```
"""This module contains different time integrators"""

from __future__ import annotations

import logging
import numpy as np

np.seterr(all="raise")
from typing import TYPE_CHECKING, Union, List, Any, Callable

from atmpy.boundary_conditions.bc_extra_operations import WallAdjustment
from atmpy.boundary_conditions.contexts import BCApplicationContext
from atmpy.infrastructure.utility import (
    directional_indices,
    one_element_inner_slice,
    one_element_inner_nodal_shape,
    dimension_directions, momentum_index,
)
from atmpy.pressure_solver.discrete_operations import AbstractDiscreteOperator
from atmpy.infrastructure.enums import VariableIndices as VI, Preconditioners
from atmpy.pressure_solver.utility import laplacian_inner_slice

if TYPE_CHECKING:
    from atmpy.flux.flux import Flux
    from atmpy.variables.variables import Variables
    from atmpy.variables.multiple_pressure_variables import MPV
    from atmpy.grid.kgrid import Grid
    from atmpy.boundary_conditions.boundary_manager import BoundaryManager
    from atmpy.physics.gravity import Gravity
    from atmpy.physics.thermodynamics import Thermodynamics
    from atmpy.time_integrators.coriolis import CoriolisOperator
    from atmpy.pressure_solver.abstract_pressure_solver import TPressureSolver

from atmpy.infrastructure.enums import AdvectionRoutines
from atmpy.pressure_solver.classical_pressure_solvers import
ClassicalPressureSolver

from atmpy.time_integrators.abstract_time_integrator import
AbstractTimeIntegrator
from atmpy.time_integrators.utility import *

from atmpy.infrastructure.enums import (
    BoundarySide as BdrySide,
    BoundaryConditions as BdryType,
)


class IMEXTimeIntegrator(AbstractTimeIntegrator):
    def __init__(
        self,
        grid: "Grid",
        variables: "Variables",
        mpv: "MPV",
        flux: "Flux",
        boundary_manager: "BoundaryManager",
        pressure_solver: "TPressureSolver",
        first_order_advection_routine: AdvectionRoutines,
        second_order_advection_routine: AdvectionRoutines,
        wind_speed: List[float],
        is_nongeostrophic: bool,
        is_nonhydrostatic: bool,
        is_compressible: bool,
        dt: float,
    ):
```

```
        # Inject dependencies
        self.grid: "Grid" = grid
        self.variables: "Variables" = variables
        self.mpv: "MPV" = mpv
        self.flux: "Flux" = flux
        self.pressure_solver: "TPressureSolver" = pressure_solver

        if not isinstance(self.pressure_solver, ClassicalPressureSolver):
            raise ValueError(
                "The current implementation of IMEXTimeIntegrator only supports
ClassicalPressureSolver."
            )

        self.boundary_manager: "BoundaryManager" = boundary_manager
        self.coriolis: "CoriolisOperator" = self.pressure_solver.coriolis
        self.gravity: "Gravity" = self.coriolis.gravity
        self.discrete_operator: "AbstractDiscreteOperator" = (
            self.pressure_solver.discrete_operator
        )
        self.advection_routine: Callable
        self._get_advection_routine(
            first_order_advection_routine, second_order_advection_routine
        )
        self.th: "Thermodynamics" = self.pressure_solver.th
        self.dt: float = dt
        self.Msq: float = self.pressure_solver.Msq
        self.is_nongeostrophic: bool = is_nongeostrophic
        self.is_nonhydrostatic: bool = is_nonhydrostatic
        self.is_compressible: bool = is_compressible
        self.wind_speed: np.ndarray = np.array(wind_speed)
        self.ndim = self.grid.ndim
        self.vertical_momentum_index =
self.coriolis.gravity.vertical_momentum_index

        # Helper for nodal boundary conditions
        # That is setting the flag is_nodal for all dimensions and all sides
(therefore ndim*2) to True
        # since p2_nodes is nodal.
        self._nodal_bc_contexts = (
            [BCApplicationContext(is_nodal=True)] * self.grid.ndim * 2
        )

    def _get_advection_routine(
        self,
        first_order_advection_routine_name: "AdvectionRoutines",
        second_order_advection_routine_name: "AdvectionRoutines",
    ) -> None:
        """Get the advection routine. The sole raison d'être of this method is
to avoid circular import
        issues from factory."""
