint("returned from MOM_initialize_coord() (initialize_MOM)")
02333
02334   if (cs%use_ALE_algorithm) then
02335     call ale_init(param_file, gv, us, dg%max_depth, cs%ALE_CSp)
02336     call calltree_waypoint("returned from ALE_init() (initialize_MOM)")
02337   endif
02338
02339   !   Shift from using the temporary dynamic grid type to using the final
02340   ! (potentially static) ocean-specific grid type.
02341   !   The next line would be needed if G%Domain had not already been init'd above:
02342   !     call clone_MOM_domain(dG%Domain, G%Domain)
02343
02344   ! NOTE: If indices are rotated, then G and G_in must both be initialized.
02345   !   If not rotated, then G_in and G are the same grid.
02346   if (cs%rotate_index) then
02347     call mom_grid_init(g, param_file, us, hi, bathymetry_at_vel=bathy_at_vel)
02348     call copy_dyngrid_to_mom_grid(dg, g, us)
02349     call destroy_dyn_horgrid(dg)
02350   endif
02351   call mom_grid_init(g_in, param_file, us, hi_in, bathymetry_at_vel=bathy_at_vel)
02352   call copy_dyngrid_to_mom_grid(dg_in, g_in, us)
02353   call destroy_dyn_horgrid(dg_in)
02354
02355   ! Set a few remaining fields that are specific to the ocean grid type.
02356   call set_first_direction(g, first_direction)
02357   ! Allocate the auxiliary non-symmetric domain for debugging or I/O purposes.
02358   if (cs%debug .or. g%symmetric) then
02359     call clone_mom_domain(g%Domain, g%Domain_aux, symmetric=.false.)
02360   else ; g%Domain_aux => g%Domain ; endif
02361   ! Copy common variables from the vertical grid to the horizontal grid.
02362   ! Consider removing this later?
02363   g%ke = gv%ke
02364
02365   if (cs%rotate_index) then
02366     g_in%ke = gv%ke
02367
02368     allocate(u_in(g_in%IsdB:g_in%IedB, g_in%jsd:g_in%jed, nz))
02369     allocate(v_in(g_in%isd:g_in%ied, g_in%JsdB:g_in%JedB, nz))
02370     allocate(h_in(g_in%isd:g_in%ied, g_in%jsd:g_in%jed, nz))
02371     u_in(:,:,:) = 0.0
02372     v_in(:,:,:) = 0.0
02373     h_in(:,:,:) = gv%Angstrom_H
02374
02375     if (use_temperature) then
02376       allocate(t_in(g_in%isd:g_in%ied, g_in%jsd:g_in%jed, nz))
02377       allocate(s_in(g_in%isd:g_in%ied, g_in%jsd:g_in%jed, nz))
02378       t_in(:,:,:) = 0.0
02379       s_in(:,:,:) = 0.0
02380
02381       cs%tv%T => t_in
02382       cs%tv%S => s_in
02383     endif
02384
02385     call mom_initialize_state(u_in, v_in, h_in, cs%tv, time, g_in, gv, us, &
02386         param_file, dirs, restart_csp, cs%ALE_CSp, cs%tracer_Reg, &
02387         sponge_in_csp, ale_sponge_in_csp, obc_in, time_in)
02388
02389     if (use_temperature) then
02390       cs%tv%T => cs%T
```

```
02391       cs%tv%S => cs%S
02392     endif
02393
02394     call rotate_initial_state(u_in, v_in, h_in, t_in, s_in, use_temperature, &
02395         turns, cs%u, cs%v, cs%h, cs%T, cs%S)
02396
02397     if (associated(sponge_in_csp)) then
02398       ! TODO: Implementation and testing of non-ALE spong rotation
02399       call mom_error(fatal, "Index rotation of non-ALE sponge is not yet implemented.")
02400     endif
02401
02402     if (associated(ale_sponge_in_csp)) then
02403       call rotate_ale_sponge(ale_sponge_in_csp, g_in, cs%ALE_sponge_CSp, g, turns, param_file)
02404       call update_ale_sponge_field(cs%ALE_sponge_CSp, t_in, g, gv, cs%T)
02405       call update_ale_sponge_field(cs%ALE_sponge_CSp, s_in, g, gv, cs%S)
02406     endif
02407
02408     if (associated(obc_in)) &
02409       call rotate_obc_init(obc_in, g, gv, us, param_file, cs%tv, restart_csp, cs%OBC)
02410
02411     deallocate(u_in)
02412     deallocate(v_in)
02413     deallocate(h_in)
02414     if (use_temperature) then
02415       deallocate(t_in)
02416       deallocate(s_in)
02417     endif
02418   else
02419     call mom_initialize_state(cs%u, cs%v, cs%h, cs%tv, time, g, gv, us, &
02420         param_file, dirs, restart_csp, cs%ALE_CSp, cs%tracer_Reg, &
02421         cs%sponge_CSp, cs%ALE_sponge_CSp, cs%OBC, time_in)
02422   endif
02423
02424   call cpu_clock_end(id_clock_mom_init)
02425   call calltree_waypoint("returned from MOM_initialize_state() (initialize_MOM)")
02426
02427 ! ! Need this after MOM_initialize_state for DOME OBC stuff.
02428 ! if (associated(CS%OBC)) &
02429 !   call open_boundary_register_restarts(G%HI, GV, CS%OBC, CS%tracer_Reg, &
02430 !                         param_file, restart_CSp, use_temperature)
02431
02432 ! call callTree_waypoint("restart registration complete (initialize_MOM)")
02433
02434   ! From this point, there may be pointers being set, so the final grid type
02435   ! that will persist throughout the run has to be used.
02436
02437   if (test_grid_copy) then
02438     !  Copy the data from the temporary grid to the dyn_hor_grid to CS%G.
02439     call create_dyn_horgrid(dg, g%HI)
02440     call clone_mom_domain(g%Domain, dg%Domain)
02441
02442     call clone_mom_domain(g%Domain, cs%G%Domain)
02443     call mom_grid_init(cs%G, param_file, us)
02444
02445     call copy_mom_grid_to_dyngrid(g, dg, us)
02446     call copy_dyngrid_to_mom_grid(dg, cs%G, us)
02447
02448     call destroy_dyn_horgrid(dg)
02449     call mom_grid_end(g) ; deallocate(g)
02450
02451     g => cs%G
02452     if (cs%debug .or. cs%G%symmetric) then
02453       call clone_mom_domain(cs%G%Domain, cs%G%Domain_aux, symmetric=.false.)
02454     else ; cs%G%Domain_aux => cs%G%Domain ; endif
02455     g%ke = gv%ke
02456   endif
02457
02458   ! At this point, all user-modified initialization code has been called.  The
02459   ! remainder of this subroutine is controlled by the parameters that have
02460   ! have already been set.
02461
02462   if (ale_remap_init_conds(cs%ALE_CSp) .and. .not. query_initialized(cs%h,"h",restart_csp)) then
02463     ! This block is controlled by the ALE parameter REMAP_AFTER_INITIALIZATION.
02464     ! \todo This block exists for legacy reasons and we should phase it out of
02465     ! all examples. !###
02466     if (cs%debug) then
02467       call uvchksum("Pre ALE adjust init cond [uv]", &
02468                     cs%u, cs%v, g%HI, haloshift=1)
02469       call hchksum(cs%h,"Pre ALE adjust init cond h", g%HI, haloshift=1, scale=gv%H_to_m)
02470     endif
02471     call calltree_waypoint("Calling adjustGridForIntegrity() to remap initial conditions (initialize_MOM)")
02472     call adjustgridforintegrity(cs%ALE_CSp, g, gv, cs%h )
02473     call calltree_waypoint("Calling ALE_main() to remap initial conditions (initialize_MOM)")
02474     if (use_ice_shelf) then
02475       filename = trim(inputdir)//trim(ice_shelf_file)
02476       if (.not.file_exists(filename, g%Domain)) call mom_error(fatal, &
02477         "MOM: Unable to open "//trim(filename))
```

```
02478
02479       allocate(area_shelf_h(isd:ied,jsd:jed))
02480       allocate(frac_shelf_h(isd:ied,jsd:jed))
02481       call mom_read_data(filename, trim(area_varname), area_shelf_h, g%Domain, scale=us%m_to_L**2)
02482       ! initialize frac_shelf_h with zeros (open water everywhere)
02483       frac_shelf_h(:,:) = 0.0
02484       ! compute fractional ice shelf coverage of h
02485       do j=jsd,jed ; do i=isd,ied
02486         if (g%areaT(i,j) > 0.0) &
02487           frac_shelf_h(i,j) = area_shelf_h(i,j) / g%areaT(i,j)
02488       enddo ; enddo
02489       ! pass to the pointer
02490       shelf_area => frac_shelf_h
02491       call ale_main(g, gv, us, cs%h, cs%u, cs%v, cs%tv, cs%tracer_Reg, cs%ALE_CSp, &
02492                     cs%OBC, frac_shelf_h=shelf_area)
02493     else
02494       call ale_main( g, gv, us, cs%h, cs%u, cs%v, cs%tv, cs%tracer_Reg, cs%ALE_CSp, cs%OBC)
02495     endif
02496
02497     call cpu_clock_begin(id_clock_pass_init)
02498     call create_group_pass(tmp_pass_uv_t_s_h, cs%u, cs%v, g%Domain)
02499     if (use_temperature) then
02500       call create_group_pass(tmp_pass_uv_t_s_h, cs%tv%T, g%Domain, halo=1)
02501       call create_group_pass(tmp_pass_uv_t_s_h, cs%tv%S, g%Domain, halo=1)
02502     endif
02503     call create_group_pass(tmp_pass_uv_t_s_h, cs%h, g%Domain, halo=1)
02504     call do_group_pass(tmp_pass_uv_t_s_h, g%Domain)
02505     call cpu_clock_end(id_clock_pass_init)
02506
02507     if (cs%debug) then
02508       call uvchksum("Post ALE adjust init cond [uv]", cs%u, cs%v, g%HI, haloshift=1)
02509       call hchksum(cs%h, "Post ALE adjust init cond h", g%HI, haloshift=1, scale=gv%H_to_m)
02510     endif
02511   endif
02512   if ( cs%use_ALE_algorithm ) call ale_updateverticalgridtype( cs%ALE_CSp, gv )
02513
02514   diag => cs%diag
02515   ! Initialize the diag mediator.
02516   call diag_mediator_init(g, gv, us, gv%ke, param_file, diag, doc_file_dir=dirs%output_directory)
02517   if (present(diag_ptr)) diag_ptr => cs%diag
02518
02519   ! Initialize the diagnostics masks for native arrays.
02520   ! This step has to be done after call to MOM_initialize_state
02521   ! and before MOM_diagnostics_init
02522   call diag_masks_set(g, gv%ke, diag)
02523
02524   ! Set up pointers within diag mediator control structure,
02525   ! this needs to occur _after_ CS%h etc. have been allocated.
02526   call diag_set_state_ptrs(cs%h, cs%T, cs%S, cs%tv%eqn_of_state, diag)
02527
02528   ! This call sets up the diagnostic axes. These are needed,
02529   ! e.g. to generate the target grids below.
02530   call set_axes_info(g, gv, us, param_file, diag)
02531
02532   ! Whenever thickness/T/S changes let the diag manager know, target grids
02533   ! for vertical remapping may need to be regenerated.
02534   ! FIXME: are h, T, S updated at the same time? Review these for T, S updates.
02535   call diag_update_remap_grids(diag)
02536
02537   ! Setup the diagnostic grid storage types
02538   call diag_grid_storage_init(cs%diag_pre_sync, g, diag)
02539   call diag_grid_storage_init(cs%diag_pre_dyn, g, diag)
02540
02541   ! Calculate masks for diagnostics arrays in non-native coordinates
02542   ! This step has to be done after set_axes_info() because the axes needed
02543   ! to be configured, and after diag_update_remap_grids() because the grids
02544   ! must be defined.
02545   call set_masks_for_axes(g, diag)
02546
02547   ! Diagnose static fields AND associate areas/volumes with axes
02548   call write_static_fields(g, gv, us, cs%tv, cs%diag)
02549   call calltree_waypoint("static fields written (initialize_MOM)")
02550
02551   ! Register the volume cell measure (must be one of first diagnostics)
02552   call register_cell_measure(g, cs%diag, time)
02553
02554   call cpu_clock_begin(id_clock_mom_init)
02555   if (cs%use_ALE_algorithm) then
02556     call ale_writecoordinatefile( cs%ALE_CSp, gv, dirs%output_directory )
02557   endif
02558   call cpu_clock_end(id_clock_mom_init)
02559   call calltree_waypoint("ALE initialized (initialize_MOM)")
02560
02561   cs%useMEKE = meke_init(time, g, us, param_file, diag, cs%MEKE_CSp, cs%MEKE, restart_csp)
02562
02563   call varmix_init(time, g, gv, us, param_file, diag, cs%VarMix)
02564   call set_visc_init(time, g, gv, us, param_file, diag, cs%visc, cs%set_visc_CSp, restart_csp, cs%OBC)
```

```
02565   call thickness_diffuse_init(time, g, gv, us, param_file, diag, cs%CDp, cs%thickness_diffuse_CSp)
02566
02567   if (cs%split) then
02568     allocate(eta(szi_(g),szj_(g))) ; eta(:,:) = 0.0
02569     call initialize_dyn_split_rk2(cs%u, cs%v, cs%h, cs%uh, cs%vh, eta, time, &
02570               g, gv, us, param_file, diag, cs%dyn_split_RK2_CSp, restart_csp, &
02571               cs%dt, cs%ADp, cs%CDp, mom_internal_state, cs%VarMix, cs%MEKE, &
02572               cs%thickness_diffuse_CSp,                                      &
02573               cs%OBC, cs%update_OBC_CSp, cs%ALE_CSp, cs%set_visc_CSp,        &
02574               cs%visc, dirs, cs%ntrunc, calc_dtbt=calc_dtbt, cont_stencil=cs%cont_stencil)
02575     if (cs%dtbt_reset_period > 0.0) then
02576       cs%dtbt_reset_interval = real_to_time(cs%dtbt_reset_period)
02577       ! Set dtbt_reset_time to be the next even multiple of dtbt_reset_interval.
02578       cs%dtbt_reset_time = time_init + cs%dtbt_reset_interval * &
02579                                ((time - time_init) / cs%dtbt_reset_interval)
02580       if ((cs%dtbt_reset_time > time) .and. calc_dtbt) then
02581         ! Back up dtbt_reset_time one interval to force dtbt to be calculated,
02582         ! because the restart was not aligned with the interval to recalculate
02583         ! dtbt, and dtbt was not read from a restart file.
02584         cs%dtbt_reset_time = cs%dtbt_reset_time - cs%dtbt_reset_interval
02585       endif
02586     endif
02587   elseif (cs%use_RK2) then
02588     call initialize_dyn_unsplit_rk2(cs%u, cs%v, cs%h, time, g, gv, us,     &
02589             param_file, diag, cs%dyn_unsplit_RK2_CSp, restart_csp,         &
02590             cs%ADp, cs%CDp, mom_internal_state, cs%MEKE, cs%OBC,           &
02591             cs%update_OBC_CSp, cs%ALE_CSp, cs%set_visc_CSp, cs%visc, dirs, &
02592             cs%ntrunc, cont_stencil=cs%cont_stencil)
02593   else
02594     call initialize_dyn_unsplit(cs%u, cs%v, cs%h, time, g, gv, us,         &
02595             param_file, diag, cs%dyn_unsplit_CSp, restart_csp,             &
02596             cs%ADp, cs%CDp, mom_internal_state, cs%MEKE, cs%OBC,           &
02597             cs%update_OBC_CSp, cs%ALE_CSp, cs%set_visc_CSp, cs%visc, dirs, &
02598             cs%ntrunc, cont_stencil=cs%cont_stencil)
02599   endif
02600
02601   call calltree_waypoint("dynamics initialized (initialize_MOM)")
02602
02603   cs%mixedlayer_restrat = mixedlayer_restrat_init(time, g, gv, us, param_file, diag, &
02604                                                   cs%mixedlayer_restrat_CSp, restart_csp)
02605   if (cs%mixedlayer_restrat) then
02606     if (.not.(bulkmixedlayer .or. cs%use_ALE_algorithm)) &
02607       call mom_error(fatal, "MOM: MIXEDLAYER_RESTRAT true requires a boundary layer scheme.")
02608     ! When DIABATIC_FIRST=False and using CS%visc%ML in mixedlayer_restrat we need to update after a
      restart
02609     if (.not. cs%diabatic_first .and. associated(cs%visc%MLD)) &
02610       call pass_var(cs%visc%MLD, g%domain, halo=1)
02611   endif
02612
02613   call mom_diagnostics_init(mom_internal_state, cs%ADp, cs%CDp, time, g, gv, us, &
02614                             param_file, diag, cs%diagnostics_CSp, cs%tv)
02615   call diag_copy_diag_to_storage(cs%diag_pre_sync, cs%h, cs%diag)
02616
02617   if (associated(cs%sponge_CSp)) &
02618     call init_sponge_diags(time, g, gv, us, diag, cs%sponge_CSp)
02619
02620   if (associated(cs%ALE_sponge_CSp)) &
02621     call init_ale_sponge_diags(time, g, diag, cs%ALE_sponge_CSp)
02622
02623   if (cs%adiabatic) then
02624     call adiabatic_driver_init(time, g, param_file, diag, cs%diabatic_CSp, &
02625                                cs%tracer_flow_CSp)
02626   else
02627     call diabatic_driver_init(time, g, gv, us, param_file, cs%use_ALE_algorithm, diag, &
02628                               cs%ADp, cs%CDp, cs%diabatic_CSp, cs%tracer_flow_CSp, &
02629                               cs%sponge_CSp, cs%ALE_sponge_CSp)
02630   endif
02631
02632   call tracer_advect_init(time, g, us, param_file, diag, cs%tracer_adv_CSp)
02633   call tracer_hor_diff_init(time, g, us, param_file, diag, cs%tv%eqn_of_state, cs%diabatic_CSp, &
02634                             cs%tracer_diff_CSp)
02635
02636   call lock_tracer_registry(cs%tracer_Reg)
02637   call calltree_waypoint("tracer registry now locked (initialize_MOM)")
02638
02639   ! now register some diagnostics since the tracer registry is now locked
02640   call register_surface_diags(time, g, us, cs%sfc_IDs, cs%diag, cs%tv)
02641   call register_diags(time, g, gv, us, cs%IDs, cs%diag)
02642   call register_transport_diags(time, g, gv, us, cs%transport_IDs, cs%diag)
02643   call register_tracer_diagnostics(cs%tracer_Reg, cs%h, time, diag, g, gv, us, &
02644                                    cs%use_ALE_algorithm)
02645   if (cs%use_ALE_algorithm) then
02646     call ale_register_diags(time, g, gv, us, diag, cs%ALE_CSp)
02647   endif
02648
02649   ! This subroutine initializes any tracer packages.
02650   new_sim = is_new_run(restart_csp)
```

```
02651   call tracer_flow_control_init(.not.new_sim, time, g, gv, us, cs%h, param_file, &
02652             cs%diag, cs%OBC, cs%tracer_flow_CSp, cs%sponge_CSp, &
02653             cs%ALE_sponge_CSp, cs%tv)
02654   if (present(tracer_flow_csp)) tracer_flow_csp => cs%tracer_flow_CSp
02655
02656   ! If running in offline tracer mode, initialize the necessary control structure and
02657   ! parameters
02658   if (present(offline_tracer_mode)) offline_tracer_mode=cs%offline_tracer_mode
02659
02660   if (cs%offline_tracer_mode) then
02661     ! Setup some initial parameterizations and also assign some of the subtypes
02662     call offline_transport_init(param_file, cs%offline_CSp, cs%diabatic_CSp, g, gv, us)
02663     call insert_offline_main( cs=cs%offline_CSp, ale_csp=cs%ALE_CSp, diabatic_csp=cs%diabatic_CSp, &
02664                               diag=cs%diag, obc=cs%OBC, tracer_adv_csp=cs%tracer_adv_CSp,              &
02665                               tracer_flow_csp=cs%tracer_flow_CSp, tracer_reg=cs%tracer_Reg,            &
02666                               tv=cs%tv, x_before_y = (mod(first_direction,2)==0), debug=cs%debug )
02667     call register_diags_offline_transport(time, cs%diag, cs%offline_CSp)
02668   endif
02669
02670   !--- set up group pass for u,v,T,S and h. pass_uv_T_S_h also is used in step_MOM
02671   call cpu_clock_begin(id_clock_pass_init)
02672   dynamics_stencil = min(3, g%Domain%nihalo, g%Domain%njhalo)
02673   call create_group_pass(pass_uv_t_s_h, cs%u, cs%v, g%Domain, halo=dynamics_stencil)
02674   if (use_temperature) then
02675     call create_group_pass(pass_uv_t_s_h, cs%tv%T, g%Domain, halo=dynamics_stencil)
02676     call create_group_pass(pass_uv_t_s_h, cs%tv%S, g%Domain, halo=dynamics_stencil)
02677   endif
02678   call create_group_pass(pass_uv_t_s_h, cs%h, g%Domain, halo=dynamics_stencil)
02679
02680   call do_group_pass(pass_uv_t_s_h, g%Domain)
02681
02682   if (associated(cs%visc%Kv_shear)) &
02683     call pass_var(cs%visc%Kv_shear, g%Domain, to_all+omit_corners, halo=1)
02684
02685   if (associated(cs%visc%Kv_slow)) &
02686     call pass_var(cs%visc%Kv_slow, g%Domain, to_all+omit_corners, halo=1)
02687
02688   call cpu_clock_end(id_clock_pass_init)
02689
02690   call register_obsolete_diagnostics(param_file, cs%diag)
02691
02692   if (use_frazil) then
02693     if (query_initialized(cs%tv%frazil, "frazil", restart_csp)) then
02694       ! Test whether the dimensional rescaling has changed for heat content.
02695       if ((us%kg_m3_to_R_restart*us%m_to_Z_restart*us%J_kg_to_Q_restart /= 0.0) .and. &
02696           ((us%J_kg_to_Q*us%kg_m3_to_R*us%m_to_Z) /= &
02697            (us%J_kg_to_Q_restart*us%kg_m3_to_R_restart*us%m_to_Z_restart)) ) then
02698         qrz_rescale = (us%J_kg_to_Q*us%kg_m3_to_R*us%m_to_Z) / &
02699                       (us%J_kg_to_Q_restart*us%kg_m3_to_R_restart*us%m_to_Z_restart)
02700         do j=js,je ; do i=is,ie
02701           cs%tv%frazil(i,j) = qrz_rescale * cs%tv%frazil(i,j)
02702         enddo ; enddo
02703       endif
02704     else
02705       cs%tv%frazil(:,:) = 0.0
02706     endif
02707   endif
02708
02709   if (cs%interp_p_surf) then
02710     cs%p_surf_prev_set = query_initialized(cs%p_surf_prev,"p_surf_prev",restart_csp)
02711
02712     if (cs%p_surf_prev_set) then
02713       ! Test whether the dimensional rescaling has changed for pressure.
02714       if ((us%kg_m3_to_R_restart*us%s_to_T_restart*us%m_to_L_restart /= 0.0) .and. &
02715           ((us%kg_m3_to_R*(us%m_to_L*us%s_to_T_restart)**2) /= &
02716            (us%kg_m3_to_R_restart*(us%m_to_L_restart*us%s_to_T)**2)) ) then
02717         rl2_t2_rescale = (us%kg_m3_to_R*(us%m_to_L*us%s_to_T_restart)**2) / &
02718                          (us%kg_m3_to_R_restart*(us%m_to_L_restart*us%s_to_T)**2)
02719         do j=js,je ; do i=is,ie
02720           cs%p_surf_prev(i,j) = rl2_t2_rescale * cs%p_surf_prev(i,j)
02721         enddo ; enddo
02722       endif
02723
02724       call pass_var(cs%p_surf_prev, g%domain)
02725     endif
02726   endif
02727
02728   if (use_ice_shelf .and. associated(cs%Hml)) then
02729     if (query_initialized(cs%Hml, "hML", restart_csp)) then
02730       ! Test whether the dimensional rescaling has changed for depths.
02731       if ((us%m_to_Z_restart /= 0.0) .and. (us%m_to_Z /= us%m_to_Z_restart) ) then
02732         z_rescale = us%m_to_Z / us%m_to_Z_restart
02733         do j=js,je ; do i=is,ie
02734           cs%Hml(i,j) = z_rescale * cs%Hml(i,j)
02735         enddo ; enddo
02736       endif
02737     endif
```

```
02738   endif
02739
02740   if (.not.query_initialized(cs%ave_ssh_ibc,"ave_ssh",restart_csp)) then
02741     if (cs%split) then
02742       call find_eta(cs%h, cs%tv, g, gv, us, cs%ave_ssh_ibc, eta, eta_to_m=1.0)
02743     else
02744       call find_eta(cs%h, cs%tv, g, gv, us, cs%ave_ssh_ibc, eta_to_m=1.0)
02745     endif
02746   endif
02747   if (cs%split) deallocate(eta)
02748
02749   cs%nstep_tot = 0
02750   if (present(count_calls)) cs%count_calls = count_calls
02751   call mom_sum_output_init(g_in, us, param_file, dirs%output_directory, &
02752                            cs%ntrunc, time_init, cs%sum_output_CSp)
02753
02754   ! Flag whether to save initial conditions in finish_MOM_initialization() or not.
02755   cs%write_IC = save_ic .and. &
02756                 .not.((dirs%input_filename(1:1) == 'r') .and. &
02757                       (len_trim(dirs%input_filename) == 1))
02758
02759   if (cs%ensemble_ocean) then
02760     call init_oda(time, g, gv, cs%odaCS)
02761   endif
02762
02763   !### This could perhaps go here instead of in finish_MOM_initialization?
02764   ! call fix_restart_scaling(GV)
02765   ! call fix_restart_unit_scaling(US)
02766
02767   call calltree_leave("initialize_MOM()")
02768   call cpu_clock_end(id_clock_init)
02769
02770 end subroutine initialize_mom
02771
02772 !> Finishes initializing MOM and writes out the initial conditions.
02773 subroutine finish_mom_initialization(Time, dirs, CS, restart_CSp)
02774   type(time_type),          intent(in)    :: Time        !< model time, used in this routine
02775   type(directories),        intent(in)    :: dirs        !< structure with directory paths
02776   type(mom_control_struct), pointer       :: CS          !< pointer to MOM control
      structure
02777   type(mom_restart_cs),     pointer       :: restart_CSp !< pointer to the restart control
02778                                                          !! structure that will be used for MOM.
02779   ! Local variables
02780   type(ocean_grid_type),   pointer :: G => null()  ! pointer to a structure containing
02781                                                    ! metrics and related information
02782   type(verticalgrid_type), pointer :: GV => null() ! Pointer to the vertical grid structure
02783   type(unit_scale_type),   pointer :: US => null() ! Pointer to a structure containing
02784                                                    ! various unit conversion factors
02785   type(mom_restart_cs),    pointer :: restart_CSp_tmp => null()
02786   real, allocatable :: z_interface(:,:,:) ! Interface heights [m]
02787   type(vardesc) :: vd
02788
02789   call cpu_clock_begin(id_clock_init)
02790   call calltree_enter("finish_MOM_initialization()")
02791
02792   ! Pointers for convenience
02793   g => cs%G ; gv => cs%GV ; us => cs%US
02794
02795   !### Move to initialize_MOM?
02796   call fix_restart_scaling(gv)
02797   call fix_restart_unit_scaling(us)
02798
02799   ! Write initial conditions
02800   if (cs%write_IC) then
02801     allocate(restart_csp_tmp)
02802     restart_csp_tmp = restart_csp
02803     allocate(z_interface(szi_(g),szj_(g),szk_(g)+1))
02804     call find_eta(cs%h, cs%tv, g, gv, us, z_interface, eta_to_m=1.0)
02805     call register_restart_field(z_interface, "eta", .true., restart_csp_tmp, &
02806                                 "Interface heights", "meter", z_grid='i')
02807
02808     call save_restart(dirs%output_directory, time, cs%G_in, &
02809                       restart_csp_tmp, filename=cs%IC_file, gv=gv)
02810     deallocate(z_interface)
02811     deallocate(restart_csp_tmp)
02812   endif
02813
02814   call write_energy(cs%u, cs%v, cs%h, cs%tv, time, 0, g, gv, us, &
02815                     cs%sum_output_CSp, cs%tracer_flow_CSp)
02816
02817   call calltree_leave("finish_MOM_initialization()")
02818   call cpu_clock_end(id_clock_init)
02819
02820 end subroutine finish_mom_initialization
02821
02822 !> Register certain diagnostics
02823 subroutine register_diags(Time, G, GV, US, IDs, diag)
```

```
02824   type(time_type),          intent(in)    :: Time  !< current model time
02825   type(ocean_grid_type),    intent(in)    :: G     !< ocean grid structure
02826   type(verticalgrid_type), intent(in)    :: GV    !< ocean vertical grid structure
02827   type(unit_scale_type),    intent(inout) :: US    !< A dimensional unit scaling type
02828   type(mom_diag_ids),       intent(inout) :: IDs   !< A structure with the diagnostic IDs.
02829   type(diag_ctrl),          intent(inout) :: diag  !< regulates diagnostic output
02830
02831   real :: H_convert
02832   character(len=48) :: thickness_units
02833
02834   thickness_units = get_thickness_units(gv)
02835   if (gv%Boussinesq) then
02836     h_convert = gv%H_to_m
02837   else
02838     h_convert = gv%H_to_kg_m2
02839   endif
02840
02841   ! Diagnostics of the rapidly varying dynamic state
02842   ids%id_u = register_diag_field('ocean_model', 'u_dyn', diag%axesCuL, time, &
02843       'Zonal velocity after the dynamics update', 'm s-1', conversion=us%L_T_to_m_s)
02844   ids%id_v = register_diag_field('ocean_model', 'v_dyn', diag%axesCvL, time, &
02845       'Meridional velocity after the dynamics update', 'm s-1', conversion=us%L_T_to_m_s)
02846   ids%id_h = register_diag_field('ocean_model', 'h_dyn', diag%axesTL, time, &
02847       'Layer Thickness after the dynamics update', thickness_units, &
02848       v_extensive=.true., conversion=h_convert)
02849   ids%id_ssh_inst = register_diag_field('ocean_model', 'SSH_inst', diag%axesT1, &
02850       time, 'Instantaneous Sea Surface Height', 'm')
02851 end subroutine register_diags
02852
02853 !> Set up CPU clock IDs for timing various subroutines.
02854 subroutine mom_timing_init(CS)
02855   type(mom_control_struct), intent(in) :: CS  !< control structure set up by
       initialize_MOM.
02856
02857  id_clock_ocean    = cpu_clock_id('Ocean', grain=clock_component)
02858  id_clock_dynamics = cpu_clock_id('Ocean dynamics', grain=clock_subcomponent)
02859  id_clock_thermo   = cpu_clock_id('Ocean thermodynamics and tracers', grain=
      clock_subcomponent)
02860  id_clock_other    = cpu_clock_id('Ocean Other', grain=clock_subcomponent)
02861  id_clock_tracer   = cpu_clock_id('(Ocean tracer advection)', grain=clock_module_driver)
02862  if (.not.cs%adiabatic) then
02863    id_clock_diabatic = cpu_clock_id('(Ocean diabatic driver)', grain=clock_module_driver)
02864  else
02865    id_clock_adiabatic = cpu_clock_id('(Ocean adiabatic driver)', grain=
      clock_module_driver)
02866  endif
02867
02868  id_clock_continuity = cpu_clock_id('(Ocean continuity equation *)', grain=clock_module)
02869  id_clock_bbl_visc   = cpu_clock_id('(Ocean set BBL viscosity)', grain=clock_module)
02870  id_clock_pass       = cpu_clock_id('(Ocean message passing *)', grain=clock_module)
02871  id_clock_mom_init   = cpu_clock_id('(Ocean MOM_initialize_state)', grain=clock_module)
02872  id_clock_pass_init  = cpu_clock_id('(Ocean init message passing *)', grain=clock_routine
      )
02873  if (cs%thickness_diffuse) &
02874    id_clock_thick_diff = cpu_clock_id('(Ocean thickness diffusion *)', grain=
      clock_module)
02875 !if (CS%mixedlayer_restrat) &
02876    id_clock_ml_restrat = cpu_clock_id('(Ocean mixed layer restrat)', grain=clock_module)
02877  id_clock_diagnostics  = cpu_clock_id('(Ocean collective diagnostics)', grain=
      clock_module)
02878  id_clock_z_diag       = cpu_clock_id('(Ocean Z-space diagnostics)', grain=clock_module)
02879  id_clock_ale          = cpu_clock_id('(Ocean ALE)', grain=clock_module)
02880  if (cs%offline_tracer_mode) then
02881   id_clock_offline_tracer = cpu_clock_id('Ocean offline tracers', grain=
      clock_subcomponent)
02882  endif
02883
02884 end subroutine mom_timing_init
02885
02886 !> Set the fields that are needed for bitwise identical restarting
02887 !! the time stepping scheme.  In addition to those specified here
02888 !! directly, there may be fields related to the forcing or to the
02889 !! barotropic solver that are needed; these are specified in sub-
02890 !! routines that are called from this one.
02891 !!
02892 !! This routine should be altered if there are any changes to the
02893 !! time stepping scheme.  The CHECK_RESTART facility may be used to
02894 !! confirm that all needed restart fields have been included.
02895 subroutine set_restart_fields(GV, US, param_file, CS, restart_CSp)
02896   type(verticalgrid_type),  intent(inout) :: GV         !< ocean vertical grid structure
02897   type(unit_scale_type),    intent(inout) :: US         !< A dimensional unit scaling type
02898   type(param_file_type),    intent(in) :: param_file    !< opened file for parsing to get parameters
02899   type(mom_control_struct), intent(in) :: CS            !< control structure set up by
       initialize_MOM
02900   type(mom_restart_cs),     pointer    :: restart_CSp   !< pointer to the restart control
02901                                                         !! structure that will be used for MOM.
02902   ! Local variables
```

```
02903   logical :: use_ice_shelf ! Needed to determine whether to add CS%Hml to restarts
02904   character(len=48) :: thickness_units, flux_units
02905   type(vardesc) :: u_desc, v_desc
02906
02907   thickness_units = get_thickness_units(gv)
02908   flux_units = get_flux_units(gv)
02909
02910   u_desc = var_desc("u", "m s-1", "Zonal velocity", hor_grid='Cu')
02911   v_desc = var_desc("v", "m s-1", "Meridional velocity", hor_grid='Cv')
02912
02913   if (associated(cs%tv%T)) &
02914     call register_restart_field(cs%tv%T, "Temp", .true., restart_csp, &
02915                                 "Potential Temperature", "degC")
02916   if (associated(cs%tv%S)) &
02917     call register_restart_field(cs%tv%S, "Salt", .true., restart_csp, &
02918                                 "Salinity", "PPT")
02919
02920   call register_restart_field(cs%h, "h", .true., restart_csp, &
02921                               "Layer Thickness", thickness_units)
02922
02923   call register_restart_pair(cs%u, cs%v, u_desc, v_desc, .true., restart_csp)
02924
02925   if (associated(cs%tv%frazil)) &
02926     call register_restart_field(cs%tv%frazil, "frazil", .false., restart_csp, &
02927                                 "Frazil heat flux into ocean", "J m-2")
02928
02929   if (cs%interp_p_surf) then
02930     call register_restart_field(cs%p_surf_prev, "p_surf_prev", .false., restart_csp, &
02931                                 "Previous ocean surface pressure", "Pa")
02932   endif
02933
02934   call register_restart_field(cs%ave_ssh_ibc, "ave_ssh", .false., restart_csp, &
02935                               "Time average sea surface height", "meter")
02936
02937   ! hML is needed when using the ice shelf module
02938   call get_param(param_file, '', "ICE_SHELF", use_ice_shelf, default=.false., &
02939                  do_not_log=.true.)
02940   if (use_ice_shelf .and. associated(cs%Hml)) then
02941     call register_restart_field(cs%Hml, "hML", .false., restart_csp, &
02942                                 "Mixed layer thickness", "meter")
02943   endif
02944
02945   ! Register scalar unit conversion factors.
02946   call register_restart_field(us%m_to_Z_restart, "m_to_Z", .false., restart_csp, &
02947                               "Height unit conversion factor", "Z meter-1")
02948   call register_restart_field(gv%m_to_H_restart, "m_to_H", .false., restart_csp, &
02949                               "Thickness unit conversion factor", "H meter-1")
02950   call register_restart_field(us%m_to_L_restart, "m_to_L", .false., restart_csp, &
02951                               "Length unit conversion factor", "L meter-1")
02952   call register_restart_field(us%s_to_T_restart, "s_to_T", .false., restart_csp, &
02953                               "Time unit conversion factor", "T second-1")
02954   call register_restart_field(us%kg_m3_to_R_restart, "kg_m3_to_R", .false., restart_csp, &
02955                               "Density unit conversion factor", "R m3 kg-1")
02956   call register_restart_field(us%J_kg_to_Q_restart, "J_kg_to_Q", .false., restart_csp, &
02957                               "Heat content unit conversion factor.", units="Q kg J-1")
02958
02959 end subroutine set_restart_fields
02960
02961 !> Apply a correction to the sea surface height to compensate
02962 !! for the atmospheric pressure (the inverse barometer).
02963 subroutine adjust_ssh_for_p_atm(tv, G, GV, US, ssh, p_atm, use_EOS)
02964   type(thermo_var_ptrs),             intent(in)    :: tv  !< A structure pointing to various thermodynamic
      variables
02965   type(ocean_grid_type),             intent(in)    :: G   !< ocean grid structure
02966   type(verticalgrid_type),           intent(in)    :: GV  !< ocean vertical grid structure
02967   type(unit_scale_type),             intent(in)    :: US  !< A dimensional unit scaling type
02968   real, dimension(SZI_(G),SZJ_(G)),  intent(inout) :: ssh !< time mean surface height [m]
02969   real, dimension(:,:),    optional, pointer       :: p_atm !< Ocean surface pressure [R L2 T-2 ~> Pa]
02970   logical,                 optional, intent(in)    :: use_EOS !< If true, calculate the density for
02971                                                               !! the SSH correction using the equation of state.
02972
02973   real :: Rho_conv(szi_(g))  ! The density used to convert surface pressure to
02974                              ! a corrected effective SSH [R ~> kg m-3].
02975   real :: IgR0       ! The SSH conversion factor from R L2 T-2 to m [m T2 R-1 L-2 ~> m Pa-1].
02976   logical :: calc_rho
02977   integer, dimension(2) :: EOSdom ! The i-computational domain for the equation of state
02978   integer :: i, j, is, ie, js, je
02979
02980   is = g%isc ; ie = g%iec ; js = g%jsc ; je = g%jec
02981   eosdom(:) = eos_domain(g%HI)
02982   if (present(p_atm)) then ; if (associated(p_atm)) then
02983     calc_rho = associated(tv%eqn_of_state)
02984     if (present(use_eos) .and. calc_rho) calc_rho = use_eos
02985     ! Correct the output sea surface height for the contribution from the ice pressure.
02986     do j=js,je
02987       if (calc_rho) then
02988         call calculate_density(tv%T(:,j,1), tv%S(:,j,1), 0.5*p_atm(:,j), rho_conv, &
```

```
02989                                 tv%eqn_of_state, eosdom)
02990         do i=is,ie
02991           igr0 = us%Z_to_m / (rho_conv(i) * gv%g_Earth)
02992           ssh(i,j) = ssh(i,j) + p_atm(i,j) * igr0
02993         enddo
02994       else
02995         do i=is,ie
02996           ssh(i,j) = ssh(i,j) + p_atm(i,j) * (us%Z_to_m / (gv%Rho0 * gv%g_Earth))
02997         enddo
02998       endif
02999     enddo
03000   endif ; endif
03001
03002 end subroutine adjust_ssh_for_p_atm
03003
03004 !> Set the surface (return) properties of the ocean model by
03005 !! setting the appropriate fields in sfc_state.  Unused fields
03006 !! are set to NULL or are unallocated.
03007 subroutine extract_surface_state(CS, sfc_state_in)
03008   type(mom_control_struct), pointer    :: CS !< Master MOM control structure
03009   type(surface), target, intent(inout) :: sfc_state_in !< transparent ocean surface state
03010                                           !! structure shared with the calling routine
03011                                           !! data in this structure is intent out.
03012
03013   ! Local variables
03014   real :: hu, hv  ! Thicknesses interpolated to velocity points [H ~> m or kg m-2]
03015   type(ocean_grid_type),   pointer :: G => null() !< pointer to a structure containing
03016                                                   !! metrics and related information
03017   type(ocean_grid_type),   pointer :: G_in => null() !< Input grid metric
03018   type(verticalgrid_type), pointer :: GV => null() !< structure containing vertical grid info
03019   type(unit_scale_type),   pointer :: US => null() !< structure containing various unit conversion factors
03020   type(surface),           pointer :: sfc_state => null()  ! surface state on the model grid
03021   real, dimension(:,:,:),  pointer :: &
03022     h => null()    !< h : layer thickness [H ~> m or kg m-2]
03023   real :: depth(szi_(cs%g))  !< Distance from the surface in depth units [Z ~> m] or [H ~> m or kg m-2]
03024   real :: depth_ml           !< Depth over which to average to determine mixed
03025                              !! layer properties [Z ~> m] or [H ~> m or kg m-2]
03026   real :: dh                 !< Thickness of a layer within the mixed layer [Z ~> m] or [H ~> m or kg m-2]
03027   real :: mass               !< Mass per unit area of a layer [R Z ~> kg m-2]
03028   real :: T_freeze           !< freezing temperature [degC]
03029   real :: I_depth            !< The inverse of depth [Z-1 ~> m-1] or [H-1 ~> m-1 or m2 kg-1]
03030   real :: missing_depth      !< The portion of depth_ml that can not be found in a column [H ~> m or kg
      m-2]
03031   real :: H_rescale          !< A conversion factor from thickness units to the units used in the
03032                              !! calculation of properties of the uppermost ocean [nondim] or [Z H-1 ~> 1 or
      m3 kg-1]
03033                              !  After the ANSWERS_2018 flag has been obsoleted, H_rescale will be 1.
03034   real :: delT(szi_(cs%g))   !< Depth integral of T-T_freeze [Z degC ~> m degC]
03035   logical :: use_temperature !< If true, temp and saln used as state variables.
03036   integer :: i, j, k, is, ie, js, je, nz, numberOfErrors, ig, jg
03037   integer :: isd, ied, jsd, jed
03038   integer :: iscB, iecB, jscB, jecB, isdB, iedB, jsdB, jedB
03039   logical :: localError
03040   character(240) :: msg
03041   integer :: turns    ! Number of quarter turns
03042
03043   call calltree_enter("extract_surface_state(), MOM.F90")
03044   g => cs%G ; g_in => cs%G_in ; gv => cs%GV ; us => cs%US
03045   is  = g%isc ; ie  = g%iec ; js  = g%jsc ; je  = g%jec ; nz = gv%ke
03046   isd = g%isd ; ied = g%ied ; jsd = g%jsd ; jed = g%jed
03047   iscb = g%iscB ; iecb = g%iecB; jscb = g%jscB ; jecb = g%jecB
03048   isdb = g%isdB ; iedb = g%iedB; jsdb = g%jsdB ; jedb = g%jedB
03049   h => cs%h
03050
03051   use_temperature = associated(cs%tv%T)
03052
03053   turns = 0
03054   if (cs%rotate_index) &
03055     turns = g%HI%turns
03056
03057   if (.not.sfc_state_in%arrays_allocated) &
03058     !  Consider using a run-time flag to determine whether to do the vertical
03059     ! integrals, since the 3-d sums are not negligible in cost.
03060     call allocate_surface_state(sfc_state_in, g_in, use_temperature, &
03061         do_integrals=.true., omit_frazil=.not.associated(cs%tv%frazil))
03062
03063   if (cs%rotate_index) then
03064     allocate(sfc_state)
03065     call allocate_surface_state(sfc_state, g, use_temperature, &
03066         do_integrals=.true., omit_frazil=.not.associated(cs%tv%frazil))
03067   else
03068     sfc_state => sfc_state_in
03069   endif
03070
03071   sfc_state%T_is_conT = cs%tv%T_is_conT
03072   sfc_state%S_is_absS = cs%tv%S_is_absS
03073
```

```
03074   do j=js,je ; do i=is,ie
03075     sfc_state%sea_lev(i,j) = us%m_to_Z*cs%ave_ssh_ibc(i,j)
03076   enddo ; enddo
03077
03078   if (allocated(sfc_state%frazil) .and. associated(cs%tv%frazil)) then ; do j=js,je ; do i=is,ie
03079     sfc_state%frazil(i,j) = cs%tv%frazil(i,j)
03080   enddo ; enddo ; endif
03081
03082   ! copy Hml into sfc_state, so that caps can access it
03083   if (associated(cs%Hml)) then
03084     do j=js,je ; do i=is,ie
03085       sfc_state%Hml(i,j) = cs%Hml(i,j)
03086     enddo ; enddo
03087   endif
03088
03089   if (cs%Hmix < 0.0) then  ! A bulk mixed layer is in use, so layer 1 has the properties
03090     if (use_temperature) then ; do j=js,je ; do i=is,ie
03091       sfc_state%SST(i,j) = cs%tv%T(i,j,1)
03092       sfc_state%SSS(i,j) = cs%tv%S(i,j,1)
03093     enddo ; enddo ; endif
03094     do j=js,je ; do i=is-1,ie
03095       sfc_state%u(i,j) = cs%u(i,j,1)
03096     enddo ; enddo
03097     do j=js-1,je ; do i=is,ie
03098       sfc_state%v(i,j) = cs%v(i,j,1)
03099     enddo ; enddo
03100
03101   else  ! (CS%Hmix >= 0.0)
03102     h_rescale = 1.0 ; if (cs%answers_2018) h_rescale = gv%H_to_Z
03103     depth_ml = cs%Hmix
03104     if (.not.cs%answers_2018) depth_ml = cs%Hmix*gv%Z_to_H
03105     ! Determine the mean tracer properties of the uppermost depth_ml fluid.
03106
03107     !$OMP parallel do default(shared) private(depth,dh)
03108     do j=js,je
03109       do i=is,ie
03110         depth(i) = 0.0
03111         if (use_temperature) then
03112           sfc_state%SST(i,j) = 0.0 ; sfc_state%SSS(i,j) = 0.0
03113         else
03114           sfc_state%sfc_density(i,j) = 0.0
03115         endif
03116       enddo
03117
03118       do k=1,nz ; do i=is,ie
03119         if (depth(i) + h(i,j,k)*h_rescale < depth_ml) then
03120           dh = h(i,j,k)*h_rescale
03121         elseif (depth(i) < depth_ml) then
03122           dh = depth_ml - depth(i)
03123         else
03124           dh = 0.0
03125         endif
03126         if (use_temperature) then
03127           sfc_state%SST(i,j) = sfc_state%SST(i,j) + dh * cs%tv%T(i,j,k)
03128           sfc_state%SSS(i,j) = sfc_state%SSS(i,j) + dh * cs%tv%S(i,j,k)
03129         else
03130           sfc_state%sfc_density(i,j) = sfc_state%sfc_density(i,j) + dh * gv%Rlay(k)
03131         endif
03132         depth(i) = depth(i) + dh
03133       enddo ; enddo
03134   ! Calculate the average properties of the mixed layer depth.
03135     do i=is,ie
03136       if (cs%answers_2018) then
03137         if (depth(i) < gv%H_subroundoff*h_rescale) &
03138             depth(i) = gv%H_subroundoff*h_rescale
03139         if (use_temperature) then
03140           sfc_state%SST(i,j) = sfc_state%SST(i,j) / depth(i)
03141           sfc_state%SSS(i,j) = sfc_state%SSS(i,j) / depth(i)
03142         else
03143           sfc_state%sfc_density(i,j) = sfc_state%sfc_density(i,j) / depth(i)
03144         endif
03145       else
03146         if (depth(i) < gv%H_subroundoff*h_rescale) then
03147           i_depth = 1.0 / (gv%H_subroundoff*h_rescale)
03148           missing_depth = gv%H_subroundoff*h_rescale - depth(i)
03149           if (use_temperature) then
03150             sfc_state%SST(i,j) = (sfc_state%SST(i,j) + missing_depth*cs%tv%T(i,j,1)) * i_depth
03151             sfc_state%SSS(i,j) = (sfc_state%SSS(i,j) + missing_depth*cs%tv%S(i,j,1)) * i_depth
03152           else
03153             sfc_state%sfc_density(i,j) = (sfc_state%sfc_density(i,j) + &
03154                                           missing_depth*gv%Rlay(1)) * i_depth
03155           endif
03156         else
03157           i_depth = 1.0 / depth(i)
03158           if (use_temperature) then
03159             sfc_state%SST(i,j) = sfc_state%SST(i,j) * i_depth
03160             sfc_state%SSS(i,j) = sfc_state%SSS(i,j) * i_depth
```

```
03161             else
03162               sfc_state%sfc_density(i,j) = sfc_state%sfc_density(i,j) * i_depth
03163             endif
03164           endif
03165         endif
03166       enddo
03167     enddo ! end of j loop
03168
03169 !   Determine the mean velocities in the uppermost depth_ml fluid.
03170     ! NOTE: Velocity loops start on `[ij]s-1` in order to update halo values
03171     !       required by the speed diagnostic on the non-symmetric grid.
03172     !       This assumes that u and v halos have already been updated.
03173     if (cs%Hmix_UV>0.) then
03174       depth_ml = cs%Hmix_UV
03175       if (.not.cs%answers_2018) depth_ml = cs%Hmix_UV*gv%Z_to_H
03176       !$OMP parallel do default(shared) private(depth,dh,hv)
03177       do j=js-1,ie
03178         do i=is,ie
03179           depth(i) = 0.0
03180           sfc_state%v(i,j) = 0.0
03181         enddo
03182         do k=1,nz ; do i=is,ie
03183           hv = 0.5 * (h(i,j,k) + h(i,j+1,k)) * h_rescale
03184           if (depth(i) + hv < depth_ml) then
03185             dh = hv
03186           elseif (depth(i) < depth_ml) then
03187             dh = depth_ml - depth(i)
03188           else
03189             dh = 0.0
03190           endif
03191           sfc_state%v(i,j) = sfc_state%v(i,j) + dh * cs%v(i,j,k)
03192           depth(i) = depth(i) + dh
03193         enddo ; enddo
03194         ! Calculate the average properties of the mixed layer depth.
03195         do i=is,ie
03196           sfc_state%v(i,j) = sfc_state%v(i,j) / max(depth(i), gv%H_subroundoff*h_rescale)
03197         enddo
03198       enddo ! end of j loop
03199
03200       !$OMP parallel do default(shared) private(depth,dh,hu)
03201       do j=js,je
03202         do i=is-1,ie
03203           depth(i) = 0.0
03204           sfc_state%u(i,j) = 0.0
03205         enddo
03206         do k=1,nz ; do i=is-1,ie
03207           hu = 0.5 * (h(i,j,k) + h(i+1,j,k)) * h_rescale
03208           if (depth(i) + hu < depth_ml) then
03209             dh = hu
03210           elseif (depth(i) < depth_ml) then
03211             dh = depth_ml - depth(i)
03212           else
03213             dh = 0.0
03214           endif
03215           sfc_state%u(i,j) = sfc_state%u(i,j) + dh * cs%u(i,j,k)
03216           depth(i) = depth(i) + dh
03217         enddo ; enddo
03218         ! Calculate the average properties of the mixed layer depth.
03219         do i=is-1,ie
03220           sfc_state%u(i,j) = sfc_state%u(i,j) / max(depth(i), gv%H_subroundoff*h_rescale)
03221         enddo
03222       enddo ! end of j loop
03223     else ! Hmix_UV<=0.
03224       do j=js,je ; do i=is-1,ie
03225         sfc_state%u(i,j) = cs%u(i,j,1)
03226       enddo ; enddo
03227       do j=js-1,je ; do i=is,ie
03228         sfc_state%v(i,j) = cs%v(i,j,1)
03229       enddo ; enddo
03230     endif
03231   endif  ! (CS%Hmix >= 0.0)
03232
03233
03234   if (allocated(sfc_state%melt_potential)) then
03235     !$OMP parallel do default(shared) private(depth_ml, dh, T_freeze, depth, delT)
03236     do j=js,je
03237       do i=is,ie
03238         depth(i) = 0.0
03239         delt(i) = 0.0
03240       enddo
03241
03242       do k=1,nz ; do i=is,ie
03243         depth_ml = min(cs%HFrz, cs%visc%MLD(i,j))
03244         if (depth(i) + h(i,j,k)*gv%H_to_Z < depth_ml) then
03245           dh = h(i,j,k)*gv%H_to_Z
03246         elseif (depth(i) < depth_ml) then
03247           dh = depth_ml - depth(i)
```

```
03248         else
03249           dh = 0.0
03250         endif
03251
03252         ! p=0 OK, HFrz ~ 10 to 20m
03253         call calculate_tfreeze(cs%tv%S(i,j,k), 0.0, t_freeze, cs%tv%eqn_of_state)
03254         depth(i) = depth(i) + dh
03255         delt(i) =  delt(i) + dh * (cs%tv%T(i,j,k) - t_freeze)
03256       enddo ; enddo
03257
03258       do i=is,ie
03259        ! set melt_potential to zero to avoid passing previous values
03260        sfc_state%melt_potential(i,j) = 0.0
03261
03262        if (g%mask2dT(i,j)>0.) then
03263          ! instantaneous melt_potential [Q R Z ~> J m-2]
03264          sfc_state%melt_potential(i,j) = cs%tv%C_p * gv%Rho0 * delt(i)
03265        endif
03266       enddo
03267     enddo ! end of j loop
03268   endif   ! melt_potential
03269
03270   if (allocated(sfc_state%salt_deficit) .and. associated(cs%tv%salt_deficit)) then
03271     !$OMP parallel do default(shared)
03272     do j=js,je ; do i=is,ie
03273       ! Convert from gSalt to kgSalt
03274       sfc_state%salt_deficit(i,j) = 0.001 * cs%tv%salt_deficit(i,j)
03275     enddo ; enddo
03276   endif
03277   if (allocated(sfc_state%TempxPmE) .and. associated(cs%tv%TempxPmE)) then
03278     !$OMP parallel do default(shared)
03279     do j=js,je ; do i=is,ie
03280       sfc_state%TempxPmE(i,j) = cs%tv%TempxPmE(i,j)
03281     enddo ; enddo
03282   endif
03283   if (allocated(sfc_state%internal_heat) .and. associated(cs%tv%internal_heat)) then
03284     !$OMP parallel do default(shared)
03285     do j=js,je ; do i=is,ie
03286       sfc_state%internal_heat(i,j) = cs%tv%internal_heat(i,j)
03287     enddo ; enddo
03288   endif
03289   if (allocated(sfc_state%taux_shelf) .and. associated(cs%visc%taux_shelf)) then
03290     !$OMP parallel do default(shared)
03291     do j=js,je ; do i=is-1,ie
03292       sfc_state%taux_shelf(i,j) = cs%visc%taux_shelf(i,j)
03293     enddo ; enddo
03294   endif
03295   if (allocated(sfc_state%tauy_shelf) .and. associated(cs%visc%tauy_shelf)) then
03296     !$OMP parallel do default(shared)
03297     do j=js-1,je ; do i=is,ie
03298       sfc_state%tauy_shelf(i,j) = cs%visc%tauy_shelf(i,j)
03299     enddo ; enddo
03300   endif
03301
03302   if (allocated(sfc_state%ocean_mass) .and. allocated(sfc_state%ocean_heat) .and. &
03303       allocated(sfc_state%ocean_salt)) then
03304     !$OMP parallel do default(shared)
03305     do j=js,je ; do i=is,ie
03306       sfc_state%ocean_mass(i,j) = 0.0
03307       sfc_state%ocean_heat(i,j) = 0.0 ; sfc_state%ocean_salt(i,j) = 0.0
03308     enddo ; enddo
03309     !$OMP parallel do default(shared) private(mass)
03310     do j=js,je ; do k=1,nz; do i=is,ie
03311       mass = gv%H_to_RZ*h(i,j,k)
03312       sfc_state%ocean_mass(i,j) = sfc_state%ocean_mass(i,j) + mass
03313       sfc_state%ocean_heat(i,j) = sfc_state%ocean_heat(i,j) + mass * cs%tv%T(i,j,k)
03314       sfc_state%ocean_salt(i,j) = sfc_state%ocean_salt(i,j) + mass * (1.0e-3*cs%tv%S(i,j,k))
03315     enddo ; enddo ; enddo
03316   else
03317     if (allocated(sfc_state%ocean_mass)) then
03318       !$OMP parallel do default(shared)
03319       do j=js,je ; do i=is,ie ; sfc_state%ocean_mass(i,j) = 0.0 ; enddo ; enddo
03320       !$OMP parallel do default(shared)
03321       do j=js,je ; do k=1,nz ; do i=is,ie
03322         sfc_state%ocean_mass(i,j) = sfc_state%ocean_mass(i,j) + gv%H_to_RZ*h(i,j,k)
03323       enddo ; enddo ; enddo
03324     endif
03325     if (allocated(sfc_state%ocean_heat)) then
03326       !$OMP parallel do default(shared)
03327       do j=js,je ; do i=is,ie ; sfc_state%ocean_heat(i,j) = 0.0 ; enddo ; enddo
03328       !$OMP parallel do default(shared) private(mass)
03329       do j=js,je ; do k=1,nz ; do i=is,ie
03330         mass = gv%H_to_RZ*h(i,j,k)
03331         sfc_state%ocean_heat(i,j) = sfc_state%ocean_heat(i,j) + mass*cs%tv%T(i,j,k)
03332       enddo ; enddo ; enddo
03333     endif
03334     if (allocated(sfc_state%ocean_salt)) then
```

