Ocean-Atmosphere Coupling Software

Raza

June 29, 2025

1 Introduction

The software, written in Python, simulates ocean-atmosphere interactions with a focus on physical processes such as heat, momentum, freshwater, and CO_2 fluxes, turbulent mixing, boundary layer dynamics, cloud microphysics, air-sea interactions, and oceanic eddies and fronts.

2 Functionalities

The simulator provides the following core functionalities:

- **Core Simulation**: Simulates coupled ocean-atmosphere dynamics on a 2D grid with variable time steps for ocean and atmosphere, computing temperature, salinity, velocity, moisture, and CO_2 concentrations, accounting for advection, diffusion, and fluxes.
- **Physical Processes**: Models momentum, heat, freshwater, and CO_2 fluxes, turbulent mixing, boundary layer schemes (Bulk and KPP), cloud microphysics, air-sea interactions, and surface layer physics, with support for potential vorticity (PV) frontogenesis for oceanic eddies and fronts.
- **Numerical Methods**: Implements Euler and Runge-Kutta 4 (RK4) time-stepping methods for boundary layer schemes, finite difference methods for advection and diffusion, and adaptive mesh refinement with variable resolution grids for computational efficiency.
- **Visualization and Analysis**: Visualizes ocean and atmosphere temperature fields, mean temperatures over time, and refinement regions, with specialized analyses for turbulent mixing, surface layer physics, boundary layer schemes, air-sea interactions, and cloud microphysics.
- **Parameter Control**: Allows configuration of physical parameters (e.g., drag coefficient, wind speed, mixing coefficient) and numerical parameters (e.g., grid size, time step), with conservation constraints for freshwater and CO_2 fluxes.

3 Simulation Logic

The simulation is driven by the `OceanAtmosphereModel` class (`Model.py`), which integrates physical processes and numerical methods. The main simulation loop in `MainApp.py` uses a timer to step through time, updating the model state and visualizing results. Specialized modules provide focused analyses of specific processes.

3.1 Main Simulation Workflow

1. **Initialization** (`Model.py`):
    - Sets up a 2D grid with initial conditions for ocean temperature, salinity, velocities, atmosphere temperature, moisture, and CO_2 concentrations.
    - Initializes a `VariableResolutionGrid` (`VariableResolution.py`) with finer resolution near coasts.
2. **Time Stepping** (`Model.py`):
    - Advances the simulation using different time steps for ocean (Δt) and atmosphere ($\Delta t/R$, where R is the time scale ratio).
    - Updates fields: computes fluxes using `TwoWayCoupling`, applies advection and diffusion, updates velocities with momentum flux and Coriolis force, refines the grid using `AdaptiveMeshRefinement`, and applies numerical stability constraints (e.g., clipping temperatures to 250–350 K).
3. **Specialized Analyses**:
    - `TurbulentMixing.py`: Simulates a thermal front and vertical mixing, computing the Ekman spiral and temperature profiles.
    - `SurfaceLayerPhysics.py`: Models wind-driven surface currents and heat fluxes (sensible and latent).
    - `BoundaryLayerSchemes.py`: Analyzes Bulk and KPP schemes for heat flux or diffusivity.
    - `HighOrderTimeStepping.py`: Compares Euler and RK4 methods for boundary layer schemes.
    - `AirSeaInteraction.py`: Computes momentum and heat fluxes across the air-sea interface.
    - `CloudMicroPhysics.py`: Models cloud formation and precipitation based on moisture and temperature.

- `PotentialVorticityFrontogenesis.py`: Simulates PV dynamics, likely used by `OceanicEddyAndFront.py` for eddy and front analyses.

4. **Visualization** (`PlotWidget.py`): Displays temperature fields as heatmaps, with contours for refinement regions and rectangles for nested grids, and plots mean temperatures over time.

5. **Parameter Input** (`ControlPanel.py`): Provides parameters for initial conditions, physical constants, time scales, grid settings, and fluxes, triggering simulation steps and analysis modules.

4 Physics and Mathematical Models

The simulator models physical processes using simplified equations rooted in ocean-atmosphere dynamics. Below are the key models and their equations, formatted to fit within margins.

4.1 Two-Way Coupling

The `TwoWayCoupling` class computes fluxes and mixing between the ocean and atmosphere, ensuring conservation where applicable.

4.1.1 Sea Surface Roughness

- **Purpose**: Adjusts drag coefficient based on wind and ocean currents.
- **Equations**:

$$
\begin{aligned}
u_* &= \sqrt{\frac{\rho_{\text{air}} C_d U^2}{\rho_{\text{water}}}}, \\
z_0 &= \alpha \frac{u_*^2 + 0.1(u_{\text{ocean}}^2 + v_{\text{ocean}}^2)}{g}, \\
C_d' &= C_d \left(1 + 0.1 \log_{10}(z_0)\right),
\end{aligned}
$$

  where u_* is friction velocity, C_d is drag coefficient, U is wind speed, $\rho_{\text{air}} = 1.225\,\text{kg/m}^3$, $\rho_{\text{water}} = 1025\,\text{kg/m}^3$, $\alpha = 0.018$, $g = 9.81\,\text{m/s}^2$, and z_0 is roughness length.

- **Implementation**: `compute_sea_surface_roughness` clips z_0 to $[10^{-6}, 10^{-2}]$ and C_d' to $[10^{-4}, 10^{-2}]$.

4.1.2 Momentum Flux

- **Purpose**: Computes wind stress on the ocean surface.
- **Equation**:

$$\tau = \rho_{\text{air}} C_d' U^2,$$

where C_d' is the adjusted drag coefficient.

- **Implementation**: `compute_momentum_flux` clips τ to $[-10^5, 10^5]$ N/m^2.

4.1.3 Heat Flux

- **Purpose**: Computes sensible and latent heat fluxes.
- **Equations**:

$$\begin{aligned} Q_{\text{sensible}} &= \rho_{\text{air}} C_p k_{\text{sensible}} (T_a - T_o), \\ k_{\text{sensible}} &= 0.01 \frac{C_d'}{C_d}, \\ Q_{\text{latent}} &= \rho_{\text{air}} L_v E \text{sign}(T_a - T_o), \\ Q_{\text{total}} &= Q_{\text{sensible}} + Q_{\text{latent}}, \end{aligned}$$

where $C_p = 1005$ J/kg/K, $L_v = 2.5 \times 10^6$ J/kg, E is evaporation rate, T_a is atmosphere temperature, and T_o is ocean temperature.

- **Implementation**: `compute_heat_flux` clips $T_a - T_o$ to $[-100, 100]$ K and Q_{total} to $[-10^6, 10^6]$ W/m^2.

4.1.4 Freshwater Flux

- **Purpose**: Computes net freshwater flux and salinity change, ensuring mass conservation.
- **Equations**:

$$\begin{aligned} P &= P_0 \left(1 + 0.5 \frac{q}{0.01}\right), \\ E &= E_0 C_{\text{freshwater}} \left(1 + 0.1 \frac{S}{35}\right), \\ F &= P - E, \\ \frac{dS}{dt} &= -\frac{SF}{\rho_{\text{water}} H}, \end{aligned}$$

where P_0 is precipitation rate, E_0 is evaporation rate, $C_{\text{freshwater}}$ is the conservation coefficient, S is salinity, $H = 1000$ m is ocean depth, and q is moisture.

- **Implementation**: `compute_freshwater_flux` clips $q/0.01$ to $[0, 2]$, $S/35$ to $[0.8, 1.2]$, and outputs to $[-10^{-3}, 10^{-3}]$.

4.1.5 CO_2 Flux

- **Purpose**: Computes CO_2 exchange between ocean and atmosphere, ensuring mass conservation.

- **Equations**:

$$
\begin{aligned}
p\mathrm{CO}_{2,\mathrm{ocean}} &= \frac{\mathrm{CO}_{2,\mathrm{ocean}}}{\alpha}, \\
p\mathrm{CO}_{2,\mathrm{atm}} &= \mathrm{CO}_{2,\mathrm{atm}}, \\
F_{\mathrm{CO}_2} &= k_{\mathrm{CO}_2} C_{\mathrm{CO}_2} (p\mathrm{CO}_{2,\mathrm{ocean}} - p\mathrm{CO}_{2,\mathrm{atm}}), \\
F_{\mathrm{CO}_2,\mathrm{ocean}} &= \frac{F_{\mathrm{CO}_2}}{H}, \\
F_{\mathrm{CO}_2,\mathrm{atm}} &= -\frac{F_{\mathrm{CO}_2}}{H_{\mathrm{atm}}},
\end{aligned}
$$

  where $\alpha = 0.03$ is CO_2 solubility, k_{CO_2} is transfer coefficient, C_{CO_2} is conservation coefficient, $H = 1000\,\mathrm{m}$, and $H_{\mathrm{atm}} = 10000\,\mathrm{m}$.

- **Implementation**: `compute_co2_flux` clips outputs to $[-10^{-3}, 10^{-3}]$.

4.1.6 Moisture Advection

- **Purpose**: Computes moisture transport in the atmosphere.

- **Equation**:

$$
\text{Advection} = -u_a \frac{\partial q}{\partial x} - v_a \frac{\partial q}{\partial y},
$$

  where u_a, v_a are atmosphere velocities, and $\frac{\partial q}{\partial x}$, $\frac{\partial q}{\partial y}$ are computed via central differences.

- **Implementation**: `compute_moisture_advection` uses finite differences and clips output to $[-10^{-4}, 10^{-4}]$.

4.1.7 Turbulent Mixing

- **Purpose**: Models mixing driven by temperature gradients and wind.

- **Equations**:

$$
\begin{aligned}
M &= k_m u_* \left(\frac{\partial T}{\partial x} + \frac{\partial T}{\partial y} \right), \\
u_* &= \sqrt{\frac{\rho_{\mathrm{air}} C_d U^2}{\rho_{\mathrm{water}}}},
\end{aligned}
$$

  where k_m is mixing coefficient, and gradients are computed via central differences.

- **Implementation**: `compute_turbulent_mixing` clips output to $[-10^3, 10^3]$.

4.1.8 Radiative Flux

- **Purpose**: Models solar and longwave radiation with greenhouse effects.

- **Equation**:

$$Q_{\text{rad}} = Q_{\text{solar}} - \varepsilon\sigma \left(\frac{T}{300}\right)^4 \times \left(1 + 0.1 \log\left(\frac{\text{CO}_{2,\text{atm}}}{400}\right)\right),$$

  where $\sigma = 5.67 \times 10^{-8}\,\text{W/m}^2/\text{K}^4$, ε is longwave coefficient, and Q_{solar} is solar forcing.

- **Implementation**: `compute_radiative_flux` clips $T/300$ to $[0.8, 1.2]$, $\log(\text{CO}_{2,\text{atm}}/400)$ to $[-1, 1]$, and output to $[-10^6, 10^6]\,\text{W/m}^2$.

4.2 Ocean-Atmosphere Model

The `OceanAtmosphereModel` integrates physical processes over time.

4.2.1 Initialization

- Sets up a 2D grid ($N \times N$) with initial conditions:

$$\begin{aligned} T_o &= T_{\text{base}} + \Delta T \sin\left(\frac{2\pi y}{N}\right), \\ T_a &= T_o - 5, \\ S &= S_0, \quad q = q_0, \quad \text{CO}_{2,\text{ocean}} = \text{CO}_{2,\text{atm}} = C_0, \end{aligned}$$

  where T_o is ocean temperature, T_a is atmosphere temperature, S is salinity, q is moisture, and velocities are initialized to zero.

4.2.2 Time Stepping

- **Ocean Update**:

$$\begin{aligned} T_o^{n+1} &= T_o^n + \Delta t \left(-\mathbf{u}_o \cdot \nabla T_o + k\nabla^2 T_o + \frac{Q_{\text{total}} + Q_{\text{rad}}}{\rho_{\text{water}} C_{p,\text{water}} H}\right), \\ S^{n+1} &= S^n + \Delta t \frac{dS}{dt}, \\ \mathbf{u}_o^{n+1} &= \mathbf{u}_o^n + \Delta t \left(\frac{\tau}{\rho_{\text{water}} H} - f\mathbf{u}_o^{\perp} + k\nabla^2 \mathbf{u}_o\right), \end{aligned}$$

  where k is diffusion coefficient, $f = 10^{-4}\,\text{s}^{-1}$ is Coriolis parameter, and ∇^2 is computed via central differences.

- **Atmosphere Update**:

$$\begin{aligned} T_a^{n+1} &= T_a^n + \frac{\Delta t}{R}\Big(-\mathbf{u}_a \cdot \nabla T_a + k\nabla^2 T_a \\ &\qquad - \frac{Q_{\text{total}} + Q_{\text{rad}}}{\rho_{\text{air}} C_{p,\text{air}} H_{\text{atm}}}\Big), \\ q^{n+1} &= q^n + \frac{\Delta t}{R}\left(\text{Advection} + k\nabla^2 q + \frac{F}{H_{\text{atm}}}\right), \\ \text{CO}_{2,\text{atm}}^{n+1} &= \text{CO}_{2,\text{atm}}^n + \frac{\Delta t}{R} F_{\text{CO}_2,\text{atm}}, \end{aligned}$$

  where R is the time scale ratio (default 10).

- **Numerical Stability**: Clips T_o, T_a to $[250, 350]$ K, velocities to $[-0.5, 0.5]$ m/s, and other fields to prevent divergence.

4.3 Boundary Layer Schemes

These modules model surface boundary layer processes using Bulk and KPP schemes.

4.3.1 Bulk Scheme

- **Heat Flux**:

$$Q = C_h U (T_o - T_a),$$

  where C_h is sensible heat coefficient.

- **Implementation**: `compute_bulk_flux` and `compute_bulk`.

4.3.2 KPP Scheme

- **Diffusivity**:

$$K = k_{\text{kpp}} \left(1 - \frac{z}{h}\right)^2,$$

  where k_{kpp} is mixing coefficient, z is depth, and h is boundary layer depth.

- **Implementation**: `compute_kpp_diffusivity` and `compute_kpp`.

4.3.3 Time Stepping

- **Euler Method**:

$$y^{n+1} = y^n + \Delta t f(y^n),$$

  where f is the flux or diffusivity rate.

- **RK4 Method**:

$$
\begin{aligned}
k_1 &= f(y^n), \\
k_2 &= f\left(y^n + \frac{\Delta t}{2} k_1\right), \\
k_3 &= f\left(y^n + \frac{\Delta t}{2} k_2\right), \\
k_4 &= f(y^n + \Delta t k_3), \\
y^{n+1} &= y^n + \frac{\Delta t}{6}(k_1 + 2k_2 + 2k_3 + k_4).
\end{aligned}
$$

- **Implementation**: `euler_step` and `rk4_step`.

4.4 Turbulent Mixing

- **Thermal Front**:

$$
\begin{aligned}
T_{\text{front}} = T_{\text{cold}} &+ (T_{\text{hot}} - T_{\text{cold}}) \cdot 0.5 \\
&\times \left(1 + \tanh\left(\frac{x - x_0}{L}\right)\right),
\end{aligned}
$$

  where $T_{\text{cold}} = 290\,\text{K}$, $T_{\text{hot}} = 300\,\text{K}$, $x_0 = N_x/2 + 10\sin(0.01Ut)$, and $L = 5$.

- **Ekman Spiral**:

$$
\begin{aligned}
u &= V_0 e^{z/d} \cos\left(\frac{\pi}{4} + \frac{z}{d}\right), \\
v &= V_0 e^{z/d} \sin\left(\frac{\pi}{4} + \frac{z}{d}\right), \\
V_0 &= \frac{\tau}{\rho\sqrt{2\nu f}}, \quad d = \sqrt{\frac{2\nu}{f}},
\end{aligned}
$$

  where $\tau = 0.1U^2$, ν is mixing coefficient, and $f = 10^{-4}\,\text{s}^{-1}$.

- **Vertical Mixing**:

$$
\begin{aligned}
\frac{dT}{dt} &= K_v \frac{d^2T}{dz^2}, \\
K_v &= \nu e^{z/50},
\end{aligned}
$$

  where $\frac{d^2T}{dz^2}$ is computed via central differences.

- **Implementation**: `update_plot` blends front with ocean temperatures and computes Ekman velocities and temperature profiles.

4.5 Surface Layer Physics

- **Sensible Heat Flux**:

$$
Q_s = \rho_{\text{air}} C_p C_h U (T_o - T_a),
$$

- **Latent Heat Flux**:
$$Q_l = \rho_{\text{air}} L_v C_e U q,$$
where $q = 0.001$ (placeholder), C_e is latent heat coefficient.

- **Wind Stress**:
$$\tau = \rho_{\text{air}} C_d U^2,$$

- **Surface Currents**:
$$u_s^{n+1} = u_s^n + \Delta t \left(\frac{\tau}{1000} \right) \mathcal{N}(0, \nu),$$
where $\mathcal{N}$ is Gaussian noise scaled by mixing coefficient.

- **Implementation**: `update_plot` computes fluxes and updates currents, clipped to $[-0.5, 0.5]$ m/s.

4.6 Air-Sea Interaction

- **Momentum Flux**:
$$\tau = \rho_{\text{air}} C_d U^2,$$

- **Heat Flux**:
$$Q = \rho_{\text{air}} C_p C_h U (T_o - T_a),$$

- **Implementation**: `update_plot` computes fluxes using `TwoWayCoupling`.

4.7 Cloud Microphysics

- **Cloud Formation**:
$$C = \begin{cases} 1 & \text{if } q > q_{\text{sat}}(T_a), \\ 0 & \text{otherwise}, \end{cases}$$
$$q_{\text{sat}}(T) = 0.01 e^{0.06(T - 273.15)},$$

- **Precipitation**:
$$P = \begin{cases} k_p (q - q_{\text{sat}}(T_a)) & \text{if } q > q_{\text{sat}}(T_a), \\ 0 & \text{otherwise}, \end{cases}$$
where $k_p = 0.01$.

- **Implementation**: `update_plot` computes cloud cover and precipitation based on moisture and temperature.

4.8 Potential Vorticity Frontogenesis

- **Potential Vorticity**:

$$q = \frac{(\zeta + f)\frac{db}{dz}}{N^2},$$
$$\frac{db}{dz} \approx \frac{b}{H_m},$$

  where ζ is vorticity, f is Coriolis parameter, b is buoyancy, H_m is mixed layer depth, and N^2 is stratification parameter.

- **Frontogenesis**:

$$F = -|\nabla q|^2 D,$$
$$q^{n+1} = q^n + 0.1F,$$
$$\frac{d\zeta}{dt} = 0.1F,$$

  where D is deformation rate, and gradients are computed via central differences.

- **Implementation**: `compute_pv_frontogenesis` updates PV and returns vorticity tendency, clipped to $[-0.1, 0.1]$.

4.9 Adaptive Mesh Refinement

- **Refinement Criterion**:

$$|\nabla T|^2 = \left(\frac{\partial T}{\partial x}\right)^2 + \left(\frac{\partial T}{\partial y}\right)^2 > \text{threshold},$$
$$\zeta = \frac{\partial v}{\partial x} - \frac{\partial u}{\partial y} > \text{vorticity\_threshold}.$$

- **Nested Grid**: Places a finer grid (size $N/4$) at the region with maximum gradient or vorticity.

- **Implementation**: `refine_grid` computes gradients/vorticity and sets a refinement mask.

4.10 Variable Resolution Grid

- **Spatial Steps**:

$$dx_{i,j} = dy_{i,j} = \begin{cases} \frac{1}{\text{coast\_factor}} & \text{if } i < \frac{N}{4} \text{ or } j < \frac{N}{4}, \\ 1 & \text{otherwise}. \end{cases}$$

- **Implementation**: `VariableResolutionGrid` initializes finer resolution near coasts.