        from atmpy.infrastructure.factory import get_advection_routines

        self.first_order_advection_routine = get_advection_routines(
            first_order_advection_routine_name
        )
        self.second_order_advection_routine = get_advection_routines(
            second_order_advection_routine_name
        )

    def _get_dimensional_sweep_order(self, global_time_step_num: int) ->
List[str]:
        """Determines the order of dimensional sweeps based on the global time
step."""
        base_directions = dimension_directions(self.grid.ndim)
```

```
        if self.grid.ndim == 1:
            return base_directions
        if global_time_step_num % 2 == 0:
            return base_directions
        else:
            return base_directions[::-1]

    def step(self, *args, **kwargs) -> None:  # Added global_time_step_num
        """
        Performs a single time step using the semi-implicit predictor-corrector
        method based on Benacchio & Klein (2019).
        """
        global_time_step_num = kwargs.pop("global_step")

        logging.info(
            f"--- Starting time step {global_time_step_num} with dt = {self.dt}
---"
        )

        # --- Save Initial State (t^n) ---
        initial_vars_arr = np.copy(self.variables.cell_vars)
        initial_p2_nodes_arr = np.copy(self.mpv.p2_nodes)

        ########################## Predictor Stage: Compute advective fluxes
(Pv)^{n+1/2} ################################
        # Sol^n and p^n are currently in self.variables and self.mpv
        self._predictor_step(self.dt, initial_vars_arr, global_time_step_num)
        # After this, self.flux contains (Pv)^{n+1/2}
        # self.variables contains Sol^{n+1/2} and self.mpv.p2_nodes contains
p^{n+1/2} (these are intermediate)

        ########################## Corrector Stage: Advance state from t^n to
t^{n+1} ###################################
        self._corrector_step(
            self.dt, initial_vars_arr, initial_p2_nodes_arr,
global_time_step_num
        )
        # After this, self.variables contains Sol^{n+1} and self.mpv.p2_nodes
contains p^{n+1}

        logging.info(f"--- Finished time step {global_time_step_num} ---")

    def _predictor_step(
        self,
        dt: float,
        initial_vars_arr_for_coeffs: np.ndarray,
        global_time_step_num: int,
    ) -> None:
        logging.debug(f"Predictor (step {global_time_step_num}): Starting")
        half_dt = 0.5 * dt

        # Current state (Sol^n, p^n) is in self.variables, self.mpv ###
        ################################## 1. Apply BCs to current state (t^n)
##########################################

self.boundary_manager.apply_boundary_on_all_sides(self.variables.cell_vars)
        self.boundary_manager.apply_boundary_on_single_var_all_sides(
            self.mpv.p2_nodes, self._nodal_bc_contexts
        )

        ######################## 2. Compute advective mass fluxes (P v)^n based
on state at t^n (Sol^n) ################
        # This updates self.flux[direction][..., VI.RHOY]
        self.flux.compute_averaging_fluxes()  # Uses self.variables (which is
```

```
Sol^n)
        logging.debug(
            f"Predictor (step {global_time_step_num}): Averaging fluxes computed
from Sol^n"
        )

        ###################### 3. Advect state by dt/2 using first-order splitting
-> Sol* ################################
        ######################################## (Eq. 14a of BK19)
############################################################
        # self.variables.cell_vars (Sol^n) becomes Sol#
        # # Advective mass fluxes in self.flux are based on Sol^n
        sweep_order = self._get_dimensional_sweep_order(global_time_step_num)
        self.first_order_advection_routine(
            self.grid,
            self.variables,  # Input: Sol^n, Output: Sol#
            self.flux,
            half_dt,
            sweep_order=sweep_order,
            boundary_manager=self.boundary_manager,
        )
        logging.debug(
            f"Predictor (step {global_time_step_num}): After first-order
advection (Sol^n -> Sol*)"
        )

        ###################### 4. Non-Advective Implicit Euler Substep for dt/2
########################################
