```
""" The utility functions for the pressure solvers"""

import numpy as np
import numba as nb
from typing import Tuple

# ============================================================================
# 1. Numba Gradient Kernels (Defined outside the class for clarity)
#    These are the core computational strategies.
# ============================================================================
# Cache=True speeds up subsequent runs after the first compilation
_NJIT_OPTIONS = {"nogil": True, "cache": True, "fastmath": True}


@nb.njit(**_NJIT_OPTIONS)
def _gradient_1d_numba(p: np.ndarray, dx: float) -> Tuple[np.ndarray]:
    """Numba kernel for 1D gradient (cell-centered).

    Parameters
    ----------
    p : np.ndarray of shape (nnx, )
        The input scalar function (mostly pressure or exner pressure
perturbation) defined on nodes.
    dx : float
        The discretization fineness in each coordinate direction

    Returns
    -------
    Dpx : np.ndarray of shape(nnx-1,)
        The derivative in each coordinate direction
    """
    # Preallocate for derivatives
    nx = p.shape[0] - 1
    Dpx = np.empty(nx, dtype=p.dtype)

    inv_dx = 1.0 / dx
    for i in range(nx):
        # Simple difference between nodes defines gradient on the cell
        Dpx[i] = (p[i + 1] - p[i]) * inv_dx
    # Return a tuple for consistency with higher dimensions
    return (Dpx,)


# @nb.njit(**_NJIT_OPTIONS)
# def _gradient_2d_numba(
#     p: np.ndarray, dx: float, dy: float
# ) -> Tuple[np.ndarray, np.ndarray]:
#     """Numba kernel for 2D gradient (cell-centered).
#
#     Parameters
#     ----------
#     p : np.ndarray of shape (nnx, nny)
#         The input scalar function (mostly pressure or exner pressure
perturbation) defined on nodes.
#     dx, dy : float
#         The discretization fineness in each coordinate direction
#
#     Returns
#     -------
#     Dpx, Dpy : np.ndarray of shape(nnx-1, nny-1)
#         The derivative in each coordinate direction
#     """
#     nx = p.shape[0] - 1
#     ny = p.shape[1] - 1
```

```
#
#     # Preallocate for derivatives
#     Dpx = np.empty((nx, ny), dtype=p.dtype)
#     Dpy = np.empty((nx, ny), dtype=p.dtype)
#
#     inv_dx_half = 0.5 / dx
#     inv_dy_half = 0.5 / dy
#
#     for i in range(nx):
#         for j in range(ny):
#             p_i_j = p[i, j]
#             p_ip1_j = p[i + 1, j]
#             p_i_jp1 = p[i, j + 1]
#             p_ip1_jp1 = p[i + 1, j + 1]
#
#             # Dpx = avg gradient in x across the cell (i,j)
#             Dpx[i, j] = (p_ip1_j + p_ip1_jp1 - p_i_j - p_i_jp1) * inv_dx_half
#             # Dpy = avg gradient in y across the cell (i,j)
#             Dpy[i, j] = (p_i_jp1 + p_ip1_jp1 - p_i_j - p_ip1_j) * inv_dy_half
#
#     return Dpx, Dpy


@nb.njit(**_NJIT_OPTIONS)
def _gradient_2d_numba(
    p: np.ndarray, dx: float, dy: float
) -> Tuple[np.ndarray, np.ndarray]:
    """
    Calculates 2D gradient at cells using the same approach as the C code.

    This follows the pattern of the C code's correction_increments_nodes
function,
    but adapted for 2D instead of 3D.