```
03335       !$OMP parallel do default(shared)
03336       do j=js,je ; do i=is,ie ; sfc_state%ocean_salt(i,j) = 0.0 ; enddo ; enddo
03337       !$OMP parallel do default(shared) private(mass)
03338       do j=js,je ; do k=1,nz ; do i=is,ie
03339         mass = gv%H_to_RZ*h(i,j,k)
03340         sfc_state%ocean_salt(i,j) = sfc_state%ocean_salt(i,j) + mass * (1.0e-3*cs%tv%S(i,j,k))
03341       enddo ; enddo ; enddo
03342     endif
03343   endif
03344
03345   if (associated(cs%tracer_flow_CSp)) then
03346     call call_tracer_surface_state(sfc_state, h, g, cs%tracer_flow_CSp)
03347   endif
03348
03349   if (cs%check_bad_sfc_vals) then
03350     numberoferrors=0 ! count number of errors
03351     do j=js,je; do i=is,ie
03352       if (g%mask2dT(i,j)>0.) then
03353         localerror = sfc_state%sea_lev(i,j) <= -g%bathyT(i,j) &
03354                 .or. sfc_state%sea_lev(i,j) >=  cs%bad_val_ssh_max  &
03355                 .or. sfc_state%sea_lev(i,j) <= -cs%bad_val_ssh_max  &
03356                 .or. sfc_state%sea_lev(i,j) + g%bathyT(i,j) < cs%bad_val_col_thick
03357         if (use_temperature) localerror = localerror &
03358                 .or. sfc_state%SSS(i,j)<0.                      &
03359                 .or. sfc_state%SSS(i,j)>=cs%bad_val_sss_max     &
03360                 .or. sfc_state%SST(i,j)< cs%bad_val_sst_min     &
03361                 .or. sfc_state%SST(i,j)>=cs%bad_val_sst_max
03362         if (localerror) then
03363           numberoferrors=numberoferrors+1
03364           if (numberoferrors<9) then ! Only report details for the first few errors
03365             ig = i + g%HI%idg_offset ! Global i-index
03366             jg = j + g%HI%jdg_offset ! Global j-index
03367             if (use_temperature) then
03368               write(msg(1:240),'(2(a,i4,x),4(a,f8.3,x),8(a,es11.4,x))') &
03369                 'Extreme surface sfc_state detected: i=',ig,'j=',jg, &
03370                 'lon=',g%geoLonT(i,j), 'lat=',g%geoLatT(i,j), &
03371                 'x=',g%gridLonT(ig), 'y=',g%gridLatT(jg), &
03372                 'D=',cs%US%Z_to_m*g%bathyT(i,j),  'SSH=',cs%US%Z_to_m*sfc_state%sea_lev(i,j), &
03373                 'SST=',sfc_state%SST(i,j), 'SSS=',sfc_state%SSS(i,j), &
03374                 'U-=',us%L_T_to_m_s*sfc_state%u(i-1,j), 'U+=',us%L_T_to_m_s*sfc_state%u(i,j), &
03375                 'V-=',us%L_T_to_m_s*sfc_state%v(i,j-1), 'V+=',us%L_T_to_m_s*sfc_state%v(i,j)
03376             else
03377               write(msg(1:240),'(2(a,i4,x),4(a,f8.3,x),6(a,es11.4))') &
03378                 'Extreme surface sfc_state detected: i=',ig,'j=',jg, &
03379                 'lon=',g%geoLonT(i,j), 'lat=',g%geoLatT(i,j), &
03380                 'x=',g%gridLonT(i), 'y=',g%gridLatT(j), &
03381                 'D=',cs%US%Z_to_m*g%bathyT(i,j),  'SSH=',cs%US%Z_to_m*sfc_state%sea_lev(i,j), &
03382                 'U-=',us%L_T_to_m_s*sfc_state%u(i-1,j), 'U+=',us%L_T_to_m_s*sfc_state%u(i,j), &
03383                 'V-=',us%L_T_to_m_s*sfc_state%v(i,j-1), 'V+=',us%L_T_to_m_s*sfc_state%v(i,j)
03384             endif
03385             call mom_error(warning, trim(msg), all_print=.true.)
03386           elseif (numberoferrors==9) then ! Indicate once that there are more errors
03387             call mom_error(warning, 'There were more unreported extreme events!', all_print=.true.)
03388           endif ! numberOfErrors
03389         endif ! localError
03390       endif ! mask2dT
03391     enddo ; enddo
03392     call sum_across_pes(numberoferrors)
03393     if (numberoferrors>0) then
03394       write(msg(1:240),'(3(a,i9,x))') 'There were a total of ',numberoferrors, &
03395           'locations detected with extreme surface values!'
03396       call mom_error(fatal, trim(msg))
03397     endif
03398   endif
03399
03400   if (cs%debug) call mom_surface_chksum("Post extract_sfc", sfc_state, g, us, haloshift=0)
03401
03402   ! Rotate sfc_state back onto the input grid, sfc_state_in
03403   if (cs%rotate_index) then
03404     call rotate_surface_state(sfc_state, g, sfc_state_in, g_in, -turns)
03405     call deallocate_surface_state(sfc_state)
03406   endif
03407
03408   call calltree_leave("extract_surface_sfc_state()")
03409 end subroutine extract_surface_state
03410
03411 !> Rotate initialization fields from input to rotated arrays.
03412 subroutine rotate_initial_state(u_in, v_in, h_in, T_in, S_in, &
03413     use_temperature, turns, u, v, h, T, S)
03414   real, dimension(:,:,:), intent(in) :: u_in, v_in, h_in, T_in, S_in
03415   logical, intent(in) :: use_temperature
03416   integer, intent(in) :: turns
03417   real, dimension(:,:,:), intent(out) :: u, v, h, T, S
03418
03419   call rotate_vector(u_in, v_in, turns, u, v)
03420   call rotate_array(h_in, turns, h)
03421   if (use_temperature) then
```

```
03422     call rotate_array(t_in, turns, t)
03423     call rotate_array(s_in, turns, s)
03424   endif
03425 end subroutine rotate_initial_state
03426
03427 !> Return true if all phases of step_MOM are at the same point in time.
03428 function mom_state_is_synchronized(CS, adv_dyn) result(in_synch)
03429   type(mom_control_struct), pointer :: CS !< MOM control structure
03430   logical,        optional, intent(in) :: adv_dyn  !< If present and true, only check
03431                                           !! whether the advection is up-to-date with
03432                                           !! the dynamics.
03433   logical :: in_synch !< True if all phases of the update are synchronized.
03434
03435   logical :: adv_only
03436
03437   adv_only = .false. ; if (present(adv_dyn)) adv_only = adv_dyn
03438
03439   if (adv_only) then
03440     in_synch = (cs%t_dyn_rel_adv == 0.0)
03441   else
03442     in_synch = ((cs%t_dyn_rel_adv == 0.0) .and. (cs%t_dyn_rel_thermo == 0.0))
03443   endif
03444
03445 end function mom_state_is_synchronized
03446
03447 !> This subroutine offers access to values or pointers to other types from within
03448 !! the MOM_control_struct, allowing the MOM_control_struct to be opaque.
03449 subroutine get_mom_state_elements(CS, G, GV, US, C_p, C_p_scaled, use_temp)
03450   type(mom_control_struct),          pointer     :: CS !< MOM control structure
03451   type(ocean_grid_type),   optional, pointer     :: G    !< structure containing metrics and grid info
03452   type(verticalgrid_type), optional, pointer     :: GV   !< structure containing vertical grid info
03453   type(unit_scale_type),   optional, pointer     :: US   !< A dimensional unit scaling type
03454   real,                    optional, intent(out) :: C_p  !< The heat capacity [J kg degC-1]
03455   real,                    optional, intent(out) :: C_p_scaled !< The heat capacity in scaled
03456                                                                !! units [Q degC-1 ~> J kg degC-1]
03457   logical,                 optional, intent(out) :: use_temp !< True if temperature is a state variable
03458
03459   if (present(g)) g => cs%G_in
03460   if (present(gv)) gv => cs%GV
03461   if (present(us)) us => cs%US
03462   if (present(c_p)) c_p = cs%US%Q_to_J_kg * cs%tv%C_p
03463   if (present(c_p_scaled)) c_p_scaled = cs%tv%C_p
03464   if (present(use_temp)) use_temp = associated(cs%tv%T)
03465 end subroutine get_mom_state_elements
03466
03467 !> Find the global integrals of various quantities.
03468 subroutine get_ocean_stocks(CS, mass, heat, salt, on_PE_only)
03469   type(mom_control_struct), pointer :: CS !< MOM control structure
03470   real,    optional, intent(out) :: heat  !< The globally integrated integrated ocean heat [J].
03471   real,    optional, intent(out) :: salt  !< The globally integrated integrated ocean salt [kg].
03472   real,    optional, intent(out) :: mass  !< The globally integrated integrated ocean mass [kg].
03473   logical, optional, intent(in)  :: on_PE_only !< If present and true, only sum on the local PE.
03474
03475   if (present(mass)) &
03476     mass = global_mass_integral(cs%h, cs%G, cs%GV, on_pe_only=on_pe_only)
03477   if (present(heat)) &
03478     heat = cs%US%Q_to_J_kg*cs%tv%C_p * global_mass_integral(cs%h, cs%G, cs%GV, cs%tv%T, on_pe_only=
    on_pe_only)
03479   if (present(salt)) &
03480     salt = 1.0e-3 * global_mass_integral(cs%h, cs%G, cs%GV, cs%tv%S, on_pe_only=on_pe_only)
03481
03482 end subroutine get_ocean_stocks
03483
03484 !> End of ocean model, including memory deallocation
03485 subroutine mom_end(CS)
03486   type(mom_control_struct), pointer :: CS   !< MOM control structure
03487
03488   if (cs%use_ALE_algorithm) call ale_end(cs%ALE_CSp)
03489
03490   dealloc_(cs%u) ; dealloc_(cs%v) ; dealloc_(cs%h)
03491   dealloc_(cs%uh) ; dealloc_(cs%vh)
03492
03493   if (associated(cs%tv%T)) then
03494     dealloc_(cs%T) ; cs%tv%T => null() ; dealloc_(cs%S) ; cs%tv%S => null()
03495   endif
03496   if (associated(cs%tv%frazil)) deallocate(cs%tv%frazil)
03497   if (associated(cs%tv%salt_deficit)) deallocate(cs%tv%salt_deficit)
03498   if (associated(cs%Hml)) deallocate(cs%Hml)
03499
03500   call tracer_advect_end(cs%tracer_adv_CSp)
03501   call tracer_hor_diff_end(cs%tracer_diff_CSp)
03502   call tracer_registry_end(cs%tracer_Reg)
03503   call tracer_flow_control_end(cs%tracer_flow_CSp)
03504
03505   call diabatic_driver_end(cs%diabatic_CSp)
03506
03507   if (cs%offline_tracer_mode) call offline_transport_end(cs%offline_CSp)
```

```
03508
03509   dealloc_(cs%uhtr) ; dealloc_(cs%vhtr)
03510   if (cs%split) then
03511     call end_dyn_split_rk2(cs%dyn_split_RK2_CSp)
03512   elseif (cs%use_RK2) then
03513     call end_dyn_unsplit_rk2(cs%dyn_unsplit_RK2_CSp)
03514   else
03515     call end_dyn_unsplit(cs%dyn_unsplit_CSp)
03516   endif
03517   dealloc_(cs%ave_ssh_ibc) ; dealloc_(cs%ssh_rint)
03518   if (associated(cs%update_OBC_CSp)) call obc_register_end(cs%update_OBC_CSp)
03519
03520   call verticalgridend(cs%GV)
03521   call unit_scaling_end(cs%US)
03522   call mom_grid_end(cs%G)
03523
03524   deallocate(cs)
03525
03526 end subroutine mom_end
03527
03528 !> \namespace mom
03529 !!
03530 !! Modular Ocean Model (MOM) Version 6.0 (MOM6)
03531 !!
03532 !! \authors Alistair Adcroft, Robert Hallberg, and Stephen Griffies
03533 !!
03534 !!  Additional contributions from:
03535 !!    * Whit Anderson
03536 !!    * Brian Arbic
03537 !!    * Will Cooke
03538 !!    * Anand Gnanadesikan
03539 !!    * Matthew Harrison
03540 !!    * Mehmet Ilicak
03541 !!    * Laura Jackson
03542 !!    * Jasmine John
03543 !!    * John Krasting
03544 !!    * Zhi Liang
03545 !!    * Bonnie Samuels
03546 !!    * Harper Simmons
03547 !!    * Laurent White
03548 !!    * Niki Zadeh
03549 !!
03550 !!  MOM ice-shelf code was developed by
03551 !!  * Daniel Goldberg
03552 !!  * Robert Hallberg
03553 !!  * Chris Little
03554 !!  * Olga Sergienko
03555 !!
03556 !!  \section section_overview Overview of MOM
03557 !!
03558 !!  This program (MOM) simulates the ocean by numerically solving
03559 !!  the hydrostatic primitive equations in generalized Lagrangian
03560 !!  vertical coordinates, typically tracking stretched pressure (p*)
03561 !!  surfaces or following isopycnals in the ocean's interior, and
03562 !!  general orthogonal horizontal coordinates. Unlike earlier versions
03563 !!  of MOM, in MOM6 these equations are horizontally discretized on an
03564 !!  Arakawa C-grid.  (It remains to be seen whether a B-grid dynamic
03565 !!  core will be revived in MOM6 at a later date; for now applications
03566 !!  requiring a B-grid discretization should use MOM5.1.)  MOM6 offers
03567 !!  a range of options for the physical parameterizations, from those
03568 !!  most appropriate to highly idealized models for geophysical fluid
03569 !!  dynamics studies to a rich suite of processes appropriate for
03570 !!  realistic ocean simulations.  The thermodynamic options typically
03571 !!  use conservative temperature and preformed salinity as conservative
03572 !!  state variables and a full nonlinear equation of state, but there
03573 !!  are also idealized adiabatic configurations of the model that use
03574 !!  fixed density layers.  Version 6.0 of MOM continues in the long
03575 !!  tradition of a commitment to climate-quality ocean simulations
03576 !!  embodied in previous versions of MOM, even as it draws extensively
03577 !!  on the lessons learned in the development of the Generalized Ocean
03578 !!  Layered Dynamics (GOLD) ocean model, which was also primarily
03579 !!  developed at NOAA/GFDL.  MOM has also benefited tremendously from
03580 !!  the FMS infrastructure, which it utilizes and shares with other
03581 !!  component models developed at NOAA/GFDL.
03582 !!
03583 !!    When run is isopycnal-coordinate mode, the uppermost few layers
03584 !!  are often used to describe a bulk mixed layer, including the
03585 !!  effects of penetrating shortwave radiation.  Either a split-
03586 !!  explicit time stepping scheme or a non-split scheme may be used
03587 !!  for the dynamics, while the time stepping may be split (and use
03588 !!  different numbers of steps to cover the same interval) for the
03589 !!  forcing, the thermodynamics, and for the dynamics.  Most of the
03590 !!  numerics are second order accurate in space.  MOM can run with an
03591 !!  absurdly thin minimum layer thickness. A variety of non-isopycnal
03592 !!  vertical coordinate options are under development, but all exploit
03593 !!  the advantages of a Lagrangian vertical coordinate, as discussed
03594 !!  in detail by Adcroft and Hallberg (Ocean Modelling, 2006).
```

```
03595 !!
03596 !!    Details of the numerics and physical parameterizations are
03597 !!  provided in the appropriate source files.  All of the available
03598 !!  options are selected at run-time by parsing the input files,
03599 !!  usually MOM_input and MOM_override, and the options choices are
03600 !!  then documented for each run in MOM_param_docs.
03601 !!
03602 !!    MOM6 integrates the equations forward in time in three distinct
03603 !!  phases.  In one phase, the dynamic equations for the velocities
03604 !!  and layer thicknesses are advanced, capturing the propagation of
03605 !!  external and internal inertia-gravity waves, Rossby waves, and
03606 !!  other strictly adiabatic processes, including lateral stresses,
03607 !!  vertical viscosity and momentum forcing, and interface height
03608 !!  diffusion (commonly called Gent-McWilliams diffusion in depth-
03609 !!  coordinate models).  In the second phase, all tracers are advected
03610 !!  and diffused along the layers.  The third phase applies diabatic
03611 !!  processes, vertical mixing of water properties, and perhaps
03612 !!  vertical remapping to cause the layers to track the desired
03613 !!  vertical coordinate.
03614 !!
03615 !!    The present file (MOM.F90) orchestrates the main time stepping
03616 !!  loops. One time integration option for the dynamics uses a split
03617 !!  explicit time stepping scheme to rapidly step the barotropic
03618 !!  pressure and velocity fields. The barotropic velocities are
03619 !!  averaged over the baroclinic time step before they are used to
03620 !!  advect thickness and determine the baroclinic accelerations.  As
03621 !!  described in Hallberg and Adcroft (2009), a barotropic correction
03622 !!  is applied to the time-mean layer velocities to ensure that the
03623 !!  sum of the layer transports agrees with the time-mean barotropic
03624 !!  transport, thereby ensuring that the estimates of the free surface
03625 !!  from the sum of the layer thicknesses agrees with the final free
03626 !!  surface height as calculated by the barotropic solver.  The
03627 !!  barotropic and baroclinic velocities are kept consistent by
03628 !!  recalculating the barotropic velocities from the baroclinic
03629 !!  transports each time step. This scheme is described in Hallberg,
03630 !!  1997, J. Comp. Phys. 135, 54-65 and in Hallberg and Adcroft, 2009,
03631 !!  Ocean Modelling, 29, 15-26.
03632 !!
03633 !!    The other time integration options use non-split time stepping
03634 !!  schemes based on the 3-step third order Runge-Kutta scheme
03635 !!  described in Matsuno, 1966, J. Met. Soc. Japan, 44, 85-88, or on
03636 !!  a two-step quasi-2nd order Runge-Kutta scheme.  These are much
03637 !!  slower than the split time-stepping scheme, but they are useful
03638 !!  for providing a more robust solution for debugging cases where the
03639 !!  more complicated split time-stepping scheme may be giving suspect
03640 !!  solutions.
03641 !!
03642 !!    There are a range of closure options available.  Horizontal
03643 !!  velocities are subject to a combination of horizontal biharmonic
03644 !!  and Laplacian friction (based on a stress tensor formalism) and a
03645 !!  vertical Fickian viscosity (perhaps using the kinematic viscosity
03646 !!  of water).  The horizontal viscosities may be constant, spatially
03647 !!  varying or may be dynamically calculated using Smagorinsky's
03648 !!  approach.  A diapycnal diffusion of density and thermodynamic
03649 !!  quantities is also allowed, but not required, as is horizontal
03650 !!  diffusion of interface heights (akin to the Gent-McWilliams
03651 !!  closure of geopotential coordinate models).  The diapycnal mixing
03652 !!  may use a fixed diffusivity or it may use the shear Richardson
03653 !!  number dependent closure, like that described in Jackson et al.
03654 !!  (JPO, 2008).  When there is diapycnal diffusion, it applies to
03655 !!  momentum as well. As this is in addition to the vertical viscosity,
03656 !!  the vertical Prandtl always exceeds 1.  A refined bulk-mixed layer
03657 !!  is often used to describe the planetary boundary layer in realistic
03658 !!  ocean simulations.
03659 !!
03660 !!    MOM has a number of noteworthy debugging capabilities.
03661 !!  Excessively large velocities are truncated and MOM will stop
03662 !!  itself after a number of such instances to keep the model from
03663 !!  crashing altogether.  This is useful in diagnosing failures,
03664 !!  or (by accepting some truncations) it may be useful for getting
03665 !!  the model past the adjustment from an ill-balanced initial
03666 !!  condition.  In addition, all of the accelerations in the columns
03667 !!  with excessively large velocities may be directed to a text file.
03668 !!  Parallelization errors may be diagnosed using the DEBUG option,
03669 !!  which causes extensive checksums to be written out along with
03670 !!  comments indicating where in the algorithm the sums originate and
03671 !!  what variable is being summed.  The point where these checksums
03672 !!  differ between runs is usually a good indication of where in the
03673 !!  code the problem lies.  All of the test cases provided with MOM
03674 !!  are routinely tested to ensure that they give bitwise identical
03675 !!  results regardless of the domain decomposition, or whether they
03676 !!  use static or dynamic memory allocation.
03677 !!
03678 !!  \section section_structure Structure of MOM
03679 !!
03680 !!  About 115 other files of source code and 4 header files comprise
03681 !!  the MOM code, although there are several hundred more files that
```

```
03682 !!  make up the FMS infrastructure upon which MOM is built.  Each of
03683 !!  the MOM files contains comments documenting what it does, and
03684 !!  most of the file names are fairly self-evident. In addition, all
03685 !!  subroutines and data types are referenced via a module use, only
03686 !!  statement, and the module names are consistent with the file names,
03687 !!  so it is not too hard to find the source file for a subroutine.
03688 !!
03689 !!    The typical MOM directory tree is as follows:
03690 !!
03691 !! \verbatim
03692 !!        ../MOM
03693 !!        |-- config_src
03694 !!        |   |-- coupled_driver
03695 !!        |   |-- dynamic
03696 !!        |   `-- solo_driver
03697 !!        |-- examples
03698 !!        |   |-- CM2G
03699 !!        |   |-- ...
03700 !!        |   `-- torus_advection_test
03701 !!        `-- src
03702 !!            |-- core
03703 !!            |-- diagnostics
03704 !!            |-- equation_of_state
03705 !!            |-- framework
03706 !!            |-- ice_shelf
03707 !!            |-- initialization
03708 !!            |-- parameterizations
03709 !!            |   |-- lateral
03710 !!            |   `-- vertical
03711 !!            |-- tracer
03712 !!            `-- user
03713 !! \endverbatim
03714 !!
03715 !!  Rather than describing each file here, each directory contents
03716 !!  will be described to give a broad overview of the MOM code
03717 !!  structure.
03718 !!
03719 !!    The directories under config_src contain files that are used for
03720 !!  configuring the code, for instance for coupled or ocean-only runs.
03721 !!  Only one or two of these directories are used in compiling any,
03722 !!  particular run.
03723 !!
03724 !!  * config_src/coupled_driver:
03725 !!    The files here are used to couple MOM as a component in a larger
03726 !!    run driven by the FMS coupler.  This includes code that converts
03727 !!    various forcing fields into the code structures and flux and unit
03728 !!    conventions used by MOM, and converts the MOM surface fields
03729 !!    back to the forms used by other FMS components.
03730 !!
03731 !!  * config_src/dynamic:
03732 !!    The only file here is the version of MOM_memory.h that is used
03733 !!    for dynamic memory configurations of MOM.
03734 !!
03735 !!  * config_src/solo_driver:
03736 !!    The files here are include the _main driver that is used when
03737 !!    MOM is configured as an ocean-only model, as well as the files
03738 !!    that specify the surface forcing in this configuration.
03739 !!
03740 !!    The directories under examples provide a large number of working
03741 !!  configurations of MOM, along with reference solutions for several
03742 !!  different compilers on GFDL's latest large computer.  The versions
03743 !!  of MOM_memory.h in these directories need not be used if dynamic
03744 !!  memory allocation is desired, and the answers should be unchanged.
03745 !!
03746 !!    The directories under src contain most of the MOM files.  These
03747 !!  files are used in every configuration using MOM.
03748 !!
03749 !!  * src/core:
03750 !!    The files here constitute the MOM dynamic core.  This directory
03751 !!    also includes files with the types that describe the model's
03752 !!    lateral grid and have defined types that are shared across
03753 !!    various MOM modules to allow for more succinct and flexible
03754 !!    subroutine argument lists.
03755 !!
03756 !!  * src/diagnostics:
03757 !!    The files here calculate various diagnostics that are anciliary
03758 !!    to the model itself.  While most of these diagnostics do not
03759 !!    directly affect the model's solution, there are some, like the
03760 !!    calculation of the deformation radius, that are used in some
03761 !!    of the process parameterizations.
03762 !!
03763 !!  * src/equation_of_state:
03764 !!    These files describe the physical properties of sea-water,
03765 !!    including both the equation of state and when it freezes.
03766 !!
03767 !!  * src/framework:
03768 !!    These files provide infrastructure utilities for MOM.  Many are
```

```
03769 !!    simply wrappers for capabilities provided by FMS, although others
03770 !!    provide capabilities (like the file_parser) that are unique to
03771 !!    MOM. When MOM is adapted to use a modeling infrastructure
03772 !!    distinct from FMS, most of the required changes are in this
03773 !!    directory.
03774 !!
03775 !!  * src/initialization:
03776 !!    These are the files that are used to initialize the MOM grid
03777 !!    or provide the initial physical state for MOM.  These files are
03778 !!    not intended to be modified, but provide a means for calling
03779 !!    user-specific initialization code like the examples in src/user.
03780 !!
03781 !!  * src/parameterizations/lateral:
03782 !!    These files implement a number of quasi-lateral (along-layer)
03783 !!    process parameterizations, including lateral viscosities,
03784 !!    parameterizations of eddy effects, and the calculation of tidal
03785 !!    forcing.
03786 !!
03787 !!  * src/parameterizations/vertical:
03788 !!    These files implement a number of vertical mixing or diabatic
03789 !!    processes, including the effects of vertical viscosity and
03790 !!    code to parameterize the planetary boundary layer.  There is a
03791 !!    separate driver that orchestrates this portion of the algorithm,
03792 !!    and there is a diversity of parameterizations to be found here.
03793 !!
03794 !!  * src/tracer:
03795 !!    These files handle the lateral transport and diffusion of
03796 !!    tracers, or are the code to implement various passive tracer
03797 !!    packages.  Additional tracer packages are readily accommodated.
03798 !!
03799 !!  * src/user:
03800 !!    These are either stub routines that a user could use to change
03801 !!    the model's initial conditions or forcing, or are examples that
03802 !!    implement specific test cases.  These files can easily  be hand
03803 !!    edited to create new analytically specified configurations.
03804 !!
03805 !!
03806 !!  Most simulations can be set up by modifying only the files
03807 !!  MOM_input, and possibly one or two of the files in src/user.
03808 !!  In addition, the diag_table (MOM_diag_table) will commonly be
03809 !!  modified to tailor the output to the needs of the question at
03810 !!  hand.  The FMS utility mkmf works with a file called path_names
03811 !!  to build an appropriate makefile, and path_names should be edited
03812 !!  to reflect the actual location of the desired source code.
03813 !!
03814 !!
03815 !!  There are 3 publicly visible subroutines in this file (MOM.F90).
03816 !!  * step_MOM steps MOM over a specified interval of time.
03817 !!  * MOM_initialize calls initialize and does other initialization
03818 !!    that does not warrant user modification.
03819 !!  * extract_surface_state determines the surface (bulk mixed layer
03820 !!    if traditional isoycnal vertical coordinate) properties of the
03821 !!    current model state and packages pointers to these fields into an
03822 !!    exported structure.
03823 !!
03824 !!    The remaining subroutines in this file (src/core/MOM.F90) are:
03825 !!  * find_total_transport determines the barotropic mass transport.
03826 !!  * register_diags registers many diagnostic fields for the dynamic
03827 !!    solver, or of the main model variables.
03828 !!  * MOM_timing_init initializes various CPU time clocks.
03829 !!  * write_static_fields writes out various time-invariant fields.
03830 !!  * set_restart_fields is used to specify those fields that are
03831 !!    written to and read from the restart file.
03832 !!
03833 !!  \section section_heat_budget Diagnosing MOM heat budget
03834 !!
03835 !!  Here are some example heat budgets for the ALE version of MOM6.
03836 !!
03837 !!  \subsection subsection_2d_heat_budget Depth integrated heat budget
03838 !!
03839 !!  Depth integrated heat budget diagnostic for MOM.
03840 !!
03841 !! * OPOTTEMPTEND_2d = T_ADVECTION_XY_2d + OPOTTEMPPMDIFF_2d + HFDS + HFGEOU
03842 !!
03843 !! * T_ADVECTION_XY_2d = horizontal advection
03844 !! * OPOTTEMPPMDIFF_2d = neutral diffusion
03845 !! * HFDS              = net surface boundary heat flux
03846 !! * HFGEOU            = geothermal heat flux
03847 !!
03848 !! * HFDS = net surface boundary heat flux entering the ocean
03849 !!        = rsntds + rlntds + hfls + hfss + heat_pme + hfsifrazil
03850 !!
03851 !! * More heat flux cross-checks
03852 !!   * hfds     = net_heat_coupler + hfsifrazil + heat_pme
03853 !!   * heat_pme = heat_content_surfwater
03854 !!              = heat_content_massin + heat_content_massout
03855 !!              = heat_content_fprec + heat_content_cond + heat_content_vprec
```

```
03856 !!              + hfrunoffds + hfevapds + hfrainds
03857 !!
03858 !!  \subsection subsection_3d_heat_budget Depth integrated heat budget
03859 !!
03860 !!  Here is an example 3d heat budget diagnostic for MOM.
03861 !!
03862 !! * OPOTTEMPTEND = T_ADVECTION_XY + TH_TENDENCY_VERT_REMAP + OPOTTEMPDIFF + OPOTTEMPPMDIFF
03863 !!                + BOUNDARY_FORCING_HEAT_TENDENCY + FRAZIL_HEAT_TENDENCY
03864 !!
03865 !! * OPOTTEMPTEND                   = net tendency of heat as diagnosed in MOM.F90
03866 !! * T_ADVECTION_XY                 = heating of a cell from lateral advection
03867 !! * TH_TENDENCY_VERT_REMAP         = heating of a cell from vertical remapping
03868 !! * OPOTTEMPDIFF                   = heating of a cell from diabatic diffusion
03869 !! * OPOTTEMPPMDIFF                 = heating of a cell from neutral diffusion
03870 !! * BOUNDARY_FORCING_HEAT_TENDENCY = heating of cell from boundary fluxes
03871 !! * FRAZIL_HEAT_TENDENCY           = heating of cell from frazil
03872 !!
03873 !! * TH_TENDENCY_VERT_REMAP has zero vertical sum, as it redistributes heat in vertical.
03874 !!
03875 !! * OPOTTEMPDIFF has zero vertical sum, as it redistributes heat in the vertical.
03876 !!
03877 !! * BOUNDARY_FORCING_HEAT_TENDENCY generally has 3d structure, with k > 1 contributions from
03878 !!   penetrative shortwave, and from other fluxes for the case when layers are tiny, in which
03879 !!   case MOM6 partitions tendencies into k > 1 layers.
03880 !!
03881 !! * FRAZIL_HEAT_TENDENCY generally has 3d structure, since MOM6 frazil calculation checks the
03882 !!   full ocean column.
03883 !!
03884 !! * FRAZIL_HEAT_TENDENCY[k=\@sum] = HFSIFRAZIL = column integrated frazil heating.
03885 !!
03886 !! * HFDS = FRAZIL_HEAT_TENDENCY[k=\@sum] + BOUNDARY_FORCING_HEAT_TENDENCY[k=\@sum]
03887 !!
03888 !!  Here is an example 2d heat budget (depth summed) diagnostic for MOM.
03889 !!
03890 !! * OPOTTEMPTEND_2d = T_ADVECTION_XY_2d + OPOTTEMPPMDIFF_2d + HFDS
03891 !!
03892 !!
03893 !!  Here is an example 3d salt budget diagnostic for MOM.
03894 !!
03895 !! * OSALTTEND = S_ADVECTION_XY + SH_TENDENCY_VERT_REMAP + OSALTDIFF + OSALTPMDIFF
03896 !!              + BOUNDARY_FORCING_SALT_TENDENCY
03897 !!
03898 !! * OSALTTEND                      = net tendency of salt as diagnosed in MOM.F90
03899 !! * S_ADVECTION_XY                 = salt convergence to cell from lateral advection
03900 !! * SH_TENDENCY_VERT_REMAP         = salt convergence to cell from vertical remapping
03901 !! * OSALTDIFF                      = salt convergence to cell from diabatic diffusion
03902 !! * OSALTPMDIFF                    = salt convergence to cell from neutral diffusion
03903 !! * BOUNDARY_FORCING_SALT_TENDENCY = salt convergence to cell from boundary fluxes
03904 !!
03905 !! * SH_TENDENCY_VERT_REMAP has zero vertical sum, as it redistributes salt in vertical.
03906 !!
03907 !! * OSALTDIFF has zero vertical sum, as it redistributes salt in the vertical.
03908 !!
03909 !! * BOUNDARY_FORCING_SALT_TENDENCY generally has 3d structure, with k > 1 contributions from
03910 !!   the case when layers are tiny, in which case MOM6 partitions tendencies into k > 1 layers.
03911 !!
03912 !! * SFDSI = BOUNDARY_FORCING_SALT_TENDENCY[k=\@sum]
03913 !!
03914 !!  Here is an example 2d salt budget (depth summed) diagnostic for MOM.
03915 !!
03916 !! * OSALTTEND_2d = S_ADVECTION_XY_2d + OSALTPMDIFF_2d + SFDSI (+ SALT_FLUX_RESTORE)
03917 !!
03918 !!
03919 !!
03920 end module mom
```

44.24 /home/cermak/src/MOM6/src/equation_of_state/_Equation_of_State.dox File Reference

44.25 /home/cermak/src/MOM6/src/equation_of_state/MOM_EOS.F90 File Reference

```
#include <MOM_memory.h>
#include "version_variable.h"
```

Data Types

- interface mom_eos::calculate_density

  Calculates density of sea water from T, S and P.
- interface mom_eos::calculate_spec_vol

  Calculates specific volume of sea water from T, S and P.
- interface mom_eos::calculate_density_derivs

  Calculate the derivatives of density with temperature and salinity from T, S, and P.
- interface mom_eos::calculate_specific_vol_derivs

  Calculate the derivatives of specific volume with temperature and salinity from T, S, and P.
- interface mom_eos::calculate_density_second_derivs

  Calculates the second derivatives of density with various combinations of temperature, salinity, and pressure from T, S and P.
- interface mom_eos::calculate_tfreeze

  Calculates the freezing point of sea water from T, S and P.
- interface mom_eos::calculate_compress

  Calculates the compressibility of water from T, S, and P.
- type mom_eos::eos_type

  A control structure for the equation of state.

Modules

- module mom_eos

  Provides subroutines for quantities specific to the equation of state.

Functions/Subroutines

- subroutine mom_eos::calculate_density_scalar (T, S, pressure, rho, EOS, rho_ref, scale)

  Calls the appropriate subroutine to calculate density of sea water for scalar inputs. If rho_ref is present, the anomaly with respect to rho_ref is returned. The pressure and density can be rescaled with the US. If both the US and scale arguments are present the density scaling uses the product of the two scaling factors.
- subroutine mom_eos::calculate_stanley_density_scalar (T, S, pressure, Tvar, TScov, Svar, rho, EOS, rho_ref, scale)

  Calls the appropriate subroutine to calculate density of sea water for scalar inputs including the variance of T, S and covariance of T-S. The calculation uses only the second order correction in a series as discussed in Stanley et al., 2020. If rho_ref is present, the anomaly with respect to rho_ref is returned. The density can be rescaled using rho_ref.
- subroutine mom_eos::calculate_density_array (T, S, pressure, rho, start, npts, EOS, rho_ref, scale)

  Calls the appropriate subroutine to calculate the density of sea water for 1-D array inputs. If rho_ref is present, the anomaly with respect to rho_ref is returned.
- subroutine mom_eos::calculate_stanley_density_array (T, S, pressure, Tvar, TScov, Svar, rho, start, npts, EOS, rho_ref, scale)

  Calls the appropriate subroutine to calculate the density of sea water for 1-D array inputs including the variance of T, S and covariance of T-S. The calculation uses only the second order correction in a series as discussed in Stanley et al., 2020. If rho_ref is present, the anomaly with respect to rho_ref is returned.
- subroutine mom_eos::calculate_density_1d (T, S, pressure, rho, EOS, dom, rho_ref, scale)

  Calls the appropriate subroutine to calculate the density of sea water for 1-D array inputs, potentially limiting the domain of indices that are worked on. If rho_ref is present, the anomaly with respect to rho_ref is returned.
- subroutine mom_eos::calculate_stanley_density_1d (T, S, pressure, Tvar, TScov, Svar, rho, EOS, dom, rho↩ _ref, scale)

  Calls the appropriate subroutine to calculate the density of sea water for 1-D array inputs including the variance of T, S and covariance of T-S, potentially limiting the domain of indices that are worked on. The calculation uses only the second order correction in a series as discussed in Stanley et al., 2020. If rho_ref is present, the anomaly with respect to rho_ref is returned.

- subroutine mom_eos::calculate_spec_vol_array (T, S, pressure, specvol, start, npts, EOS, spv_ref, scale)

  Calls the appropriate subroutine to calculate the specific volume of sea water for 1-D array inputs.
- subroutine mom_eos::calc_spec_vol_scalar (T, S, pressure, specvol, EOS, spv_ref, scale)

  Calls the appropriate subroutine to calculate specific volume of sea water for scalar inputs.
- subroutine mom_eos::calc_spec_vol_1d (T, S, pressure, specvol, EOS, dom, spv_ref, scale)

  Calls the appropriate subroutine to calculate the specific volume of sea water for 1-D array inputs, potentially limiting the domain of indices that are worked on.
- subroutine mom_eos::calculate_tfreeze_scalar (S, pressure, T_fr, EOS, pres_scale)

  Calls the appropriate subroutine to calculate the freezing point for scalar inputs.
- subroutine mom_eos::calculate_tfreeze_array (S, pressure, T_fr, start, npts, EOS, pres_scale)

  Calls the appropriate subroutine to calculate the freezing point for a 1-D array.
- subroutine mom_eos::calculate_density_derivs_array (T, S, pressure, drho_dT, drho_dS, start, npts, EOS, scale)

  Calls the appropriate subroutine to calculate density derivatives for 1-D array inputs.
- subroutine mom_eos::calculate_density_derivs_1d (T, S, pressure, drho_dT, drho_dS, EOS, dom, scale)

  Calls the appropriate subroutine to calculate density derivatives for 1-D array inputs.
- subroutine mom_eos::calculate_density_derivs_scalar (T, S, pressure, drho_dT, drho_dS, EOS, scale)

  Calls the appropriate subroutines to calculate density derivatives by promoting a scalar to a one-element array.
- subroutine mom_eos::calculate_density_second_derivs_array (T, S, pressure, drho_dS_dS, drho_dS_dT, drho_dT_dT, drho_dS_dP, drho_dT_dP, start, npts, EOS, scale)

  Calls the appropriate subroutine to calculate density second derivatives for 1-D array inputs.
- subroutine mom_eos::calculate_density_second_derivs_scalar (T, S, pressure, drho_dS_dS, drho_dS_dT, drho_dT_dT, drho_dS_dP, drho_dT_dP, EOS, scale)

  Calls the appropriate subroutine to calculate density second derivatives for scalar nputs.
- subroutine mom_eos::calculate_spec_vol_derivs_array (T, S, pressure, dSV_dT, dSV_dS, start, npts, EOS)

  Calls the appropriate subroutine to calculate specific volume derivatives for an array.
- subroutine mom_eos::calc_spec_vol_derivs_1d (T, S, pressure, dSV_dT, dSV_dS, EOS, dom, scale)

  Calls the appropriate subroutine to calculate specific volume derivatives for 1-d array inputs, potentially limiting the domain of indices that are worked on.
- subroutine mom_eos::calculate_compress_array (T, S, press, rho, drho_dp, start, npts, EOS)

  Calls the appropriate subroutine to calculate the density and compressibility for 1-D array inputs. If US is present, the units of the inputs and outputs are rescaled.
- subroutine mom_eos::calculate_compress_scalar (T, S, pressure, rho, drho_dp, EOS)

  Calculate density and compressibility for a scalar. This just promotes the scalar to an array with a singleton dimension and calls calculate_compress_array. If US is present, the units of the inputs and outputs are rescaled.
- integer function, dimension(2), public mom_eos::eos_domain (HI, halo)

  This subroutine returns a two point integer array indicating the domain of i-indices to work on in EOS calls based on information from a hor_index type.
- subroutine, public mom_eos::analytic_int_specific_vol_dp (T, S, p_t, p_b, alpha_ref, HI, EOS, dza, intp_dza, intx_dza, inty_dza, halo_size, bathyP, dP_tiny, useMassWghtInterp)

  Calls the appropriate subroutine to calculate analytical and nearly-analytical integrals in pressure across layers of geopotential anomalies, which are required for calculating the finite-volume form pressure accelerations in a non-Boussinesq model. There are essentially no free assumptions, apart from the use of Boole's rule to do the horizontal integrals, and from a truncation in the series for log(1-eps/1+eps) that assumes that $|eps| < 0.34$.
- subroutine, public mom_eos::analytic_int_density_dz (T, S, z_t, z_b, rho_ref, rho_0, G_e, HI, EOS, dpa, intz_dpa, intx_dpa, inty_dpa, bathyT, dz_neglect, useMassWghtInterp)

  This subroutine calculates analytical and nearly-analytical integrals of pressure anomalies across layers, which are required for calculating the finite-volume form pressure accelerations in a Boussinesq model.
- logical function, public mom_eos::query_compressible (EOS)

  Returns true if the equation of state is compressible (i.e. has pressure dependence)
- subroutine, public mom_eos::eos_init (param_file, EOS, US)

  Initializes EOS_type by allocating and reading parameters.

- subroutine, public mom_eos::eos_manual_init (EOS, form_of_EOS, form_of_TFreeze, EOS_quadrature, Compressible, Rho_T0_S0, drho_dT, dRho_dS, TFr_S0_P0, dTFr_dS, dTFr_dp)

  Manually initialized an EOS type (intended for unit testing of routines which need a specific EOS)
- subroutine, public mom_eos::eos_allocate (EOS)

  Allocates EOS_type.
- subroutine, public mom_eos::eos_end (EOS)

  Deallocates EOS_type.
- subroutine, public mom_eos::eos_use_linear (Rho_T0_S0, dRho_dT, dRho_dS, EOS, use_quadrature)

  Set equation of state structure (EOS) to linear with given coefficients.
- subroutine, public mom_eos::convert_temp_salt_for_teos10 (T, S, HI, kd, mask_z, EOS)

  Convert T&S to Absolute Salinity and Conservative Temperature if using TEOS10.
- logical function, public mom_eos::eos_quadrature (EOS)

  Return value of EOS_quadrature.
- subroutine, public mom_eos::extract_member_eos (EOS, form_of_EOS, form_of_TFreeze, EOS_quadrature, Compressible, Rho_T0_S0, drho_dT, dRho_dS, TFr_S0_P0, dTFr_dS, dTFr_dp)

  Extractor routine for the EOS type if the members need to be accessed outside this module.

Variables

- integer, parameter, public mom_eos::eos_linear = 1

  A named integer specifying an equation of state.
- integer, parameter, public mom_eos::eos_unesco = 2

  A named integer specifying an equation of state.
- integer, parameter, public mom_eos::eos_wright = 3

  A named integer specifying an equation of state.
- integer, parameter, public mom_eos::eos_teos10 = 4

  A named integer specifying an equation of state.
- integer, parameter, public mom_eos::eos_nemo = 5

  A named integer specifying an equation of state.
- character ∗(10), parameter mom_eos::eos_linear_string = "LINEAR"

  A string for specifying the equation of state.
- character ∗(10), parameter mom_eos::eos_unesco_string = "UNESCO"

  A string for specifying the equation of state.
- character ∗(10), parameter mom_eos::eos_wright_string = "WRIGHT"

  A string for specifying the equation of state.
- character ∗(10), parameter mom_eos::eos_teos10_string = "TEOS10"

  A string for specifying the equation of state.
- character ∗(10), parameter mom_eos::eos_nemo_string = "NEMO"

  A string for specifying the equation of state.
- character ∗(10), parameter mom_eos::eos_default = EOS_WRIGHT_STRING

  The default equation of state.
- integer, parameter mom_eos::tfreeze_linear = 1

  A named integer specifying a freezing point expression.
- integer, parameter mom_eos::tfreeze_millero = 2

  A named integer specifying a freezing point expression.
- integer, parameter mom_eos::tfreeze_teos10 = 3

  A named integer specifying a freezing point expression.
- character ∗(10), parameter mom_eos::tfreeze_linear_string = "LINEAR"

  A string for specifying the freezing point expression.
- character ∗(10), parameter mom_eos::tfreeze_millero_string = "MILLERO_78"

A string for specifying freezing point expression.

- character ∗(10), parameter mom_eos::tfreeze_teos10_string = "TEOS10"

  A string for specifying the freezing point expression.

- character ∗(10), parameter mom_eos::tfreeze_default = TFREEZE_LINEAR_STRING

  The default freezing point expression.

44.26 MOM_EOS.F90

```
00001 !> Provides subroutines for quantities specific to the equation of state
00002 module mom_eos
00003
00004 ! This file is part of MOM6. See LICENSE.md for the license.
00005
00006 use mom_eos_linear, only : calculate_density_linear, calculate_spec_vol_linear
00007 use mom_eos_linear, only : calculate_density_derivs_linear
00008 use mom_eos_linear, only : calculate_specvol_derivs_linear, int_density_dz_linear
00009 use mom_eos_linear, only : calculate_density_second_derivs_linear
00010 use mom_eos_linear, only : calculate_compress_linear, int_spec_vol_dp_linear
00011 use mom_eos_wright, only : calculate_density_wright, calculate_spec_vol_wright
00012 use mom_eos_wright, only : calculate_density_derivs_wright
00013 use mom_eos_wright, only : calculate_specvol_derivs_wright, int_density_dz_wright
00014 use mom_eos_wright, only : calculate_compress_wright, int_spec_vol_dp_wright
00015 use mom_eos_wright, only : calculate_density_second_derivs_wright
00016 use mom_eos_unesco, only : calculate_density_unesco, calculate_spec_vol_unesco
00017 use mom_eos_unesco, only : calculate_density_derivs_unesco, calculate_density_unesco
00018 use mom_eos_unesco, only : calculate_compress_unesco
00019 use mom_eos_nemo,   only : calculate_density_nemo
00020 use mom_eos_nemo,   only : calculate_density_derivs_nemo, calculate_density_nemo
00021 use mom_eos_nemo,   only : calculate_compress_nemo
00022 use mom_eos_teos10, only : calculate_density_teos10, calculate_spec_vol_teos10
00023 use mom_eos_teos10, only : calculate_density_derivs_teos10
00024 use mom_eos_teos10, only : calculate_specvol_derivs_teos10
00025 use mom_eos_teos10, only : calculate_density_second_derivs_teos10
00026 use mom_eos_teos10, only : calculate_compress_teos10
00027 use mom_eos_teos10, only : gsw_sp_from_sr, gsw_pt_from_ct
00028 use mom_tfreeze,    only : calculate_tfreeze_linear, calculate_tfreeze_millero
00029 use mom_tfreeze,    only : calculate_tfreeze_teos10
00030 use mom_error_handler, only : mom_error, fatal, warning, mom_mesg
00031 use mom_file_parser, only : get_param, log_version, param_file_type
00032 use mom_hor_index,   only : hor_index_type
00033 use mom_string_functions, only : uppercase
00034 use mom_unit_scaling, only : unit_scale_type
00035
00036 implicit none ; private
00037
00038 #include <MOM_memory.h>
00039
00040 public eos_allocate
00041 public eos_domain
00042 public eos_end
00043 public eos_init
00044 public eos_manual_init
00045 public eos_quadrature
00046 public eos_use_linear
00047 public analytic_int_density_dz
00048 public analytic_int_specific_vol_dp
00049 public calculate_compress
00050 public calculate_density
00051 public calculate_density_derivs
00052 public calculate_density_second_derivs
00053 public calculate_spec_vol
00054 public calculate_specific_vol_derivs
00055 public calculate_tfreeze
00056 public convert_temp_salt_for_teos10
00057 public extract_member_eos
00058 public gsw_sp_from_sr
00059 public gsw_pt_from_ct
00060 public query_compressible
00061
00062 ! A note on unit descriptions in comments: MOM6 uses units that can be rescaled for dimensional
00063 ! consistency testing. These are noted in comments with units like Z, H, L, and T, along with
00064 ! their mks counterparts with notation like "a velocity [Z T-1 ~> m s-1]".  If the units
00065 ! vary with the Boussinesq approximation, the Boussinesq variant is given first.
00066
00067 !> Calculates density of sea water from T, S and P
00068 interface calculate_density
00069   module procedure calculate_density_scalar,
      calculate_density_array, calculate_density_1d
00070   module procedure calculate_stanley_density_scalar,
```

```
      calculate_stanley_density_array
00071   module procedure calculate_stanley_density_1d
00072 end interface calculate_density
00073
00074 !> Calculates specific volume of sea water from T, S and P
00075 interface calculate_spec_vol
00076   module procedure calc_spec_vol_scalar,
      calculate_spec_vol_array, &
00077                    calc_spec_vol_1d
00078 end interface calculate_spec_vol
00079
00080 !> Calculate the derivatives of density with temperature and salinity from T, S, and P
00081 interface calculate_density_derivs
00082   module procedure calculate_density_derivs_scalar,
      calculate_density_derivs_array, &
00083                    calculate_density_derivs_1d
00084 end interface calculate_density_derivs
00085
00086 !> Calculate the derivatives of specific volume with temperature and salinity from T, S, and P
00087 interface calculate_specific_vol_derivs
00088   module procedure calculate_spec_vol_derivs_array,
      calc_spec_vol_derivs_1d
00089 end interface calculate_specific_vol_derivs
00090
00091 !> Calculates the second derivatives of density with various combinations of temperature,
00092 !! salinity, and pressure from T, S and P
00093 interface calculate_density_second_derivs
00094   module procedure calculate_density_second_derivs_scalar,
      calculate_density_second_derivs_array
00095 end interface calculate_density_second_derivs
00096
00097 !> Calculates the freezing point of sea water from T, S and P
00098 interface calculate_tfreeze
00099   module procedure calculate_tfreeze_scalar,
      calculate_tfreeze_array
00100 end interface calculate_tfreeze
00101
00102 !> Calculates the compressibility of water from T, S, and P
00103 interface calculate_compress
00104   module procedure calculate_compress_scalar,
      calculate_compress_array
00105 end interface calculate_compress
00106
00107 !> A control structure for the equation of state
00108 type, public :: eos_type ; private
00109   integer :: form_of_eos = 0 !< The equation of state to use.
00110   integer :: form_of_tfreeze = 0 !< The expression for the potential temperature
00111                              !! of the freezing point.
00112   logical :: eos_quadrature  !< If true, always use the generic (quadrature)
00113                              !! code for the integrals of density.
00114   logical :: compressible = .true. !< If true, in situ density is a function of pressure.
00115 ! The following parameters are used with the linear equation of state only.
00116   real :: rho_t0_s0 !< The density at T=0, S=0 [kg m-3]
00117   real :: drho_dt   !< The partial derivative of density with temperature [kg m-3 degC-1]
00118   real :: drho_ds   !< The partial derivative of density with salinity [kg m-3 ppt-1]
00119 ! The following parameters are use with the linear expression for the freezing
00120 ! point only.
00121   real :: tfr_s0_p0 !< The freezing potential temperature at S=0, P=0 [degC]
00122   real :: dtfr_ds   !< The derivative of freezing point with salinity [degC ppt-1]
00123   real :: dtfr_dp   !< The derivative of freezing point with pressure [degC Pa-1]
00124
00125 ! Unit conversion factors (normally used for dimensional testing but could also allow for
00126 ! change of units of arguments to functions)
00127   real :: m_to_z = 1.      !< A constant that translates distances in meters to the units of depth.
00128   real :: kg_m3_to_r = 1.  !< A constant that translates kilograms per meter cubed to the units of density.
00129   real :: r_to_kg_m3 = 1.  !< A constant that translates the units of density to kilograms per meter cubed.
00130   real :: rl2_t2_to_pa = 1.!< Convert pressures from R L2 T-2 to Pa.
00131   real :: l_t_to_m_s = 1.  !< Convert lateral velocities from L T-1 to m s-1.
00132
00133 !  logical :: test_EOS = .true. ! If true, test the equation of state
00134 end type eos_type
00135
00136 ! The named integers that might be stored in eqn_of_state_type%form_of_EOS.
00137 integer, parameter, public :: eos_linear = 1 !< A named integer specifying an equation of state
00138 integer, parameter, public :: eos_unesco = 2 !< A named integer specifying an equation of state
00139 integer, parameter, public :: eos_wright = 3 !< A named integer specifying an equation of state
00140 integer, parameter, public :: eos_teos10 = 4 !< A named integer specifying an equation of state
00141 integer, parameter, public :: eos_nemo   = 5 !< A named integer specifying an equation of state
00142
00143 character*(10), parameter :: eos_linear_string = "LINEAR" !< A string for specifying the
       equation of state
00144 character*(10), parameter :: eos_unesco_string = "UNESCO" !< A string for specifying the
       equation of state
00145 character*(10), parameter :: eos_wright_string = "WRIGHT" !< A string for specifying the
       equation of state
00146 character*(10), parameter :: eos_teos10_string = "TEOS10" !< A string for specifying the
       equation of state
```

```
00147 character*(10), parameter :: eos_nemo_string   = "NEMO"   !< A string for specifying the
       equation of state
00148 character*(10), parameter :: eos_default = eos_wright_string !< The default
       equation of state
00149
00150 integer, parameter :: tfreeze_linear = 1  !< A named integer specifying a freezing point
       expression
00151 integer, parameter :: tfreeze_millero = 2 !< A named integer specifying a freezing point
       expression
00152 integer, parameter :: tfreeze_teos10 = 3  !< A named integer specifying a freezing point
       expression
00153 character*(10), parameter :: tfreeze_linear_string = "LINEAR" !< A string for
       specifying the freezing point expression
00154 character*(10), parameter :: tfreeze_millero_string = "MILLERO_78" !< A string for
       specifying
00155                                                              !! freezing point expression
00156 character*(10), parameter :: tfreeze_teos10_string = "TEOS10" !< A string for
       specifying the freezing point expression
00157 character*(10), parameter :: tfreeze_default =
      tfreeze_linear_string !< The default freezing point expression
00158
00159 contains
00160
00161 !> Calls the appropriate subroutine to calculate density of sea water for scalar inputs.
00162 !! If rho_ref is present, the anomaly with respect to rho_ref is returned.  The pressure and
00163 !! density can be rescaled with the US.  If both the US and scale arguments are present the density
00164 !! scaling uses the product of the two scaling factors.
00165 subroutine calculate_density_scalar(T, S, pressure, rho, EOS, rho_ref, scale)
00166   real,           intent(in)  :: T        !< Potential temperature referenced to the surface [degC]
00167   real,           intent(in)  :: S        !< Salinity [ppt]
00168   real,           intent(in)  :: pressure !< Pressure [Pa] or [R L2 T-2 ~> Pa]
00169   real,           intent(out) :: rho      !< Density (in-situ if pressure is local) [kg m-3] or [R ~> kg
       m-3]
00170   type(eos_type), pointer     :: EOS      !< Equation of state structure
00171   real, optional, intent(in)  :: rho_ref  !< A reference density [kg m-3]
00172   real, optional, intent(in)  :: scale    !< A multiplicative factor by which to scale density in
00173                                           !! combination with scaling given by US [various]
00174
00175   real :: rho_scale ! A factor to convert density from kg m-3 to the desired units [R m3 kg-1 ~> 1]
00176   real :: p_scale   ! A factor to convert pressure to units of Pa [Pa T2 R-1 L-2 ~> 1]
00177
00178   if (.not.associated(eos)) call mom_error(fatal, &
00179     "calculate_density_scalar called with an unassociated EOS_type EOS.")
00180
00181   p_scale = eos%RL2_T2_to_Pa
00182
00183   select case (eos%form_of_EOS)
00184     case (eos_linear)
00185       call calculate_density_linear(t, s, p_scale*pressure, rho, &
00186                                     eos%Rho_T0_S0, eos%dRho_dT, eos%dRho_dS, rho_ref)
00187     case (eos_unesco)
00188       call calculate_density_unesco(t, s, p_scale*pressure, rho, rho_ref)
00189     case (eos_wright)
00190       call calculate_density_wright(t, s, p_scale*pressure, rho, rho_ref)
00191     case (eos_teos10)
00192       call calculate_density_teos10(t, s, p_scale*pressure, rho, rho_ref)
00193     case (eos_nemo)
00194       call calculate_density_nemo(t, s, p_scale*pressure, rho, rho_ref)
00195     case default
00196       call mom_error(fatal, "calculate_density_scalar: EOS is not valid.")
00197   end select
00198
00199   rho_scale = eos%kg_m3_to_R
00200   if (present(scale)) rho_scale = rho_scale * scale
00201   rho = rho_scale * rho
00202
00203 end subroutine calculate_density_scalar
00204
00205 !> Calls the appropriate subroutine to calculate density of sea water for scalar inputs
00206 !! including the variance of T, S and covariance of T-S.
00207 !! The calculation uses only the second order correction in a series as discussed
00208 !! in Stanley et al., 2020.
00209 !! If rho_ref is present, the anomaly with respect to rho_ref is returned. The
00210 !! density can be rescaled using rho_ref.
00211 subroutine calculate_stanley_density_scalar(T, S, pressure, Tvar, TScov,
       Svar, rho, EOS, rho_ref, scale)
00212   real,           intent(in)  :: T        !< Potential temperature referenced to the surface [degC]
00213   real,           intent(in)  :: S        !< Salinity [ppt]
00214   real,           intent(in)  :: Tvar     !< Variance of potential temperature referenced to the surface
       [degC2]
00215   real,           intent(in)  :: TScov    !< Covariance of potential temperature and salinity [degC ppt]
00216   real,           intent(in)  :: Svar     !< Variance of salinity [ppt2]
00217   real,           intent(in)  :: pressure !< Pressure [Pa]
00218   real,           intent(out) :: rho      !< Density (in-situ if pressure is local) [kg m-3] or [R ~> kg
       m-3]
00219   type(eos_type), pointer     :: EOS      !< Equation of state structure
00220   real, optional, intent(in)  :: rho_ref  !< A reference density [kg m-3].
```

