```python
""" Utility module for solver class"""

import numpy as np
from typing import TYPE_CHECKING
from atmpy.infrastructure.enums import VariableIndices as VI
import logging

if TYPE_CHECKING:
    from atmpy.variables.variables import Variables
    from atmpy.grid.kgrid import Grid
    from atmpy.configuration.simulation_configuration import SimulationConfig
    from atmpy.physics.eos import ExnerBasedEOS  # Or your relevant EOS


MACHINE_EPSILON = np.finfo(float).eps


def calculate_dynamic_dt(
    variables: "Variables",
    grid: "Grid",
    config: "SimulationConfig",
    current_sim_time: float,
    next_output_time: float,
    current_step_number: int,
    # eos: "ExnerBasedEOS" # Pass EOS if needed for pressure/sound speed
) -> float:
    """
    Calculates an adaptive time step based on CFL conditions.

    Args:
        variables: The Variables object containing current solution state.
        grid: The Grid object.
        config: The SimulationConfig object.
        current_sim_time: The current simulation time.
        next_output_time: The time of the next scheduled output.
        current_step_number: The current simulation step number.
        # eos: The equation of state object.

    Returns:
        The calculated dynamic time step (dt).
    """
    gamma = config.global_constants.gamma

    # CFL number from configuration
    cfl_number = config.temporal.CFL

    # Mach number squared for sound speed scaling (if needed, depends on how
sound speed is calculated)
    Msq = config.model_regimes.Msq
    Minv = 1.0 / np.sqrt(Msq) if Msq > 0 else float("inf")

    # Get inner slice for calculations on physical domain
    inner_slice = grid.get_inner_slice()

    # --- Calculate local sound speed (c) ---
    # This depends heavily on your EOS and variable definitions.
    # Assuming ExnerBasedEOS and rhoY stores rho * Theta_nd (non-dim potential
temp)
    # P_exner_nd ~ rhoY (if scaled appropriately in your system)
    # c_physical = sqrt(gamma * R_gas * T_physical)
    # c_nondim = c_physical / u_ref
    # T_physical = (rhoY / rho) * T_ref (if rhoY/rho is Theta_nd)

    rho_inner = variables.cell_vars[inner_slice + (VI.RHO,)]
```

```
    rhoY_inner = variables.cell_vars[inner_slice + (VI.RHOY,)]

    # Avoid division by zero if rho is zero anywhere (e.g., uninitialized
ghosts)
    valid_rho_mask = rho_inner > MACHINE_EPSILON

    Y_prim_inner = np.zeros_like(rho_inner)
    Y_prim_inner[valid_rho_mask] = (
        rhoY_inner[valid_rho_mask] / rho_inner[valid_rho_mask]
    )

    # Physical Temperature: T_phys = Theta_nd * T_ref
    T_physical_inner = Y_prim_inner * config.global_constants.T_ref

    # Calculate primitive potential temperature (Theta_nd)
    # This assumes Y_prim is Theta_nd = Theta_physical / T_ref
    # If variables.primitives is not populated, calculate it:
    # variables.to_primitive(eos) # Need eos here
    # Y_prim_inner = variables.primitives[inner_slice + (PVI.Y,)]
    # If using conservative directly:
    Y_prim_inner = np.zeros_like(rho_inner)
    Y_prim_inner[valid_rho_mask] = (
        rhoY_inner[valid_rho_mask] / rho_inner[valid_rho_mask]
    )

    # Physical Temperature: T_phys = Theta_nd * T_ref
    T_physical_inner = Y_prim_inner * config.global_constants.T_ref

    # Physical sound speed: c_phys = sqrt(gamma * R_gas * T_phys)
    c_physical_inner = np.sqrt(
        gamma * config.global_constants.R_gas * T_physical_inner[valid_rho_mask]
    )

    # Non-dimensional sound speed: c_nd = c_phys / u_ref
    sound_speed_nd_inner = np.zeros_like(rho_inner)
    if config.global_constants.u_ref > MACHINE_EPSILON:
        sound_speed_nd_inner[valid_rho_mask] = (
            c_physical_inner / config.global_constants.u_ref
        )
    else:  # Should not happen in a well-defined problem
        sound_speed_nd_inner[valid_rho_mask] = float("inf")

    # --- Calculate flow speeds (u, v, w) ---
    # These are already non-dimensional in 'variables' if initialized correctly
    u_abs_inner = np.zeros_like(rho_inner)
    u_abs_inner[valid_rho_mask] = np.abs(
        variables.cell_vars[inner_slice + (VI.RHOU,)][valid_rho_mask]
        / rho_inner[valid_rho_mask]
    )

    v_abs_inner = np.zeros_like(rho_inner)
    if grid.ndim >= 2:
        v_abs_inner[valid_rho_mask] = np.abs(
            variables.cell_vars[inner_slice + (VI.RHOV,)][valid_rho_mask]
            / rho_inner[valid_rho_mask]
        )

    w_abs_inner = np.zeros_like(rho_inner)
    if grid.ndim >= 3:
        w_abs_inner[valid_rho_mask] = np.abs(
            variables.cell_vars[inner_slice + (VI.RHOW,)][valid_rho_mask]
            / rho_inner[valid_rho_mask]
        )
```

```
    # --- Determine max speeds ---
    # Maximize over the valid (physical) part of the domain
    max_u_abs = (
        np.max(u_abs_inner[valid_rho_mask])
        if np.any(valid_rho_mask)
        else MACHINE_EPSILON
    )
    max_v_abs = (
        np.max(v_abs_inner[valid_rho_mask])
        if grid.ndim >= 2 and np.any(valid_rho_mask)
        else MACHINE_EPSILON
    )
    max_w_abs = (
        np.max(w_abs_inner[valid_rho_mask])
        if grid.ndim >= 3 and np.any(valid_rho_mask)
        else MACHINE_EPSILON
    )

    max_sound_speed = (
        np.max(sound_speed_nd_inner[valid_rho_mask])
        if np.any(valid_rho_mask)
        else MACHINE_EPSILON
    )

    # --- Calculate CFL based on acoustic or advective speeds ---
    # config.temporal might need a new flag like 'use_acoustic_cfl'
    use_acoustic_cfl = getattr(
        config.temporal, "use_acoustic_cfl", True
    )  # Default to True

    if use_acoustic_cfl:
        max_char_speed_x = max_u_abs + max_sound_speed
        max_char_speed_y = max_v_abs + max_sound_speed
        max_char_speed_z = max_w_abs + max_sound_speed
    else:
        max_char_speed_x = max_u_abs
        max_char_speed_y = max_v_abs
        max_char_speed_z = max_w_abs

    # Ensure speeds are not zero to avoid division by zero for dt calculation
    max_char_speed_x = max(max_char_speed_x, MACHINE_EPSILON)
    max_char_speed_y = max(max_char_speed_y, MACHINE_EPSILON)
    max_char_speed_z = max(max_char_speed_z, MACHINE_EPSILON)

    dt_x = cfl_number * grid.dxyz[0] / max_char_speed_x
    dt_y = float("inf")
    if grid.ndim >= 2:
        dt_y = cfl_number * grid.dxyz[1] / max_char_speed_y
    dt_z = float("inf")
    if grid.ndim >= 3:
        dt_z = cfl_number * grid.dxyz[2] / max_char_speed_z

    dt_cfl = min(dt_x, dt_y, dt_z)
    dt = dt_cfl  # Default to CFL driven

    # --- Adjust dt to hit next_output_time precisely ---
    time_to_next_output = next_output_time - current_sim_time
    if time_to_next_output < MACHINE_EPSILON:  # Already at or past output time
        # This case should ideally be handled by the Solver's loop logic to not
take a step.
        # But if we must take a step, make it very small or the fixed dt.
        # For now, let's assume the solver handles the t >= tmax check.
        # If an output is missed, the next step will try to hit the *next*
next_output_time.
```

```
        pass
    elif dt > time_to_next_output > MACHINE_EPSILON:
        dt = time_to_next_output + MACHINE_EPSILON  # Ensure we step slightly
past

    return dt
```