5 Algorithms

5.1 Main Simulation Loop

1. Initialize `OceanAtmosphereModel` with parameters from `ControlPanel`.
2. Start timer to call `step_simulation` every 100 ms.
3. In each step:
   - Call `OceanAtmosphereModel.step` to update fields.
   - Update `PlotWidget` with new temperatures, refinement mask, and nested grid parameters.
   - Log events to `ConsoleWidget`.

5.2 Advection and Diffusion

- **Advection**:

$$\frac{\partial \phi}{\partial t} = -\mathbf{u} \cdot \nabla \phi,$$
$$\frac{\partial \phi}{\partial x} \approx \frac{\phi_{i+1,j} - \phi_{i-1,j}}{2dx_{i,j}},$$

  using central differences for spatial derivatives.

- **Diffusion**:

$$\frac{\partial \phi}{\partial t} = k \nabla^2 \phi,$$
$$\nabla^2 \phi \approx \frac{\phi_{i+1,j} - 2\phi_{i,j} + \phi_{i-1,j}}{dx_{i,j}^2} + \frac{\phi_{i,j+1} - 2\phi_{i,j} + \phi_{i,j-1}}{dy_{i,j}^2},$$

  applying diffusion to stabilize numerical solutions.

5.3 Time Stepping for Boundary Layers

- **Euler**: Single-step update using current rate.
- **RK4**: Computes four intermediate slopes (k_1, k_2, k_3, k_4) and combines them for higher accuracy.

5.4 Turbulent Mixing Simulation

1. Initialize a thermal front and vertical grid (z) for the Ekman spiral.

2. Update temperature field: $T = 0.8T_o + 0.2T_{\text{front}}$.
3. Compute mixing field using `TwoWayCoupling.compute_turbulent_mixing`.
4. Update vertical temperature profile with diffusion.
5. Compute Ekman spiral velocities (u, v).
6. Visualize thermal front and vertical profiles.

5.5 Surface Layer Physics Simulation

1. Compute sensible and latent heat fluxes using input parameters.
2. Compute wind stress and update surface currents with random mixing.
3. Visualize surface currents and time series of mean fluxes.

5.6 Air-Sea Interaction Simulation

1. Compute momentum and heat fluxes using `TwoWayCoupling`.
2. Visualize fluxes as heatmaps.

5.7 Cloud Microphysics Simulation

1. Compute cloud cover based on moisture saturation.
2. Calculate precipitation rate for supersaturated regions.
3. Visualize cloud cover and precipitation.

5.8 Potential Vorticity Frontogenesis

1. Initialize PV field using vorticity and buoyancy.
2. Compute PV gradients and frontogenesis term.
3. Update PV and return vorticity tendency.

Ocean-Atmosphere Coupling Model

June 29, 2025

1 Overview of `Model.py`

The `OceanAtmosphereModel` class in `Model.py` serves as the backbone of the simulator, managing the time evolution of ocean and atmosphere fields. It handles initialization, time stepping, and integration of physical processes such as heat, momentum, freshwater, and CO_2 fluxes, advection, diffusion, and turbulent mixing. It interfaces with `VariableResolution.py` for spatially variable grids, `NestedGrid.py` for nested grid updates, `AdaptiveMeshRefinement.py` for dynamic grid refinement, and `TwoWayCoupling.py` for flux and mixing calculations.

2 Functionalities

The `OceanAtmosphereModel` class provides the following functionalities:

- **Initialization of Fields**: Sets up a 2D grid ($N \times N$) with initial conditions for ocean temperature (T_o), atmosphere temperature (T_a), salinity (S), ocean velocities ($u_{\text{ocean}}, v_{\text{ocean}}$), moisture ($q$), and CO_2 concentrations ($CO_{2,\text{ocean}}, CO_{2,\text{atm}}$).$Configuresvariable$
$Advancesthesimulationusingdistincttimestepsforocean(\Delta t \cdot$ocean_time_scale) and atmosphere ($\Delta t \cdot$ atm_time_scale), updating fields with fluxes, advection, diffusion, and turbulent mixing, while applying numerical stability constraints.

- **Physical Process Integration**: Computes advection and diffusion for temperature, salinity, and CO_2 fields using finite difference methods, integrates fluxes from `TwoWayCoupling`, and applies Coriolis effects via momentum flux.

- **Grid Management**: Uses `VariableResolutionGrid` for spatially variable resolution, supports nested grids via `NestedGrid`, and employs `AdaptiveMeshRefinement` for dynamic grid refinement based on temperature gradients or vorticity.

- **Logging and Error Handling**: Logs initialization and step progress using the `logging` module, with exception handling to catch and log errors.

3 Simulation Logic

The simulation logic in `Model.py` is centered around the `OceanAtmosphereModel` class, which manages the time evolution of the coupled system.

3.1 Initialization

- **Purpose**: Sets up the initial state of ocean and atmosphere fields and configures numerical and physical parameters.
- **Process**:
  - Initializes 2D arrays for T_o (ocean temperature), T_a (atmosphere temperature), S (salinity), $u_{\text{ocean}}, v_{\text{ocean}}$ (ocean velocities), q (moisture), $\text{CO}_{2,\text{ocean}}, and \text{CO}_{2,\text{at}}$ ocean_temp, $T_a =$ atm_temp, $S = 35.0$ psu, $q = 0.01$, $\text{CO}_{2,\text{atm}} = 400.0$ ppm$).Clipsinitialf$ $[250, 350]$ K, $S \in [30, 40]$ psu, $q \in [0, 0.05]$, $\text{CO}_{2,\text{ocean}} \in [0, 10]$, $\text{CO}_{2,\text{atm}} \in [200, 1000]$ ppm.
  - Creates a `VariableResolutionGrid` instance to define spatially variable Δx and Δy (finer near coasts).
  - Optionally initializes a `NestedGrid` for high-resolution subdomains.
  - Sets up an `AdaptiveMeshRefinement` instance with a specified threshold.
  - Configures a `TwoWayCoupling` instance with parameters for drag coefficient, wind speed, precipitation, evaporation, solar forcing, longwave coefficient, mixing coefficient, and CO_2 transfer coefficient.
  - Applies distinct time scales: $\Delta t_{\text{ocean}} = \Delta t \cdot$ ocean_time_scale (default 1.0), $\Delta t_{\text{atm}} = \Delta t \cdot$ atm_time_scale (default 0.1).

3.2 Time Stepping (`step` Method)

- **Purpose**: Advances the simulation by one time step, updating all fields.
- **Process**:
  1. Copies current fields ($T_o, T_a, S, u_{\text{ocean}}, v_{\text{ocean}}, q, \text{CO}_{2,\text{ocean}}, \text{CO}_{2,\text{atm}}$) to avoid in-place modifications.
  2. Computes a refinement mask using `AdaptiveMeshRefinement.compute_refiner` based on temperature gradients or vorticity.
  3. Updates fields on the nested grid (if enabled) using `NestedGrid.update`.

4. Defines time-varying atmosphere velocities:

$$u_{\text{atm}} = \text{adv\_velocity} \cdot \cos\left(\frac{2\pi \cdot \text{step} \cdot \Delta t}{\text{total\_time}}\right),$$
$$v_{\text{atm}} = \text{adv\_velocity} \cdot \sin\left(\frac{2\pi \cdot \text{step} \cdot \Delta t}{\text{total\_time}}\right).$$

5. Computes fluxes and mixing using `TwoWayCoupling`:
   - Heat flux (Q) using $T_a, T_o, u_{\text{ocean}}, v_{\text{ocean}}$.
   - Radiative flux ($R_{\text{ocean}}, R_{\text{atm}}$) using T_o, T_a, and $\text{CO}_{2,\text{atm}}$.$Freshwaterflux($d S$_{dt,F_{\text{freshwater}}}$) using S and q.
   - Momentum flux (τ) using wind speed, $u_{\text{ocean}}, v_{\text{ocean}}$.
   - Moisture advection (M_{adv}) using $q, \Delta x, \Delta y, u_{\text{atm}}, v_{\text{atm}}$.
   - Turbulent mixing ($\text{mix}_{\text{ocean}}, \text{mix}_{\text{atm}}$) using $T_o, T_a, S, \Delta x, \Delta y$, wind speed.
   - CO_2 flux ($F_{\text{CO}_2,\text{ocean}}, F_{\text{CO}_2,\text{atm}}$) using $\text{CO}_{2,\text{ocean}}, \text{CO}_{2,\text{atm}}$.
6. Updates ocean velocities:

$$u_{\text{ocean}}^{n+1} = u_{\text{ocean}}^{n} + \Delta t_{\text{ocean}} \cdot \frac{\tau}{\rho_{\text{water}}},$$
$$v_{\text{ocean}}^{n+1} = v_{\text{ocean}}^{n} + \Delta t_{\text{ocean}} \cdot \frac{\tau}{\rho_{\text{water}}}.$$

7. Computes advection for $T_o, T_a, S, \text{CO}_{2,\text{ocean}}, \text{CO}_{2,\text{atm}}$ using `compute_advection`.
8. Computes diffusion for T_o, T_a, S using `compute_diffusion`.
9. Updates fields using a semi-implicit scheme ($\alpha = 0.5$):
   - Ocean temperature, atmosphere temperature, salinity, moisture, and CO_2 concentrations (detailed in Section 5).
10. Clips updated fields to physical ranges.
11. Applies AMR to refine T_o, T_a, S where the refinement mask is active.
12. Returns current time, updated T_o, T_a, and refinement mask.

4 Physics and Mathematical Models

The `OceanAtmosphereModel` integrates physical processes through numerical updates, relying on `TwoWayCoupling` for flux calculations. Below are the key models and equations implemented in `Model.py`, formatted to fit within margins.

4.1 Ocean Temperature Update

- **Purpose**: Updates ocean temperature (T_o) based on heat flux, radiative flux, advection, diffusion, and turbulent mixing.
- **Equation**:

$$T_o^{n+1} = T_o^n + \Delta t_{\text{ocean}} \cdot \left[\alpha \cdot \left(\frac{Q}{C_o} + \frac{R_{\text{ocean}}}{C_o} - \text{adv}_{\text{ocean}} + \text{diff}_{\text{ocean}} + \frac{\text{mix}_{\text{ocean}}}{C_o} \right) + (1-\alpha) \cdot \left(\frac{Q}{C_o} + \frac{R_{\text{ocean}}}{C_o} \right) \right],$$

  where:
  - Q is heat flux from `compute_heat_flux` ($\mathrm{W/m^2}$).
  - R_{ocean} is radiative flux from `compute_radiative_flux` ($\mathrm{W/m^2}$).
  - $\text{adv}_{\text{ocean}}$ is advection from `compute_advection` ($\mathrm{K/s}$).
  - $\text{diff}_{\text{ocean}}$ is diffusion from `compute_diffusion` ($\mathrm{K/s}$).
  - $\text{mix}_{\text{ocean}}$ is turbulent mixing from `compute_turbulent_mixing` ($\mathrm{W/m^2}$).
  - C_o is ocean heat capacity ($\mathrm{J/kg/K}$).
  - $\alpha = 0.5$ is the semi-implicit factor.
  - $\Delta t_{\text{ocean}} = \Delta t \cdot \text{ocean\_time\_scale}$ (s).
- **Implementation**: Clips terms to $[-10^3, 10^3]$ and T_o to $[250, 350]$ K.

4.2 Atmosphere Temperature Update

- **Purpose**: Updates atmosphere temperature (T_a) based on heat flux, radiative flux, advection, diffusion, and turbulent mixing.
- **Equation**:

$$T_a^{n+1} = T_a^n + \Delta t_{\text{atm}} \cdot \left[\alpha \cdot \left(-\frac{Q}{C_a} + \frac{R_{\text{atm}}}{C_a} - \text{adv}_{\text{atm}} + \text{diff}_{\text{atm}} + \frac{\text{mix}_{\text{atm}}}{C_a} \right) + (1-\alpha) \cdot \left(-\frac{Q}{C_a} + \frac{R_{\text{atm}}}{C_a} \right) \right],$$

  where:
  - Q is heat flux (negative for atmosphere due to coupling).
  - R_{atm} is radiative flux.

- adv_{atm} is advection.
- diff_{atm} is diffusion.
- mix_{atm} is turbulent mixing.
- C_a is atmosphere heat capacity (J/kg/K).
- $\Delta t_{\text{atm}} = \Delta t \cdot \text{atm\_time\_scale}$ (s).

- **Implementation**: Clips terms to $[-10^3, 10^3]$ and T_a to $[250, 350]$ K.

4.3 Salinity Update

- **Purpose**: Updates salinity (S) based on freshwater flux, diffusion, and turbulent mixing.
- **Equation**:

$$S^{n+1} = S^n + \Delta t \cdot \left(\frac{dS}{dt} + \text{diff}_{\text{salinity}} + \text{mix}_{\text{ocean}} \right),$$

where:

- $\frac{dS}{dt}$ is salinity change from `compute_freshwater_flux`.
- $\text{diff}_{\text{salinity}}$ is diffusion.
- $\text{mix}_{\text{ocean}}$ is turbulent mixing.

- **Implementation**: Clips terms to $[-10^{-2}, 10^{-2}]$ and S to $[30, 40]$ psu.

4.4 Moisture Update

- **Purpose**: Updates atmospheric moisture (q) based on advection and freshwater flux.
- **Equation**:

$$q^{n+1} = q^n + \Delta t \cdot (M_{\text{adv}} - F_{\text{freshwater}}),$$

where:

- M_{adv} is moisture advection from `compute_moisture_advection`.
- $F_{\text{freshwater}}$ is freshwater flux from `compute_freshwater_flux`.

- **Implementation**: Clips terms to $[-10^{-4}, 10^{-4}]$ and q to $[0, 0.05]$.

4.5 CO_2 Concentration Updates

- **Purpose**: Updates ocean and atmosphere CO_2 concentrations ($CO_{2,\text{ocean}}, CO_{2,\text{atm}}) basedonC$ flux and advection.

- **Equations**:

$$\begin{aligned}\mathrm{CO}_{2,\mathrm{ocean}}^{n+1} &= \mathrm{CO}_{2,\mathrm{ocean}}^{n} + \Delta t \cdot \left(F_{\mathrm{CO}_2,\mathrm{ocean}} + \mathrm{adv}_{\mathrm{co2\_o}}\right), \\ \mathrm{CO}_{2,\mathrm{atm}}^{n+1} &= \mathrm{CO}_{2,\mathrm{atm}}^{n} + \Delta t \cdot \left(F_{\mathrm{CO}_2,\mathrm{atm}} + \mathrm{adv}_{\mathrm{co2\_a}}\right),\end{aligned}$$

  where:

  - $F_{\mathrm{CO}_2,\mathrm{ocean}}, F_{\mathrm{CO}_2,\mathrm{atm}}$ are CO_2 fluxes from `compute_co2_flux`.
  - $\mathrm{adv}_{\mathrm{co2\_o}}, \mathrm{adv}_{\mathrm{co2\_a}}$ are advection terms from `compute_advection`.

- **Implementation**: Clips terms to $[-10^{-2}, 10^{-2}]$, $\mathrm{CO}_{2,\mathrm{ocean}} to$[0, 10], $and \mathrm{CO}_{2,\mathrm{atm}} to$[200, 1000] ppm.

4.6 Ocean Velocity Update

- **Purpose**: Updates ocean velocities ($u_{\mathrm{ocean}}, v_{\mathrm{ocean}}$) based on momentum flux.
- **Equations**:

$$\begin{aligned}u_{\mathrm{ocean}}^{n+1} &= u_{\mathrm{ocean}}^{n} + \Delta t_{\mathrm{ocean}} \cdot \frac{\tau}{\rho_{\mathrm{water}}}, \\ v_{\mathrm{ocean}}^{n+1} &= v_{\mathrm{ocean}}^{n} + \Delta t_{\mathrm{ocean}} \cdot \frac{\tau}{\rho_{\mathrm{water}}},\end{aligned}$$

  where:

  - τ is momentum flux from `compute_momentum_flux` ($\mathrm{N/m^2}$).
  - $\rho_{\mathrm{water}} = 1025\,\mathrm{kg/m^3}$ (from `TwoWayCoupling`).

- **Implementation**: Clips velocities to $[-10, 10]$ m/s.

4.7 Advection

- **Purpose**: Computes advection for any field (T, S, CO_2).
- **Equations**:

$$\begin{aligned}\frac{\partial T}{\partial t} &= -u \cdot \frac{\partial T}{\partial x} - v \cdot \frac{\partial T}{\partial y}, \\ \frac{\partial T}{\partial x} &\approx \frac{T_{i+1,j} - T_{i-1,j}}{2\Delta x_{i,j}}, \\ \frac{\partial T}{\partial y} &\approx \frac{T_{i,j+1} - T_{i,j-1}}{2\Delta y_{i,j}}.\end{aligned}$$

- **Implementation**: Uses central differences, supports array-based or scalar velocities, and clips output to $[-10^5, 10^5]$.

4.8 Diffusion

- **Purpose**: Computes diffusion for temperature or salinity fields.
- **Equations**:

$$\frac{\partial T}{\partial t} = D \cdot \nabla^2 T,$$
$$\nabla^2 T \approx \frac{T_{i+1,j} - 2T_{i,j} + T_{i-1,j}}{\Delta x_{i,j}^2} + \frac{T_{i,j+1} - 2T_{i,j} + T_{i,j-1}}{\Delta y_{i,j}^2},$$

  where $D = 10^{-6}\,\mathrm{m}^2/\mathrm{s}$.

- **Implementation**: Uses central differences and clips output to $[-10^5, 10^5]$.

5 Algorithms

5.1 Initialization Algorithm

- **Steps**:
  1. Log initialization parameters.
  2. Store input parameters as instance variables.
  3. Compute $\Delta t_{\text{ocean}} = \Delta t \cdot$ocean_time_scale and $\Delta t_{\text{atm}} = \Delta t \cdot$atm_time_scale.
  4. Initialize `TwoWayCoupling` with flux parameters.
  5. Create `VariableResolutionGrid` and get $\Delta x, \Delta y$.
  6. If `use_nested_grid`, initialize `NestedGrid`.
  7. Initialize `AdaptiveMeshRefinement` with `amr_threshold`.
  8. Set up 2D arrays:
     - `ocean_temps` = `ocean_temp`, clipped to $[250, 350]$ K.
     - `atm_temps` = `atm_temp`, clipped to $[250, 350]$ K.
     - `salinity` = 35.0 psu.
     - `u_ocean`, `v_ocean` = 0.
     - `moisture` = 0.01.
     - `co2_ocean` = 2.0, `co2_atm` = 400.0 ppm.
  9. Log completion.

5.2 Time Stepping Algorithm (`step`)

- **Input**: Step number (`step`).
- **Steps**:
  1. Copy current fields to avoid in-place modification.
  2. Compute `refinement_mask` using `AdaptiveMeshRefinement.compute_refinem`
  3. If `nested_grid` exists, update T_o, T_a using `NestedGrid.update`.
  4. Compute atmosphere velocities:
$$u_{\text{atm}} = \text{adv\_velocity} \cdot \cos\left(\frac{2\pi \cdot \text{step} \cdot \Delta t}{\text{total\_time}}\right),$$
$$v_{\text{atm}} = \text{adv\_velocity} \cdot \sin\left(\frac{2\pi \cdot \text{step} \cdot \Delta t}{\text{total\_time}}\right).$$
  5. Compute fluxes and mixing using `TwoWayCoupling` methods.
  6. Update velocities: $u_{\text{new}}, v_{\text{new}}$ using momentum flux.
  7. Compute advection for $T_o, T_a, S, \text{CO}_{2,\text{ocean}}, \text{CO}_{2,\text{atm}}$.
  8. Compute diffusion for T_o, T_a, S.
  9. Update fields with semi-implicit scheme ($\alpha = 0.5$).
  10. Clip updated fields to physical ranges.
  11. Apply AMR refinement where `refinement_mask` is True.
  12. Log step completion and return time, `ocean_temps`, `atm_temps`, `refinement_mask`
  13. Handle exceptions and log errors if they occur.