```
00221   real, optional, intent(in)  :: scale    !< A multiplicative factor by which to scale density
00222                                           !! from kg m-3 to the desired units [R m3 kg-1]
00223   ! Local variables
00224   real :: d2RdTT, d2RdST, d2RdSS, d2RdSp, d2RdTp ! Second derivatives of density wrt T,S,p
00225   real :: rho_scale ! A factor to convert density from kg m-3 to the desired units [R m3 kg-1 ~> 1]
00226   real :: p_scale   ! A factor to convert pressure to units of Pa [Pa T2 R-1 L-2 ~> 1]
00227
00228   if (.not.associated(eos)) call mom_error(fatal, &
00229     "calculate_stanley_density_scalar called with an unassociated EOS_type EOS.")
00230
00231   p_scale = eos%RL2_T2_to_Pa
00232
00233   select case (eos%form_of_EOS)
00234     case (eos_linear)
00235       call calculate_density_linear(t, s, p_scale*pressure, rho, &
00236                                     eos%Rho_T0_S0, eos%dRho_dT, eos%dRho_dS, rho_ref)
00237       call calculate_density_second_derivs_linear(t, s, pressure, d2rdss, d2rdst, &
00238                                                   d2rdtt, d2rdsp, d2rdtp)
00239     case (eos_wright)
00240       call calculate_density_wright(t, s, p_scale*pressure, rho, rho_ref)
00241       call calculate_density_second_derivs_wright(t, s, pressure, d2rdss, d2rdst, &
00242                                                   d2rdtt, d2rdsp, d2rdtp)
00243     case (eos_teos10)
00244       call calculate_density_teos10(t, s, p_scale*pressure, rho, rho_ref)
00245       call calculate_density_second_derivs_teos10(t, s, pressure, d2rdss, d2rdst, &
00246                                                   d2rdtt, d2rdsp, d2rdtp)
00247     case default
00248       call mom_error(fatal, "calculate_stanley_density_scalar: EOS is not valid.")
00249   end select
00250
00251   ! Equation 25 of Stanley et al., 2020.
00252   rho = rho + ( 0.5 * d2rdtt * tvar + ( d2rdst * tscov + 0.5 * d2rdss * svar ) )
00253
00254   rho_scale = eos%kg_m3_to_R
00255   if (present(scale)) rho_scale = rho_scale * scale
00256   rho = rho_scale * rho
00257
00258 end subroutine calculate_stanley_density_scalar
00259
00260 !> Calls the appropriate subroutine to calculate the density of sea water for 1-D array inputs.
00261 !! If rho_ref is present, the anomaly with respect to rho_ref is returned.
00262 subroutine calculate_density_array(T, S, pressure, rho, start, npts, EOS, rho_ref,
       scale)
00263   real, dimension(:), intent(in)    :: T        !< Potential temperature referenced to the surface [degC]
00264   real, dimension(:), intent(in)    :: S        !< Salinity [ppt]
00265   real, dimension(:), intent(in)    :: pressure !< Pressure [Pa] or [R L2 T-2 ~> Pa]
00266   real, dimension(:), intent(inout) :: rho      !< Density (in-situ if pressure is local) [kg m-3] or [R ~>
       kg m-3]
00267   integer,            intent(in)    :: start    !< Start index for computation
00268   integer,            intent(in)    :: npts     !< Number of point to compute
00269   type(eos_type),     pointer       :: EOS      !< Equation of state structure
00270   real,                  optional, intent(in) :: rho_ref  !< A reference density [kg m-3]
00271   real,                  optional, intent(in) :: scale    !< A multiplicative factor by which to scale
       density
00272                                                 !! in combination with scaling given by US [various]
00273   integer :: j
00274
00275   if (.not.associated(eos)) call mom_error(fatal, &
00276     "calculate_density_array called with an unassociated EOS_type EOS.")
00277
00278   select case (eos%form_of_EOS)
00279     case (eos_linear)
00280       call calculate_density_linear(t, s, pressure, rho, start, npts, &
00281                                     eos%Rho_T0_S0, eos%dRho_dT, eos%dRho_dS, rho_ref)
00282     case (eos_unesco)
00283       call calculate_density_unesco(t, s, pressure, rho, start, npts, rho_ref)
00284     case (eos_wright)
00285       call calculate_density_wright(t, s, pressure, rho, start, npts, rho_ref)
00286     case (eos_teos10)
00287       call calculate_density_teos10(t, s, pressure, rho, start, npts, rho_ref)
00288     case (eos_nemo)
00289     call calculate_density_nemo(t, s, pressure, rho, start, npts, rho_ref)
00290     case default
00291       call mom_error(fatal, "calculate_density_array: EOS%form_of_EOS is not valid.")
00292   end select
00293
00294   if (present(scale)) then ; if (scale /= 1.0) then ; do j=start,start+npts-1
00295     rho(j) = scale * rho(j)
00296   enddo ; endif ; endif
00297
00298 end subroutine calculate_density_array
00299
00300 !> Calls the appropriate subroutine to calculate the density of sea water for 1-D array inputs
00301 !! including the variance of T, S and covariance of T-S.
00302 !! The calculation uses only the second order correction in a series as discussed
00303 !! in Stanley et al., 2020.
00304 !! If rho_ref is present, the anomaly with respect to rho_ref is returned.
```

```
00305 subroutine calculate_stanley_density_array(T, S, pressure, Tvar, TScov,
      Svar, rho, start, npts, EOS, rho_ref, scale)
00306   real, dimension(:), intent(in)    :: T        !< Potential temperature referenced to the surface [degC]
00307   real, dimension(:), intent(in)    :: S        !< Salinity [ppt]
00308   real, dimension(:), intent(in)    :: pressure !< Pressure [Pa]
00309   real, dimension(:), intent(in)    :: Tvar     !< Variance of potential temperature referenced to the
      surface [degC2]
00310   real, dimension(:), intent(in)    :: TScov    !< Covariance of potential temperature and salinity [degC
      ppt]
00311   real, dimension(:), intent(in)    :: Svar     !< Variance of salinity [ppt2]
00312   real, dimension(:), intent(inout) :: rho      !< Density (in-situ if pressure is local) [kg m-3]
00313   integer,            intent(in)    :: start    !< Start index for computation
00314   integer,            intent(in)    :: npts     !< Number of point to compute
00315   type(eos_type),     pointer       :: EOS      !< Equation of state structure
00316   real,     optional, intent(in)    :: rho_ref  !< A reference density [kg m-3].
00317   real,     optional, intent(in)    :: scale    !< A multiplicative factor by which to scale density
00318                                                 !! from kg m-3 to the desired units [R m3 kg-1]
00319   ! Local variables
00320   real, dimension(size(T)) :: d2RdTT, d2RdST, d2RdSS, d2RdSp, d2RdTp ! Second derivatives of density wrt
      T,S,p
00321   integer :: j
00322
00323   if (.not.associated(eos)) call mom_error(fatal, &
00324     "calculate_density_array called with an unassociated EOS_type EOS.")
00325
00326   select case (eos%form_of_EOS)
00327     case (eos_linear)
00328       call calculate_density_linear(t, s, pressure, rho, start, npts, &
00329                                     eos%Rho_T0_S0, eos%dRho_dT, eos%dRho_dS, rho_ref)
00330       call calculate_density_second_derivs_linear(t, s, pressure, d2rdss, d2rdst, &
00331                                                   d2rdtt, d2rdsp, d2rdtp, start, npts)
00332     case (eos_wright)
00333       call calculate_density_wright(t, s, pressure, rho, start, npts, rho_ref)
00334       call calculate_density_second_derivs_wright(t, s, pressure, d2rdss, d2rdst, &
00335                                                   d2rdtt, d2rdsp, d2rdtp, start, npts)
00336     case (eos_teos10)
00337       call calculate_density_teos10(t, s, pressure, rho, start, npts, rho_ref)
00338       call calculate_density_second_derivs_teos10(t, s, pressure, d2rdss, d2rdst, &
00339                                                   d2rdtt, d2rdsp, d2rdtp, start, npts)
00340     case default
00341       call mom_error(fatal, "calculate_stanley_density_array: EOS%form_of_EOS is not valid.")
00342   end select
00343
00344   ! Equation 25 of Stanley et al., 2020.
00345   do j=start,start+npts-1
00346     rho(j) = rho(j) &
00347              + ( 0.5 * d2rdtt(j) * tvar(j) + ( d2rdst(j) * tscov(j) + 0.5 * d2rdss(j) * svar(j) ) )
00348   enddo
00349
00350   if (present(scale)) then ; if (scale /= 1.0) then ; do j=start,start+npts-1
00351     rho(j) = scale * rho(j)
00352   enddo ; endif ; endif
00353
00354 end subroutine calculate_stanley_density_array
00355
00356 !> Calls the appropriate subroutine to calculate the density of sea water for 1-D array inputs,
00357 !! potentially limiting the domain of indices that are worked on.
00358 !! If rho_ref is present, the anomaly with respect to rho_ref is returned.
00359 subroutine calculate_density_1d(T, S, pressure, rho, EOS, dom, rho_ref, scale)
00360   real, dimension(:),    intent(in)    :: T        !< Potential temperature referenced to the surface
      [degC]
00361   real, dimension(:),    intent(in)    :: S        !< Salinity [ppt]
00362   real, dimension(:),    intent(in)    :: pressure !< Pressure [R L2 T-2 ~> Pa]
00363   real, dimension(:),    intent(inout) :: rho      !< Density (in-situ if pressure is local) [R ~> kg m-3]
00364   type(eos_type),        pointer       :: EOS      !< Equation of state structure
00365   integer, dimension(2), optional, intent(in) :: dom   !< The domain of indices to work on, taking
00366                                                        !! into account that arrays start at 1.
00367   real,                  optional, intent(in) :: rho_ref !< A reference density [kg m-3]
00368   real,                  optional, intent(in) :: scale !< A multiplicative factor by which to scale density
00369                                                        !! in combination with scaling given by US [various]
00370   ! Local variables
00371   real :: p_scale   ! A factor to convert pressure to units of Pa [Pa T2 R-1 L-2 ~> 1]
00372   real :: rho_scale ! A factor to convert density from kg m-3 to the desired units [R m3 kg-1 ~> 1]
00373   real :: rho_unscale ! A factor to convert density from R to kg m-3 [kg m-3 R-1 ~> 1]
00374   real :: rho_reference ! rho_ref converted to [kg m-3]
00375   real, dimension(size(rho)) :: pres  ! Pressure converted to [Pa]
00376   integer :: i, is, ie, npts
00377
00378   if (.not.associated(eos)) call mom_error(fatal, &
00379     "calculate_density_1d called with an unassociated EOS_type EOS.")
00380
00381   if (present(dom)) then
00382     is = dom(1) ; ie = dom(2) ; npts = 1 + ie - is
00383   else
00384     is = 1 ; ie = size(rho) ; npts = 1 + ie - is
00385   endif
00386
```

```
00387   p_scale = eos%RL2_T2_to_Pa
00388   rho_unscale = eos%R_to_kg_m3
00389
00390   if ((p_scale == 1.0) .and. (rho_unscale == 1.0)) then
00391     call calculate_density_array(t, s, pressure, rho, is, npts, eos, rho_ref=rho_ref
      )
00392   elseif (present(rho_ref)) then ! This is the same as above, but with some extra work to rescale
       variables.
00393     do i=is,ie ; pres(i) = p_scale * pressure(i) ; enddo
00394     rho_reference = rho_unscale*rho_ref
00395     call calculate_density_array(t, s, pres, rho, is, npts, eos, rho_ref=
      rho_reference)
00396   else  ! There is rescaling of variables, but rho_ref is not present. Passing a 0 value of rho_ref
00397         ! changes answers at roundoff for some equations of state, like Wright and UNESCO.
00398     do i=is,ie ; pres(i) = p_scale * pressure(i) ; enddo
00399     call calculate_density_array(t, s, pres, rho, is, npts, eos)
00400   endif
00401
00402   rho_scale = eos%kg_m3_to_R
00403   if (present(scale)) rho_scale = rho_scale * scale
00404   if (rho_scale /= 1.0) then ; do i=is,ie
00405     rho(i) = rho_scale * rho(i)
00406   enddo ; endif
00407
00408 end subroutine calculate_density_1d
00409
00410 !> Calls the appropriate subroutine to calculate the density of sea water for 1-D array inputs
00411 !! including the variance of T, S and covariance of T-S,
00412 !! potentially limiting the domain of indices that are worked on.
00413 !! The calculation uses only the second order correction in a series as discussed
00414 !! in Stanley et al., 2020.
00415 !! If rho_ref is present, the anomaly with respect to rho_ref is returned.
00416 subroutine calculate_stanley_density_1d(T, S, pressure, Tvar, TScov, Svar, rho,
       EOS, dom, rho_ref, scale)
00417   real, dimension(:),    intent(in)    :: T        !< Potential temperature referenced to the surface
       [degC]
00418   real, dimension(:),    intent(in)    :: S        !< Salinity [ppt]
00419   real, dimension(:),    intent(in)    :: pressure !< Pressure [R L2 T-2 ~> Pa]
00420   real, dimension(:),    intent(in)    :: Tvar     !< Variance of potential temperature [degC2]
00421   real, dimension(:),    intent(in)    :: TScov    !< Covariance of potential temperature and salinity
       [degC ppt]
00422   real, dimension(:),    intent(in)    :: Svar     !< Variance of salinity [ppt2]
00423   real, dimension(:),    intent(inout) :: rho      !< Density (in-situ if pressure is local) [R ~> kg m-3]
00424   type(eos_type),        pointer       :: EOS      !< Equation of state structure
00425   integer, dimension(2), optional, intent(in) :: dom   !< The domain of indices to work on, taking
00426                                                        !! into account that arrays start at 1.
00427   real,                  optional, intent(in) :: rho_ref !< A reference density [kg m-3]
00428   real,                  optional, intent(in) :: scale !< A multiplicative factor by which to scale density
00429                                                        !! in combination with scaling given by US [various]
00430   ! Local variables
00431   real :: p_scale   ! A factor to convert pressure to units of Pa [Pa T2 R-1 L-2 ~> 1]
00432   real :: rho_scale ! A factor to convert density from kg m-3 to the desired units [R m3 kg-1 ~> 1]
00433   real, dimension(size(rho)) :: pres  ! Pressure converted to [Pa]
00434   real, dimension(size(T)) :: d2RdTT, d2RdST, d2RdSS, d2RdSp, d2RdTp ! Second derivatives of density wrt
       T,S,p
00435   integer :: i, is, ie, npts
00436
00437   if (.not.associated(eos)) call mom_error(fatal, &
00438     "calculate_density_1d called with an unassociated EOS_type EOS.")
00439
00440   if (present(dom)) then
00441     is = dom(1) ; ie = dom(2) ; npts = 1 + ie - is
00442   else
00443     is = 1 ; ie = size(rho) ; npts = 1 + ie - is
00444   endif
00445
00446   p_scale = eos%RL2_T2_to_Pa
00447   do i=is,ie
00448     pres(i) = p_scale * pressure(i)
00449   enddo
00450
00451   select case (eos%form_of_EOS)
00452     case (eos_linear)
00453       call calculate_density_linear(t, s, pres, rho, 1, npts, &
00454                                     eos%Rho_T0_S0, eos%dRho_dT, eos%dRho_dS, rho_ref)
00455       call calculate_density_second_derivs_linear(t, s, pres, d2rdss, d2rdst, &
00456                                                   d2rdtt, d2rdsp, d2rdtp, 1, npts)
00457     case (eos_wright)
00458       call calculate_density_wright(t, s, pres, rho, 1, npts, rho_ref)
00459       call calculate_density_second_derivs_wright(t, s, pres, d2rdss, d2rdst, &
00460                                                   d2rdtt, d2rdsp, d2rdtp, 1, npts)
00461     case (eos_teos10)
00462       call calculate_density_teos10(t, s, pres, rho, 1, npts, rho_ref)
00463       call calculate_density_second_derivs_teos10(t, s, pres, d2rdss, d2rdst, &
00464                                                   d2rdtt, d2rdsp, d2rdtp, 1, npts)
00465     case default
00466       call mom_error(fatal, "calculate_stanley_density_scalar: EOS is not valid.")
```

```
00467   end select
00468
00469   ! Equation 25 of Stanley et al., 2020.
00470   do i=is,ie
00471     rho(i) = rho(i) &
00472              + ( 0.5 * d2rdtt(i) * tvar(i) + ( d2rdst(i) * tscov(i) + 0.5 * d2rdss(i) * svar(i) ) )
00473   enddo
00474
00475   rho_scale = eos%kg_m3_to_R
00476   if (present(scale)) rho_scale = rho_scale * scale
00477   if (rho_scale /= 1.0) then ; do i=is,ie
00478     rho(i) = rho_scale * rho(i)
00479   enddo ; endif
00480
00481 end subroutine calculate_stanley_density_1d
00482
00483 !> Calls the appropriate subroutine to calculate the specific volume of sea water
00484 !! for 1-D array inputs.
00485 subroutine calculate_spec_vol_array(T, S, pressure, specvol, start, npts, EOS,
       spv_ref, scale)
00486   real, dimension(:), intent(in)    :: T        !< potential temperature relative to the surface [degC]
00487   real, dimension(:), intent(in)    :: S        !< salinity [ppt]
00488   real, dimension(:), intent(in)    :: pressure !< pressure [Pa]
00489   real, dimension(:), intent(inout) :: specvol  !< in situ specific volume [kg m-3]
00490   integer,            intent(in)    :: start    !< the starting point in the arrays.
00491   integer,            intent(in)    :: npts     !< the number of values to calculate.
00492   type(eos_type),     pointer       :: EOS      !< Equation of state structure
00493   real,     optional, intent(in)    :: spv_ref  !< A reference specific volume [m3 kg-1]
00494   real,     optional, intent(in)    :: scale    !< A multiplicative factor by which to scale specific
00495                                                 !! volume in combination with scaling given by US [various]
00496
00497   real, dimension(size(specvol))  :: rho   ! Density [kg m-3]
00498   integer :: j
00499
00500   if (.not.associated(eos)) call mom_error(fatal, &
00501     "calculate_spec_vol_array called with an unassociated EOS_type EOS.")
00502
00503   select case (eos%form_of_EOS)
00504     case (eos_linear)
00505       call calculate_spec_vol_linear(t, s, pressure, specvol, start, npts, &
00506                eos%rho_T0_S0, eos%drho_dT, eos%drho_dS, spv_ref)
00507     case (eos_unesco)
00508       call calculate_spec_vol_unesco(t, s, pressure, specvol, start, npts, spv_ref)
00509     case (eos_wright)
00510       call calculate_spec_vol_wright(t, s, pressure, specvol, start, npts, spv_ref)
00511     case (eos_teos10)
00512       call calculate_spec_vol_teos10(t, s, pressure, specvol, start, npts, spv_ref)
00513     case (eos_nemo)
00514       call calculate_density_nemo(t, s, pressure, rho, start, npts)
00515       if (present(spv_ref)) then
00516         specvol(:) = 1.0 / rho(:) - spv_ref
00517       else
00518         specvol(:) = 1.0 / rho(:)
00519       endif
00520     case default
00521       call mom_error(fatal, "calculate_spec_vol_array: EOS%form_of_EOS is not valid.")
00522   end select
00523
00524   if (present(scale)) then ; if (scale /= 1.0) then ; do j=start,start+npts-1
00525     specvol(j) = scale * specvol(j)
00526   enddo ; endif ; endif
00527
00528 end subroutine calculate_spec_vol_array
00529
00530 !> Calls the appropriate subroutine to calculate specific volume of sea water
00531 !! for scalar inputs.
00532 subroutine calc_spec_vol_scalar(T, S, pressure, specvol, EOS, spv_ref, scale)
00533   real,           intent(in)  :: T        !< Potential temperature referenced to the surface [degC]
00534   real,           intent(in)  :: S        !< Salinity [ppt]
00535   real,           intent(in)  :: pressure !< Pressure [Pa] or [R L2 T-2 ~> Pa]
00536   real,           intent(out) :: specvol  !< In situ? specific volume [m3 kg-1] or [R-1 ~> m3 kg-1]
00537   type(eos_type), pointer     :: EOS      !< Equation of state structure
00538   real, optional, intent(in)  :: spv_ref  !< A reference specific volume [m3 kg-1] or [R-1 m3 kg-1]
00539   real, optional, intent(in)  :: scale    !< A multiplicative factor by which to scale specific
00540                                           !! volume in combination with scaling given by US [various]
00541
00542   real, dimension(1) :: Ta, Sa, pres, spv  ! Rescaled single element array versions of the arguments.
00543   real :: spv_reference ! spv_ref converted to [m3 kg-1]
00544   real :: spv_scale ! A factor to convert specific volume from m3 kg-1 to the desired units [kg R-1 m-3 ~>
       1]
00545
00546   if (.not.associated(eos)) call mom_error(fatal, &
00547     "calc_spec_vol_scalar called with an unassociated EOS_type EOS.")
00548
00549   pres(1) = eos%RL2_T2_to_Pa*pressure
00550   ta(1) = t ; sa(1) = s
00551
```

```
00552   if (present(spv_ref)) then
00553     spv_reference = eos%kg_m3_to_R*spv_ref
00554     call calculate_spec_vol_array(ta, sa, pres, spv, 1, 1, eos, spv_reference)
00555   else
00556     call calculate_spec_vol_array(ta, sa, pres, spv, 1, 1, eos)
00557   endif
00558   specvol = spv(1)
00559
00560   spv_scale = eos%R_to_kg_m3
00561   if (present(scale)) spv_scale = spv_scale * scale
00562   if (spv_scale /= 1.0) then
00563     specvol = spv_scale * specvol
00564   endif
00565
00566 end subroutine calc_spec_vol_scalar
00567
00568 !> Calls the appropriate subroutine to calculate the specific volume of sea water for 1-D array
00569 !! inputs, potentially limiting the domain of indices that are worked on.
00570 subroutine calc_spec_vol_1d(T, S, pressure, specvol, EOS, dom, spv_ref, scale)
00571   real, dimension(:),    intent(in)    :: T        !< Potential temperature referenced to the surface
      [degC]
00572   real, dimension(:),    intent(in)    :: S        !< Salinity [ppt]
00573   real, dimension(:),    intent(in)    :: pressure !< Pressure [R L2 T-2 ~> Pa]
00574   real, dimension(:),    intent(inout) :: specvol  !< In situ specific volume [R-1 ~> m3 kg-1]
00575   type(eos_type),        pointer       :: EOS      !< Equation of state structure
00576   integer, dimension(2), optional, intent(in) :: dom   !< The domain of indices to work on, taking
00577                                                        !! into account that arrays start at 1.
00578   real,                  optional, intent(in) :: spv_ref !< A reference specific volume [R-1 ~> m3 kg-1]
00579   real,                  optional, intent(in) :: scale !< A multiplicative factor by which to scale
00580                                                        !! output specific volume in combination with
00581                                                        !! scaling given by US [various]
00582   ! Local variables
00583   real, dimension(size(specvol)) :: pres  ! Pressure converted to [Pa]
00584   real :: p_scale   ! A factor to convert pressure to units of Pa [Pa T2 R-1 L-2 ~> 1]
00585   real :: spv_unscale ! A factor to convert specific volume from R-1 to m3 kg-1 [m3 kg-1 R ~> 1]
00586   real :: spv_scale ! A factor to convert specific volume from m3 kg-1 to the desired units [kg m-3 R-1 ~>
      1]
00587   real :: spv_reference ! spv_ref converted to [m3 kg-1]
00588   integer :: i, is, ie, npts
00589
00590   if (.not.associated(eos)) call mom_error(fatal, &
00591     "calc_spec_vol_1d called with an unassociated EOS_type EOS.")
00592
00593   if (present(dom)) then
00594     is = dom(1) ; ie = dom(2) ; npts = 1 + ie - is
00595   else
00596     is = 1 ; ie = size(specvol) ; npts = 1 + ie - is
00597   endif
00598
00599   p_scale = eos%RL2_T2_to_Pa
00600   spv_unscale = eos%kg_m3_to_R
00601
00602   if ((p_scale == 1.0) .and. (spv_unscale == 1.0)) then
00603     call calculate_spec_vol_array(t, s, pressure, specvol, is, npts, eos, spv_ref)
00604   elseif (present(spv_ref)) then ! This is the same as above, but with some extra work to rescale
      variables.
00605     do i=is,ie ; pres(i) = p_scale * pressure(i) ; enddo
00606     spv_reference = spv_unscale*spv_ref
00607     call calculate_spec_vol_array(t, s, pres, specvol, is, npts, eos, spv_reference
      )
00608   else  ! There is rescaling of variables, but spv_ref is not present. Passing a 0 value of spv_ref
00609         ! changes answers at roundoff for some equations of state, like Wright and UNESCO.
00610     do i=is,ie ; pres(i) = p_scale * pressure(i) ; enddo
00611     call calculate_spec_vol_array(t, s, pres, specvol, is, npts, eos)
00612   endif
00613
00614   spv_scale = eos%R_to_kg_m3
00615   if (present(scale)) spv_scale = spv_scale * scale
00616   if (spv_scale /= 1.0) then ; do i=is,ie
00617     specvol(i) = spv_scale * specvol(i)
00618   enddo ; endif
00619
00620 end subroutine calc_spec_vol_1d
00621
00622
00623 !> Calls the appropriate subroutine to calculate the freezing point for scalar inputs.
00624 subroutine calculate_tfreeze_scalar(S, pressure, T_fr, EOS, pres_scale)
00625   real,           intent(in)  :: S !< Salinity [ppt]
00626   real,           intent(in)  :: pressure !< Pressure [Pa] or [other]
00627   real,           intent(out) :: T_fr !< Freezing point potential temperature referenced
00628                                       !! to the surface [degC]
00629   type(eos_type), pointer     :: EOS !< Equation of state structure
00630   real, optional, intent(in)  :: pres_scale !< A multiplicative factor to convert pressure into Pa
00631
00632   ! Local variables
00633   real :: p_scale ! A factor to convert pressure to units of Pa.
00634
```

```
00635   if (.not.associated(eos)) call mom_error(fatal, &
00636     "calculate_TFreeze_scalar called with an unassociated EOS_type EOS.")
00637
00638   p_scale = 1.0 ; if (present(pres_scale)) p_scale = pres_scale
00639
00640   select case (eos%form_of_TFreeze)
00641     case (tfreeze_linear)
00642       call calculate_tfreeze_linear(s, p_scale*pressure, t_fr, eos%TFr_S0_P0, &
00643                                     eos%dTFr_dS, eos%dTFr_dp)
00644     case (tfreeze_millero)
00645       call calculate_tfreeze_millero(s, p_scale*pressure, t_fr)
00646     case (tfreeze_teos10)
00647       call calculate_tfreeze_teos10(s, p_scale*pressure, t_fr)
00648     case default
00649       call mom_error(fatal, "calculate_TFreeze_scalar: form_of_TFreeze is not valid.")
00650   end select
00651
00652 end subroutine calculate_tfreeze_scalar
00653
00654 !> Calls the appropriate subroutine to calculate the freezing point for a 1-D array.
00655 subroutine calculate_tfreeze_array(S, pressure, T_fr, start, npts, EOS, pres_scale)
00656   real, dimension(:), intent(in)    :: S        !< Salinity [ppt]
00657   real, dimension(:), intent(in)    :: pressure !< Pressure [Pa] or [other]
00658   real, dimension(:), intent(inout) :: T_fr     !< Freezing point potential temperature referenced
00659                                                 !! to the surface [degC]
00660   integer,            intent(in)    :: start    !< Starting index within the array
00661   integer,            intent(in)    :: npts     !< The number of values to calculate
00662   type(eos_type),     pointer       :: EOS      !< Equation of state structure
00663   real,     optional, intent(in)    :: pres_scale !< A multiplicative factor to convert pressure into Pa.
00664
00665   ! Local variables
00666   real, dimension(size(pressure)) :: pres  ! Pressure converted to [Pa]
00667   real :: p_scale ! A factor to convert pressure to units of Pa.
00668   integer :: j
00669
00670   if (.not.associated(eos)) call mom_error(fatal, &
00671     "calculate_TFreeze_scalar called with an unassociated EOS_type EOS.")
00672
00673   p_scale = 1.0 ; if (present(pres_scale)) p_scale = pres_scale
00674
00675   if (p_scale == 1.0) then
00676     select case (eos%form_of_TFreeze)
00677       case (tfreeze_linear)
00678         call calculate_tfreeze_linear(s, pressure, t_fr, start, npts, &
00679                                       eos%TFr_S0_P0, eos%dTFr_dS, eos%dTFr_dp)
00680       case (tfreeze_millero)
00681         call calculate_tfreeze_millero(s, pressure, t_fr, start, npts)
00682       case (tfreeze_teos10)
00683         call calculate_tfreeze_teos10(s, pressure, t_fr, start, npts)
00684       case default
00685         call mom_error(fatal, "calculate_TFreeze_scalar: form_of_TFreeze is not valid.")
00686     end select
00687   else
00688     do j=start,start+npts-1 ; pres(j) = p_scale * pressure(j) ; enddo
00689     select case (eos%form_of_TFreeze)
00690       case (tfreeze_linear)
00691         call calculate_tfreeze_linear(s, pres, t_fr, start, npts, &
00692                                       eos%TFr_S0_P0, eos%dTFr_dS, eos%dTFr_dp)
00693       case (tfreeze_millero)
00694         call calculate_tfreeze_millero(s, pres, t_fr, start, npts)
00695       case (tfreeze_teos10)
00696         call calculate_tfreeze_teos10(s, pres, t_fr, start, npts)
00697       case default
00698         call mom_error(fatal, "calculate_TFreeze_scalar: form_of_TFreeze is not valid.")
00699     end select
00700   endif
00701
00702 end subroutine calculate_tfreeze_array
00703
00704 !> Calls the appropriate subroutine to calculate density derivatives for 1-D array inputs.
00705 subroutine calculate_density_derivs_array(T, S, pressure, drho_dT, drho_dS,
      start, npts, EOS, scale)
00706   real, dimension(:), intent(in)    :: T        !< Potential temperature referenced to the surface [degC]
00707   real, dimension(:), intent(in)    :: S        !< Salinity [ppt]
00708   real, dimension(:), intent(in)    :: pressure !< Pressure [Pa] or [R L2 T-2 ~> Pa]
00709   real, dimension(:), intent(inout) :: drho_dT  !< The partial derivative of density with potential
00710                                                 !! temperature [kg m-3 degC-1] or [R degC-1 ~> kg m-3
      degC-1]
00711   real, dimension(:), intent(inout) :: drho_dS  !< The partial derivative of density with salinity,
00712                                                 !! in [kg m-3 ppt-1] or [R degC-1 ~> kg m-3 ppt-1]
00713   integer,            intent(in)    :: start    !< Starting index within the array
00714   integer,            intent(in)    :: npts     !< The number of values to calculate
00715   type(eos_type),     pointer       :: EOS      !< Equation of state structure
00716   real,     optional, intent(in)    :: scale !< A multiplicative factor by which to scale density
00717                                                 !! in combination with scaling given by US [various]
00718
00719   ! Local variables
```

```
00720   integer :: j
00721
00722   if (.not.associated(eos)) call mom_error(fatal, &
00723     "calculate_density_derivs called with an unassociated EOS_type EOS.")
00724
00725   select case (eos%form_of_EOS)
00726     case (eos_linear)
00727       call calculate_density_derivs_linear(t, s, pressure, drho_dt, drho_ds, eos%Rho_T0_S0, &
00728                                            eos%dRho_dT, eos%dRho_dS, start, npts)
00729     case (eos_unesco)
00730       call calculate_density_derivs_unesco(t, s, pressure, drho_dt, drho_ds, start, npts)
00731     case (eos_wright)
00732       call calculate_density_derivs_wright(t, s, pressure, drho_dt, drho_ds, start, npts)
00733     case (eos_teos10)
00734       call calculate_density_derivs_teos10(t, s, pressure, drho_dt, drho_ds, start, npts)
00735     case (eos_nemo)
00736       call calculate_density_derivs_nemo(t, s, pressure, drho_dt, drho_ds, start, npts)
00737     case default
00738       call mom_error(fatal, "calculate_density_derivs_array: EOS%form_of_EOS is not valid.")
00739   end select
00740
00741   if (present(scale)) then ; if (scale /= 1.0) then ; do j=start,start+npts-1
00742     drho_dt(j) = scale * drho_dt(j)
00743     drho_ds(j) = scale * drho_ds(j)
00744   enddo ; endif ; endif
00745
00746 end subroutine calculate_density_derivs_array
00747
00748
00749 !> Calls the appropriate subroutine to calculate density derivatives for 1-D array inputs.
00750 subroutine calculate_density_derivs_1d(T, S, pressure, drho_dT, drho_dS, EOS,
      dom, scale)
00751   real, dimension(:),     intent(in)    :: T        !< Potential temperature referenced to the surface
      [degC]
00752   real, dimension(:),     intent(in)    :: S        !< Salinity [ppt]
00753   real, dimension(:),     intent(in)    :: pressure !< Pressure [R L2 T-2 ~> Pa]
00754   real, dimension(:),     intent(inout) :: drho_dT  !< The partial derivative of density with potential
00755                                                     !! temperature [R degC-1 ~> kg m-3 degC-1]
00756   real, dimension(:),     intent(inout) :: drho_dS  !< The partial derivative of density with salinity
00757                                                     !! [R degC-1 ~> kg m-3 ppt-1]
00758   type(eos_type),         pointer       :: EOS      !< Equation of state structure
00759   integer, dimension(2), optional, intent(in) :: dom   !< The domain of indices to work on, taking
00760                                                        !! into account that arrays start at 1.
00761   real,                  optional, intent(in) :: scale !< A multiplicative factor by which to scale density
00762                                                        !! in combination with scaling given by US [various]
00763   ! Local variables
00764   real, dimension(size(drho_dT)) :: pres  ! Pressure converted to [Pa]
00765   real :: rho_scale ! A factor to convert density from kg m-3 to the desired units [R m3 kg-1 ~> 1]
00766   real :: p_scale   ! A factor to convert pressure to units of Pa [Pa T2 R-1 L-2 ~> 1]
00767   integer :: i, is, ie, npts
00768
00769   if (.not.associated(eos)) call mom_error(fatal, &
00770     "calculate_density_derivs called with an unassociated EOS_type EOS.")
00771
00772   if (present(dom)) then
00773     is = dom(1) ; ie = dom(2) ; npts = 1 + ie - is
00774   else
00775     is = 1 ; ie = size(drho_dt) ; npts = 1 + ie - is
00776   endif
00777
00778   p_scale = eos%RL2_T2_to_Pa
00779
00780   if (p_scale == 1.0) then
00781     call calculate_density_derivs_array(t, s, pressure, drho_dt, drho_ds, is,
      npts, eos)
00782   else
00783     do i=is,ie ; pres(i) = p_scale * pressure(i) ; enddo
00784     call calculate_density_derivs_array(t, s, pres, drho_dt, drho_ds, is,
     npts, eos)
00785   endif
00786
00787   rho_scale = eos%kg_m3_to_R
00788   if (present(scale)) rho_scale = rho_scale * scale
00789   if (rho_scale /= 1.0) then ; do i=is,ie
00790     drho_dt(i) = rho_scale * drho_dt(i)
00791     drho_ds(i) = rho_scale * drho_ds(i)
00792   enddo ; endif
00793
00794 end subroutine calculate_density_derivs_1d
00795
00796
00797 !> Calls the appropriate subroutines to calculate density derivatives by promoting a scalar
00798 !! to a one-element array
00799 subroutine calculate_density_derivs_scalar(T, S, pressure, drho_dT, drho_dS,
      EOS, scale)
00800   real,           intent(in)  :: T !< Potential temperature referenced to the surface [degC]
00801   real,           intent(in)  :: S !< Salinity [ppt]
```

```
00802   real,           intent(in)  :: pressure !< Pressure [Pa] or [R L2 T-2 ~> Pa]
00803   real,           intent(out) :: drho_dT !< The partial derivative of density with potential
00804                                  !! temperature [kg m-3 degC-1] or [R degC-1 ~> kg m-3 degC-1]
00805   real,           intent(out) :: drho_dS !< The partial derivative of density with salinity,
00806                                  !! in [kg m-3 ppt-1] or [R ppt-1 ~> kg m-3 ppt-1]
00807   type(eos_type), pointer     :: EOS     !< Equation of state structure
00808   real, optional, intent(in)  :: scale   !< A multiplicative factor by which to scale density
00809                                  !! in combination with scaling given by US [various]
00810   ! Local variables
00811   real :: rho_scale ! A factor to convert density from kg m-3 to the desired units [R m3 kg-1 ~> 1]
00812   real :: p_scale   ! A factor to convert pressure to units of Pa [Pa T2 R-1 L-2 ~> 1]
00813   integer :: j
00814
00815   if (.not.associated(eos)) call mom_error(fatal, &
00816     "calculate_density_derivs called with an unassociated EOS_type EOS.")
00817
00818   p_scale = eos%RL2_T2_to_Pa
00819
00820   select case (eos%form_of_EOS)
00821     case (eos_linear)
00822       call calculate_density_derivs_linear(t, s, p_scale*pressure, drho_dt, drho_ds, &
00823                                            eos%Rho_T0_S0, eos%dRho_dT, eos%dRho_dS)
00824     case (eos_wright)
00825       call calculate_density_derivs_wright(t, s, p_scale*pressure, drho_dt, drho_ds)
00826     case (eos_teos10)
00827       call calculate_density_derivs_teos10(t, s, p_scale*pressure, drho_dt, drho_ds)
00828     case default
00829       call mom_error(fatal, "calculate_density_derivs_scalar: EOS%form_of_EOS is not valid.")
00830   end select
00831
00832   rho_scale = eos%kg_m3_to_R
00833   if (present(scale)) rho_scale = rho_scale * scale
00834   if (rho_scale /= 1.0) then
00835     drho_dt = rho_scale * drho_dt
00836     drho_ds = rho_scale * drho_ds
00837   endif
00838
00839 end subroutine calculate_density_derivs_scalar
00840
00841 !> Calls the appropriate subroutine to calculate density second derivatives for 1-D array inputs.
00842 subroutine calculate_density_second_derivs_array(T, S, pressure,
      drho_dS_dS, drho_dS_dT, drho_dT_dT, &
00843                                                  drho_dS_dP, drho_dT_dP, start, npts, EOS, scale)
00844   real, dimension(:), intent(in)  :: T !< Potential temperature referenced to the surface [degC]
00845   real, dimension(:), intent(in)  :: S !< Salinity [ppt]
00846   real, dimension(:), intent(in)  :: pressure   !< Pressure [Pa] or [R L2 T-2 ~> Pa]
00847   real, dimension(:), intent(inout) :: drho_dS_dS !< Partial derivative of beta with respect to S
00848                                                   !!  [kg m-3 ppt-2] or [R ppt-2 ~> kg m-3 ppt-2]
00849   real, dimension(:), intent(inout) :: drho_dS_dT !< Partial derivative of beta with respect to T
00850                                                   !! [kg m-3 ppt-1 degC-1] or [R ppt-1 degC-1 ~> kg m-3
      ppt-1 degC-1]
00851   real, dimension(:), intent(inout) :: drho_dT_dT !< Partial derivative of alpha with respect to T
00852                                                   !! [kg m-3 degC-2] or [R degC-2 ~> kg m-3 degC-2]
00853   real, dimension(:), intent(inout) :: drho_dS_dP !< Partial derivative of beta with respect to pressure
00854                                                   !! [kg m-3 ppt-1 Pa-1] or [R ppt-1 Pa-1 ~> kg m-3 ppt-1
      Pa-1]
00855   real, dimension(:), intent(inout) :: drho_dT_dP !< Partial derivative of alpha with respect to pressure
00856                                                   !! [kg m-3 degC-1 Pa-1] or [R degC-1 Pa-1 ~> kg m-3
      degC-1 Pa-1]
00857   integer,            intent(in)  :: start !< Starting index within the array
00858   integer,            intent(in)  :: npts  !< The number of values to calculate
00859   type(eos_type),     pointer     :: EOS   !< Equation of state structure
00860   real,                  optional, intent(in) :: scale !< A multiplicative factor by which to scale density
00861                                                   !! in combination with scaling given by US [various]
00862   ! Local variables
00863   real, dimension(size(pressure)) :: pres  ! Pressure converted to [Pa]
00864   real :: rho_scale ! A factor to convert density from kg m-3 to the desired units [R m3 kg-1 ~> 1]
00865   real :: p_scale   ! A factor to convert pressure to units of Pa [Pa T2 R-1 L-2 ~> 1]
00866   real :: I_p_scale ! The inverse of the factor to convert pressure to units of Pa [R L2 T-2 Pa-1 ~> 1]
00867   integer :: j
00868
00869   if (.not.associated(eos)) call mom_error(fatal, &
00870     "calculate_density_derivs called with an unassociated EOS_type EOS.")
00871
00872   p_scale = eos%RL2_T2_to_Pa
00873
00874   if (p_scale == 1.0) then
00875     select case (eos%form_of_EOS)
00876       case (eos_linear)
00877         call calculate_density_second_derivs_linear(t, s, pressure, drho_ds_ds, drho_ds_dt, &
00878                                                     drho_dt_dt, drho_ds_dp, drho_dt_dp, start, npts)
00879       case (eos_wright)
00880         call calculate_density_second_derivs_wright(t, s, pressure, drho_ds_ds, drho_ds_dt, &
00881                                                     drho_dt_dt, drho_ds_dp, drho_dt_dp, start, npts)
00882       case (eos_teos10)
00883         call calculate_density_second_derivs_teos10(t, s, pressure, drho_ds_ds, drho_ds_dt, &
00884                                                     drho_dt_dt, drho_ds_dp, drho_dt_dp, start, npts)
```

```
00885       case default
00886         call mom_error(fatal, "calculate_density_derivs: EOS%form_of_EOS is not valid.")
00887     end select
00888   else
00889     do j=start,start+npts-1 ; pres(j) = p_scale * pressure(j) ; enddo
00890     select case (eos%form_of_EOS)
00891       case (eos_linear)
00892         call calculate_density_second_derivs_linear(t, s, pres, drho_ds_ds, drho_ds_dt, &
00893                                                     drho_dt_dt, drho_ds_dp, drho_dt_dp, start, npts)
00894       case (eos_wright)
00895         call calculate_density_second_derivs_wright(t, s, pres, drho_ds_ds, drho_ds_dt, &
00896                                                     drho_dt_dt, drho_ds_dp, drho_dt_dp, start, npts)
00897       case (eos_teos10)
00898         call calculate_density_second_derivs_teos10(t, s, pres, drho_ds_ds, drho_ds_dt, &
00899                                                     drho_dt_dt, drho_ds_dp, drho_dt_dp, start, npts)
00900       case default
00901         call mom_error(fatal, "calculate_density_derivs: EOS%form_of_EOS is not valid.")
00902     end select
00903   endif
00904
00905   rho_scale = eos%kg_m3_to_R
00906   if (present(scale)) rho_scale = rho_scale * scale
00907   if (rho_scale /= 1.0) then ; do j=start,start+npts-1
00908     drho_ds_ds(j) = rho_scale * drho_ds_ds(j)
00909     drho_ds_dt(j) = rho_scale * drho_ds_dt(j)
00910     drho_dt_dt(j) = rho_scale * drho_dt_dt(j)
00911     drho_ds_dp(j) = rho_scale * drho_ds_dp(j)
00912     drho_dt_dp(j) = rho_scale * drho_dt_dp(j)
00913   enddo ; endif
00914
00915   if (p_scale /= 1.0) then
00916     i_p_scale = 1.0 / p_scale
00917     do j=start,start+npts-1
00918       drho_ds_dp(j) = i_p_scale * drho_ds_dp(j)
00919       drho_dt_dp(j) = i_p_scale * drho_dt_dp(j)
00920     enddo
00921   endif
00922
00923 end subroutine calculate_density_second_derivs_array
00924
00925 !> Calls the appropriate subroutine to calculate density second derivatives for scalar nputs.
00926 subroutine calculate_density_second_derivs_scalar(T, S, pressure,
       drho_dS_dS, drho_dS_dT, drho_dT_dT, &
00927                                                  drho_dS_dP, drho_dT_dP, EOS, scale)
00928   real, intent(in)  :: T !< Potential temperature referenced to the surface [degC]
00929   real, intent(in)  :: S !< Salinity [ppt]
00930   real, intent(in)  :: pressure   !< Pressure [Pa] or [R L2 T-2 ~> Pa]
00931   real, intent(out) :: drho_dS_dS !< Partial derivative of beta with respect to S
00932                                   !! [kg m-3 ppt-2] or [R ppt-2 ~> kg m-3 ppt-2]
00933   real, intent(out) :: drho_dS_dT !< Partial derivative of beta with respect to T
00934                                   !! [kg m-3 ppt-1 degC-1] or [R ppt-1 degC-1 ~> kg m-3 ppt-1 degC-1]
00935   real, intent(out) :: drho_dT_dT !< Partial derivative of alpha with respect to T
00936                                   !! [kg m-3 degC-2] or [R degC-2 ~> kg m-3 degC-2]
00937   real, intent(out) :: drho_dS_dP !< Partial derivative of beta with respect to pressure
00938                                   !! [kg m-3 ppt-1 Pa-1] or [R ppt-1 Pa-1 ~> kg m-3 ppt-1 Pa-1]
00939   real, intent(out) :: drho_dT_dP !< Partial derivative of alpha with respect to pressure
00940                                   !! [kg m-3 degC-1 Pa-1] or [R degC-1 Pa-1 ~> kg m-3 degC-1 Pa-1]
00941   type(eos_type), pointer    :: EOS !< Equation of state structure
00942   real, optional, intent(in) :: scale !< A multiplicative factor by which to scale density
00943                                   !! in combination with scaling given by US [various]
00944   ! Local variables
00945   real :: rho_scale ! A factor to convert density from kg m-3 to the desired units [R m3 kg-1 ~> 1]
00946   real :: p_scale   ! A factor to convert pressure to units of Pa [Pa T2 R-1 L-2 ~> 1]
00947   real :: I_p_scale ! The inverse of the factor to convert pressure to units of Pa [R L2 T-2 Pa-1 ~> 1]
00948
00949   if (.not.associated(eos)) call mom_error(fatal, &
00950     "calculate_density_derivs called with an unassociated EOS_type EOS.")
00951
00952   p_scale = eos%RL2_T2_to_Pa
00953
00954   select case (eos%form_of_EOS)
00955     case (eos_linear)
00956       call calculate_density_second_derivs_linear(t, s, p_scale*pressure, drho_ds_ds, drho_ds_dt, &
00957                                                   drho_dt_dt, drho_ds_dp, drho_dt_dp)
00958     case (eos_wright)
00959       call calculate_density_second_derivs_wright(t, s, p_scale*pressure, drho_ds_ds, drho_ds_dt, &
00960                                                   drho_dt_dt, drho_ds_dp, drho_dt_dp)
00961     case (eos_teos10)
00962       call calculate_density_second_derivs_teos10(t, s, p_scale*pressure, drho_ds_ds, drho_ds_dt, &
00963                                                   drho_dt_dt, drho_ds_dp, drho_dt_dp)
00964     case default
00965       call mom_error(fatal, "calculate_density_derivs: EOS%form_of_EOS is not valid.")
00966   end select
00967
00968   rho_scale = eos%kg_m3_to_R
00969   if (present(scale)) rho_scale = rho_scale * scale
00970   if (rho_scale /= 1.0) then
```

```
00971     drho_ds_ds = rho_scale * drho_ds_ds
00972     drho_ds_dt = rho_scale * drho_ds_dt
00973     drho_dt_dt = rho_scale * drho_dt_dt
00974     drho_ds_dp = rho_scale * drho_ds_dp
00975     drho_dt_dp = rho_scale * drho_dt_dp
00976   endif
00977
00978   if (p_scale /= 1.0) then
00979     i_p_scale = 1.0 / p_scale
00980     drho_ds_dp = i_p_scale * drho_ds_dp
00981     drho_dt_dp = i_p_scale * drho_dt_dp
00982   endif
00983
00984 end subroutine calculate_density_second_derivs_scalar
00985
00986 !> Calls the appropriate subroutine to calculate specific volume derivatives for an array.
00987 subroutine calculate_spec_vol_derivs_array(T, S, pressure, dSV_dT, dSV_dS,
       start, npts, EOS)
00988   real, dimension(:), intent(in)  :: T !< Potential temperature referenced to the surface [degC]
00989   real, dimension(:), intent(in)  :: S !< Salinity [ppt]
00990   real, dimension(:), intent(in)  :: pressure !< Pressure [Pa]
00991   real, dimension(:), intent(inout) :: dSV_dT !< The partial derivative of specific volume with potential
00992                                                !! temperature [m3 kg-1 degC-1]
00993   real, dimension(:), intent(inout) :: dSV_dS !< The partial derivative of specific volume with salinity
00994                                                !! [m3 kg-1 ppt-1]
00995   integer,            intent(in)  :: start  !< Starting index within the array
00996   integer,            intent(in)  :: npts   !< The number of values to calculate
00997   type(eos_type),     pointer     :: EOS    !< Equation of state structure
00998
00999   ! Local variables
01000   real, dimension(size(T)) :: press   ! Pressure converted to [Pa]
01001   real, dimension(size(T)) :: rho     ! In situ density [kg m-3]
01002   real, dimension(size(T)) :: dRho_dT ! Derivative of density with temperature [kg m-3 degC-1]
01003   real, dimension(size(T)) :: dRho_dS ! Derivative of density with salinity [kg m-3 ppt-1]
01004   integer :: j
01005
01006   if (.not.associated(eos)) call mom_error(fatal, &
01007     "calculate_spec_vol_derivs_array called with an unassociated EOS_type EOS.")
01008
01009   select case (eos%form_of_EOS)
01010     case (eos_linear)
01011       call calculate_specvol_derivs_linear(t, s, pressure, dsv_dt, dsv_ds, start, &
01012                                            npts, eos%Rho_T0_S0, eos%dRho_dT, eos%dRho_dS)
01013     case (eos_unesco)
01014       call calculate_density_unesco(t, s, pressure, rho, start, npts)
01015       call calculate_density_derivs_unesco(t, s, pressure, drho_dt, drho_ds, start, npts)
01016       do j=start,start+npts-1
01017         dsv_dt(j) = -drho_dt(j)/(rho(j)**2)
01018         dsv_ds(j) = -drho_ds(j)/(rho(j)**2)
01019       enddo
01020     case (eos_wright)
01021       call calculate_specvol_derivs_wright(t, s, pressure, dsv_dt, dsv_ds, start, npts)
01022     case (eos_teos10)
01023       call calculate_specvol_derivs_teos10(t, s, pressure, dsv_dt, dsv_ds, start, npts)
01024     case (eos_nemo)
01025       call calculate_density_nemo(t, s, pressure, rho, start, npts)
01026       call calculate_density_derivs_nemo(t, s, pressure, drho_dt, drho_ds, start, npts)
01027       do j=start,start+npts-1
01028         dsv_dt(j) = -drho_dt(j)/(rho(j)**2)
01029         dsv_ds(j) = -drho_ds(j)/(rho(j)**2)
01030       enddo
01031     case default
01032       call mom_error(fatal, "calculate_spec_vol_derivs_array: EOS%form_of_EOS is not valid.")
01033   end select
01034
01035 end subroutine calculate_spec_vol_derivs_array
01036
01037 !> Calls the appropriate subroutine to calculate specific volume derivatives for 1-d array inputs,
01038 !! potentially limiting the domain of indices that are worked on.
01039 subroutine calc_spec_vol_derivs_1d(T, S, pressure, dSV_dT, dSV_dS, EOS, dom, scale)
01040   real, dimension(:), intent(in)    :: T        !< Potential temperature referenced to the surface [degC]
01041   real, dimension(:), intent(in)    :: S        !< Salinity [ppt]
01042   real, dimension(:), intent(in)    :: pressure !< Pressure [R L2 T-2 ~> Pa]
01043   real, dimension(:), intent(inout) :: dSV_dT   !< The partial derivative of specific volume with potential
01044                                                 !! temperature [R-1 degC-1 ~> m3 kg-1 degC-1]
01045   real, dimension(:), intent(inout) :: dSV_dS   !< The partial derivative of specific volume with salinity
01046                                                 !! [R-1 ppt-1 ~> m3 kg-1 ppt-1]
01047   type(eos_type),     pointer       :: EOS      !< Equation of state structure
01048   integer, dimension(2), optional, intent(in) :: dom   !< The domain of indices to work on, taking
01049                                                        !! into account that arrays start at 1.
01050   real,                  optional, intent(in) :: scale !< A multiplicative factor by which to scale
       specific
01051                                                !! volume in combination with scaling given by US [various]
01052
01053   ! Local variables
01054   real, dimension(size(dSV_dT)) :: press   ! Pressure converted to [Pa]
01055   real :: spv_scale ! A factor to convert specific volume from m3 kg-1 to the desired units [kg R-1 m-3 ~>
```

```
      1]
01056   real :: p_scale   ! A factor to convert pressure to units of Pa [Pa T2 R-1 L-2 ~> 1]
01057   integer :: i, is, ie, npts
01058
01059   if (.not.associated(eos)) call mom_error(fatal, &
01060     "calculate_spec_vol_derivs_1d called with an unassociated EOS_type EOS.")
01061
01062   if (present(dom)) then
01063     is = dom(1) ; ie = dom(2) ; npts = 1 + ie - is
01064   else
01065     is = 1 ; ie = size(dsv_dt) ; npts = 1 + ie - is
01066   endif
01067   p_scale = eos%RL2_T2_to_Pa
01068
01069   if (p_scale == 1.0) then
01070     call calculate_spec_vol_derivs_array(t, s, pressure, dsv_dt, dsv_ds, is,
      npts, eos)
01071   else
01072     do i=is,ie ; press(i) = p_scale * pressure(i) ; enddo
01073     call calculate_spec_vol_derivs_array(t, s, press, dsv_dt, dsv_ds, is,
     npts, eos)
01074   endif
01075
01076   spv_scale = eos%R_to_kg_m3
01077   if (present(scale)) spv_scale = spv_scale * scale
01078   if (spv_scale /= 1.0) then ; do i=is,ie
01079     dsv_dt(i) = spv_scale * dsv_dt(i)
01080     dsv_ds(i) = spv_scale * dsv_ds(i)
01081   enddo ; endif
01082
01083 end subroutine calc_spec_vol_derivs_1d
01084
01085
01086 !> Calls the appropriate subroutine to calculate the density and compressibility for 1-D array
01087 !! inputs.  If US is present, the units of the inputs and outputs are rescaled.
01088 subroutine calculate_compress_array(T, S, press, rho, drho_dp, start, npts, EOS)
01089   real, dimension(:), intent(in)  :: T        !< Potential temperature referenced to the surface [degC]
01090   real, dimension(:), intent(in)  :: S        !< Salinity [PSU]
01091   real, dimension(:), intent(in)  :: press    !< Pressure [Pa] or [R L2 T-2 ~> Pa]
01092   real, dimension(:), intent(inout) :: rho      !< In situ density [kg m-3] or [R ~> kg m-3]
01093   real, dimension(:), intent(inout) :: drho_dp  !< The partial derivative of density with pressure
01094                                                 !! (also the inverse of the square of sound speed)
01095                                                 !! [s2 m-2] or [T2 L-2]
01096   integer,            intent(in)  :: start    !< Starting index within the array
01097   integer,            intent(in)  :: npts     !< The number of values to calculate
01098   type(eos_type),     pointer     :: EOS      !< Equation of state structure
01099
01100   ! Local variables
01101   real, dimension(size(press)) :: pressure  ! Pressure converted to [Pa]
01102   integer :: i, is, ie
01103
01104   if (.not.associated(eos)) call mom_error(fatal, &
01105     "calculate_compress called with an unassociated EOS_type EOS.")
01106
01107   is = start ; ie = is + npts - 1
01108   do i=is,ie ; pressure(i) = eos%RL2_T2_to_Pa * press(i) ; enddo
01109
01110   select case (eos%form_of_EOS)
01111     case (eos_linear)
01112       call calculate_compress_linear(t, s, pressure, rho, drho_dp, start, npts, &
01113                                      eos%Rho_T0_S0, eos%dRho_dT, eos%dRho_dS)
01114     case (eos_unesco)
01115       call calculate_compress_unesco(t, s, pressure, rho, drho_dp, start, npts)
01116     case (eos_wright)
01117       call calculate_compress_wright(t, s, pressure, rho, drho_dp, start, npts)
01118     case (eos_teos10)
01119       call calculate_compress_teos10(t, s, pressure, rho, drho_dp, start, npts)
01120     case (eos_nemo)
01121       call calculate_compress_nemo(t, s, pressure, rho, drho_dp, start, npts)
01122     case default
01123       call mom_error(fatal, "calculate_compress: EOS%form_of_EOS is not valid.")
01124   end select
01125
01126   if (eos%kg_m3_to_R /= 1.0) then ; do i=is,ie
01127     rho(i) = eos%kg_m3_to_R * rho(i)
01128   enddo ; endif
01129   if (eos%L_T_to_m_s /= 1.0) then ; do i=is,ie
01130     drho_dp(i) = eos%L_T_to_m_s**2 * drho_dp(i)
01131   enddo ; endif
01132
01133 end subroutine calculate_compress_array
01134
01135 !> Calculate density and compressibility for a scalar. This just promotes the scalar to an array
01136 !! with a singleton dimension and calls calculate_compress_array.  If US is present, the units of
01137 !! the inputs and outputs are rescaled.
01138 subroutine calculate_compress_scalar(T, S, pressure, rho, drho_dp, EOS)
01139   real, intent(in)        :: T        !< Potential temperature referenced to the surface [degC]
```

