```
"""This module handles solving the equation for the pressure variable including
the Laplace/Poisson equation."""

import numpy as np
import scipy as sp
from typing import TYPE_CHECKING, Union, Tuple, Optional, Callable, Dict

from atmpy.infrastructure.enums import (
    VariableIndices as VI,
    BoundarySide as BdrySide,
    BoundaryConditions as BdryType,
)
from atmpy.boundary_conditions.bc_extra_operations import WallAdjustment
from atmpy.infrastructure.utility import (
    one_element_inner_slice,
    one_element_inner_nodal_shape,
)
from atmpy.physics.thermodynamics import Thermodynamics
from atmpy.pressure_solver import preconditioners
from atmpy.pressure_solver.abstract_pressure_solver import
AbstractPressureSolver
from atmpy.pressure_solver.preconditioners import *
from atmpy.pressure_solver.utility import laplacian_inner_slice

if TYPE_CHECKING:
    from atmpy.pressure_solver.discrete_operations import
AbstractDiscreteOperator
    from atmpy.pressure_solver.linear_solvers import ILinearSolver
    from atmpy.variables.variables import Variables
    from atmpy.variables.multiple_pressure_variables import MPV
    from atmpy.boundary_conditions.boundary_manager import BoundaryManager
    from atmpy.time_integrators.coriolis import CoriolisOperator
    from atmpy.physics.thermodynamics import Thermodynamics
    from atmpy.grid.kgrid import Grid
    from atmpy.infrastructure.enums import Preconditioners


class ClassicalPressureSolver(AbstractPressureSolver):
    """
    PressureSolver encapsulates the pressure correction procedure.
    It assembles the operator for the pressure correction (using, for example,
    a discrete Laplacian), builds the right–hand side from the divergence and
other
    source terms, and then solves the resulting linear system via an injected
linear solver.

    After obtaining the pressure correction, it updates the pressure (or p2-
like)
    diagnostic in the Variables object. In a complete implementation the solver
would
    also update ghost node values via boundary routines.
    """

    def __init__(
        self,
        discrete_operator: "AbstractDiscreteOperator",
        linear_solver: "ILinearSolver",
        precondition_type: "Preconditioners",
        grid: "Grid",
        variables: "Variables",
        mpv: "MPV",
        boundary_manager: "BoundaryManager",
        coriolis: "CoriolisOperator",
        thermodynamics: "Thermodynamics",
```

```
        Msq: float,
    ):
        super().__init__(
            discrete_operator,
            linear_solver,
            precondition_type,
            coriolis,
            thermodynamics,
            Msq,
        )
        self.grid = grid
        self.variables: "Variables" = variables
        self.mpv: "MPV" = mpv
        self.boundary_manager: "BoundaryManager" = boundary_manager
        self.ndim = self.variables.ndim
        self.vertical_momentum_index: int = (
            self.coriolis.gravity.vertical_momentum_index
        )
        self.precondition_type: "Preconditioners" = precondition_type
        self.precon_compute: Optional[Callable] = (
            self._get_preconditioner_compute_components()
        )
        self.precon_apply_inverse: Optional[Callable] = (
            self._get_preconditioner_apply_inverse()
        )
        self.precon_data: Optional[Dict[str, Any]] = None  # Initialize

        if self.precondition_type is None:
            raise ValueError("The preconditioner type must be specified")

    def _get_preconditioner_compute_components(self):
        """Get the precondtioning compute function. The sole raison d'être of
this method is to avoid circular import
        issues from factory."""
        from atmpy.infrastructure.factory import get_preconditioner_components

        return get_preconditioner_components(self.precondition_type)

    def _get_preconditioner_apply_inverse(self):
        """Get the precondtioning function. The sole raison d'être of this
method is to avoid circular import
        issues from factory."""
        from atmpy.infrastructure.factory import get_preconditioner

        return get_preconditioner(self.precondition_type)

    def _compute_and_store_precondition_data(
        self,
        dt: float,
        is_nongeostrophic: bool,
        is_nonhydrostatic: bool,
        is_compressible: bool,
    ) -> None:
        """
        Computes the necessary data for the selected preconditioner based on the
        current state and stores it in self.precon_data.