    Parameters:
    -----------
    p : np.ndarray of shape (nnx, nny)
        The input scalar function defined on nodes
    dx, dy : float
        Grid spacing in each direction

    Returns:
    --------
    Dpx, Dpy : np.ndarray of shape (nnx-1, nny-1)
        Gradient components at cell centers
    """
    nnx, nny = p.shape
    nx = nnx - 1  # Number of cells in x
    ny = nny - 1  # Number of cells in y

    # Initialize gradient components
    Dpx = np.empty((nx, ny), dtype=p.dtype)
    Dpy = np.empty((nx, ny), dtype=p.dtype)

    # Inverse grid spacing
    oodx = 1.0 / dx
    oody = 1.0 / dy

    # We keep 0.25 to maintain the same structure as the C code
    factor_x = 0.5 * oodx
    factor_y = 0.5 * oody

    for j in range(ny):
        for i in range(nx):
```

```
            # Node indices - corners of the current cell (i,j)
            # Following the same naming convention as the C code:
            #    n01 --- n11
            #     |       |
            #    n00 --- n10
            n00 = (i, j)  # bottom-left
            n10 = (i + 1, j)  # bottom-right
            n01 = (i, j + 1)  # top-left
            n11 = (i + 1, j + 1)  # top-right

            # Get pressure values at nodes
            p00 = p[n00]
            p10 = p[n10]
            p01 = p[n01]
            p11 = p[n11]

            # Calculate gradients using the same pattern as the C code
            # Dpx calculation - differences in x-direction
            Dpx[i, j] = factor_x * (p10 - p00 + p11 - p01)

            # Dpy calculation - differences in y-direction
            Dpy[i, j] = factor_y * (p01 - p00 + p11 - p10)

    return Dpx, Dpy


@nb.njit(**_NJIT_OPTIONS)
def _gradient_3d_numba(
    p: np.ndarray, dx: float, dy: float, dz: float
) -> Tuple[np.ndarray, np.ndarray, np.ndarray]:
    """Numba kernel for 3D gradient (cell-centered).

    Parameters
    ----------
    p : np.ndarray of shape (nnx, nny, nnz)
        The input scalar function (mostly pressure or exner pressure
perturbation) defined on nodes.
    dx, dy, dz : float
        The discretization fineness in each coordinate direction

    Returns
    -------
    Dpx, Dpy, Dpz : np.ndarray of shape (nnx-1, nny-1, nnz-1)
        The derivative in each coordinate direction
    """
    nx = p.shape[0] - 1
    ny = p.shape[1] - 1
    nz = p.shape[2] - 1

    # Preallocate for derivatives
    Dpx = np.empty((nx, ny, nz), dtype=p.dtype)
    Dpy = np.empty((nx, ny, nz), dtype=p.dtype)
    Dpz = np.empty((nx, ny, nz), dtype=p.dtype)

    inv_dx_quarter = 0.25 / dx
    inv_dy_quarter = 0.25 / dy
    inv_dz_quarter = 0.25 / dz

    for i in range(nx):
        for j in range(ny):
            for k in range(nz):
                # Nodal values surrounding the cell (i, j, k) center
                p000 = p[i, j, k]
                p100 = p[i + 1, j, k]
```

```
                p010 = p[i, j + 1, k]
                p001 = p[i, j, k + 1]
                p110 = p[i + 1, j + 1, k]
                p101 = p[i + 1, j, k + 1]
                p011 = p[i, j + 1, k + 1]
                p111 = p[i + 1, j + 1, k + 1]

                # Average gradient in x across the cell
                Dpx[i, j, k] = (
                    p100 + p110 + p101 + p111 - p000 - p010 - p001 - p011
                ) * inv_dx_quarter
                # Average gradient in y across the cell
                Dpy[i, j, k] = (
                    p010 + p110 + p011 + p111 - p000 - p100 - p001 - p101
                ) * inv_dy_quarter
                # Average gradient in z across the cell
                Dpz[i, j, k] = (
                    p001 + p101 + p011 + p111 - p000 - p100 - p010 - p110
                ) * inv_dz_quarter
    return Dpx, Dpy, Dpz


# ============================================================================
# 1D Numba Kernel for Node-Centered Divergence
# ============================================================================
@nb.njit(**_NJIT_OPTIONS)
def _divergence_1d_numba(
    vector_field: np.ndarray, dx: float  # Shape (nx, 1) - Cell-centered
) -> np.ndarray:
    """
    Calculates 1D divergence at nodes based on cell-centered vector component.