```
01140   real, intent(in)         :: S        !< Salinity [ppt]
01141   real, intent(in)         :: pressure !< Pressure [Pa] or [R L2 T-2 ~> Pa]
01142   real, intent(out)        :: rho      !< In situ density [kg m-3] or [R ~> kg m-3]
01143   real, intent(out)        :: drho_dp  !< The partial derivative of density with pressure (also the
01144                                        !! inverse of the square of sound speed) [s2 m-2] or [T2 L-2]
01145   type(eos_type), pointer :: EOS       !< Equation of state structure
01146
01147   ! Local variables
01148   real, dimension(1) :: Ta, Sa, pa, rhoa, drho_dpa
01149
01150   if (.not.associated(eos)) call mom_error(fatal, &
01151     "calculate_compress called with an unassociated EOS_type EOS.")
01152   ta(1) = t ; sa(1) = s; pa(1) = pressure
01153
01154   call calculate_compress_array(ta, sa, pa, rhoa, drho_dpa, 1, 1, eos)
01155   rho = rhoa(1) ; drho_dp = drho_dpa(1)
01156
01157 end subroutine calculate_compress_scalar
01158
01159
01160 !> This subroutine returns a two point integer array indicating the domain of i-indices
01161 !! to work on in EOS calls based on information from a hor_index type
01162 function eos_domain(HI, halo) result(EOSdom)
01163   type(hor_index_type), intent(in)  :: HI    !< The horizontal index structure
01164   integer,    optional, intent(in)  :: halo  !< The halo size to work on; missing is equivalent to 0.
01165   integer, dimension(2) :: EOSdom   !< The index domain that the EOS will work on, taking into account
01166                                     !! that the arrays inside the EOS routines will start at 1.
01167
01168   ! Local variables
01169   integer :: halo_sz
01170
01171   halo_sz = 0 ; if (present(halo)) halo_sz = halo
01172
01173   eosdom(1) = hi%isc - (hi%isd-1) - halo_sz
01174   eosdom(2) = hi%iec - (hi%isd-1) + halo_sz
01175
01176 end function eos_domain
01177
01178
01179 !> Calls the appropriate subroutine to calculate analytical and nearly-analytical
01180 !! integrals in pressure across layers of geopotential anomalies, which are
01181 !! required for calculating the finite-volume form pressure accelerations in a
01182 !! non-Boussinesq model.  There are essentially no free assumptions, apart from the
01183 !! use of Boole's rule to do the horizontal integrals, and from a truncation in the
01184 !! series for log(1-eps/1+eps) that assumes that |eps| < 0.34.
01185 subroutine analytic_int_specific_vol_dp(T, S, p_t, p_b, alpha_ref, HI, EOS, &
01186                                         dza, intp_dza, intx_dza, inty_dza, halo_size, &
01187                                         bathyP, dP_tiny, useMassWghtInterp)
01188   type(hor_index_type), intent(in)  :: HI  !< The horizontal index structure
01189   real, dimension(HI%isd:HI%ied,HI%jsd:HI%jed), &
01190                         intent(in)  :: T   !< Potential temperature referenced to the surface [degC]
01191   real, dimension(HI%isd:HI%ied,HI%jsd:HI%jed), &
01192                         intent(in)  :: S   !< Salinity [ppt]
01193   real, dimension(HI%isd:HI%ied,HI%jsd:HI%jed), &
01194                         intent(in)  :: p_t !< Pressure at the top of the layer [R L2 T-2 ~> Pa] or [Pa]
01195   real, dimension(HI%isd:HI%ied,HI%jsd:HI%jed), &
01196                         intent(in)  :: p_b !< Pressure at the bottom of the layer [R L2 T-2 ~> Pa] or [Pa]
01197   real,                 intent(in)  :: alpha_ref !< A mean specific volume that is subtracted out
01198                             !! to reduce the magnitude of each of the integrals [R-1 ~> m3 kg-1]
01199                             !! The calculation is mathematically identical with different values of
01200                             !! alpha_ref, but this reduces the effects of roundoff.
01201   type(eos_type),       pointer     :: EOS !< Equation of state structure
01202   real, dimension(HI%isd:HI%ied,HI%jsd:HI%jed), &
01203                         intent(inout) :: dza !< The change in the geopotential anomaly across
01204                             !! the layer [L2 T-2 ~> m2 s-2] or [m2 s-2]
01205   real, dimension(HI%isd:HI%ied,HI%jsd:HI%jed), &
01206               optional, intent(inout) :: intp_dza !< The integral in pressure through the layer of the
01207                             !! geopotential anomaly relative to the anomaly at the bottom of the
01208                             !! layer [R L4 T-4 ~> Pa m2 s-2] or [Pa m2 s-2]
01209   real, dimension(HI%IsdB:HI%IedB,HI%jsd:HI%jed), &
01210               optional, intent(inout) :: intx_dza !< The integral in x of the difference between the
01211                             !! geopotential anomaly at the top and bottom of the layer divided by
01212                             !! the x grid spacing [L2 T-2 ~> m2 s-2] or [m2 s-2]
01213   real, dimension(HI%isd:HI%ied,HI%JsdB:HI%JedB), &
01214               optional, intent(inout) :: inty_dza !< The integral in y of the difference between the
01215                             !! geopotential anomaly at the top and bottom of the layer divided by
01216                             !! the y grid spacing [L2 T-2 ~> m2 s-2] or [m2 s-2]
01217   integer,    optional, intent(in)  :: halo_size !< The width of halo points on which to calculate dza.
01218   real, dimension(HI%isd:HI%ied,HI%jsd:HI%jed), &
01219               optional, intent(in)  :: bathyP  !< The pressure at the bathymetry [R L2 T-2 ~> Pa] or [Pa]
01220   real,       optional, intent(in)  :: dP_tiny !< A miniscule pressure change with
01221                             !! the same units as p_t [R L2 T-2 ~> Pa] or [Pa]
01222   logical,    optional, intent(in)  :: useMassWghtInterp !< If true, uses mass weighting
01223                             !! to interpolate T/S for top and bottom integrals.
01224   ! Local variables
01225   real :: pres_scale    ! A unit conversion factor from the rescaled units of pressure to Pa [Pa T2 R-1 L-2
      ~> 1]
```

```
01226   real :: SV_scale      ! A multiplicative factor by which to scale specific
01227                         ! volume from m3 kg-1 to the desired units [kg m-3 R-1 ~> 1]
01228
01229   if (.not.associated(eos)) call mom_error(fatal, &
01230     "int_specific_vol_dp called with an unassociated EOS_type EOS.")
01231
01232   ! We should never reach this point with quadrature. EOS_quadrature indicates that numerical
01233   ! integration be used instead of analytic. This is a safety check.
01234   if (eos%EOS_quadrature) call mom_error(fatal, "EOS_quadrature is set!")
01235
01236   select case (eos%form_of_EOS)
01237     case (eos_linear)
01238       call int_spec_vol_dp_linear(t, s, p_t, p_b, alpha_ref, hi, eos%kg_m3_to_R*eos%Rho_T0_S0, &
01239                                   eos%kg_m3_to_R*eos%dRho_dT, eos%kg_m3_to_R*eos%dRho_dS, dza, &
01240                                   intp_dza, intx_dza, inty_dza, halo_size, &
01241                                   bathyp, dp_tiny, usemasswghtinterp)
01242     case (eos_wright)
01243       call int_spec_vol_dp_wright(t, s, p_t, p_b, alpha_ref, hi, dza, intp_dza, intx_dza, &
01244                                   inty_dza, halo_size, bathyp, dp_tiny, usemasswghtinterp, &
01245                                   sv_scale=eos%R_to_kg_m3, pres_scale=eos%RL2_T2_to_Pa)
01246     case default
01247       call mom_error(fatal, "No analytic integration option is available with this EOS!")
01248   end select
01249
01250 end subroutine analytic_int_specific_vol_dp
01251
01252 !> This subroutine calculates analytical and nearly-analytical integrals of
01253 !! pressure anomalies across layers, which are required for calculating the
01254 !! finite-volume form pressure accelerations in a Boussinesq model.
01255 subroutine analytic_int_density_dz(T, S, z_t, z_b, rho_ref, rho_0, G_e, HI, EOS,
       dpa, &
01256                                    intz_dpa, intx_dpa, inty_dpa, bathyT, dz_neglect, useMassWghtInterp)
01257   type(hor_index_type), intent(in)  :: HI !< Ocean horizontal index structure
01258   real, dimension(HI%isd:HI%ied,HI%jsd:HI%jed), &
01259                         intent(in)  :: T   !< Potential temperature referenced to the surface [degC]
01260   real, dimension(HI%isd:HI%ied,HI%jsd:HI%jed), &
01261                         intent(in)  :: S   !< Salinity [ppt]
01262   real, dimension(HI%isd:HI%ied,HI%jsd:HI%jed), &
01263                         intent(in)  :: z_t !< Height at the top of the layer in depth units [Z ~> m]
01264   real, dimension(HI%isd:HI%ied,HI%jsd:HI%jed), &
01265                         intent(in)  :: z_b !< Height at the bottom of the layer [Z ~> m]
01266   real,                 intent(in)  :: rho_ref !< A mean density [R ~> kg m-3] or [kg m-3], that is
01267                                                !! subtracted out to reduce the magnitude of each of the
01268                                                !! integrals.
01269   real,                 intent(in)  :: rho_0 !< A density [R ~> kg m-3] or [kg m-3], that is used
01270                                                !! to calculate the pressure (as p~=-z*rho_0*G_e)
01271                                                !! used in the equation of state.
01272   real,                 intent(in)  :: G_e !< The Earth's gravitational acceleration
01273                                                !! [L2 Z-1 T-2 ~> m s-2] or [m2 Z-1 s-2 ~> m s-2]
01274   type(eos_type),       pointer     :: EOS !< Equation of state structure
01275   real, dimension(HI%isd:HI%ied,HI%jsd:HI%jed), &
01276                         intent(inout) :: dpa !< The change in the pressure anomaly
01277                                                !! across the layer [R L2 T-2 ~> Pa] or [Pa]
01278   real, dimension(HI%isd:HI%ied,HI%jsd:HI%jed), &
01279               optional, intent(inout) :: intz_dpa !< The integral through the thickness of the
01280                                                !! layer of the pressure anomaly relative to the
01281                                                !! anomaly at the top of the layer [R L2 Z T-2 ~> Pa m]
01282   real, dimension(HI%IsdB:HI%IedB,HI%jsd:HI%jed), &
01283               optional, intent(inout) :: intx_dpa !< The integral in x of the difference between
01284                                                !! the pressure anomaly at the top and bottom of the
01285                                                !! layer divided by the x grid spacing [R L2 T-2 ~> Pa]
01286   real, dimension(HI%isd:HI%ied,HI%JsdB:HI%JedB), &
01287               optional, intent(inout) :: inty_dpa !< The integral in y of the difference between
01288                                                !! the pressure anomaly at the top and bottom of the
01289                                                !! layer divided by the y grid spacing [R L2 T-2 ~> Pa]
01290   real, dimension(HI%isd:HI%ied,HI%jsd:HI%jed), &
01291                 optional, intent(in)  :: bathyT !< The depth of the bathymetry [Z ~> m]
01292   real,         optional, intent(in)  :: dz_neglect !< A miniscule thickness change [Z ~> m]
01293   logical,      optional, intent(in)  :: useMassWghtInterp !< If true, uses mass weighting to
01294                                                !! interpolate T/S for top and bottom integrals.
01295   ! Local variables
01296   real :: rho_scale  ! A multiplicative factor by which to scale density from kg m-3 to the
01297                      ! desired units [R m3 kg-1 ~> 1]
01298   real :: pres_scale ! A multiplicative factor to convert pressure into Pa [Pa T2 R-1 L-2 ~> 1]
01299
01300   if (.not.associated(eos)) call mom_error(fatal, &
01301     "int_density_dz called with an unassociated EOS_type EOS.")
01302
01303   ! We should never reach this point with quadrature. EOS_quadrature indicates that numerical
01304   ! integration be used instead of analytic. This is a safety check.
01305   if (eos%EOS_quadrature) call mom_error(fatal, "EOS_quadrature is set!")
01306
01307   select case (eos%form_of_EOS)
01308     case (eos_linear)
01309       rho_scale = eos%kg_m3_to_R
01310       if (rho_scale /= 1.0) then
01311         call int_density_dz_linear(t, s, z_t, z_b, rho_ref, rho_0, g_e, hi, &
```

```
01312                         rho_scale*eos%Rho_T0_S0, rho_scale*eos%dRho_dT, rho_scale*eos%dRho_dS, &
01313                         dpa, intz_dpa, intx_dpa, inty_dpa, bathyt, dz_neglect, usemasswghtinterp)
01314       else
01315         call int_density_dz_linear(t, s, z_t, z_b, rho_ref, rho_0, g_e, hi, &
01316                         eos%Rho_T0_S0, eos%dRho_dT, eos%dRho_dS, &
01317                         dpa, intz_dpa, intx_dpa, inty_dpa, bathyt, dz_neglect, usemasswghtinterp)
01318       endif
01319     case (eos_wright)
01320       rho_scale = eos%kg_m3_to_R
01321       pres_scale = eos%RL2_T2_to_Pa
01322       if ((rho_scale /= 1.0) .or. (pres_scale /= 1.0)) then
01323         call int_density_dz_wright(t, s, z_t, z_b, rho_ref, rho_0, g_e, hi, &
01324                                    dpa, intz_dpa, intx_dpa, inty_dpa, bathyt, &
01325                                    dz_neglect, usemasswghtinterp, rho_scale, pres_scale)
01326       else
01327         call int_density_dz_wright(t, s, z_t, z_b, rho_ref, rho_0, g_e, hi, &
01328                                    dpa, intz_dpa, intx_dpa, inty_dpa, bathyt, &
01329                                    dz_neglect, usemasswghtinterp)
01330       endif
01331     case default
01332       call mom_error(fatal, "No analytic integration option is available with this EOS!")
01333   end select
01334
01335 end subroutine analytic_int_density_dz
01336
01337 !> Returns true if the equation of state is compressible (i.e. has pressure dependence)
01338 logical function query_compressible(EOS)
01339   type(eos_type), pointer :: EOS !< Equation of state structure
01340
01341   if (.not.associated(eos)) call mom_error(fatal, &
01342     "query_compressible called with an unassociated EOS_type EOS.")
01343
01344   query_compressible = eos%compressible
01345 end function query_compressible
01346
01347 !> Initializes EOS_type by allocating and reading parameters
01348 subroutine eos_init(param_file, EOS, US)
01349   type(param_file_type), intent(in) :: param_file !< Parameter file structure
01350   type(eos_type),        pointer    :: EOS !< Equation of state structure
01351   type(unit_scale_type), intent(in) :: US  !< A dimensional unit scaling type
01352   optional :: us
01353   ! Local variables
01354 #include "version_variable.h"
01355   character(len=40)  :: mdl = "MOM_EOS" ! This module's name.
01356   character(len=40)  :: tmpstr
01357
01358   if (.not.associated(eos)) call eos_allocate(eos)
01359
01360   ! Read all relevant parameters and write them to the model log.
01361   call log_version(param_file, mdl, version, "")
01362
01363   call get_param(param_file, mdl, "EQN_OF_STATE", tmpstr, &
01364                  "EQN_OF_STATE determines which ocean equation of state "//&
01365                  "should be used.  Currently, the valid choices are "//&
01366                  '"LINEAR", "UNESCO", "WRIGHT", "NEMO" and "TEOS10". '//&
01367                  "This is only used if USE_EOS is true.", default=eos_default)
01368   select case (uppercase(tmpstr))
01369     case (eos_linear_string)
01370       eos%form_of_EOS = eos_linear
01371     case (eos_unesco_string)
01372       eos%form_of_EOS = eos_unesco
01373     case (eos_wright_string)
01374       eos%form_of_EOS = eos_wright
01375     case (eos_teos10_string)
01376       eos%form_of_EOS = eos_teos10
01377     case (eos_nemo_string)
01378       eos%form_of_EOS = eos_nemo
01379     case default
01380       call mom_error(fatal, "interpret_eos_selection: EQN_OF_STATE "//&
01381                              trim(tmpstr) // "in input file is invalid.")
01382   end select
01383   call mom_mesg('interpret_eos_selection: equation of state set to "' // &
01384                 trim(tmpstr)//'"', 5)
01385
01386   if (eos%form_of_EOS == eos_linear) then
01387     eos%Compressible = .false.
01388     call get_param(param_file, mdl, "RHO_T0_S0", eos%Rho_T0_S0, &
01389                    "When EQN_OF_STATE="//trim(eos_linear_string)//", "//&
01390                    "this is the density at T=0, S=0.", units="kg m-3", &
01391                    default=1000.0)
01392     call get_param(param_file, mdl, "DRHO_DT", eos%dRho_dT, &
01393                    "When EQN_OF_STATE="//trim(eos_linear_string)//", "//&
01394                    "this is the partial derivative of density with "//&
01395                    "temperature.", units="kg m-3 K-1", default=-0.2)
01396     call get_param(param_file, mdl, "DRHO_DS", eos%dRho_dS, &
01397                    "When EQN_OF_STATE="//trim(eos_linear_string)//", "//&
01398                    "this is the partial derivative of density with "//&
```

```
01399                 "salinity.", units="kg m-3 PSU-1", default=0.8)
01400   endif
01401
01402   call get_param(param_file, mdl, "EOS_QUADRATURE", eos%EOS_quadrature, &
01403                 "If true, always use the generic (quadrature) code "//&
01404                 "code for the integrals of density.", default=.false.)
01405
01406   call get_param(param_file, mdl, "TFREEZE_FORM", tmpstr, &
01407                 "TFREEZE_FORM determines which expression should be "//&
01408                 "used for the freezing point.  Currently, the valid "//&
01409                 'choices are "LINEAR", "MILLERO_78", "TEOS10"', &
01410                 default=tfreeze_default)
01411   select case (uppercase(tmpstr))
01412     case (tfreeze_linear_string)
01413       eos%form_of_TFreeze = tfreeze_linear
01414     case (tfreeze_millero_string)
01415       eos%form_of_TFreeze = tfreeze_millero
01416     case (tfreeze_teos10_string)
01417       eos%form_of_TFreeze = tfreeze_teos10
01418     case default
01419       call mom_error(fatal, "interpret_eos_selection:  TFREEZE_FORM "//&
01420                              trim(tmpstr) // "in input file is invalid.")
01421   end select
01422
01423   if (eos%form_of_TFreeze == tfreeze_linear) then
01424     call get_param(param_file, mdl, "TFREEZE_S0_P0",eos%TFr_S0_P0, &
01425                 "When TFREEZE_FORM="//trim(tfreeze_linear_string)//", "//&
01426                 "this is the freezing potential temperature at "//&
01427                 "S=0, P=0.", units="deg C", default=0.0)
01428     call get_param(param_file, mdl, "DTFREEZE_DS",eos%dTFr_dS, &
01429                 "When TFREEZE_FORM="//trim(tfreeze_linear_string)//", "//&
01430                 "this is the derivative of the freezing potential "//&
01431                 "temperature with salinity.", &
01432                 units="deg C PSU-1", default=-0.054)
01433     call get_param(param_file, mdl, "DTFREEZE_DP",eos%dTFr_dP, &
01434                 "When TFREEZE_FORM="//trim(tfreeze_linear_string)//", "//&
01435                 "this is the derivative of the freezing potential "//&
01436                 "temperature with pressure.", &
01437                 units="deg C Pa-1", default=0.0)
01438   endif
01439
01440   if ((eos%form_of_EOS == eos_teos10 .OR. eos%form_of_EOS == eos_nemo) .AND. &
01441       eos%form_of_TFreeze /= tfreeze_teos10) then
01442       call mom_error(fatal, "interpret_eos_selection:  EOS_TEOS10 or EOS_NEMO \n" //&
01443       "should only be used along with TFREEZE_FORM = TFREEZE_TEOS10 .")
01444   endif
01445
01446   ! Unit conversions
01447   eos%m_to_Z = 1. ; if (present(us)) eos%m_to_Z = us%m_to_Z
01448   eos%kg_m3_to_R = 1. ; if (present(us)) eos%kg_m3_to_R = us%kg_m3_to_R
01449   eos%R_to_kg_m3 = 1. ; if (present(us)) eos%R_to_kg_m3 = us%R_to_kg_m3
01450   eos%RL2_T2_to_Pa = 1. ; if (present(us)) eos%RL2_T2_to_Pa = us%RL2_T2_to_Pa
01451   eos%L_T_to_m_s = 1. ; if (present(us)) eos%L_T_to_m_s = us%L_T_to_m_s
01452
01453 end subroutine eos_init
01454
01455 !> Manually initialized an EOS type (intended for unit testing of routines which need a specific EOS)
01456 subroutine eos_manual_init(EOS, form_of_EOS, form_of_TFreeze, EOS_quadrature, Compressible,
       &
01457                            Rho_T0_S0, drho_dT, dRho_dS, TFr_S0_P0, dTFr_dS, dTFr_dp)
01458   type(eos_type),    pointer    :: EOS !< Equation of state structure
01459   integer, optional, intent(in) :: form_of_EOS !< A coded integer indicating the equation of state to use.
01460   integer, optional, intent(in) :: form_of_TFreeze !< A coded integer indicating the expression for
01461                                        !! the potential temperature of the freezing point.
01462   logical, optional, intent(in) :: EOS_quadrature !< If true, always use the generic (quadrature)
01463                                        !! code for the integrals of density.
01464   logical, optional, intent(in) :: Compressible  !< If true, in situ density is a function of pressure.
01465   real   , optional, intent(in) :: Rho_T0_S0 !< Density at T=0 degC and S=0 ppt [kg m-3]
01466   real   , optional, intent(in) :: drho_dT   !< Partial derivative of density with temperature
01467                                             !! in [kg m-3 degC-1]
01468   real   , optional, intent(in) :: dRho_dS   !< Partial derivative of density with salinity
01469                                             !! in [kg m-3 ppt-1]
01470   real   , optional, intent(in) :: TFr_S0_P0 !< The freezing potential temperature at S=0, P=0 [degC]
01471   real   , optional, intent(in) :: dTFr_dS   !< The derivative of freezing point with salinity
01472                                             !! in [degC ppt-1]
01473   real   , optional, intent(in) :: dTFr_dp   !< The derivative of freezing point with pressure
01474                                             !! in [degC Pa-1]
01475
01476   if (present(form_of_eos    ))  eos%form_of_EOS     = form_of_eos
01477   if (present(form_of_tfreeze))  eos%form_of_TFreeze = form_of_tfreeze
01478   if (present(eos_quadrature ))  eos%EOS_quadrature  = eos_quadrature
01479   if (present(compressible   ))  eos%Compressible    = compressible
01480   if (present(rho_t0_s0      ))  eos%Rho_T0_S0       = rho_t0_s0
01481   if (present(drho_dt        ))  eos%drho_dT         = drho_dt
01482   if (present(drho_ds        ))  eos%dRho_dS         = drho_ds
01483   if (present(tfr_s0_p0      ))  eos%TFr_S0_P0       = tfr_s0_p0
01484   if (present(dtfr_ds        ))  eos%dTFr_dS         = dtfr_ds
```

```
01485   if (present(dtfr_dp        ))  eos%dTFr_dp         = dtfr_dp
01486
01487 end subroutine eos_manual_init
01488
01489 !> Allocates EOS_type
01490 subroutine eos_allocate(EOS)
01491   type(eos_type), pointer :: EOS !< Equation of state structure
01492
01493   if (.not.associated(eos)) allocate(eos)
01494 end subroutine eos_allocate
01495
01496 !> Deallocates EOS_type
01497 subroutine eos_end(EOS)
01498   type(eos_type), pointer :: EOS !< Equation of state structure
01499
01500   if (associated(eos)) deallocate(eos)
01501 end subroutine eos_end
01502
01503 !> Set equation of state structure (EOS) to linear with given coefficients
01504 !!
01505 !! \note This routine is primarily for testing and allows a local copy of the
01506 !! EOS_type (EOS argument) to be set to use the linear equation of state
01507 !! independent from the rest of the model.
01508 subroutine eos_use_linear(Rho_T0_S0, dRho_dT, dRho_dS, EOS, use_quadrature)
01509   real,              intent(in) :: Rho_T0_S0 !< Density at T=0 degC and S=0 ppt [kg m-3]
01510   real,              intent(in) :: dRho_dT   !< Partial derivative of density with temperature [kg m-3
       degC-1]
01511   real,              intent(in) :: dRho_dS   !< Partial derivative of density with salinity [kg m-3 ppt-1]
01512   logical, optional, intent(in) :: use_quadrature !< If true, always use the generic (quadrature)
01513                                                   !! code for the integrals of density.
01514   type(eos_type),    pointer    :: EOS       !< Equation of state structure
01515
01516   if (.not.associated(eos)) call mom_error(fatal, &
01517     "MOM_EOS.F90: EOS_use_linear() called with an unassociated EOS_type EOS.")
01518
01519   eos%form_of_EOS = eos_linear
01520   eos%Compressible = .false.
01521   eos%Rho_T0_S0 = rho_t0_s0
01522   eos%dRho_dT = drho_dt
01523   eos%dRho_dS = drho_ds
01524   eos%EOS_quadrature = .false.
01525   if (present(use_quadrature)) eos%EOS_quadrature = use_quadrature
01526
01527 end subroutine eos_use_linear
01528
01529
01530 !> Convert T&S to Absolute Salinity and Conservative Temperature if using TEOS10
01531 subroutine convert_temp_salt_for_teos10(T, S, HI, kd, mask_z, EOS)
01532   integer,                intent(in)    :: kd  !< The number of layers to work on
01533   type(hor_index_type),   intent(in)    :: HI       !< The horizontal index structure
01534   real, dimension(HI%isd:HI%ied,HI%jsd:HI%jed,kd), &
01535                           intent(inout) :: T   !< Potential temperature referenced to the surface [degC]
01536   real, dimension(HI%isd:HI%ied,HI%jsd:HI%jed,kd), &
01537                           intent(inout) :: S   !< Salinity [ppt]
01538   real, dimension(HI%isd:HI%ied,HI%jsd:HI%jed,kd), &
01539                           intent(in)    :: mask_z !< 3d mask regulating which points to convert.
01540   type(eos_type),         pointer       :: EOS !< Equation of state structure
01541
01542   integer :: i, j, k
01543   real :: gsw_sr_from_sp, gsw_ct_from_pt, gsw_sa_from_sp
01544   real :: p
01545
01546   if (.not.associated(eos)) call mom_error(fatal, &
01547     "convert_temp_salt_to_TEOS10 called with an unassociated EOS_type EOS.")
01548
01549   if ((eos%form_of_EOS /= eos_teos10) .and. (eos%form_of_EOS /=
      eos_nemo)) return
01550
01551   do k=1,kd ; do j=hi%jsc,hi%jec ; do i=hi%isc,hi%iec
01552     if (mask_z(i,j,k) >= 1.0) then
01553      s(i,j,k) = gsw_sr_from_sp(s(i,j,k))
01554 !     Get absolute salinity from practical salinity, converting pressures from Pascal to dbar.
01555 !     If this option is activated, pressure will need to be added as an argument, and it should be
01556 !     moved out into module that is not shared between components, where the ocean_grid can be used.
01557 !     S(i,j,k) = gsw_sa_from_sp(S(i,j,k),pres(i,j,k)*1.0e-4,G%geoLonT(i,j),G%geoLatT(i,j))
01558      t(i,j,k) = gsw_ct_from_pt(s(i,j,k), t(i,j,k))
01559     endif
01560   enddo ; enddo ; enddo
01561 end subroutine convert_temp_salt_for_teos10
01562
01563 !> Return value of EOS_quadrature
01564 logical function eos_quadrature(EOS)
01565   type(eos_type),    pointer     :: EOS !< Equation of state structure
01566
01567   eos_quadrature  = eos%EOS_quadrature
01568
01569 end function eos_quadrature
```

```
01570
01571 !> Extractor routine for the EOS type if the members need to be accessed outside this module
01572 subroutine extract_member_eos(EOS, form_of_EOS, form_of_TFreeze, EOS_quadrature,
      Compressible, &
01573                                Rho_T0_S0, drho_dT, dRho_dS, TFr_S0_P0, dTFr_dS, dTFr_dp)
01574   type(eos_type),    pointer     :: EOS !< Equation of state structure
01575   integer, optional, intent(out) :: form_of_EOS !< A coded integer indicating the equation of state to use.
01576   integer, optional, intent(out) :: form_of_TFreeze !< A coded integer indicating the expression for
01577                                        !! the potential temperature of the freezing point.
01578   logical, optional, intent(out) :: EOS_quadrature !< If true, always use the generic (quadrature)
01579                                        !! code for the integrals of density.
01580   logical, optional, intent(out) :: Compressible !< If true, in situ density is a function of pressure.
01581   real   , optional, intent(out) :: Rho_T0_S0 !< Density at T=0 degC and S=0 ppt [kg m-3]
01582   real   , optional, intent(out) :: drho_dT   !< Partial derivative of density with temperature
01583                                                !! in [kg m-3 degC-1]
01584   real   , optional, intent(out) :: dRho_dS   !< Partial derivative of density with salinity
01585                                                !! in [kg m-3 ppt-1]
01586   real   , optional, intent(out) :: TFr_S0_P0 !< The freezing potential temperature at S=0, P=0 [degC]
01587   real   , optional, intent(out) :: dTFr_dS   !< The derivative of freezing point with salinity
01588                                                !! [degC PSU-1]
01589   real   , optional, intent(out) :: dTFr_dp   !< The derivative of freezing point with pressure
01590                                                !! [degC Pa-1]
01591
01592   if (present(form_of_eos    ))  form_of_eos     = eos%form_of_EOS
01593   if (present(form_of_tfreeze))  form_of_tfreeze = eos%form_of_TFreeze
01594   if (present(eos_quadrature ))  eos_quadrature  = eos%EOS_quadrature
01595   if (present(compressible   ))  compressible    = eos%Compressible
01596   if (present(rho_t0_s0      ))  rho_t0_s0       = eos%Rho_T0_S0
01597   if (present(drho_dt        ))  drho_dt         = eos%drho_dT
01598   if (present(drho_ds        ))  drho_ds         = eos%dRho_dS
01599   if (present(tfr_s0_p0      ))  tfr_s0_p0       = eos%TFr_S0_P0
01600   if (present(dtfr_ds        ))  dtfr_ds         = eos%dTFr_dS
01601   if (present(dtfr_dp        ))  dtfr_dp         = eos%dTFr_dp
01602
01603 end subroutine extract_member_eos
01604
01605 end module mom_eos
01606
01607 !> \namespace mom_eos
01608 !!
01609 !! The MOM_EOS module is a wrapper for various equations of state (e.g. Linear,
01610 !! Wright, UNESCO) and provides a uniform interface to the rest of the model
01611 !! independent of which equation of state is being used.
```

44.27 /home/cermak/src/MOM6/src/framework/_Diagnostics.dox File Reference

44.28 /home/cermak/src/MOM6/src/framework/_Domain_decomposition.dox File Reference

44.29 /home/cermak/src/MOM6/src/framework/_Global_grids.dox File Reference

44.30 /home/cermak/src/MOM6/src/framework/_Horizontal_indexing.dox File Reference

44.31 /home/cermak/src/MOM6/src/framework/_Parallel_IO.dox File Reference

44.32 /home/cermak/src/MOM6/src/framework/_Regional_grids.dox File Reference

44.33 /home/cermak/src/MOM6/src/framework/_Runtime_parameter_system.dox File Reference

44.34 /home/cermak/src/MOM6/src/framework/_Testing.dox File Reference

44.35 /home/cermak/src/MOM6/src/framework/_Vertical_grids.dox File Reference

44.36 /home/cermak/src/MOM6/src/framework/MOM_unit_scaling.F90 File Reference

```
#include "version_variable.h"
```

Data Types

- type mom_unit_scaling::unit_scale_type

  Describes various unit conversion factors.

Modules

- module mom_unit_scaling

  Provides a transparent unit rescaling type to facilitate dimensional consistency testing.

Functions/Subroutines

- subroutine, public mom_unit_scaling::unit_scaling_init (param_file, US)

  Allocates and initializes the ocean model unit scaling type.
- subroutine, public mom_unit_scaling::fix_restart_unit_scaling (US)

  Set the unit scaling factors for output to restart files to the unit scaling factors for this run.
- subroutine, public mom_unit_scaling::unit_scaling_end (US)

  Deallocates a unit scaling structure.

44.37 MOM_unit_scaling.F90

```
00001 !> Provides a transparent unit rescaling type to facilitate dimensional consistency testing
00002 module mom_unit_scaling
00003
00004 ! This file is part of MOM6. See LICENSE.md for the license.
00005
00006 use mom_error_handler, only : mom_error, mom_mesg, fatal
00007 use mom_file_parser, only : get_param, log_param, log_version, param_file_type
00008
00009 implicit none ; private
00010
00011 public unit_scaling_init, unit_scaling_end,
      fix_restart_unit_scaling
00012
00013 !> Describes various unit conversion factors
00014 type, public :: unit_scale_type
00015   real :: m_to_z !< A constant that translates distances in meters to the units of depth.
00016   real :: z_to_m !< A constant that translates distances in the units of depth to meters.
00017   real :: m_to_l !< A constant that translates lengths in meters to the units of horizontal lengths.
00018   real :: l_to_m !< A constant that translates lengths in the units of horizontal lengths to meters.
00019   real :: s_to_t !< A constant that translates time intervals in seconds to the units of time.
00020   real :: t_to_s !< A constant that translates the units of time to seconds.
00021   real :: r_to_kg_m3 !< A constant that translates the units of density to kilograms per meter cubed.
00022   real :: kg_m3_to_r !< A constant that translates kilograms per meter cubed to the units of density.
00023   real :: q_to_j_kg  !< A constant that translates the units of enthalpy to Joules per kilogram.
00024   real :: j_kg_to_q  !< A constant that translates Joules per kilogram to the units of enthalpy.
```

```
00025
00026   ! These are useful combinations of the fundamental scale conversion factors above.
00027   real :: z_to_l          !< Convert vertical distances to lateral lengths
00028   real :: l_to_z          !< Convert lateral lengths to vertical distances
00029   real :: l_t_to_m_s      !< Convert lateral velocities from L T-1 to m s-1.
00030   real :: m_s_to_l_t      !< Convert lateral velocities from m s-1 to L T-1.
00031   real :: l_t2_to_m_s2    !< Convert lateral accelerations from L T-2 to m s-2.
00032   real :: z2_t_to_m2_s    !< Convert vertical diffusivities from Z2 T-1 to m2 s-1.
00033   real :: m2_s_to_z2_t    !< Convert vertical diffusivities from m2 s-1 to Z2 T-1.
00034   real :: w_m2_to_qrz_t   !< Convert heat fluxes from W m-2 to Q R Z T-1.
00035   real :: qrz_t_to_w_m2   !< Convert heat fluxes from Q R Z T-1 to W m-2.
00036   ! Not used enough:  real :: kg_m2_to_RZ   !< Convert mass loads from kg m-2 to R Z.
00037   real :: rz_to_kg_m2     !< Convert mass loads from R Z to kg m-2.
00038   real :: kg_m2s_to_rz_t  !< Convert mass fluxes from kg m-2 s-1 to R Z T-1.
00039   real :: rz_t_to_kg_m2s  !< Convert mass fluxes from R Z T-1 to kg m-2 s-1.
00040   real :: rz3_t3_to_w_m2  !< Convert turbulent kinetic energy fluxes from R Z3 T-3 to W m-2.
00041   real :: w_m2_to_rz3_t3  !< Convert turbulent kinetic energy fluxes from W m-2 to R Z3 T-3.
00042   real :: rl2_t2_to_pa    !< Convert pressures from R L2 T-2 to Pa.
00043   ! Not used enough:  real :: Pa_to_RL2_T2    !< Convert pressures from Pa to R L2 T-2.
00044
00045   ! These are used for changing scaling across restarts.
00046   real :: m_to_z_restart = 0.0 !< A copy of the m_to_Z that is used in restart files.
00047   real :: m_to_l_restart = 0.0 !< A copy of the m_to_L that is used in restart files.
00048   real :: s_to_t_restart = 0.0 !< A copy of the s_to_T that is used in restart files.
00049   real :: kg_m3_to_r_restart = 0.0 !< A copy of the kg_m3_to_R that is used in restart files.
00050   real :: j_kg_to_q_restart = 0.0 !< A copy of the J_kg_to_Q that is used in restart files.
00051 end type unit_scale_type
00052
00053 contains
00054
00055 !> Allocates and initializes the ocean model unit scaling type
00056 subroutine unit_scaling_init( param_file, US )
00057   type(param_file_type), optional, intent(in) :: param_file !< Parameter file handle/type
00058   type(unit_scale_type), optional, pointer    :: US         !< A dimensional unit scaling
      type
00059
00060   ! This routine initializes a unit_scale_type structure (US).
00061
00062   ! Local variables
00063   integer :: Z_power, L_power, T_power, R_power, Q_power
00064   real    :: Z_rescale_factor, L_rescale_factor, T_rescale_factor, R_rescale_factor, Q_rescale_factor
00065   ! This include declares and sets the variable "version".
00066 # include "version_variable.h"
00067   character(len=16) :: mdl = "MOM_unit_scaling"
00068
00069   if (.not.present(us)) return
00070
00071   if (associated(us)) call mom_error(fatal, &
00072      'unit_scaling_init: called with an associated US pointer.')
00073   allocate(us)
00074
00075   if (present(param_file)) then
00076     ! Read all relevant parameters and write them to the model log.
00077     call log_version(param_file, mdl, version, &
00078                 "Parameters for doing unit scaling of variables.", debugging=.true.)
00079     call get_param(param_file, mdl, "Z_RESCALE_POWER", z_power, &
00080                 "An integer power of 2 that is used to rescale the model's "//&
00081                 "internal units of depths and heights.  Valid values range from -300 to 300.", &
00082                 units="nondim", default=0, debuggingparam=.true.)
00083     call get_param(param_file, mdl, "L_RESCALE_POWER", l_power, &
00084                 "An integer power of 2 that is used to rescale the model's "//&
00085                 "internal units of lateral distances.  Valid values range from -300 to 300.", &
00086                 units="nondim", default=0, debuggingparam=.true.)
00087     call get_param(param_file, mdl, "T_RESCALE_POWER", t_power, &
00088                 "An integer power of 2 that is used to rescale the model's "//&
00089                 "internal units of time.  Valid values range from -300 to 300.", &
00090                 units="nondim", default=0, debuggingparam=.true.)
00091     call get_param(param_file, mdl, "R_RESCALE_POWER", r_power, &
00092                 "An integer power of 2 that is used to rescale the model's "//&
00093                 "internal units of density.  Valid values range from -300 to 300.", &
00094                 units="nondim", default=0, debuggingparam=.true.)
00095     call get_param(param_file, mdl, "Q_RESCALE_POWER", q_power, &
00096                 "An integer power of 2 that is used to rescale the model's "//&
00097                   "internal units of heat content.  Valid values range from -300 to 300.", &
00098                   units="nondim", default=0, debuggingparam=.true.)
00099   else
00100     z_power = 0 ; l_power = 0 ; t_power = 0 ; r_power = 0 ; q_power = 0
00101   endif
00102
00103   if (abs(z_power) > 300) call mom_error(fatal, "unit_scaling_init: "//&
00104                 "Z_RESCALE_POWER is outside of the valid range of -300 to 300.")
00105   if (abs(l_power) > 300) call mom_error(fatal, "unit_scaling_init: "//&
00106                 "L_RESCALE_POWER is outside of the valid range of -300 to 300.")
00107   if (abs(t_power) > 300) call mom_error(fatal, "unit_scaling_init: "//&
00108                 "T_RESCALE_POWER is outside of the valid range of -300 to 300.")
00109   if (abs(r_power) > 300) call mom_error(fatal, "unit_scaling_init: "//&
00110                 "R_RESCALE_POWER is outside of the valid range of -300 to 300.")
```

```
00111   if (abs(q_power) > 300) call mom_error(fatal, "unit_scaling_init: "//&
00112                 "Q_RESCALE_POWER is outside of the valid range of -300 to 300.")
00113
00114   z_rescale_factor = 1.0
00115   if (z_power /= 0) z_rescale_factor = 2.0**z_power
00116   us%Z_to_m = 1.0 * z_rescale_factor
00117   us%m_to_Z = 1.0 / z_rescale_factor
00118
00119   l_rescale_factor = 1.0
00120   if (l_power /= 0) l_rescale_factor = 2.0**l_power
00121   us%L_to_m = 1.0 * l_rescale_factor
00122   us%m_to_L = 1.0 / l_rescale_factor
00123
00124   t_rescale_factor = 1.0
00125   if (t_power /= 0) t_rescale_factor = 2.0**t_power
00126   us%T_to_s = 1.0 * t_rescale_factor
00127   us%s_to_T = 1.0 / t_rescale_factor
00128
00129   r_rescale_factor = 1.0
00130   if (r_power /= 0) r_rescale_factor = 2.0**r_power
00131   us%R_to_kg_m3 = 1.0 * r_rescale_factor
00132   us%kg_m3_to_R = 1.0 / r_rescale_factor
00133
00134   q_rescale_factor = 1.0
00135   if (q_power /= 0) q_rescale_factor = 2.0**q_power
00136   us%Q_to_J_kg = 1.0 * q_rescale_factor
00137   us%J_kg_to_Q = 1.0 / q_rescale_factor
00138
00139   ! These are useful combinations of the fundamental scale conversion factors set above.
00140   us%Z_to_L = us%Z_to_m * us%m_to_L
00141   us%L_to_Z = us%L_to_m * us%m_to_Z
00142   ! Horizontal velocities:
00143   us%L_T_to_m_s = us%L_to_m * us%s_to_T
00144   us%m_s_to_L_T = us%m_to_L * us%T_to_s
00145   ! Horizontal accelerations:
00146   us%L_T2_to_m_s2 = us%L_to_m * us%s_to_T**2
00147     ! It does not look like US%m_s2_to_L_T2 would be used, so it does not exist.
00148   ! Vertical diffusivities and viscosities:
00149   us%Z2_T_to_m2_s = us%Z_to_m**2 * us%s_to_T
00150   us%m2_s_to_Z2_T = us%m_to_Z**2 * us%T_to_s
00151   ! Column mass loads:
00152   us%RZ_to_kg_m2  = us%R_to_kg_m3 * us%Z_to_m
00153     ! It does not seem like US%kg_m2_to_RZ would be used enough in MOM6 to justify its existence.
00154   ! Vertical mass fluxes:
00155   us%kg_m2s_to_RZ_T = us%kg_m3_to_R * us%m_to_Z * us%T_to_s
00156   us%RZ_T_to_kg_m2s = us%R_to_kg_m3 * us%Z_to_m * us%s_to_T
00157   ! Turbulent kinetic energy vertical fluxes:
00158   us%RZ3_T3_to_W_m2 = us%R_to_kg_m3 * us%Z_to_m**3 * us%s_to_T**3
00159   us%W_m2_to_RZ3_T3 = us%kg_m3_to_R * us%m_to_Z**3 * us%T_to_s**3
00160   ! Vertical heat fluxes:
00161   us%W_m2_to_QRZ_T = us%J_kg_to_Q * us%kg_m3_to_R * us%m_to_Z * us%T_to_s
00162   us%QRZ_T_to_W_m2 = us%Q_to_J_kg * us%R_to_kg_m3 * us%Z_to_m * us%s_to_T
00163   ! Pressures:
00164   us%RL2_T2_to_Pa = us%R_to_kg_m3 * us%L_T_to_m_s**2
00165     ! It does not seem like US%Pa_to_RL2_T2 would be used enough in MOM6 to justify its existence.
00166   ! US%Pa_to_RL2_T2 = US%kg_m3_to_R * US%m_s_to_L_T**2
00167
00168 end subroutine unit_scaling_init
00169
00170 !> Set the unit scaling factors for output to restart files to the unit scaling
00171 !! factors for this run.
00172 subroutine fix_restart_unit_scaling(US)
00173   type(unit_scale_type), intent(inout) :: US !< A dimensional unit scaling type
00174
00175   us%m_to_Z_restart = us%m_to_Z
00176   us%m_to_L_restart = us%m_to_L
00177   us%s_to_T_restart = us%s_to_T
00178   us%kg_m3_to_R_restart = us%kg_m3_to_R
00179   us%J_kg_to_Q_restart = us%J_kg_to_Q
00180
00181 end subroutine fix_restart_unit_scaling
00182
00183 !> Deallocates a unit scaling structure.
00184 subroutine unit_scaling_end( US )
00185   type(unit_scale_type), pointer :: US !< A dimensional unit scaling type
00186
00187   deallocate( us )
00188
00189 end subroutine unit_scaling_end
00190
00191 end module mom_unit_scaling
```

44.38 /home/cermak/src/MOM6/src/ice_shelf/MOM_ice_shelf.F90 File Reference

```
#include <MOM_memory.h>
#include "version_variable.h"
```

Data Types

- type mom_ice_shelf::ice_shelf_cs

  Control structure that contains ice shelf parameters and diagnostics handles.

Modules

- module mom_ice_shelf

  Implements the thermodynamic aspects of ocean / ice-shelf interactions, along with a crude placeholder for a later implementation of full ice shelf dynamics, all using the MOM framework and coding style.

Macros

- #define GRID_SYM_ .false.

Functions/Subroutines

- subroutine, public mom_ice_shelf::shelf_calc_flux (sfc_state, fluxes, Time, time_step, CS, forces)

  Calculates fluxes between the ocean and ice-shelf using the three-equations formulation (optional to use just two equations). See ICE_SHELF equations.
- subroutine mom_ice_shelf::change_thickness_using_melt (ISS, G, US, time_step, fluxes, density_ice, debug)

  Changes the thickness (mass) of the ice shelf based on sub-ice-shelf melting.
- subroutine, public mom_ice_shelf::add_shelf_forces (G, US, CS, forces, do_shelf_area)

  This subroutine adds the mechanical forcing fields and perhaps shelf areas, based on the ice state in ice_shelf_CS.
- subroutine mom_ice_shelf::add_shelf_pressure (G, US, CS, fluxes)

  This subroutine adds the ice shelf pressure to the fluxes type.
- subroutine, public mom_ice_shelf::add_shelf_flux (G, US, CS, sfc_state, fluxes)

  Updates surface fluxes that are influenced by sub-ice-shelf melting.
- subroutine, public mom_ice_shelf::initialize_ice_shelf (param_file, ocn_grid, Time, CS, diag, forces, fluxes, Time_in, solo_ice_sheet_in)

  Initializes shelf model data, parameters and diagnostics.
- subroutine mom_ice_shelf::initialize_shelf_mass (G, param_file, CS, ISS, new_sim)

  Initializes shelf mass based on three options (file, zero and user)
- subroutine mom_ice_shelf::update_shelf_mass (G, US, CS, ISS, Time)

  Updates the ice shelf mass using data from a file.
- subroutine, public mom_ice_shelf::ice_shelf_save_restart (CS, Time, directory, time_stamped, filename_↩ suffix)

  Save the ice shelf restart file.
- subroutine, public mom_ice_shelf::ice_shelf_end (CS)

  Deallocates all memory associated with this module.
- subroutine, public mom_ice_shelf::solo_step_ice_shelf (CS, time_interval, nsteps, Time, min_time_step_in)

  This routine is for stepping a stand-alone ice shelf model without an ocean.

Variables

- integer mom_ice_shelf::id_clock_shelf

  CPU Clock for the ice shelf code.
- integer mom_ice_shelf::id_clock_pass

  CPU Clock for group pass calls.

44.38.1 Macro Definition Documentation

44.38.1.1 GRID_SYM_

```
#define GRID_SYM_ .false.
```

44.39 MOM_ice_shelf.F90

```
00001 !> Implements the thermodynamic aspects of ocean / ice-shelf interactions,
00002 !!  along with a crude placeholder for a later implementation of full
00003 !!  ice shelf dynamics, all using the MOM framework and coding style.
00004 module mom_ice_shelf
00005
00006 ! This file is part of MOM6. See LICENSE.md for the license.
00007
00008 use mom_constants, only : hlf
00009 use mom_cpu_clock, only : cpu_clock_id, cpu_clock_begin, cpu_clock_end
00010 use mom_cpu_clock, only : clock_component, clock_routine
00011 use mom_diag_mediator, only : post_data, register_diag_field, safe_alloc_ptr
00012 use mom_diag_mediator, only : diag_mediator_init, set_diag_mediator_grid, diag_ctrl, time_type
00013 use mom_diag_mediator, only : enable_averages, enable_averaging, disable_averaging
00014 use mom_domains, only : mom_domains_init, clone_mom_domain
00015 use mom_domains, only : pass_var, pass_vector, to_all, cgrid_ne, bgrid_ne, corner
00016 use mom_dyn_horgrid, only : dyn_horgrid_type, create_dyn_horgrid, destroy_dyn_horgrid
00017 use mom_dyn_horgrid, only : rescale_dyn_horgrid_bathymetry
00018 use mom_error_handler, only : mom_error, mom_mesg, fatal, warning, is_root_pe
00019 use mom_file_parser, only : read_param, get_param, log_param, log_version, param_file_type
00020 use mom_grid, only : mom_grid_init, ocean_grid_type
00021 use mom_grid_initialize, only : set_grid_metrics
00022 use mom_fixed_initialization, only : mom_initialize_topography
00023 use mom_fixed_initialization, only : mom_initialize_rotation
00024 use user_initialization, only : user_initialize_topography
00025 use mom_io, only : field_exists, file_exists, mom_read_data, write_version_number
00026 use mom_io, only : slasher, fieldtype
00027 use mom_io, only : write_field, close_file, single_file, multiple
00028 use mom_restart, only : register_restart_field, query_initialized, save_restart
00029 use mom_restart, only : restart_init, restore_state, mom_restart_cs
00030 use mom_time_manager, only : time_type, time_type_to_real, real_to_time, operator(>), operator(-)
00031 use mom_transcribe_grid, only : copy_dyngrid_to_mom_grid, copy_mom_grid_to_dyngrid
00032 use mom_unit_scaling, only : unit_scale_type,
      unit_scaling_init, fix_restart_unit_scaling
00033 use mom_variables, only : surface
00034 use mom_forcing_type, only : forcing, allocate_forcing_type, mom_forcing_chksum
00035 use mom_forcing_type, only : mech_forcing, allocate_mech_forcing, mom_mech_forcing_chksum
00036 use mom_forcing_type, only : copy_common_forcing_fields
00037 use mom_get_input, only : directories, get_mom_input
00038 use mom_eos, only : calculate_density,
      calculate_density_derivs, calculate_tfreeze,
      eos_domain
00039 use mom_eos, only : eos_type, eos_init
00040 use mom_ice_shelf_dynamics, only : ice_shelf_dyn_cs, update_ice_shelf
00041 use mom_ice_shelf_dynamics, only : register_ice_shelf_dyn_restarts, initialize_ice_shelf_dyn
00042 use mom_ice_shelf_dynamics, only : ice_shelf_min_thickness_calve
00043 use mom_ice_shelf_dynamics, only : ice_time_step_cfl, ice_shelf_dyn_end
00044 use mom_ice_shelf_initialize, only : initialize_ice_thickness
00045 !MJH use MOM_ice_shelf_initialize, only : initialize_ice_shelf_boundary
00046 use mom_ice_shelf_state, only : ice_shelf_state, ice_shelf_state_end, ice_shelf_state_init
00047 use user_shelf_init, only : user_initialize_shelf_mass, user_update_shelf_mass
00048 use user_shelf_init, only : user_ice_shelf_cs
00049 use mom_coms, only : reproducing_sum
```

```
00050 use mom_spatial_means, only : global_area_integral
00051 use mom_checksums, only : hchksum, qchksum, chksum, uchksum, vchksum, uvchksum
00052 use time_interp_external_mod, only : init_external_field, time_interp_external
00053 use time_interp_external_mod, only : time_interp_external_init
00054 implicit none ; private
00055
00056 #include <MOM_memory.h>
00057 #ifdef SYMMETRIC_LAND_ICE
00058 #  define GRID_SYM_ .true.
00059 #else
00060 #  define GRID_SYM_ .false.
00061 #endif
00062
00063 public shelf_calc_flux, add_shelf_flux,
      initialize_ice_shelf, ice_shelf_end
00064 public ice_shelf_save_restart, solo_step_ice_shelf,
      add_shelf_forces
00065
00066 ! A note on unit descriptions in comments: MOM6 uses units that can be rescaled for dimensional
00067 ! consistency testing. These are noted in comments with units like Z, H, L, and T, along with
00068 ! their mks counterparts with notation like "a velocity [Z T-1 ~> m s-1]".  If the units
00069 ! vary with the Boussinesq approximation, the Boussinesq variant is given first.
00070
00071 !> Control structure that contains ice shelf parameters and diagnostics handles
00072 type, public :: ice_shelf_cs ; private
00073   ! Parameters
00074   type(mom_restart_cs), pointer :: restart_csp => null() !< A pointer to the restart control
00075                                                          !! structure for the ice shelves
00076   type(ocean_grid_type) :: grid                          !< Grid for the ice-shelf model
00077   type(unit_scale_type), pointer :: &
00078     us => null()       !< A structure containing various unit conversion factors
00079   !type(dyn_horgrid_type), pointer :: dG  !< Dynamic grid for the ice-shelf model
00080   type(ocean_grid_type), pointer :: ocn_grid => null() !< A pointer to the ocean model grid
00081                                                          !! The rest is private
00082   real ::   flux_factor = 1.0                            !< A factor that can be used to turn off ice shelf
00083                                                          !! melting (flux_factor = 0) [nondim].
00084   character(len=128) :: restart_output_dir = ' ' !< The directory in which to write restart files
00085   type(ice_shelf_state), pointer :: iss => null() !< A structure with elements that describe
00086                                                          !! the ice-shelf state
00087   type(ice_shelf_dyn_cs), pointer :: dcs => null() !< The control structure for the ice-shelf dynamics.
00088
00089   real, pointer, dimension(:,:) :: &
00090     utide   => null()  !< An unresolved tidal velocity [L T-1 ~> m s-1]
00091
00092   real :: ustar_bg     !< A minimum value for ustar under ice shelves [Z T-1 ~> m s-1].
00093   real :: cdrag        !< drag coefficient under ice shelves [nondim].
00094   real :: g_earth      !< The gravitational acceleration [L2 Z-1 T-2 ~> m s-2]
00095   real :: cp           !< The heat capacity of sea water [Q degC-1 ~> J kg-1 degC-1].
00096   real :: rho_ocn      !< A reference ocean density [R ~> kg m-3].
00097   real :: cp_ice       !< The heat capacity of fresh ice [Q degC-1 ~> J kg-1 degC-1].
00098   real :: gamma_t      !< The (fixed) turbulent exchange velocity in the
00099                        !< 2-equation formulation [Z T-1 ~> m s-1].
00100   real :: salin_ice    !< The salinity of shelf ice [ppt].
00101   real :: temp_ice     !< The core temperature of shelf ice [degC].
00102   real :: kv_ice       !< The viscosity of ice [L4 Z-2 T-1 ~> m2 s-1].
00103   real :: density_ice  !< A typical density of ice [R ~> kg m-3].
00104   real :: kv_molec     !< The molecular kinematic viscosity of sea water [Z2 T-1 ~> m2 s-1].
00105   real :: kd_molec_salt!< The molecular diffusivity of salt [Z2 T-1 ~> m2 s-1].
00106   real :: kd_molec_temp!< The molecular diffusivity of heat [Z2 T-1 ~> m2 s-1].
00107   real :: lat_fusion   !< The latent heat of fusion [Q ~> J kg-1].
00108   real :: gamma_t_3eq  !<  Nondimensional heat-transfer coefficient, used in the 3Eq. formulation
00109   real :: gamma_s_3eq  !<  Nondimensional salt-transfer coefficient, used in the 3Eq. formulation
00110                        !<  This number should be specified by the user.
00111   real :: col_mass_melt_threshold !< An ocean column mass below the iceshelf below which melting
00112                        !! does not occur [R Z ~> kg m-2]
00113   logical :: mass_from_file !< Read the ice shelf mass from a file every dt
00114
00115   !!!! PHYSICAL AND NUMERICAL PARAMETERS FOR ICE DYNAMICS !!!!!!
00116
00117   real :: time_step    !< this is the shortest timestep that the ice shelf sees, and
00118                        !! is equal to the forcing timestep (it is passed in when the shelf
00119                        !! is initialized - so need to reorganize MOM driver.
00120                        !! it will be the prognistic timestep ... maybe.
00121
00122   logical :: solo_ice_sheet !< whether the ice model is running without being
00123                             !! coupled to the ocean
00124   logical :: gl_regularize  !< whether to regularize the floatation condition
00125                             !! at the grounding line a la Goldberg Holland Schoof 2009
00126   logical :: gl_couple      !< whether to let the floatation condition be
00127                             !!determined by ocean column thickness means update_OD_ffrac
00128                             !! will be called (note: GL_regularize and GL_couple
00129                             !! should be exclusive)
00130   logical :: calve_to_mask  !< If true, calve any ice that passes outside of a masked area
00131   real :: min_thickness_simple_calve !< min. ice shelf thickness criteria for calving [Z ~> m].
00132   real :: t0                !< temperature at ocean surface in the restoring region [degC]
00133   real :: s0                !< Salinity at ocean surface in the restoring region [ppt].
00134   real :: input_flux        !< Ice volume flux at an upstream open boundary [m3 s-1].
```