        """

        self.precon_data = self.precon_compute(
            self, dt, is_nongeostrophic, is_nonhydrostatic, is_compressible
        )

    def pressure_coefficients_nodes(self, cellvars: np.ndarray, dt: float):
        """Calculate the coefficients for the pressure equation. Notice the
```

```
coefficients are nodal.

        Basically it calculates there are two sets of coefficients needed to be
calculated:
        1. alpha_h*(P*Theta) in Momentum equations (Coefficient of pressure term
in Helmholtz equation)
        2. alpha_p*(dP/dpi) in Pressure equation   (Coefficient of pressure term
in Momentum equation)

        The first will be stored in mpv.wplus
        The second will be stored in mpv.wcenter

        Parameters
        ----------
        cellvars: np.ndarray
            The array of variables on cells.
        dt: float
            The time step.

        Notes
        -----
        1. A variable container is passed to the method to decouple it from the
attribute variable to be able to use the
            method on the initial variables in the trapezoidal rule.

        2. A difference in calculation of alpha_p*(dP/dpi):
            it considers that the alpha_p = 1, since for incompressible case, we
handle the case elsewhere in the
            time update.
        """

        ###################### Calculate the coefficients
##################################################################
        pTheta = self._calculate_coefficient_pTheta(cellvars)  # Cell-centered
        dPdpi = self._calculate_coefficient_dPdpi(cellvars, dt)  # Node-centered

        ######### Fill wplux and wcenter containers with the corresponding
values above ################################
        for dim in range(self.ndim):
            self.mpv.wplus[dim][...] = pTheta

        inner_slice = one_element_inner_slice(self.ndim, full=False)
        self.mpv.wcenter[inner_slice] = dPdpi

        ###################### Update the boundary nodes for the dP/dpi container.
#######################################
        # Create the operation context to scale down the nodes. 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=0.5
            )
        ]
        self.boundary_manager.apply_extra_all_sides(
            self.mpv.wcenter, boundary_operation, target_mpv=False
        )

    def calculate_enthalpy_weighted_pressure_gradient(
        self, p: np.ndarray, dt: float, is_nongeostrophic: bool,
is_nonhydrostatic: bool
    ) -> Tuple[np.ndarray, np.ndarray, np.ndarray]:
        """
```

```
        Calculate the M^(-1)*dt*pTheta*grad(pi), where M is the extended
coriolis matrix. The divergence of this term
        is the isentropic laplacian part of the Helmholtz operator. The whole
operation takes place on cells.

        Parameters
        ----------
        p : np.ndarray
            The nodal pressure vector (or perturbation). The gradient bring this
to cells.
        dt : float
            The time step
        is_nongeostrophic : bool
            The switch between geostrophic and non-geostrophic regimes
        is_nonhydrostatic : bool
            The switch between hydrostatic and non-hydrostatic regimes

        Returns
        -------
        Tuple[np.ndarray, np.ndarray, np.ndarray]
            Tuple of updated momenta arrays
        """
        # Get necessary variables/coefficients
        cellvars = self.variables.cell_vars

        #### Calculate the needed values for the updates in the RHS of the
momenta equations (the Exner pressure #######
        #### perturbation (Pi^prime)_x, _y, _z (see RHS of momenta eq.) and the
P*Theta coefficients)  ##################
        dpdx, dpdy, dpdz = self.discrete_operator.gradient(p)
        pTheta = self._calculate_coefficient_pTheta(
            cellvars
        )  # Capital P. it is written small for naming in Python.

