```
""" This module contains the function for different preconditioning functions"""

import numpy as np
import scipy as sp

from atmpy.infrastructure.utility import (
    one_element_inner_slice,
    directional_indices,
    one_element_inner_nodal_shape,
)
from typing import Callable, TYPE_CHECKING, Dict, Tuple, Any, List

if TYPE_CHECKING:
    from atmpy.pressure_solver.classical_pressure_solvers import
ClassicalPressureSolver


if TYPE_CHECKING:
    from atmpy.grid.kgrid import Grid
    from atmpy.variables.multiple_pressure_variables import MPV
    from atmpy.pressure_solver.classical_pressure_solvers import
ClassicalPressureSolver

    # Assuming utility function is accessible
    from atmpy.pressure_solver.utility import one_element_inner_slice

# ============================================================================
# Utility: Matrix Probing
# ============================================================================


def _perform_operator_probing(
    pressure_solver: "ClassicalPressureSolver",
    dt: float,
    is_nongeostrophic: bool,
    is_nonhydrostatic: bool,
    is_compressible: bool,
    inner_shape: Tuple[int, ...],  # Shape of the inner grid (operator domain)
    inner_slice_nodes: Tuple[slice, ...],  # Slice to get inner nodes from full
grid
) -> List[Tuple[Tuple[int, ...], np.ndarray]]:
    """
    Helper function to probe the Helmholtz operator numerically.

    Applies the operator to sparse test vectors (staggered 1s) (see Notes) and
returns
    the results along with the corresponding staggering pattern.

    Parameters
    ----------
    pressure_solver : ClassicalPressureSolver
        The solver instance.
    dt : float
        Time step and regime flags for the operator.
    inner_shape : Tuple[int, ...]
        The shape of the one element inner nodal grid (e.g., nx-1, ny-1, nz-1).
    inner_slice: Tuple[slice, ...]
        The slice to access the one element inner nodes from the full nodal
grid.

    Returns
    -------
    List of tuples.
    Each tuple contains:
```

```
            - index_tuple: The staggering pattern (e.g., (0,1,0)).
            - helmholtz_op_unflat: The unflattened operator result for that
pattern.

    Notes
    -----
    This function uses the matrix probing on colored grid to compute the
diagonal components of the Helmholtz operator:
    Idea: The basic classical idea is to apply the basis vectors on the
operator:
          A[k, k] = Σ_j A[k, j] * e_k[j] = A @ e_k
          This requires to apply the Helmholtz operator on basis vectors N times
          Instead , here the function applies operator on the test vectors that
are specially designed as follows:
    1. Define a set S = grid points that are not neighbor to each other
        a. if i ∈ S and j ∈ S then j ∉ Neigh(i)
    2. Define the test vector x_S as characteristic (indicator) vectors of the
set S:
        x_S[k] = 1 if k ∈ S
        x_S[k] = 0 if k ∉ S
    3. To find the k-th diagonal, assume k ∈ S and apply the operator on x_S:
        A @ x_S = Σ_j A[k, j] * e_k[j] = A[k, k]*x_S[k] + Σ_{j≠k} A[k, j] *
x_S[j]
        a. notice k ∈ S therefore x_S[k] = 1
        b. if j ∈ S, since also k ∈ S, we know j ∉ Neigh(k), therefore A[k, j] =
0 (due to sparsity of FDM differentiation)
        c. if j ∉ S, then x_S[j] = 0
        Therefore A @ x_S = A[k, k].

    In order to create this algorithm, first we create S as the staggered grid
of combinations of even/odd nodes in each direction
    *   In 1D: Two sets are needed (even indices `0::2`, odd indices `1::2`).
    *   In 2D: Four sets are needed (even-even `(0::2, 0::2)`, even-odd `(0::2,
1::2)`, odd-even `(1::2, 0::2)`, odd-odd `(1::2, 1::2)`).
    *   In 3D: Eight sets are needed (combinations like even-even-even, even-
even-odd, etc.).