5.3 Advection Algorithm (`compute_advection`)

- **Input**: Field T, Δx, Δy, `step`, optional velocities u, v.
- **Steps**:
  1. Initialize advection array (zeros, same shape as T).
  2. If u, v not provided, use:
$$u_{\text{vel}} = \text{adv\_velocity} \cdot \cos\left(\frac{2\pi \cdot \text{step} \cdot \Delta t}{\text{total\_time}}\right),$$
$$v_{\text{vel}} = \text{adv\_velocity} \cdot \sin\left(\frac{2\pi \cdot \text{step} \cdot \Delta t}{\text{total\_time}}\right).$$

3. For each grid point (i, j):
   - Compute u_{ij}, v_{ij} (from $u_{\text{vel}}, v_{\text{vel}}$, handling scalar or array inputs).
   - $\text{adv}_x = -u_{ij} \cdot \frac{T_{i+1,j} - T_{i-1,j}}{2\Delta x_{i,j}}$.
   - $\text{adv}_y = -v_{ij} \cdot \frac{T_{i,j+1} - T_{i,j-1}}{2\Delta y_{i,j}}$.
   - $\text{advection}[i, j] = \text{adv}_x + \text{adv}_y$, clipped to $[-10^5, 10^5]$.
4. Return advection array.
5. Log errors if computation fails.

5.4 Diffusion Algorithm (`compute_diffusion`)

- **Input**: Field T, Δx, Δy.
- **Steps**:
  1. Initialize diffusion array (zeros, same shape as T).
  2. Set $D = 10^{-6}\,\text{m}^2/\text{s}$.
  3. For each grid point (i, j):
     - $\text{diff}_x = \frac{T_{i+1,j} - 2T_{i,j} + T_{i-1,j}}{\Delta x_{i,j}^2}$.
     - $\text{diff}_y = \frac{T_{i,j+1} - 2T_{i,j} + T_{i,j-1}}{\Delta y_{i,j}^2}$.
     - $\text{diffusion}[i, j] = D \cdot (\text{diff}_x + \text{diff}_y)$, clipped to $[-10^5, 10^5]$.
  4. Return diffusion array.
  5. Log errors if computation fails.

Two-Way Coupling

June 29, 2025

1 Functionalities

The `TwoWayCoupling` class provides the following functionalities:

- **Initialization**: Configures physical parameters such as drag coefficient, wind speed, precipitation rate, evaporation rate, solar forcing, longwave coefficient, mixing coefficient, CO_2 transfer coefficient, and conservation coefficients for freshwater and CO_2.
- **Sea Surface Roughness**: Computes an adjusted drag coefficient based on wind speed and ocean currents, accounting for sea surface roughness.
- **Momentum Flux**: Calculates wind stress on the ocean surface using the adjusted drag coefficient.
- **Heat Flux**: Computes sensible and latent heat fluxes across the air-sea interface, driven by temperature differences and evaporation.
- **Freshwater Flux**: Models precipitation and evaporation to compute net freshwater flux and salinity change, ensuring mass conservation.
- **Moisture Advection**: Calculates moisture transport in the atmosphere using finite difference methods.
- **Turbulent Mixing**: Models mixing driven by temperature gradients and wind speed, applied to ocean and atmosphere fields.
- **CO_2 Flux**: Computes CO_2 exchange between ocean and atmosphere, ensuring mass conservation.
- **Radiative Flux**: Models solar and longwave radiation, incorporating greenhouse effects from CO_2 concentrations.
- **Logging and Error Handling**: Logs initialization and computation progress using the `logging` module, with exception handling to catch and log errors.

2 Simulation Logic

The simulation logic in `TwoWayCoupling.py` revolves around the `TwoWayCoupling` class, which provides methods to compute fluxes and mixing terms used in the time stepping of `Model.py`.

2.1 Initialization

- **Purpose**: Sets up physical parameters and constants for flux and mixing calculations.
- **Process**:
  - Initializes parameters: `drag_coeff`, `wind_speed`, `precip_rate`, `evap_rate`, `solar_forcing`, `longwave_coeff`, `mixing_coeff`, `co2_transfer_coeff`, `freshwater_conservation_coeff` (default 1.0), `co2_conservation_coeff` (default 1.0).
  - Sets physical constants: $\rho_{\text{air}} = 1.225\,\text{kg/m}^3$, $\rho_{\text{water}} = 1025\,\text{kg/m}^3$, $C_p^{\text{air}} = 1005\,\text{J/kg/K}$, $L_v = 2.5 \times 10^6\,\text{J/kg}$, $\alpha = 0.03$ (CO_2 solubility).
  - Logs initialization details and handles exceptions.

2.2 Flux and Mixing Computations

- **Purpose**: Computes fluxes and mixing terms for use in `Model.py`'s time stepping.
- **Process**: Each method (`compute_sea_surface_roughness`, `compute_momentum_flux` etc.) performs the following:
  1. Logs the start of computation.
  2. Applies the relevant physical equations (detailed in Section 4).
  3. Clips outputs to predefined ranges to ensure numerical stability.
  4. Returns the computed values for use in `Model.py`.
  5. Handles exceptions and logs errors if computations fail.

3 Physics and Mathematical Models

The `TwoWayCoupling` class implements physical models for ocean-atmosphere interactions, with equations formatted to fit within margins.

3.1 Sea Surface Roughness

- **Purpose**: Adjusts the drag coefficient based on wind and ocean currents to account for sea surface roughness.
- **Equations**:

$$
\begin{aligned}
u_* &= \sqrt{\frac{\rho_{\text{air}} C_d U^2}{\rho_{\text{water}}}}, \\
z_0 &= \alpha \frac{u_*^2 + 0.1(u_{\text{ocean}}^2 + v_{\text{ocean}}^2)}{g}, \\
C_d' &= C_d \left(1 + 0.1 \log_{10}(z_0)\right),
\end{aligned}
$$

  where u_* is friction velocity, C_d is the drag coefficient, U is wind speed, $\rho_{\text{air}} = 1.225\,\text{kg/m}^3$, $\rho_{\text{water}} = 1025\,\text{kg/m}^3$, $\alpha = 0.018$, $g = 9.81\,\text{m/s}^2$, and z_0 is roughness length.

- **Implementation**: `compute_sea_surface_roughness` clips z_0 to $[10^{-6}, 10^{-2}]$ and C_d' to $[10^{-4}, 10^{-2}]$. Ocean speed is computed as $\sqrt{u_{\text{ocean}}^2 + v_{\text{ocean}}^2}$.

3.2 Momentum Flux

- **Purpose**: Computes wind stress on the ocean surface.
- **Equation**:

$$\tau = \rho_{\text{air}} C_d' U^2,$$

  where C_d' is the adjusted drag coefficient from `compute_sea_surface_roughness`.

- **Implementation**: `compute_momentum_flux` clips U^2 to $[0, 10^3]$ and τ to $[-10^5, 10^5]\,\text{N/m}^2$.

3.3 Heat Flux

- **Purpose**: Computes sensible and latent heat fluxes across the air-sea interface.
- **Equations**:

$$
\begin{aligned}
k_{\text{sensible}} &= 0.01 \frac{C_d'}{C_d}, \\
Q_{\text{sensible}} &= \rho_{\text{air}} C_p^{\text{air}} k_{\text{sensible}} (T_a - T_o), \\
Q_{\text{latent}} &= \rho_{\text{air}} L_v E \text{sign}(T_a - T_o), \\
Q_{\text{total}} &= Q_{\text{sensible}} + Q_{\text{latent}},
\end{aligned}
$$

  where $C_p^{\text{air}} = 1005\,\text{J/kg/K}$, $L_v = 2.5 \times 10^6\,\text{J/kg}$, E is evaporation rate, T_a is atmosphere temperature, and T_o is ocean temperature.

- **Implementation**: `compute_heat_flux` clips $T_a - T_o$ to $[-100, 100]\,\text{K}$ and Q_{total} to $[-10^6, 10^6]\,\text{W/m}^2$.

3.4 Freshwater Flux

- **Purpose**: Computes net freshwater flux and salinity change, ensuring mass conservation.
- **Equations**:

$$\begin{aligned} P &= P_0 \left(1 + 0.5 \frac{q}{0.01}\right), \\ E &= E_0 C_{\text{freshwater}} \left(1 + 0.1 \frac{S}{35}\right), \\ F &= P - E, \\ \frac{dS}{dt} &= -\frac{SF}{\rho_{\text{water}} H}, \end{aligned}$$

  where P_0 is precipitation rate, E_0 is evaporation rate, $C_{\text{freshwater}}$ is the freshwater conservation coefficient, S is salinity, q is moisture, $\rho_{\text{water}} = 1025\,\text{kg/m}^3$, and $H = 1000\,\text{m}$ is ocean depth.

- **Implementation**: `compute_freshwater_flux` clips $q/0.01$ to $[0, 2]$, $S/35$ to $[0.8, 1.2]$, and outputs $\frac{dS}{dt}$ and F to $[-10^{-3}, 10^{-3}]$.

3.5 Moisture Advection

- **Purpose**: Computes moisture transport in the atmosphere.
- **Equation**:

$$\text{Advection} = -u_{\text{atm}} \frac{\partial q}{\partial x} - v_{\text{atm}} \frac{\partial q}{\partial y},$$

  where:

$$\begin{aligned} \frac{\partial q}{\partial x} &\approx \frac{q_{i+1,j} - q_{i-1,j}}{2\Delta x_{i,j}}, \\ \frac{\partial q}{\partial y} &\approx \frac{q_{i,j+1} - q_{i,j-1}}{2\Delta y_{i,j}}, \end{aligned}$$

  and $u_{\text{atm}}, v_{\text{atm}}$ are atmosphere velocities.

- **Implementation**: `compute_moisture_advection` uses central differences, handles scalar or array velocities, and clips output to $[-10^{-4}, 10^{-4}]$.

3.6 Turbulent Mixing

- **Purpose**: Models mixing driven by temperature gradients and wind speed.

- **Equations**:

$$
\begin{aligned}
u_* &= \sqrt{\frac{\rho_{\text{air}} C_d U^2}{\rho_{\text{water}}}}, \\
M &= k_m u_* \left(\frac{\partial T}{\partial x} + \frac{\partial T}{\partial y} \right), \\
\frac{\partial T}{\partial x} &\approx \frac{T_{i+1,j} - T_{i-1,j}}{2\Delta x_{i,j}}, \\
\frac{\partial T}{\partial y} &\approx \frac{T_{i,j+1} - T_{i,j-1}}{2\Delta y_{i,j}},
\end{aligned}
$$

  where k_m is the mixing coefficient, and C_d is the drag coefficient.

- **Implementation**: `compute_turbulent_mixing` clips gradients to $[-10^3, 10^3]$ and output to $[-10^3, 10^3]$.

3.7 CO_2 Flux

- **Purpose**: Computes CO_2 exchange between ocean and atmosphere, ensuring mass conservation.

- **Equations**:

$$
\begin{aligned}
p\text{CO}_{2,\text{ocean}} &= \frac{\text{CO}_{2,\text{ocean}}}{\alpha}, \\
p\text{CO}_{2,\text{atm}} &= \text{CO}_{2,\text{atm}}, \\
F_{\text{CO}_2} &= k_{\text{CO}_2} C_{\text{CO}_2} (p\text{CO}_{2,\text{ocean}} - p\text{CO}_{2,\text{atm}}), \\
F_{\text{CO}_2,\text{ocean}} &= \frac{F_{\text{CO}_2}}{H}, \\
F_{\text{CO}_2,\text{atm}} &= -\frac{F_{\text{CO}_2}}{H_{\text{atm}}},
\end{aligned}
$$

  where $\alpha = 0.03$ is CO_2 solubility, k_{CO_2} is the transfer coefficient, C_{CO_2} is the CO_2 conservation coefficient, $H = 1000\,\text{m}$ is ocean depth, and $H_{\text{atm}} = 10000\,\text{m}$ is atmosphere height.

- **Implementation**: `compute_co2_flux` clips outputs to $[-10^{-3}, 10^{-3}]$.

3.8 Radiative Flux

- **Purpose**: Models solar and longwave radiation with greenhouse effects.

- **Equation**:

$$
\begin{aligned}
Q_{\text{rad}} = Q_{\text{solar}} &- \varepsilon\sigma \left(\frac{T}{300} \right)^4 \\
&\times \left(1 + 0.1 \log \left(\frac{\text{CO}_{2,\text{atm}}}{400} \right) \right),
\end{aligned}
$$

where $\sigma = 5.67 \times 10^{-8}\,\text{W/m}^2/\text{K}^4$, ε is the longwave coefficient, and Q_{solar} is solar forcing.

- **Implementation**: `compute_radiative_flux` clips $T/300$ to $[0.8, 1.2]$, $\log(\text{CO}_{2,\text{atm}}/400)$ to $[-1, 1]$, and output to $[-10^6, 10^6]\,\text{W/m}^2$.

4 Algorithms

4.1 Initialization Algorithm

- **Input**: Parameters (`drag_coeff`, `wind_speed`, `precip_rate`, `evap_rate`, `solar_forcing`, `longwave_coeff`, `mixing_coeff`, `co2_transfer_coeff`, `freshwater_conservation_coeff`, `co2_conservation_coeff`).
- **Steps**:
  1. Log initialization parameters.
  2. Store input parameters as instance variables.
  3. Set physical constants: $\rho_{\text{air}} = 1.225\,\text{kg/m}^3$, $\rho_{\text{water}} = 1025\,\text{kg/m}^3$, $C_p^{\text{air}} = 1005\,\text{J/kg/K}$, $L_v = 2.5 \times 10^6\,\text{J/kg}$, $\alpha = 0.03$.
  4. Log completion.
  5. Handle exceptions and log errors if initialization fails.

4.2 Sea Surface Roughness Algorithm (`compute_sea_surface_roughnes`

- **Input**: `wind_speed`, u_{ocean}, v_{ocean}.
- **Steps**:
  1. Log start of computation.
  2. Set $g = 9.81\,\text{m/s}^2$, $\alpha = 0.018$.
  3. Compute friction velocity: $u_* = \sqrt{\frac{\rho_{\text{air}} C_d U^2}{\rho_{\text{water}}}}$.
  4. Compute ocean speed: $\sqrt{u_{\text{ocean}}^2 + v_{\text{ocean}}^2}$.
  5. Compute roughness length: $z_0 = \alpha \frac{u_*^2 + 0.1(u_{\text{ocean}}^2 + v_{\text{ocean}}^2)}{g}$, clipped to $[10^{-6}, 10^{-2}]$.
  6. Compute adjusted drag coefficient: $C_d' = C_d(1 + 0.1\log_{10}(z_0))$, clipped to $[10^{-4}, 10^{-2}]$.
  7. Return C_d'.

8. Log errors if computation fails.

4.3 Momentum Flux Algorithm (`compute_momentum_flux`)

- **Input**: `wind_speed`, u_{ocean}, v_{ocean}.
- **Steps**:
  1. Log start of computation.
  2. Compute C_d' using `compute_sea_surface_roughness`.
  3. Compute wind stress: $\tau = \rho_{\text{air}} C_d' U^2$, with U^2 clipped to $[0, 10^3]$.
  4. Clip τ to $[-10^5, 10^5]$ N/m^2.
  5. Return τ.
  6. Log errors if computation fails.

4.4 Heat Flux Algorithm (`compute_heat_flux`)

- **Input**: T_a, T_o, u_{ocean}, v_{ocean}.
- **Steps**:
  1. Log start of computation.
  2. Compute C_d' using `compute_sea_surface_roughness`.
  3. Compute sensible heat coefficient: $k_{\text{sensible}} = 0.01 \frac{C_d'}{C_d}$.
  4. Compute sensible heat flux: $Q_{\text{sensible}} = \rho_{\text{air}} C_p^{\text{air}} k_{\text{sensible}} (T_a - T_o)$, with $T_a - T_o$ clipped to $[-100, 100]$ K.
  5. Compute latent heat flux: $Q_{\text{latent}} = \rho_{\text{air}} L_v E \text{sign}(T_a - T_o)$.
  6. Compute total heat flux: $Q_{\text{total}} = Q_{\text{sensible}} + Q_{\text{latent}}$, clipped to $[-10^6, 10^6]$ W/m^2.
  7. Return Q_{total}.
  8. Log errors if computation fails.

4.5 Freshwater Flux Algorithm (`compute_freshwater_flux`)

- **Input**: S, q, `ocean_depth` (default 1000 m).
- **Steps**:
  1. Log start of computation.

2. Compute precipitation: $P = P_0(1 + 0.5\frac{q}{0.01})$, with $q/0.01$ clipped to $[0, 2]$.
3. Compute evaporation: $E = E_0 C_{\text{freshwater}}(1 + 0.1\frac{S}{35})$, with $S/35$ clipped to $[0.8, 1.2]$.
4. Compute net freshwater flux: $F = P - E$, clipped to $[-10^{-3}, 10^{-3}]$.
5. Compute salinity change: $\frac{dS}{dt} = -\frac{SF}{\rho_{\text{water}} H}$, clipped to $[-10^{-3}, 10^{-3}]$.
6. Return $\frac{dS}{dt}$, F.
7. Log errors if computation fails.

4.6 Moisture Advection Algorithm (`compute_moisture_advection`)

- **Input**: q, Δx, Δy, `step`, u_{atm}, v_{atm}.
- **Steps**:
  1. Log start of computation.
  2. Initialize advection array (zeros, same shape as q).
  3. For each grid point (i, j):
     - Compute u_{ij}, v_{ij} (from $u_{\text{atm}}, v_{\text{atm}}$, handling scalar or array inputs).
     - $\text{adv}_x = -u_{ij}\frac{q_{i+1,j} - q_{i-1,j}}{2\Delta x_{i,j}}$.
     - $\text{adv}_y = -v_{ij}\frac{q_{i,j+1} - q_{i,j-1}}{2\Delta y_{i,j}}$.
     - $\text{advection}[i, j] = \text{adv}_x + \text{adv}_y$, clipped to $[-10^{-4}, 10^{-4}]$.
  4. Return advection array.
  5. Log errors if computation fails.

4.7 Turbulent Mixing Algorithm (`compute_turbulent_mixing`)

- **Input**: T, S, Δx, Δy, `wind_speed`.
- **Steps**:
  1. Log start of computation.
  2. Compute friction velocity: $u_* = \sqrt{\frac{\rho_{\text{air}} C_d U^2}{\rho_{\text{water}}}}$.
  3. Initialize mixing array (zeros, same shape as T).

4. For each grid point (i, j):
   - $\text{grad}_x = \frac{T_{i+1,j} - T_{i-1,j}}{2\Delta x_{i,j}}$.
   - $\text{grad}_y = \frac{T_{i,j+1} - T_{i,j-1}}{2\Delta y_{i,j}}$.
   - $\text{mixing}[i, j] = k_m u_* (\text{grad}_x + \text{grad}_y)$, with gradients clipped to $[-10^3, 10^3]$.
5. Clip mixing to $[-10^3, 10^3]$.
6. Return mixing array.
7. Log errors if computation fails.