        ######################################## (Eq. 15 of BK19)
############################################################
        # Applied to Sol# to get Sol^{n+1/2} and p^{n+1/2}
        self.backward_update_explicit(half_dt)  # Operates on self.variables
(Sol#)
        logging.debug(
            f"Predictor (step {global_time_step_num}): After
backward_update_explicit on Sol*"
        )

        # Solve for Exner pressure
        self.backward_update_implicit(half_dt, self.variables.cell_vars.copy())
        # self.variables is now Sol^{n+1/2}, self.mpv.p2_nodes is p^{n+1/2}
        logging.debug(
            f"Predictor (step {global_time_step_num}): After
backward_update_implicit (Sol* -> Sol^(n+1/2))"
        )

        ######################### 5. Compute predicted advective mass fluxes
(Pv)^{n+1/2} ##############################
        # self.variables.cell_vars is now Sol^{n+1/2}
        self.flux.compute_averaging_fluxes()  # Uses self.variables
(Sol^{n+1/2})
        # This updates self.flux[... ,VI.RHOY] to represent (Pv)^{n+1/2} for the
corrector
        logging.debug(
            f"Predictor (step {global_time_step_num}): Averaging fluxes
(Pv)^(n+1/2) computed from Sol^(n+1/2)"
        )
        logging.debug(f"--- Predictor (step {global_time_step_num}): Finished
---")

    def _corrector_step(
        self,
        dt: float,
        initial_vars_arr: np.ndarray,
```

```
        initial_p2_nodes_arr: np.ndarray,
        global_time_step_num: int,
    ) -> None:
        logging.debug(f"Corrector (step {global_time_step_num}): Starting")
        half_dt = 0.5 * dt

        ############################## 1. Reset state variables to t^n
##################################################
        self.variables.cell_vars[...] = initial_vars_arr
        self.mpv.p2_nodes[...] = initial_p2_nodes_arr
        logging.debug(f"Corrector (step {global_time_step_num}): State reset to
t^n")

        ############################## 2. Apply BCs to state at t^n
########################################################

self.boundary_manager.apply_boundary_on_all_sides(self.variables.cell_vars)
        self.boundary_manager.apply_boundary_on_single_var_all_sides(
            self.mpv.p2_nodes, self._nodal_bc_contexts
        )

        ############################ 3. Explicit Euler for non-advective terms
############################################
        ###################################### (Eq. 17a of BK19)
############################################################
        # Based on state t^n (Sol^n). Result is Sol*
        self.forward_update(half_dt)  # Operates on self.variables (Sol^n ->
Sol*)
        # self.mpv.p2_nodes is still p^n
        logging.debug(
            f"Corrector (step {global_time_step_num}): After forward_update
(Sol^n -> Sol*)"
        )

        ########################### 4. Advection using Strang Splitting
################################################
        ################################### (Full Step, Eq.17b of BK19)
################################################
        # Uses the advective mass fluxes in self.flux[..., RHOY] (which are
(Pv)^{n+1/2})
        # self.variables.cell_vars (Sol*) becomes Sol**
        sweep_order = self._get_dimensional_sweep_order(global_time_step_num)
        self.second_order_advection_routine(
            self.grid,
            self.variables,  # Input: Sol*, Output: Sol**
            self.flux,  # Contains (Pv)^{n+1/2}
            dt,  # Full dt for this advection operation
            sweep_order=sweep_order,
            boundary_manager=self.boundary_manager,
        )
        logging.debug(
            f"Corrector (step {global_time_step_num}): After Strang-split
advection (Sol* -> Sol**)"
        )

        ################################## 5. Implicit Euler substep for dt/2
############################################
        ############################################## (Eq. 17c of BK19)
################################################
        # Applied to Sol** to get Sol^{n+1}
        self.backward_update_explicit(half_dt)  # Operates on self.variables
(Sol**)
        logging.debug(
            f"Corrector (step {global_time_step_num}): After
```

```
backward_update_explicit on Sol**"
        )