```
00135   real :: input_thickness   !< Ice thickness at an upstream open boundary [m].
00136
00137   type(time_type) :: time               !< The component's time.
00138   type(eos_type), pointer :: eqn_of_state => null() !< Type that indicates the
00139                                         !! equation of state to use.
00140   logical :: active_shelf_dynamics      !< True if the ice shelf mass changes as a result
00141                                         !! the dynamic ice-shelf model.
00142   logical :: override_shelf_movement    !< If true, user code specifies the shelf movement
00143                                         !! instead of using the dynamic ice-shelf mode.
00144   logical :: isthermo                   !< True if the ice shelf can exchange heat and
00145                                         !! mass with the underlying ocean.
00146   logical :: threeeq                    !< If true, the 3 equation consistency equations are
00147                                         !! used to calculate the flux at the ocean-ice
00148                                         !! interface.
00149   logical :: insulator                  !< If true, ice shelf is a perfect insulator
00150   logical :: const_gamma                !< If true, gamma_T is specified by the user.
00151   logical :: constant_sea_level         !< if true, apply an evaporative, heat and salt
00152                                         !! fluxes. It will avoid large increase in sea level.
00153   real    :: min_ocean_mass_float       !< The minimum ocean mass per unit area before the ice
00154                                         !! shelf is considered to float when constant_sea_level
00155                                         !! is used [R Z ~> kg m-2]
00156   real    :: cutoff_depth               !< Depth above which melt is set to zero (>= 0) [Z ~> m].
00157   logical :: find_salt_root             !< If true, if true find Sbdry using a quadratic eq.
00158   real    :: tfr_0_0                    !< The freezing point at 0 pressure and 0 salinity [degC]
00159   real    :: dtfr_ds                    !< Partial derivative of freezing temperature with salinity [degC
    ppt-1]
00160   real    :: dtfr_dp                    !< Partial derivative of freezing temperature with
00161                                         !! pressure [degC T2 R-1 L-2 ~> degC Pa-1]
00162   !>@{ Diagnostic handles
00163   integer :: id_melt = -1, id_exch_vel_s = -1, id_exch_vel_t = -1, &
00164              id_tfreeze = -1, id_tfl_shelf = -1, &
00165              id_thermal_driving = -1, id_haline_driving = -1, &
00166              id_u_ml = -1, id_v_ml = -1, id_sbdry = -1, &
00167              id_h_shelf = -1, id_h_mask = -1, &
00168              id_surf_elev = -1, id_bathym = -1, &
00169              id_area_shelf_h = -1, &
00170              id_ustar_shelf = -1, id_shelf_mass = -1, id_mass_flux = -1
00171   !>@}
00172
00173   integer :: id_read_mass !< An integer handle used in time interpolation of
00174                           !! the ice shelf mass read from a file
00175   integer :: id_read_area !< An integer handle used in time interpolation of
00176                           !! the ice shelf mass read from a file
00177
00178   type(diag_ctrl), pointer :: diag => null() !< A structure that is used to control diagnostic output.
00179   type(user_ice_shelf_cs), pointer :: user_cs => null() !< A pointer to the control structure for
00180                                    !! user-supplied modifications to the ice shelf code.
00181
00182   logical :: debug                 !< If true, write verbose checksums for debugging purposes
00183                                    !! and use reproducible sums
00184 end type ice_shelf_cs
00185
00186 integer :: id_clock_shelf !< CPU Clock for the ice shelf code
00187 integer :: id_clock_pass !< CPU Clock for group pass calls
00188
00189 contains
00190
00191 !> Calculates fluxes between the ocean and ice-shelf using the three-equations
00192 !! formulation (optional to use just two equations).
00193 !! See \ref section_ICE_SHELF_equations
00194 subroutine shelf_calc_flux(sfc_state, fluxes, Time, time_step, CS, forces)
00195   type(surface),         intent(inout) :: sfc_state !< A structure containing fields that
00196                                                     !! describe the surface state of the ocean.  The
00197                                                     !! intent is only inout to allow for halo updates.
00198   type(forcing),         intent(inout) :: fluxes !< structure containing pointers to any possible
00199                                                     !! thermodynamic or mass-flux forcing fields.
00200   type(time_type),       intent(in)    :: Time  !< Start time of the fluxes.
00201   real,                  intent(in)    :: time_step !< Length of time over which these fluxes
00202                                                     !! will be applied [s].
00203   type(ice_shelf_cs),    pointer       :: CS    !< A pointer to the control structure returned
00204                                                     !! by a previous call to initialize_ice_shelf.
00205   type(mech_forcing), optional, intent(inout) :: forces !< A structure with the driving mechanical forces
00206
00207   ! Local variables
00208   type(ocean_grid_type), pointer :: G => null()  !< The grid structure used by the ice shelf.
00209   type(unit_scale_type), pointer :: US => null() !< Pointer to a structure containing
00210                                                  !! various unit conversion factors
00211   type(ice_shelf_state), pointer :: ISS => null() !< A structure with elements that describe
00212                                                  !! the ice-shelf state
00213
00214   real, dimension(SZI_(CS%grid)) :: &
00215     Rhoml, &   !< Ocean mixed layer density [R ~> kg m-3].
00216     dR0_dT, &  !< Partial derivative of the mixed layer density
00217                !< with temperature [R degC-1 ~> kg m-3 degC-1].
00218     dr0_ds, &  !< Partial derivative of the mixed layer density
00219                !< with salinity [R ppt-1 ~> kg m-3 ppt-1].
00220     p_int      !< The pressure at the ice-ocean interface [R L2 T-2 ~> Pa].
```

```
00221
00222   real, dimension(SZI_(CS%grid),SZJ_(CS%grid)) :: &
00223     exch_vel_t, &  !< Sub-shelf thermal exchange velocity [Z T-1 ~> m s-1]
00224     exch_vel_s     !< Sub-shelf salt exchange velocity [Z T-1 ~> m s-1]
00225
00226   real, dimension(SZDI_(CS%grid),SZDJ_(CS%grid)) :: &
00227     mass_flux  !< Total mass flux of freshwater across the ice-ocean interface. [R Z L2 T-1 ~> kg/s]
00228   real, dimension(SZDI_(CS%grid),SZDJ_(CS%grid)) :: &
00229     haline_driving !< (SSS - S_boundary) ice-ocean
00230                    !! interface, positive for melting and negative for freezing.
00231                    !! This is computed as part of the ISOMIP diagnostics.
00232   real, parameter :: VK    = 0.40 !< Von Karman's constant - dimensionless
00233   real :: ZETA_N = 0.052 !> The fraction of the boundary layer over which the
00234                    !! viscosity is linearly increasing [nondim]. (Was 1/8. Why?)
00235   real, parameter :: RC    = 0.20     ! critical flux Richardson number.
00236   real :: I_ZETA_N !< The inverse of ZETA_N [nondim].
00237   real :: I_LF     !< The inverse of the latent heat of fusion [Q-1 ~> kg J-1].
00238   real :: I_VK     !< The inverse of the Von Karman constant [nondim].
00239   real :: PR, SC   !< The Prandtl number and Schmidt number [nondim].
00240
00241   ! 3 equations formulation variables
00242   real, dimension(SZDI_(CS%grid),SZDJ_(CS%grid)) :: &
00243     Sbdry     !< Salinities in the ocean at the interface with the ice shelf [ppt].
00244   real :: Sbdry_it
00245   real :: Sbdry1, Sbdry2
00246   real :: S_a, S_b, S_c  ! Variables used to find salt roots
00247   real :: dS_it    !< The interface salinity change during an iteration [ppt].
00248   real :: hBL_neut !< The neutral boundary layer thickness [Z ~> m].
00249   real :: hBL_neut_h_molec !< The ratio of the neutral boundary layer thickness
00250                    !! to the molecular boundary layer thickness [nondim].
00251   real :: wT_flux !< The downward vertical flux of heat just inside the ocean [degC Z T-1 ~> degC m s-1].
00252   real :: wB_flux !< The downward vertical flux of buoyancy just inside the ocean [Z2 T-3 ~> m2 s-3].
00253   real :: dB_dS   !< The derivative of buoyancy with salinity [Z T-2 ppt-1 ~> m s-2 ppt-1].
00254   real :: dB_dT   !< The derivative of buoyancy with temperature [Z T-2 degC-1 ~> m s-2 degC-1].
00255   real :: I_n_star ! [nondim]
00256   real :: n_star_term ! A term in the expression for nstar [T3 Z-2 ~> s3 m-2]
00257   real :: absf     ! The absolute value of the Coriolis parameter [T-1 ~> s-1]
00258   real :: dIns_dwB !< The partial derivative of I_n_star with wB_flux, in [T3 Z-2 ~> s3 m-2]
00259   real :: dT_ustar ! The difference between the the freezing point and the ocean boundary layer
00260                    ! temperature times the friction velocity [degC Z T-1 ~> degC m s-1]
00261   real :: dS_ustar ! The difference between the salinity at the ice-ocean interface and the ocean
00262                    ! boundary layer salinity times the friction velocity [ppt Z T-1 ~> ppt m s-1]
00263   real :: ustar_h  ! The friction velocity in the water below the ice shelf [Z T-1 ~> m s-1]
00264   real :: Gam_turb ! [nondim]
00265   real :: Gam_mol_t, Gam_mol_s ! Relative coefficients of molecular diffusivites [nondim]
00266   real :: RhoCp     ! A typical ocean density times the heat capacity of water [Q R ~> J m-3]
00267   real :: ln_neut
00268   real :: mass_exch ! A mass exchange rate [R Z T-1 ~> kg m-2 s-1]
00269   real :: Sb_min, Sb_max
00270   real :: dS_min, dS_max
00271   ! Variables used in iterating for wB_flux.
00272   real :: wB_flux_new, dDwB_dwB_in
00273   real :: I_Gam_T, I_Gam_S
00274   real :: dG_dwB   ! The derivative of Gam_turb with wB [T3 Z-2 ~> s3 m-2]
00275   real :: taux2, tauy2 ! The squared surface stresses [R2 L2 Z2 T-4 ~> Pa2].
00276   real :: u2_av, v2_av ! The ice-area weighted average squared ocean velocities [L2 T-2 ~> m2 s-2]
00277   real :: asu1, asu2   ! Ocean areas covered by ice shelves at neighboring u-
00278   real :: asv1, asv2   ! and v-points [L2 ~> m2].
00279   real :: I_au, I_av   ! The Adcroft reciprocals of the ice shelf areas at adjacent points [L-2 ~> m-2]
00280   real :: Irho0        ! The inverse of the mean density times a unit conversion factor [R-1 L Z-1 ~> m3
      kg-1]
00281   logical :: Sb_min_set, Sb_max_set
00282   logical :: update_ice_vel ! If true, it is time to update the ice shelf velocities.
00283   logical :: coupled_GL     ! If true, the grouding line position is determined based on
00284                             ! coupled ice-ocean dynamics.
00285
00286   real, parameter :: c2_3 = 2.0/3.0
00287   character(len=160) :: mesg  ! The text of an error message
00288   integer, dimension(2) :: EOSdom ! The i-computational domain for the equation of state
00289   integer :: i, j, is, ie, js, je, ied, jed, it1, it3
00290
00291   if (.not. associated(cs)) call mom_error(fatal, "shelf_calc_flux: "// &
00292        "initialize_ice_shelf must be called before shelf_calc_flux.")
00293   call cpu_clock_begin(id_clock_shelf)
00294
00295   g => cs%grid ; us => cs%US
00296   iss => cs%ISS
00297
00298   ! useful parameters
00299   is = g%isc ; ie = g%iec ; js = g%jsc ; je = g%jec ; ied = g%ied ; jed = g%jed
00300   i_zeta_n = 1.0 / zeta_n
00301   i_lf = 1.0 / cs%Lat_fusion
00302   sc = cs%kv_molec/cs%kd_molec_salt
00303   pr = cs%kv_molec/cs%kd_molec_temp
00304   i_vk = 1.0/vk
00305   rhocp = cs%Rho_ocn * cs%Cp
00306
```

```
00307   !first calculate molecular component
00308   gam_mol_t = 12.5 * (pr**c2_3) - 6.0
00309   gam_mol_s = 12.5 * (sc**c2_3) - 6.0
00310
00311   ! GMM, zero some fields of the ice shelf structure (ice_shelf_CS)
00312   ! these fields are already set to zero during initialization
00313   ! However, they seem to be changed somewhere and, for diagnostic
00314   ! reasons, it is better to set them to zero again.
00315   exch_vel_t(:,:) = 0.0 ; exch_vel_s(:,:) = 0.0
00316   iss%tflux_shelf(:,:) = 0.0 ; iss%water_flux(:,:) = 0.0
00317   iss%salt_flux(:,:) = 0.0 ; iss%tflux_ocn(:,:) = 0.0 ; iss%tfreeze(:,:) = 0.0
00318   ! define Sbdry to avoid Run-Time Check Failure, when melt is not computed.
00319   haline_driving(:,:) = 0.0
00320   sbdry(:,:) = sfc_state%sss(:,:)
00321
00322   !update time
00323   cs%Time = time
00324
00325   if (cs%override_shelf_movement) then
00326     cs%time_step = time_step
00327     ! update shelf mass
00328     if (cs%mass_from_file) call update_shelf_mass(g, us, cs, iss, time)
00329   endif
00330
00331   if (cs%debug) then
00332     call hchksum(fluxes%frac_shelf_h, "frac_shelf_h before apply melting", g%HI, haloshift=0)
00333     call hchksum(sfc_state%sst, "sst before apply melting", g%HI, haloshift=0)
00334     call hchksum(sfc_state%sss, "sss before apply melting", g%HI, haloshift=0)
00335     call hchksum(sfc_state%u, "u_ml before apply melting", g%HI, haloshift=0, scale=us%L_T_to_m_s)
00336     call hchksum(sfc_state%v, "v_ml before apply melting", g%HI, haloshift=0, scale=us%L_T_to_m_s)
00337     call hchksum(sfc_state%ocean_mass, "ocean_mass before apply melting", g%HI, haloshift=0, &
00338                  scale=us%RZ_to_kg_m2)
00339   endif
00340
00341   ! Calculate the friction velocity under ice shelves, using taux_shelf and tauy_shelf if possible.
00342   if (allocated(sfc_state%taux_shelf) .and. allocated(sfc_state%tauy_shelf)) then
00343     call pass_vector(sfc_state%taux_shelf, sfc_state%tauy_shelf, g%domain, to_all, cgrid_ne)
00344   endif
00345   irho0 = us%Z_to_L / cs%Rho_ocn
00346   do j=js,je ; do i=is,ie ; if (fluxes%frac_shelf_h(i,j) > 0.0) then
00347     taux2 = 0.0 ; tauy2 = 0.0 ; u2_av = 0.0 ; v2_av = 0.0
00348     asu1 = (iss%area_shelf_h(i-1,j) + iss%area_shelf_h(i,j))
00349     asu2 = (iss%area_shelf_h(i,j) + iss%area_shelf_h(i+1,j))
00350     asv1 = (iss%area_shelf_h(i,j-1) + iss%area_shelf_h(i,j))
00351     asv2 = (iss%area_shelf_h(i,j) + iss%area_shelf_h(i,j+1))
00352     i_au = 0.0 ; if (asu1 + asu2 > 0.0) i_au = 1.0 / (asu1 + asu2)
00353     i_av = 0.0 ; if (asv1 + asv2 > 0.0) i_av = 1.0 / (asv1 + asv2)
00354     if (allocated(sfc_state%taux_shelf) .and. allocated(sfc_state%tauy_shelf)) then
00355       taux2 = (asu1 * sfc_state%taux_shelf(i-1,j)**2 + asu2 * sfc_state%taux_shelf(i,j)**2  ) * i_au
00356       tauy2 = (asv1 * sfc_state%tauy_shelf(i,j-1)**2 + asv2 * sfc_state%tauy_shelf(i,j)**2  ) * i_av
00357     endif
00358     u2_av = (asu1 * sfc_state%u(i-1,j)**2 + asu2 * sfc_state%u(i,j)**2) * i_au
00359     v2_av = (asv1 * sfc_state%v(i,j-1)**2 + asu2 * sfc_state%v(i,j)**2) * i_av
00360
00361     if (taux2 + tauy2 > 0.0) then
00362       fluxes%ustar_shelf(i,j) = max(cs%ustar_bg, us%L_to_Z * &
00363           sqrt(irho0 * sqrt(taux2 + tauy2) + cs%cdrag*cs%utide(i,j)**2))
00364     else   ! Take care of the cases when taux_shelf is not set or not allocated.
00365       fluxes%ustar_shelf(i,j) = max(cs%ustar_bg, us%L_TO_Z * &
00366           sqrt(cs%cdrag*((u2_av + v2_av) + cs%utide(i,j)**2)))
00367     endif
00368   else ! There is no shelf here.
00369     fluxes%ustar_shelf(i,j) = 0.0
00370   endif ; enddo ; enddo
00371
00372   eosdom(:) = eos_domain(g%HI)
00373   do j=js,je
00374     ! Find the pressure at the ice-ocean interface, averaged only over the
00375     ! part of the cell covered by ice shelf.
00376     do i=is,ie ; p_int(i) = cs%g_Earth * iss%mass_shelf(i,j) ; enddo
00377
00378     ! Calculate insitu densities and expansion coefficients
00379     call calculate_density(sfc_state%sst(:,j), sfc_state%sss(:,j), p_int, rhoml(:), &
00380                            cs%eqn_of_state, eosdom)
00381     call calculate_density_derivs(sfc_state%sst(:,j), sfc_state%sss(:,j), p_int,
    dr0_dt, dr0_ds, &
00382                            cs%eqn_of_state, eosdom)
00383
00384     do i=is,ie
00385       if ((sfc_state%ocean_mass(i,j) > cs%col_mass_melt_threshold) .and. &
00386           (iss%area_shelf_h(i,j) > 0.0) .and. cs%isthermo) then
00387
00388         if (cs%threeeq) then
00389           !   Iteratively determine a self-consistent set of fluxes, with the ocean
00390           ! salinity just below the ice-shelf as the variable that is being
00391           ! iterated for.
00392
```

```
00393           ustar_h = fluxes%ustar_shelf(i,j)
00394
00395           ! Estimate the neutral ocean boundary layer thickness as the minimum of the
00396           ! reported ocean mixed layer thickness and the neutral Ekman depth.
00397           absf = 0.25*((abs(g%CoriolisBu(i,j)) + abs(g%CoriolisBu(i-1,j-1))) + &
00398                                (abs(g%CoriolisBu(i,j-1)) + abs(g%CoriolisBu(i-1,j))))
00399           if (absf*sfc_state%Hml(i,j) <= vk*ustar_h) then ; hbl_neut = sfc_state%Hml(i,j)
00400           else ; hbl_neut = (vk*ustar_h) / absf ; endif
00401           hbl_neut_h_molec = zeta_n * ((hbl_neut * ustar_h) / (5.0 * cs%kv_molec))
00402
00403           ! Determine the mixed layer buoyancy flux, wB_flux.
00404           db_ds = (us%L_to_Z**2*cs%g_Earth / rhoml(i)) * dr0_ds(i)
00405           db_dt = (us%L_to_Z**2*cs%g_Earth / rhoml(i)) * dr0_dt(i)
00406           ln_neut = 0.0 ; if (hbl_neut_h_molec > 1.0) ln_neut = log(hbl_neut_h_molec)
00407
00408           if (cs%find_salt_root) then
00409             ! Solve for the skin salinity using the linearized liquidus parameters and
00410             ! balancing the turbulent fresh water flux in the near-boundary layer with
00411             ! the net fresh water or salt added by melting:
00412             ! (Cp/Lat_fusion)*Gamma_T_3Eq*(TFr_skin-T_ocn) = Gamma_S_3Eq*(S_skin-S_ocn)/S_skin
00413
00414             ! S_a is always < 0.0 with a realistic expression for the freezing point.
00415             s_a = cs%dTFr_dS * cs%Gamma_T_3EQ * cs%Cp
00416             s_b = cs%Gamma_T_3EQ*cs%Cp*(cs%TFr_0_0 + cs%dTFr_dp*p_int(i) - sfc_state%sst(i,j)) - &
00417                   cs%Lat_fusion * cs%Gamma_S_3EQ    ! S_b Can take either sign, but is usually negative.
00418             s_c = cs%Lat_fusion * cs%Gamma_S_3EQ * sfc_state%sss(i,j) ! Always >= 0
00419
00420             if (s_c == 0.0) then  ! The solution for fresh water.
00421               sbdry(i,j) = 0.0
00422             elseif (s_a < 0.0) then ! This is the usual ocean case
00423               if (s_b < 0.0) then ! This is almost always the case
00424                 sbdry(i,j) = 2.0*s_c / (-s_b + sqrt(s_b*s_b - 4.*s_a*s_c))
00425               else
00426                 sbdry(i,j) = (s_b + sqrt(s_b*s_b - 4.*s_a*s_c)) / (-2.*s_a)
00427               endif
00428             elseif ((s_a == 0.0) .and. (s_b < 0.0)) then ! It should be the case that S_b < 0.
00429               sbdry(i,j) = -s_c / s_b
00430             else
00431               call mom_error(fatal, "Impossible conditions found in 3-equation skin salinity calculation.")
00432             endif
00433
00434             ! Safety check
00435             if (sbdry(i,j) < 0.) then
00436               write(mesg,*) 'sfc_state%sss(i,j) = ',sfc_state%sss(i,j), 'S_a, S_b, S_c', s_a, s_b, s_c
00437               call mom_error(warning, mesg, .true.)
00438               write(mesg,*) 'I,J,Sbdry1,Sbdry2',i,j,sbdry1,sbdry2
00439               call mom_error(warning, mesg, .true.)
00440               call mom_error(fatal, "shelf_calc_flux: Negative salinity (Sbdry).")
00441             endif
00442           else
00443             ! Guess sss as the iteration starting point for the boundary salinity.
00444             sbdry(i,j) = sfc_state%sss(i,j) ; sb_max_set = .false.
00445             sb_min_set = .false.
00446           endif !find_salt_root
00447
00448           do it1 = 1,20
00449             ! Determine the potential temperature at the ice-ocean interface.
00450             call calculate_tfreeze(sbdry(i,j), p_int(i), iss%tfreeze(i,j), cs%eqn_of_state
    , &
00451                                    pres_scale=us%RL2_T2_to_Pa)
00452
00453             dt_ustar = (iss%tfreeze(i,j) - sfc_state%sst(i,j)) * ustar_h
00454             ds_ustar = (sbdry(i,j) - sfc_state%sss(i,j)) * ustar_h
00455
00456             ! First, determine the buoyancy flux assuming no effects of stability
00457             ! on the turbulence.  Following H & J '99, this limit also applies
00458             ! when the buoyancy flux is destabilizing.
00459
00460             if (cs%const_gamma) then ! if using a constant gamma_T
00461               ! note the different form, here I_Gam_T is NOT 1/Gam_T!
00462               i_gam_t = cs%Gamma_T_3EQ
00463               i_gam_s = cs%Gamma_S_3EQ
00464             else
00465               gam_turb = i_vk * (ln_neut + (0.5 * i_zeta_n - 1.0))
00466               i_gam_t = 1.0 / (gam_mol_t + gam_turb)
00467               i_gam_s = 1.0 / (gam_mol_s + gam_turb)
00468             endif
00469
00470             wt_flux = dt_ustar * i_gam_t
00471             wb_flux = db_ds * (ds_ustar * i_gam_s) + db_dt * wt_flux
00472
00473             if (wb_flux < 0.0) then
00474               ! The buoyancy flux is stabilizing and will reduce the tubulent
00475               ! fluxes, and iteration is required.
00476               n_star_term = (zeta_n/rc) * (hbl_neut * vk) / (ustar_h)**3
00477               do it3 = 1,30
00478                ! n_star <= 1.0 is the ratio of working boundary layer thickness
```

```
00479                ! to the neutral thickness.
00480                ! hBL = n_star*hBL_neut ; hSub = 1/8*n_star*hBL
00481
00482                 i_n_star = sqrt(1.0 - n_star_term * wb_flux)
00483                 dins_dwb = 0.5 * n_star_term / i_n_star
00484                 if (hbl_neut_h_molec > i_n_star**2) then
00485                   gam_turb = i_vk * ((ln_neut - 2.0*log(i_n_star)) + &
00486                                      (0.5*i_zeta_n*i_n_star - 1.0))
00487                   dg_dwb =  i_vk * ( -2.0 / i_n_star + (0.5 * i_zeta_n)) * dins_dwb
00488                 else
00489                   !   The layer dominated by molecular viscosity is smaller than
00490                   ! the assumed boundary layer.  This should be rare!
00491                   gam_turb = i_vk * (0.5 * i_zeta_n*i_n_star - 1.0)
00492                   dg_dwb = i_vk * (0.5 * i_zeta_n) * dins_dwb
00493                 endif
00494
00495                 if (cs%const_gamma) then ! if using a constant gamma_T
00496                   ! note the different form, here I_Gam_T is NOT 1/Gam_T!
00497                   i_gam_t = cs%Gamma_T_3EQ
00498                   i_gam_s = cs%Gamma_S_3EQ
00499                 else
00500                   i_gam_t = 1.0 / (gam_mol_t + gam_turb)
00501                   i_gam_s = 1.0 / (gam_mol_s + gam_turb)
00502                 endif
00503
00504                 wt_flux = dt_ustar * i_gam_t
00505                 wb_flux_new = db_ds * (ds_ustar * i_gam_s) + db_dt * wt_flux
00506
00507                 ! Find the root where wB_flux_new = wB_flux.  Make the 1.0e-4 below into a parameter?
00508                 if (abs(wb_flux_new - wb_flux) < 1.0e-4*(abs(wb_flux_new) + abs(wb_flux))) exit
00509
00510                 ddwb_dwb_in = dg_dwb * (db_ds * (ds_ustar * i_gam_s**2) + &
00511                                         db_dt * (dt_ustar * i_gam_t**2)) - 1.0
00512                 ! This is Newton's method without any bounds.  Should bounds be needed?
00513                 wb_flux_new = wb_flux - (wb_flux_new - wb_flux) / ddwb_dwb_in
00514               enddo !it3
00515             endif
00516
00517             iss%tflux_ocn(i,j)  = rhocp * wt_flux
00518             exch_vel_t(i,j) = ustar_h * i_gam_t
00519             exch_vel_s(i,j) = ustar_h * i_gam_s
00520
00521             ! Calculate the heat flux inside the ice shelf.
00522             ! Vertical adv/diff as in H+J 1999, eqns (26) & approx from (31).
00523             !   Q_ice = density_ice * CS%Cp_ice * K_ice * dT/dz (at interface)
00524             ! vertical adv/diff as in H+J 1999, eqs (31) & (26)...
00525             !   dT/dz ~= min( (lprec/(density_ice*K_ice))*(CS%Temp_Ice-T_freeze) , 0.0 )
00526             ! If this approximation is not made, iterations are required... See H+J Fig 3.
00527
00528             if (iss%tflux_ocn(i,j) >= 0.0) then
00529               ! Freezing occurs due to downward ocean heat flux, so zero iout ce heat flux.
00530               iss%water_flux(i,j) = -i_lf * iss%tflux_ocn(i,j)
00531               iss%tflux_shelf(i,j) = 0.0
00532             else
00533               if (cs%insulator) then
00534                 !no conduction/perfect insulator
00535                 iss%tflux_shelf(i,j) = 0.0
00536                 iss%water_flux(i,j) = i_lf * (iss%tflux_shelf(i,j) - iss%tflux_ocn(i,j))
00537
00538               else
00539                 ! With melting, from H&J 1999, eqs (31) & (26)...
00540                 !   Q_ice ~= Cp_ice * (CS%Temp_Ice-T_freeze) * lprec
00541                 !   RhoLF*lprec = Q_ice - ISS%tflux_ocn(i,j)
00542                 !   lprec = -(ISS%tflux_ocn(i,j)) / (CS%Lat_fusion + Cp_ice * (T_freeze-CS%Temp_Ice))
00543                 iss%water_flux(i,j) = -iss%tflux_ocn(i,j) / &
00544                      (cs%Lat_fusion + cs%Cp_ice * (iss%tfreeze(i,j) - cs%Temp_Ice))
00545
00546                 iss%tflux_shelf(i,j) = iss%tflux_ocn(i,j) + cs%Lat_fusion*iss%water_flux(i,j)
00547               endif
00548
00549             endif
00550             !other options: dTi/dz linear through shelf, with draft in [Z ~> m], KTI in [Z2 T-1 ~> m2 s-1]
00551             !    dTi_dz = (CS%Temp_Ice - ISS%tfreeze(i,j)) / draft(i,j)
00552             !    ISS%tflux_shelf(i,j) = Rho_Ice * CS%Cp_ice * KTI * dTi_dz
00553
00554
00555             if (cs%find_salt_root) then
00556               exit ! no need to do interaction, so exit loop
00557             else
00558
00559               mass_exch = exch_vel_s(i,j) * cs%Rho_ocn
00560               sbdry_it = (sfc_state%sss(i,j) * mass_exch + cs%Salin_ice * iss%water_flux(i,j)) / &
00561                          (mass_exch + iss%water_flux(i,j))
00562               ds_it = sbdry_it - sbdry(i,j)
00563               if (abs(ds_it) < 1.0e-4*(0.5*(sfc_state%sss(i,j) + sbdry(i,j) + 1.0e-10))) exit
00564
00565
```

```
00566               if (ds_it < 0.0) then ! Sbdry is now the upper bound.
00567                 if (sb_max_set .and. (sbdry(i,j) > sb_max)) &
00568                   call mom_error(fatal,"shelf_calc_flux: Irregular iteration for Sbdry (max).")
00569                 sb_max = sbdry(i,j) ; ds_max = ds_it ; sb_max_set = .true.
00570               else ! Sbdry is now the lower bound.
00571                 if (sb_min_set .and. (sbdry(i,j) < sb_min)) &
00572                    call mom_error(fatal, "shelf_calc_flux: Irregular iteration for Sbdry (min).")
00573                  sb_min = sbdry(i,j) ; ds_min = ds_it ; sb_min_set = .true.
00574               endif ! dS_it < 0.0
00575
00576               if (sb_min_set .and. sb_max_set) then
00577                 ! Use the false position method for the next iteration.
00578                 sbdry(i,j) = sb_min + (sb_max-sb_min) * (ds_min / (ds_min - ds_max))
00579               else
00580                 sbdry(i,j) = sbdry_it
00581               endif ! Sb_min_set
00582
00583               sbdry(i,j) = sbdry_it
00584             endif ! CS%find_salt_root
00585
00586           enddo !it1
00587           ! Check for non-convergence and/or non-boundedness?
00588
00589         else
00590           !   In the 2-equation form, the mixed layer turbulent exchange velocity
00591           ! is specified and large enough that the ocean salinity at the interface
00592           ! is about the same as the boundary layer salinity.
00593
00594           call calculate_tfreeze(sfc_state%sss(i,j), p_int(i), iss%tfreeze(i,j), cs
    %eqn_of_state, &
00595                                  pres_scale=us%RL2_T2_to_Pa)
00596
00597           exch_vel_t(i,j) = cs%gamma_t
00598           iss%tflux_ocn(i,j) = rhocp * exch_vel_t(i,j) * (iss%tfreeze(i,j) - sfc_state%sst(i,j))
00599           iss%tflux_shelf(i,j) = 0.0
00600           iss%water_flux(i,j) = -i_lf * iss%tflux_ocn(i,j)
00601           sbdry(i,j) = 0.0
00602         endif
00603       elseif (iss%area_shelf_h(i,j) > 0.0) then ! This is an ice-sheet, not a floating shelf.
00604         iss%tflux_ocn(i,j) = 0.0
00605       else ! There is no ice shelf or sheet here.
00606         iss%tflux_ocn(i,j) = 0.0
00607       endif
00608
00609 !      haline_driving(i,j) = sfc_state%sss(i,j) - Sbdry(i,j)
00610
00611     enddo ! i-loop
00612   enddo ! j-loop
00613
00614
00615   do j=js,je ; do i=is,ie
00616     ! ISS%water_flux = net liquid water into the ocean [R Z T-1 ~> kg m-2 s-1]
00617     fluxes%iceshelf_melt(i,j) = iss%water_flux(i,j) * cs%flux_factor
00618
00619     if ((sfc_state%ocean_mass(i,j) > cs%col_mass_melt_threshold) .and. &
00620         (iss%area_shelf_h(i,j) > 0.0) .and.  (cs%isthermo)) then
00621
00622       ! Set melt to zero above a cutoff pressure (CS%Rho_ocn*CS%cutoff_depth*CS%g_Earth).
00623       ! This is needed for the ISOMIP test case.
00624       if (iss%mass_shelf(i,j) < cs%Rho_ocn*cs%cutoff_depth) then
00625         iss%water_flux(i,j) = 0.0
00626         fluxes%iceshelf_melt(i,j) = 0.0
00627       endif
00628       ! Compute haline driving, which is one of the diags. used in ISOMIP
00629       haline_driving(i,j) = (iss%water_flux(i,j) * sbdry(i,j)) / (cs%Rho_ocn * exch_vel_s(i,j))
00630
00631       !!!!!!!!!!!!!!!!!!!!!!!!!!!!Safety checks !!!!!!!!!!!!!!!!!!!!!!!!
00632       !1)Check if haline_driving computed above is consistent with
00633       ! haline_driving = sfc_state%sss - Sbdry
00634       !if (fluxes%iceshelf_melt(i,j) /= 0.0) then
00635       !   if (haline_driving(i,j) /= (sfc_state%sss(i,j) - Sbdry(i,j))) then
00636       !     write(mesg,*) 'at i,j=',i,j,' haline_driving, sss-Sbdry',haline_driving(i,j), &
00637       !                   (sfc_state%sss(i,j) - Sbdry(i,j))
00638       !     call MOM_error(FATAL, &
00639       !            "shelf_calc_flux: Inconsistency in melt and haline_driving"//trim(mesg))
00640       !   endif
00641       !endif
00642
00643       ! 2) check if |melt| > 0 when ustar_shelf = 0.
00644       ! this should never happen
00645       if ((abs(fluxes%iceshelf_melt(i,j))>0.0) .and. (fluxes%ustar_shelf(i,j) == 0.0)) then
00646         write(mesg,*) "|melt| = ",fluxes%iceshelf_melt(i,j)," > 0 and ustar_shelf = 0. at i,j", i, j
00647         call mom_error(fatal, "shelf_calc_flux: "//trim(mesg))
00648       endif
00649       !!!!!!!!!!!!!!!!!!!!!!!!!!!!End of safety checks !!!!!!!!!!!!!!!!!!!
00650     elseif (iss%area_shelf_h(i,j) > 0.0) then
00651       ! This is grounded ice, that could be modified to melt if a geothermal heat flux were used.
```

```
00652       haline_driving(i,j) = 0.0
00653       iss%water_flux(i,j) = 0.0
00654       fluxes%iceshelf_melt(i,j) = 0.0
00655     endif ! area_shelf_h
00656
00657     ! mass flux [R Z L2 T-1 ~> kg s-1], part of ISOMIP diags.
00658     mass_flux(i,j) = iss%water_flux(i,j) * iss%area_shelf_h(i,j)
00659   enddo ; enddo ! i- and j-loops
00660
00661   if (cs%active_shelf_dynamics .or. cs%override_shelf_movement) then
00662     call cpu_clock_begin(id_clock_pass)
00663     call pass_var(iss%area_shelf_h, g%domain, complete=.false.)
00664     call pass_var(iss%mass_shelf, g%domain)
00665     call cpu_clock_end(id_clock_pass)
00666   endif
00667
00668   ! Melting has been computed, now is time to update thickness and mass
00669   if ( cs%override_shelf_movement .and. (.not.cs%mass_from_file)) then
00670     call change_thickness_using_melt(iss, g, us, us%s_to_T*time_step, fluxes, cs
      %density_ice, cs%debug)
00671
00672     if (cs%debug) then
00673       call hchksum(iss%h_shelf, "h_shelf after change thickness using melt", g%HI, haloshift=0, scale=us
      %Z_to_m)
00674       call hchksum(iss%mass_shelf, "mass_shelf after change thickness using melt", g%HI, haloshift=0, &
00675                    scale=us%RZ_to_kg_m2)
00676     endif
00677   endif
00678
00679   if (cs%debug) call mom_forcing_chksum("Before add shelf flux", fluxes, g, cs%US, haloshift=0)
00680
00681   call add_shelf_flux(g, us, cs, sfc_state, fluxes)
00682
00683   ! now the thermodynamic data is passed on... time to update the ice dynamic quantities
00684
00685   if (cs%active_shelf_dynamics) then
00686     update_ice_vel = .false.
00687     coupled_gl = (cs%GL_couple .and. .not.cs%solo_ice_sheet)
00688
00689     ! advect the ice shelf, and advance the front. Calving will be in here somewhere as well..
00690     ! when we decide on how to do it
00691     call update_ice_shelf(cs%dCS, iss, g, us, us%s_to_T*time_step, time, &
00692                           sfc_state%ocean_mass, coupled_gl)
00693
00694   endif
00695
00696   call enable_averaging(time_step,time,cs%diag)
00697   if (cs%id_shelf_mass > 0) call post_data(cs%id_shelf_mass, iss%mass_shelf, cs%diag)
00698   if (cs%id_area_shelf_h > 0) call post_data(cs%id_area_shelf_h, iss%area_shelf_h, cs%diag)
00699   if (cs%id_ustar_shelf > 0) call post_data(cs%id_ustar_shelf, fluxes%ustar_shelf, cs%diag)
00700   if (cs%id_melt > 0) call post_data(cs%id_melt, fluxes%iceshelf_melt, cs%diag)
00701   if (cs%id_thermal_driving > 0) call post_data(cs%id_thermal_driving, (sfc_state%sst-iss%tfreeze), cs%diag
      )
00702   if (cs%id_Sbdry > 0) call post_data(cs%id_Sbdry, sbdry, cs%diag)
00703   if (cs%id_haline_driving > 0) call post_data(cs%id_haline_driving, haline_driving, cs%diag)
00704   if (cs%id_mass_flux > 0) call post_data(cs%id_mass_flux, mass_flux, cs%diag)
00705   if (cs%id_u_ml > 0) call post_data(cs%id_u_ml, sfc_state%u, cs%diag)
00706   if (cs%id_v_ml > 0) call post_data(cs%id_v_ml, sfc_state%v, cs%diag)
00707   if (cs%id_tfreeze > 0) call post_data(cs%id_tfreeze, iss%tfreeze, cs%diag)
00708   if (cs%id_tfl_shelf > 0) call post_data(cs%id_tfl_shelf, iss%tflux_shelf, cs%diag)
00709   if (cs%id_exch_vel_t > 0) call post_data(cs%id_exch_vel_t, exch_vel_t, cs%diag)
00710   if (cs%id_exch_vel_s > 0) call post_data(cs%id_exch_vel_s, exch_vel_s, cs%diag)
00711   if (cs%id_h_shelf > 0) call post_data(cs%id_h_shelf, iss%h_shelf, cs%diag)
00712   if (cs%id_h_mask > 0) call post_data(cs%id_h_mask,iss%hmask,cs%diag)
00713   call disable_averaging(cs%diag)
00714
00715   if (present(forces)) then
00716     call add_shelf_forces(g, us, cs, forces, do_shelf_area=(cs%active_shelf_dynamics .or. &
00717                                                             cs%override_shelf_movement))
00718   endif
00719
00720   call cpu_clock_end(id_clock_shelf)
00721
00722   if (cs%debug) call mom_forcing_chksum("End of shelf calc flux", fluxes, g, cs%US, haloshift=0)
00723
00724 end subroutine shelf_calc_flux
00725
00726 !> Changes the thickness (mass) of the ice shelf based on sub-ice-shelf melting
00727 subroutine change_thickness_using_melt(ISS, G, US, time_step, fluxes,
       density_ice, debug)
00728   type(ocean_grid_type), intent(inout) :: G  !< The ocean's grid structure.
00729   type(ice_shelf_state), intent(inout) :: ISS !< A structure with elements that describe
00730                                               !! the ice-shelf state
00731   type(unit_scale_type), intent(in)    :: US   !< A dimensional unit scaling type
00732   real,                  intent(in)    :: time_step !< The time step for this update [T ~> s].
00733   type(forcing),         intent(inout) :: fluxes !< structure containing pointers to any possible
00734                                                  !! thermodynamic or mass-flux forcing fields.
```

```
00735   real,                 intent(in)    :: density_ice !< The density of ice-shelf ice [R ~> kg m-3].
00736   logical,     optional, intent(in)    :: debug !< If present and true, write chksums
00737
00738   ! locals
00739   real :: I_rho_ice ! Ice specific volume [R-1 ~> m3 kg-1]
00740   integer :: i, j
00741
00742   i_rho_ice = 1.0 / density_ice
00743
00744
00745   do j=g%jsc,g%jec ; do i=g%isc,g%iec
00746     if ((iss%hmask(i,j) == 1) .or. (iss%hmask(i,j) == 2)) then
00747       ! first, zero out fluxes applied during previous time step
00748       if (associated(fluxes%lprec)) fluxes%lprec(i,j) = 0.0
00749       if (associated(fluxes%sens)) fluxes%sens(i,j) = 0.0
00750       if (associated(fluxes%frac_shelf_h)) fluxes%frac_shelf_h(i,j) = 0.0
00751       if (associated(fluxes%salt_flux)) fluxes%salt_flux(i,j) = 0.0
00752
00753       if (iss%water_flux(i,j) * time_step / density_ice < iss%h_shelf(i,j)) then
00754         iss%h_shelf(i,j) = iss%h_shelf(i,j) - iss%water_flux(i,j) * time_step / density_ice
00755       else
00756         ! the ice is about to melt away, so set thickness, area, and mask to zero
00757         ! NOTE: this is not mass conservative should maybe scale salt & heat flux for this cell
00758         iss%h_shelf(i,j) = 0.0
00759         iss%hmask(i,j) = 0.0
00760         iss%area_shelf_h(i,j) = 0.0
00761       endif
00762       iss%mass_shelf(i,j) = iss%h_shelf(i,j) * density_ice
00763     endif
00764   enddo ; enddo
00765
00766   call pass_var(iss%area_shelf_h, g%domain)
00767   call pass_var(iss%h_shelf, g%domain)
00768   call pass_var(iss%hmask, g%domain)
00769   call pass_var(iss%mass_shelf, g%domain)
00770
00771 end subroutine change_thickness_using_melt
00772
00773 !> This subroutine adds the mechanical forcing fields and perhaps shelf areas, based on
00774 !! the ice state in ice_shelf_CS.
00775 subroutine add_shelf_forces(G, US, CS, forces, do_shelf_area)
00776   type(ocean_grid_type), intent(inout) :: G    !< The ocean's grid structure.
00777   type(unit_scale_type), intent(in)    :: US   !< A dimensional unit scaling type
00778   type(ice_shelf_cs),    pointer       :: CS   !< This module's control structure.
00779   type(mech_forcing),    intent(inout) :: forces !< A structure with the driving mechanical forces
00780   logical, optional,     intent(in)    :: do_shelf_area !< If true find the shelf-covered areas.
00781
00782   real :: kv_rho_ice ! The viscosity of ice divided by its density [L4 T-1 R-1 Z-2 ~> m5 kg-1 s-1].
00783   real :: press_ice  ! The pressure of the ice shelf per unit area of ocean (not ice) [R L2 T-2 ~> Pa].
00784   logical :: find_area ! If true find the shelf areas at u & v points.
00785   type(ice_shelf_state), pointer :: ISS => null() ! A structure with elements that describe
00786                                                   ! the ice-shelf state
00787
00788   integer :: i, j, is, ie, js, je, isd, ied, jsd, jed
00789   is = g%isc ; ie = g%iec ; js = g%jsc ; je = g%jec
00790   isd = g%isd ; jsd = g%jsd ; ied = g%ied ; jed = g%jed
00791
00792   if ((cs%grid%isc /= g%isc) .or. (cs%grid%iec /= g%iec) .or. &
00793       (cs%grid%jsc /= g%jsc) .or. (cs%grid%jec /= g%jec)) &
00794     call mom_error(fatal,"add_shelf_forces: Incompatible ocean and ice shelf grids.")
00795
00796   iss => cs%ISS
00797
00798   find_area = .true. ; if (present(do_shelf_area)) find_area = do_shelf_area
00799
00800   if (find_area) then
00801     ! The frac_shelf is set over the widest possible area. Could it be smaller?
00802     do j=jsd,jed ; do i=isd,ied-1
00803       forces%frac_shelf_u(i,j) = 0.0
00804       if ((g%areaT(i,j) + g%areaT(i+1,j) > 0.0)) & ! .and. (G%areaCu(I,j) > 0.0)) &
00805         forces%frac_shelf_u(i,j) = (iss%area_shelf_h(i,j) + iss%area_shelf_h(i+1,j)) / &
00806                                    (g%areaT(i,j) + g%areaT(i+1,j))
00807     enddo ; enddo
00808     do j=jsd,jed-1 ; do i=isd,ied
00809       forces%frac_shelf_v(i,j) = 0.0
00810       if ((g%areaT(i,j) + g%areaT(i,j+1) > 0.0)) & ! .and. (G%areaCv(i,J) > 0.0)) &
00811         forces%frac_shelf_v(i,j) = (iss%area_shelf_h(i,j) + iss%area_shelf_h(i,j+1)) / &
00812                                    (g%areaT(i,j) + g%areaT(i,j+1))
00813     enddo ; enddo
00814     call pass_vector(forces%frac_shelf_u, forces%frac_shelf_v, g%domain, to_all, cgrid_ne)
00815   endif
00816
00817   do j=js,je ; do i=is,ie
00818     press_ice = (iss%area_shelf_h(i,j) * g%IareaT(i,j)) * (cs%g_Earth * iss%mass_shelf(i,j))
00819     if (associated(forces%p_surf)) then
00820       if (.not.forces%accumulate_p_surf) forces%p_surf(i,j) = 0.0
00821       forces%p_surf(i,j) = forces%p_surf(i,j) + press_ice
```

```
00822     endif
00823     if (associated(forces%p_surf_full)) then
00824       if (.not.forces%accumulate_p_surf) forces%p_surf_full(i,j) = 0.0
00825       forces%p_surf_full(i,j) = forces%p_surf_full(i,j) + press_ice
00826     endif
00827   enddo ; enddo
00828
00829   ! For various reasons, forces%rigidity_ice_[uv] is always updated here. Note
00830   ! that it may have been zeroed out where IOB is translated to forces and
00831   ! contributions from icebergs and the sea-ice pack added subsequently.
00832   !### THE RIGIDITY SHOULD ALSO INCORPORATE AREAL-COVERAGE INFORMATION.
00833   kv_rho_ice = cs%kv_ice / cs%density_ice
00834   do j=js,je ; do i=is-1,ie
00835     if (.not.forces%accumulate_rigidity) forces%rigidity_ice_u(i,j) = 0.0
00836     forces%rigidity_ice_u(i,j) = forces%rigidity_ice_u(i,j) + &
00837             kv_rho_ice * min(iss%mass_shelf(i,j), iss%mass_shelf(i+1,j))
00838   enddo ; enddo
00839   do j=js-1,je ; do i=is,ie
00840     if (.not.forces%accumulate_rigidity) forces%rigidity_ice_v(i,j) = 0.0
00841     forces%rigidity_ice_v(i,j) = forces%rigidity_ice_v(i,j) + &
00842             kv_rho_ice * min(iss%mass_shelf(i,j), iss%mass_shelf(i,j+1))
00843   enddo ; enddo
00844
00845   if (cs%debug) then
00846     call uvchksum("rigidity_ice_[uv]", forces%rigidity_ice_u, forces%rigidity_ice_v, &
00847                   g%HI, symmetric=.true., scale=us%L_to_m**3*us%L_to_Z*us%s_to_T)
00848     call uvchksum("frac_shelf_[uv]", forces%frac_shelf_u, forces%frac_shelf_v, &
00849                   g%HI, symmetric=.true.)
00850   endif
00851
00852 end subroutine add_shelf_forces
00853
00854 !> This subroutine adds the ice shelf pressure to the fluxes type.
00855 subroutine add_shelf_pressure(G, US, CS, fluxes)
00856   type(ocean_grid_type), intent(inout) :: G    !< The ocean's grid structure.
00857   type(unit_scale_type), intent(in)    :: US   !< A dimensional unit scaling type
00858   type(ice_shelf_cs),    intent(in)    :: CS   !< This module's control structure.
00859   type(forcing),         intent(inout) :: fluxes  !< A structure of surface fluxes that may be updated.
00860
00861   real :: press_ice       !< The pressure of the ice shelf per unit area of ocean (not ice) [R L2 T-2 ~>
       Pa].
00862   integer :: i, j, is, ie, js, je, isd, ied, jsd, jed
00863   is = g%isc ; ie = g%iec ; js = g%jsc ; je = g%jec
00864
00865   if ((cs%grid%isc /= g%isc) .or. (cs%grid%iec /= g%iec) .or. &
00866       (cs%grid%jsc /= g%jsc) .or. (cs%grid%jec /= g%jec)) &
00867     call mom_error(fatal,"add_shelf_pressure: Incompatible ocean and ice shelf grids.")
00868
00869   do j=js,je ; do i=is,ie
00870     press_ice = (cs%ISS%area_shelf_h(i,j) * g%IareaT(i,j)) * (cs%g_Earth * cs%ISS%mass_shelf(i,j))
00871     if (associated(fluxes%p_surf)) then
00872       if (.not.fluxes%accumulate_p_surf) fluxes%p_surf(i,j) = 0.0
00873       fluxes%p_surf(i,j) = fluxes%p_surf(i,j) + press_ice
00874     endif
00875     if (associated(fluxes%p_surf_full)) then
00876       if (.not.fluxes%accumulate_p_surf) fluxes%p_surf_full(i,j) = 0.0
00877       fluxes%p_surf_full(i,j) = fluxes%p_surf_full(i,j) + press_ice
00878     endif
00879   enddo ; enddo
00880
00881 end subroutine add_shelf_pressure
00882
00883 !> Updates surface fluxes that are influenced by sub-ice-shelf melting
00884 subroutine add_shelf_flux(G, US, CS, sfc_state, fluxes)
00885   type(ocean_grid_type), intent(inout) :: G    !< The ocean's grid structure.
00886   type(unit_scale_type), intent(in)    :: US   !< A dimensional unit scaling type
00887   type(ice_shelf_cs),    pointer       :: CS   !< This module's control structure.
00888   type(surface),         intent(inout) :: sfc_state !< Surface ocean state
00889   type(forcing),         intent(inout) :: fluxes  !< A structure of surface fluxes that may be
       used/updated.
00890
00891   ! local variables
00892   real :: frac_shelf       !< The fractional area covered by the ice shelf [nondim].
00893   real :: frac_open        !< The fractional area of the ocean that is not covered by the ice shelf
       [nondim].
00894   real :: delta_mass_shelf !< Change in ice shelf mass over one time step [R Z m2 T-1 ~> kg s-1]
00895   real :: balancing_flux   !< The fresh water flux that balances the integrated melt flux [R Z T-1 ~> kg
      m-2 s-1]
00896   real :: balancing_area   !< total area where the balancing flux is applied [m2]
00897   type(time_type) :: dTime !< The time step as a time_type
00898   type(time_type) :: Time0 !< The previous time (Time-dt)
00899   real, dimension(SZDI_(G),SZDJ_(G)) :: bal_frac  !< Fraction of the cell where the mass flux
00900                            !! balancing the net melt flux occurs, 0 to 1 [nondim]
00901   real, dimension(SZDI_(G),SZDJ_(G)) :: last_mass_shelf !< Ice shelf mass
00902                            !! at at previous time (Time-dt) [R Z ~> kg m-2]
00903   real, dimension(SZDI_(G),SZDJ_(G)) :: delta_float_mass   !< The change in the floating mass between
00904                            !! the two timesteps at (Time) and (Time-dt) [R Z ~> kg m-2].
```

```
00905   real, dimension(SZDI_(G),SZDJ_(G))  :: last_h_shelf !< Ice shelf thickness [Z ~> m]
00906                           !! at at previous time (Time-dt)
00907   real, dimension(SZDI_(G),SZDJ_(G))  :: last_hmask !< Ice shelf mask [nondim]
00908                           !! at at previous time (Time-dt)
00909   real, dimension(SZDI_(G),SZDJ_(G))  :: last_area_shelf_h !< Ice shelf area [L2 ~> m2]
00910                           !! at at previous time (Time-dt)
00911   type(ice_shelf_state), pointer :: ISS => null() !< A structure with elements that describe
00912                                           !! the ice-shelf state
00913
00914   character(len=160) :: mesg  ! The text of an error message
00915   integer :: i, j, is, ie, js, je, isd, ied, jsd, jed
00916   is = g%isc ; ie = g%iec ; js = g%jsc ; je = g%jec
00917   isd = g%isd ; jsd = g%jsd ; ied = g%ied ; jed = g%jed
00918
00919   if ((cs%grid%isc /= g%isc) .or. (cs%grid%iec /= g%iec) .or. &
00920       (cs%grid%jsc /= g%jsc) .or. (cs%grid%jec /= g%jec)) &
00921     call mom_error(fatal,"add_shelf_flux: Incompatible ocean and ice shelf grids.")
00922
00923   iss => cs%ISS
00924
00925
00926   call add_shelf_pressure(g, us, cs, fluxes)
00927
00928   ! Determine ustar and the square magnitude of the velocity in the
00929   ! bottom boundary layer. Together these give the TKE source and
00930   ! vertical decay scale.
00931
00932   if (cs%debug) then
00933     if (allocated(sfc_state%taux_shelf) .and. allocated(sfc_state%tauy_shelf)) then
00934       call uvchksum("tau[xy]_shelf", sfc_state%taux_shelf, sfc_state%tauy_shelf, &
00935                     g%HI, haloshift=0, scale=us%RZ_T_to_kg_m2s*us%L_T_to_m_s)
00936     endif
00937   endif
00938
00939   if (cs%active_shelf_dynamics .or. cs%override_shelf_movement) then
00940     do j=jsd,jed ; do i=isd,ied
00941       if (g%areaT(i,j) > 0.0) &
00942         fluxes%frac_shelf_h(i,j) = min(1.0, iss%area_shelf_h(i,j) * g%IareaT(i,j))
00943     enddo ; enddo
00944   endif
00945
00946   if (cs%debug) then
00947     call mom_forcing_chksum("Before adding shelf fluxes", fluxes, g, cs%US, haloshift=0)
00948   endif
00949
00950   do j=js,je ; do i=is,ie ; if (iss%area_shelf_h(i,j) > 0.0) then
00951     ! Replace fluxes intercepted by the ice shelf with fluxes from the ice shelf
00952     frac_shelf = min(1.0, iss%area_shelf_h(i,j) * g%IareaT(i,j))
00953     frac_open = max(0.0, 1.0 - frac_shelf)
00954
00955     if (associated(fluxes%sw)) fluxes%sw(i,j) = frac_open * fluxes%sw(i,j)
00956     if (associated(fluxes%sw_vis_dir)) fluxes%sw_vis_dir(i,j) = frac_open * fluxes%sw_vis_dir(i,j)
00957     if (associated(fluxes%sw_vis_dif)) fluxes%sw_vis_dif(i,j) = frac_open * fluxes%sw_vis_dif(i,j)
00958     if (associated(fluxes%sw_nir_dir)) fluxes%sw_nir_dir(i,j) = frac_open * fluxes%sw_nir_dir(i,j)
00959     if (associated(fluxes%sw_nir_dif)) fluxes%sw_nir_dif(i,j) = frac_open * fluxes%sw_nir_dif(i,j)
00960     if (associated(fluxes%lw)) fluxes%lw(i,j) = frac_open * fluxes%lw(i,j)
00961     if (associated(fluxes%latent)) fluxes%latent(i,j) = frac_open * fluxes%latent(i,j)
00962     if (associated(fluxes%evap)) fluxes%evap(i,j) = frac_open * fluxes%evap(i,j)
00963     if (associated(fluxes%lprec)) then
00964       if (iss%water_flux(i,j) > 0.0) then
00965         fluxes%lprec(i,j) =  frac_shelf*iss%water_flux(i,j)*cs%flux_factor + frac_open * fluxes%lprec(i,j)
00966       else
00967         fluxes%lprec(i,j) = frac_open * fluxes%lprec(i,j)
00968         fluxes%evap(i,j) = fluxes%evap(i,j) + frac_shelf*iss%water_flux(i,j)*cs%flux_factor
00969       endif
00970     endif
00971
00972     if (associated(fluxes%sens)) &
00973       fluxes%sens(i,j) = frac_shelf*iss%tflux_ocn(i,j)*cs%flux_factor + frac_open * fluxes%sens(i,j)
00974     ! The salt flux should be mostly from sea ice, so perhaps none should be intercepted and this should be
      changed.
00975     if (associated(fluxes%salt_flux)) &
00976       fluxes%salt_flux(i,j) = frac_shelf * iss%salt_flux(i,j)*cs%flux_factor + frac_open * fluxes%salt_flux
     (i,j)
00977   endif ; enddo ; enddo
00978
00979   if (cs%debug) then
00980     call hchksum(iss%water_flux, "water_flux add shelf fluxes", g%HI, haloshift=0, scale=us%RZ_T_to_kg_m2s)
00981     call hchksum(iss%tflux_ocn, "tflux_ocn add shelf fluxes", g%HI, haloshift=0, scale=us%QRZ_T_to_W_m2)
00982     call mom_forcing_chksum("After adding shelf fluxes", fluxes, g, cs%US, haloshift=0)
00983   endif
00984
00985   ! Keep sea level constant by removing mass via a balancing flux that might be applied
00986   ! in the open ocean or the sponge region (via virtual precip, vprec). Apply additional
00987   ! salt/heat fluxes so that the resultant surface buoyancy forcing is ~ 0.
00988   ! This is needed for some of the ISOMIP+ experiments.
00989
```