        ###################### Calculate exner pressure gradient times
coefficient (cell-centered) ######################
        ###################### These are the nominator terms under the divergence
in the Helmholtz equation #############
        Pu = -dt * pTheta * dpdx
        Pv = -dt * pTheta * dpdy
        Pw = -dt * pTheta * dpdz if self.ndim == 3 else np.zeros_like(dpdz)

        ###################### Apply Coriolis/Buoyancy transform (M_inv)
################################################
        ###################### Result are the full terms under the divergence in
the Helmholtz equation #################
        self.coriolis.apply_inverse(
            Pu,
            Pv,
            Pw,
            self.variables,
            self.mpv,
            is_nongeostrophic,
            is_nonhydrostatic,
            self.Msq,
            dt,
        )

        return Pu, Pv, Pw

    def apply_pressure_gradient_update(
        self,
        p: np.ndarray,
        updt_chi: Union[np.ndarray, float],
```

```
        dt: float,
        is_nongeostrophic: bool,
        is_nonhydrostatic: bool,
    ):
        """Update the momenta and Chi variables using the pressure term with
their coefficients on the RHS of the
        momenta equation.

        Parameters
        ----------
        p : np.ndarray
            The pressure vector. Basically placeholder for the perturbation of
the Exner pressure (pi')
        updt_chi : np.ndarray
            The new updated value for Chi
        dt : float
            The time step
        is_nongeostrophic : bool
            The switch between geostrophic and non-geostrophic regimes
        is_nonhydrostatic : bool
            The switch between hydrostatic and non-hydrostatic regimes
        """

        ################################## Calculate the RHS of momenta equations
######################################
        Pu_incr, Pv_incr, Pw_incr =
self.calculate_enthalpy_weighted_pressure_gradient(
            p, dt, is_nongeostrophic, is_nonhydrostatic
        )

        ################################## Create the necessary variables
############################################
        cellvars = self.variables.cell_vars
        # Theta inverse to convert update of Pu, Pv and Pw to updates of rhou,
rhov and rhow
        chi = cellvars[..., VI.RHO] / cellvars[..., VI.RHOY]
        dS = self.mpv.compute_dS_on_nodes()  # Assuming cell-centered dS/dy

        ########################## Update the full variables using the
intermediate variables. ##########################
        # Notice the u_incr, v_incr and w_incr are update of Pu, Pv and Pw,
where P = rho*Theta. In order to update the
        # main momenta variables (rhou, rhov and rhow) we need to multiply the
result by 1.0/Theta = chi.
        # The plus sign is due to the fact that Pu_incr has an inherent minus
sign.
        cellvars[..., VI.RHOU] += chi * Pu_incr
        cellvars[..., VI.RHOV] += chi * Pv_incr if self.ndim >= 2 else 0.0
        cellvars[..., VI.RHOW] += chi * Pw_incr if self.ndim == 3 else 0.0
        cellvars[..., VI.RHOX] += (
            -updt_chi * dt * dS * cellvars[..., self.vertical_momentum_index]
        )

    def isentropic_laplacian(
        self,
        p: np.ndarray,
        dt: float,
        is_nongeostrophic: bool,
        is_nonhydrostatic: bool,
    ):
        """Compute the isentropic laplacian operator:  -∇·( M_inv · ( dt * (P0)°
* ∇p ) )

        Parameters
```

```
        ----------
        p : np.ndarray
            The input Exner pressure perturbation. This is the scalar field on
which the laplacian is computed.
        dt : float
            The time step
        is_nongeostrophic : bool
            The switch between geostrophic and non-geostrophic regimes
        is_nonhydrostatic : bool
            The switch between hydrostatic and non-hydrostatic regimes

        Returns
        -------
        np.ndarray
            The isentropic laplacian operator

        Notes
        -----
        The output is of shape (nnx-2, nny-2, nnz-2):
        - p is of shape (nnx, nny, nnz)
        - grad(p) is of shape (nnx-1, nny-1, nnz-1)
        - divergence(grad(p)) is of shape (nnx-2, ny-2, nz-2)
        """

        ######### Calculate the needed term inside the divergence: M_inv · ( dt
* (P0)° * ∇p )  ########################
        # The results are cell-centered
        u, v, w = self.calculate_enthalpy_weighted_pressure_gradient(
            p, dt, is_nongeostrophic, is_nonhydrostatic
        )

        ######### Stack the values above so that they can be passed to the
divergence ###################################
        vector_field_cell = np.stack([u, v, w], axis=-1)[
            ..., : self.ndim
        ]  # Ensure correct dims

        ######### Apply divergence
################################################################################
#####
        divergence = self.discrete_operator.divergence(vector_field_cell)

        ######### Negate the result and flatten for future usage in scipy
LinearOperator ################################
        laplacian = (
            -divergence
        )  # This is the sign between the first and second term in the Helmholtz
equation

        return laplacian

    def helmholtz_operator(
        self,
        p: np.ndarray,
        dt: float,
        is_nongeostrophic: bool,
        is_nonhydrostatic: bool,
        is_compressible: bool,
    ):
        """Calculate the full Helmholtz operator:
        [α_p * (∂P/∂π)°/dt]p₂ - ∇·(M_inv·(dt*(P0)°*∇p₂))

        Parameters
        ----------
```

```
        p : np.ndarray
            The input Exner pressure perturbation. This is the scalar field on
which the laplacian is computed.