    """
    grid = pressure_solver.grid
    full_nodal_shape = grid.nshape

    ################### Create the staggered grid and the test characteristic
vector x_s (See notes) ####################
    # create list of tuple of ndim elements less than 2: For example for ndim=3
(0,0,0), (0,0,1), (0,1,0), etc
    x_s = np.zeros(full_nodal_shape, dtype=np.float64)
    # Generate all 2^N staggering patterns
    indices = list(np.ndindex(*([2] * grid.ndim)))

    # Initialize results
    probe_results = []

    # Create different sets of S (staggered grid of point that are not
neighbors)
    for index_tuple in indices:
        # Refresh x_S (characteristic vector) for each new S
        x_s.fill(0.0)
        # Indices of nodes in the staggered grid S
        slicing_inner = tuple(slice(start, None, 2) for start in index_tuple)

        #### Create the test vector (characteristic vector x_S)
        # Create a mask for the nodes in the staggered grid and set them to 1
        inner_nodes_mask = np.zeros(inner_shape, dtype=bool)
        inner_nodes_mask[slicing_inner] = True
        x_s[inner_slice_nodes][inner_nodes_mask] = 1.0
```

```
        # Apply helmholtz operator on the characteristic vector of current S
        helmholtz_op_unflat = pressure_solver.helmholtz_operator(
            x_s, dt, is_nongeostrophic, is_nonhydrostatic, is_compressible
        )

        # Store the result associated with its staggering pattern
        probe_results.append((index_tuple, helmholtz_op_unflat))

    return probe_results


# ==============================================================================
# Diagonal Preconditioner Components & Application
# ==============================================================================


def compute_inverse_diagonal_components(
    pressure_solver: "ClassicalPressureSolver",
    dt: float,
    is_nongeostrophic: bool,
    is_nonhydrostatic: bool,
    is_compressible: bool,
) -> Dict[str, Any]:
    """Compute the inverse diagonal values of the operator using matrix probing.
For details see the docstring of
    _perform_operator_probing() function."""
    grid = pressure_solver.grid
    mpv = pressure_solver.mpv

    ################## Create shape and slice of one element inner ((nx-1, ny-1,
nz-1)) ################################
    inner_slice = grid.get_inner_slice()
    inner_shape = mpv.wcenter[inner_slice].shape

    ##################################### Perform the probing
##############################################################
    probe_results = _perform_operator_probing(
        pressure_solver,
        dt,
        is_nongeostrophic,
        is_nonhydrostatic,
        is_compressible,
        inner_shape,
        inner_slice,
    )

    ######################### Assemble the results from probing to get the
diagonal ####################################
    diag_inner = np.zeros(inner_shape, dtype=np.float64)
    for index_tuple, helmholtz_op_unflat in probe_results:
        # Define the slice corresponding to this probe pattern
        slicing_inner = tuple(slice(start, None, 2) for start in index_tuple)

        # Since every diagonal entry represent a vector, all those vectors are
linearly independent. The += ensures
        # that all values for all diagonals are stored.
        diag_inner[slicing_inner] += helmholtz_op_unflat[slicing_inner]

    ########################### Compute inverse
#################################################################################
    diag_inv_inner = np.zeros_like(diag_inner)
    non_zero_mask = np.abs(diag_inner) > 1e-15
    diag_inv_inner[non_zero_mask] = 1.0 / diag_inner[non_zero_mask]
```

```
    return {"diag_inv": diag_inv_inner}


def apply_inverse_diagonal(r_flat: np.ndarray, *, diag_inv: np.ndarray) ->
np.ndarray:
    """
    Applies the inverse diagonal preconditioner.
    Accepts keyword arguments matching the output of
compute_inverse_diagonal_components.
    """
    original_shape = diag_inv.shape
    if r_flat.shape[0] != np.prod(original_shape):
        raise ValueError(
            f"Shape mismatch: r flat {r_flat.shape} vs diag_inv
{original_shape}"
        )

    r_unflat = r_flat.reshape(original_shape)
    z_unflat = diag_inv * r_unflat
    return z_unflat.flatten()


#
==========================================================================
=
# Column (Tridiagonal) Preconditioner Components & Application (Example
Structure)
#
==========================================================================
=


def compute_tridiagonal_components(
    pressure_solver: "ClassicalPressureSolver",
    dt: float,
    is_nongeostrophic: bool,
    is_nonhydrostatic: bool,
    is_compressible: bool,
) -> Dict[str, Any]:
    """Computes tridiagonal components using the shared probing helper."""
    grid = pressure_solver.grid
    mpv = pressure_solver.mpv
    gravity_axis = pressure_solver.coriolis.gravity.axis

    raise NotImplementedError(
        "The correct version of tridiagonal preconditioner is not yet
implemented."