4.8 CO_2 Flux Algorithm (`compute_co2_flux`)

- **Input**: $\text{CO}_{2,\text{ocean}}, \text{CO}_{2,\text{atm}}, \texttt{ocean\_depth}(default 1000m), \texttt{atm\_height}(default 10000m)$.**Steps**

- - Log start of computation.
  - Compute partial pressures: $p\text{CO}_{2,\text{ocean}} = \frac{\text{CO}_{2,\text{ocean}}}{\alpha}$, $p\text{CO}_{2,\text{atm}} = \text{CO}_{2,\text{atm}}$.
  - Compute flux: $F_{\text{CO}_2} = k_{\text{CO}_2} C_{\text{CO}_2} (p\text{CO}_{2,\text{ocean}} - p\text{CO}_{2,\text{atm}})$.
  - Normalize fluxes: $F_{\text{CO}_2,\text{ocean}} = \frac{F_{\text{CO}_2}}{H}$, $F_{\text{CO}_2,\text{atm}} = -\frac{F_{\text{CO}_2}}{H_{\text{atm}}}$, clipped to $[-10^{-3}, 10^{-3}]$.
  - Return $F_{\text{CO}_2,\text{ocean}}$, $F_{\text{CO}_2,\text{atm}}$.
  - Log errors if computation fails.

4.9 Radiative Flux Algorithm (`compute_radiative_flux`)

- **Input**: T, $\text{CO}_{2,\text{atm}}$.**Steps** :

- - Log start of computation.
  - Set $\sigma = 5.67 \times 10^{-8}\ \text{W/m}^2/\text{K}^4$.
  - Compute normalized temperature: $T_{\text{norm}} = \frac{T}{300}$, clipped to $[0.8, 1.2]$.
  - Compute longwave radiation: $\text{longwave} = \varepsilon\sigma(T_{\text{norm}} \cdot 300)^4$.
  - Compute greenhouse effect: $\text{greenhouse} = 0.1 \log\left(\frac{\text{CO}_{2,\text{atm}}}{400}\right)$, clipped to $[-1, 1]$.
  - Compute radiative flux: $Q_{\text{rad}} = Q_{\text{solar}} - \text{longwave}(1 + \text{greenhouse})$, clipped to $[-10^6, 10^6]\ \text{W/m}^2$.

- Return Q_{rad}.
- Log errors if computation fails.

Variable Resolution

June 29, 2025

1 Functionalities

The `VariableResolutionGrid` class provides the following functionalities:

- **Grid Initialization**: Creates a 2D grid of size $N \times N$ with spatially variable resolution, adjusting Δx and Δy based on a coastal factor to refine resolution near coasts.
- **Spatial Step Retrieval**: Provides access to the spatial step arrays (Δx, Δy) for use in numerical computations in `Model.py`.
- **Coastal Refinement**: Simulates coastal regions by reducing spatial steps in a designated coastal zone, enhancing resolution where ocean-atmosphere interactions are more complex.
- **Logging and Error Handling**: Logs initialization and retrieval operations using the `logging` module, with exception handling to catch and log errors.

2 Simulation Logic

The simulation logic in `VariableResolution.py` centers around the `VariableResolutionG` class, which sets up and manages a variable resolution grid for the simulator.

2.1 Initialization

- **Purpose**: Initializes a 2D grid with spatially variable resolution, refining spatial steps near coastal regions.
- **Process**:
  - Initializes parameters: `grid_size` (N) and `coast_factor`, which determines the degree of refinement near coasts.
  - Creates 2D arrays Δx and Δy, initially set to 1.0 across the $N \times N$ grid.

- Defines a coastal region as the first $N/4$ grid points in both x and y directions.
- Reduces Δx and Δy by dividing by `coast_factor` in the coastal region to achieve finer resolution.
- Logs initialization details and handles exceptions.

2.2 Spatial Step Retrieval

- **Purpose**: Provides access to the spatial step arrays (Δx, Δy) for use in numerical computations.
- **Process**:
  - Returns the Δx and Δy arrays as a tuple.
  - Logs the retrieval operation and handles exceptions.

3 Algorithms

3.1 Initialization Algorithm

- **Input**: `grid_size` (N), `coast_factor`.
- **Steps**:
  1. Log initialization parameters: `grid_size`, `coast_factor`.
  2. Store `grid_size` and `coast_factor` as instance variables.
  3. Initialize Δx and Δy as $N \times N$ arrays filled with 1.0.
  4. Compute coastal width: coast_width $= \lfloor N/4 \rfloor$.
  5. For each grid point (i, j):
     - If $i <$ coast_width or $j <$ coast_width:
       - Set $\Delta x_{i,j} = \Delta x_{i,j}/$coast_factor.
       - Set $\Delta y_{i,j} = \Delta y_{i,j}/$coast_factor.
  6. Log completion of initialization.
  7. Handle exceptions and log errors if initialization fails.

3.2 Spatial Step Retrieval Algorithm (`get_spatial_steps`)

- **Input**: None.
- **Steps**:
  1. Log start of spatial step retrieval.
  2. Return the tuple $(\Delta x, \Delta y)$.
  3. Log errors if retrieval fails.

Adaptive Mesh Refinement

June 29, 2025

1 Functionalities

The `AdaptiveMeshRefinement` class provides the following functionalities:

- **Initialization**: Sets up the adaptive mesh refinement system with a specified grid size and refinement threshold, maintaining a list of cells targeted for refinement.
- **Refinement Computation**: Identifies grid cells requiring refinement based on temperature gradients and vorticity of ocean and atmosphere temperature fields.
- **Grid Refinement**: Refines the resolution of specified grid cells by interpolating values from neighboring cells, applied to temperature or salinity fields.
- **Logging and Error Handling**: Logs initialization, refinement computation, and grid refinement operations using the `logging` module, with exception handling to catch and log errors.

2 Simulation Logic

The simulation logic in `AdaptiveMeshRefinement.py` centers around the `AdaptiveMeshRef` class, which manages dynamic grid refinement for the simulator.

2.1 Initialization

- **Purpose**: Initializes the adaptive mesh refinement system with grid parameters and a threshold for refinement criteria.
- **Process**:
  - Initializes parameters: `grid_size` (N) and `threshold`, which determines when refinement is triggered based on gradients or vorticity.
  - Creates an empty list (`refined_cells`) to store coordinates of cells

targeted for refinement.

- Logs initialization details and handles exceptions.

2.2 Refinement Computation

- **Purpose**: Identifies grid cells for refinement based on physical criteria derived from ocean and atmosphere temperature fields.
- **Process**:
  - Clips input temperature fields (T_o, T_a) to $[250, 350]$ K to prevent extreme values.
  - Computes gradients of T_o and T_a (∇T_o, ∇T_a) using numerical differentiation.
  - Calculates a combined gradient magnitude across both fields.
  - Computes simplified 2D vorticity from ∇T_o.
  - Creates a refinement mask based on cells where gradient magnitude or vorticity exceeds the threshold.
  - Stores coordinates of cells to refine in `refined_cells`.
  - Returns the refinement mask for use in `Model.py`.

2.3 Grid Refinement

- **Purpose**: Refines specified grid cells by interpolating field values from neighboring cells.
- **Process**:
  - Copies the input field (T) to avoid in-place modification.
  - For each cell in `refined_cells`, creates a 2×2 subgrid with interpolated values based on the current and neighboring grid points.
  - Assigns the mean of the subgrid to the cell, clipped to $[250, 350]$ K.
  - Returns the refined field.

3 Physics and Mathematical Models

The `AdaptiveMeshRefinement` class implements models to identify and refine grid cells based on physical criteria, using numerical approximations for gradients and vorticity.

3.1 Gradient Magnitude

- **Purpose**: Computes the combined gradient magnitude of ocean and atmosphere temperature fields to identify regions with steep changes.
- **Equation**:

$$\nabla T_o = \left(\frac{\partial T_o}{\partial x}, \frac{\partial T_o}{\partial y} \right),$$
$$\nabla T_a = \left(\frac{\partial T_a}{\partial x}, \frac{\partial T_a}{\partial y} \right),$$
$$|\nabla| = \sqrt{\left(\frac{\partial T_o}{\partial x} \right)^2 + \left(\frac{\partial T_o}{\partial y} \right)^2 + \left(\frac{\partial T_a}{\partial x} \right)^2 + \left(\frac{\partial T_a}{\partial y} \right)^2},$$

  where gradients are approximated using central differences, and each term is clipped to $[-10^{10}, 10^{10}]$ before computing the magnitude, which is clipped to $[0, 10^5]$.

- **Implementation**: `compute_refinement` uses `np.gradient` to compute ∇T_o and ∇T_a, ensuring numerical stability through clipping.

3.2 Vorticity

- **Purpose**: Computes a simplified 2D vorticity from ocean temperature gradients to identify regions of rotational flow.
- **Equation**:

$$\omega_x = \frac{\partial}{\partial y} \left(\frac{\partial T_o}{\partial x} \right),$$
$$\omega_y = \frac{\partial}{\partial x} \left(\frac{\partial T_o}{\partial y} \right),$$
$$\omega = |\omega_x - \omega_y|,$$

  where gradients are approximated using central differences, and ω is clipped to $[0, 10^5]$.

- **Implementation**: `compute_refinement` applies `np.gradient` to ∇T_o components to compute vorticity, used as a refinement criterion.

3.3 Refinement Criteria

- **Purpose**: Determines which grid cells require refinement.
- **Equation**:

$$\text{mask}_{i,j} = (|\nabla|_{i,j} > \theta) \text{ or } (\omega_{i,j} > \theta/10),$$

  where θ is the `threshold` parameter.

- **Implementation**: `compute_refinement` generates a boolean mask for cells where either condition is met, storing coordinates in `refined_cells`.

3.4 Grid Refinement

- **Purpose**: Interpolates field values in targeted cells to increase resolution.
- **Equation**:

$$
\begin{aligned}
T_{\text{subgrid},0,0} &= 0.5T_{i,j} + 0.125(T_{i-1,j} + T_{i+1,j} + T_{i,j-1} + T_{i,j+1}),\\
T_{\text{subgrid},0,1} &= 0.5T_{i,j} + 0.125(T_{i-1,j} + T_{i+1,j} + T_{i,j+1} + T_{i,j-1}),\\
T_{\text{subgrid},1,0} &= 0.5T_{i,j} + 0.125(T_{i+1,j} + T_{i-1,j} + T_{i,j-1} + T_{i,j+1}),\\
T_{\text{subgrid},1,1} &= 0.5T_{i,j} + 0.125(T_{i+1,j} + T_{i-1,j} + T_{i,j+1} + T_{i,j-1}),\\
T_{\text{refined},i,j} &= \text{mean}(T_{\text{subgrid}}),
\end{aligned}
$$

  where $T_{\text{refined},i,j}$ is clipped to $[250, 350]$ K.

- **Implementation**: `refine` applies this interpolation to each cell in `refined_cells`, averaging the subgrid to update the field value.

4 Algorithms

4.1 Initialization Algorithm

- **Input**: `grid_size` (N), `threshold`.
- **Steps**:
  1. Log initialization parameters: `grid_size`, `threshold`.
  2. Store `grid_size` and `threshold` as instance variables.
  3. Initialize an empty list `refined_cells`.
  4. Log completion of initialization.
  5. Handle exceptions and log errors if initialization fails.

4.2 Refinement Computation Algorithm (`compute_refinement`)

- **Input**: Ocean temperature T_o, atmosphere temperature T_a.
- **Steps**:
  1. Log start of computation.
  2. Clip T_o and T_a to $[250, 350]$ K.

3. Compute gradients: $\nabla T_o = (\frac{\partial T_o}{\partial x}, \frac{\partial T_o}{\partial y})$, $\nabla T_a = (\frac{\partial T_a}{\partial x}, \frac{\partial T_a}{\partial y})$ using `np.gradient`.
4. Compute gradient magnitude: $|\nabla| = \sqrt{\left(\frac{\partial T_o}{\partial x}\right)^2 + \left(\frac{\partial T_o}{\partial y}\right)^2 + \left(\frac{\partial T_a}{\partial x}\right)^2 + \left(\frac{\partial T_a}{\partial y}\right)^2}$, with terms clipped to $[-10^{10}, 10^{10}]$ and result to $[0, 10^5]$.
5. Compute vorticity: $\omega_x = \frac{\partial}{\partial y}\left(\frac{\partial T_o}{\partial x}\right)$, $\omega_y = \frac{\partial}{\partial x}\left(\frac{\partial T_o}{\partial y}\right)$, $\omega = |\omega_x - \omega_y|$, clipped to $[0, 10^5]$.
6. Create refinement mask: $\text{mask}_{i,j} = (|\nabla|_{i,j} > \theta)$ or $(\omega_{i,j} > \theta/10)$.
7. Store coordinates (i, j) where $\text{mask}_{i,j}$ = True in `refined_cells` for $i, j \in [1, N-2]$.
8. Return `mask`.
9. Log completion or errors if computation fails.

4.3 Grid Refinement Algorithm (`refine`)

- **Input**: Field T (e.g., temperature or salinity).
- **Steps**:
  1. Log start of refinement.
  2. Copy T to `T_refined` to avoid in-place modification.
  3. For each (i, j) in `refined_cells`:
     - Create a 2×2 subgrid:
       * $T_{\text{subgrid},0,0} = 0.5T_{i,j} + 0.125(T_{i-1,j} + T_{i+1,j} + T_{i,j-1} + T_{i,j+1})$.
       * $T_{\text{subgrid},0,1} = 0.5T_{i,j} + 0.125(T_{i-1,j} + T_{i+1,j} + T_{i,j+1} + T_{i,j-1})$.
       * $T_{\text{subgrid},1,0} = 0.5T_{i,j} + 0.125(T_{i+1,j} + T_{i-1,j} + T_{i,j-1} + T_{i,j+1})$.
       * $T_{\text{subgrid},1,1} = 0.5T_{i,j} + 0.125(T_{i+1,j} + T_{i-1,j} + T_{i,j+1} + T_{i,j-1})$.
     - Set $T_{\text{refined},i,j} = \text{mean}(T_{\text{subgrid}})$, clipped to $[250, 350]$ K.
  4. Return `T_refined`.
  5. Log completion or errors if refinement fails.

Turbulent Mixing

June 29, 2025

1 Functionalities

The `TurbulentMixingWindow` class provides the following functionalities:

- **Initialization**: Sets up a window for turbulent mixing analysis, integrating with the model to access ocean temperatures, salinity, and coupling parameters.
- **Simulation Control**: Allows users to start and stop a turbulent mixing simulation, updating model parameters (mixing coefficient, wind speed) dynamically.
- **Visualization**: Displays a thermal front with mixing gradients and an Ekman spiral with vertical temperature profiles, updated in real-time via animation.
- **Parameter Input**: Enables users to input mixing coefficient and wind speed, with validation to ensure physical constraints.
- **Logging and Error Handling**: Logs initialization, simulation control, and plot updates using the `logging` module, with exception handling to display errors via message boxes.

2 Simulation Logic

The simulation logic in `TurbulentMixing.py` centers around the `TurbulentMixingWindow` class, which manages the turbulent mixing analysis and visualization.

2.1 Initialization

- **Purpose**: Initializes the turbulent mixing analysis window with model integration and visualization setup.
- **Process**:
  - Stores the model instance for access to ocean temperatures, salinity,

and coupling parameters.

  - Sets up a window with a control panel for inputting mixing coefficient and wind speed, and buttons for starting/stopping the simulation.

  - Initializes a matplotlib figure with two subplots: one for the thermal front and mixing gradients, and one for the Ekman spiral and vertical temperature profile.

  - Configures a vertical grid (z) from 0 to $-100\,\text{m}$ with 50 points ($\Delta z = -2\,\text{m}$) and an initial linear temperature profile from 300 K to 290 K.

  - Sets up a timer for animation updates and initializes the plot.

  - Logs initialization details and handles exceptions, displaying errors via `QMessageBox`.

2.2 Simulation Control

- **Purpose**: Manages the start and stop of the turbulent mixing simulation.
- **Process (Start)**:
  - Validates inputs: mixing coefficient (> 0) and wind speed (≥ 0).
  - Updates model parameters (`model.coupling.mixing_coeff`, `model.coupling.w`
  - Resets the simulation step and temperature profile.
  - Starts a timer to trigger animation updates every 100 ms.
  - Disables the start button and enables the stop button.
- **Process (Stop)**:
  - Stops the timer and animation.
  - Enables the start button and disables the stop button.
  - Logs actions and handles exceptions, displaying errors via `QMessageBox`.

2.3 Plot Update

- **Purpose**: Updates the visualization of the thermal front, mixing gradients, Ekman spiral, and vertical temperature profile.
- **Process**:
  - Checks if the simulation should continue (current step $<$ total_time$/\Delta t$).

- Validates and updates model parameters from user inputs.
- Generates a synthetic thermal front based on ocean temperatures and a sinusoidal perturbation driven by wind speed.
- Computes turbulent mixing using `model.coupling.compute_turbulent_mixing`
- Calculates mixing gradients for visualization.
- Computes Ekman spiral velocities (u, v) and updates the vertical temperature profile using a diffusion equation.
- Updates the plots: a contour plot with quiver arrows for the thermal front and mixing gradients, and a line plot for the Ekman spiral and temperature profile.
- Increments the simulation step and logs actions.

3 Physics and Mathematical Models

The `TurbulentMixingWindow` class implements models for visualizing turbulent mixing and Ekman dynamics, using numerical approximations for gradients and diffusion.

3.1 Thermal Front

- **Purpose**: Generates a synthetic thermal front to visualize turbulent mixing effects.
- **Equation**:

$$T_{\text{front}}(x, y) = T_{\text{cold}} + \frac{T_{\text{hot}} - T_{\text{cold}}}{2} \left[1 + \tanh\left(\frac{x - x_0}{L}\right)\right],$$

  where $T_{\text{cold}} = 290\,\text{K}$, $T_{\text{hot}} = 300\,\text{K}$, $L = 5.0$, and $x_0 = \frac{\text{nx}}{2} + 10\sin(0.01 \cdot \text{step} \cdot U)$, with U as wind speed.
- **Implementation**: `update_plot` blends the front with ocean temperatures: $T_{\text{ocean}} = 0.8 \cdot T_{\text{ocean}} + 0.2 \cdot T_{\text{front}}$.

3.2 Mixing Gradients

- **Purpose**: Visualizes the spatial variation of turbulent mixing.

- **Equation**:

$$\begin{aligned}\nabla M &= \left(\frac{\partial M}{\partial x}, \frac{\partial M}{\partial y}\right), \\ \frac{\partial M}{\partial x} &\approx \frac{M_{i+1,j} - M_{i-1,j}}{2\Delta x}, \\ \frac{\partial M}{\partial y} &\approx \frac{M_{i,j+1} - M_{i,j-1}}{2\Delta y},\end{aligned}$$

  where M is the mixing field from `compute_turbulent_mixing`.

- **Implementation**: `update_plot` uses `np.gradient` to compute gradients, visualized with quiver arrows on a subsampled grid (every 5th point).

3.3 Ekman Spiral Velocities

- **Purpose**: Models the velocity profile in the ocean surface layer due to wind stress and Coriolis effects.