    Output shape: (nnx-2, ) - Node-centered
    """
    nx = vector_field.shape[0]
    nnx = nx - 1  # number of nodes in x direction
    dtype = vector_field.dtype

    div_result = np.empty(nnx, dtype=dtype)
    inv_dx = 1.0 / dx if dx != 0 else 0.0
    u = vector_field[:, 0]  # Component along axis 0

    for i in range(nnx):  # Node index i is between cell i and i+1
        # Derivative d(u)/dx at node i: uses diff across cell i.
        term_x = (u[i + 1] - u[i]) * inv_dx
        div_result[i] = term_x

    return div_result


# ============================================================================
# 2D Numba Kernel for Node-Centered Divergence
# ============================================================================
@nb.njit(**_NJIT_OPTIONS)
def _divergence_2d_numba(
    vector_field: np.ndarray, dx: float, dy: float  # Shape (nx, ny, 2) - Cell-
centered
) -> np.ndarray:
    """
    Calculates 2D divergence at nodes based on cell-centered vector components.
    Output shape: (nnx-2, nny-2) - Node-centered
    """
    nx, ny = vector_field.shape[0], vector_field.shape[1]
    nnx = nx - 1
    nny = ny - 1
```

```
    dtype = vector_field.dtype

    if nnx == 0 or nny == 0:
        return np.empty((nnx, nny), dtype=dtype)

    div_result = np.empty((nnx, nny), dtype=dtype)
    inv_dx = 1.0 / dx
    inv_dy = 1.0 / dy
    scale_xy = 0.5  # Averaging factor

    u = vector_field[:, :, 0]  # Component along axis 0
    v = vector_field[:, :, 1]  # Component along axis 1

    for i in range(nnx):  # Node index i
        for j in range(nny):  # Node index j
            # Term d(u)/dx: diff along axis 0, average along axis 1
            diff_u_j = u[i + 1, j] - u[i, j]
            diff_u_jp1 = u[i + 1, j + 1] - u[i, j + 1]
            term_x = scale_xy * (diff_u_j + diff_u_jp1) * inv_dx

            # Term d(v)/dy: diff along axis 1, average along axis 0
            diff_v_i = v[i, j + 1] - v[i, j]
            diff_v_ip1 = v[i + 1, j + 1] - v[i + 1, j]
            term_y = scale_xy * (diff_v_i + diff_v_ip1) * inv_dy

            div_result[i, j] = term_x + term_y

    return div_result


# @nb.njit(**_NJIT_OPTIONS)
# def _divergence_2d_numba(
#     vector_field: np.ndarray,
#     dx: float,
#     dy: float,
#     is_x_periodic: bool = False,
#     is_y_periodic: bool = False,
# ) -> np.ndarray:
#     """
#     Extended version with proper boundary handling.
#
#     Input:
#         vector_field: Cell-centered vector field with shape (nx, ny, 2)
#         dx, dy: Grid spacing in x and y directions
#         is_x_periodic, is_y_periodic: Flags for periodic boundary conditions
#
#     Output:
#         Divergence at nodes with shape (nx+1, ny+1)
#     """
#     nx, ny = vector_field.shape[0], vector_field.shape[1]
#     nnx = nx + 1
#     nny = ny + 1
#     dtype = vector_field.dtype
#
#     # Use ones for scaling factors by default (can be modified for special
boundaries)
#     x_scaling = np.ones((ny, 2), dtype=dtype)  # [j, 0/1] - left/right
boundary scaling
#     y_scaling = np.ones((nx, 2), dtype=dtype)  # [i, 0/1] - bottom/top
boundary scaling
#
#     # Initialize divergence array
#     div_result = np.zeros((nnx, nny), dtype=dtype)
#
```

```