        dt : float
            The time step
        is_nongeostrophic : bool
            The switch between geostrophic and non-geostrophic regimes
        is_nonhydrostatic : bool
            The switch between hydrostatic and non-hydrostatic regimes
        is_compressible : bool
            The switch between compressible and incompressible regimes

        Returns
        -------
        np.ndarray
            The full Helmholtz operator of shape (nnx-2, nny-2, nnz-2)
        """
        ############### Calculate the Laplacian
#########################################################################
        laplacian_full = self.isentropic_laplacian(
            p, dt, is_nongeostrophic, is_nonhydrostatic
        )

        ################ Extract inner node values from laplacian
#########################################################
        laplacian_inner_nodes_slice = laplacian_inner_slice(self.grid.ng)
        laplacian = laplacian_full[laplacian_inner_nodes_slice]
        assert laplacian.shape == self.grid.inshape

        ####### Creating the pressure term corresponding to the dPdpi and add it
to the laplacian ######################
        inner_slice = self.grid.get_inner_slice()
        helmholtz_result = laplacian + self.mpv.wcenter[inner_slice] *
p[inner_slice]

        return helmholtz_result

    def helmholtz_operator_linear_wrapper(
        self,
        dt: float,
        is_nongeostrophic: bool,
        is_nonhydrostatic: bool,
        is_compressible: bool,
    ) -> sp.sparse.linalg.LinearOperator:
        """
        Wraps the Helmholtz operator and return in as Scipy LinearOperator.
        """

        # Get inner slice and inner shape of node grid
        inshape = self.grid.inshape

        ######## Create the shape of the flat vector containing the inner nodes
########################################
        flat_size = np.prod(inshape)
        operator_shape = (flat_size, flat_size)

        ######## Helmholtz operator wrapper
#########################################################################
        def _matvec(p_flat):
            ############## Pad p_flat to the full nodal shape expected by
helmholtz_operator ##########################
            inner_slice = self.grid.get_inner_slice()
            p_full = np.zeros(self.grid.nshape, dtype=p_flat.dtype)
            p_full[inner_slice] = p_flat.reshape(inshape)
```

```
            # Apply the physics-based Helmholtz operator
            result = self.helmholtz_operator(
                p_full, dt, is_nongeostrophic, is_nonhydrostatic,
is_compressible
            )  # Shape is (nx-1, ny-1, nz-1)
            return result.flatten()

        return sp.sparse.linalg.LinearOperator(operator_shape, matvec=_matvec)

    def solve_helmholtz(
        self,
        rhs_flat: np.ndarray,
        dt: float,
        is_nongeostrophic: bool,
        is_nonhydrostatic: bool,
        is_compressible: bool,
        rtol: float = 1e-6,
        max_iter: Optional[int] = None,
    ) -> Tuple[np.ndarray, int]:
        """
        Solve the Helmholtz equation Ax = b using the configured linear solver
and preconditioner.

        Parameters
        ----------
        rhs_flat : np.ndarray
            The RHS of the pressure equation. In BK19 corresponds to R^n. The
shape of the array before
            flattening should have been (nx-1, ny-1, nz-1).

        dt : float
            The time step
        is_nongeostrophic : bool
            The switch between geostrophic and non-geostrophic regimes
        is_nonhydrostatic : bool
            The switch between hydrostatic and non-hydrostatic regimes
        is_compressible : bool
            The switch between compressible and incompressible regimes

        Returns
        -------
        Tuple[np.ndarray, int]
            The output of the scipy linear solver.