        # Operator coefficients (P0) for implicit solve use current state
(Sol**)
        self.backward_update_implicit(
            half_dt, initial_vars=self.variables.cell_vars.copy()
        )  # Pass a copy if it's modified
        # self.variables is now Sol^{n+1}, self.mpv.p2_nodes is p^{n+1}
        logging.debug(
            f"Corrector (step {global_time_step_num}): After
backward_update_implicit (Sol** -> Sol^(n+1))"
        )

        ########################## 6. Final Boundary Conditions
############################################################

self.boundary_manager.apply_boundary_on_all_sides(self.variables.cell_vars)
        self.boundary_manager.apply_boundary_on_single_var_all_sides(
            self.mpv.p2_nodes, self._nodal_bc_contexts
        )
        logging.debug(f"--- Corrector (step {global_time_step_num}): Finished
---")

    def forward_update(self, dt: float) -> None:
        """Integrates the problem on time step using the explicit euler in the
NON-ADVECTIVE stage. This means
        that the algorithm does not use the half-time advective values to
compute the updates. Rather, we compute
        directly from the full-variables in the main euler equation."""
        g = self.gravity.strength  # The gravity strength
        cellvars = self.variables.cell_vars  # Temp variable for cell variables
        p2n = np.copy(self.mpv.p2_nodes)  # Temp allocation for p2_nodes

        # Calculate the buoyancy PX'
        dbuoy = cellvars[..., VI.RHOY] * cellvars[..., VI.RHOX] / cellvars[...,
VI.RHO]
        ######################## Update variables
        self._forward_momenta_update(cellvars, dt, p2n, dbuoy, g)
        self._forward_buoyancy_update(cellvars, dt)
        self._forward_pressure_update(cellvars, dt, p2n)

        ######################## Update boundary values
        # Update the boundary nodes for pressure variable
        self.boundary_manager.apply_boundary_on_single_var_all_sides(
            self.mpv.p2_nodes, self._nodal_bc_contexts
        )

        # Update all other variables on the boundary.
        self.boundary_manager.apply_boundary_on_all_sides(cellvars)

    def backward_update_explicit(self, dt: float):
        """This is the first part of implicit Euler update. This method does the
job to calculate the terms involving the
        n-th timestep in the implicit scheme. The method
backward_implicit_update involves the terms evaluated at the
        (n+1)-th timestep in the implicit scheme.

        Parameters
        ----------
        dt: float
            The time step used for the update at this stage. It could be a full
global step or half global step.
        """
```

```
        cellvars = self.variables.cell_vars

        # First calculate the extra explicit buoyancy term that is not
calculated in the coriolis matrix inversion:
        bouyoncy = cellvars[..., VI.RHOY] * (
            cellvars[..., VI.RHOX] / cellvars[..., VI.RHO]
        )

        # Update the corresponding vertical momentum explicitly
        g = self.gravity.strength

        if self.is_nonhydrostatic:
            cellvars[..., self.vertical_momentum_index] -= dt * (
                (g / self.Msq) * bouyoncy
            )

        # Remove background wind
        self.variables.adjust_background_wind(
            self.wind_speed, scale=-1.0, in_place=True
        )

        # Apply the solver inverse matrix (Matrix combining the switches, the
coriolis force and the singular buoyancy term)
        self.coriolis.apply_inverse(
            cellvars[..., VI.RHOU],
            cellvars[..., VI.RHOV],
            cellvars[..., VI.RHOW],
            self.variables,
            self.mpv,
            self.is_nongeostrophic,
            self.is_nonhydrostatic,
            self.Msq,
            dt,
        )

        # Restore background wind
        self.variables.adjust_background_wind(self.wind_speed, scale=1.0,
in_place=True)

        # Update all other variables on the boundary.
        self.boundary_manager.apply_boundary_on_all_sides(cellvars)

    def backward_update_implicit(self, dt: float, initial_vars: np.ndarray =
None):
        """Compute the one step of the implicit part of the Euler backward
scheme. This is a part of the BK19 algorithm.