    )

    inner_slice_nodes = one_element_inner_slice(grid.ndim, full=False)
    inner_shape = mpv.wcenter[inner_slice_nodes].shape
    num_inner_vert = inner_shape[gravity_axis]

    # Perform the probing
    probe_results = _perform_operator_probing(
        pressure_solver,
        dt,
        is_nongeostrophic,
        is_nonhydrostatic,
        is_compressible,
        inner_shape,
        inner_slice_nodes,
    )
```

```
    # Process results to get tridiagonal bands
    lower_band = np.zeros(inner_shape, dtype=np.float64)
    diag_band = np.zeros(inner_shape, dtype=np.float64)
    upper_band = np.zeros(inner_shape, dtype=np.float64)

    for index_tuple, helmholtz_op_unflat in probe_results:
        # Define the slice and mask corresponding to this probe pattern
        slicing_inner = tuple(slice(start, None, 2) for start in index_tuple)
        mask_j = np.zeros(inner_shape, dtype=bool)
        mask_j[slicing_inner] = True

        # Diagonal: Result at j where input was at j
        diag_band[mask_j] += helmholtz_op_unflat[mask_j]

        # Lower: Result at j+1 where input was at j
        if np.any(mask_j[..., :-1, :]):  # Check if shift is possible within
mask bounds
            mask_jp1 = np.roll(mask_j, shift=1, axis=gravity_axis)
            valid_points_lower = (
                mask_j & mask_jp1
            )  # Ensure both j and j+1 are valid inner points hit by the probe
            # Add result at j+1 locations where input was at j
            lower_band[valid_points_lower] +=
helmholtz_op_unflat[valid_points_lower]

        # Upper: Result at j-1 where input was at j
        if np.any(mask_j[..., 1:, :]):  # Check if shift is possible
            mask_jm1 = np.roll(mask_j, shift=-1, axis=gravity_axis)
            valid_points_upper = mask_j & mask_jm1
            # Add result at j-1 locations where input was at j
            upper_band[valid_points_upper] +=
helmholtz_op_unflat[valid_points_upper]

    # --- Optional: Add horizontal probing for diagonal refinement ---
    # If needed, add loops similar to BK19's ii/kk here, calling the operator
    # again with horizontally staggered p_test vectors and adding ONLY
    # helmholtz_op_unflat[mask_horiz] to diag_band.

    return {
        "lower": lower_band,
        "diag": diag_band,
        "upper": upper_band,
        "grid": grid,
        "gravity_axis": gravity_axis,
    }


def apply_inverse_tridiagonal(
    r_flat: np.ndarray,
    *,
    lower: np.ndarray,
    diag: np.ndarray,
    upper: np.ndarray,
    grid: "Grid",
    gravity_axis: int,
) -> np.ndarray:
    """
    Applies the inverse tridiagonal preconditioner by solving column-wise
systems
    using scipy.linalg.solve_banded.

    Accepts keyword arguments matching the output of
compute_tridiagonal_components.
```

```
    """
    inner_shape = one_element_inner_nodal_shape(grid.ndim)
    num_inner_vert = inner_shape[gravity_axis]

    # Check shapes
    if not (lower.shape == diag.shape == upper.shape == inner_shape):
        raise ValueError("Shape mismatch between bands and inner grid shape.")
    if r_flat.shape[0] != np.prod(inner_shape):
        raise ValueError(
            f"Shape mismatch: r_flat {r_flat.shape} vs inner_shape
{inner_shape}"
        )

    r_unflat = r_flat.reshape(inner_shape)
    z_unflat = np.zeros_like(r_unflat)

    iter_dims = [grid.icshape[d] for d in range(grid.ndim) if d != gravity_axis]
    iter_indices = np.ndindex(*iter_dims)

    # --- Loop over all columns ---
    for index_tuple in iter_indices:
        col_slice = list(slice(None) for _ in range(grid.ndim))
        idx_counter = 0
        for d in range(grid.ndim):
            if d != gravity_axis:
                col_slice[d] = index_tuple[idx_counter]
                idx_counter += 1
        col_slice = tuple(col_slice)

        r_col = r_unflat[col_slice]
        l_col = lower[col_slice]
        d_col = diag[col_slice]
        u_col = upper[col_slice]

        # Assemble banded matrix for scipy.linalg.solve_banded
        ab = np.zeros((3, num_inner_vert), dtype=r_col.dtype)
        ab[0, 1:] = u_col[:-1]  # Upper diagonal u_j -> row j-1
        ab[1, :] = d_col  # Main diagonal d_j -> row j
        ab[2, :-1] = l_col[1:]  # Lower diagonal l_j -> row j+1

        # Solve
        try:
            z_col = sp.linalg.solve_banded((1, 1), ab, r_col,
check_finite=False)
            z_unflat[col_slice] = z_col
        except np.linalg.LinAlgError:
            # Handle singularity - check if diagonal is near zero
            if np.any(np.abs(d_col) < 1e-15):
                print(
                    f"Warning: Near-zero diagonal found in tridiagonal solve for
column {col_slice}. Setting result to zero."
                )
                z_unflat[col_slice] = 0.0
            else:
                print(
                    f"Warning: LinAlgError (possibly singular) in tridiagonal
solve for column {col_slice}. Setting result to zero."
                )
                z_unflat[col_slice] = 0.0  # Fallback

    return z_unflat.flatten()
```