        """
        ######################### 1. Get the Helmholtz operator A
############################################################
        A = self.helmholtz_operator_linear_wrapper(
            dt, is_nongeostrophic, is_nonhydrostatic, is_compressible
        )

        ########################## 2. Prepare the Preconditioner Operator M_op
(representing M^-1) ######################
        M_op = None

        # Compute/update preconditioner data for the current state
        self._compute_and_store_precondition_data(
            dt, is_nongeostrophic, is_nonhydrostatic, is_compressible
        )

        if self.precon_data is not None:

            # Capture the current preconditioner data and the specific apply
function
```

```
            current_data = self.precon_data
            apply_func = self.precon_apply_inverse

            # Define the matvec for M^-1
            def matvec_M_inv(r_flat: np.ndarray) -> np.ndarray:
                return apply_func(r_flat, **current_data)

            # Wrap in SciPy LinearOperator
            precon_shape = A.shape
            M_op = sp.sparse.linalg.LinearOperator(precon_shape,
matvec=matvec_M_inv)
        else:
            print(
                f"Warning: Preconditioner data for {self.precondition_type} was
not computed."
            )

        ########################### 3. Call the configured linear solver
################################################
        solution_flat, info = self.linear_solver.solve(
            A, rhs_flat, rtol=rtol, max_iter=max_iter, M=M_op
        )

        # Optional Logging
        if info > 0:
            print(f"WARNING: Linear solver did not converge in {info}
iterations.")
        elif info < 0:
            print(f"ERROR: Linear solver failed with error code {info}.")

        return solution_flat, info

    def _calculate_P_over_Gamma(self, cellvars: np.ndarray):
        """Calculates P/Gamma. This is an intermediate function to avoid
duplicate codes.

        Parameters
        ----------
        cellvars : np.ndarray
            The full variable container for cell-centered variables
        th: Thermodynamics
            The object of thermodynamics class

        """
        return self.th.Gammainv * cellvars[..., VI.RHOY]

    def _calculate_coefficient_pTheta(self, cellvars: np.ndarray):
        """First part of coefficient calculation. Calculates P*Theta."""

        # Calculate (P*Theta): Coefficient of pressure term in momentum equation
        Y = cellvars[..., VI.RHOY] / cellvars[..., VI.RHO]
        return self._calculate_P_over_Gamma(cellvars) * Y

    def _calculate_coefficient_dPdpi(self, cellvars: np.ndarray, dt: float):
        """Calculate the second part of the coefficient calculation. Calculate
dP/dpi. See the docstring of
        pressure_coefficients_nodes for more information.

        Notes
        -----
        The shape of the output is the same as the inner NODES, which
incidentally (but evidently) is equal to the cell
        shape of the grid (cshape).
```

```
        It calculates this term for the Helmholtz equation of pressure.
        Remember the P and pi are connected to each other through the following
formula:
        pi = (1/Msq) * P^(gamma - 1). Therefore, dpi/dP = (gamma - 1)/Msq *
P^(gamma - 2). Since we need the inverse
        (dP/dpi), every part of this will be inverted in the code.

        The dt in the denominator has an obvious reason: This will be part of
the Helmholtz equation operator, the
        laplacian and this term should be created as an overall operator, the dt
(which basically belongs to the
        laplacian update) should be divided before we create the operator
([C/dt]p₂ + ∇·(M_inv·(dt*C0*∇p₂)) = DivV).
        """

        # Calculate the coefficient and the exponent of the dP/dpi using the
formula directly. (see the docstring)
        ccenter = -self.Msq * self.th.gm1inv / (dt)
        cexp = 2.0 - self.th.gamma

        # Temp variable for rhoTheta=P for readability
        P = cellvars[..., VI.RHOY]

        # Averaging over the nodes and fill the mpv container (Eq. 29 BK19)
        kernel = np.ones([2] * self.ndim)
        return (
            ccenter
            * sp.signal.fftconvolve(P**cexp, kernel, mode="valid")
            / kernel.sum()
        )
```