        Notice before the call to this method, the coefficient variables must be
created at half timestep. Then going back
        to the initial variables, we start anew to advance the time stepping in
a implicit trapezoidal rule

        Parameters
        ----------
        dt: float
            The time step used for this update. It could be a full global step
or half global step.
        initial_vars: np.ndarray
            The initial variables of the problem before any changes were made by
calculation of half-time step.
        """
        cellvars = self.variables.cell_vars if initial_vars is None else
initial_vars
```

```
        ################################## 1. Preparation
################################################################
        # Update the boundary value from current variables/initial variable and
compute the pressure coefficients
        # The coefficient values depend on the current values in the variables.
Therefore, they are pre update coefficients
        self.boundary_manager.apply_boundary_on_all_sides(cellvars)
        self.pressure_solver.pressure_coefficients_nodes(cellvars, dt)

        ################################## 2. Apply boundary conditions on pi'
########################################
        # Update the boundary nodes for pressure variable
        self.boundary_manager.apply_boundary_on_single_var_all_sides(
            self.mpv.p2_nodes, self._nodal_bc_contexts
        )

        ################################## 3. "Pre-Correction" using Current
Pressure p^k ###############################
        # This method will put M_inv·(dt*C0*∇p₂)/0 in the momenta container,
where M is extended coriolis inverse.
        # In a couple of steps the divergence will be applied on [Pu, Pv, Pw],
using these values.
        # The buoyancy term is NOT adjusted in this preparatory step
(updt_chi=0.0). Only the momenta is needed for div.
        CHI_UPDT_VALUE = 0.0
        self.pressure_solver.apply_pressure_gradient_update(
            p=self.mpv.p2_nodes,  # Use current pressure
            updt_chi=CHI_UPDT_VALUE,  # Don't adjust buoyancy here
            dt=dt,
            is_nongeostrophic=self.is_nongeostrophic,
            is_nonhydrostatic=self.is_nonhydrostatic,
        )

        ############################### 4. Apply BCs to Pre-Corrected State
################################################
        # TODO: Check if necessary

self.boundary_manager.apply_boundary_on_all_sides(self.variables.cell_vars)

        ############################### 5. Compute RHS (Divergence of Pre-
Corrected Momenta) ##############################
        # Get the pre-corrected variables
        cellvars = self.variables.cell_vars

        # Calculate pressure-weighted momenta P*v = (rhoY/rho) * rho*v
        # [Pu, Pv, Pw]
        pressure_weighted_momenta =
self._calculate_enthalpy_weighted_momenta(cellvars)

        # Calculate divergence on one element inner nodes [shape: (nx-1, ny-1,
nz-1)]
        # Since the current momenta contain the pressure gradient update, this
divergence
        # is basically ∇·(M_inv·(dt*(P0)*∇p₂)) where M is extended coriolis
inverse
        divergence_inner =
self.discrete_operator.divergence(pressure_weighted_momenta)

        laplacian_inner_node_slice = laplacian_inner_slice(self.grid.ng)
        # Final RHS for A * delta_p = rhs
        rhs_flat = divergence_inner[laplacian_inner_node_slice].flatten()

        # Adjust the coefficients of the pressure gradient term for the solver
as a result of compressibility regime
```

```
        self.mpv.wcenter *= self.is_compressible

        ############################## 6. Solve the elliptic helmholtz equation
##########################################
        rtol = 1.0e-11
        p2_inner_flat, solver_info = self.pressure_solver.solve_helmholtz(
            rhs_flat,
            dt,
            self.is_nongeostrophic,
            self.is_nonhydrostatic,
            self.is_compressible,
            rtol=rtol,
            # Add tol/max_iter if needed, e.g., tol=1e-7
        )

        ############################## 7. Prepare p2 for Correction Step
################################################
        inner_slice = self.grid.get_inner_slice()
        inner_shape = self.grid.inshape
        p_unflat = p2_inner_flat.reshape(inner_shape)

        # Pad increment to full nodal shape
        p2_full = np.zeros_like(self.mpv.p2_nodes)
        p2_full[inner_slice] = p_unflat

        ############################## 8. Update boundary with the new values
##########################################
        # Use the current existing context (is_nodal for all sides)
        self.boundary_manager.apply_boundary_on_single_var_all_sides(
            p2_full, self._nodal_bc_contexts
        )