#     oodx = 1.0 / dx
#     oody = 1.0 / dy
#
#     # Extract vector components
#     u = vector_field[:, :, 0]
#     v = vector_field[:, :, 1]
#
#     # Process interior and periodic boundary cells
#     for j in range(ny):
#         for i in range(nx):
#             # Skip ghost cells if not periodic
#             if (not is_x_periodic and (i < 1 or i >= nx - 1)) or (
#                 not is_y_periodic and (j < 1 or j >= ny - 1)
#             ):
#                 continue
#
#             # Calculate contributions
#             tmpfx = 0.25 * oodx * u[i, j]
#             tmpfy = 0.25 * oody * v[i, j]
#
#             # Apply scaling for boundary cells (like Xbot/Xtop in C code)
#             scale_bottom = 1.0 if j > 0 else y_scaling[i, 0]
#             scale_top = 1.0 if j < ny - 1 else y_scaling[i, 1]
#
#             # Node indices - handle periodic wrapping
#             i_next = (i + 1) % nx if is_x_periodic else i + 1
#             j_next = (j + 1) % ny if is_y_periodic else j + 1
#
#             # Scatter to the four surrounding nodes
#             div_result[i, j] += (+tmpfx + tmpfy) * scale_bottom
#             div_result[i_next, j] += (-tmpfx + tmpfy) * scale_bottom
#             div_result[i, j_next] += (+tmpfx - tmpfy) * scale_top
#             div_result[i_next, j_next] += (-tmpfx - tmpfy) * scale_top
#
#     # Additional boundary flux corrections could be added here
#     # similar to the C code's wall flux corrections
#
#     # For comparison with the gathering approach, we'll trim ghost cells
#     result = (
#         div_result[1:-1, 1:-1]
#         if not is_x_periodic and not is_y_periodic
#         else div_result
#     )
#
#     return result


# ============================================================================
# 3D Numba Kernel for Node-Centered Divergence
# ============================================================================
@nb.njit(**_NJIT_OPTIONS)
def _divergence_3d_numba(
    vector_field: np.ndarray,  # Shape (nx, ny, nz, 3) - Cell-centered
    dx: float,
    dy: float,
    dz: float,
) -> np.ndarray:
    """
    Calculates 3D divergence at nodes based on cell-centered vector components.
    Output shape: (nnx-2, nny-2, nnz-2) - Node-centered
    """
    nx, ny, nz = vector_field.shape[0], vector_field.shape[1],
vector_field.shape[2]
    nnx = nx - 1
```

```
nny = ny - 1
nnz = nz - 1
dtype = vector_field.dtype

div_result = np.empty((nnx, nny, nnz), dtype=dtype)
inv_dx = 1.0 / dx if dx != 0 else 0.0
inv_dy = 1.0 / dy if dy != 0 else 0.0
inv_dz = 1.0 / dz if dz != 0 else 0.0
scale_xyz = 0.25  # Averaging factor

u = vector_field[:, :, :, 0]  # Component along axis 0
v = vector_field[:, :, :, 1]  # Component along axis 1
w = vector_field[:, :, :, 2]  # Component along axis 2

for i in range(nnx):  # Node index i
    for j in range(nny):  # Node index j
        for k in range(nnz):  # Node index k
            # Term d(u)/dx: diff along 0, average along 1, 2
            term_x = (
                scale_xyz
                * (
                    (u[i + 1, j, k] - u[i, j, k])
                    + (u[i + 1, j + 1, k] - u[i, j + 1, k])
                    + (u[i + 1, j, k + 1] - u[i, j, k + 1])
                    + (u[i + 1, j + 1, k + 1] - u[i, j + 1, k + 1])
                )
                * inv_dx
            )

            # Term d(v)/dy: diff along 1, average along 0, 2
            term_y = (
                scale_xyz
                * (
                    (v[i, j + 1, k] - v[i, j, k])
                    + (v[i + 1, j + 1, k] - v[i + 1, j, k])
                    + (v[i, j + 1, k + 1] - v[i, j, k + 1])
                    + (v[i + 1, j + 1, k + 1] - v[i + 1, j, k + 1])
                )
                * inv_dy
            )

            # Term d(w)/dz: diff along 2, average along 0, 1
            term_z = (
                scale_xyz
                * (
                    (w[i, j, k + 1] - w[i, j, k])
                    + (w[i + 1, j, k + 1] - w[i + 1, j, k])
                    + (w[i, j + 1, k + 1] - w[i, j + 1, k])
                    + (w[i + 1, j + 1, k + 1] - w[i + 1, j + 1, k])
                )
                * inv_dz
            )

            div_result[i, j, k] = term_x + term_y + term_z

return div_result
```