```
00990   if (cs%constant_sea_level) then
00991     if (.not. associated(fluxes%salt_flux)) allocate(fluxes%salt_flux(ie,je))
00992     if (.not. associated(fluxes%vprec)) allocate(fluxes%vprec(ie,je))
00993     fluxes%salt_flux(:,:) = 0.0 ; fluxes%vprec(:,:) = 0.0
00994
00995     ! take into account changes in mass (or thickness) when imposing ice shelf mass
00996     if (cs%override_shelf_movement .and. cs%mass_from_file) then
00997       dtime = real_to_time(cs%time_step)
00998
00999       ! Compute changes in mass after at least one full time step
01000       if (cs%Time > dtime) then
01001         time0 = cs%Time - dtime
01002         do j=js,je ; do i=is,ie
01003           last_hmask(i,j) = iss%hmask(i,j) ; last_area_shelf_h(i,j) = iss%area_shelf_h(i,j)
01004         enddo ; enddo
01005         call time_interp_external(cs%id_read_mass, time0, last_mass_shelf)
01006         do j=js,je ; do i=is,ie
01007         ! This should only be done if time_interp_external did an update.
01008           last_mass_shelf(i,j) = us%kg_m3_to_R*us%m_to_Z * last_mass_shelf(i,j) ! Rescale after time_interp
01009           last_h_shelf(i,j) = last_mass_shelf(i,j) / cs%density_ice
01010         enddo ; enddo
01011
01012         ! apply calving
01013         if (cs%min_thickness_simple_calve > 0.0) then
01014           call ice_shelf_min_thickness_calve(g, last_h_shelf, last_area_shelf_h, last_hmask, &
01015                                              cs%min_thickness_simple_calve, halo=0)
01016           ! convert to mass again
01017           do j=js,je ; do i=is,ie
01018             last_mass_shelf(i,j) = last_h_shelf(i,j) * cs%density_ice
01019           enddo ; enddo
01020         endif
01021
01022         ! get total ice shelf mass at (Time-dt) and (Time), in kg
01023         do j=js,je ; do i=is,ie
01024           ! Just consider the change in the mass of the floating shelf.
01025           if ((sfc_state%ocean_mass(i,j) > cs%min_ocean_mass_float) .and. &
01026               (iss%area_shelf_h(i,j) > 0.0)) then
01027             delta_float_mass(i,j) = iss%mass_shelf(i,j) - last_mass_shelf(i,j)
01028           else
01029             delta_float_mass(i,j) = 0.0
01030           endif
01031         enddo ; enddo
01032         delta_mass_shelf = us%kg_m2s_to_RZ_T*(global_area_integral(delta_float_mass, g, scale=us
      %RZ_to_kg_m2, &
01033                                                                    area=iss%area_shelf_h) / cs%time_step)
01034       else! first time step
01035         delta_mass_shelf = 0.0
01036       endif
01037     else ! ice shelf mass does not change
01038       delta_mass_shelf = 0.0
01039     endif
01040
01041     ! average total melt flux over sponge area
01042     do j=js,je ; do i=is,ie
01043       if ((g%mask2dT(i,j) > 0.0) .AND. (iss%area_shelf_h(i,j) * g%IareaT(i,j) < 1.0)) then
01044          ! Uncomment this for some ISOMIP cases:
01045          !  .AND. (G%geoLonT(i,j) >= 790.0) .AND. (G%geoLonT(i,j) <= 800.0)) then
01046         bal_frac(i,j) = max(1.0 - iss%area_shelf_h(i,j) * g%IareaT(i,j), 0.0)
01047       else
01048         bal_frac(i,j) = 0.0
01049       endif
01050     enddo ; enddo
01051
01052     balancing_area = global_area_integral(bal_frac, g)
01053     if (balancing_area > 0.0) then
01054       balancing_flux = ( us%kg_m2s_to_RZ_T*global_area_integral(iss%water_flux, g, scale=us%RZ_T_to_kg_m2s,
      &
01055                                                                 area=iss%area_shelf_h) + &
01056                          delta_mass_shelf ) / balancing_area
01057     else
01058       balancing_flux = 0.0
01059     endif
01060
01061     ! apply fluxes
01062     do j=js,je ; do i=is,ie
01063       if (bal_frac(i,j) > 0.0) then
01064         ! evap is negative, and vprec has units of [R Z T-1 ~> kg m-2 s-1]
01065         fluxes%vprec(i,j) = -balancing_flux
01066         fluxes%sens(i,j) = fluxes%vprec(i,j) * cs%Cp * cs%T0 ! [ Q R Z T-1 ~> W /m^2 ]
01067         fluxes%salt_flux(i,j) = fluxes%vprec(i,j) * cs%S0*1.0e-3 ! [kgSalt/kg R Z T-1 ~> kgSalt m-2 s-1]
01068       endif
01069     enddo ; enddo
01070
01071     if (cs%debug) then
01072       write(mesg,*) 'Balancing flux (kg/(m^2 s)), dt = ', balancing_flux*us%RZ_T_to_kg_m2s, cs%time_step
01073       call mom_mesg(mesg)
01074       call mom_forcing_chksum("After constant sea level", fluxes, g, cs%US, haloshift=0)
```

```
01075     endif
01076
01077   endif ! constant_sea_level
01078
01079 end subroutine add_shelf_flux
01080
01081
01082 !> Initializes shelf model data, parameters and diagnostics
01083 subroutine initialize_ice_shelf(param_file, ocn_grid, Time, CS, diag, forces, fluxes,
      Time_in, solo_ice_sheet_in)
01084   type(param_file_type),        intent(in)    :: param_file !< A structure to parse for run-time parameters
01085   type(ocean_grid_type),        pointer       :: ocn_grid   !< The calling ocean model's horizontal grid
      structure
01086   type(time_type),              intent(inout) :: Time !< The clock that that will indicate the model time
01087   type(ice_shelf_cs),           pointer       :: CS   !< A pointer to the ice shelf control
      structure
01088   type(diag_ctrl),    target,   intent(in)    :: diag !< A structure that is used to regulate the
      diagnostic output.
01089   type(forcing),      optional, intent(inout) :: fluxes !< A structure containing pointers to any possible
01090                                                    !! thermodynamic or mass-flux forcing fields.
01091   type(mech_forcing), optional, intent(inout) :: forces !< A structure with the driving mechanical forces
01092   type(time_type),    optional, intent(in)    :: Time_in !< The time at initialization.
01093   logical,            optional, intent(in)    :: solo_ice_sheet_in !< If present, this indicates whether
01094                                                    !! a solo ice-sheet driver.
01095
01096   type(ocean_grid_type), pointer :: G  => null(), og  => null() ! Pointers to grids for convenience.
01097   type(unit_scale_type), pointer :: US => null() ! Pointer to a structure containing
01098                                                  ! various unit conversion factors
01099   type(ice_shelf_state), pointer :: ISS => null() !< A structure with elements that describe
01100                                          !! the ice-shelf state
01101   type(directories)  :: dirs
01102   type(dyn_horgrid_type), pointer :: dG => null()
01103   real    :: Z_rescale  ! A rescaling factor for heights from the representation in
01104                         ! a restart file to the internal representation in this run.
01105   real    :: RZ_rescale ! A rescaling factor for mass loads from the representation in
01106                         ! a restart file to the internal representation in this run.
01107   real    :: L_rescale  ! A rescaling factor for horizontal lengths from the representation in
01108                         ! a restart file to the internal representation in this run.
01109   real :: meltrate_conversion ! The conversion factor to use for in the melt rate diagnostic.
01110   real :: dz_ocean_min_float ! The minimum ocean thickness above which the ice shelf is considered
01111                         ! to be floating when CONST_SEA_LEVEL = True [Z ~> m].
01112   real :: cdrag, drag_bg_vel
01113   logical :: new_sim, save_IC, var_force
01114   !This include declares and sets the variable "version".
01115 # include "version_variable.h"
01116   character(len=200) :: config
01117   character(len=200) :: IC_file,filename,inputdir
01118   character(len=40)  :: mdl = "MOM_ice_shelf"  ! This module's name.
01119   integer :: i, j, is, ie, js, je, isd, ied, jsd, jed, Isdq, Iedq, Jsdq, Jedq
01120   integer :: wd_halos(2)
01121   logical :: read_TideAmp, shelf_mass_is_dynamic, debug
01122   character(len=240) :: Tideamp_file
01123   real    :: utide  ! A tidal velocity [L T-1 ~> m s-1]
01124   real    :: col_thick_melt_thresh ! An ocean column thickness below which iceshelf melting
01125                     ! does not occur [Z ~> m]
01126   if (associated(cs)) then
01127     call mom_error(fatal, "MOM_ice_shelf.F90, initialize_ice_shelf: "// &
01128                           "called with an associated control structure.")
01129     return
01130   endif
01131   allocate(cs)
01132
01133   !   Go through all of the infrastructure initialization calls, since this is
01134   ! being treated as an independent component that just happens to use the
01135   ! MOM's grid and infrastructure.
01136   call get_mom_input(dirs=dirs)
01137
01138   ! Determining the internal unit scaling factors for this run.
01139   call unit_scaling_init(param_file, cs%US)
01140
01141   ! Set up the ice-shelf domain and grid
01142   wd_halos(:)=0
01143   call mom_domains_init(cs%grid%domain, param_file, min_halo=wd_halos, symmetric=grid_sym_)
01144   ! call diag_mediator_init(CS%grid,param_file,CS%diag)
01145   ! this needs to be fixed - will probably break when not using coupled driver 0
01146   call mom_grid_init(cs%grid, param_file, cs%US)
01147
01148   call create_dyn_horgrid(dg, cs%grid%HI)
01149   call clone_mom_domain(cs%grid%Domain, dg%Domain)
01150
01151   call set_grid_metrics(dg, param_file, cs%US)
01152   ! call set_diag_mediator_grid(CS%grid, CS%diag)
01153
01154   ! The ocean grid possibly uses different symmetry.
01155   if (associated(ocn_grid)) then ; cs%ocn_grid => ocn_grid
01156   else ; cs%ocn_grid => cs%grid ; endif
01157
```

```
01158   ! Convenience pointers
01159   g => cs%grid
01160   og => cs%ocn_grid
01161   us => cs%US
01162
01163   if (is_root_pe()) then
01164     write(0,*) 'OG: ', og%isd, og%isc, og%iec, og%ied, og%jsd, og%jsc, og%jsd, og%jed
01165     write(0,*) 'IG: ', g%isd, g%isc, g%iec, g%ied, g%jsd, g%jsc, g%jsd, g%jed
01166   endif
01167
01168   cs%diag => diag
01169
01170   ! Are we being called from the solo ice-sheet driver? When called by the ocean
01171   ! model solo_ice_sheet_in is not preset.
01172   cs%solo_ice_sheet = .false.
01173   if (present(solo_ice_sheet_in)) cs%solo_ice_sheet = solo_ice_sheet_in
01174
01175   if (present(time_in)) time = time_in
01176
01177   is = g%isc ; ie = g%iec ; js = g%jsc ; je = g%jec
01178   isd = g%isd ; jsd = g%jsd ; ied = g%ied ; jed = g%jed
01179   isdq = g%IsdB ; iedq = g%IedB ; jsdq = g%JsdB ; jedq = g%JedB
01180
01181   cs%override_shelf_movement = .false. ; cs%active_shelf_dynamics = .false.
01182
01183   call log_version(param_file, mdl, version, "")
01184   call get_param(param_file, mdl, "DEBUG", debug, default=.false.)
01185   call get_param(param_file, mdl, "DEBUG_IS", cs%debug, &
01186                  "If true, write verbose debugging messages for the ice shelf.", &
01187                  default=debug)
01188   call get_param(param_file, mdl, "DYNAMIC_SHELF_MASS", shelf_mass_is_dynamic, &
01189                  "If true, the ice sheet mass can evolve with time.", &
01190                  default=.false.)
01191   if (shelf_mass_is_dynamic) then
01192     call get_param(param_file, mdl, "OVERRIDE_SHELF_MOVEMENT", cs%override_shelf_movement, &
01193                  "If true, user provided code specifies the ice-shelf "//&
01194                  "movement instead of the dynamic ice model.", default=.false.)
01195     cs%active_shelf_dynamics = .not.cs%override_shelf_movement
01196     call get_param(param_file, mdl, "GROUNDING_LINE_INTERPOLATE", cs%GL_regularize, &
01197                  "If true, regularize the floatation condition at the "//&
01198                  "grounding line as in Goldberg Holland Schoof 2009.", default=.false.)
01199     call get_param(param_file, mdl, "GROUNDING_LINE_COUPLE", cs%GL_couple, &
01200                  "If true, let the floatation condition be determined by "//&
01201                  "ocean column thickness. This means that update_OD_ffrac "//&
01202                  "will be called.  GL_REGULARIZE and GL_COUPLE are exclusive.", &
01203                  default=.false., do_not_log=cs%GL_regularize)
01204     if (cs%GL_regularize) cs%GL_couple = .false.
01205   endif
01206
01207   call get_param(param_file, mdl, "SHELF_THERMO", cs%isthermo, &
01208                  "If true, use a thermodynamically interactive ice shelf.", &
01209                  default=.false.)
01210   call get_param(param_file, mdl, "LATENT_HEAT_FUSION", cs%Lat_fusion, &
01211                  "The latent heat of fusion.", units="J/kg", default=hlf, scale=us%J_kg_to_Q)
01212   call get_param(param_file, mdl, "SHELF_THREE_EQN", cs%threeeq, &
01213                  "If true, use the three equation expression of "//&
01214                  "consistency to calculate the fluxes at the ice-ocean "//&
01215                  "interface.", default=.true.)
01216   call get_param(param_file, mdl, "SHELF_INSULATOR", cs%insulator, &
01217                  "If true, the ice shelf is a perfect insulatior "//&
01218                  "(no conduction).", default=.false.)
01219   call get_param(param_file, mdl, "MELTING_CUTOFF_DEPTH", cs%cutoff_depth, &
01220                  "Depth above which the melt is set to zero (it must be >= 0) "//&
01221                  "Default value won't affect the solution.", units="m", default=0.0, scale=us%m_to_Z)
01222   if (cs%cutoff_depth < 0.) &
01223     call mom_error(warning,"Initialize_ice_shelf: MELTING_CUTOFF_DEPTH must be >= 0.")
01224
01225   call get_param(param_file, mdl, "CONST_SEA_LEVEL", cs%constant_sea_level, &
01226                  "If true, apply evaporative, heat and salt fluxes in "//&
01227                  "the sponge region. This will avoid a large increase "//&
01228                  "in sea level. This option is needed for some of the "//&
01229                  "ISOMIP+ experiments (Ocean3 and Ocean4). "//&
01230                  "IMPORTANT: it is not currently possible to do "//&
01231                  "prefect restarts using this flag.", default=.false.)
01232   call get_param(param_file, mdl, "MIN_OCEAN_FLOAT_THICK", dz_ocean_min_float, &
01233                  "The minimum ocean thickness above which the ice shelf is considered to be "//&
01234                  "floating when CONST_SEA_LEVEL = True.", &
01235                  default=0.1, units="m", scale=us%m_to_Z, do_not_log=.not.cs%constant_sea_level)
01236
01237   call get_param(param_file, mdl, "ISOMIP_S_SUR_SPONGE", cs%S0, &
01238                  "Surface salinity in the restoring region.", &
01239                 default=33.8, units='ppt', do_not_log=.true.)
01240
01241   call get_param(param_file, mdl, "ISOMIP_T_SUR_SPONGE", cs%T0, &
01242                 "Surface temperature in the restoring region.", &
01243                 default=-1.9, units='degC', do_not_log=.true.)
01244
```

```
01245   call get_param(param_file, mdl, "SHELF_3EQ_GAMMA", cs%const_gamma, &
01246                  "If true, user specifies a constant nondimensional heat-transfer coefficient "//&
01247                  "(GAMMA_T_3EQ), from which the default salt-transfer coefficient is set "//&
01248                  "as GAMMA_T_3EQ/35. This is used with SHELF_THREE_EQN.", default=.false.)
01249   if (cs%threeeq) then
01250     call get_param(param_file, mdl, "SHELF_S_ROOT", cs%find_salt_root, &
01251                  "If SHELF_S_ROOT = True, salinity at the ice/ocean interface (Sbdry) "//&
01252                  "is computed from a quadratic equation. Otherwise, the previous "//&
01253                  "interactive method to estimate Sbdry is used.", default=.false.)
01254   else
01255     call get_param(param_file, mdl, "SHELF_2EQ_GAMMA_T", cs%gamma_t, &
01256                  "If SHELF_THREE_EQN is false, this the fixed turbulent "//&
01257                  "exchange velocity at the ice-ocean interface.", &
01258                  units="m s-1", scale=us%m_to_Z*us%T_to_s, fail_if_missing=.true.)
01259   endif
01260   if (cs%const_gamma .or. cs%find_salt_root) then
01261     call get_param(param_file, mdl, "SHELF_3EQ_GAMMA_T", cs%Gamma_T_3EQ, &
01262                  "Nondimensional heat-transfer coefficient.", &
01263                   units="nondim", default=2.2e-2)
01264     call get_param(param_file, mdl, "SHELF_3EQ_GAMMA_S", cs%Gamma_S_3EQ, &
01265                  "Nondimensional salt-transfer coefficient.", &
01266                  default=cs%Gamma_T_3EQ/35.0, units="nondim")
01267   endif
01268
01269   call get_param(param_file, mdl, "ICE_SHELF_MASS_FROM_FILE", &
01270                  cs%mass_from_file, "Read the mass of the "//&
01271                  "ice shelf (every time step) from a file.", default=.false.)
01272
01273   if (cs%find_salt_root) then ! read liquidus coeffs.
01274      call get_param(param_file, mdl, "TFREEZE_S0_P0", cs%TFr_0_0, &
01275                  "this is the freezing potential temperature at "//&
01276                  "S=0, P=0.", units="degC", default=0.0, do_not_log=.true.)
01277     call get_param(param_file, mdl, "DTFREEZE_DS", cs%dTFr_dS, &
01278                  "this is the derivative of the freezing potential temperature with salinity.", &
01279                  units="degC psu-1", default=-0.054, do_not_log=.true.)
01280     call get_param(param_file, mdl, "DTFREEZE_DP", cs%dTFr_dp, &
01281                  "this is the derivative of the freezing potential temperature with pressure.", &
01282                  units="degC Pa-1", default=0.0, scale=us%RL2_T2_to_Pa, do_not_log=.true.)
01283   endif
01284
01285   call get_param(param_file, mdl, "G_EARTH", cs%g_Earth, &
01286                  "The gravitational acceleration of the Earth.", &
01287                  units="m s-2", default = 9.80, scale=us%m_s_to_L_T**2*us%Z_to_m)
01288   call get_param(param_file, mdl, "C_P", cs%Cp, &
01289                  "The heat capacity of sea water, approximated as a constant. "//&
01290                  "The default value is from the TEOS-10 definition of conservative temperature.", &
01291                  units="J kg-1 K-1", default=3991.86795711963, scale=us%J_kg_to_Q)
01292   call get_param(param_file, mdl, "RHO_0", cs%Rho_ocn, &
01293                  "The mean ocean density used with BOUSSINESQ true to "//&
01294                  "calculate accelerations and the mass for conservation "//&
01295                  "properties, or with BOUSSINSEQ false to convert some "//&
01296                  "parameters from vertical units of m to kg m-2.", &
01297                  units="kg m-3", default=1035.0, scale=us%kg_m3_to_R)
01298   call get_param(param_file, mdl, "C_P_ICE", cs%Cp_ice, &
01299                  "The heat capacity of ice.", units="J kg-1 K-1", scale=us%J_kg_to_Q, &
01300                  default=2.10e3)
01301   if (cs%constant_sea_level) cs%min_ocean_mass_float = dz_ocean_min_float*cs%Rho_ocn
01302
01303   call get_param(param_file, mdl, "ICE_SHELF_FLUX_FACTOR", cs%flux_factor, &
01304                  "Non-dimensional factor applied to shelf thermodynamic "//&
01305                  "fluxes.", units="none", default=1.0)
01306
01307   call get_param(param_file, mdl, "KV_ICE", cs%kv_ice, &
01308                  "The viscosity of the ice.", &
01309                  units="m2 s-1", default=1.0e10, scale=us%Z_to_L**2*us%m_to_L**2*us%T_to_s)
01310   call get_param(param_file, mdl, "KV_MOLECULAR", cs%kv_molec, &
01311                  "The molecular kinimatic viscosity of sea water at the "//&
01312                  "freezing temperature.", units="m2 s-1", default=1.95e-6, scale=us%m2_s_to_Z2_T)
01313   call get_param(param_file, mdl, "ICE_SHELF_SALINITY", cs%Salin_ice, &
01314                  "The salinity of the ice inside the ice shelf.", units="psu", &
01315                  default=0.0)
01316   call get_param(param_file, mdl, "ICE_SHELF_TEMPERATURE", cs%Temp_ice, &
01317                  "The temperature at the center of the ice shelf.", &
01318                  units = "degC", default=-15.0)
01319   call get_param(param_file, mdl, "KD_SALT_MOLECULAR", cs%kd_molec_salt, &
01320                  "The molecular diffusivity of salt in sea water at the "//&
01321                  "freezing point.", units="m2 s-1", default=8.02e-10, scale=us%m2_s_to_Z2_T)
01322   call get_param(param_file, mdl, "KD_TEMP_MOLECULAR", cs%kd_molec_temp, &
01323                  "The molecular diffusivity of heat in sea water at the "//&
01324                  "freezing point.", units="m2 s-1", default=1.41e-7, scale=us%m2_s_to_Z2_T)
01325   call get_param(param_file, mdl, "DT_FORCING", cs%time_step, &
01326                  "The time step for changing forcing, coupling with other "//&
01327                  "components, or potentially writing certain diagnostics. "//&
01328                  "The default value is given by DT.", units="s", default=0.0)
01329
01330   call get_param(param_file, mdl, "COL_THICK_MELT_THRESHOLD", col_thick_melt_thresh, &
01331                  "The minimum ocean column thickness where melting is allowed.", &
```

```
01332                 units="m", scale=us%m_to_Z, default=0.0)
01333   cs%col_mass_melt_threshold =  cs%Rho_ocn * col_thick_melt_thresh
01334
01335   call get_param(param_file, mdl, "READ_TIDEAMP", read_tideamp, &
01336                 "If true, read a file (given by TIDEAMP_FILE) containing "//&
01337                 "the tidal amplitude with INT_TIDE_DISSIPATION.", default=.false.)
01338
01339   call safe_alloc_ptr(cs%utide,isd,ied,jsd,jed)   ; cs%utide(:,:) = 0.0
01340
01341   if (read_tideamp) then
01342     call get_param(param_file, mdl, "TIDEAMP_FILE", tideamp_file, &
01343                 "The path to the file containing the spatially varying "//&
01344                 "tidal amplitudes.", &
01345                 default="tideamp.nc")
01346     call get_param(param_file, mdl, "INPUTDIR", inputdir, default=".")
01347     inputdir = slasher(inputdir)
01348     tideamp_file = trim(inputdir) // trim(tideamp_file)
01349     call mom_read_data(tideamp_file, 'tideamp', cs%utide, g%domain, timelevel=1, scale=us%m_s_to_L_T)
01350   else
01351     call get_param(param_file, mdl, "UTIDE", utide, &
01352                 "The constant tidal amplitude used with INT_TIDE_DISSIPATION.", &
01353                 units="m s-1", default=0.0 , scale=us%m_s_to_L_T)
01354     cs%utide(:,:) = utide
01355   endif
01356
01357   call eos_init(param_file, cs%eqn_of_state)
01358
01359   !! new parameters that need to be in MOM_input
01360
01361   if (cs%active_shelf_dynamics) then
01362
01363     call get_param(param_file, mdl, "DENSITY_ICE", cs%density_ice, &
01364                 "A typical density of ice.", units="kg m-3", default=917.0, scale=us%kg_m3_to_R)
01365
01366     call get_param(param_file, mdl, "INPUT_FLUX_ICE_SHELF", cs%input_flux, &
01367                 "volume flux at upstream boundary", units="m2 s-1", default=0.)
01368     call get_param(param_file, mdl, "INPUT_THICK_ICE_SHELF", cs%input_thickness, &
01369                 "flux thickness at upstream boundary", units="m", default=1000.)
01370   else
01371     ! This is here because of inconsistent defaults.  I don't know why.  RWH
01372     call get_param(param_file, mdl, "DENSITY_ICE", cs%density_ice, &
01373                 "A typical density of ice.", units="kg m-3", default=900.0, scale=us%kg_m3_to_R)
01374   endif
01375   call get_param(param_file, mdl, "MIN_THICKNESS_SIMPLE_CALVE", &
01376                 cs%min_thickness_simple_calve, &
01377                  "Min thickness rule for the very simple calving law",&
01378                  units="m", default=0.0, scale=us%m_to_Z)
01379
01380   call get_param(param_file, mdl, "USTAR_SHELF_BG", cs%ustar_bg, &
01381                 "The minimum value of ustar under ice shelves.", &
01382                 units="m s-1", default=0.0, scale=us%m_to_Z*us%T_to_s)
01383   call get_param(param_file, mdl, "CDRAG_SHELF", cdrag, &
01384        "CDRAG is the drag coefficient relating the magnitude of "//&
01385        "the velocity field to the surface stress.", units="nondim", &
01386        default=0.003)
01387   cs%cdrag = cdrag
01388   if (cs%ustar_bg <= 0.0) then
01389     call get_param(param_file, mdl, "DRAG_BG_VEL_SHELF", drag_bg_vel, &
01390                 "DRAG_BG_VEL is either the assumed bottom velocity (with "//&
01391                 "LINEAR_DRAG) or an unresolved  velocity that is "//&
01392                 "combined with the resolved velocity to estimate the "//&
01393                 "velocity magnitude.", units="m s-1", default=0.0, scale=us%m_to_Z*us%T_to_s)
01394     if (cs%cdrag*drag_bg_vel > 0.0) cs%ustar_bg = sqrt(cs%cdrag)*drag_bg_vel
01395   endif
01396
01397   ! Allocate and initialize state variables to default values
01398   call ice_shelf_state_init(cs%ISS, cs%grid)
01399   iss => cs%ISS
01400
01401   ! Allocate the arrays for passing ice-shelf data through the forcing type.
01402   if (.not. cs%solo_ice_sheet) then
01403     call mom_mesg("MOM_ice_shelf.F90, initialize_ice_shelf: allocating fluxes.")
01404      ! GMM: the following assures that water/heat fluxes are just allocated
01405      ! when SHELF_THERMO = True. These fluxes are necessary if one wants to
01406      ! use either ENERGETICS_SFC_PBL (ALE mode) or BULKMIXEDLAYER (layer mode).
01407     if (present(fluxes)) &
01408       call allocate_forcing_type(cs%ocn_grid, fluxes, ustar=.true., shelf=.true., &
01409                                  press=.true., water=cs%isthermo, heat=cs%isthermo)
01410     if (present(forces)) &
01411       call allocate_mech_forcing(cs%ocn_grid, forces, ustar=.true., shelf=.true., press=.true.)
01412   else
01413     call mom_mesg("MOM_ice_shelf.F90, initialize_ice_shelf: allocating fluxes in solo mode.")
01414     if (present(fluxes)) &
01415       call allocate_forcing_type(g, fluxes, ustar=.true., shelf=.true., press=.true.)
01416     if (present(forces)) &
01417       call allocate_mech_forcing(g, forces, ustar=.true., shelf=.true., press=.true.)
01418   endif
```

```
01419
01420   ! Set up the bottom depth, G%D either analytically or from file
01421   call mom_initialize_topography(dg%bathyT, g%max_depth, dg, param_file)
01422   call rescale_dyn_horgrid_bathymetry(dg, us%Z_to_m)
01423
01424   ! Set up the Coriolis parameter, G%f, usually analytically.
01425   call mom_initialize_rotation(dg%CoriolisBu, dg, param_file, us)
01426   ! This copies grid elements, including bathyT and CoriolisBu from dG to CS%grid.
01427   call copy_dyngrid_to_mom_grid(dg, cs%grid, us)
01428
01429   call destroy_dyn_horgrid(dg)
01430
01431   ! Set up the restarts.
01432   call restart_init(param_file, cs%restart_CSp, "Shelf.res")
01433   call register_restart_field(iss%mass_shelf, "shelf_mass", .true., cs%restart_CSp, &
01434                               "Ice shelf mass", "kg m-2")
01435   call register_restart_field(iss%area_shelf_h, "shelf_area", .true., cs%restart_CSp, &
01436                               "Ice shelf area in cell", "m2")
01437   call register_restart_field(iss%h_shelf, "h_shelf", .true., cs%restart_CSp, &
01438                               "ice sheet/shelf thickness", "m")
01439   call register_restart_field(us%m_to_Z_restart, "m_to_Z", .false., cs%restart_CSp, &
01440                               "Height unit conversion factor", "Z meter-1")
01441   call register_restart_field(us%m_to_L_restart, "m_to_L", .false., cs%restart_CSp, &
01442                               "Length unit conversion factor", "L meter-1")
01443   call register_restart_field(us%kg_m3_to_R_restart, "kg_m3_to_R", .false., cs%restart_CSp, &
01444                               "Density unit conversion factor", "R m3 kg-1")
01445   if (cs%active_shelf_dynamics) then
01446     call register_restart_field(iss%hmask, "h_mask", .true., cs%restart_CSp, &
01447                                 "ice sheet/shelf thickness mask" ,"none")
01448   endif
01449
01450   if (cs%active_shelf_dynamics) then
01451     ! Allocate CS%dCS and specify additional restarts for ice shelf dynamics
01452     call register_ice_shelf_dyn_restarts(g, param_file, cs%dCS, cs%restart_CSp)
01453   endif
01454
01455   !GMM - I think we do not need to save ustar_shelf and iceshelf_melt in the restart file
01456   !if (.not. CS%solo_ice_sheet) then
01457   !  call register_restart_field(fluxes%ustar_shelf, "ustar_shelf", .false., CS%restart_CSp, &
01458   !                              "Friction velocity under ice shelves", "m s-1")
01459   !endif
01460
01461   cs%restart_output_dir = dirs%restart_output_dir
01462
01463   new_sim = .false.
01464   if ((dirs%input_filename(1:1) == 'n') .and. &
01465       (len_trim(dirs%input_filename) == 1)) new_sim = .true.
01466
01467   if (cs%override_shelf_movement .and. cs%mass_from_file) then
01468
01469     ! initialize the ids for reading shelf mass from a netCDF
01470     call initialize_shelf_mass(g, param_file, cs, iss)
01471
01472     if (new_sim) then
01473       ! new simulation, initialize ice thickness as in the static case
01474       call initialize_ice_thickness(iss%h_shelf, iss%area_shelf_h, iss%hmask, g, us, param_file)
01475
01476     ! next make sure mass is consistent with thickness
01477       do j=g%jsd,g%jed ; do i=g%isd,g%ied
01478         if ((iss%hmask(i,j) == 1) .or. (iss%hmask(i,j) == 2)) then
01479           iss%mass_shelf(i,j) = iss%h_shelf(i,j)*cs%density_ice
01480         endif
01481       enddo ; enddo
01482
01483       if (cs%min_thickness_simple_calve > 0.0) &
01484         call ice_shelf_min_thickness_calve(g, iss%h_shelf, iss%area_shelf_h, iss%hmask, &
01485                                            cs%min_thickness_simple_calve)
01486     endif
01487   endif
01488
01489   if (cs%active_shelf_dynamics) then
01490     ! the only reason to initialize boundary conds is if the shelf is dynamic - MJH
01491
01492     ! call initialize_ice_shelf_boundary ( CS%u_face_mask_bdry, CS%v_face_mask_bdry, &
01493     !                                      CS%u_flux_bdry_val, CS%v_flux_bdry_val, &
01494     !                                      CS%u_bdry_val, CS%v_bdry_val, CS%h_bdry_val, &
01495     !                                      ISS%hmask, G, param_file)
01496
01497   endif
01498
01499   if (new_sim .and. (.not. (cs%override_shelf_movement .and. cs%mass_from_file))) then
01500
01501     ! This model is initialized internally or from a file.
01502     call initialize_ice_thickness(iss%h_shelf, iss%area_shelf_h, iss%hmask, g, us, param_file)
01503
01504     ! next make sure mass is consistent with thickness
01505     do j=g%jsd,g%jed ; do i=g%isd,g%ied
```

```
01506       if ((iss%hmask(i,j) == 1) .or. (iss%hmask(i,j) == 2)) then
01507         iss%mass_shelf(i,j) = iss%h_shelf(i,j)*cs%density_ice
01508       endif
01509     enddo ; enddo
01510
01511   ! else ! Previous block for new_sim=.T., this block restores the state.
01512   elseif (.not.new_sim) then
01513     ! This line calls a subroutine that reads the initial conditions from a restart file.
01514     call mom_mesg("MOM_ice_shelf.F90, initialize_ice_shelf: Restoring ice shelf from file.")
01515     call restore_state(dirs%input_filename, dirs%restart_input_dir, time, &
01516                        g, cs%restart_CSp)
01517
01518     if ((us%m_to_Z_restart /= 0.0) .and. (us%m_to_Z_restart /= us%m_to_Z)) then
01519       z_rescale = us%m_to_Z / us%m_to_Z_restart
01520       do j=g%jsc,g%jec ; do i=g%isc,g%iec
01521         iss%h_shelf(i,j) = z_rescale * iss%h_shelf(i,j)
01522       enddo ; enddo
01523     endif
01524
01525     if ((us%m_to_Z_restart*us%kg_m3_to_R_restart /= 0.0) .and. &
01526         (us%m_to_Z*us%kg_m3_to_R /= us%m_to_Z_restart * us%kg_m3_to_R_restart)) then
01527       rz_rescale = us%m_to_Z*us%kg_m3_to_R / (us%m_to_Z_restart * us%kg_m3_to_R_restart)
01528       do j=g%jsc,g%jec ; do i=g%isc,g%iec
01529         iss%mass_shelf(i,j) = rz_rescale * iss%mass_shelf(i,j)
01530       enddo ; enddo
01531     endif
01532
01533     if ((us%m_to_L_restart /= 0.0) .and. (us%m_to_L_restart /= us%m_to_L)) then
01534       l_rescale = us%m_to_L / us%m_to_L_restart
01535       do j=g%jsc,g%jec ; do i=g%isc,g%iec
01536         iss%area_shelf_h(i,j) = l_rescale**2 * iss%area_shelf_h(i,j)
01537       enddo ; enddo
01538     endif
01539
01540   endif ! .not. new_sim
01541
01542   cs%Time = time
01543
01544   call cpu_clock_begin(id_clock_pass)
01545   call pass_var(iss%area_shelf_h, g%domain)
01546   call pass_var(iss%h_shelf, g%domain)
01547   call pass_var(iss%mass_shelf, g%domain)
01548   call pass_var(iss%hmask, g%domain)
01549   call pass_var(g%bathyT, g%domain)
01550   call cpu_clock_end(id_clock_pass)
01551
01552   do j=jsd,jed ; do i=isd,ied
01553     if (iss%area_shelf_h(i,j) > g%areaT(i,j)) then
01554       call mom_error(warning,"Initialize_ice_shelf: area_shelf_h exceeds G%areaT.")
01555       iss%area_shelf_h(i,j) = g%areaT(i,j)
01556     endif
01557   enddo ; enddo
01558   if (present(fluxes)) then ; do j=jsd,jed ; do i=isd,ied
01559     if (g%areaT(i,j) > 0.0) fluxes%frac_shelf_h(i,j) = iss%area_shelf_h(i,j) / g%areaT(i,j)
01560   enddo ; enddo ; endif
01561
01562   if (cs%debug) then
01563     call hchksum(fluxes%frac_shelf_h, "IS init: frac_shelf_h", g%HI, haloshift=0)
01564   endif
01565
01566   if (present(forces)) &
01567     call add_shelf_forces(g, us, cs, forces, do_shelf_area=.not.cs%solo_ice_sheet)
01568
01569   if (present(fluxes)) call add_shelf_pressure(g, us, cs, fluxes)
01570
01571   if (cs%active_shelf_dynamics .and. .not.cs%isthermo) then
01572     iss%water_flux(:,:) = 0.0
01573   endif
01574
01575   if (shelf_mass_is_dynamic) &
01576     call initialize_ice_shelf_dyn(param_file, time, iss, cs%dCS, g, us, diag, new_sim, solo_ice_sheet_in)
01577
01578   call fix_restart_unit_scaling(us)
01579
01580   call get_param(param_file, mdl, "SAVE_INITIAL_CONDS", save_ic, &
01581                  "If true, save the ice shelf initial conditions.", &
01582                  default=.false.)
01583   if (save_ic) call get_param(param_file, mdl, "SHELF_IC_OUTPUT_FILE", ic_file,&
01584                  "The name-root of the output file for the ice shelf "//&
01585                  "initial conditions.", default="MOM_Shelf_IC")
01586
01587   if (save_ic .and. .not.((dirs%input_filename(1:1) == 'r') .and. &
01588                           (len_trim(dirs%input_filename) == 1))) then
01589     call save_restart(dirs%output_directory, cs%Time, g, &
01590                       cs%restart_CSp, filename=ic_file)
01591   endif
01592
```

```
01593
01594   cs%id_area_shelf_h = register_diag_field('ocean_model', 'area_shelf_h', cs%diag%axesT1, cs%Time, &
01595      'Ice Shelf Area in cell', 'meter-2', conversion=us%L_to_m**2)
01596   cs%id_shelf_mass = register_diag_field('ocean_model', 'shelf_mass', cs%diag%axesT1, cs%Time, &
01597      'mass of shelf', 'kg/m^2', conversion=us%RZ_to_kg_m2)
01598   cs%id_h_shelf = register_diag_field('ocean_model', 'h_shelf', cs%diag%axesT1, cs%Time, &
01599        'ice shelf thickness', 'm', conversion=us%Z_to_m)
01600   cs%id_mass_flux = register_diag_field('ocean_model', 'mass_flux', cs%diag%axesT1,&
01601      cs%Time, 'Total mass flux of freshwater across the ice-ocean interface.', &
01602      'kg/s', conversion=us%RZ_T_to_kg_m2s*us%L_to_m**2)
01603
01604   if (cs%const_gamma) then ! use ISOMIP+ eq. with rho_fw = 1000. kg m-3
01605     meltrate_conversion = 86400.0*365.0*us%Z_to_m*us%s_to_T / (1000.0*us%kg_m3_to_R)
01606   else ! use original eq.
01607     meltrate_conversion = 86400.0*365.0*us%Z_to_m*us%s_to_T / cs%density_ice
01608   endif
01609   cs%id_melt = register_diag_field('ocean_model', 'melt', cs%diag%axesT1, cs%Time, &
01610      'Ice Shelf Melt Rate', 'm yr-1', conversion= meltrate_conversion)
01611   cs%id_thermal_driving = register_diag_field('ocean_model', 'thermal_driving', cs%diag%axesT1, cs%Time, &
01612      'pot. temp. in the boundary layer minus freezing pot. temp. at the ice-ocean interface.', 'Celsius')
01613   cs%id_haline_driving = register_diag_field('ocean_model', 'haline_driving', cs%diag%axesT1, cs%Time, &
01614      'salinity in the boundary layer minus salinity at the ice-ocean interface.', 'psu')
01615   cs%id_Sbdry = register_diag_field('ocean_model', 'sbdry', cs%diag%axesT1, cs%Time, &
01616      'salinity at the ice-ocean interface.', 'psu')
01617   cs%id_u_ml = register_diag_field('ocean_model', 'u_ml', cs%diag%axesCu1, cs%Time, &
01618      'Eastward vel. in the boundary layer (used to compute ustar)', 'm s-1', conversion=us%L_T_to_m_s)
01619   cs%id_v_ml = register_diag_field('ocean_model', 'v_ml', cs%diag%axesCv1, cs%Time, &
01620      'Northward vel. in the boundary layer (used to compute ustar)', 'm s-1', conversion=us%L_T_to_m_s)
01621   cs%id_exch_vel_s = register_diag_field('ocean_model', 'exch_vel_s', cs%diag%axesT1, cs%Time, &
01622      'Sub-shelf salinity exchange velocity', 'm s-1', conversion=us%Z_to_m*us%s_to_T)
01623   cs%id_exch_vel_t = register_diag_field('ocean_model', 'exch_vel_t', cs%diag%axesT1, cs%Time, &
01624      'Sub-shelf thermal exchange velocity', 'm s-1' , conversion=us%Z_to_m*us%s_to_T)
01625   cs%id_tfreeze = register_diag_field('ocean_model', 'tfreeze', cs%diag%axesT1, cs%Time, &
01626      'In Situ Freezing point at ice shelf interface', 'degC')
01627   cs%id_tfl_shelf = register_diag_field('ocean_model', 'tflux_shelf', cs%diag%axesT1, cs%Time, &
01628      'Heat conduction into ice shelf', 'W m-2', conversion=-us%QRZ_T_to_W_m2)
01629   cs%id_ustar_shelf = register_diag_field('ocean_model', 'ustar_shelf', cs%diag%axesT1, cs%Time, &
01630      'Fric vel under shelf', 'm/s', conversion=us%Z_to_m*us%s_to_T)
01631   if (cs%active_shelf_dynamics) then
01632     cs%id_h_mask = register_diag_field('ocean_model', 'h_mask', cs%diag%axesT1, cs%Time, &
01633        'ice shelf thickness mask', 'none')
01634   endif
01635
01636   id_clock_shelf = cpu_clock_id('Ice shelf', grain=clock_component)
01637   id_clock_pass = cpu_clock_id(' Ice shelf halo updates', grain=clock_routine)
01638
01639 end subroutine initialize_ice_shelf
01640
01641 !> Initializes shelf mass based on three options (file, zero and user)
01642 subroutine initialize_shelf_mass(G, param_file, CS, ISS, new_sim)
01643
01644   type(ocean_grid_type), intent(in) :: G   !< The ocean's grid structure.
01645   type(param_file_type), intent(in) :: param_file !< A structure to parse for run-time parameters
01646   type(ice_shelf_cs),    pointer    :: CS !< A pointer to the ice shelf control structure
01647   type(ice_shelf_state), intent(inout) :: ISS !< The ice shelf state type that is being updated
01648   logical,     optional, intent(in) :: new_sim !< If present and false, this run is being restarted
01649
01650   integer :: i, j, is, ie, js, je
01651   logical :: read_shelf_area, new_sim_2
01652   character(len=240) :: config, inputdir, shelf_file, filename
01653   character(len=120) :: shelf_mass_var  ! The name of shelf mass in the file.
01654   character(len=120) :: shelf_area_var ! The name of shelf area in the file.
01655   character(len=40)  :: mdl = "MOM_ice_shelf"
01656   is = g%isc ; ie = g%iec ; js = g%jsc ; je = g%jec
01657
01658   new_sim_2 = .true. ; if (present(new_sim)) new_sim_2 = new_sim
01659
01660   call get_param(param_file, mdl, "ICE_SHELF_CONFIG", config, &
01661                  "A string that specifies how the ice shelf is "//&
01662                  "initialized. Valid options include:\n"//&
01663                  " \tfile\t Read from a file.\n"//&
01664                  " \tzero\t Set shelf mass to 0 everywhere.\n"//&
01665                  " \tUSER\t Call USER_initialize_shelf_mass.\n", &
01666                  fail_if_missing=.true.)
01667
01668   select case ( trim(config) )
01669     case ("file")
01670
01671       call time_interp_external_init()
01672
01673       call get_param(param_file, mdl, "INPUTDIR", inputdir, default=".")
01674       inputdir = slasher(inputdir)
01675
01676       call get_param(param_file, mdl, "SHELF_FILE", shelf_file, &
01677              "If DYNAMIC_SHELF_MASS = True, OVERRIDE_SHELF_MOVEMENT = True "//&
01678              "and ICE_SHELF_MASS_FROM_FILE = True, this is the file from "//&
01679              "which to read the shelf mass and area.", &
```

```
01680                default="shelf_mass.nc")
01681       call get_param(param_file, mdl, "SHELF_MASS_VAR", shelf_mass_var, &
01682                  "The variable in SHELF_FILE with the shelf mass.", &
01683                  default="shelf_mass")
01684       call get_param(param_file, mdl, "READ_SHELF_AREA", read_shelf_area, &
01685                  "If true, also read the area covered by ice-shelf from SHELF_FILE.", &
01686                  default=.false.)
01687
01688       filename = trim(slasher(inputdir))//trim(shelf_file)
01689       call log_param(param_file, mdl, "INPUTDIR/SHELF_FILE", filename)
01690
01691       cs%id_read_mass = init_external_field(filename, shelf_mass_var, &
01692                           domain=g%Domain%mpp_domain, verbose=cs%debug)
01693
01694       if (read_shelf_area) then
01695          call get_param(param_file, mdl, "SHELF_AREA_VAR", shelf_area_var, &
01696                   "The variable in SHELF_FILE with the shelf area.", &
01697                   default="shelf_area")
01698
01699          cs%id_read_area = init_external_field(filename,shelf_area_var, &
01700                              domain=g%Domain%mpp_domain)
01701       endif
01702
01703       if (.not.file_exists(filename, g%Domain)) call mom_error(fatal, &
01704            " initialize_shelf_mass: Unable to open "//trim(filename))
01705
01706     case ("zero")
01707       do j=js,je ; do i=is,ie
01708         iss%mass_shelf(i,j) = 0.0
01709         iss%area_shelf_h(i,j) = 0.0
01710       enddo ; enddo
01711
01712     case ("USER")
01713       call user_initialize_shelf_mass(iss%mass_shelf, iss%area_shelf_h, &
01714                     iss%h_shelf, iss%hmask, g, cs%US, cs%user_CS, param_file, new_sim_2)
01715
01716     case default ;  call mom_error(fatal,"initialize_ice_shelf: "// &
01717       "Unrecognized ice shelf setup "//trim(config))
01718   end select
01719
01720 end subroutine initialize_shelf_mass
01721
01722 !> Updates the ice shelf mass using data from a file.
01723 subroutine update_shelf_mass(G, US, CS, ISS, Time)
01724   type(ocean_grid_type), intent(inout) :: G   !< The ocean's grid structure.
01725   type(unit_scale_type), intent(in)    :: US   !< A dimensional unit scaling type
01726   type(ice_shelf_cs),    intent(in)    :: CS  !< A pointer to the ice shelf control structure
01727   type(ice_shelf_state), intent(inout) :: ISS !< The ice shelf state type that is being updated
01728   type(time_type),       intent(in)    :: Time !< The current model time
01729
01730   ! local variables
01731   integer :: i, j, is, ie, js, je
01732   is = g%isc ; ie = g%iec ; js = g%jsc ; je = g%jec
01733
01734   call time_interp_external(cs%id_read_mass, time, iss%mass_shelf)
01735   ! This should only be done if time_interp_external did an update.
01736   do j=js,je ; do i=is,ie
01737     iss%mass_shelf(i,j) = us%kg_m3_to_R*us%m_to_Z * iss%mass_shelf(i,j) ! Rescale after time_interp
01738   enddo ; enddo
01739
01740   do j=js,je ; do i=is,ie
01741     iss%area_shelf_h(i,j) = 0.0
01742     iss%hmask(i,j) = 0.
01743     if (iss%mass_shelf(i,j) > 0.0) then
01744       iss%area_shelf_h(i,j) = g%areaT(i,j)
01745       iss%h_shelf(i,j) = iss%mass_shelf(i,j) / cs%density_ice
01746       iss%hmask(i,j) = 1.
01747     endif
01748   enddo ; enddo
01749
01750   !call USER_update_shelf_mass(ISS%mass_shelf, ISS%area_shelf_h, ISS%h_shelf, &
01751   !                            ISS%hmask, CS%grid, CS%user_CS, Time, .true.)
01752
01753   if (cs%min_thickness_simple_calve > 0.0) then
01754     call ice_shelf_min_thickness_calve(g, iss%h_shelf, iss%area_shelf_h, iss%hmask, &
01755                                        cs%min_thickness_simple_calve, halo=0)
01756   endif
01757
01758   call pass_var(iss%area_shelf_h, g%domain)
01759   call pass_var(iss%h_shelf, g%domain)
01760   call pass_var(iss%hmask, g%domain)
01761   call pass_var(iss%mass_shelf, g%domain)
01762
01763 end subroutine update_shelf_mass
01764
01765 !> Save the ice shelf restart file
01766 subroutine ice_shelf_save_restart(CS, Time, directory, time_stamped, filename_suffix)
```

```
01767   type(ice_shelf_cs),         pointer    :: CS !< ice shelf control structure
01768   type(time_type),            intent(in) :: Time !< model time at this call
01769   character(len=*), optional, intent(in) :: directory !< An optional directory into which to write
01770                                                 !! these restart files.
01771   logical,          optional, intent(in) :: time_stamped !< f true, the restart file names include
01772                                                 !! a unique time stamp.  The default is false.
01773   character(len=*), optional, intent(in) :: filename_suffix !< An optional suffix (e.g., a
01774                                                 !! time-stamp) to append to the restart file names.
01775   ! local variables
01776   type(ocean_grid_type), pointer :: G => null()
01777   character(len=200) :: restart_dir
01778
01779   g => cs%grid
01780
01781   if (present(directory)) then ; restart_dir = directory
01782   else ; restart_dir = cs%restart_output_dir ; endif
01783
01784   call save_restart(restart_dir, time, cs%grid, cs%restart_CSp, time_stamped)
01785
01786 end subroutine ice_shelf_save_restart
01787
01788 !> Deallocates all memory associated with this module
01789 subroutine ice_shelf_end(CS)
01790   type(ice_shelf_cs), pointer   :: CS !< A pointer to the ice shelf control structure
01791
01792   if (.not.associated(cs)) return
01793
01794   call ice_shelf_state_end(cs%ISS)
01795
01796   if (cs%active_shelf_dynamics) call ice_shelf_dyn_end(cs%dCS)
01797
01798   deallocate(cs)
01799
01800 end subroutine ice_shelf_end
01801
01802 !> This routine is for stepping a stand-alone ice shelf model without an ocean.
01803 subroutine solo_step_ice_shelf(CS, time_interval, nsteps, Time, min_time_step_in)
01804   type(ice_shelf_cs), pointer    :: CS      !< A pointer to the ice shelf control structure
01805   type(time_type), intent(in)    :: time_interval !< The time interval for this update [s].
01806   integer,         intent(inout) :: nsteps  !< The running number of ice shelf steps.
01807   type(time_type), intent(inout) :: Time    !< The current model time
01808   real,  optional, intent(in)    :: min_time_step_in !< The minimum permitted time step [T ~> s].
01809
01810   type(ocean_grid_type), pointer :: G => null()  ! A pointer to the ocean's grid structure
01811   type(unit_scale_type), pointer :: US => null() ! Pointer to a structure containing
01812                                                  ! various unit conversion factors
01813   type(ice_shelf_state), pointer :: ISS => null() !< A structure with elements that describe
01814                                                   !! the ice-shelf state
01815   real :: remaining_time    ! The remaining time in this call [T ~> s]
01816   real :: time_step         ! The internal time step during this call [T ~> s]
01817   real :: min_time_step     ! The minimal required timestep that would indicate a fatal problem [T ~> s]
01818   character(len=240) :: mesg
01819   logical :: update_ice_vel ! If true, it is time to update the ice shelf velocities.
01820   logical :: coupled_GL     ! If true the grouding line position is determined based on
01821                             ! coupled ice-ocean dynamics.
01822   integer :: is, iec, js, jec, i, j
01823
01824   g => cs%grid
01825   us => cs%US
01826   iss => cs%ISS
01827   is = g%isc ; iec = g%iec ; js = g%jsc ; jec = g%jec
01828
01829   remaining_time = us%s_to_T*time_type_to_real(time_interval)
01830
01831   if (present (min_time_step_in)) then
01832     min_time_step = min_time_step_in
01833   else
01834     min_time_step = 1000.0*us%s_to_T ! At 1 km resolution this would imply ice is moving at ~1 meter per
       second
01835   endif
01836
01837   write (mesg,*) "TIME in ice shelf call, yrs: ", time_type_to_real(time)/(365. * 86400.)
01838   call mom_mesg("solo_step_ice_shelf: "//mesg, 5)
01839
01840   do while (remaining_time > 0.0)
01841     nsteps = nsteps+1
01842
01843     ! If time_interval is not too long, this is unnecessary.
01844     time_step = min(ice_time_step_cfl(cs%dCS, iss, g), remaining_time)
01845
01846     write (mesg,*) "Ice model timestep = ", us%T_to_s*time_step, " seconds"
01847     if ((time_step < min_time_step) .and. (time_step < remaining_time))  then
01848       call mom_error(fatal, "MOM_ice_shelf:solo_step_ice_shelf: abnormally small timestep "//mesg)
01849     else
01850       call mom_mesg("solo_step_ice_shelf: "//mesg, 5)
01851     endif
01852
```

```
01853     remaining_time = remaining_time - time_step
01854
01855     ! If the last mini-timestep is a day or less, we cannot expect velocities to change by much.
01856     ! Do not update the velocities if the last step is very short.
01857     update_ice_vel = ((time_step > min_time_step) .or. (remaining_time > 0.0))
01858     coupled_gl = .false.
01859
01860     call update_ice_shelf(cs%dCS, iss, g, us, time_step, time, must_update_vel=update_ice_vel)
01861
01862     call enable_averages(time_step, time, cs%diag)
01863     if (cs%id_area_shelf_h > 0) call post_data(cs%id_area_shelf_h, iss%area_shelf_h, cs%diag)
01864     if (cs%id_h_shelf > 0) call post_data(cs%id_h_shelf, iss%h_shelf, cs%diag)
01865     if (cs%id_h_mask > 0) call post_data(cs%id_h_mask, iss%hmask, cs%diag)
01866     call disable_averaging(cs%diag)
01867
01868   enddo
01869
01870 end subroutine solo_step_ice_shelf
01871
01872 !> \namespace mom_ice_shelf
01873 !!
01874 !! \section section_ICE_SHELF
01875 !!
01876 !! This module implements the thermodynamic aspects of ocean/ice-shelf
01877 !! inter-actions using the MOM framework and coding style.
01878 !!
01879 !! Derived from code by Chris Little, early 2010.
01880 !!
01881 !!   The ice-sheet dynamics subroutines do the following:
01882 !!  initialize_shelf_mass - Initializes the ice shelf mass distribution.
01883 !!      - Initializes h_shelf, h_mask, area_shelf_h
01884 !!      - CURRENTLY: initializes mass_shelf as well, but this is unnecessary, as mass_shelf is initialized based on
01885 !!             h_shelf and density_ice immediately afterwards. Possibly subroutine should be renamed
01886 !!  update_shelf_mass - updates ice shelf mass via netCDF file
01887 !!                      USER_update_shelf_mass (TODO).
01888 !!    solo_step_ice_shelf - called only in ice-only mode.
01889 !!    shelf_calc_flux - after melt rate & fluxes are calculated, ice dynamics are done. currently mass_shelf is
01890 !! updated immediately after ice_shelf_advect in fully dynamic mode.
01891 !!
01892 !!   NOTES: be aware that hmask(:,:) has a number of functions; it is used for front advancement,
01893 !! for subroutines in the velocity solve, and for thickness boundary conditions (this last one may be removed).
01894 !! in other words, interfering with its updates will have implications you might not expect.
01895 !!
01896 !!  Overall issues: Many variables need better documentation and units and the
01897 !!                  subgrid on which they are discretized.
01898 !!
01899 !! \subsection section_ICE_SHELF_equations ICE_SHELF equations
01900 !!
01901 !! The three fundamental equations are:
01902 !! Heat flux
01903 !! \f[ \qquad \rho_w  C_{pw} \gamma_T (T_w - T_b) = \rho_i  \dot{m}  L_f \f]
01904 !! Salt flux
01905 !! \f[  \qquad \rho_w \gamma_s (S_w - S_b) =  \rho_i \dot{m} S_b \f]
01906 !! Freezing temperature
01907 !! \f[  \qquad T_b = a S_b + b + c P \f]
01908 !!
01909 !! where ....
01910 !!
01911 !! \subsection section_ICE_SHELF_references References
01912 !!
01913 !! Asay-Davis, Xylar S., Stephen L. Cornford, Benjamin K. Galton-Fenzi, Rupert M. Gladstone, G. Hilmar Gudmundsson,
01914 !! David M. Holland, Paul R. Holland, and Daniel F. Martin. Experimental design for three interrelated marine ice sheet
01915 !! and ocean model intercomparison projects: MISMIP v. 3 (MISMIP+), ISOMIP v. 2 (ISOMIP+) and MISOMIP v. 1 (MISOMIP1).
01916 !! Geoscientific Model Development 9, no. 7 (2016): 2471.
01917 !!
01918 !! Goldberg, D. N., et al. Investigation of land ice-ocean interaction with a fully coupled ice-ocean model: 1.
01919 !!  Model description and behavior. Journal of Geophysical Research: Earth Surface 117.F2 (2012).
01920 !!
01921 !! Goldberg, D. N., et al. Investigation of land ice-ocean interaction with a fully coupled ice-ocean model: 2.
01922 !! Sensitivity to external forcings. Journal of Geophysical Research: Earth Surface 117.F2 (2012).
01923 !!
01924 !! Holland, David M., and Adrian Jenkins. Modeling thermodynamic ice-ocean interactions at the base of an ice shelf.
01925 !! Journal of Physical Oceanography 29.8 (1999): 1787-1800.
01926
01927 end module mom_ice_shelf
```

44.40 /home/cermak/src/MOM6/src/parameterizations/lateral/MOM_MEKE.F90 File Reference

```
#include <MOM_memory.h>
#include "version_variable.h"
```

Data Types

- type mom_meke::meke_cs

  Control structure that contains MEKE parameters and diagnostics handles.

Modules

- module mom_meke

  Implements the Mesoscale Eddy Kinetic Energy framework with topographic beta effect included in computing beta in Rhines scale.

Functions/Subroutines

- subroutine, public mom_meke::step_forward_meke (MEKE, h, SN_u, SN_v, visc, dt, G, GV, US, CS, hu, hv)

  Integrates forward-in-time the MEKE eddy energy equation. See MEKE equations.
- subroutine mom_meke::meke_equilibrium (CS, MEKE, G, GV, US, SN_u, SN_v, drag_rate_visc, I_mass)

  Calculates the equilibrium solutino where the source depends only on MEKE diffusivity and there is no lateral diffusion of MEKE. Results is in MEKEMEKE.
- subroutine mom_meke::meke_equilibrium_restoring (CS, G, US, SN_u, SN_v)
- subroutine mom_meke::meke_lengthscales (CS, MEKE, G, GV, US, SN_u, SN_v, EKE, bottomFac2, barotr↩ Fac2, LmixScale)

  Calculates the eddy mixing length scale and γ_b and γ_t functions that are ratios of either bottom or barotropic eddy energy to the column eddy energy, respectively. See MEKE equations.
- subroutine mom_meke::meke_lengthscales_0d (CS, US, area, beta, depth, Rd_dx, SN, EKE, bottomFac2, barotrFac2, LmixScale, Lrhines, Leady)

  Calculates the eddy mixing length scale and γ_b and γ_t functions that are ratios of either bottom or barotropic eddy energy to the column eddy energy, respectively. See MEKE equations.
- logical function, public mom_meke::meke_init (Time, G, US, param_file, diag, CS, MEKE, restart_CS)

  Initializes the MOM_MEKE module and reads parameters. Returns True if module is to be used, otherwise returns False.
- subroutine, public mom_meke::meke_alloc_register_restart (HI, param_file, MEKE, restart_CS)

  Allocates memory and register restart fields for the MOM_MEKE module.
- subroutine, public mom_meke::meke_end (MEKE, CS)

  Deallocates any variables allocated in MEKE_init or MEKE_alloc_register_restart.

44.41 MOM_MEKE.F90