        ############################## 9. Final Correction using Pressure
Increment delta_p ##############################
        # Apply the update using the *solved increment*.
        # The buoyancy *is* adjusted now (updt_chi=1.0).
        CHI_UPDT_VALUE = 1.0
        self.pressure_solver.apply_pressure_gradient_update(
            p=p2_full,  # Use solved increment
            updt_chi=CHI_UPDT_VALUE,  # Adjust buoyancy now
            dt=dt,
            is_nongeostrophic=self.is_nongeostrophic,
            is_nonhydrostatic=self.is_nonhydrostatic,
        )

        ############################## 10. Update The Main Exner Pressure
################################################
        self.mpv.p2_nodes += p2_full

        ############################## 11. Apply Final Boundary Conditions
################################################

self.boundary_manager.apply_boundary_on_all_sides(self.variables.cell_vars)

        # Use the current existing contexts (is_nodal for all sides)
        self.boundary_manager.apply_boundary_on_single_var_all_sides(
            self.mpv.p2_nodes, self._nodal_bc_contexts
        )

    def _forward_momenta_update(
        self,
        cellvars: np.ndarray,
        dt: float,
        p2n: np.ndarray,
```

```
        dbuoy: np.ndarray,
        g: float,
    ):
        """Update the momenta

        Parameters
        ----------
        cellvars: np.ndarray
            The cell variables
        p2n: np.ndarray
            The nodal pressure variables
        dbuoy : np.ndarray
            The pressured perturbation of Chi: PX'
        g: float
            The gravity strength
        """
        coriolis = self.coriolis.strength
        adjusted_momenta =
self.variables.adjust_background_wind(self.wind_speed, -1.0)

        # pressure gradient factor: (P/Gamma)
        rhoYovG = self.pressure_solver._calculate_P_over_Gamma(cellvars)

        # Calculate the Exner pressure perturbation (Pi^prime) gradiant (RHS of
the momenta equations)
        dpdx, dpdy, dpdz = self.discrete_operator.gradient(p2n)


################################################################################
################################
        ## UPDATING VARIABLES

################################################################################
################################
        # Updates: First the shared terms without regarding which one is in the
gravity direction
        # Horizontal momentum in x
        cellvars[..., VI.RHOU] -= dt * (
            rhoYovG * dpdx
            - coriolis[2] * adjusted_momenta[..., 1]
            + coriolis[1] * adjusted_momenta[..., 2]
        )
        if self.grid.ndim >= 2:
            cellvars[..., VI.RHOV] -= dt * (
                rhoYovG * dpdy
                - coriolis[0] * adjusted_momenta[..., 2]
                + coriolis[2] * adjusted_momenta[..., 0]
            )
        if self.grid.ndim == 3:
            cellvars[..., VI.RHOW] -= dt * (
                rhoYovG * dpdy
                - coriolis[1] * adjusted_momenta[..., 0]
                + coriolis[0] * adjusted_momenta[..., 1]
            )

        # Updates: The momentum in the direction of gravity
        # Find vertical vs horizontal velocities:
        cellvars[..., self.pressure_solver.vertical_momentum_index] -= dt * (
            (g / self.Msq) * dbuoy * self.is_nonhydrostatic
        )

    def _forward_pressure_update(
        self, cellvars: np.ndarray, dt: float, p2n: np.ndarray
    ):
```

```
        """Update the Exner pressure."""