- **Equations**:

$$\begin{aligned}V_0 &= \frac{\tau}{\rho\sqrt{2\nu f}}, \\ d &= \sqrt{\frac{2\nu}{f}}, \\ u(z) &= V_0 e^{z/d} \cos\left(\frac{\pi}{4} + \frac{z}{d}\right), \\ v(z) &= V_0 e^{z/d} \sin\left(\frac{\pi}{4} + \frac{z}{d}\right),\end{aligned}$$

  where $\tau = 0.1U^2$ is wind stress, $\rho = 1000\,\text{kg/m}^3$, ν is the mixing coefficient, $f = 10^{-4}\,\text{s}^{-1}$ is the Coriolis parameter, and $z \in [0, -100]\,\text{m}$.

- **Implementation**: `update_plot` computes u and v along the vertical grid for visualization.

3.4 Vertical Temperature Diffusion

- **Purpose**: Updates the vertical temperature profile due to turbulent mixing.

- **Equations**:

$$\begin{aligned}\frac{dT}{dt} &= K_v(z)\frac{\partial^2 T}{\partial z^2}, \\ K_v(z) &= \nu e^{z/50}, \\ \frac{\partial^2 T}{\partial z^2} &\approx \frac{T_{k+1} - 2T_k + T_{k-1}}{\Delta z^2},\end{aligned}$$

  where ν is the mixing coefficient, $\Delta z = -2\,\text{m}$, and boundary conditions set $\frac{dT}{dt} = 0$ at the surface and $\frac{dT}{dt}_{k=-1} = \frac{dT}{dt}_{k=-2}$ at the bottom.

- **Implementation**: `update_plot` applies the diffusion equation with $\Delta t = 0.1$ s, updating the temperature profile and setting the surface temperature to the ocean temperature at the grid center.

4 Algorithms

4.1 Initialization Algorithm

- **Input**: Model instance (`model`).
- **Steps**:
  1. Log initialization.
  2. Store `model` as an instance variable.
  3. Set window title to "Turbulent Mixing Analysis" and size to 900×600.
  4. Create a control panel with inputs for mixing coefficient (`model.coupling.mix` and wind speed (`model.coupling.wind_speed`).
  5. Create start and stop buttons, with stop initially disabled.
  6. Set up a matplotlib figure with two subplots.
  7. Initialize a vertical grid: $z = [0, -100]$ m, nz $= 50$, $\Delta z = -2$ m.
  8. Initialize a linear temperature profile: $T = [300, 290]$ K.
  9. Set $\Delta t = 0.1$ s and initialize a timer for 100 ms updates.
  10. Call `update_plot(0)` to initialize the visualization.
  11. Log completion or errors, displaying critical errors via `QMessageBox`.

4.2 Start Simulation Algorithm (`start_simulation`)

- **Input**: None (uses input fields).
- **Steps**:
  1. Log start of simulation.
  2. Read and validate inputs: mixing coefficient (> 0), wind speed (≥ 0).
  3. Update `model.coupling.mixing_coeff` and `model.coupling.wind_speed`.

4. Reset `current_step` to 0 and temperature profile to linear $[300, 290]$ K.
5. Start timer (100 ms interval).
6. Disable start button and enable stop button.
7. Log completion or errors, displaying warnings or critical errors via `QMessageBox`.

4.3 Stop Simulation Algorithm (`stop_simulation`)

– **Input**: None.

– **Steps**:

1. Log stop of simulation.
2. Stop timer and animation (`anim.event_source.stop()`).
3. Set `anim` to None.
4. Enable start button and disable stop button.
5. Log completion or errors, displaying critical errors via `QMessageBox`.

4.4 Plot Update Algorithm (`update_plot`)

– **Input**: Frame number (unused directly, tracks via `current_step`).

– **Steps**:

1. Log start of update at `current_step`.
2. If `current_step` $\geq$ total_time$/\Delta t$ or model is invalid, call `stop_simulation`.
3. Validate and update mixing coefficient (> 0) and wind speed (≥ 0).
4. Generate thermal front:
   * Compute $x_0 = \text{nx}/2 + 10\sin(0.01 \cdot \text{step} \cdot U)$.
   * Compute $T_{\text{front}} = 290 + 5(1 + \tanh((x - x_0)/5))$.
   * Blend: $T_{\text{ocean}} = 0.8 \cdot T_{\text{ocean}} + 0.2 \cdot T_{\text{front}}$.
5. Compute mixing field using `model.coupling.compute_turbulent_mixing`.
6. Compute gradients: $\nabla M = (\frac{\partial M}{\partial x}, \frac{\partial M}{\partial y})$ using `np.gradient`.
7. Subsample gradients every 5th point for quiver plot.

8. Compute Ekman spiral:
    * $\tau = 0.1U^2$, $f = 10^{-4}\,\mathrm{s}^{-1}$, $\rho = 1000\,\mathrm{kg/m}^3$.
    * $V_0 = \tau/(\rho\sqrt{2\nu f})$, $d = \sqrt{2\nu/f}$.
    * $u(z) = V_0 e^{z/d}\cos(\pi/4 + z/d)$, $v(z) = V_0 e^{z/d}\sin(\pi/4 + z/d)$.
9. Update temperature profile:
    * Set surface temperature to $T_{\text{ocean}}[\text{ny}/2, \text{nx}/2]$.
    * Compute $K_v(z) = \nu e^{z/50}$.
    * Compute $\frac{\partial^2 T}{\partial z^2} = \frac{T_{k+1} - 2T_k + T_{k-1}}{\Delta z^2}$.
    * Update: $T+ = \Delta t \cdot K_v \cdot \frac{\partial^2 T}{\partial z^2}$, with boundary conditions.
10. Clear and update plots:
    * Left: Contour plot of T_{ocean} with quiver arrows for ∇M.
    * Right: Line plots of $u(z)$, $v(z)$, and $T(z)$.
11. Increment `current_step`.
12. Log completion or errors, displaying warnings or critical errors via `QMessageBox`.

Surface Layer Physics

June 29, 2025

1 Functionalities

The `SurfaceLayerPhysicsWindow` class provides the following functionalities:

- **Initialization**: Sets up a window for surface layer physics analysis, integrating with the model to access ocean and atmosphere temperatures and coupling parameters.
- **Simulation Control**: Allows users to start and stop a surface layer physics simulation, updating model parameters (drag coefficient, wind speed, sensible and latent heat coefficients) dynamically.
- **Visualization**: Displays a heatmap of surface currents and time series of mean heat flux and wind stress, updated in real-time via animation.
- **Parameter Input**: Enables users to input drag coefficient, wind speed, sensible heat coefficient, and latent heat coefficient, with validation to ensure physical constraints.
- **Logging and Error Handling**: Logs initialization, simulation control, and plot updates using the `logging` module, with exception handling to display errors via message boxes.

2 Simulation Logic

The simulation logic in `SurfaceLayerPhysicsWindow` centers around managing the surface layer physics analysis and visualization.

2.1 Initialization

- **Purpose**: Initializes the surface layer physics analysis window with model integration and visualization setup.
- **Process**:

- Stores the model instance for access to ocean and atmosphere temperatures and coupling parameters.
- Sets up a window with a control panel for inputting drag coefficient, wind speed, sensible heat coefficient, and latent heat coefficient, and buttons for starting/stopping the simulation.
- Initializes a matplotlib figure with two subplots: one for a heatmap of surface currents and one for time series of heat flux and wind stress.
- Sets up a simulation grid matching the model's ocean temperature dimensions (ny $\times$ nx) and initializes surface currents to zero.
- Sets the time step $\Delta t = 1800$ s to match the main simulation.
- Sets up a timer for animation updates every 100 ms and initializes the plot.
- Logs initialization details and handles exceptions, displaying errors via `QMessageBox`.

2.2 Simulation Control

- **Purpose**: Manages the start and stop of the surface layer physics simulation.
- **Process (Start)**:
  - Validates inputs: drag coefficient (> 0), wind speed (≥ 0), sensible heat coefficient (≥ 0), latent heat coefficient (≥ 0).
  - Updates model parameters (`model.coupling.drag_coeff`, `model.coupling.wir`
  - Resets the simulation step, time steps, heat fluxes, wind stresses, and surface currents.
  - Starts a timer to trigger animation updates every 100 ms.
  - Disables the start button and enables the stop button.
- **Process (Stop)**:
  - Stops the timer and animation.
  - Enables the start button and disables the stop button.
  - Logs actions and handles exceptions, displaying errors via `QMessageBox`.

2.3 Plot Update

- **Purpose**: Updates the visualization of surface currents, mean heat flux, and wind stress.
- **Process**:
  - Checks if the simulation should continue (current step $<$ total_time$/\Delta t$).
  - Validates and updates model parameters from user inputs.
  - Computes sensible and latent heat fluxes based on ocean and atmosphere temperatures and wind speed.
  - Computes wind stress based on drag coefficient and wind speed.
  - Updates surface currents using wind stress with a random perturbation scaled by the mixing coefficient.
  - Stores mean heat flux and wind stress for time series visualization.
  - Updates plots: a heatmap of surface currents and a time series of mean heat flux and wind stress.
  - Increments the simulation step and logs actions.

3 Physics and Mathematical Models

The `SurfaceLayerPhysicsWindow` class implements models for surface layer dynamics, focusing on heat fluxes and wind stress.

3.1 Sensible Heat Flux

- **Purpose**: Computes the sensible heat flux due to temperature differences between ocean and atmosphere.
- **Equation**:
$$Q_s = \rho_{\text{air}} C_p^{\text{air}} C_h U (T_o - T_a),$$
where $\rho_{\text{air}} = 1.225\,\text{kg/m}^3$, $C_p^{\text{air}} = 1005\,\text{J/kg/K}$, C_h is the sensible heat coefficient (W/m^2/K), U is wind speed (m/s), T_o is ocean temperature (K), and T_a is atmosphere temperature (K).
- **Implementation**: `update_plot` computes Q_s using user-input C_h and model temperature fields.

3.2 Latent Heat Flux

- **Purpose**: Computes the latent heat flux due to evaporation.

- **Equation**:
$$Q_l = \rho_{\text{air}} L_v C_e U q,$$
where $L_v = 2.5 \times 10^6$ J/kg, C_e is the latent heat coefficient (W/m^2), $q = 0.001$ is a placeholder specific humidity difference, and other variables are as above.

- **Implementation**: `update_plot` uses a constant q for simplicity, with user-input C_e.

3.3 Total Heat Flux

- **Purpose**: Combines sensible and latent heat fluxes.

- **Equation**:
$$Q_{\text{total}} = Q_s + Q_l,$$
where Q_{total} is the total heat flux (W/m^2).

- **Implementation**: `update_plot` computes the mean of Q_{total} for time series visualization.

3.4 Wind Stress

– **Purpose**: Computes the wind stress on the ocean surface.

– **Equation**:
$$\tau = \rho_{\text{air}} C_d U^2,$$
where C_d is the drag coefficient, and other variables are as above.

– **Implementation**: `update_plot` uses user-input C_d and U.

3.5 Surface Currents

∗ **Purpose**: Updates surface currents driven by wind stress with turbulent mixing effects.

∗ **Equation**:
$$\mathbf{u}_{\text{surface}}(t + \Delta t) = \mathbf{u}_{\text{surface}}(t) + \Delta t \cdot \frac{\tau}{\rho_{\text{water}}} \cdot \mathbf{n},$$
$$\mathbf{n} \sim \mathcal{N}(0, k_m),$$
where $\rho_{\text{water}} = 1000\,\text{kg/m}^3$, k_m is the mixing coefficient, $\mathbf{n}$ is a random normal vector, and $\mathbf{u}_{\text{surface}}$ is clipped to $[-0.5, 0.5]$ m/s.

∗ **Implementation**: `update_plot` updates currents with a random perturbation scaled by k_m.

4 Algorithms

4.1 Initialization Algorithm

· **Input**: Model instance (`model`).

· **Steps**:

1. Log initialization.
2. Store `model` as an instance variable.
3. Set window title to ”Surface Layer Physics Analysis” and size to 900×600.
4. Create a control panel with inputs for drag coefficient (`model.coupling.`
   wind speed (`model.coupling.wind_speed`), sensible heat coefficient ($10.0\,\mathrm{W/m^2/K}$), and latent heat coefficient ($20.0\,\mathrm{W/m^2}$).
5. Create start and stop buttons, with stop initially disabled.
6. Initialize a matplotlib figure with two subplots: heatmap and time series.
7. Set up simulation grid (ny $\times$ nx) from model’s ocean temperatures and initialize surface currents to zero.
8. Set $\Delta t = 1800\,\mathrm{s}$.
9. Initialize empty lists for time steps, heat fluxes, and wind stresses.
10. Set up a timer for 100 ms updates and call `update_plot(0)`.
11. Log completion or errors, displaying critical errors via `QMessageBox`.

4.2 Start Simulation Algorithm (`start_simulation`)

· **Input**: None (uses input fields).

· **Steps**:

1. Log start of simulation.
2. Read and validate inputs: drag coefficient (> 0), wind speed (≥ 0), sensible heat coefficient (≥ 0), latent heat coefficient (≥ 0).
3. Update `model.coupling.drag_coeff` and `model.coupling.wind_spe`

4. Reset `current_step`, `time_steps`, `heat_fluxes`, `wind_stresses`, and `surface_currents`.

5. Start timer (100 ms interval).

6. Disable start button and enable stop button.

7. Log completion or errors, displaying warnings or critical errors via `QMessageBox`.

4.3 Stop Simulation Algorithm (`stop_simulation`)

· **Input**: None.

· **Steps**:

1. Log stop of simulation.

2. Stop timer and animation (`anim.event_source.stop()`).

3. Set `anim` to None.

4. Enable start button and disable stop button.

5. Log completion or errors, displaying critical errors via `QMessageBox`.

4.4 Plot Update Algorithm (`update_plot`)

· **Input**: Frame number (unused directly, tracks via `current_step`).

· **Steps**:

1. Log start of update at `current_step`.

2. If `current_step` $\geq$ total_time/Δt or model is invalid, call `stop_simulation`.

3. Validate and read inputs: drag coefficient, wind speed, sensible and latent heat coefficients.

4. Compute sensible heat flux: $Q_s = \rho_{\text{air}} C_p^{\text{air}} C_h U (T_o - T_a)$.

5. Compute latent heat flux: $Q_l = \rho_{\text{air}} L_v C_e U q$, with $q = 0.001$.

6. Compute total heat flux: $Q_{\text{total}} = Q_s + Q_l$.

7. Compute wind stress: $\tau = \rho_{\text{air}} C_d U^2$.

8. Update surface currents: $\mathbf{u}_{\text{surface}} += \Delta t \cdot (\tau/1000) \cdot \mathbf{n}$, where $\mathbf{n} \sim \mathcal{N}(0, k_m)$, clipped to $[-0.5, 0.5]$ m/s.

9. Append mean Q_{total}, τ, and current time (step $\cdot\, \Delta t$) to respective lists.

10. Clear and update plots:

11. Top: Heatmap of $\mathbf{u}_{\text{surface}}$ with `coolwarm` colormap, $[-0.5, 0.5]$ m/s.

12. Bottom: Time series of mean Q_{total} and τ.

13. Increment `current_step`.

14. Log completion or errors, displaying warnings or critical errors via `QMessageBox`.

High-Order Time Stepping

June 29, 2025

1 Functionalities

The `HighOrderTimeSteppingWindow` class provides the following functionalities:

- **Initialization**: Sets up a window for high-order time stepping analysis, integrating with the model to access ocean and atmosphere temperatures and coupling parameters.
- **Simulation Control**: Allows users to start and stop a simulation, updating parameters (time step, drag coefficient, sensible heat coefficient, boundary layer depth, KPP mixing coefficient) and selecting between Bulk and KPP schemes and Euler or RK4 methods.
- **Visualization**: Displays a heatmap of heat flux or diffusivity and a time series of mean values, comparing Euler and RK4 methods, updated in real-time via animation.
- **Parameter Input**: Enables users to input simulation parameters and select schemes and methods, with validation to ensure physical constraints.
- **Logging and Error Handling**: Logs initialization, simulation control, and plot updates using the `logging` module, with exception handling to display errors via message boxes.

2 Simulation Logic

The simulation logic in `HighOrderTimeSteppingWindow` centers around managing high-order time stepping analysis and visualization.

2.1 Initialization

- **Purpose**: Initializes the analysis window with model integration and visualization setup.

- **Process**:
  - Stores the model instance for access to ocean and atmosphere temperatures and coupling parameters.
  - Sets up a window with a control panel for inputting time step, drag coefficient, sensible heat coefficient, boundary layer depth, KPP mixing coefficient, and selecting Bulk/KPP schemes and Euler/RK4 methods.
  - Initializes a matplotlib figure with two subplots: one for a heatmap of heat flux or diffusivity, and one for a time series of mean values.
  - Sets up a simulation grid (ny $\times$ nx) matching the model's ocean temperatures and initializes arrays for Euler and RK4 heat fluxes and diffusivities.
  - Sets default time step $\Delta t = 1800$ s.
  - Sets up a timer for animation updates every 100 ms and initializes the plot.
  - Logs initialization details and handles exceptions, displaying errors via `QMessageBox`.

2.2 Simulation Control

- **Purpose**: Manages the start and stop of the time stepping simulation.
- **Process (Start)**:
  - Validates inputs: time step (> 0), drag coefficient (> 0), sensible heat coefficient (≥ 0), boundary layer depth (> 0), KPP mixing coefficient (> 0).
  - Updates model parameters (`model.coupling.drag_coeff`) and sets `dt`.
  - Resets simulation step, time steps, Euler/RK4 values, and heat flux/diffusivity arrays.
  - Starts a timer to trigger animation updates every 100 ms.
  - Disables the start button and enables the stop button.
- **Process (Stop)**:
  - Stops the timer and animation.
  - Enables the start button and disables the stop button.

– Logs actions and handles exceptions, displaying errors via `QMessageBox`.

2.3 Plot Update

- **Purpose**: Updates the visualization of heat flux or diffusivity and their mean values over time.
- **Process**:
  - Checks if the simulation should continue (current step $<$ total_time$/\Delta t$).
  - Validates and updates parameters from user inputs and selected scheme/method.
  - Computes heat flux (Bulk scheme) or diffusivity (KPP scheme) using Euler and RK4 methods.
  - Stores mean values for time series visualization.
  - Updates plots: a heatmap of heat flux or diffusivity with annotations, and a time series comparing Euler and RK4 mean values.
  - Increments the simulation step and logs actions.

3 Physics and Mathematical Models

The `HighOrderTimeSteppingWindow` class implements models for heat flux and diffusivity, comparing Euler and RK4 time stepping methods.

3.1 Bulk Heat Flux (Bulk Scheme)

- **Purpose**: Computes sensible heat flux between ocean and atmosphere.
- **Equation**:
$$Q_b = \rho_{\text{air}} C_p^{\text{air}} C_h U (T_o - T_a),$$
where $\rho_{\text{air}} = 1.225\,\text{kg/m}^3$, $C_p^{\text{air}} = 1005\,\text{J/kg/K}$, C_h is the sensible heat coefficient (W/m^2/K), U is wind speed (m/s), T_o is ocean temperature (K), and T_a is atmosphere temperature (K).
- **Implementation**: `update_plot` computes Q_b for the Euler method directly and as the initial step for RK4.

3.2 KPP Diffusivity (KPP Scheme)

- **Purpose**: Computes diffusivity for the K-Profile Parameterization (KPP) scheme.

- **Equation**:

$$K(z) = k_m \left(1 - \frac{z}{h}\right)^2,$$
$$Q_{\text{KPP}} = -K(z)\frac{\partial T_o}{\partial y},$$

where k_m is the KPP mixing coefficient (m^2/s), h is the boundary layer depth (m), $z \in [0, h]$, and $\frac{\partial T_o}{\partial y}$ is approximated as $\frac{T_{o,i+1,j} - T_{o,i-1,j}}{2\Delta y}$. $K(z)$ is clipped to $[0, k_m]$.

