```
"""This class handles different definitions of discrete operators such as
divergence and gradient."""

from abc import ABC, abstractmethod
import numpy as np
from typing import List, Tuple, TypeVar, Callable, TYPE_CHECKING

if TYPE_CHECKING:
    from atmpy.grid.kgrid import Grid

from atmpy.pressure_solver.operators_numba_kernels import (
    _gradient_1d_numba,
    _gradient_2d_numba,
    _gradient_3d_numba,
    _divergence_1d_numba,
    _divergence_2d_numba,
    _divergence_3d_numba,
)


class AbstractDiscreteOperator(ABC):
    """Abstract class for discrete operators."""

    def __init__(self, grid: "Grid"):
        self.grid = grid

    @abstractmethod
    def gradient(self, p: np.ndarray) -> Tuple[np.ndarray, np.ndarray,
np.ndarray]:
        """Calculate the discrete gradient of a given scalar field in 1D, 2D, or
3D. The variable 'p' is defined on
        the nodes.

        Parameters
        ----------
        p: np.ndarray
            The nodal variable."""
        pass

    @abstractmethod
    def divergence(self, vector: np.ndarray):
        """Calculate the divergence of the given variables. The variables are
defined cell-centered. The result is
        on the nodes. The number of arguments passed should be the same as the
number of dimensions.

        Parameters
        ----------
        vector : np.ndarray
            The vector consisting on which the divergence is calculated. The
shape is (nx, [ny], [nz], num_components).

        """
        pass


TDiscreteOperator = TypeVar("TDiscreteOperator", bound=AbstractDiscreteOperator)


class ClassicalDiscreteOperator(AbstractDiscreteOperator):
    def __init__(self, grid: "Grid"):
        super().__init__(grid)
        self._gradient_kernel: Callable
        self._gradient_kernel_args: Tuple
        # --- Divergence Strategy Selection ---
        self._divergence_kernel: Callable
        self._divergence_kernel_args: Tuple

        dxyz = self.grid.dxyz
        ndim = self.grid.ndim

        self._select_gradient_kernel(ndim, dxyz)
        self._select_divergence_kernel(ndim, dxyz)
        self._precompile_kernels(ndim)

    def _select_gradient_kernel(self, ndim, dxyz):
        """Get the corresponding kernel for the dimension"""
        if ndim == 1:
            self._gradient_kernel = _gradient_1d_numba
            self._gradient_kernel_args = (dxyz[0],)
        elif ndim == 2:
            self._gradient_kernel = _gradient_2d_numba
            self._gradient_kernel_args = (dxyz[0], dxyz[1])
        elif ndim == 3:
            self._gradient_kernel = _gradient_3d_numba
            self._gradient_kernel_args = (dxyz[0], dxyz[1], dxyz[2])
        else:
            raise ValueError("Dimension must be 1, 2 or 3.")

    def _select_divergence_kernel(self, ndim, dxyz):
        """Get the corresponding kernel for the dimension"""
        if ndim == 1:
            self._divergence_kernel = _divergence_1d_numba
            self._divergence_kernel_args = (dxyz[0],)
        elif ndim == 2:
            self._divergence_kernel = _divergence_2d_numba
            self._divergence_kernel_args = (dxyz[0], dxyz[1])
        elif ndim == 3:
            self._divergence_kernel = _divergence_3d_numba
            self._divergence_kernel_args = (dxyz[0], dxyz[1], dxyz[2])
        else:
            raise RuntimeError(
                "Invalid ndim configuration for divergence."
            )  # Should be unreachable

    def _precompile_kernels(self, ndim):
        """Precompile all the kernels for speed-up of the main computation."""
        print(f"Pre-compiling Numba gradient kernel for ndim={ndim}...")
        try:
            # Create dummy nodal data with minimal size but correct dimensions
            # Nodal grid shape is cell grid shape + 1 in each dimension
            dummy_nodal_shape = tuple(s + 1 for s in self.grid.nshape)
            dummy_p = np.zeros(dummy_nodal_shape, dtype=np.float64)
            _ = self._gradient_kernel(dummy_p, *self._gradient_kernel_args)
            print(f"Numba gradient kernel for ndim={ndim} pre-compiled
successfully.")
        except Exception as e:
            # Log or print a warning, as failure here might indicate issues
later
            print(
                f"\nWARNING: Could not pre-compile Numba gradient kernel for
ndim={ndim}."