        # Compute the divergence of the pressure-weighted momenta: (Pu)_x +
(Pv)_y + (Pw)_z where
        # # P = rho*Y = rho*Theta
        pressure_weighted_momenta = self._calculate_enthalpy_weighted_momenta(
            self.variables.cell_vars
        )

        inner_slice = one_element_inner_slice(self.grid.ndim, full=False)
        self.mpv.rhs[inner_slice] = self.discrete_operator.divergence(
            pressure_weighted_momenta,
        )

        # Adjust wall boundary nodes (scale). Notice the side is set to be
BdrySide.ALL.
        # This will apply the 'extra' method whenever the boundary is defined to
be WALL.
        boundary_operation = [
            WallAdjustment(
                target_side=BdrySide.ALL, target_type=BdryType.WALL, factor=2.0
            )
        ]
        self.boundary_manager.apply_extra_all_sides(self.mpv.rhs,
boundary_operation)

        # Calculate the derivative of the Exner pressure with respect to P
        dpidP = calculate_dpi_dp(cellvars[..., VI.RHOY], self.Msq)


        # Create a nodal variable to store the intermediate updates
        dp2n = np.zeros_like(p2n)
        dp2n[inner_slice] -= dt * dpidP * self.mpv.rhs[inner_slice]
        self.mpv.p2_nodes[...] += self.is_compressible * dp2n

    def _forward_buoyancy_update(self, cellvars: np.ndarray, dt: float):
        """Update the X' variable (rho X in the variables)

        Parameters
        ----------
        cellvars: np.ndarray
           The cell variables
        dt : float
            The time step
        """
        # get Chi variable and the derivative
        S0c = self.mpv.get_S0c_on_cells()
        dSdy = self.mpv.compute_dS_on_nodes()

        # Intermediate variable for current Chi
        currentX = cellvars[..., VI.RHO] * (
            (cellvars[..., VI.RHO] / cellvars[..., VI.RHOY]) - S0c
        )  # Calculate the perturbation of Chi, then multiply with density rho


################################################################################
##############################
        # Update the variable

################################################################################
##############################
        cellvars[..., VI.RHOX] = (
            currentX
            - dt
```

```
            * cellvars[..., self.pressure_solver.vertical_momentum_index]
            * dSdy
            * cellvars[..., VI.RHO]
        )

    def _calculate_enthalpy_weighted_momenta(self, cellvars: np.ndarray):
        """Calculate the vector [Pu, [Pv], [Pw]] which is equal to
rho*Theta*velocities. This is needed in multiple
        parts of the code"""
        Y = cellvars[..., VI.RHOY] / cellvars[..., VI.RHO]
        momenta_indices = [VI.RHOU, VI.RHOV, VI.RHOW][: self.grid.ndim]
        return cellvars[..., momenta_indices] * Y[..., np.newaxis]

    def get_dt(self):
        pass


def example_usage():
    from atmpy.physics.thermodynamics import Thermodynamics
    from atmpy.grid.utility import DimensionSpec, create_grid
    from atmpy.variables.variables import Variables
    from atmpy.infrastructure.enums import (
        VariableIndices as VI,
        HydrostateIndices as HI,
    )
    from atmpy.boundary_conditions.bc_extra_operations import (
        WallAdjustment,
        PeriodicAdjustment,
    )

    np.set_printoptions(linewidth=300)
    np.set_printoptions(suppress=True)
    np.set_printoptions(precision=10)


########################################################################################
####################################
    ############################## CHOOSE TEST CASE
########################################################################

########################################################################################
####################################
    from atmpy.test_cases.traveling_vortex import TravelingVortex

    case = TravelingVortex()

    grid = case.config.grid
    gravity_vec = case.config.physics.gravity_strength
    Msq = case.config.model_regimes.Msq

    th = Thermodynamics()


########################################################################################
####################################
    ######### MPV
########################################################################################
######################

########################################################################################
####################################
    from atmpy.variables.multiple_pressure_variables import MPV

    mpv = MPV(grid)
```

```
################################################################################
####################################