- **Implementation**: `update_plot` computes $K(z)$ and Q_{KPP} for the Euler method and as the initial step for RK4.

3.3 Euler Method

– **Purpose**: Updates heat flux or diffusivity using a first-order explicit method.

– **Equation**:

$$Q(t + \Delta t) = Q(t),$$

where Q is Q_b (Bulk) or Q_{KPP} (KPP), computed directly from the current state.

– **Implementation**: `update_plot` applies the equations above for Euler updates.

3.4 RK4 Method

∗ **Purpose**: Updates heat flux or diffusivity using a fourth-order Runge-Kutta method.

∗ **Equations**:

$$\begin{aligned}
k_1 &= f(T_o, T_a), \\
k_2 &= f\left(T_o + \frac{\Delta t}{2}\frac{k_1}{C_h}, T_a + \frac{\Delta t}{2}\frac{k_1}{C_h}\right) \quad \text{(Bulk)}, \\
k_2 &= f(T_o, T_a) \quad \text{(KPP)}, \\
k_3 &= f\left(T_o + \frac{\Delta t}{2}\frac{k_2}{C_h}, T_a + \frac{\Delta t}{2}\frac{k_2}{C_h}\right) \quad \text{(Bulk)}, \\
k_3 &= f(T_o, T_a) \quad \text{(KPP)}, \\
k_4 &= f\left(T_o + \Delta t\frac{k_3}{C_h}, T_a + \Delta t\frac{k_3}{C_h}\right) \quad \text{(Bulk)}, \\
k_4 &= f(T_o, T_a) \quad \text{(KPP)}, \\
Q(t + \Delta t) &= Q(t) + \frac{\Delta t}{6}(k_1 + 2k_2 + 2k_3 + k_4),
\end{aligned}$$

where f is the flux derivative function (Q_b for Bulk, Q_{KPP} for KPP), and C_h is omitted for KPP as temperatures are not updated directly.

* **Implementation**: `update_plot` computes intermediate steps using `compute_flux_derivative` and updates Q with the RK4 formula.

4 Algorithms

4.1 Initialization Algorithm

· **Input**: Model instance (`model`).

· **Steps**:

1. Log initialization.
2. Store `model` as an instance variable.
3. Set window title to "High-Order Time Stepping Analysis" and size to 900×600.
4. Create a control panel with inputs for time step (1800 s), drag coefficient (`model.coupling.drag_coeff`), sensible heat coefficient (10.0 W/m^2/K), boundary layer depth (50.0 m), KPP mixing coefficient (0.01 m^2/s), and selectors for Bulk/KPP schemes and Euler/RK4 methods.
5. Create start and stop buttons, with stop initially disabled.
6. Initialize a matplotlib figure with two subplots: heatmap and time series.
7. Set up simulation grid (ny $\times$ nx) from model's ocean temperatures and initialize arrays for Euler and RK4 heat fluxes/diffusivities.
8. Set default $\Delta t = 1800$ s.
9. Initialize empty lists for time steps and Euler/RK4 values.
10. Set up a timer for 100 ms updates and call `update_plot(0)`.
11. Log completion or errors, displaying critical errors via `QMessageBox`.

4.2 Start Simulation Algorithm (`start_simulation`)

· **Input**: None (uses input fields).

· **Steps**:

1. Log start of simulation.
2. Read and validate inputs: time step (> 0), drag coefficient (> 0), sensible heat coefficient (≥ 0), boundary layer depth (> 0), KPP mixing coefficient (> 0).
3. Update `model.coupling.drag_coeff` and set `dt`.
4. Reset `current_step`, `time_steps`, `euler_values`, `rk4_values`, and heat flux/diffusivity arrays.
5. Start timer (100 ms interval).
6. Disable start button and enable stop button.
7. Log completion or errors, displaying warnings or critical errors via `QMessageBox`.

4.3 Stop Simulation Algorithm (`stop_simulation`)

· **Input**: None.

· **Steps**:

1. Log stop of simulation.
2. Stop timer and animation (`anim.event_source.stop()`).
3. Set `anim` to None.
4. Enable start button and disable stop button.
5. Log completion or errors, displaying critical errors via `QMessageBox`.

4.4 Plot Update Algorithm (`update_plot`)

· **Input**: Frame number (unused directly, tracks via `current_step`).

· **Steps**:

1. Log start of update at `current_step`.
2. If `current_step` $\geq$ total_time/Δt or model is invalid, call `stop_simulation`.
3. Validate and read inputs: time step, sensible heat coefficient, boundary layer depth, KPP mixing coefficient, scheme, and method.
4. For Euler method:

5. Bulk: Compute $Q_b = \rho_{\text{air}} C_p^{\text{air}} C_h U (T_o - T_a)$.
6. KPP: Compute $K(z) = k_m(1 - z/h)^2$, clipped to $[0, k_m]$, and $Q_{\text{KPP}} = -K \frac{\partial T_o}{\partial y}$.
7. For RK4 method:
8. Compute $k_1 = f(T_o, T_a)$.
9. Compute $k_2 = f(T_o + \frac{\Delta t}{2} \frac{k_1}{C_h}, T_a + \frac{\Delta t}{2} \frac{k_1}{C_h})$ (Bulk) or $f(T_o, T_a)$ (KPP).
10. Compute $k_3 = f(T_o + \frac{\Delta t}{2} \frac{k_2}{C_h}, T_a + \frac{\Delta t}{2} \frac{k_2}{C_h})$ (Bulk) or $f(T_o, T_a)$ (KPP).
11. Compute $k_4 = f(T_o + \Delta t \frac{k_3}{C_h}, T_a + \Delta t \frac{k_3}{C_h})$ (Bulk) or $f(T_o, T_a)$ (KPP).
12. Update: $Q = Q + \frac{\Delta t}{6}(k_1 + 2k_2 + 2k_3 + k_4)$.
13. Store mean values of heat flux (Bulk) or diffusivity (KPP) for Euler and RK4.
14. Update plots:
15. Left: Heatmap of Q_b or $K(z)$ with annotations every ny$/5 \times$ nx$/5$ points, using `coolwarm` (Bulk) or `viridis` (KPP).
16. Right: Time series of mean Euler and RK4 values.
17. Increment `current_step`.
18. Log completion or errors, displaying warnings or critical errors via `QMessageBox`.

Boundary Layer Schemes

June 29, 2025

1 Functionalities

The `BoundaryLayerSchemesWindow` class provides the following functionalities:

- **Initialization**: Sets up a window for boundary layer schemes analysis, integrating with the model to access ocean and atmosphere temperatures and coupling parameters.
- **Simulation Control**: Allows users to start and stop a simulation, updating parameters (drag coefficient, sensible heat coefficient, boundary layer depth, KPP mixing coefficient) and selecting between Bulk and KPP schemes.
- **Visualization**: Displays a heatmap of heat flux (Bulk scheme) or diffusivity (KPP scheme) with annotations and a scatter plot of flux/diffusivity versus temperature gradient, updated in real-time via animation.
- **Parameter Input**: Enables users to input simulation parameters and select schemes, with validation to ensure physical constraints.
- **Logging and Error Handling**: Logs initialization, simulation control, and plot updates using the `logging` module, with exception handling to display errors via message boxes.

2 Simulation Logic

The simulation logic in `BoundaryLayerSchemesWindow` centers around managing boundary layer schemes analysis and visualization.

2.1 Initialization

- **Purpose**: Initializes the analysis window with model integration and visualization setup.
- **Process**:

- Stores the model instance for access to ocean and atmosphere temperatures and coupling parameters.
- Sets up a window with a control panel for inputting drag coefficient, sensible heat coefficient, boundary layer depth, KPP mixing coefficient, and selecting Bulk/KPP schemes.
- Initializes a matplotlib figure with two subplots: one for a heatmap of heat flux or diffusivity with annotations, and one for a scatter plot of flux/diffusivity versus temperature gradient.
- Sets up a simulation grid (ny $\times$ nx) matching the model's ocean temperatures and initializes arrays for heat flux and diffusivity.
- Sets the time step $\Delta t = 1800$ s to match the main simulation.
- Sets up a timer for animation updates every 100 ms and initializes the plot.
- Logs initialization details and handles exceptions, displaying errors via `QMessageBox`.

2.2 Simulation Control

- **Purpose**: Manages the start and stop of the boundary layer schemes simulation.
- **Process (Start)**:
  - Validates inputs: drag coefficient (> 0), sensible heat coefficient (≥ 0), boundary layer depth (> 0), KPP mixing coefficient (> 0).
  - Updates model parameters (`model.coupling.drag_coeff`).
  - Resets simulation step, time steps, flux arrays, and diffusivity arrays.
  - Starts a timer to trigger animation updates every 100 ms.
  - Disables the start button and enables the stop button.
- **Process (Stop)**:
  - Stops the timer and animation.
  - Enables the start button and disables the stop button.
  - Logs actions and handles exceptions, displaying errors via `QMessageBox`.

2.3 Plot Update

- **Purpose**: Updates the visualization of heat flux or diffusivity and their relationship with temperature gradients.
- **Process**:
  - Checks if the simulation should continue (current step $<$ total_time$/\Delta t$).
  - Validates and updates parameters from user inputs and selected scheme.
  - Computes heat flux (Bulk scheme) or diffusivity and heat flux (KPP scheme) based on ocean and atmosphere temperatures and wind speed.
  - Stores flux and temperature gradient data for visualization.
  - Updates plots: a heatmap of heat flux or diffusivity with sparse annotations, and a scatter plot of flux/diffusivity versus temperature gradient colored by time.
  - Increments the simulation step and logs actions.

3 Physics and Mathematical Models

The `BoundaryLayerSchemesWindow` class implements models for boundary layer dynamics, comparing Bulk and KPP schemes.

3.1 Bulk Heat Flux (Bulk Scheme)

- **Purpose**: Computes sensible heat flux between ocean and atmosphere.
- **Equation**:
$$Q_b = \rho_{\text{air}} C_p^{\text{air}} C_h U (T_o - T_a),$$
where $\rho_{\text{air}} = 1.225\,\text{kg/m}^3$, $C_p^{\text{air}} = 1005\,\text{J/kg/K}$, C_h is the sensible heat coefficient (W/m^2/K), U is wind speed (m/s), T_o is ocean temperature (K), and T_a is atmosphere temperature (K).
- **Implementation**: `update_plot` computes Q_b using user-input C_h and model temperature fields.

3.2 KPP Scheme

- **Purpose**: Computes turbulent diffusivity and heat flux using the K-Profile Parameterization (KPP) scheme.

- **Equations**:

$$K(z) = k_m \left(1 - \frac{z}{h}\right)^2,$$
$$Q_{\text{KPP}} = -K(z)\frac{\partial T_o}{\partial y},$$

where k_m is the KPP mixing coefficient (m^2/s), h is the boundary layer depth (m), $z \in [0, h]$, and $\frac{\partial T_o}{\partial y} \approx \frac{T_{o,i+1,j} - T_{o,i-1,j}}{2\Delta y}$. $K(z)$ is clipped to $[0, k_m]$.

- **Implementation**: `update_plot` computes $K(z)$ and Q_{KPP} using user-input k_m and h.

3.3 Temperature Gradient

- **Purpose**: Computes the temperature difference driving the heat flux.
- **Equation**:

$$\Delta T = T_o - T_a,$$

where ΔT is the temperature gradient (K).

- **Implementation**: `update_plot` computes ΔT for scatter plot visualization.

4 Algorithms

4.1 Initialization Algorithm

- **Input**: Model instance (`model`).
- **Steps**:
  1. Log initialization.
  2. Store `model` as an instance variable.
  3. Set window title to "Boundary Layer Schemes Analysis" and size to 900×600.
  4. Create a control panel with inputs for drag coefficient (`model.coupling.drag_coe` sensible heat coefficient ($10.0\,\text{W/m}^2/\text{K}$), boundary layer depth ($50.0\,\text{m}$), KPP mixing coefficient ($0.01\,\text{m}^2/\text{s}$), and a selector for Bulk/KPP schemes.
  5. Create start and stop buttons, with stop initially disabled.
  6. Initialize a matplotlib figure with two subplots: heatmap and scatter plot.

7. Set up simulation grid (ny $\times$ nx) from model's ocean temperatures and initialize arrays for heat flux and diffusivity.
8. Set $\Delta t = 1800$ s.
9. Initialize empty lists for time steps, bulk fluxes, KPP fluxes, and temperature gradients.
10. Set up a timer for 100 ms updates and call `update_plot(0)`.
11. Log completion or errors, displaying critical errors via `QMessageBox`.

4.2 Start Simulation Algorithm (`start_simulation`)

- **Input**: None (uses input fields).
- **Steps**:
  1. Log start of simulation.
  2. Read and validate inputs: drag coefficient (> 0), sensible heat coefficient (≥ 0), boundary layer depth (> 0), KPP mixing coefficient (> 0).
  3. Update `model.coupling.drag_coeff`.
  4. Reset `current_step`, `time_steps`, `bulk_fluxes`, `kpp_fluxes`, `temp_gradients`, and heat flux/diffusivity arrays.
  5. Start timer (100 ms interval).
  6. Disable start button and enable stop button.
  7. Log completion or errors, displaying warnings or critical errors via `QMessageBox`.

4.3 Stop Simulation Algorithm (`stop_simulation`)

- **Input**: None.
- **Steps**:
  1. Log stop of simulation.
  2. Stop timer and animation (`anim.event_source.stop()`).
  3. Set `anim` to None.
  4. Enable start button and disable stop button.
  5. Log completion or errors, displaying critical errors via `QMessageBox`.

4.4 Plot Update Algorithm (`update_plot`)

- **Input**: Frame number (unused directly, tracks via `current_step`).
- **Steps**:
  1. Log start of update at `current_step`.
  2. If `current_step` $\geq$ total_time$/\Delta t$ or model is invalid, call `stop_simulation`.
  3. Validate and read inputs: drag coefficient, sensible heat coefficient, boundary layer depth, KPP mixing coefficient, and scheme.
  4. Compute Bulk heat flux: $Q_b = \rho_{\text{air}} C_p^{\text{air}} C_h U (T_o - T_a)$.
  5. Compute KPP diffusivity and flux:
     - $K(z) = k_m \left(1 - \frac{z}{h}\right)^2$, clipped to $[0, k_m]$.
     - $Q_{\text{KPP}} = -K \frac{\partial T_o}{\partial y}$, with $\frac{\partial T_o}{\partial y} \approx \frac{T_{o,i+1,j} - T_{o,i-1,j}}{2\Delta y}$.
  6. Update fields: Q_b for Bulk scheme, $K(z)$ for KPP scheme.
  7. Store flattened Q_b, Q_{KPP}, and $\Delta T = T_o - T_a$ for scatter plot.
  8. Update plots:
     - Left: Heatmap of Q_b (Bulk, `coolwarm`) or $K(z)$ (KPP, `viridis`) with annotations every ny$/5 \times$ nx$/5$.
     - Right: Scatter plot of Q_b or $K(z)$ vs. ΔT, colored by time (`plasma`).
  9. Increment `current_step`.
  10. Log completion or errors, displaying warnings or critical errors via `QMessageBox`.

Air-Sea Interaction

June 29, 2025

1 Functionalities

The `AirSeaInteractionWindow` class provides the following functionalities:

- **Initialization**: Sets up a window for air-sea interaction analysis, using synthetic data for ocean and atmosphere fields (temperatures, velocities, CO2 concentrations, salinity, moisture).
- **Simulation Control**: Allows users to start, pause, resume, and stop a simulation, updating parameters (drag coefficient, wind speed, wind angle, ocean current speed, evaporation rate, precipitation rate, CO2 transfer coefficient, plot coordinates) and selecting variables (wind stress, heat flux, freshwater flux, CO2 flux).
- **Visualization**: Displays a Hovmöller diagram, a heatmap with wind stress vectors and contour lines, and a time series of flux values at a selected point, updated in real-time via animation with adjustable speed.
- **Parameter Input**: Enables users to input simulation parameters, select variables, and adjust animation speed, with validation to ensure physical constraints.
- **Logging and Error Handling**: Logs initialization, simulation control, and plot updates using the `logging` module, with exception handling to display errors via message boxes.

2 Simulation Logic

The simulation logic in `AirSeaInteractionWindow` centers around managing air-sea interaction analysis and visualization with synthetic data.

2.1 Initialization

- **Purpose**: Initializes the analysis window with synthetic data and visualization setup.

- **Process**:
  - Sets up a 100×100 grid with $\Delta x = \Delta y = 1000$ m, $\Delta t = 0.1$ s, and total time 100 s.
  - Initializes synthetic fields: ocean/atmosphere temperatures, ocean velocities, salinity, moisture, and CO2 concentrations using sinusoidal/cosinusoidal patterns.
  - Sets default parameters: drag coefficient (0.001), wind speed (10.0 m/s), wind angle ($0°$), ocean current speed (0.1 m/s), evaporation/precipitation rates (10^{-6} kg/m^2/s), CO2 transfer coefficient (10^{-5} m/s).
  - Sets up a window with a control panel for inputting parameters, selecting variables, and adjusting animation speed (50–500 ms).
  - Initializes a matplotlib figure with three subplots: Hovmöller diagram, heatmap with vectors, and time series.
  - Initializes arrays for wind stress, heat flux, freshwater flux, and CO2 flux, and lists for time steps and time series data.
  - Sets up a timer for animation updates and initializes the plot.
  - Logs initialization details and handles exceptions, displaying errors via `QMessageBox`.

2.2 Simulation Control

- **Purpose**: Manages the start, pause, resume, and stop of the air-sea interaction simulation.
- **Process (Start)**:
  - Validates inputs: drag coefficient (> 0), wind speed (≥ 0), ocean current speed (≥ 0), evaporation rate (≥ 0), precipitation rate (≥ 0), CO2 transfer coefficient (> 0), plot coordinates (within grid).
  - Updates simulation parameters and resets synthetic fields with updated ocean current speed and time-varying patterns.
  - Resets simulation step, time steps, flux arrays, and time series data.
  - Starts a timer with user-defined animation speed.
  - Disables start button and enables pause/stop buttons.
- **Process (Pause)**:
  - Stops the timer and animation, setting `is_paused` to True.

  - Disables pause button and enables resume/stop buttons.

- **Process (Resume)**:
  - Restarts the animation and timer with current animation speed.
  - Sets `is_paused` to False, enables pause/stop buttons, and disables resume button.

- **Process (Stop)**:
  - Stops the timer and animation, resets `is_paused`.
  - Enables start button and disables pause/resume/stop buttons.

- **Logging**: Logs actions and handles exceptions, displaying errors via `QMessageBox`.

2.3 Plot Update

- **Purpose**: Updates the visualization of selected fluxes and their temporal/spatial evolution.

- **Process**:
  - Checks if the simulation should continue (current step $<$ total_time$/\Delta t$).
  - Updates synthetic fields with time-varying sinusoidal patterns.
  - Computes wind stress, heat flux, freshwater flux, and CO2 flux.
  - Stores data for Hovmöller (mid-y slice) and time series (at user-defined coordinates).
  - Updates plots: Hovmöller diagram, heatmap with wind stress vectors and contours, and time series of all fluxes at the selected point.
  - Increments the simulation step and logs actions.

3 Physics and Mathematical Models

The `AirSeaInteractionWindow` class implements models for air-sea interaction fluxes, incorporating momentum, heat, freshwater, and CO2 exchanges.