```
00001 !> Implements the Mesoscale Eddy Kinetic Energy framework
00002 !! with topographic beta effect included in computing beta in Rhines scale
00003
00004 module mom_meke
00005
00006 ! This file is part of MOM6. See LICENSE.md for the license.
00007
00008 use mom_debugging,      only : hchksum, uvchksum
00009 use mom_cpu_clock,      only : cpu_clock_id, cpu_clock_begin, cpu_clock_end, clock_routine
00010 use mom_diag_mediator, only : post_data, register_diag_field, safe_alloc_ptr
00011 use mom_diag_mediator, only : diag_ctrl, time_type
00012 use mom_domains,        only : create_group_pass, do_group_pass, group_pass_type
00013 use mom_error_handler, only : mom_error, fatal, warning, note, mom_mesg
00014 use mom_file_parser,    only : read_param, get_param, log_version, param_file_type
00015 use mom_grid,           only : ocean_grid_type
00016 use mom_hor_index,      only : hor_index_type
00017 use mom_io,             only : vardesc, var_desc
00018 use mom_restart,        only : mom_restart_cs, register_restart_field, query_initialized
00019 use mom_unit_scaling,  only : unit_scale_type
00020 use mom_variables,      only : vertvisc_type
00021 use mom_verticalgrid,  only : verticalgrid_type
00022 use mom_meke_types,     only : meke_type
00023
00024 implicit none ; private
00025
00026 #include <MOM_memory.h>
00027
00028 public step_forward_meke, meke_init,
      meke_alloc_register_restart, meke_end
00029
00030 !> Control structure that contains MEKE parameters and diagnostics handles
00031 type, public :: meke_cs ; private
00032   ! Parameters
00033   real, dimension(:,:), pointer :: equilibrium_value => null() !< The equilbrium value
00034                                 !! of MEKE to be calculated at each time step [L2 T-2 ~> m2 s-2]
00035   real :: meke_frcoeff  !< Efficiency of conversion of ME into MEKE [nondim]
00036   real :: meke_gmcoeff  !< Efficiency of conversion of PE into MEKE [nondim]
00037   real :: meke_gmecoeff !< Efficiency of conversion of MEKE into ME by GME [nondim]
00038   real :: meke_damping  !< Local depth-independent MEKE dissipation rate [T-1 ~> s-1].
00039   real :: meke_cd_scale !< The ratio of the bottom eddy velocity to the column mean
00040                         !! eddy velocity, i.e. sqrt(2*MEKE). This should be less than 1
00041                         !! to account for the surface intensification of MEKE.
00042   real :: meke_cb       !< Coefficient in the \f$\gamma_{bot}\f$ expression [nondim]
00043   real :: meke_min_gamma!< Minimum value of gamma_b^2 allowed [nondim]
00044   real :: meke_ct       !< Coefficient in the \f$\gamma_{bt}\f$ expression [nondim]
00045   logical :: visc_drag  !< If true use the vertvisc_type to calculate bottom drag.
00046   logical :: meke_geometric !< If true, uses the GM coefficient formulation from the GEOMETRIC
00047                         !! framework (Marshall et al., 2012)
00048   real    :: meke_geometric_alpha !< The nondimensional coefficient governing the efficiency of the
00049                         !! GEOMETRIC thickness diffusion.
00050   logical :: meke_equilibrium_alt !< If true, use an alternative calculation for the
00051                         !! equilibrium value of MEKE.
00052   logical :: meke_equilibrium_restoring !< If true, restore MEKE back to its
      equilibrium value,
00053                         !!  which is calculated at each time step.
00054   logical :: gm_src_alt !< If true, use the GM energy conversion form S^2*N^2*kappa rather
00055                         !! than the streamfunction for the MEKE GM source term.
00056   logical :: rd_as_max_scale !< If true the length scale can not exceed the
00057                         !! first baroclinic deformation radius.
00058   logical :: use_old_lscale !< Use the old formula for mixing length scale.
00059   logical :: use_min_lscale !< Use simple minimum for mixing length scale.
00060   real :: cdrag         !< The bottom drag coefficient for MEKE [nondim].
00061   real :: meke_bgsrc    !< Background energy source for MEKE [L2 T-3 ~> W kg-1] (= m2 s-3).
00062   real :: meke_dtscale  !< Scale factor to accelerate time-stepping [nondim]
00063   real :: meke_khcoeff  !< Scaling factor to convert MEKE into Kh [nondim]
00064   real :: meke_uscale   !< MEKE velocity scale for bottom drag [L T-1 ~> m s-1]
00065   real :: meke_kh       !< Background lateral diffusion of MEKE [L2 T-1 ~> m2 s-1]
00066   real :: meke_k4       !< Background bi-harmonic diffusivity (of MEKE) [L4 T-1 ~> m4 s-1]
00067   real :: khmeke_fac    !< A factor relating MEKE%Kh to the diffusivity used for
00068                         !! MEKE itself [nondim].
00069   real :: viscosity_coeff_ku !< The scaling coefficient in the expression for
00070                         !! viscosity used to parameterize lateral harmonic momentum mixing
00071                         !! by unresolved eddies represented by MEKE.
00072   real :: viscosity_coeff_au !< The scaling coefficient in the expression for
00073                         !! viscosity used to parameterize lateral biharmonic momentum mixing
00074                         !! by unresolved eddies represented by MEKE.
00075   real :: lfixed        !< Fixed mixing length scale [L ~> m].
00076   real :: adeform       !< Weighting towards deformation scale of mixing length [nondim]
00077   real :: arhines       !< Weighting towards Rhines scale of mixing length [nondim]
00078   real :: africt        !< Weighting towards frictional arrest scale of mixing length [nondim]
00079   real :: aeady         !< Weighting towards Eady scale of mixing length [nondim]
00080   real :: agrid         !< Weighting towards grid scale of mixing length [nondim]
00081   real :: meke_advection_factor !< A scaling in front of the advection of MEKE [nondim]
00082   real :: meke_topographic_beta !< Weight for how much topographic beta is considered
```

```
00083                                !! when computing beta in Rhines scale [nondim]
00084   real :: meke_restoring_rate !< Inverse of the timescale used to nudge MEKE toward its equilibrium value
      [s-1].
00085
00086   logical :: kh_flux_enabled !< If true, lateral diffusive MEKE flux is enabled.
00087   logical :: initialize !< If True, invokes a steady state solver to calculate MEKE.
00088   logical :: debug      !< If true, write out checksums of data for debugging
00089
00090   type(diag_ctrl), pointer :: diag => null() !< A type that regulates diagnostics output
00091   !>@{ Diagnostic handles
00092   integer :: id_meke = -1, id_ue = -1, id_kh = -1, id_src = -1
00093   integer :: id_ub = -1, id_ut = -1
00094   integer :: id_gm_src = -1, id_mom_src = -1, id_gme_snk = -1, id_decay = -1
00095   integer :: id_khmeke_u = -1, id_khmeke_v = -1, id_ku = -1, id_au = -1
00096   integer :: id_le = -1, id_gamma_b = -1, id_gamma_t = -1
00097   integer :: id_lrhines = -1, id_leady = -1
00098   integer :: id_meke_equilibrium = -1
00099   !>@}
00100
00101   ! Infrastructure
00102   integer :: id_clock_pass !< Clock for group pass calls
00103   type(group_pass_type) :: pass_meke !< Group halo pass handle for MEKE%MEKE and maybe MEKE%Kh_diff
00104   type(group_pass_type) :: pass_kh   !< Group halo pass handle for MEKE%Kh, MEKE%Ku, and/or MEKE%Au
00105 end type meke_cs
00106
00107 contains
00108
00109 !> Integrates forward-in-time the MEKE eddy energy equation.
00110 !! See \ref section_MEKE_equations.
00111 subroutine step_forward_meke(MEKE, h, SN_u, SN_v, visc, dt, G, GV, US, CS, hu, hv)
00112   type(meke_type),                          pointer       :: MEKE !< MEKE data.
00113   type(ocean_grid_type),                    intent(inout) :: G    !< Ocean grid.
00114   type(verticalgrid_type),                  intent(in)    :: GV   !< Ocean vertical grid structure.
00115   type(unit_scale_type),                    intent(in)    :: US   !< A dimensional unit
      scaling type
00116   real, dimension(SZI_(G),SZJ_(G),SZK_(G)), intent(in)    :: h    !< Layer thickness [H ~> m or kg m-2].
00117   real, dimension(SZIB_(G),SZJ_(G)),        intent(in)    :: SN_u !< Eady growth rate at u-points [T-1 ~>
      s-1].
00118   real, dimension(SZI_(G),SZJB_(G)),        intent(in)    :: SN_v !< Eady growth rate at v-points [T-1 ~>
      s-1].
00119   type(vertvisc_type),                      intent(in)    :: visc !< The vertical viscosity type.
00120   real,                                     intent(in)    :: dt   !< Model(baroclinic) time-step [T ~> s].
00121   type(meke_cs),                            pointer       :: CS   !< MEKE control structure.
00122   real, dimension(SZIB_(G),SZJ_(G),SZK_(G)), intent(in)   :: hu   !< Accumlated zonal mass flux [H L2 ~> m3
      or kg].
00123   real, dimension(SZI_(G),SZJB_(G),SZK_(G)), intent(in)   :: hv   !< Accumlated meridional mass flux [H L2
      ~> m3 or kg]
00124
00125   ! Local variables
00126   real, dimension(SZI_(G),SZJ_(G)) :: &
00127     mass, &         ! The total mass of the water column [R Z ~> kg m-2].
00128     I_mass, &       ! The inverse of mass [R-1 Z-1 ~> m2 kg-1].
00129     src, &          ! The sum of all MEKE sources [L2 T-3 ~> W kg-1] (= m2 s-3).
00130     MEKE_decay, &   ! A diagnostic of the MEKE decay timescale [T-1 ~> s-1].
00131     drag_rate_visc, & ! Near-bottom velocity contribution to bottom dratg [L T-1 ~> m s-1]
00132     drag_rate, &    ! The MEKE spindown timescale due to bottom drag [T-1 ~> s-1].
00133     drag_rate_J15, &  ! The MEKE spindown timescale due to bottom drag with the Jansen 2015 scheme.
00134                     ! Unfortunately, as written the units seem inconsistent. [T-1 ~> s-1].
00135     del2meke, &     ! Laplacian of MEKE, used for bi-harmonic diffusion [T-2 ~> s-2].
00136     del4meke, &     ! Time-integrated MEKE tendency arising from the biharmonic of MEKE [L2 T-2 ~> m2 s-2].
00137     lmixscale, &    ! Eddy mixing length [L ~> m].
00138     barotrfac2, &   ! Ratio of EKE_barotropic / EKE [nondim]
00139     bottomfac2, &   ! Ratio of EKE_bottom / EKE [nondim]
00140     tmp             ! Temporary variable for diagnostic computation
00141
00142   real, dimension(SZIB_(G),SZJ_(G)) :: &
00143     MEKE_uflux, &   ! The zonal advective and diffusive flux of MEKE with units of [R Z L4 T-3 ~> kg m-2
      s-3].
00144                     ! In one place, MEKE_uflux is used as temporary work space with units of [L2 T-2 ~> m2
      s-2].
00145     kh_u, &         ! The zonal diffusivity that is actually used [L2 T-1 ~> m2 s-1].
00146     barohu, &       ! Depth integrated accumulated zonal mass flux [R Z L2 ~> kg].
00147     drag_vel_u      ! A (vertical) viscosity associated with bottom drag at
00148                     ! u-points [Z T-1 ~> m s-1].
00149   real, dimension(SZI_(G),SZJB_(G)) :: &
00150     MEKE_vflux, &   ! The meridional advective and diffusive flux of MEKE with units of [R Z L4 T-3 ~> kg
      m-2 s-3].
00151                     ! In one place, MEKE_vflux is used as temporary work space with units of [L2 T-2 ~> m2
      s-2].
00152     kh_v, &         ! The meridional diffusivity that is actually used [L2 T-1 ~> m2 s-1].
00153     barohv, &       ! Depth integrated accumulated meridional mass flux [R Z L2 ~> kg].
00154     drag_vel_v      ! A (vertical) viscosity associated with bottom drag at
00155                     ! v-points [Z T-1 ~> m s-1].
00156   real :: Kh_here   ! The local horizontal viscosity [L2 T-1 ~> m2 s-1]
00157   real :: Inv_Kh_max ! The inverse of the local horizontal viscosity [T L-2 ~> s m-2]
00158   real :: K4_here   ! The local horizontal biharmonic viscosity [L4 T-1 ~> m4 s-1]
00159   real :: Inv_K4_max ! The inverse of the local horizontal biharmonic viscosity [T L-4 ~> s m-4]
```

```
00160   real :: cdrag2
00161   real :: advFac    ! The product of the advection scaling factor and 1/dt [T-1 ~> s-1]
00162   real :: mass_neglect ! A negligible mass [R Z ~> kg m-2].
00163   real :: ldamping  ! The MEKE damping rate [T-1 ~> s-1].
00164   real :: Rho0      ! A density used to convert mass to distance [R ~> kg m-3].
00165   real :: sdt       ! dt to use locally [T ~> s] (could be scaled to accelerate)
00166   real :: sdt_damp  ! dt for damping [T ~> s] (sdt could be split).
00167   logical :: use_drag_rate ! Flag to indicate drag_rate is finite
00168   integer :: i, j, k, is, ie, js, je, Isq, Ieq, Jsq, Jeq, nz
00169
00170   is = g%isc ; ie = g%iec ; js = g%jsc ; je = g%jec ; nz = g%ke
00171   isq = g%IscB ; ieq = g%IecB ; jsq = g%JscB ; jeq = g%JecB
00172
00173   if (.not.associated(cs)) call mom_error(fatal, &
00174          "MOM_MEKE: Module must be initialized before it is used.")
00175   if (.not.associated(meke)) call mom_error(fatal, &
00176          "MOM_MEKE: MEKE must be initialized before it is used.")
00177
00178   if ((cs%MEKE_damping > 0.0) .or. (cs%MEKE_Cd_scale > 0.0) .or. (cs%MEKE_Cb>0.) &
00179       .or. cs%visc_drag) then
00180     use_drag_rate = .true.
00181   else
00182     use_drag_rate = .false.
00183   endif
00184
00185   ! Only integrate the MEKE equations if MEKE is required.
00186   if (.not.associated(meke%MEKE)) then
00187 !   call MOM_error(FATAL, "MOM_MEKE: MEKE%MEKE is not associated!")
00188     return
00189   endif
00190
00191     if (cs%debug) then
00192       if (associated(meke%mom_src)) &
00193         call hchksum(meke%mom_src, 'MEKE mom_src', g%HI, scale=us%RZ3_T3_to_W_m2*us%L_to_Z**2)
00194       if (associated(meke%GME_snk)) &
00195         call hchksum(meke%GME_snk, 'MEKE GME_snk', g%HI, scale=us%RZ3_T3_to_W_m2*us%L_to_Z**2)
00196       if (associated(meke%GM_src)) &
00197         call hchksum(meke%GM_src, 'MEKE GM_src', g%HI, scale=us%RZ3_T3_to_W_m2*us%L_to_Z**2)
00198       if (associated(meke%MEKE)) call hchksum(meke%MEKE, 'MEKE MEKE', g%HI, scale=us%L_T_to_m_s**2)
00199       call uvchksum("MEKE SN_[uv]", sn_u, sn_v, g%HI, scale=us%s_to_T, &
00200                     scalar_pair=.true.)
00201       call uvchksum("MEKE h[uv]", hu, hv, g%HI, haloshift=1, &
00202                     scale=gv%H_to_m*(us%L_to_m**2))
00203     endif
00204
00205     sdt = dt*cs%MEKE_dtScale ! Scaled dt to use for time-stepping
00206     rho0 = gv%Rho0
00207     mass_neglect = gv%H_to_RZ * gv%H_subroundoff
00208     cdrag2 = cs%cdrag**2
00209
00210     ! With a depth-dependent (and possibly strong) damping, it seems
00211     ! advisable to use Strang splitting between the damping and diffusion.
00212     sdt_damp = sdt ; if (cs%MEKE_KH >= 0.0 .or. cs%MEKE_K4 >= 0.) sdt_damp = 0.5*sdt
00213
00214     ! Calculate depth integrated mass exchange if doing advection [R Z L2 ~> kg]
00215     if (cs%MEKE_advection_factor>0.) then
00216       do j=js,je ; do i=is-1,ie
00217         barohu(i,j) = 0.
00218       enddo ; enddo
00219       do k=1,nz
00220         do j=js,je ; do i=is-1,ie
00221           barohu(i,j) = hu(i,j,k) * gv%H_to_RZ
00222         enddo ; enddo
00223       enddo
00224       do j=js-1,je ; do i=is,ie
00225         barohv(i,j) = 0.
00226       enddo ; enddo
00227       do k=1,nz
00228         do j=js-1,je ; do i=is,ie
00229           barohv(i,j) = hv(i,j,k) * gv%H_to_RZ
00230         enddo ; enddo
00231       enddo
00232     endif
00233
00234     if (cs%MEKE_Cd_scale == 0.0 .and. .not. cs%visc_drag) then
00235       !$OMP parallel do default(shared) private(ldamping)
00236       do j=js,je ; do i=is,ie
00237         drag_rate(i,j) = 0. ; drag_rate_j15(i,j) = 0.
00238       enddo ; enddo
00239     endif
00240
00241     ! Calculate drag_rate_visc(i,j) which accounts for the model bottom mean flow
00242     if (cs%visc_drag) then
00243       !$OMP parallel do default(shared)
00244       do j=js,je ; do i=is-1,ie
00245         drag_vel_u(i,j) = 0.0
00246         if ((g%mask2dCu(i,j) > 0.0) .and. (visc%bbl_thick_u(i,j) > 0.0)) &
```

```
00247           drag_vel_u(i,j) = visc%Kv_bbl_u(i,j) / visc%bbl_thick_u(i,j)
00248       enddo ; enddo
00249       !$OMP parallel do default(shared)
00250       do j=js-1,je ; do i=is,ie
00251         drag_vel_v(i,j) = 0.0
00252         if ((g%mask2dCv(i,j) > 0.0) .and. (visc%bbl_thick_v(i,j) > 0.0)) &
00253           drag_vel_v(i,j) = visc%Kv_bbl_v(i,j) / visc%bbl_thick_v(i,j)
00254       enddo ; enddo
00255
00256       !$OMP parallel do default(shared)
00257       do j=js,je ; do i=is,ie
00258         drag_rate_visc(i,j) = (0.25*g%IareaT(i,j) * us%Z_to_L * &
00259                 ((g%areaCu(i-1,j)*drag_vel_u(i-1,j) + &
00260                   g%areaCu(i,j)*drag_vel_u(i,j)) + &
00261                  (g%areaCv(i,j-1)*drag_vel_v(i,j-1) + &
00262                   g%areaCv(i,j)*drag_vel_v(i,j)) ) )
00263       enddo ; enddo
00264     else
00265       !$OMP parallel do default(shared)
00266       do j=js,je ; do i=is,ie
00267         drag_rate_visc(i,j) = 0.
00268       enddo ; enddo
00269     endif
00270
00271     !$OMP parallel do default(shared)
00272     do j=js-1,je+1
00273       do i=is-1,ie+1 ; mass(i,j) = 0.0 ; enddo
00274       do k=1,nz ; do i=is-1,ie+1
00275         mass(i,j) = mass(i,j) + g%mask2dT(i,j) * (gv%H_to_RZ * h(i,j,k)) ! [R Z ~> kg m-2]
00276       enddo ; enddo
00277       do i=is-1,ie+1
00278         i_mass(i,j) = 0.0
00279         if (mass(i,j) > 0.0) i_mass(i,j) = 1.0 / mass(i,j) ! [R-1 Z-1 ~> m2 kg-1]
00280       enddo
00281     enddo
00282
00283     if (cs%initialize) then
00284       call meke_equilibrium(cs, meke, g, gv, us, sn_u, sn_v, drag_rate_visc, i_mass)
00285       cs%initialize = .false.
00286     endif
00287
00288     ! Calculates bottomFac2, barotrFac2 and LmixScale
00289     call meke_lengthscales(cs, meke, g, gv, us, sn_u, sn_v, meke%MEKE, bottomfac2,
    barotrfac2, lmixscale)
00290     if (cs%debug) then
00291       if (cs%visc_drag) &
00292         call uvchksum("MEKE drag_vel_[uv]", drag_vel_u, drag_vel_v, g%HI, &
00293                       scale=us%Z_to_m*us%s_to_T, scalar_pair=.true.)
00294       call hchksum(mass, 'MEKE mass',g%HI,haloshift=1, scale=us%RZ_to_kg_m2)
00295       call hchksum(drag_rate_visc, 'MEKE drag_rate_visc', g%HI, scale=us%L_T_to_m_s)
00296       call hchksum(bottomfac2, 'MEKE bottomFac2', g%HI)
00297       call hchksum(barotrfac2, 'MEKE barotrFac2', g%HI)
00298       call hchksum(lmixscale, 'MEKE LmixScale', g%HI,scale=us%L_to_m)
00299     endif
00300
00301     ! Aggregate sources of MEKE (background, frictional and GM)
00302     !$OMP parallel do default(shared)
00303     do j=js,je ; do i=is,ie
00304       src(i,j) = cs%MEKE_BGsrc
00305     enddo ; enddo
00306
00307     if (associated(meke%mom_src)) then
00308       !$OMP parallel do default(shared)
00309       do j=js,je ; do i=is,ie
00310         src(i,j) = src(i,j) - cs%MEKE_FrCoeff*i_mass(i,j)*meke%mom_src(i,j)
00311       enddo ; enddo
00312     endif
00313
00314     if (associated(meke%GME_snk)) then
00315       !$OMP parallel do default(shared)
00316       do j=js,je ; do i=is,ie
00317         src(i,j) = src(i,j) - cs%MEKE_GMECoeff*i_mass(i,j)*meke%GME_snk(i,j)
00318       enddo ; enddo
00319     endif
00320
00321     if (associated(meke%GM_src)) then
00322       if (cs%GM_src_alt) then
00323         !$OMP parallel do default(shared)
00324         do j=js,je ; do i=is,ie
00325           src(i,j) = src(i,j) - cs%MEKE_GMcoeff*meke%GM_src(i,j) / &
00326                       (gv%Rho0 * max(1.0*us%m_to_Z, g%bathyT(i,j)))
00327         enddo ; enddo
00328       else
00329         !$OMP parallel do default(shared)
00330         do j=js,je ; do i=is,ie
00331           src(i,j) = src(i,j) - cs%MEKE_GMcoeff*i_mass(i,j)*meke%GM_src(i,j)
00332         enddo ; enddo
```

```
00333       endif
00334     endif
00335
00336     if (cs%MEKE_equilibrium_restoring) then
00337       call meke_equilibrium_restoring(cs, g, us, sn_u, sn_v)
00338       do j=js,je ; do i=is,ie
00339         src(i,j) = src(i,j) - cs%MEKE_restoring_rate*(meke%MEKE(i,j) - cs%equilibrium_value(i,j))
00340       enddo ; enddo
00341     endif
00342
00343     ! Increase EKE by a full time-steps worth of source
00344     !$OMP parallel do default(shared)
00345     do j=js,je ; do i=is,ie
00346       meke%MEKE(i,j) = (meke%MEKE(i,j) + sdt*src(i,j))*g%mask2dT(i,j)
00347     enddo ; enddo
00348
00349     if (use_drag_rate) then
00350       ! Calculate a viscous drag rate (includes BBL contributions from mean flow and eddies)
00351       !$OMP parallel do default(shared)
00352       do j=js,je ; do i=is,ie
00353         drag_rate(i,j) = (us%L_to_Z*rho0 * i_mass(i,j)) * sqrt( drag_rate_visc(i,j)**2 + &
00354                  cdrag2 * ( max(0.0, 2.0*bottomfac2(i,j)*meke%MEKE(i,j)) + cs%MEKE_Uscale**2 ) )
00355       enddo ; enddo
00356     endif
00357
00358     ! First stage of Strang splitting
00359     !$OMP parallel do default(shared)
00360     do j=js,je ; do i=is,ie
00361       ldamping = cs%MEKE_damping + drag_rate(i,j) * bottomfac2(i,j)
00362       if (meke%MEKE(i,j) < 0.) ldamping = 0.
00363       ! notice that the above line ensures a damping only if MEKE is positive,
00364       ! while leaving MEKE unchanged if it is negative
00365       meke%MEKE(i,j) =  meke%MEKE(i,j) / (1.0 + sdt_damp*ldamping)
00366       meke_decay(i,j) = ldamping*g%mask2dT(i,j)
00367     enddo ; enddo
00368
00369     if (cs%kh_flux_enabled .or. cs%MEKE_K4 >= 0.0) then
00370       ! Update MEKE in the halos for lateral or bi-harmonic diffusion
00371       call cpu_clock_begin(cs%id_clock_pass)
00372       call do_group_pass(cs%pass_MEKE, g%Domain)
00373       call cpu_clock_end(cs%id_clock_pass)
00374     endif
00375
00376     if (cs%MEKE_K4 >= 0.0) then
00377       ! Calculate Laplacian of MEKE using MEKE_uflux and MEKE_vflux as temporary work space.
00378       !$OMP parallel do default(shared)
00379       do j=js-1,je+1 ; do i=is-2,ie+1
00380         ! MEKE_uflux is used here as workspace with units of [L2 T-2 ~> m2 s-2].
00381         meke_uflux(i,j) = ((g%dy_Cu(i,j)*g%IdxCu(i,j)) * g%mask2dCu(i,j)) * &
00382             (meke%MEKE(i+1,j) - meke%MEKE(i,j))
00383       ! This would have units of [R Z L2 T-2 ~> kg s-2]
00384       ! MEKE_uflux(I,j) = ((G%dy_Cu(I,j)*G%IdxCu(I,j)) * &
00385       !     ((2.0*mass(i,j)*mass(i+1,j)) / ((mass(i,j)+mass(i+1,j)) + mass_neglect)) ) * &
00386       !     (MEKE%MEKE(i+1,j) - MEKE%MEKE(i,j))
00387       enddo ; enddo
00388       !$OMP parallel do default(shared)
00389       do j=js-2,je+1 ; do i=is-1,ie+1
00390         ! MEKE_vflux is used here as workspace with units of [L2 T-2 ~> m2 s-2].
00391         meke_vflux(i,j) = ((g%dx_Cv(i,j)*g%IdyCv(i,j)) * g%mask2dCv(i,j)) * &
00392             (meke%MEKE(i,j+1) - meke%MEKE(i,j))
00393       ! This would have units of [R Z L2 T-2 ~> kg s-2]
00394       ! MEKE_vflux(i,J) = ((G%dx_Cv(i,J)*G%IdyCv(i,J)) * &
00395       !     ((2.0*mass(i,j)*mass(i,j+1)) / ((mass(i,j)+mass(i,j+1)) + mass_neglect)) ) * &
00396       !     (MEKE%MEKE(i,j+1) - MEKE%MEKE(i,j))
00397       enddo ; enddo
00398
00399       !$OMP parallel do default(shared)
00400       do j=js-1,je+1 ; do i=is-1,ie+1 ! del2MEKE has units [T-2 ~> s-2].
00401         del2meke(i,j) = g%IareaT(i,j) * &
00402             ((meke_uflux(i,j) - meke_uflux(i-1,j)) + (meke_vflux(i,j) - meke_vflux(i,j-1)))
00403       enddo ; enddo
00404
00405       ! Bi-harmonic diffusion of MEKE
00406       !$OMP parallel do default(shared) private(K4_here,Inv_K4_max)
00407       do j=js,je ; do i=is-1,ie
00408         k4_here = cs%MEKE_K4 ! [L4 T-1 ~> m4 s-1]
00409         ! Limit Kh to avoid CFL violations.
00410         inv_k4_max = 64.0 * sdt * ((g%dy_Cu(i,j)*g%IdxCu(i,j)) * &
00411                      max(g%IareaT(i,j), g%IareaT(i+1,j)))**2
00412         if (k4_here*inv_k4_max > 0.3) k4_here = 0.3 / inv_k4_max
00413
00414         ! Here the units of MEKE_uflux are [R Z L4 T-3 ~> kg m2 s-3].
00415         meke_uflux(i,j) = ((k4_here * (g%dy_Cu(i,j)*g%IdxCu(i,j))) * &
00416             ((2.0*mass(i,j)*mass(i+1,j)) / ((mass(i,j)+mass(i+1,j)) + mass_neglect)) ) * &
00417             (del2meke(i+1,j) - del2meke(i,j))
00418       enddo ; enddo
00419       !$OMP parallel do default(shared) private(K4_here,Inv_K4_max)
```

```
00420       do j=js-1,je ; do i=is,ie
00421         k4_here = cs%MEKE_K4 ! [L4 T-1 ~> m4 s-1]
00422         inv_k4_max = 64.0 * sdt * ((g%dx_Cv(i,j)*g%IdyCv(i,j)) * max(g%IareaT(i,j), g%IareaT(i,j+1)))**2
00423         if (k4_here*inv_k4_max > 0.3) k4_here = 0.3 / inv_k4_max
00424
00425         ! Here the units of MEKE_vflux are [R Z L4 T-3 ~> kg m2 s-3].
00426         meke_vflux(i,j) = ((k4_here * (g%dx_Cv(i,j)*g%IdyCv(i,j))) * &
00427             ((2.0*mass(i,j)*mass(i,j+1)) / ((mass(i,j)+mass(i,j+1)) + mass_neglect)) ) * &
00428             (del2meke(i,j+1) - del2meke(i,j))
00429       enddo ; enddo
00430       ! Store change in MEKE arising from the bi-harmonic in del4MEKE [L2 T-2 ~> m2 s-2].
00431       !$OMP parallel do default(shared)
00432       do j=js,je ; do i=is,ie
00433         del4meke(i,j) = (sdt*(g%IareaT(i,j)*i_mass(i,j))) * &
00434             ((meke_uflux(i-1,j) - meke_uflux(i,j)) + &
00435              (meke_vflux(i,j-1) - meke_vflux(i,j)))
00436       enddo ; enddo
00437     endif !
00438
00439     if (cs%kh_flux_enabled) then
00440       ! Lateral diffusion of MEKE
00441       kh_here = max(0., cs%MEKE_Kh)
00442       !$OMP parallel do default(shared) firstprivate(Kh_here) private(Inv_Kh_max)
00443       do j=js,je ; do i=is-1,ie
00444         ! Limit Kh to avoid CFL violations.
00445         if (associated(meke%Kh)) &
00446           kh_here = max(0., cs%MEKE_Kh) + &
00447               cs%KhMEKE_Fac*0.5*(meke%Kh(i,j)+meke%Kh(i+1,j))
00448         if (associated(meke%Kh_diff)) &
00449           kh_here = max(0.,cs%MEKE_Kh) + &
00450               cs%KhMEKE_Fac*0.5*(meke%Kh_diff(i,j)+meke%Kh_diff(i+1,j))
00451         inv_kh_max = 2.0*sdt * ((g%dy_Cu(i,j)*g%IdxCu(i,j)) * &
00452                     max(g%IareaT(i,j),g%IareaT(i+1,j)))
00453         if (kh_here*inv_kh_max > 0.25) kh_here = 0.25 / inv_kh_max
00454         kh_u(i,j) = kh_here
00455
00456         ! Here the units of MEKE_uflux and MEKE_vflux are [R Z L4 T-3 ~> kg m2 s-3].
00457         meke_uflux(i,j) = ((kh_here * (g%dy_Cu(i,j)*g%IdxCu(i,j))) * &
00458             ((2.0*mass(i,j)*mass(i+1,j)) / ((mass(i,j)+mass(i+1,j)) + mass_neglect)) ) * &
00459             (meke%MEKE(i,j) - meke%MEKE(i+1,j))
00460       enddo ; enddo
00461       !$OMP parallel do default(shared) firstprivate(Kh_here) private(Inv_Kh_max)
00462       do j=js-1,je ; do i=is,ie
00463         if (associated(meke%Kh)) &
00464           kh_here = max(0.,cs%MEKE_Kh) + cs%KhMEKE_Fac * 0.5*(meke%Kh(i,j)+meke%Kh(i,j+1))
00465         if (associated(meke%Kh_diff)) &
00466           kh_here = max(0.,cs%MEKE_Kh) + cs%KhMEKE_Fac * 0.5*(meke%Kh_diff(i,j)+meke%Kh_diff(i,j+1))
00467         inv_kh_max = 2.0*sdt * ((g%dx_Cv(i,j)*g%IdyCv(i,j)) * max(g%IareaT(i,j),g%IareaT(i,j+1)))
00468         if (kh_here*inv_kh_max > 0.25) kh_here = 0.25 / inv_kh_max
00469         kh_v(i,j) = kh_here
00470
00471         ! Here the units of MEKE_uflux and MEKE_vflux are [R Z L4 T-3 ~> kg m2 s-3].
00472         meke_vflux(i,j) = ((kh_here * (g%dx_Cv(i,j)*g%IdyCv(i,j))) * &
00473             ((2.0*mass(i,j)*mass(i,j+1)) / ((mass(i,j)+mass(i,j+1)) + mass_neglect)) ) * &
00474             (meke%MEKE(i,j) - meke%MEKE(i,j+1))
00475       enddo ; enddo
00476       if (cs%MEKE_advection_factor>0.) then
00477         advfac = cs%MEKE_advection_factor / sdt ! [T-1 ~> s-1]
00478         !$OMP parallel do default(shared)
00479         do j=js,je ; do i=is-1,ie
00480           ! Here the units of the quantities added to MEKE_uflux are [R Z L4 T-3 ~> kg m2 s-3].
00481           if (barohu(i,j)>0.) then
00482             meke_uflux(i,j) = meke_uflux(i,j) + barohu(i,j)*meke%MEKE(i,j)*advfac
00483           elseif (barohu(i,j)<0.) then
00484             meke_uflux(i,j) = meke_uflux(i,j) + barohu(i,j)*meke%MEKE(i+1,j)*advfac
00485           endif
00486         enddo ; enddo
00487         !$OMP parallel do default(shared)
00488         do j=js-1,je ; do i=is,ie
00489           ! Here the units of the quantities added to MEKE_vflux are [R Z L4 T-3 ~> kg m2 s-3].
00490           if (barohv(i,j)>0.) then
00491             meke_vflux(i,j) = meke_vflux(i,j) + barohv(i,j)*meke%MEKE(i,j)*advfac
00492           elseif (barohv(i,j)<0.) then
00493             meke_vflux(i,j) = meke_vflux(i,j) + barohv(i,j)*meke%MEKE(i,j+1)*advfac
00494           endif
00495         enddo ; enddo
00496       endif
00497
00498       !$OMP parallel do default(shared)
00499       do j=js,je ; do i=is,ie
00500         meke%MEKE(i,j) = meke%MEKE(i,j) + (sdt*(g%IareaT(i,j)*i_mass(i,j))) * &
00501             ((meke_uflux(i-1,j) - meke_uflux(i,j)) + &
00502              (meke_vflux(i,j-1) - meke_vflux(i,j)))
00503       enddo ; enddo
00504     endif ! MEKE_KH>0
00505
00506     ! Add on bi-harmonic tendency
```

```
00507     if (cs%MEKE_K4 >= 0.0) then
00508       !$OMP parallel do default(shared)
00509       do j=js,je ; do i=is,ie
00510         meke%MEKE(i,j) = meke%MEKE(i,j) + del4meke(i,j)
00511       enddo ; enddo
00512     endif
00513
00514     ! Second stage of Strang splitting
00515     if (cs%MEKE_KH >= 0.0 .or. cs%MEKE_K4 >= 0.0) then
00516       if (sdt>sdt_damp) then
00517         ! Recalculate the drag rate, since MEKE has changed.
00518         if (use_drag_rate) then
00519           !$OMP parallel do default(shared)
00520           do j=js,je ; do i=is,ie
00521             drag_rate(i,j) = (us%L_to_Z*rho0 * i_mass(i,j)) * sqrt( drag_rate_visc(i,j)**2 + &
00522                    cdrag2 * ( max(0.0, 2.0*bottomfac2(i,j)*meke%MEKE(i,j)) + cs%MEKE_Uscale**2 ) )
00523           enddo ; enddo
00524           !$OMP parallel do default(shared)
00525           do j=js,je ; do i=is,ie
00526             ldamping = cs%MEKE_damping + drag_rate(i,j) * bottomfac2(i,j)
00527             if (meke%MEKE(i,j) < 0.) ldamping = 0.
00528             ! notice that the above line ensures a damping only if MEKE is positive,
00529             ! while leaving MEKE unchanged if it is negative
00530             meke%MEKE(i,j) =  meke%MEKE(i,j) / (1.0 + sdt_damp*ldamping)
00531             meke_decay(i,j) = ldamping*g%mask2dT(i,j)
00532           enddo ; enddo
00533         endif
00534       endif
00535     endif ! MEKE_KH>=0
00536
00537 !   do j=js,je ; do i=is,ie
00538 !     MEKE%MEKE(i,j) =  MAX(MEKE%MEKE(i,j),0.0)
00539 !   enddo ; enddo
00540
00541     call cpu_clock_begin(cs%id_clock_pass)
00542     call do_group_pass(cs%pass_MEKE, g%Domain)
00543     call cpu_clock_end(cs%id_clock_pass)
00544
00545     ! Calculate diffusivity for main model to use
00546     if (cs%MEKE_KhCoeff>0.) then
00547       if (.not.cs%MEKE_GEOMETRIC) then
00548         if (cs%use_old_lscale) then
00549           if (cs%Rd_as_max_scale) then
00550             !$OMP parallel do default(shared)
00551             do j=js,je ; do i=is,ie
00552               meke%Kh(i,j) = (cs%MEKE_KhCoeff * &
00553                           sqrt(2.*max(0.,barotrfac2(i,j)*meke%MEKE(i,j))*g%areaT(i,j)) ) * &
00554                           min(meke%Rd_dx_h(i,j), 1.0)
00555             enddo ; enddo
00556           else
00557             !$OMP parallel do default(shared)
00558             do j=js,je ; do i=is,ie
00559               meke%Kh(i,j) = cs%MEKE_KhCoeff * &
00560                   sqrt(2.*max(0., barotrfac2(i,j)*meke%MEKE(i,j))*g%areaT(i,j))
00561             enddo ; enddo
00562           endif
00563         else
00564           !$OMP parallel do default(shared)
00565           do j=js,je ; do i=is,ie
00566             meke%Kh(i,j) = cs%MEKE_KhCoeff * &
00567                 sqrt(2.*max(0., barotrfac2(i,j)*meke%MEKE(i,j))) * lmixscale(i,j)
00568           enddo ; enddo
00569         endif
00570       endif
00571     endif
00572
00573     ! Calculate viscosity for the main model to use
00574     if (cs%viscosity_coeff_Ku /=0.) then
00575       do j=js,je ; do i=is,ie
00576         meke%Ku(i,j) = cs%viscosity_coeff_Ku * sqrt(2.*max(0.,meke%MEKE(i,j))) * lmixscale(i,j)
00577       enddo ; enddo
00578     endif
00579
00580     if (cs%viscosity_coeff_Au /=0.) then
00581       do j=js,je ; do i=is,ie
00582         meke%Au(i,j) = cs%viscosity_coeff_Au * sqrt(2.*max(0.,meke%MEKE(i,j))) * lmixscale(i,j)**3
00583       enddo ; enddo
00584     endif
00585
00586     if (associated(meke%Kh) .or. associated(meke%Ku) .or. associated(meke%Au)) then
00587       call cpu_clock_begin(cs%id_clock_pass)
00588       call do_group_pass(cs%pass_Kh, g%Domain)
00589       call cpu_clock_end(cs%id_clock_pass)
00590     endif
00591
00592     ! Offer fields for averaging.
00593     if (any([cs%id_Ue, cs%id_Ub, cs%id_Ut] > 0)) &
```

```
00594       tmp(:,:) = 0.
00595     if (cs%id_MEKE>0) call post_data(cs%id_MEKE, meke%MEKE, cs%diag)
00596     if (cs%id_Ue>0) then
00597       do j=js,je ; do i=is,ie
00598         tmp(i,j) = sqrt(max(0., 2. * meke%MEKE(i,j)))
00599       enddo ; enddo
00600       call post_data(cs%id_Ue, tmp, cs%diag)
00601     endif
00602     if (cs%id_Ub>0) then
00603       do j=js,je ; do i=is,ie
00604         tmp(i,j) = sqrt(max(0., 2. * meke%MEKE(i,j) * bottomfac2(i,j)))
00605       enddo ; enddo
00606       call post_data(cs%id_Ub, tmp, cs%diag)
00607     endif
00608     if (cs%id_Ut>0) then
00609       do j=js,je ; do i=is,ie
00610         tmp(i,j) = sqrt(max(0., 2. * meke%MEKE(i,j) * barotrfac2(i,j)))
00611       enddo ; enddo
00612       call post_data(cs%id_Ut, tmp, cs%diag)
00613     endif
00614     if (cs%id_Kh>0) call post_data(cs%id_Kh, meke%Kh, cs%diag)
00615     if (cs%id_Ku>0) call post_data(cs%id_Ku, meke%Ku, cs%diag)
00616     if (cs%id_Au>0) call post_data(cs%id_Au, meke%Au, cs%diag)
00617     if (cs%id_KhMEKE_u>0) call post_data(cs%id_KhMEKE_u, kh_u, cs%diag)
00618     if (cs%id_KhMEKE_v>0) call post_data(cs%id_KhMEKE_v, kh_v, cs%diag)
00619     if (cs%id_src>0) call post_data(cs%id_src, src, cs%diag)
00620     if (cs%id_decay>0) call post_data(cs%id_decay, meke_decay, cs%diag)
00621     if (cs%id_GM_src>0) call post_data(cs%id_GM_src, meke%GM_src, cs%diag)
00622     if (cs%id_mom_src>0) call post_data(cs%id_mom_src, meke%mom_src, cs%diag)
00623     if (cs%id_GME_snk>0) call post_data(cs%id_GME_snk, meke%GME_snk, cs%diag)
00624     if (cs%id_Le>0) call post_data(cs%id_Le, lmixscale, cs%diag)
00625     if (cs%id_gamma_b>0) then
00626       do j=js,je ; do i=is,ie
00627         bottomfac2(i,j) = sqrt(bottomfac2(i,j))
00628       enddo ; enddo
00629       call post_data(cs%id_gamma_b, bottomfac2, cs%diag)
00630     endif
00631     if (cs%id_gamma_t>0) then
00632       do j=js,je ; do i=is,ie
00633         barotrfac2(i,j) = sqrt(barotrfac2(i,j))
00634       enddo ; enddo
00635       call post_data(cs%id_gamma_t, barotrfac2, cs%diag)
00636     endif
00637
00638 end subroutine step_forward_meke
00639
00640 !> Calculates the equilibrium solutino where the source depends only on MEKE diffusivity
00641 !! and there is no lateral diffusion of MEKE.
00642 !! Results is in MEKE%MEKE.
00643 subroutine meke_equilibrium(CS, MEKE, G, GV, US, SN_u, SN_v, drag_rate_visc, I_mass)
00644   type(ocean_grid_type),              intent(inout) :: G    !< Ocean grid.
00645   type(verticalgrid_type),            intent(in)    :: GV   !< Ocean vertical grid structure.
00646   type(unit_scale_type),              intent(in)    :: US   !< A dimensional unit scaling
      type
00647   type(meke_cs),                      pointer       :: CS   !< MEKE control structure.
00648   type(meke_type),                    pointer       :: MEKE !< A structure with MEKE data.
00649   real, dimension(SZIB_(G),SZJ_(G)), intent(in)    :: SN_u !< Eady growth rate at u-points [T-1 ~> s-1].
00650   real, dimension(SZI_(G),SZJB_(G)), intent(in)    :: SN_v !< Eady growth rate at v-points [T-1 ~> s-1].
00651   real, dimension(SZI_(G),SZJ_(G)),  intent(in)    :: drag_rate_visc !< Mean flow velocity contribution
00652                                                                       !! to the MEKE drag rate [L T-1 ~> m s-1]
00653   real, dimension(SZI_(G),SZJ_(G)),  intent(in)    :: I_mass  !< Inverse of column mass [R-1 Z-1 ~> m2
      kg-1].
00654   ! Local variables
00655   real :: beta ! Combined topograpic and planetary vorticity gradient [T-1 L-1 ~> s-1 m-1]
00656   real :: SN   ! The local Eady growth rate [T-1 ~> s-1]
00657   real :: bottomFac2, barotrFac2    ! Vertical structure factors [nondim]
00658   real :: LmixScale, LRhines, LEady ! Various mixing length scales [L ~> m]
00659   real :: I_H, KhCoeff
00660   real :: Kh    ! A lateral diffusivity [L2 T-1 ~> m2 s-1]
00661   real :: Ubg2  ! Background (tidal?) velocity squared [L2 T-2 ~> m2 s-2]
00662   real :: cd2
00663   real :: drag_rate ! The MEKE spindown timescale due to bottom drag [T-1 ~> s-1].
00664   real :: src   ! The sum of MEKE sources [L2 T-3 ~> W kg-1]
00665   real :: ldamping  ! The MEKE damping rate [T-1 ~> s-1].
00666   real :: EKE, EKEmin, EKEmax, EKEerr ! [L2 T-2 ~> m2 s-2]
00667   real :: resid, ResMin, ResMax ! Residuals [L2 T-3 ~> W kg-1]
00668   real :: FatH    ! Coriolis parameter at h points; to compute topographic beta [T-1 ~> s-1]
00669   real :: beta_topo_x, beta_topo_y    ! Topographic PV gradients in x and y [T-1 L-1 ~> s-1 m-1]
00670   integer :: i, j, is, ie, js, je, n1, n2
00671   real :: tolerance ! Width of EKE bracket [L2 T-2 ~> m2 s-2].
00672   logical :: useSecant, debugIteration
00673
00674   real :: Lgrid, Ldeform, Lfrict
00675
00676   is = g%isc ; ie = g%iec ; js = g%jsc ; je = g%jec
00677
00678   debugiteration = .false.
```

```
00679   khcoeff = cs%MEKE_KhCoeff
00680   ubg2 = cs%MEKE_Uscale**2
00681   cd2 = cs%cdrag**2
00682   tolerance = 1.0e-12*us%m_s_to_L_T**2
00683
00684 !$OMP do
00685   do j=js,je ; do i=is,ie
00686     ! SN = 0.25*max( (SN_u(I,j) + SN_u(I-1,j)) + (SN_v(i,J) + SN_v(i,J-1)), 0.)
00687     ! This avoids extremes values in equilibrium solution due to bad values in SN_u, SN_v
00688     sn = min(sn_u(i,j), sn_u(i-1,j), sn_v(i,j), sn_v(i,j-1))
00689
00690     if (cs%MEKE_equilibrium_alt) then
00691       meke%MEKE(i,j) = (cs%MEKE_GEOMETRIC_alpha * sn * us%Z_to_m*g%bathyT(i,j))**2 / cd2
00692     else
00693       fath = 0.25*((g%CoriolisBu(i,j) + g%CoriolisBu(i-1,j-1)) + &
00694                    (g%CoriolisBu(i-1,j) + g%CoriolisBu(i,j-1))) ! Coriolis parameter at h points
00695
00696       ! Since zero-bathymetry cells are masked, this avoids calculations on land
00697       if (cs%MEKE_topographic_beta == 0. .or. g%bathyT(i,j) == 0.) then
00698         beta_topo_x = 0. ; beta_topo_y = 0.
00699       else
00700         !### Consider different combinations of these estimates of topographic beta, and the use
00701         !    of the water column thickness instead of the bathymetric depth.
00702         beta_topo_x = -cs%MEKE_topographic_beta * fath * 0.5 * ( &
00703                        (g%bathyT(i+1,j)-g%bathyT(i,j)) * g%IdxCu(i,j)  &
00704                   / max(g%bathyT(i+1,j),g%bathyT(i,j), gv%H_subroundoff) &
00705                 +       (g%bathyT(i,j)-g%bathyT(i-1,j)) * g%IdxCu(i-1,j) &
00706                   / max(g%bathyT(i,j),g%bathyT(i-1,j), gv%H_subroundoff) )
00707         beta_topo_y = -cs%MEKE_topographic_beta * fath * 0.5 * ( &
00708                        (g%bathyT(i,j+1)-g%bathyT(i,j)) * g%IdyCv(i,j)  &
00709                   / max(g%bathyT(i,j+1),g%bathyT(i,j), gv%H_subroundoff) + &
00710                        (g%bathyT(i,j)-g%bathyT(i,j-1)) * g%IdyCv(i,j-1) &
00711                   / max(g%bathyT(i,j),g%bathyT(i,j-1), gv%H_subroundoff) )
00712       endif
00713       beta =  sqrt((g%dF_dx(i,j) + beta_topo_x)**2 + &
00714                    (g%dF_dy(i,j) + beta_topo_y)**2 )
00715
00716       i_h = us%L_to_Z*gv%Rho0 * i_mass(i,j)
00717
00718       if (khcoeff*sn*i_h>0.) then
00719         ! Solve resid(E) = 0, where resid = Kh(E) * (SN)^2 - damp_rate(E) E
00720         ekemin = 0.   ! Use the trivial root as the left bracket
00721         resmin = 0.   ! Need to detect direction of left residual
00722         ekemax = 0.01*us%m_s_to_L_T**2 ! First guess at right bracket
00723         usesecant = .false. ! Start using a bisection method
00724
00725         ! First find right bracket for which resid<0
00726         resid = 1.0*us%m_to_L**2*us%T_to_s**3 ; n1 = 0
00727         do while (resid>0.)
00728           n1 = n1 + 1
00729           eke = ekemax
00730           call meke_lengthscales_0d(cs, us, g%areaT(i,j), beta, g%bathyT(i,j), &
00731                                     meke%Rd_dx_h(i,j), sn, eke, &
00732                                     bottomfac2, barotrfac2, lmixscale, lrhines, leady)
00733           ! TODO: Should include resolution function in Kh
00734           kh = (khcoeff * sqrt(2.*barotrfac2*eke) * lmixscale)
00735           src = kh * (sn * sn)
00736           drag_rate = i_h * sqrt(drag_rate_visc(i,j)**2 + cd2 * ( 2.0*bottomfac2*eke + ubg2 ) )
00737           ldamping = cs%MEKE_damping + drag_rate * bottomfac2
00738           resid = src - ldamping * eke
00739           ! if (debugIteration) then
00740           !   write(0,*) n1, 'EKE=',EKE,'resid=',resid
00741           !   write(0,*) 'EKEmin=',EKEmin,'ResMin=',ResMin
00742           !   write(0,*) 'src=',src,'ldamping=',ldamping
00743           !   write(0,*) 'gamma-b=',bottomFac2,'gamma-t=',barotrFac2
00744           !   write(0,*) 'drag_visc=',drag_rate_visc(i,j),'Ubg2=',Ubg2
00745           ! endif
00746           if (resid>0.) then    ! EKE is to the left of the root
00747             ekemin = eke        ! so we move the left bracket here
00748             ekemax = 10. * eke  ! and guess again for the right bracket
00749             if (resid<resmin) usesecant = .true.
00750             resmin = resid
00751             if (ekemax > 2.e17*us%m_s_to_L_T**2) then
00752               if (debugiteration) stop 'Something has gone very wrong'
00753               debugiteration = .true.
00754               resid = 1. ; n1 = 0
00755               ekemin = 0. ; resmin = 0.
00756               ekemax = 0.01*us%m_s_to_L_T**2
00757               usesecant = .false.
00758             endif
00759           endif
00760         enddo ! while(resid>0.) searching for right bracket
00761         resmax = resid
00762
00763         ! Bisect the bracket
00764         n2 = 0 ; ekeerr = ekemax - ekemin
00765         do while (ekeerr > tolerance)
```

```
00766           n2 = n2 + 1
00767           if (usesecant) then
00768             eke = ekemin + (ekemax - ekemin) * (resmin / (resmin - resmax))
00769           else
00770             eke = 0.5 * (ekemin + ekemax)
00771           endif
00772           ekeerr = min( eke-ekemin, ekemax-eke )
00773           ! TODO: Should include resolution function in Kh
00774           kh = (khcoeff * sqrt(2.*barotrfac2*eke) * lmixscale)
00775           src = kh * (sn * sn)
00776           drag_rate = i_h * sqrt( drag_rate_visc(i,j)**2 + cd2 * ( 2.0*bottomfac2*eke + ubg2 ) )
00777           ldamping = cs%MEKE_damping + drag_rate * bottomfac2
00778           resid = src - ldamping * eke
00779           if (usesecant .and. resid>resmin) usesecant = .false.
00780           if (resid>0.) then              ! EKE is to the left of the root
00781             ekemin = eke                  ! so we move the left bracket here
00782             if (resid<resmin) usesecant = .true.
00783             resmin = resid                ! Save this for the secant method
00784           elseif (resid<0.) then          ! EKE is to the right of the root
00785             ekemax = eke                  ! so we move the right bracket here
00786             resmax = resid                ! Save this for the secant method
00787           else
00788             exit                          ! resid=0 => EKE is exactly at the root
00789           endif
00790           if (n2>200) stop 'Failing to converge?'
00791         enddo ! while(EKEmax-EKEmin>tolerance)
00792
00793       else
00794         eke = 0.
00795       endif
00796       meke%MEKE(i,j) = eke
00797     endif
00798   enddo ; enddo
00799
00800 end subroutine meke_equilibrium
00801
00802
00803 !< This subroutine calculates a new equilibrium value for MEKE at each time step. This is not copied into
00804 !! MEKE%MEKE; rather, it is used as a restoring term to nudge MEKE%MEKE back to an equilibrium value
00805 subroutine meke_equilibrium_restoring(CS, G, US, SN_u, SN_v)
00806   type(ocean_grid_type),              intent(inout) :: G    !< Ocean grid.
00807   type(unit_scale_type),              intent(in)    :: US   !< A dimensional unit scaling
      type.
00808   type(meke_cs),                      pointer       :: CS   !< MEKE control structure.
00809   real, dimension(SZIB_(G),SZJ_(G)), intent(in)    :: SN_u !< Eady growth rate at u-points [T-1 ~> s-1].
00810   real, dimension(SZI_(G),SZJB_(G)), intent(in)    :: SN_v !< Eady growth rate at v-points [T-1 ~> s-1].
00811   ! Local variables
00812   real :: SN                      ! The local Eady growth rate [T-1 ~> s-1]
00813   integer :: i, j, is, ie, js, je ! local indices
00814   real :: cd2                     ! bottom drag
00815
00816   is = g%isc ; ie = g%iec ; js = g%jsc ; je = g%jec
00817   cd2 = cs%cdrag**2
00818
00819   if (.not. associated(cs%equilibrium_value)) allocate(cs%equilibrium_value(szi_(g),szj_(g)))
00820   cs%equilibrium_value(:,:) = 0.0
00821
00822 !$OMP do
00823   do j=js,je ; do i=is,ie
00824     ! SN = 0.25*max( (SN_u(I,j) + SN_u(I-1,j)) + (SN_v(i,J) + SN_v(i,J-1)), 0.)
00825     ! This avoids extremes values in equilibrium solution due to bad values in SN_u, SN_v
00826     sn = min(sn_u(i,j), sn_u(i-1,j), sn_v(i,j), sn_v(i,j-1))
00827     cs%equilibrium_value(i,j) = (cs%MEKE_GEOMETRIC_alpha * sn * us%Z_to_m*g%bathyT(i,j))**2 / cd2
00828   enddo ; enddo
00829
00830   if (cs%id_MEKE_equilibrium>0) call post_data(cs%id_MEKE_equilibrium, cs%equilibrium_value, cs%diag)
00831
00832 end subroutine meke_equilibrium_restoring
00833
00834 !> Calculates the eddy mixing length scale and \f$\gamma_b\f$ and \f$\gamma_t\f$
00835 !! functions that are ratios of either bottom or barotropic eddy energy to the
00836 !! column eddy energy, respectively.  See \ref section_MEKE_equations.
00837 subroutine meke_lengthscales(CS, MEKE, G, GV, US, SN_u, SN_v, &
00838             EKE, bottomFac2, barotrFac2, LmixScale)
00839   type(meke_cs),                      pointer       :: CS   !< MEKE control structure.
00840   type(meke_type),                    pointer       :: MEKE !< MEKE data.
00841   type(ocean_grid_type),              intent(inout) :: G    !< Ocean grid.
00842   type(verticalgrid_type),            intent(in)    :: GV   !< Ocean vertical grid structure.
00843   type(unit_scale_type),              intent(in)    :: US   !< A dimensional unit scaling
      type
00844   real, dimension(SZIB_(G),SZJ_(G)), intent(in)    :: SN_u !< Eady growth rate at u-points [T-1 ~> s-1].
00845   real, dimension(SZI_(G),SZJB_(G)), intent(in)    :: SN_v !< Eady growth rate at v-points [T-1 ~> s-1].
00846   real, dimension(SZI_(G),SZJ_(G)),  intent(in)    :: EKE  !< Eddy kinetic energy [L2 T-2 ~> m2 s-2].
00847   real, dimension(SZI_(G),SZJ_(G)),  intent(out)   :: bottomFac2 !< gamma_b^2
00848   real, dimension(SZI_(G),SZJ_(G)),  intent(out)   :: barotrFac2 !< gamma_t^2
00849   real, dimension(SZI_(G),SZJ_(G)),  intent(out)   :: LmixScale !< Eddy mixing length [L ~> m].
00850   ! Local variables
```