    ######### Variables
################################################################################
################

################################################################################
####################################

    variables = Variables(grid, 6, 1)

    # Initialize variables
    case.initialize_solution(variables, mpv)


################################################################################
####################################
    ######### BOUNDARY MANAGER
################################################################################
#########

################################################################################
####################################
    from atmpy.boundary_conditions.boundary_manager import BoundaryManager

    bm_config = case.config.get_boundary_manager_config()
    manager = BoundaryManager(bm_config)
    manager.apply_boundary_on_all_sides(variables.cell_vars)

################################################################################
####################################
    ######### FLUX
################################################################################
####################

################################################################################
####################################
    from atmpy.physics.eos import ExnerBasedEOS
    from atmpy.flux.flux import Flux

    eos = ExnerBasedEOS()
    flux = Flux(grid, variables, eos)

    ########## STRATIFICATION
################################################################################
##########
    stratification = case.config.physics.stratification


################################################################################
####################################
    ########## DISCRETE OPERATOR AND PRESSURE SOLVER
#########################################################################

################################################################################
####################################
    from atmpy.infrastructure.enums import (
        PressureSolvers,
        DiscreteOperators,
        LinearSolvers,
    )
    from atmpy.pressure_solver.contexts import (
```

```
        DiscreteOperatorsContext,
        PressureContext,
    )

    coriolis = case.config.physics.coriolis

    op_context = DiscreteOperatorsContext(
        operator_type=DiscreteOperators.CLASSIC_OPERATOR, grid=grid
    )
    linear_solver = LinearSolvers.BICGSTAB

    # Instantiate the pressure solver context by specifying enums for pressure
solver and discrete operator.
    ps_context: PressureContext[ClassicalPressureSolver] = PressureContext(
        solver_type=PressureSolvers.CLASSIC_PRESSURE_SOLVER,
        op_context=op_context,
        linear_solver_type=linear_solver,
        precondition_type=Preconditioners.DIAGONAL,
        extra_dependencies={
            "grid": grid,
            "variables": variables,
            "mpv": mpv,
            "boundary_manager": manager,
            "coriolis": coriolis,
            "Msq": Msq,
            "thermodynamics": th,
        },
    )

    pressure = ps_context.instantiate()


################################################################################
####################################
    ######## TIME INTEGRATION
################################################################################
##########

################################################################################
####################################

    ##### Approach 1 ##########################################################
    ### Using instantiation context for central instantiation of all classes:
    from atmpy.time_integrators.contexts import TimeIntegratorContext
    from atmpy.infrastructure.enums import TimeIntegrators

    dt = 0.01
    context: TimeIntegratorContext[IMEXTimeIntegrator] = TimeIntegratorContext(
        integrator_type=TimeIntegrators.IMEX,
        grid=grid,
        variables=variables,
        flux=flux,
        boundary_manager=manager,
        advection_routine=AdvectionRoutines.STRANG_SPLIT,
        dt=dt,
        extra_dependencies={
            "mpv": mpv,
            "pressure_solver": pressure,
            "wind_speed": [0.0, 0.0, 0.0],  # optional: override default wind
speed
            "is_nonhydrostatic": True,
            "is_nongeostrophic": True,
            "is_compressible": True,
        },
```

```
    )
    time_integrator = context.instantiate()

    print(variables.cell_vars[..., VI.RHO])

    print(mpv.p2_nodes)
    time_integrator.step()
    print(mpv.p2_nodes)


if __name__ == "__main__":
    example_usage()
```