3.1 Sea Surface Roughness

- **Purpose**: Computes an adjusted drag coefficient based on sea surface roughness.

- **Equations**:

$$
\begin{aligned}
u_* &= \sqrt{\frac{\rho_{\text{air}} C_d U^2}{\rho_{\text{water}}}}, \\
z_0 &= \alpha \frac{u_*^2 + 0.1 U_{\text{ocean}}^2}{g}, \\
C_d^{\text{adj}} &= C_d \left(1 + 0.1 \ln\left(\max(z_0, 10^{-6})\right)\right),
\end{aligned}
$$

  where u_* is the friction velocity (m/s), C_d is the drag coefficient, U is wind speed (m/s), $\rho_{\text{air}} = 1.225\,\text{kg/m}^3$, $\rho_{\text{water}} = 1025\,\text{kg/m}^3$, $U_{\text{ocean}} = \sqrt{u_{\text{ocean}}^2 + v_{\text{ocean}}^2}$ (m/s), $\alpha = 0.018$, $g = 9.81\,\text{m/s}^2$, and C_d^{adj} is clipped to $[10^{-4}, 10^{-2}]$.

- **Implementation**: `compute_sea_surface_roughness` calculates C_d^{adj} for use in momentum and heat flux computations.

3.2 Momentum Flux (Wind Stress)

- **Purpose**: Computes wind stress and its components based on relative wind-ocean velocity.

- **Equations**:

$$
\begin{aligned}
u_{\text{rel}} &= U \cos(\theta) - u_{\text{ocean}}, \\
v_{\text{rel}} &= U \sin(\theta) - v_{\text{ocean}}, \\
s_{\text{rel}} &= \sqrt{u_{\text{rel}}^2 + v_{\text{rel}}^2}, \\
\tau &= \rho_{\text{air}} C_d^{\text{adj}} s_{\text{rel}}^2, \\
\tau_u &= \tau \frac{u_{\text{rel}}}{\max(s_{\text{rel}}, 10^{-6})}, \\
\tau_v &= \tau \frac{v_{\text{rel}}}{\max(s_{\text{rel}}, 10^{-6})},
\end{aligned}
$$

  where θ is the wind angle (radians), s_{rel} is clipped to $[0, 10^3]$, and τ, τ_u, τ_v are clipped to $[-10^5, 10^5]\,\text{N/m}^2$.

- **Implementation**: `compute_momentum_flux` calculates τ, τ_u, and τ_v for visualization and vector plotting.

3.3 Heat Flux

- **Purpose**: Computes total heat flux (sensible and latent).

- **Equations**:

$$
\begin{aligned}
k_{\text{sensible}} &= 0.01 \frac{C_d^{\text{adj}}}{C_d}, \\
Q_{\text{sensible}} &= \rho_{\text{air}} C_p^{\text{air}} k_{\text{sensible}} (T_a - T_o), \\
Q_{\text{latent}} &= \rho_{\text{air}} L_v E \text{sgn}(T_a - T_o), \\
Q_{\text{total}} &= Q_{\text{sensible}} + Q_{\text{latent}},
\end{aligned}
$$

  where $C_p^{\text{air}} = 1005$ J/kg/K, $L_v = 2.5 \times 10^6$ J/kg, E is the evaporation rate (kg/m^2/s), $T_a - T_o$ is clipped to $[-100, 100]$ K, and Q_{total} is clipped to $[-10^6, 10^6]$ W/m^2.

- **Implementation**: `compute_heat_flux` calculates Q_{total} using adjusted drag coefficient and user-input evaporation rate.

3.4 Freshwater Flux

- **Purpose**: Computes freshwater flux and salinity tendency.

- **Equations**:

$$
\begin{aligned}
P &= P_0 \left(1 + 0.5 \frac{q}{0.01}\right), \\
E &= E_0 \left(1 + 0.1 \frac{S}{35.0}\right), \\
F &= P - E, \\
\frac{\partial S}{\partial t} &= -\frac{SF}{\rho_{\text{water}} h},
\end{aligned}
$$

  where P_0 is the precipitation rate (kg/m^2/s), $q/0.01$ is clipped to $[0, 2]$, $S/35.0$ is clipped to $[0.8, 1.2]$, $h = 1000$ m, and F, $\frac{\partial S}{\partial t}$ are clipped to $[-10^{-3}, 10^{-3}]$ kg/m^2/s.

- **Implementation**: `compute_freshwater_flux` calculates F and $\frac{\partial S}{\partial t}$ using user-input rates and synthetic salinity/moisture.

3.5 CO2 Flux

4 Algorithms

4.1 Initialization Algorithm

- **Input**: None (uses synthetic data).

- **Steps**:

  1. Log initialization.

  2. Set grid (100×100), $\Delta x = \Delta y = 1000$ m, $\Delta t = 0.1$ s, total time 100 s.

3. Initialize synthetic fields: $T_o = 290 + 10\sin(2\pi x/L_x)$, $T_a = 295 + 5\cos(2\pi y/L_y)$, $u_{\text{ocean}} = 0.1\cos(2\pi x/L_x)$, $v_{\text{ocean}} = 0.1\sin(2\pi y/L_y)$, $S = 35$ psu, $q = 0.01$ kg/kg, $C_{\text{ocean}} = 400$ ppm, $C_{\text{atm}} = 410$ ppm.

4. Set default parameters: $C_d = 0.001$, $U = 10.0$ m/s, $\theta = 0°$, $U_{\text{ocean}} = 0.1$ m/s, $E_0 = P_0 = 10^{-6}$ kg/m^2/s, $k_0 = 10^{-5}$ m/s.

5. Set window title to "Enhanced Air-Sea Interaction Analysis" and size to 1000×700.

6. Create a control panel with inputs for parameters, variable selector, and animation speed slider (50–500 ms).

7. Create start, pause, resume, and stop buttons, with pause/resume/stop initially disabled.

8. Initialize a matplotlib figure with three subplots: Hovmöller, heatmap, and time series.

9. Initialize arrays for fluxes and lists for time steps and time series data.

10. Set up a timer and call `update_plot(0)`.

11. Log completion or errors, displaying critical errors via `QMessageBox`.

4.2 Start Simulation Algorithm (`start_simulation`)

4.3 Pause Simulation Algorithm (`pause_simulation`)

4.4 Resume Simulation Algorithm (`resume_simulation`)

4.5 Stop Simulation Algorithm (`stop_simulation`)

4.6 Plot Update Algorithm (`update_plot`)

Cloud Microphysics

June 29, 2025

1 Functionalities

The `CloudMicroPhysicsWindow` class provides core functionalities for simulating and visualizing cloud microphysics processes:

- **Initialization**: Configures a window for analyzing cloud microphysics, simulating 100 cloud droplets with synthetic data for radii, concentrations, and liquid water content (LWC).
- **Simulation Control**: Supports starting, pausing, resuming, and stopping the simulation, with user-defined parameters (updraft velocity, supersaturation, droplet radius, concentration, aerosol properties, air temperature, wind speed, wind angle, autoconversion threshold, wind shear) and visualization style selection.
- **Visualization**: Displays three plots updated via animation (50–500 ms): a scatterplot or histogram (colored by LWC or droplet size), a histogram or vertical profile, and a time series of mean LWC, concentration, and rain rate.
- **Parameter Input**: Allows input of simulation parameters, visualization style selection (LWC-colored scatterplot, size-colored scatterplot, droplet size histogram, vertical profile), and animation speed adjustment, with validation for physical constraints.
- **Logging and Output**: Logs initialization, simulation control, and plot updates using the `logging` module, with a console displaying real-time simulation status and parameters.

2 Simulation Logic

The simulation logic in `CloudMicroPhysicsWindow` manages cloud microphysics processes and visualization for a droplet population.

2.1 Initialization

- **Purpose**: Sets up the analysis window with synthetic droplet data and visualization.
- **Process**:
  - Configures simulation for 100 droplets with time step $\Delta t = 0.1\,\mathrm{s}$, total time $100\,\mathrm{s}$.
  - Initializes synthetic data: droplet radii ($\sim \mathcal{N}(10, 1)\,\mu\mathrm{m}$), concentrations ($100\,\mathrm{cm}^{-3}$), LWC, supersaturation (0.5%), heights ($0\,\mathrm{m}$).
  - Sets default parameters: updraft velocity ($1.0\,\mathrm{m/s}$), wind speed ($10.0\,\mathrm{m/s}$), wind angle (0°), drag coefficient (0.001), air temperature ($293.15\,\mathrm{K}$), autoconversion threshold ($20.0\,\mu\mathrm{m}$), wind shear ($0.01\,\mathrm{s}^{-1}$), aerosol concentration ($100\,\mathrm{cm}^{-3}$), soluble fraction (0.5).
  - Creates a control panel for parameter inputs, visualization style selection, animation speed slider, status label, and console.
  - Initializes a matplotlib figure with three subplots: scatterplot/histogram, histogram/vertical profile, time series.
  - Sets up arrays for droplet properties and lists for time steps, LWC, concentration, and rain rate histories.
  - Configures a timer for animation updates and initializes the plot and console.
  - Logs initialization and handles exceptions via `QMessageBox` and console.

2.2 Simulation Control

- **Purpose**: Manages simulation start, pause, resume, and stop operations.
- **Process (Start)**:
  - Validates inputs: $w \geq 0$, $0 \leq S \leq 0.01$, $r > 0$, $C > 0$, aerosol concentration > 0, $0 \leq f \leq 1$, $200 \leq T \leq 350\,\mathrm{K}$, $U \geq 0$, $r_{\text{auto}} > 0$, $s \geq 0$.
  - Updates parameters, resets droplet radii ($\sim \mathcal{N}(\text{initial radius}, 0.1 \times \text{initial radius})$), concentrations, LWC, heights, and histories.
  - Clears console, logs parameters, and starts timer with user-defined animation speed.
  - Disables start button, enables pause/stop buttons, sets status to "Running".

- **Process (Pause)**:
  - Stops timer and animation, sets `is_paused` to True.
  - Disables pause button, enables resume/stop buttons, sets status to ”Paused”.
- **Process (Resume)**:
  - Restarts animation and timer, sets `is_paused` to False.
  - Enables pause/stop buttons, disables resume button, sets status to ”Running”.
- **Process (Stop)**:
  - Stops timer and animation, resets `is_paused` and `anim`.
  - Enables start button, disables pause/resume/stop buttons, sets status to ”Stopped”.
- **Logging**: Logs actions and exceptions to file and console, displaying errors via `QMessageBox`.

2.3 Plot Update

- **Purpose**: Updates visualizations of droplet properties and their evolution.
- **Process**:
  - Checks if simulation should continue (`current_step` $<$ total_time$/\Delta t$).
  - Updates droplet properties via Köhler activation, condensation growth, evaporation, collision-coalescence, autoconversion, and wind shear effects.
  - Updates LWC, heights, supersaturation, and rain rate.
  - Computes wind stress for vector visualization.
  - Logs mean properties to console and updates status label.
  - Stores mean LWC, concentration, and rain rate for time series.
  - Updates plots: scatterplot/histogram (top-left), histogram/vertical profile (top-right), time series (bottom).
  - Increments simulation step and logs actions.

3 Physics and Mathematical Models

The `CloudMicroPhysicsWindow` class implements simplified models for cloud microphysics processes, simulating droplet activation, growth, and interactions.

3.1 Köhler Activation

- **Purpose**: Determines droplet activation based on aerosol properties.
- **Equations**:

$$
\begin{aligned}
A &= \frac{3.3 \times 10^{-5}}{T}, \\
B &= 4.3 \times 10^{-6} f, \\
r_c &= \sqrt{\frac{3B}{A}},
\end{aligned}
$$

  where A is the curvature term (μm), B is the solute term ($\mu\mathrm{m}^3$), T is air temperature (K), f is the soluble fraction, and r_c is the critical radius (μm). Droplets with $r > r_c$ have concentration set to $0.5\times$aerosol concentration cm^{-3}.

- **Implementation**: `compute_kohler_activation` updates concentrations.

3.2 Condensation Growth

- **Purpose**: Computes droplet radius growth due to condensation.
- **Equations**:

$$
\begin{aligned}
e_s &= 611.2 \exp\left(\frac{L_v}{R_v}\left(\frac{1}{273.15} - \frac{1}{T}\right)\right), \\
G &= \left(\frac{\rho_w L_v}{e_s T} + 1\right)^{-1}, \\
\frac{dr}{dt} &= \frac{G S f_v}{\max(r, 10^{-6})},
\end{aligned}
$$

  where e_s is saturation vapor pressure (Pa), $L_v = 2.5\times10^6\,\mathrm{J/kg}$, $R_v = 461.5\,\mathrm{J/kg/K}$, $\rho_w = 1000\,\mathrm{kg/m^3}$, S is supersaturation (fraction), $f_v = 1 + 0.1w$ is the ventilation factor with w as updraft velocity (m/s), and $\frac{dr}{dt}$ is clipped to $[-10^{-3}, 10^{-3}]\,\mu\mathrm{m/s}$.

- **Implementation**: `compute_condensation_growth` calculates radius growth rate.

3.3 Collision-Coalescence

- **Purpose**: Simulates stochastic droplet growth via collisions.

- **Equations**:

$$
\begin{aligned}
P_{\text{coll}} &= 0.01 \left(\frac{r}{r_{\text{auto}}} \right)^2, \\
r_{\text{new}} &= 1.2r \quad \text{(if collision occurs)}, \\
C_{\text{new}} &= 0.8C,
\end{aligned}
$$

  where P_{coll} is the collision probability (clipped to $[0, 0.1]$), r_{auto} is the autoconversion threshold (μm), and C is concentration (cm^{-3}).

- **Implementation**: `compute_collision_coalescence` updates radii and concentrations.

3.4 Autoconversion

- **Purpose**: Converts cloud water to rain based on droplet size.

- **Equations**:

$$
\begin{aligned}
a &= 0.001 \frac{\sum_{r_i > r_{\text{auto}}} (r_i - r_{\text{auto}})}{r_{\text{auto}}}, \\
R &= a \sum_{r_i > r_{\text{auto}}} \text{LWC}_i \frac{3600}{\rho_w}, \\
C_{\text{new}} &= C \exp(-a \Delta t),
\end{aligned}
$$

  where a is the autoconversion rate (s^{-1}), R is the rain rate (mm/hr), LWC_i is liquid water content (g/m^3), and $\Delta t = 0.1\,\text{s}$.

- **Implementation**: `compute_autoconversion` calculates rain rate and updates concentrations.

3.5 Evaporation

- **Purpose**: Computes droplet radius reduction due to evaporation.

- **Equation**:

$$
\frac{dr}{dt} = \frac{0.1S}{\max(r, 10^{-6})} \quad \text{(if } S < 0\text{)},
$$

  where $\frac{dr}{dt}$ is clipped to $[-10^{-3}, 0]\,\mu\text{m/s}$.

- **Implementation**: `compute_evaporation` calculates radius reduction for negative supersaturation.

3.6 Wind Shear Effect

- **Purpose**: Perturbs droplet concentrations due to wind shear.

- **Equation**:
$$C_{\text{new}} = C\left(1 + 0.1\frac{sr}{10}\right),$$
where s is wind shear (s^{-1}), and the factor is clipped to $[0.5, 1.5]$.

- **Implementation**: `compute_wind_shear_effect` updates concentrations.

3.7 Supersaturation Update

- **Purpose**: Updates supersaturation based on cooling and condensation.

- **Equations**:
$$\begin{aligned} e_s &= 611.2\exp\left(\frac{L_v}{R_v}\left(\frac{1}{273.15} - \frac{1}{T}\right)\right), \\ \frac{\partial T}{\partial t} &= -0.01w, \\ \frac{\partial S}{\partial t} &= -\frac{0.1\sum \text{LWC}_i}{\rho_w} + \frac{L_v}{c_p T}\frac{\partial T}{\partial t}, \end{aligned}$$
where $\frac{\partial T}{\partial t}$ is the cooling rate (K/s), $c_p = 1004\,\text{J/kg/K}$, and S is clipped to $[-0.01, 0.01]$.

- **Implementation**: `update_supersaturation` updates air temperature and supersaturation.

3.8 Wind Stress

- **Purpose**: Computes wind stress for vector visualization.

- **Equations**:
$$\begin{aligned} \theta &= \theta_0 + 0.1t, \\ \tau_x &= \rho_{\text{air}} C_d U^2 \cos(\theta), \\ \tau_y &= \rho_{\text{air}} C_d U^2 \sin(\theta), \end{aligned}$$
where θ_0 is the initial wind angle (radians), U is wind speed (m/s), $\rho_{\text{air}} = 1.225\,\text{kg/m}^3$, and $C_d = 0.001$.

- **Implementation**: `update_plot` calculates τ_x, τ_y for quiver plots.

4 Algorithms

4.1 Initialization Algorithm

- **Input**: None (uses synthetic data).

- **Steps**:

1. Log initialization to file and console.
2. Set parameters: 100 droplets, $\Delta t = 0.1\,\mathrm{s}$, total time $100\,\mathrm{s}$.
3. Initialize synthetic data: $r \sim \mathcal{N}(10, 1)\,\mu\mathrm{m}$, $C = 100\,\mathrm{cm}^{-3}$, LWC $= \frac{4}{3}\pi(r \times 10^{-6})^3 \times 10^6 C\,\mathrm{g/m^3}$, $S = 0.005$, $z = 0\,\mathrm{m}$.
4. Set defaults: $w = 1.0\,\mathrm{m/s}$, $U = 10.0\,\mathrm{m/s}$, $\theta = 0°$, $C_d = 0.001$, $T = 293.15\,\mathrm{K}$, $r_{\text{auto}} = 20.0\,\mu\mathrm{m}$, $s = 0.01\,\mathrm{s}^{-1}$, aerosol concentration $100\,\mathrm{cm}^{-3}$, soluble fraction 0.5.
5. Set window title to "Enhanced Cloud Microphysics Analysis" and size to 1000×700.
6. Create control panel with inputs, visualization selector ("Color by LWC", "Color by Droplet Size", "Droplet Size Histogram", "Vertical Profile"), speed slider (50–500 ms), status label, console.
7. Create start, pause, resume, stop buttons (pause/resume/stop initially disabled).
8. Initialize matplotlib figure with three subplots: scatterplot/histogram, histogram/vertical profile, time series.
9. Initialize arrays for droplet properties and lists for histories.
10. Set up timer, initialize plot with `update_plot(0)`, log to console.
11. Log completion or errors, displaying critical errors via `QMessageBox` and console.

4.2 Start Simulation Algorithm (`start_simulation`)

- **Input**: None (uses input fields).
- **Steps**:
  1. Log start to file and console.
  2. Validate inputs: $w \geq 0$, $0 \leq S \leq 0.01$, $r > 0$, $C > 0$, aerosol concentration > 0, $0 \leq f \leq 1$, $200 \leq T \leq 350\,\mathrm{K}$, $U \geq 0$, $r_{\text{auto}} > 0$, $s \geq 0$.
  3. Update parameters, reset: $r \sim \mathcal{N}(\text{initial radius}, 0.1 \times \text{initial radius})\,\mu\mathrm{m}$, $C =$ initial concentration cm^{-3}, LWC, $z = 0\,\mathrm{m}$, histories.
  4. Clear console, log parameters (updraft, supersaturation, radius, concentration, aerosol).
  5. Start timer with user-defined animation speed.

6. Disable start button, enable pause/stop buttons, set status to ”Running”.
7. Log completion or errors, displaying warnings or critical errors via `QMessageBox` and console.