```
00851   real, dimension(SZI_(G),SZJ_(G)) :: LRhines, LEady  ! Possible mixing length scales [L ~> m]
00852   real :: beta ! Combined topograpic and planetary vorticity gradient [T-1 L-1 ~> s-1 m-1]
00853   real :: SN   ! The local Eady growth rate [T-1 ~> s-1]
00854   real :: FatH ! Coriolis parameter at h points [T-1 ~> s-1]
00855   real :: beta_topo_x, beta_topo_y  ! Topographic PV gradients in x and y [T-1 L-1 ~> s-1 m-1]
00856   integer :: i, j, is, ie, js, je
00857
00858   is = g%isc ; ie = g%iec ; js = g%jsc ; je = g%jec
00859
00860 !$OMP do
00861   do j=js,je ; do i=is,ie
00862     if (.not.cs%use_old_lscale) then
00863       if (cs%aEady > 0.) then
00864         sn = 0.25 * ( (sn_u(i,j) + sn_u(i-1,j)) + (sn_v(i,j) + sn_v(i,j-1)) )
00865       else
00866         sn = 0.
00867       endif
00868       fath = 0.25* ( ( g%CoriolisBu(i,j) + g%CoriolisBu(i-1,j-1) ) + &
00869                      ( g%CoriolisBu(i-1,j) + g%CoriolisBu(i,j-1) ) )  ! Coriolis parameter at h points
00870
00871       ! If bathyT is zero, then a division by zero FPE will be raised.  In this
00872       ! case, we apply Adcroft's rule of reciprocals and set the term to zero.
00873       ! Since zero-bathymetry cells are masked, this should not affect values.
00874       if (cs%MEKE_topographic_beta == 0. .or. g%bathyT(i,j) == 0.0) then
00875         beta_topo_x = 0. ; beta_topo_y = 0.
00876       else
00877         !### Consider different combinations of these estimates of topographic beta, and the use
00878         !    of the water column thickness instead of the bathymetric depth.
00879         beta_topo_x = -cs%MEKE_topographic_beta * fath * 0.5 * ( &
00880                       (g%bathyT(i+1,j)-g%bathyT(i,j)) * g%IdxCu(i,j)  &
00881                  / max(g%bathyT(i+1,j),g%bathyT(i,j), gv%H_subroundoff) &
00882                +       (g%bathyT(i,j)-g%bathyT(i-1,j)) * g%IdxCu(i-1,j) &
00883                  / max(g%bathyT(i,j),g%bathyT(i-1,j), gv%H_subroundoff) )
00884         beta_topo_y = -cs%MEKE_topographic_beta * fath * 0.5 * ( &
00885                       (g%bathyT(i,j+1)-g%bathyT(i,j)) * g%IdyCv(i,j)  &
00886                  / max(g%bathyT(i,j+1),g%bathyT(i,j), gv%H_subroundoff) + &
00887                       (g%bathyT(i,j)-g%bathyT(i,j-1)) * g%IdyCv(i,j-1) &
00888                  / max(g%bathyT(i,j),g%bathyT(i,j-1), gv%H_subroundoff) )
00889       endif
00890       beta =  sqrt((g%dF_dx(i,j) + beta_topo_x)**2 + &
00891                    (g%dF_dy(i,j) + beta_topo_y)**2 )
00892
00893     else
00894       beta = 0.
00895     endif
00896     ! Returns bottomFac2, barotrFac2 and LmixScale
00897     call meke_lengthscales_0d(cs, us, g%areaT(i,j), beta, g%bathyT(i,j),  &
00898                               meke%Rd_dx_h(i,j), sn, meke%MEKE(i,j), &
00899                               bottomfac2(i,j), barotrfac2(i,j), lmixscale(i,j), &
00900                               lrhines(i,j), leady(i,j))
00901   enddo ; enddo
00902   if (cs%id_Lrhines>0) call post_data(cs%id_LRhines, lrhines, cs%diag)
00903   if (cs%id_Leady>0) call post_data(cs%id_LEady, leady, cs%diag)
00904
00905 end subroutine meke_lengthscales
00906
00907 !> Calculates the eddy mixing length scale and \f$\gamma_b\f$ and \f$\gamma_t\f$
00908 !! functions that are ratios of either bottom or barotropic eddy energy to the
00909 !! column eddy energy, respectively.  See \ref section_MEKE_equations.
00910 subroutine meke_lengthscales_0d(CS, US, area, beta, depth, Rd_dx, SN, EKE, & ! Z_to_L,
      &
00911                                 bottomFac2, barotrFac2, LmixScale, Lrhines, Leady)
00912   type(meke_cs), pointer       :: CS         !< MEKE control structure.
00913   type(unit_scale_type), intent(in) :: US    !< A dimensional unit scaling type
00914   real,          intent(in)    :: area       !< Grid cell area [L2 ~> m2]
00915   real,          intent(in)    :: beta       !< Planetary beta = |grad F| [T-1 L-1 ~> s-1 m-1]
00916   real,          intent(in)    :: depth      !< Ocean depth [Z ~> m]
00917   real,          intent(in)    :: Rd_dx      !< Resolution Ld/dx [nondim].
00918   real,          intent(in)    :: SN         !< Eady growth rate [T-1 ~> s-1].
00919   real,          intent(in)    :: EKE        !< Eddy kinetic energy [L2 T-2 ~> m2 s-2].
00920 !  real,          intent(in)    :: Z_to_L     !< A conversion factor from depth units (Z) to
00921 !                                             !! the units for lateral distances (L).
00922   real,          intent(out)   :: bottomFac2 !< gamma_b^2
00923   real,          intent(out)   :: barotrFac2 !< gamma_t^2
00924   real,          intent(out)   :: LmixScale  !< Eddy mixing length [L ~> m].
00925   real,          intent(out)   :: Lrhines    !< Rhines length scale [L ~> m].
00926   real,          intent(out)   :: Leady      !< Eady length scale [L ~> m].
00927   ! Local variables
00928   real :: Lgrid, Ldeform, Lfrict ! Length scales [L ~> m]
00929   real :: Ue  ! An eddy velocity [L T-1 ~> m s-1]
00930
00931   ! Length scale for MEKE derived diffusivity
00932   lgrid = sqrt(area)                ! Grid scale
00933   ldeform = lgrid * rd_dx           ! Deformation scale
00934   lfrict = (us%Z_to_L * depth) / cs%cdrag  ! Frictional arrest scale
00935   ! gamma_b^2 is the ratio of bottom eddy energy to mean column eddy energy
00936   ! used in calculating bottom drag
```

```
00937   bottomfac2 = cs%MEKE_CD_SCALE**2
00938   if (lfrict*cs%MEKE_Cb>0.) bottomfac2 = bottomfac2 + 1./( 1. + cs%MEKE_Cb*(ldeform/lfrict) )**0.8
00939   bottomfac2 = max(bottomfac2, cs%MEKE_min_gamma)
00940   ! gamma_t^2 is the ratio of barotropic eddy energy to mean column eddy energy
00941   ! used in the velocity scale for diffusivity
00942   barotrfac2 = 1.
00943   if (lfrict*cs%MEKE_Ct>0.) barotrfac2 = 1. / ( 1. + cs%MEKE_Ct*(ldeform/lfrict) )**0.25
00944   barotrfac2 = max(barotrfac2, cs%MEKE_min_gamma)
00945   if (cs%use_old_lscale) then
00946     if (cs%Rd_as_max_scale) then
00947       lmixscale = min(ldeform, lgrid) ! The smaller of Ld or dx
00948     else
00949       lmixscale = lgrid
00950     endif
00951   else
00952     ue = sqrt( 2.0 * max( 0., barotrfac2*eke ) ) ! Barotropic eddy flow scale
00953     lrhines = sqrt( ue / max( beta, 1.e-30*us%T_to_s*us%L_to_m ) )       ! Rhines scale
00954     if (cs%aEady > 0.) then
00955       leady = ue / max( sn, 1.e-15*us%T_to_s ) ! Bound Eady time-scale < 1e15 seconds
00956     else
00957       leady = 0.
00958     endif
00959     if (cs%use_min_lscale) then
00960       lmixscale = 1.e7
00961       if (cs%aDeform*ldeform > 0.) lmixscale = min(lmixscale,cs%aDeform*ldeform)
00962       if (cs%aFrict *lfrict  > 0.) lmixscale = min(lmixscale,cs%aFrict *lfrict)
00963       if (cs%aRhines*lrhines > 0.) lmixscale = min(lmixscale,cs%aRhines*lrhines)
00964       if (cs%aEady  *leady   > 0.) lmixscale = min(lmixscale,cs%aEady  *leady)
00965       if (cs%aGrid  *lgrid   > 0.) lmixscale = min(lmixscale,cs%aGrid  *lgrid)
00966       if (cs%Lfixed          > 0.) lmixscale = min(lmixscale,cs%Lfixed)
00967     else
00968       lmixscale = 0.
00969       if (cs%aDeform*ldeform > 0.) lmixscale = lmixscale + 1./(cs%aDeform*ldeform)
00970       if (cs%aFrict *lfrict  > 0.) lmixscale = lmixscale + 1./(cs%aFrict *lfrict)
00971       if (cs%aRhines*lrhines > 0.) lmixscale = lmixscale + 1./(cs%aRhines*lrhines)
00972       if (cs%aEady  *leady   > 0.) lmixscale = lmixscale + 1./(cs%aEady  *leady)
00973       if (cs%aGrid  *lgrid   > 0.) lmixscale = lmixscale + 1./(cs%aGrid  *lgrid)
00974       if (cs%Lfixed          > 0.) lmixscale = lmixscale + 1./cs%Lfixed
00975       if (lmixscale > 0.) lmixscale = 1. / lmixscale
00976     endif
00977   endif
00978
00979 end subroutine meke_lengthscales_0d
00980
00981 !> Initializes the MOM_MEKE module and reads parameters.
00982 !! Returns True if module is to be used, otherwise returns False.
00983 logical function meke_init(Time, G, US, param_file, diag, CS, MEKE, restart_CS)
00984   type(time_type),         intent(in)    :: Time       !< The current model time.
00985   type(ocean_grid_type),   intent(inout) :: G          !< The ocean's grid structure.
00986   type(unit_scale_type),   intent(in)    :: US         !< A dimensional unit scaling type
00987   type(param_file_type),   intent(in)    :: param_file !< Parameter file parser structure.
00988   type(diag_ctrl), target, intent(inout) :: diag       !< Diagnostics structure.
00989   type(meke_cs),           pointer       :: CS         !< MEKE control structure.
00990   type(meke_type),         pointer       :: MEKE       !< MEKE-related fields.
00991   type(mom_restart_cs),    pointer       :: restart_CS !< Restart control structure for MOM_MEKE.
00992
00993   ! Local variables
00994   real    :: I_T_rescale   ! A rescaling factor for time from the internal representation in this
00995                            ! run to the representation in a restart file.
00996   real    :: L_rescale     ! A rescaling factor for length from the internal representation in this
00997                            ! run to the representation in a restart file.
00998   real    :: MEKE_restoring_timescale ! The timescale used to nudge MEKE toward its equilibrium value.
00999   real :: cdrag            ! The default bottom drag coefficient [nondim].
01000   integer :: i, j, is, ie, js, je, isd, ied, jsd, jed
01001   logical :: laplacian, biharmonic, useVarMix, coldStart
01002   ! This include declares and sets the variable "version".
01003 # include "version_variable.h"
01004   character(len=40)  :: mdl = "MOM_MEKE" ! This module's name.
01005
01006   is = g%isc ; ie = g%iec ; js = g%jsc ; je = g%jec
01007   isd = g%isd ; ied = g%ied ; jsd = g%jsd ; jed = g%jed
01008
01009   ! Determine whether this module will be used
01010   call get_param(param_file, mdl, "USE_MEKE", meke_init, default=.false., do_not_log=.true.)
01011   call log_version(param_file, mdl, version, "", all_default=.not.meke_init)
01012   call get_param(param_file, mdl, "USE_MEKE", meke_init, &
01013                  "If true, turns on the MEKE scheme which calculates "// &
01014                  "a sub-grid mesoscale eddy kinetic energy budget.", &
01015                  default=.false.)
01016   if (.not. meke_init) return
01017
01018   if (.not. associated(meke)) then
01019     ! The MEKE structure should have been allocated in MEKE_alloc_register_restart()
01020     call mom_error(warning, "MEKE_init called with NO associated "// &
01021                             "MEKE-type structure.")
01022     return
01023   endif
```

```
01024   if (associated(cs)) then
01025     call mom_error(warning, &
01026       "MEKE_init called with an associated control structure.")
01027     return
01028   else ; allocate(cs) ; endif
01029
01030   call mom_mesg("MEKE_init: reading parameters ", 5)
01031
01032   ! Read all relevant parameters and write them to the model log.
01033   call get_param(param_file, mdl, "MEKE_DAMPING", cs%MEKE_damping, &
01034                  "The local depth-independent MEKE dissipation rate.", &
01035                  units="s-1", default=0.0, scale=us%T_to_s)
01036   call get_param(param_file, mdl, "MEKE_CD_SCALE", cs%MEKE_Cd_scale, &
01037                  "The ratio of the bottom eddy velocity to the column mean "//&
01038                  "eddy velocity, i.e. sqrt(2*MEKE). This should be less than 1 "//&
01039                  "to account for the surface intensification of MEKE.", &
01040                  units="nondim", default=0.)
01041   call get_param(param_file, mdl, "MEKE_CB", cs%MEKE_Cb, &
01042                  "A coefficient in the expression for the ratio of bottom projected "//&
01043                  "eddy energy and mean column energy (see Jansen et al. 2015).",&
01044                  units="nondim", default=25.)
01045   call get_param(param_file, mdl, "MEKE_MIN_GAMMA2", cs%MEKE_min_gamma, &
01046                  "The minimum allowed value of gamma_b^2.",&
01047                  units="nondim", default=0.0001)
01048   call get_param(param_file, mdl, "MEKE_CT", cs%MEKE_Ct, &
01049                  "A coefficient in the expression for the ratio of barotropic "//&
01050                  "eddy energy and mean column energy (see Jansen et al. 2015).",&
01051                  units="nondim", default=50.)
01052   call get_param(param_file, mdl, "MEKE_GMCOEFF", cs%MEKE_GMcoeff, &
01053                  "The efficiency of the conversion of potential energy "//&
01054                  "into MEKE by the thickness mixing parameterization. "//&
01055                  "If MEKE_GMCOEFF is negative, this conversion is not "//&
01056                  "used or calculated.", units="nondim", default=-1.0)
01057   call get_param(param_file, mdl, "MEKE_GEOMETRIC", cs%MEKE_GEOMETRIC, &
01058                  "If MEKE_GEOMETRIC is true, uses the GM coefficient formulation "//&
01059                  "from the GEOMETRIC framework (Marshall et al., 2012).", default=.false.)
01060   call get_param(param_file, mdl, "MEKE_GEOMETRIC_ALPHA", cs%MEKE_GEOMETRIC_alpha, &
01061                  "The nondimensional coefficient governing the efficiency of the GEOMETRIC \n"//&
01062                  "thickness diffusion.", units="nondim", default=0.05)
01063   call get_param(param_file, mdl, "MEKE_EQUILIBRIUM_ALT", cs%MEKE_equilibrium_alt, &
01064                  "If true, use an alternative formula for computing the (equilibrium)"//&
01065                  "initial value of MEKE.", default=.false.)
01066   call get_param(param_file, mdl, "MEKE_EQUILIBRIUM_RESTORING", cs%MEKE_equilibrium_restoring, &
01067                  "If true, restore MEKE back to its equilibrium value, which is calculated at "//&
01068                  "each time step.", default=.false.)
01069   if (cs%MEKE_equilibrium_restoring) then
01070     call get_param(param_file, mdl, "MEKE_RESTORING_TIMESCALE", meke_restoring_timescale, &
01071                    "The timescale used to nudge MEKE toward its equilibrium value.", units="s", &
01072                    default=1e6, scale=us%T_to_s)
01073     cs%MEKE_restoring_rate = 1.0 / meke_restoring_timescale
01074   endif
01075
01076   call get_param(param_file, mdl, "MEKE_FRCOEFF", cs%MEKE_FrCoeff, &
01077                  "The efficiency of the conversion of mean energy into "//&
01078                  "MEKE.  If MEKE_FRCOEFF is negative, this conversion "//&
01079                  "is not used or calculated.", units="nondim", default=-1.0)
01080   call get_param(param_file, mdl, "MEKE_GMECOEFF", cs%MEKE_GMECoeff, &
01081                  "The efficiency of the conversion of MEKE into mean energy "//&
01082                  "by GME.  If MEKE_GMECOEFF is negative, this conversion "//&
01083                  "is not used or calculated.", units="nondim", default=-1.0)
01084   call get_param(param_file, mdl, "MEKE_BGSRC", cs%MEKE_BGsrc, &
01085                  "A background energy source for MEKE.", units="W kg-1", &
01086                  default=0.0, scale=us%m_to_L**2*us%T_to_s**3)
01087   call get_param(param_file, mdl, "MEKE_KH", cs%MEKE_Kh, &
01088                  "A background lateral diffusivity of MEKE. "//&
01089                  "Use a negative value to not apply lateral diffusion to MEKE.", &
01090                  units="m2 s-1", default=-1.0, scale=us%m_to_L**2*us%T_to_s)
01091   call get_param(param_file, mdl, "MEKE_K4", cs%MEKE_K4, &
01092                  "A lateral bi-harmonic diffusivity of MEKE. "//&
01093                  "Use a negative value to not apply bi-harmonic diffusion to MEKE.", &
01094                  units="m4 s-1", default=-1.0, scale=us%m_to_L**4*us%T_to_s)
01095   call get_param(param_file, mdl, "MEKE_DTSCALE", cs%MEKE_dtScale, &
01096                  "A scaling factor to accelerate the time evolution of MEKE.", &
01097                  units="nondim", default=1.0)
01098   call get_param(param_file, mdl, "MEKE_KHCOEFF", cs%MEKE_KhCoeff, &
01099                  "A scaling factor in the expression for eddy diffusivity "//&
01100                  "which is otherwise proportional to the MEKE velocity- "//&
01101                  "scale times an eddy mixing-length. This factor "//&
01102                  "must be >0 for MEKE to contribute to the thickness/ "//&
01103                  "and tracer diffusivity in the rest of the model.", &
01104                  units="nondim", default=1.0)
01105   call get_param(param_file, mdl, "MEKE_USCALE", cs%MEKE_Uscale, &
01106                  "The background velocity that is combined with MEKE to "//&
01107                  "calculate the bottom drag.", units="m s-1", default=0.0, scale=us%m_s_to_L_T)
01108   call get_param(param_file, mdl, "MEKE_GM_SRC_ALT", cs%GM_src_alt, &
01109                  "If true, use the GM energy conversion form S^2*N^2*kappa rather "//&
01110                  "than the streamfunction for the MEKE GM source term.", default=.false.)
```

```
01111   call get_param(param_file, mdl, "MEKE_VISC_DRAG", cs%visc_drag, &
01112                  "If true, use the vertvisc_type to calculate the bottom "//&
01113                  "drag acting on MEKE.", default=.true.)
01114   call get_param(param_file, mdl, "MEKE_KHTH_FAC", meke%KhTh_fac, &
01115                  "A factor that maps MEKE%Kh to KhTh.", units="nondim", &
01116                  default=0.0)
01117   call get_param(param_file, mdl, "MEKE_KHTR_FAC", meke%KhTr_fac, &
01118                  "A factor that maps MEKE%Kh to KhTr.", units="nondim", &
01119                  default=0.0)
01120   call get_param(param_file, mdl, "MEKE_KHMEKE_FAC", cs%KhMEKE_Fac, &
01121                  "A factor that maps MEKE%Kh to Kh for MEKE itself.", &
01122                  units="nondim", default=0.0)
01123   call get_param(param_file, mdl, "MEKE_OLD_LSCALE", cs%use_old_lscale, &
01124                  "If true, use the old formula for length scale which is "//&
01125                  "a function of grid spacing and deformation radius.",  &
01126                  default=.false.)
01127   call get_param(param_file, mdl, "MEKE_MIN_LSCALE", cs%use_min_lscale, &
01128                  "If true, use a strict minimum of provided length scales "//&
01129                  "rather than harmonic mean.",  &
01130                  default=.false.)
01131   call get_param(param_file, mdl, "MEKE_RD_MAX_SCALE", cs%Rd_as_max_scale, &
01132                  "If true, the length scale used by MEKE is the minimum of "//&
01133                  "the deformation radius or grid-spacing. Only used if "//&
01134                  "MEKE_OLD_LSCALE=True", units="nondim", default=.false.)
01135   call get_param(param_file, mdl, "MEKE_VISCOSITY_COEFF_KU", cs%viscosity_coeff_Ku, &
01136                  "If non-zero, is the scaling coefficient in the expression for"//&
01137                  "viscosity used to parameterize harmonic lateral momentum mixing by"//&
01138                  "unresolved eddies represented by MEKE. Can be negative to"//&
01139                  "represent backscatter from the unresolved eddies.", &
01140                  units="nondim", default=0.0)
01141   call get_param(param_file, mdl, "MEKE_VISCOSITY_COEFF_AU", cs%viscosity_coeff_Au, &
01142                  "If non-zero, is the scaling coefficient in the expression for"//&
01143                  "viscosity used to parameterize biharmonic lateral momentum mixing by"//&
01144                  "unresolved eddies represented by MEKE. Can be negative to"//&
01145                  "represent backscatter from the unresolved eddies.", &
01146                  units="nondim", default=0.0)
01147   call get_param(param_file, mdl, "MEKE_FIXED_MIXING_LENGTH", cs%Lfixed, &
01148                  "If positive, is a fixed length contribution to the expression "//&
01149                  "for mixing length used in MEKE-derived diffusivity.", &
01150                  units="m", default=0.0, scale=us%m_to_L)
01151   call get_param(param_file, mdl, "MEKE_ALPHA_DEFORM", cs%aDeform, &
01152                  "If positive, is a coefficient weighting the deformation scale "//&
01153                  "in the expression for mixing length used in MEKE-derived diffusivity.", &
01154                  units="nondim", default=0.0)
01155   call get_param(param_file, mdl, "MEKE_ALPHA_RHINES", cs%aRhines, &
01156                  "If positive, is a coefficient weighting the Rhines scale "//&
01157                  "in the expression for mixing length used in MEKE-derived diffusivity.", &
01158                  units="nondim", default=0.0)
01159   call get_param(param_file, mdl, "MEKE_ALPHA_EADY", cs%aEady, &
01160                  "If positive, is a coefficient weighting the Eady length scale "//&
01161                  "in the expression for mixing length used in MEKE-derived diffusivity.", &
01162                  units="nondim", default=0.0)
01163   call get_param(param_file, mdl, "MEKE_ALPHA_FRICT", cs%aFrict, &
01164                  "If positive, is a coefficient weighting the frictional arrest scale "//&
01165                  "in the expression for mixing length used in MEKE-derived diffusivity.", &
01166                  units="nondim", default=0.0)
01167   call get_param(param_file, mdl, "MEKE_ALPHA_GRID", cs%aGrid, &
01168                  "If positive, is a coefficient weighting the grid-spacing as a scale "//&
01169                  "in the expression for mixing length used in MEKE-derived diffusivity.", &
01170                  units="nondim", default=0.0)
01171   call get_param(param_file, mdl, "MEKE_COLD_START", coldstart, &
01172                  "If true, initialize EKE to zero. Otherwise a local equilibrium solution "//&
01173                  "is used as an initial condition for EKE.", default=.false.)
01174   call get_param(param_file, mdl, "MEKE_BACKSCAT_RO_C", meke%backscatter_Ro_c, &
01175                  "The coefficient in the Rossby number function for scaling the biharmonic "//&
01176                  "frictional energy source. Setting to non-zero enables the Rossby number function.", &
01177                  units="nondim", default=0.0)
01178   call get_param(param_file, mdl, "MEKE_BACKSCAT_RO_POW", meke%backscatter_Ro_pow, &
01179                  "The power in the Rossby number function for scaling the biharmonic "//&
01180                  "frictional energy source.", units="nondim", default=0.0)
01181   call get_param(param_file, mdl, "MEKE_ADVECTION_FACTOR", cs%MEKE_advection_factor, &
01182                  "A scale factor in front of advection of eddy energy. Zero turns advection off. "//&
01183                  "Using unity would be normal but other values could accommodate a mismatch "//&
01184                  "between the advecting barotropic flow and the vertical structure of MEKE.", &
01185                  units="nondim", default=0.0)
01186   call get_param(param_file, mdl, "MEKE_TOPOGRAPHIC_BETA", cs%MEKE_topographic_beta, &
01187                  "A scale factor to determine how much topographic beta is weighed in " //&
01188                  "computing beta in the expression of Rhines scale. Use 1 if full "//&
01189                  "topographic beta effect is considered; use 0 if it's completely ignored.", &
01190                  units="nondim", default=0.0)
01191
01192   ! Nonlocal module parameters
01193   call get_param(param_file, mdl, "CDRAG", cdrag, &
01194                  "CDRAG is the drag coefficient relating the magnitude of "//&
01195                  "the velocity field to the bottom stress.", units="nondim", &
01196                  default=0.003)
01197   call get_param(param_file, mdl, "MEKE_CDRAG", cs%cdrag, &
```

```
01198                 "Drag coefficient relating the magnitude of the velocity "//&
01199                 "field to the bottom stress in MEKE.", units="nondim", &
01200                 default=cdrag)
01201   call get_param(param_file, mdl, "LAPLACIAN", laplacian, default=.false., do_not_log=.true.)
01202   call get_param(param_file, mdl, "BIHARMONIC", biharmonic, default=.false., do_not_log=.true.)
01203
01204   if (cs%viscosity_coeff_Ku/=0. .and. .not. laplacian) call mom_error(fatal, &
01205                 "LAPLACIAN must be true if MEKE_VISCOSITY_COEFF_KU is true.")
01206
01207   if (cs%viscosity_coeff_Au/=0. .and. .not. biharmonic) call mom_error(fatal, &
01208                 "BIHARMONIC must be true if MEKE_VISCOSITY_COEFF_AU is true.")
01209
01210   call get_param(param_file, mdl, "DEBUG", cs%debug, default=.false., do_not_log=.true.)
01211
01212   ! Identify if any lateral diffusive processes are active
01213   cs%kh_flux_enabled = .false.
01214   if ((cs%MEKE_KH >= 0.0)  .or. (cs%KhMEKE_FAC > 0.0) .or. (cs%MEKE_advection_factor > 0.0)) &
01215     cs%kh_flux_enabled = .true.
01216
01217 ! Register fields for output from this module.
01218   cs%diag => diag
01219   cs%id_MEKE = register_diag_field('ocean_model', 'MEKE', diag%axesT1, time, &
01220      'Mesoscale Eddy Kinetic Energy', 'm2 s-2', conversion=us%L_T_to_m_s**2)
01221   if (.not. associated(meke%MEKE)) cs%id_MEKE = -1
01222   cs%id_Kh = register_diag_field('ocean_model', 'MEKE_KH', diag%axesT1, time, &
01223      'MEKE derived diffusivity', 'm2 s-1', conversion=us%L_to_m**2*us%s_to_T)
01224   if (.not. associated(meke%Kh)) cs%id_Kh = -1
01225   cs%id_Ku = register_diag_field('ocean_model', 'MEKE_KU', diag%axesT1, time, &
01226      'MEKE derived lateral viscosity', 'm2 s-1', conversion=us%L_to_m**2*us%s_to_T)
01227   if (.not. associated(meke%Ku)) cs%id_Ku = -1
01228   cs%id_Au = register_diag_field('ocean_model', 'MEKE_AU', diag%axesT1, time, &
01229      'MEKE derived lateral biharmonic viscosity', 'm4 s-1', conversion=us%L_to_m**4*us%s_to_T)
01230   if (.not. associated(meke%Au)) cs%id_Au = -1
01231   cs%id_Ue = register_diag_field('ocean_model', 'MEKE_Ue', diag%axesT1, time, &
01232      'MEKE derived eddy-velocity scale', 'm s-1', conversion=us%L_T_to_m_s)
01233   if (.not. associated(meke%MEKE)) cs%id_Ue = -1
01234   cs%id_Ub = register_diag_field('ocean_model', 'MEKE_Ub', diag%axesT1, time, &
01235      'MEKE derived bottom eddy-velocity scale', 'm s-1', conversion=us%L_T_to_m_s)
01236   if (.not. associated(meke%MEKE)) cs%id_Ub = -1
01237   cs%id_Ut = register_diag_field('ocean_model', 'MEKE_Ut', diag%axesT1, time, &
01238      'MEKE derived barotropic eddy-velocity scale', 'm s-1', conversion=us%L_T_to_m_s)
01239   if (.not. associated(meke%MEKE)) cs%id_Ut = -1
01240   cs%id_src = register_diag_field('ocean_model', 'MEKE_src', diag%axesT1, time, &
01241      'MEKE energy source', 'm2 s-3', conversion=(us%L_T_to_m_s**2)*us%s_to_T)
01242   cs%id_decay = register_diag_field('ocean_model', 'MEKE_decay', diag%axesT1, time, &
01243      'MEKE decay rate', 's-1', conversion=us%s_to_T)
01244   cs%id_GM_src = register_diag_field('ocean_model', 'MEKE_GM_src', diag%axesT1, time, &
01245      'MEKE energy available from thickness mixing', &
01246      'W m-2', conversion=us%RZ3_T3_to_W_m2*us%L_to_Z**2)
01247   if (.not. associated(meke%GM_src)) cs%id_GM_src = -1
01248   cs%id_mom_src = register_diag_field('ocean_model', 'MEKE_mom_src',diag%axesT1, time, &
01249      'MEKE energy available from momentum', &
01250      'W m-2', conversion=us%RZ3_T3_to_W_m2*us%L_to_Z**2)
01251   if (.not. associated(meke%mom_src)) cs%id_mom_src = -1
01252   cs%id_GME_snk = register_diag_field('ocean_model', 'MEKE_GME_snk',diag%axesT1, time, &
01253      'MEKE energy lost to GME backscatter', &
01254      'W m-2', conversion=us%RZ3_T3_to_W_m2*us%L_to_Z**2)
01255   if (.not. associated(meke%GME_snk)) cs%id_GME_snk = -1
01256   cs%id_Le = register_diag_field('ocean_model', 'MEKE_Le', diag%axesT1, time, &
01257      'Eddy mixing length used in the MEKE derived eddy diffusivity', 'm', conversion=us%L_to_m)
01258   cs%id_Lrhines = register_diag_field('ocean_model', 'MEKE_Lrhines', diag%axesT1, time, &
01259      'Rhines length scale used in the MEKE derived eddy diffusivity', 'm', conversion=us%L_to_m)
01260   cs%id_Leady = register_diag_field('ocean_model', 'MEKE_Leady', diag%axesT1, time, &
01261      'Eady length scale used in the MEKE derived eddy diffusivity', 'm', conversion=us%L_to_m)
01262   cs%id_gamma_b = register_diag_field('ocean_model', 'MEKE_gamma_b', diag%axesT1, time, &
01263      'Ratio of bottom-projected eddy velocity to column-mean eddy velocity', 'nondim')
01264   cs%id_gamma_t = register_diag_field('ocean_model', 'MEKE_gamma_t', diag%axesT1, time, &
01265      'Ratio of barotropic eddy velocity to column-mean eddy velocity', 'nondim')
01266
01267   if (cs%kh_flux_enabled) then
01268     cs%id_KhMEKE_u = register_diag_field('ocean_model', 'KHMEKE_u', diag%axesCu1, time, &
01269      'Zonal diffusivity of MEKE', 'm2 s-1', conversion=us%L_to_m**2*us%s_to_T)
01270     cs%id_KhMEKE_v = register_diag_field('ocean_model', 'KHMEKE_v', diag%axesCv1, time, &
01271      'Meridional diffusivity of MEKE', 'm2 s-1', conversion=us%L_to_m**2*us%s_to_T)
01272   endif
01273
01274   if (cs%MEKE_equilibrium_restoring) then
01275     cs%id_MEKE_equilibrium = register_diag_field('ocean_model', 'MEKE_equilibrium', diag%axesT1, time, &
01276      'Equilibrated Mesoscale Eddy Kinetic Energy', 'm2 s-2', conversion=us%L_T_to_m_s**2)
01277   endif
01278
01279   cs%id_clock_pass = cpu_clock_id('(Ocean continuity halo updates)', grain=clock_routine)
01280
01281   ! Detect whether this instance of MEKE_init() is at the beginning of a run
01282   ! or after a restart. If at the beginning, we will initialize MEKE to a local
01283   ! equilibrium.
01284   cs%initialize = .not.query_initialized(meke%MEKE, "MEKE", restart_cs)
```

```
01285   if (coldstart) cs%initialize = .false.
01286   if (cs%initialize) call mom_error(warning, &
01287                       "MEKE_init: Initializing MEKE with a local equilibrium balance.")
01288
01289   ! Account for possible changes in dimensional scaling for variables that have been
01290   ! read from a restart file.
01291   i_t_rescale = 1.0
01292   if ((us%s_to_T_restart /= 0.0) .and. (us%s_to_T_restart /= us%s_to_T)) &
01293     i_t_rescale = us%s_to_T_restart / us%s_to_T
01294   l_rescale = 1.0
01295   if ((us%m_to_L_restart /= 0.0) .and. (us%m_to_L_restart /= us%m_to_L)) &
01296     l_rescale = us%m_to_L / us%m_to_L_restart
01297
01298   if (l_rescale*i_t_rescale /= 1.0) then
01299     if (associated(meke%MEKE)) then ; if (query_initialized(meke%MEKE, "MEKE_MEKE", restart_cs)) then
01300       do j=js,je ; do i=is,ie
01301         meke%MEKE(i,j) = l_rescale*i_t_rescale * meke%MEKE(i,j)
01302       enddo ; enddo
01303     endif ; endif
01304   endif
01305   if (l_rescale**2*i_t_rescale /= 1.0) then
01306     if (associated(meke%Kh)) then ; if (query_initialized(meke%Kh, "MEKE_Kh", restart_cs)) then
01307       do j=js,je ; do i=is,ie
01308         meke%Kh(i,j) = l_rescale**2*i_t_rescale * meke%Kh(i,j)
01309       enddo ; enddo
01310     endif ; endif
01311     if (associated(meke%Ku)) then ; if (query_initialized(meke%Ku, "MEKE_Ku", restart_cs)) then
01312       do j=js,je ; do i=is,ie
01313         meke%Ku(i,j) = l_rescale**2*i_t_rescale * meke%Ku(i,j)
01314       enddo ; enddo
01315     endif ; endif
01316     if (associated(meke%Kh_diff)) then ; if (query_initialized(meke%Kh, "MEKE_Kh_diff", restart_cs)) then
01317       do j=js,je ; do i=is,ie
01318         meke%Kh_diff(i,j) = l_rescale**2*i_t_rescale * meke%Kh_diff(i,j)
01319       enddo ; enddo
01320     endif ; endif
01321   endif
01322   if (l_rescale**4*i_t_rescale /= 1.0) then
01323     if (associated(meke%Au)) then ; if (query_initialized(meke%Au, "MEKE_Au", restart_cs)) then
01324       do j=js,je ; do i=is,ie
01325         meke%Au(i,j) = l_rescale**4*i_t_rescale * meke%Au(i,j)
01326       enddo ; enddo
01327     endif ; endif
01328   endif
01329
01330   ! Set up group passes.  In the case of a restart, these fields need a halo update now.
01331   if (associated(meke%MEKE)) then
01332     call create_group_pass(cs%pass_MEKE, meke%MEKE, g%Domain)
01333     if (associated(meke%Kh_diff)) call create_group_pass(cs%pass_MEKE, meke%Kh_diff, g%Domain)
01334     if (.not.cs%initialize) call do_group_pass(cs%pass_MEKE, g%Domain)
01335   endif
01336   if (associated(meke%Kh)) call create_group_pass(cs%pass_Kh, meke%Kh, g%Domain)
01337   if (associated(meke%Ku)) call create_group_pass(cs%pass_Kh, meke%Ku, g%Domain)
01338   if (associated(meke%Au)) call create_group_pass(cs%pass_Kh, meke%Au, g%Domain)
01339
01340   if (associated(meke%Kh) .or. associated(meke%Ku) .or. associated(meke%Au)) &
01341     call do_group_pass(cs%pass_Kh, g%Domain)
01342
01343 end function meke_init
01344
01345 !> Allocates memory and register restart fields for the MOM_MEKE module.
01346 subroutine meke_alloc_register_restart(HI, param_file, MEKE, restart_CS)
01347 ! Arguments
01348   type(hor_index_type),  intent(in)    :: HI         !< Horizontal index structure
01349   type(param_file_type), intent(in)    :: param_file !< Parameter file parser structure.
01350   type(meke_type),       pointer       :: MEKE       !< A structure with MEKE-related fields.
01351   type(mom_restart_cs),  pointer       :: restart_CS !< Restart control structure for MOM_MEKE.
01352 ! Local variables
01353   type(vardesc) :: vd
01354   real :: MEKE_GMcoeff, MEKE_FrCoeff, MEKE_GMECoeff, MEKE_KHCoeff, MEKE_viscCoeff_Ku, MEKE_viscCoeff_Au
01355   logical :: Use_KH_in_MEKE
01356   logical :: useMEKE
01357   integer :: isd, ied, jsd, jed
01358
01359 ! Determine whether this module will be used
01360   usemeke = .false.; call read_param(param_file,"USE_MEKE",usemeke)
01361
01362 ! Read these parameters to determine what should be in the restarts
01363   meke_gmcoeff =-1.; call read_param(param_file,"MEKE_GMCOEFF",meke_gmcoeff)
01364   meke_frcoeff =-1.; call read_param(param_file,"MEKE_FRCOEFF",meke_frcoeff)
01365   meke_gmecoeff =-1.; call read_param(param_file,"MEKE_GMECOEFF",meke_gmecoeff)
01366   meke_khcoeff =1.; call read_param(param_file,"MEKE_KHCOEFF",meke_khcoeff)
01367   meke_visccoeff_ku =0.; call read_param(param_file,"MEKE_VISCOSITY_COEFF_KU",meke_visccoeff_ku)
01368   meke_visccoeff_au =0.; call read_param(param_file,"MEKE_VISCOSITY_COEFF_AU",meke_visccoeff_au)
01369   use_kh_in_meke = .false.; call read_param(param_file,"USE_KH_IN_MEKE", use_kh_in_meke)
01370 ! Allocate control structure
01371   if (associated(meke)) then
```

```
01372     call mom_error(warning, "MEKE_alloc_register_restart called with an associated "// &
01373                             "MEKE type.")
01374     return
01375   else; allocate(meke); endif
01376
01377   if (.not. usemeke) return
01378
01379 ! Allocate memory
01380   call mom_mesg("MEKE_alloc_register_restart: allocating and registering", 5)
01381   isd = hi%isd ; ied = hi%ied ; jsd = hi%jsd ; jed = hi%jed
01382   allocate(meke%MEKE(isd:ied,jsd:jed)) ; meke%MEKE(:,:) = 0.0
01383   vd = var_desc("MEKE", "m2 s-2", hor_grid='h', z_grid='1', &
01384            longname="Mesoscale Eddy Kinetic Energy")
01385   call register_restart_field(meke%MEKE, vd, .false., restart_cs)
01386   if (meke_gmcoeff>=0.) then
01387     allocate(meke%GM_src(isd:ied,jsd:jed)) ; meke%GM_src(:,:) = 0.0
01388   endif
01389   if (meke_frcoeff>=0. .or. meke_gmecoeff>=0.)  then
01390     allocate(meke%mom_src(isd:ied,jsd:jed)) ; meke%mom_src(:,:) = 0.0
01391   endif
01392   if (meke_gmecoeff>=0.) then
01393     allocate(meke%GME_snk(isd:ied,jsd:jed)) ; meke%GME_snk(:,:) = 0.0
01394   endif
01395   if (meke_khcoeff>=0.) then
01396     allocate(meke%Kh(isd:ied,jsd:jed)) ; meke%Kh(:,:) = 0.0
01397     vd = var_desc("MEKE_Kh", "m2 s-1", hor_grid='h', z_grid='1', &
01398              longname="Lateral diffusivity from Mesoscale Eddy Kinetic Energy")
01399     call register_restart_field(meke%Kh, vd, .false., restart_cs)
01400   endif
01401   allocate(meke%Rd_dx_h(isd:ied,jsd:jed)) ; meke%Rd_dx_h(:,:) = 0.0
01402   if (meke_visccoeff_ku/=0.) then
01403     allocate(meke%Ku(isd:ied,jsd:jed)) ; meke%Ku(:,:) = 0.0
01404     vd = var_desc("MEKE_Ku", "m2 s-1", hor_grid='h', z_grid='1', &
01405               longname="Lateral viscosity from Mesoscale Eddy Kinetic Energy")
01406     call register_restart_field(meke%Ku, vd, .false., restart_cs)
01407   endif
01408   if (use_kh_in_meke) then
01409     allocate(meke%Kh_diff(isd:ied,jsd:jed)) ; meke%Kh_diff(:,:) = 0.0
01410     vd = var_desc("MEKE_Kh_diff", "m2 s-1",hor_grid='h',z_grid='1', &
01411               longname="Copy of thickness diffusivity for diffusing MEKE")
01412     call register_restart_field(meke%Kh_diff, vd, .false., restart_cs)
01413   endif
01414
01415   if (meke_visccoeff_au/=0.) then
01416     allocate(meke%Au(isd:ied,jsd:jed)) ; meke%Au(:,:) = 0.0
01417     vd = var_desc("MEKE_Au", "m4 s-1", hor_grid='h', z_grid='1', &
01418               longname="Lateral biharmonic viscosity from Mesoscale Eddy Kinetic Energy")
01419     call register_restart_field(meke%Au, vd, .false., restart_cs)
01420   endif
01421
01422 end subroutine meke_alloc_register_restart
01423
01424 !> Deallocates any variables allocated in MEKE_init or
01425 !! MEKE_alloc_register_restart.
01426 subroutine meke_end(MEKE, CS)
01427   type(meke_type), pointer :: MEKE !< A structure with MEKE-related fields.
01428   type(meke_cs),   pointer :: CS   !< The control structure for MOM_MEKE.
01429
01430   if (associated(cs)) deallocate(cs)
01431
01432   if (.not.associated(meke)) return
01433
01434   if (associated(meke%MEKE)) deallocate(meke%MEKE)
01435   if (associated(meke%GM_src)) deallocate(meke%GM_src)
01436   if (associated(meke%mom_src)) deallocate(meke%mom_src)
01437   if (associated(meke%GME_snk)) deallocate(meke%GME_snk)
01438   if (associated(meke%Kh)) deallocate(meke%Kh)
01439   if (associated(meke%Kh_diff)) deallocate(meke%Kh_diff)
01440   if (associated(meke%Ku)) deallocate(meke%Ku)
01441   if (associated(meke%Au)) deallocate(meke%Au)
01442   deallocate(meke)
01443
01444 end subroutine meke_end
01445
01446 !> \namespace mom_meke
01447 !!
01448 !! \section section_MEKE The Mesoscale Eddy Kinetic Energy (MEKE) framework
01449 !!
01450 !! The MEKE framework accounts for the mean potential energy removed by
01451 !! the first order closures used to parameterize mesoscale eddies.
01452 !! It requires closure at the second order, namely dissipation and transport
01453 !! of eddy energy.
01454 !!
01455 !! Monitoring the sub-grid scale eddy energy budget provides a means to predict
01456 !! a sub-grid eddy-velocity scale which can be used in the lower order closures.
01457 !!
01458 !! \subsection section_MEKE_equations MEKE equations
```

```
01459 !!
01460 !! The eddy kinetic energy equation is:
01461 !! \f[ \partial_{\tilde{t}} E =
01462 !!   \overbrace{ \dot{E}_b + \gamma_\eta \dot{E}_\eta + \gamma_v \dot{E}_v
01463 !!             }^\text{sources}
01464 !! - \overbrace{ ( \lambda + C_d | U_d | \gamma_b^2 ) E
01465 !!             }^\text{local dissipation}
01466 !! + \overbrace{ \nabla \cdot ( ( \kappa_E + \gamma_M \kappa_M ) \nabla E
01467 !!                            - \kappa_4 \nabla^3 E )
01468 !!             }^\text{smoothing}
01469 !! \f]
01470 !! where \f$ E \f$ is the eddy kinetic energy (variable <code>MEKE</code>) with units of
01471 !! m<sup>2</sup>s<sup>-2</sup>,
01472 !! and \f$\tilde{t} = a t\f$ is a scaled time. The non-dimensional factor
01473 !! \f$ a\geq 1 \f$ is used to accelerate towards equilibrium.
01474 !!
01475 !! The MEKE equation is two-dimensional and obtained by depth averaging the
01476 !! the three-dimensional eddy energy equation. In the following expressions
01477 !! \f$ \left< \phi \right> = \frac{1}{H} \int^\eta_{-D} \phi \, dz \f$ maps
01478 !! three dimensional terms into the two-dimensional quantities needed.
01479 !!
01480 !! \subsubsection section_MEKE_source_terms MEKE source terms
01481 !!
01482 !! The source term \f$ \dot{E}_b \f$ is a constant background source
01483 !! of energy intended to avoid the limit \f$E\rightarrow 0\f$.
01484 !!
01485 !! The "GM" source term
01486 !! \f[ \dot{E}_\eta = - \left< \overline{w^\prime b^\prime} \right>
01487 !! = \left< \kappa_h N^2S^2 \right>
01488 !! \approx \left< \kappa_h g\prime |\nabla_\sigma \eta|^2 \right>\f]
01489 !! equals the mean potential energy removed by the Gent-McWilliams closure,
01490 !! and is excluded/included in the MEKE budget by the efficiency parameter
01491 !! \f$ \gamma_\eta \in [0,1] \f$.
01492 !!
01493 !! The "frictional" source term
01494 !! \f[ \dot{E}_{v} = \left<  \partial_i u_j \tau_{ij} \right> \f]
01495 !! equals the mean kinetic energy removed by lateral viscous fluxes, and
01496 !! is excluded/included in the MEKE budget by the efficiency parameter
01497 !! \f$ \gamma_v \in [0,1] \f$.
01498 !!
01499 !! \subsubsection section_MEKE_dissipation_terms MEKE dissipation terms
01500 !!
01501 !! The local dissipation of \f$ E \f$ is parameterized through a linear
01502 !! damping, \f$\lambda\f$, and bottom drag, \f$ C_d | U_d | \gamma_b^2 \f$.
01503 !! The \f$ \gamma_b \f$ accounts for the weak projection of the column-mean
01504 !! eddy velocty to the bottom. In other words, the bottom velocity is
01505 !! estimated as \f$ \gamma_b U_e \f$.
01506 !! The bottom drag coefficient, \f$ C_d \f$ is the same as that used in the bottom
01507 !! friction in the mean model equations.
01508 !!
01509 !! The bottom drag velocity scale, \f$ U_d \f$, has contributions from the
01510 !! resolved state and \f$ E \f$:
01511 !! \f[ U_d = \sqrt{ U_b^2 + |u|^2_{z=-D} + |\gamma_b U_e|^2 } .\f]
01512 !! where the eddy velocity scale, \f$ U_e \f$, is given by:
01513 !! \f[ U_e = \sqrt{ 2 E } .\f]
01514 !! \f$ U_b \f$ is a constant background bottom velocity scale and is
01515 !! typically not used (i.e. set to zero).
01516 !!
01517 !! Following Jansen et al., 2015, the projection of eddy energy on to the bottom
01518 !! is given by the ratio of bottom energy to column mean energy:
01519 !! \f[
01520 !! \gamma_b^2  = \frac{E_b}{E} = \gamma_{d0}
01521 !!    + \left( 1 + c_{b} \frac{L_d}{L_f} \right)^{-\frac{4}{5}}
01522 !! ,
01523 !! \f]
01524 !! \f[
01525 !! \gamma_b^2  \leftarrow  \max{\left( \gamma_b^2, \gamma_{min}^2 \right)}
01526 !! .
01527 !! \f]
01528 !!
01529 !! \subsection section_MEKE_smoothing MEKE smoothing terms
01530 !!
01531 !! \f$ E \f$ is laterally diffused by a diffusivity \f$ \kappa_E + \gamma_M
01532 !! \kappa_M \f$ where \f$ \kappa_E \f$ is a constant diffusivity and the term
01533 !! \f$ \gamma_M \kappa_M \f$ is a "self diffusion" using the diffusivity
01534 !! calculated in the section \ref section_MEKE_diffusivity.
01535 !! \f$ \kappa_4 \f$ is a constant bi-harmonic diffusivity.
01536 !!
01537 !! \subsection section_MEKE_diffusivity Diffusivity derived from MEKE
01538 !!
01539 !! The predicted eddy velocity scale, \f$ U_e \f$, can be combined with a
01540 !! mixing length scale to form a diffusivity.
01541 !! The primary use of a MEKE derived diffusivity is for use in thickness
01542 !! diffusion (module mom_thickness_diffuse) and optionally in along
01543 !! isopycnal mixing of tracers (module mom_tracer_hor_diff).
01544 !! The original form used (enabled with MEKE_OLD_LSCALE=True):
01545 !!
```

```
01546 !! \f[  \kappa_M = \gamma_\kappa \sqrt{ \gamma_t^2 U_e^2 A_\Delta } \f]
01547 !!
01548 !! where \f$ A_\Delta \f$ is the area of the grid cell.
01549 !! Following Jansen et al., 2015, we now use
01550 !!
01551 !! \f[  \kappa_M = \gamma_\kappa l_M \sqrt{ \gamma_t^2 U_e^2 } \f]
01552 !!
01553 !! where \f$ \gamma_\kappa \in [0,1] \f$ is a non-dimensional factor and,
01554 !! following Jansen et al., 2015, \f$\gamma_t^2\f$ is the ratio of barotropic
01555 !! eddy energy to column mean eddy energy given by
01556 !! \f[
01557 !! \gamma_t^2  = \frac{E_t}{E} = \left( 1 + c_{t} \frac{L_d}{L_f} \right)^{-\frac{1}{4}}
01558 !! ,
01559 !! \f]
01560 !! \f[
01561 !! \gamma_t^2  \leftarrow  \max{\left( \gamma_t^2, \gamma_{min}^2 \right)}
01562 !! .
01563 !! \f]
01564 !!
01565 !! The length-scale is a configurable combination of multiple length scales:
01566 !!
01567 !! \f[
01568 !! l_M = \left(
01569 !!       \frac{\alpha_d}{L_d}
01570 !!     + \frac{\alpha_f}{L_f}
01571 !!     + \frac{\alpha_R}{L_R}
01572 !!     + \frac{\alpha_e}{L_e}
01573 !!     + \frac{\alpha_\Delta}{L_\Delta}
01574 !!     + \frac{\delta[L_c]}{L_c}
01575 !!       \right)^{-1}
01576 !! \f]
01577 !!
01578 !! where
01579 !!
01580 !! \f{eqnarray*}{
01581 !! L_d &= & \sqrt{\frac{c_g^2}{f^2+2\beta c_g}} \sim \frac{ c_g }{f} \\\\
01582 !! L_R &= & \sqrt{\frac{U_e}{\beta^*}} \\\\
01583 !! L_e &= & \frac{U_e}{|S| N} \\\\
01584 !! L_f &= & \frac{H}{c_d} \\\\
01585 !! L_\Delta &= & \sqrt{A_\Delta} .
01586 !! \f}
01587 !!
01588 !! \f$L_c\f$ is a constant and \f$\delta[L_c]\f$ is the impulse function so that the term
01589 !! \f$\frac{\delta[L_c]}{L_c}\f$ evaluates to \f$\frac{1}{L_c}\f$ when \f$L_c\f$ is non-zero
01590 !! but is dropped if \f$L_c=0\f$.
01591 !!
01592 !! \f$\beta^*\f$ is the effective \f$\beta\f$ that combines both the planetary vorticity
01593 !! gradient (i.e. \f$\beta=\nabla f\f$) and the topographic \f$\beta\f$ effect,
01594 !! with the latter weighed by a weighting constant, \f$c_\beta\f$, that varies
01595 !! from 0 to 1, so that \f$c_\beta=0\f$ means the topographic \f$\beta\f$ effect is ignored,
01596 !! while \f$c_\beta=1\f$ means it is fully considered. The new \f$\beta^*\f$ therefore
01597 !! takes the form of
01598 !!
01599 !! \f[
01600 !! \beta^* = \sqrt{( \partial_xf - c_\beta\frac{f}{D}\partial_xD )^2 +
01601 !!           ( \partial_yf - c_\beta\frac{f}{D}\partial_yD )^2}
01602 !! \f]
01603 !! where \f$D\f$ is water column depth at T points.
01604 !!
01605 !! \subsection section_MEKE_viscosity Viscosity derived from MEKE
01606 !!
01607 !! As for \f$ \kappa_M \f$, the predicted eddy velocity scale can be
01608 !! used to form a harmonic eddy viscosity,
01609 !!
01610 !! \f[  \kappa_u = \gamma_u \sqrt{ U_e^2 A_\Delta }  \f]
01611 !!
01612 !! as well as a biharmonic eddy viscosity,
01613 !!
01614 !! \f[  \kappa_4 = \gamma_4 \sqrt{ U_e^2 A_\Delta^3 }  \f]
01615 !!
01616 !! \subsection section_MEKE_limit_case Limit cases for local source-dissipative balance
01617 !!
01618 !! Note that in steady-state (or when \f$ a>>1 \f$) and there is no
01619 !! diffusion of \f$ E \f$ then
01620 !! \f[ \overline{E} \approx \frac{ \dot{E}_b + \gamma_\eta \dot{E}_\eta +
01621 !!               \gamma_v \dot{E}_v }{ \lambda + C_d|U_d|\gamma_b^2 } . \f]
01622 !!
01623 !! In the linear drag limit, where
01624 !! \f$ U_e << \min(U_b, |u|_{z=-D}, C_d^{-1}\lambda) \f$, the equilibrium becomes
01625 !! \f$ \overline{E} \approx \frac{ \dot{E}_b + \gamma_\eta \dot{E}_\eta +
01626 !!               \gamma_v \dot{E}_v }{ \lambda + C_d \sqrt{ U_b^2 + |u|^2_{z=-D} } } \f$.
01627 !!
01628 !! In the nonlinear drag limit, where \f$ U_e >> \max(U_b, |u|_{z=-D}, C_d^{-1}\lambda) \f$,
01629 !! the equilibrium becomes
01630 !! \f$ \overline{E} \approx \left( \frac{ \dot{E}_b + \gamma_\eta \dot{E}_\eta +
01631 !!               \gamma_v \dot{E}_v }{ \sqrt{2} C_d \gamma_b^3 } \right)^\frac{2}{3} \f$.
01632 !!
```

```
01633 !! \subsubsection section_MEKE_module_parameters MEKE module parameters
01634 !!
01635 !! | Symbol                | Module parameter |
01636 !! | ------                | ---------------- |
01637 !! | -                     | <code>USE_MEKE</code> |
01638 !! | \f$ a \f$             | <code>MEKE_DTSCALE</code> |
01639 !! | \f$ \dot{E}_b \f$     | <code>MEKE_BGSRC</code> |
01640 !! | \f$ \gamma_\eta \f$   | <code>MEKE_GMCOEFF</code> |
01641 !! | \f$ \gamma_v \f$      | <code>MEKE_FrCOEFF</code> |
01642 !! | \f$ \lambda \f$       | <code>MEKE_DAMPING</code> |
01643 !! | \f$ U_b \f$           | <code>MEKE_USCALE</code> |
01644 !! | \f$ \gamma_{d0} \f$   | <code>MEKE_CD_SCALE</code> |
01645 !! | \f$ c_{b} \f$         | <code>MEKE_CB</code> |
01646 !! | \f$ c_{t} \f$         | <code>MEKE_CT</code> |
01647 !! | \f$ \kappa_E \f$      | <code>MEKE_KH</code> |
01648 !! | \f$ \kappa_4 \f$      | <code>MEKE_K4</code> |
01649 !! | \f$ \gamma_\kappa \f$ | <code>MEKE_KHCOEFF</code> |
01650 !! | \f$ \gamma_M \f$      | <code>MEKE_KHMEKE_FAC</code> |
01651 !! | \f$ \gamma_u \f$      | <code>MEKE_VISCOSITY_COEFF_KU</code> |
01652 !! | \f$ \gamma_4 \f$      | <code>MEKE_VISCOSITY_COEFF_AU</code> |
01653 !! | \f$ \gamma_{min}^2 \f$| <code>MEKE_MIN_GAMMA2</code> |
01654 !! | \f$ \alpha_d \f$      | <code>MEKE_ALPHA_DEFORM</code> |
01655 !! | \f$ \alpha_f \f$      | <code>MEKE_ALPHA_FRICT</code> |
01656 !! | \f$ \alpha_R \f$      | <code>MEKE_ALPHA_RHINES</code> |
01657 !! | \f$ \alpha_e \f$      | <code>MEKE_ALPHA_EADY</code> |
01658 !! | \f$ \alpha_\Delta \f$ | <code>MEKE_ALPHA_GRID</code> |
01659 !! | \f$ L_c \f$           | <code>MEKE_FIXED_MIXING_LENGTH</code> |
01660 !! | \f$ c_\beta \f$       | <code>MEKE_TOPOGRAPHIC_BETA</code> |
01661 !! | -                     | <code>MEKE_KHTH_FAC</code> |
01662 !! | -                     | <code>MEKE_KHTR_FAC</code> |
01663 !!
01664 !! | Symbol                | Model parameter |
01665 !! | ------                | ---------------- |
01666 !! | \f$ C_d \f$           | <code>CDRAG</code> |
01667 !!
01668 !! \subsection section_MEKE_references References
01669 !!
01670 !! Jansen, M. F., A. J. Adcroft, R. Hallberg, and I. M. Held, 2015: Parameterization of eddy fluxes based
       on a
01671 !! mesoscale energy budget. Ocean Modelling, 92, 28--41, http://doi.org/10.1016/j.ocemod.2015.05.007 .
01672 !!
01673 !! Marshall, D. P., and A. J. Adcroft, 2010: Parameterization of ocean eddies: Potential vorticity mixing,
       energetics
01674 !! and Arnold first stability theorem. Ocean Modelling, 32, 188--204,
       http://doi.org/10.1016/j.ocemod.2010.02.001 .
01675
01676 end module mom_meke
01677
```

44.42 /home/cermak/src/MOM6/src/parameterizations/vertical/_CVMix_KPP.dox File Reference

44.43 /home/cermak/src/MOM6/src/tracer/_Advection.dox File Reference

44.44 /home/cermak/src/MOM6/src/tracer/_Discrete_tracer.dox File Reference

44.45 /home/cermak/src/MOM6/src/tracer/_Horizontal_diffusion.dox File Reference

44.46 /home/cermak/src/MOM6/src/tracer/_Passive_tracer.dox File Reference

44.47 /home/cermak/src/MOM6/src/tracer/_Tracer_fluxes.dox File Reference

44.48 /home/cermak/src/MOM6/src/tracer/_Tracer_timestep.dox File Reference

44.49 /home/cermak/src/MOM6/src/tracer/_Tracer_Transport.dox File Reference

44.50 /home/cermak/src/MOM6/src/tracer/_Vertical_diffusion.dox File Reference