```python
"""This module contains the different advection routines to be passed to the
solver class. The signature of the following
function are the same."""

import numpy as np
from typing import TYPE_CHECKING

from atmpy.infrastructure.utility import (
    dimension_directions,
    directional_indices,
    direction_axis,
)

if TYPE_CHECKING:
    from atmpy.flux.flux import Flux
    from atmpy.variables.variables import Variables
    from atmpy.grid.kgrid import Grid
from atmpy.infrastructure.enums import (
    VariableIndices as VI,
    PrimitiveVariableIndices as PI,
)

from typing import List, Literal


# --- First-Order Sequential (Godunov) Splitting ---
def first_order_splitting_advection(
    grid: "Grid",
    variables: "Variables",
    flux: "Flux",
    dt: float,
    sweep_order: List[str],  # e.g., ['x', 'y', 'z']
    boundary_manager: "BoundaryManager",
) -> None:
    """
    Compute the first order advection in all directions. It does a single sweep
of advection in
    dimensions given the order of dimension in parameter sweep_order.

    Parameters
    ----------
    grid: Grid
        The spatial grid object
    variables: Variables
        The variables container.
    flux: Flux
        The flux object.
    dt: float
        The time step.
    sweep_order: List[str]
        The sweep order. It indicates whether sweep the dimensions in a standard
x-y-z-z-y-x or some alternative way
    """

    ndim: int = grid.ndim

    for direction_str in sweep_order:
        _1d_directional_advection(
            grid, variables, flux, direction_str, dt, boundary_manager, order=2
        )

    # for direction_str in sweep_order:
    #     direction_int: int = direction_axis(direction_str)
    #     left_idx, right_idx, _ = directional_indices(ndim, direction_str,
```

```
full=True)
    #     lmbda: float = dt / grid.dxyz[direction_int]
    #     variables.cell_vars[...] += lmbda * (
    #         flux.flux[direction_str][left_idx] - flux.flux[direction_str]
[right_idx]
    #     )


# --- Second-Order Strang Splitting ---
def upwind_strang_split_advection(
    grid: "Grid",
    variables: "Variables",
    flux: "Flux",
    dt: float,
    sweep_order: List[str],
    boundary_manager: "BoundaryManager",
) -> None:
    """
    Compute the second order Strang-split upwind advection in all directions. It
does a double sweep of advection in
    dimensions given the order of dimension in parameter sweep_order.

    Parameters
    ----------
    grid: Grid
        The spatial grid object
    variables: Variables
        The variables container.
    flux: Flux
        The flux object.
    dt: float
        The time step.
    sweep_order: List[str]
        The sweep order. It indicates whether sweep the dimensions in a standard
x-y-z-z-y-x or some alternative way
    """

    ################################ Prepare timestep and sweep order in
directions ###################################
    half_dt = 0.5 * dt
    current_sweep_order = list(sweep_order)

    ################################ First half-sweep
##################################################################
    for direction_str in current_sweep_order:
        _1d_directional_advection(
            grid, variables, flux, direction_str, half_dt, boundary_manager
        )

    ################################# Second half-sweep
################################################################
    for direction_str in current_sweep_order[::-1]:
        _1d_directional_advection(
            grid, variables, flux, direction_str, half_dt, boundary_manager
        )


def _1d_directional_advection(
    grid: "Grid",
    variables: "Variables",
    flux: "Flux",
    direction: str,
    dt: float,
    boundary_manager: "BoundaryManager",
```

```
    order: int = 2,
) -> None:
    """
    Core 1D advection in the given direction. It will be used to update in both
1D and 2D strang splitting routines.

    Parameters
    ----------
    grid: Grid
        The spatial grid object
    variables: Variables
        The variables container.
    flux: Flux
        The flux object.
    direction: str
        The direction of advection.
    dt: float
        The time step.
    """
    ############################## Apply boundary conditions
###########################################################
    boundary_manager.apply_boundary_on_direction(variables.cell_vars, direction)

    ############################## Parameters
##########################################################################
    ndim: int = grid.ndim
    direction_int: int = direction_axis(direction)
    lmbda: float = dt / grid.dxyz[direction_int]  # if order == 2 else 0

    # Find the left and right indices
    left_idx, right_idx, _ = directional_indices(ndim, direction, full=True)

    # ################################ Apply Riemann Solver
##############################################################
    flux.apply_riemann_solver(lmbda, direction)

    ################################ Update variables
##################################################################
    # if order == 2:
    #     variables.cell_vars[...] += lmbda * (
    #         flux.flux[direction][left_idx] - flux.flux[direction][right_idx]
    #     )

    variables.cell_vars[...] += lmbda * (
        flux.flux[direction][left_idx] - flux.flux[direction][right_idx]
    )

    ############################## Apply boundary conditions
###########################################################
    boundary_manager.apply_boundary_on_direction(variables.cell_vars, direction)


if __name__ == "__main__":
    from atmpy.physics.eos import ExnerBasedEOS
    from atmpy.grid.utility import DimensionSpec, create_grid
    from atmpy.variables.variables import Variables
    from atmpy.flux.flux import Flux
    from atmpy.boundary_conditions.utility import create_params
    from atmpy.infrastructure.enums import (
        BoundarySide as BdrySide,
        BoundaryConditions as BdryType,
    )
    from atmpy.boundary_conditions.boundary_manager import BoundaryManager
```

```
    np.set_printoptions(linewidth=100)

    dt = 0.1

    nx = 1
    ngx = 2
    nnx = nx + 2 * ngx
    ny = 2
    ngy = 2
    nny = ny + 2 * ngy

    dim = [DimensionSpec(nx, 0, 2, ngx), DimensionSpec(ny, 0, 2, ngy)]
    grid = create_grid(dim)
    rng = np.random.default_rng()
    arr = np.arange(nnx * nny)
    rng.shuffle(arr)
    array = arr.reshape(nnx, nny)

    variables = Variables(grid, 5, 1)
    variables.cell_vars[..., VI.RHO] = 1
    variables.cell_vars[..., VI.RHO][1:-1, 1:-1] = 4
    variables.cell_vars[..., VI.RHOU] = array
    variables.cell_vars[..., VI.RHOY] = 2

    rng.shuffle(arr)
    array = arr.reshape(nnx, nny)
    variables.cell_vars[..., VI.RHOV] = array
    eos = ExnerBasedEOS()
    flux = Flux(grid, variables, eos, dt)

    bc_data = {}
    create_params(bc_data, BdrySide.LEFT, BdryType.PERIODIC, direction="x",
grid=grid)
    create_params(bc_data, BdrySide.RIGHT, BdryType.PERIODIC, direction="x",
grid=grid)
    create_params(bc_data, BdrySide.BOTTOM, BdryType.PERIODIC, direction="y",
grid=grid)
    create_params(bc_data, BdrySide.TOP, BdryType.PERIODIC, direction="y",
grid=grid)

    manager = BoundaryManager()
    manager.setup_conditions(bc_data)

    upwind_strang_split_advection(grid, variables, flux, dt,
boundary_manager=manager)
    print(variables.cell_vars[..., VI.RHOU])
```