4.3 Pause Simulation Algorithm (`pause_simulation`)

- **Input**: None.
- **Steps**:
  1. Log pause to file and console.
  2. Stop timer and animation, set `is_paused` to True.
  3. Disable pause button, enable resume/stop buttons, set status to ”Paused”.
  4. Log completion or errors, displaying critical errors via `QMessageBox` and console.

4.4 Resume Simulation Algorithm (`resume_simulation`)

- **Input**: None.
- **Steps**:
  1. Log resume to file and console.
  2. If `is_paused`, restart animation and timer with current speed.
  3. Set `is_paused` to False, enable pause/stop buttons, disable resume button, set status to ”Running”.
  4. Log completion or errors, displaying critical errors via `QMessageBox` and console.

4.5 Stop Simulation Algorithm (`stop_simulation`)

- **Input**: None.
- **Steps**:
  1. Log stop to file and console.
  2. Stop timer and animation, set `anim` to None and `is_paused` to False.
  3. Enable start button, disable pause/resume/stop buttons, set status to ”Stopped”.

4. Log completion or errors, displaying critical errors via `QMessageBox` and console.

4.6 Plot Update Algorithm (`update_plot`)

- **Input**: Frame number (tracks via `current_step`).
- **Steps**:
  1. Log update start at `current_step` to file and console.
  2. If `current_step` $\geq$ total_time/Δt, call `stop_simulation`.
  3. Update droplet properties:
     - Köhler activation: $C = 0.5 \times$ aerosol concentration if $r > r_c$.
     - Condensation growth: $\frac{dr}{dt} = \frac{GSf_v}{\max(r, 10^{-6})}$.
     - Evaporation: $\frac{dr}{dt} = \frac{0.1S}{\max(r, 10^{-6})} (S < 0)$.
     - Collision-coalescence: $r \to 1.2r$, $C \to 0.8C$ if $P_{\text{coll}} >$ random.
     - Autoconversion: $R = a \sum \text{LWC}_i \frac{3600}{\rho_w}$, $C \to C \exp(-a\Delta t)$.
     - Wind shear: $C \to C \left(1 + 0.1 \frac{sr}{10}\right)$.
  4. Update LWC: $\text{LWC} = \frac{4}{3}\pi(r \times 10^{-6})^3 \times 10^6 C \, \text{g/m}^3$.
  5. Update heights: $z \to z + w\Delta t$.
  6. Update supersaturation: $\frac{\partial S}{\partial t} = -\frac{0.1 \sum \text{LWC}_i}{\rho_w} + \frac{L_v}{c_p T} \frac{\partial T}{\partial t}$.
  7. Compute wind stress: $\tau_x = \rho_{\text{air}} C_d U^2 \cos(\theta)$, $\tau_y = \rho_{\text{air}} C_d U^2 \sin(\theta)$.
  8. Update status label and console with mean radius (μm), LWC (g/m^3), concentration (cm^{-3}), rain rate (mm/hr), supersaturation (%), height (m).
  9. Store mean LWC, concentration, rain rate for time series.
  10. Update plots:
      - Top-left: Scatterplot (r vs. C, colored by LWC or r) or histogram (r, bins=20, range=[0,100]).
      - Top-right: Histogram (r) or vertical profile (r vs. z, colored by LWC).
      - Bottom: Time series of mean LWC, concentration, rain rate.

11. Add wind stress vectors to scatterplot/vertical profile.

12. Increment `current_step`.

13. Log completion or errors, displaying warnings or critical errors via `QMessageBox` and console.

Oceanic Eddy & Front Simulation

June 29, 2025

1 Functionalities

The `OceanicEddyAndFrontWindow` and `EddySimulationWidget` classes provide core functionalities for simulating and visualizing oceanic eddies and fronts:

- **Initialization**: Sets up a simulation with a coarse grid (size 10–200) and synthetic eddy data (1–10 eddies, strength 0.5–2.0, radius 0.1–0.3), supporting non-hydrostatic dynamics ($< 1\,\mathrm{km}$ grid spacing), baroclinic/barotropic instabilities, mixed-layer instability (MLI, $< 10\,\mathrm{km}$), and potential vorticity (PV) frontogenesis.
- **Simulation Control**: Manages start, pause, reset, and export operations, with input validation for parameters like grid spacing (0.1–10.0 km), Coriolis parameter ($10^{-5}10^{-3}\,\mathrm{s}^{-1}$), and Rossby number (0.05–0.5).
- **Visualization**: Displays a 400x400 pixel image of vorticity, colored by vorticity magnitude and gradient, with optional overlays for vertical velocity, density, velocity shear, buoyancy, and PV. Fine grid patches (refinement factor 2–8) are shown for high-vorticity regions, and streamlines visualize flow.
- **Parameter Input**: Allows configuration of grid size, eddy properties, vorticity diffusion (0.005–0.02), rotation rate (0.05–0.2 $\mathrm{rad/s}$), background flow (0.02–0.1 $\mathrm{m/s}$), and instability parameters (e.g., baroclinic shear, mixed-layer depth).
- **Data Export**: Exports simulation data (vorticity, vertical velocity, density, shear, buoyancy, PV statistics) to a timestamped text file.

2 Simulation Logic

The simulation logic in `OceanicEddyAndFrontWindow` and `EddySimulationWidget` manages eddy and front dynamics and visualization.

2.1 Initialization

- **Purpose**: Configures the simulation with synthetic eddy data and fields.
- **Process**:
  - Initializes coarse grid ($N \times N$, $N = 10200$) over $[-1, 1] \times [-1, 1]$ with spacing $\Delta x = 2/N$.
  - Sets up 1–10 eddies with random centers ($x_0, y_0 \in [-0.8, 0.8]$), strengths ($0.81.2 \times$ input strength), and radii ($0.81.2 \times$ input radius).
  - Computes vorticity: $\zeta = \sum \zeta_i \exp(-r^2/(2R_i^2)) \cdot (1 - Ro \tanh(r/R_i))$, where $r = \sqrt{(x - x_0)^2 + (y - y_0)^2}$, R_i is eddy radius, and Ro is Rossby number.
  - Initializes non-hydrostatic fields (w, p') for grid spacing $< 1\,\mathrm{km}$, density (ρ) for baroclinic instability, velocity shear for barotropic instability, buoyancy for MLI ($< 10\,\mathrm{km}$), and PV for frontogenesis.
  - Creates fine grid patches (size floor$(N/5) \times$ refinement factor) for eddies with strength above threshold.
  - Sets up visualization with a 400x400 QImage, logging actions to `debug.log`.

2.2 Simulation Control

- **Purpose**: Manages simulation operations (start, pause, reset, export).
- **Process (Start)**:
  - Validates inputs: grid size (10–200), grid spacing (0.1–10.0 km), eddy strength (0.5–2.0), etc.
  - Initializes fields if not set, starts a 100 ms timer to call `update_simulation`.
  - Disables start button, enables pause/reset/export buttons, logs to `debug.log`.
- **Process (Pause)**:
  - Stops timer, enables start/reset/export buttons, disables pause button, logs to `debug.log`.
- **Process (Reset)**:
  - Stops timer, clears fields (vorticity, velocity, density, etc.), resets time and eddy data.
  - Enables reset/initialize buttons, disables start/pause/export, logs to `debug.log`.
- **Process (Export)**:

- Collects simulation data (time, grid size, eddy info, statistics) and writes to a timestamped file.
- Logs export action, displays success message via `QMessageBox`.

2.3 Visualization Update

- **Purpose**: Updates the visualization of eddy and front dynamics.
- **Process**:
  - Fills 400x400 QImage with white, normalizes vorticity (ζ) and its gradient magnitude ($|\nabla\zeta|$).
  - Colors pixels: red ($255\times$grad if grad > 0.7), green ($255\times(1-$vorticity$)(1-$grad$)$), blue ($255 \times$ vorticity$(1 -$ grad$)$).
  - Adjusts colors for non-hydrostatic (vertical velocity, pressure), baroclinic (density), barotropic (shear), MLI (buoyancy), and PV frontogenesis effects.
  - Visualizes fine grids with higher resolution, draws red rectangles around patches.
  - Draws 10 streamlines starting at random points, using induced velocities.
  - Updates widget via `paintEvent`.

3 Physics and Mathematical Models

The simulation implements models for oceanic eddy and front dynamics.

3.1 Vorticity Field

- **Purpose**: Models eddy vorticity with ageostrophic effects.
- **Equations**:

$$
\begin{aligned}
r &= \sqrt{(x - x_0)^2 + (y - y_0)^2}, \\
\zeta_i &= \begin{cases} \zeta_0(r/R_i), & r < R_i, \\ \zeta_0(R_i/r), & r \geq R_i, \end{cases} \\
\zeta &= \sum_i \zeta_i \exp\left(-\frac{r^2}{2R_i^2}\right)\left(1 - Ro\tanh\left(\frac{r}{R_i}\right)\right),
\end{aligned}
$$

where ζ_0 is eddy strength, R_i is radius, Ro is Rossby number, clipped to $[-10, 10]$.

- **Implementation**: `initialize_field` and `update_simulation` compute total vorticity.

3.2 Induced Velocity

- **Purpose**: Computes velocity field with Coriolis effects.
- **Equations**:

$$\begin{aligned} u_{\text{geo}} &= -\frac{\zeta_0(y-y_0)}{2\pi r^2}, \\ v_{\text{geo}} &= \frac{\zeta_0(x-x_0)}{2\pi r^2}, \\ u_{\text{ageo}} &= -Ro\, v_{\text{geo}}, \\ v_{\text{ageo}} &= Ro\, u_{\text{geo}}, \\ u &= \sum_i (u_{\text{geo}} + f u_{\text{ageo}}), \\ v &= \sum_i (v_{\text{geo}} + f v_{\text{ageo}}), \end{aligned}$$

  where f is the Coriolis parameter, $r^2 > 10^{-6}$.

- **Implementation**: `compute_induced_velocity` calculates velocities for streamlines and eddy motion.

3.3 Non-Hydrostatic Effects

- **Purpose**: Models vertical velocity and pressure perturbation for scales $< 1\,\text{km}$.
- **Equations**:

$$\begin{aligned} w &= -0.1\left(\frac{\partial u}{\partial x} + \frac{\partial v}{\partial y}\right), \\ p' &= -0.05\zeta, \end{aligned}$$

  where $\frac{\partial u}{\partial x}$, $\frac{\partial v}{\partial y}$ are computed via finite differences ($\Delta x = 2/N$).

- **Implementation**: `compute_nonhydrostatic_effects` updates fields.

3.4 Baroclinic Instability

- **Purpose**: Models vorticity tendency from density gradients.
- **Equations**:

$$\begin{aligned} \frac{\partial \zeta}{\partial t} &= -0.1\left(\frac{\partial \rho}{\partial x} + \frac{\partial \rho}{\partial y}\right), \\ \frac{\partial \rho}{\partial t} &= -\left(\frac{\partial \zeta}{\partial x}\frac{\partial \rho}{\partial x} + \frac{\partial \zeta}{\partial y}\frac{\partial \rho}{\partial y}\right), \end{aligned}$$

where $\rho \in [1020, 1030]\,\mathrm{kg/m^3}$, gradients use finite differences.

- **Implementation**: `compute_baroclinic_effects` updates vorticity and density.

3.5 Barotropic Instability

- **Purpose**: Models vorticity tendency from velocity shear.
- **Equations**:

$$\frac{\partial \zeta}{\partial t} = 0.05 \left(\frac{\partial s}{\partial x} - \frac{\partial s}{\partial y} \right),$$
$$\frac{\partial s}{\partial t} = - \left(\frac{\partial \zeta}{\partial x}\frac{\partial s}{\partial x} + \frac{\partial \zeta}{\partial y}\frac{\partial s}{\partial y} \right),$$

  where $s \in [-0.1, 0.1]$ is shear, clipped after updates.

- **Implementation**: `compute_barotropic_effects` updates vorticity and shear.

3.6 Mixed-Layer Instability

- **Purpose**: Models vorticity tendency from buoyancy gradients ($< 10\,\mathrm{km}$).
- **Equations**:

$$\frac{\partial \zeta}{\partial t} = 0.1h \left(\frac{\partial b}{\partial x} - \frac{\partial b}{\partial y} \right),$$
$$\frac{\partial b}{\partial t} = - \left(\frac{\partial \zeta}{\partial x}\frac{\partial b}{\partial x} + \frac{\partial \zeta}{\partial y}\frac{\partial b}{\partial y} \right),$$

  where $b \in [-0.1, 0.1]$ is buoyancy, h is mixed-layer depth.

- **Implementation**: `compute_mli_effects` updates vorticity and buoyancy.

3.7 Potential Vorticity Frontogenesis

- **Purpose**: Models PV and vorticity tendencies from strain.
- **Equations**:

$$\frac{\partial q}{\partial t} = -\sigma \left(\left(\frac{\partial q}{\partial x} \right)^2 + \left(\frac{\partial q}{\partial y} \right)^2 \right),$$
$$\frac{\partial \zeta}{\partial t} = 0.05 \frac{\partial q}{\partial t},$$
$$\frac{\partial q}{\partial t} = - \left(\frac{\partial \zeta}{\partial x}\frac{\partial q}{\partial x} + \frac{\partial \zeta}{\partial y}\frac{\partial q}{\partial y} \right),$$

  where $q \in [-1, 1]$ is PV, σ is strain rate.

- **Implementation**: `compute_pv_frontogenesis_effects` updates PV and vorticity.

4 Algorithms

4.1 Initialization Algorithm

- **Input**: Grid size (N), eddy strength, radius, number of eddies, refinement threshold, factor, Coriolis parameter, Rossby number, grid spacing, instability parameters.
- **Steps**:
  1. Log initialization to `debug.log`.
  2. Validate inputs: $N \in [10, 200]$, grid spacing $\in [0.1, 10]\,\mathrm{km}$, etc.
  3. Create grid: $x, y \in [-1, 1]$, $\Delta x = 2/N$, $X, Y = \text{meshgrid}(x, y)$.
  4. Initialize vorticity: $\zeta = 0$, size $N \times N$.
  5. For each eddy (1–10):
     - Set random center $(x_0, y_0) \in [-0.8, 0.8]$, strength ($0.81.2 \times$ input), radius ($0.81.2 \times$ input).
     - Compute $r = \sqrt{(X - x_0)^2 + (Y - y_0)^2}$, $\zeta_i = \zeta_0(r/R_i \text{ if } r < R_i, \text{ else } R_i/r)$.
     - Add $\zeta_i \exp(-r^2/(2R_i^2))(1 - Ro \tanh(r/R_i))$ to ζ.
  6. If non-hydrostatic (grid spacing $< 1\,\mathrm{km}$):
     - Initialize $w, p' = 0$, add $w = \zeta_0 0.1 \exp(-r^2/R_i^2) \sin(\arctan 2(Y - y_0, X - x_0))$, $p' = \zeta_0 0.05 \exp(-r^2/R_i^2)$.
  7. If baroclinic: Initialize $\rho = 1025 + sYN_s$, clip to $[1020, 1030]\,\mathrm{kg/m^3}$, add $\zeta_0 0.01 \exp(-r^2/(2R_i^2))$.
  8. If barotropic: Initialize shear $s = sY$, clip to $[-0.1, 0.1]$, add $\zeta_0 0.05 \exp(-r^2/(2R_i^2))$.
  9. If MLI (grid spacing $< 10\,\mathrm{km}$): Initialize $b = b_g Y + 0.01 \sin(2\pi X/0.2)$, clip to $[-0.1, 0.1]$, add $\zeta_0 0.02 \exp(-r^2/(2R_i^2))$.
  10. If PV frontogenesis: Initialize $q = \zeta + f(b/h)/N_s$ (if MLI), add $\zeta_0 f_{\text{pv}} \exp(-r^2/(2R_i^2))$, clip to $[-1, 1]$.
  11. For eddies with $|\zeta_0| >$ threshold: Create fine grid (floor$(N/5)\times$refinement factor), compute fine vorticity and fields.
  12. Call `update_visualization`, log completion or errors via `QMessageBox`.

4.2 Start Simulation Algorithm

- **Input**: Simulation parameters (vorticity diffusion, rotation rate, etc.).
- **Steps**:
  1. Log start to `debug.log`.
  2. If $\zeta = \text{None}$, call `initialize_simulation`.
  3. Validate inputs: vorticity diffusion (0.005–0.02), rotation rate (0.05–0.2 $\mathrm{rad/s}$), etc.
  4. Connect timer (100 ms) to `update_simulation`.
  5. Disable start button, enable pause/reset/export, log completion or errors.

4.3 Pause Simulation Algorithm

- **Input**: None.
- **Steps**:
  1. Log pause to `debug.log`.
  2. Stop timer, enable start/reset/export buttons, disable pause button.
  3. Log completion or errors via `QMessageBox`.

4.4 Reset Simulation Algorithm

- **Input**: None.
- **Steps**:
  1. Log reset to `debug.log`.
  2. Stop timer, clear fields ($\zeta, w, p', \rho, s, b, q$), reset time, eddy data, fine grids.
  3. Enable reset/initialize buttons, disable start/pause/export, clear QImage.
  4. Call `update`, log completion or errors via `QMessageBox`.

4.5 Update Simulation Algorithm

- **Input**: Simulation parameters.
- **Steps**:

1. Log update to debug.log.
2. Compute $\Delta t = \min(0.1, 0.5\Delta x / \max(|\text{background flow}|, 0.1))$, increment time.
3. Update eddy centers: $x_0 += (\text{background flow} + u_{\text{ind}})\Delta t$, $y_0 += v_{\text{ind}}\Delta t$, wrap if $x_0 > 1$.
4. Decay vorticity: $\zeta \to \zeta \exp(-\text{decay rate}\Delta t)$.
5. Compute new vorticity: $\zeta_{\text{new}} = \sum \zeta_i \exp(-r^2/(2R_i^2))(1 - Ro \tanh(r/R_i))$.
6. Update vorticity: $\zeta = \zeta_{\text{new}} + \nu\Delta t\nabla^2\zeta - \sigma\Delta t(x\frac{\partial\zeta}{\partial x} + y\frac{\partial\zeta}{\partial y}) + \text{background flow}(0.1Y)$, clip to $[-10, 10]$.
7. Apply instabilities:
   - Baroclinic: Add $\Delta t \cdot (-0.1(\frac{\partial\rho}{\partial x} + \frac{\partial\rho}{\partial y}))$, update ρ.
   - Barotropic: Add $\Delta t \cdot 0.05(\frac{\partial s}{\partial x} - \frac{\partial s}{\partial y})$, update s.
   - MLI: Add $\Delta t \cdot 0.1h(\frac{\partial b}{\partial x} - \frac{\partial b}{\partial y})$, update b.
   - PV frontogenesis: Add $\Delta t \cdot 0.05(-\sigma((\frac{\partial q}{\partial x})^2 + (\frac{\partial q}{\partial y})^2))$, update q.
8. If non-hydrostatic: Update w, p'.
9. Update fine grids, interpolate to coarse grid.
10. Call update_visualization, log completion or errors.

4.6 Export Simulation Algorithm

- **Input**: None.
- **Steps**:
  1. Log export to debug.log.
  2. Validate: If $\zeta =$ None, raise error.
  3. Collect data: time, grid size, eddy info, statistics ($\min, \max,$ mean, std) for vorticity, vertical velocity, density, shear, buoyancy, PV.
  4. Write to oceanic_eddies_output_YYYYMMDD_HHMM.txt.
  5. Log completion, show success via QMessageBox, or errors if failed.