            )
            print(f"Error during pre-compilation: {e}")
            print(
                "Execution might be slower on the first call or fail if
incompatible.\n"
```

```
            )

        try:
            # Divergence Pre-compilation
            dummy_cell_shape = self.grid.cshape + (ndim,)
            dummy_vec = np.zeros(dummy_cell_shape, dtype=np.float64)
            _ = self._divergence_kernel(dummy_vec,
*self._divergence_kernel_args)
            print(f"  - Numba divergence kernel pre-compiled.")
        except Exception as e:
            print(
                f"\nWARNING: Could not pre-compile Numba divergence kernel for
ndim={ndim}."
            )
            print(f"Error: {e}\n")

    def gradient(self, p: np.ndarray) -> Tuple[np.ndarray, np.ndarray,
np.ndarray]:
        """Calculate the discrete gradient of a given scalar field in 1D, 2D, or
3D. The algorithm mimics the calculation of
        nodal pressure gradient specified in eq. (30a) in BK19 paper.


        Parameters
        ----------
        p : np.ndarray of shape (nx+1, [ny+1], [nz+1])
            The nodal scalar field on which the gradient is applied.

        Returns
        -------
        Tuple[np.ndarray, ...]. Arrays are of shape (nx, ny, nz)
            The gradient components (Dpx, Dpy, Dpz). For ndim < 3, unused
components are zero.
            The gradient is defined on cells.

        """
        ndim = self.grid.ndim
        cshape = self.grid.cshape  # Nodal shape of the grid

        grad_comps: Tuple[np.ndarray, ...] = self._gradient_kernel(
            p, *self._gradient_kernel_args
        )

        # --- Pad the result for uniform output ---
        if ndim == 1:
            # Need placeholder shapes for Dpy, Dpz if they are zero arrays
            return (
                grad_comps[0],
                np.zeros(cshape, dtype=p.dtype),
                np.zeros(cshape, dtype=p.dtype),
            )
        elif ndim == 2:
            return (grad_comps[0], grad_comps[1], np.zeros(cshape,
dtype=p.dtype))
        elif ndim == 3:
            return grad_comps  # Already has 3 components
        else:
            # Should be unreachable
            raise RuntimeError("Invalid ndim")

    def divergence(self, vector: np.ndarray) -> np.ndarray:
        """
        Calculates the divergence of a cell-centered vector field at the nodes.
        Parameters
        ----------
        vector : np.ndarray
            The cell-centered vector field.
            Shape: (nx, [ny], [nz], ndim)

        Returns
        -------
        np.ndarray
            The divergence evaluated at the nodes.
            Shape: (nx-1, [ny-1], [nz-1])
        """
        div_result = self._divergence_kernel(vector,
*self._divergence_kernel_args)

        return div_result


if __name__ == "__main__":
    from atmpy.grid.utility import DimensionSpec, create_grid

    np.set_printoptions(linewidth=300)
    np.set_printoptions(suppress=True)
    np.set_printoptions(precision=5)

    ############################################################################
    ################### GRID ###################################################
    nx = 6
    ngx = 2
    nnx = nx + 2 * ngx
    ny = 5
    ngy = 2
    nny = ny + 2 * ngy

    dim = [DimensionSpec(nx, 0, 2, ngx), DimensionSpec(ny, 0, 2, ngy)]
    grid = create_grid(dim)

    discreteOp = ClassicalDiscreteOperator(grid)


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

    p_nodal_2d = np.random.rand(*grid.nshape)
    print(p_nodal_2d)

    dpx, dpy, dpz = discreteOp.gradient(p_nodal_2d)
    print(dpx)
```