```
import numpy as np
from dataclasses import field  # Removed dataclass, not needed here
from typing import List, Tuple, Dict, Any, TYPE_CHECKING
import matplotlib
from setuptools.sandbox import run_setup

matplotlib.use("TkAgg")

import logging

from atmpy.infrastructure.utility import (
    one_element_inner_nodal_shape,
    one_element_inner_slice,
    directional_indices,
)
from atmpy.test_cases.base_test_case import BaseTestCase
from atmpy.configuration.simulation_configuration import SimulationConfig
from atmpy.infrastructure.enums import (
    BoundaryConditions as BdryType,
    BoundarySide,
    AdvectionRoutines,
)
from atmpy.infrastructure.enums import (
    VariableIndices as VI,
    HydrostateIndices as HI,
    SlopeLimiters as LimiterType,
)
from atmpy.physics.thermodynamics import Thermodynamics

if TYPE_CHECKING:
    from atmpy.variables.variables import Variables
    from atmpy.variables.multiple_pressure_variables import MPV


def traveling_vortex_stratification(y: float) -> float:
    """
    Isothermal stratification for the Traveling Vortex case.
    Returns a constant value (typically 1.0 in non-dimensional setups).
    """
    return 1.0


class TravelingVortex(BaseTestCase):
    """
    Traveling Vortex test case based on the setup described in PyBella.
    This involves an isentropic vortex embedded in a uniform flow on a doubly
    periodic domain with zero gravity.
    """

    def __init__(self, config_override: SimulationConfig = None):
        # Initialize with a default SimulationConfig, which will be modified in
setup
        _effective_config: SimulationConfig
        run_setup_method = False

        if config_override is not None:
            _effective_config = config_override
        else:
            # No override, create a default config. BaseTestCase will get this,
            # and then setup() will populate it.
            _effective_config = SimulationConfig()
            run_setup_method = True

        super().__init__(name="TravelingVortex", config=_effective_config)
```

```
        ################################ Vortex Specific Parameters
################################################################
        self.u0: float = 1.0  # Background velocity U
        self.v0: float = 1.0  # Background velocity V
        self.w0: float = 0.0  # Background velocity W
        self.p0: float = 1.0  # Background pressure (dimensionless)
        self.rho0: float = 1.0  # Background density (dimensionless)
        self.rotdir: float = 1.0  # Rotation direction
        self.alpha: float = -1.0  # Vortex strength parameter 1
        self.alpha_const: float = 3.0  # Vortex strength parameter 2
        self.del_rho: float = -0.5  # Density deficit at center
        self.R0: float = 0.4  # Vortex radius scale
        self.fac: float = (
            1.0 * 1024.0
        )  # Vortex magnitude factor (affects velocity/pressure)
        self.xc: float = 0.0  # Vortex center x
        self.yc: float = 0.0  # Vortex center y

        self.u0: float = 1.0

        ######################## Polynomial coefficients for pressure
perturbation ########################################
        self.coe = np.array(
            [
                1.0 / 24.0,
                -6.0 / 13.0,
                15.0 / 7.0,
                -74.0 / 15.0,
                57.0 / 16.0,
                174.0 / 17.0,
                -269.0 / 9.0,
                450.0 / 19.0,
                1071.0 / 40.0,
                -1564.0 / 21.0,
                510.0 / 11.0,
                1020.0 / 23.0,
                -1105.0 / 12.0,
                204.0 / 5.0,
                510.0 / 13.0,
                -1564.0 / 27.0,
                153.0 / 8.0,
                450.0 / 29.0,
                -269.0 / 15.0,
                174.0 / 31.0,
                57.0 / 32.0,
                -74.0 / 33.0,
                15.0 / 17.0,
                -6.0 / 35.0,
                1.0 / 72.0,
            ]
        )
        self.const_coe = np.array(
            [
                1.0 / 24,
                -6.0 / 13,
                33.0 / 14,
                -22.0 / 3,
                495.0 / 32,
                -396.0 / 17,
                +77.0 / 3,
                -396.0 / 19,
                99.0 / 8,
                -110.0 / 21,
```

```
                +3.0 / 2,
                -6.0 / 23,
                +1.0 / 48,
            ]
        )

        ############################3 Call setup to configure the simulation
##########################################
        if run_setup_method:
            self.setup()

    def setup(self):
        """Configure the SimulationConfig for the Traveling Vortex case."""
        print("Setting up Traveling Vortex configuration...")

        ###################################### Grid Configuration
##########################################################
        n = 64
        grid_updates = {
            "ndim": 2,
            "nx": n,
            "ny": n,
            "nz": 0,
            "xmin": -0.5,
            "xmax": 0.5,
            "ymin": -0.5,
            "ymax": 0.5,
            "ngx": 2,  # Standard ghost cells
            "ngy": 2,
        }
        self.set_grid_configuration(grid_updates)

        ###################################### Boundary Conditions
##########################################################
        self.set_boundary_condition(
            BoundarySide.LEFT, BdryType.PERIODIC, mpv_type=BdryType.PERIODIC
        )
        self.set_boundary_condition(
            BoundarySide.RIGHT, BdryType.PERIODIC, mpv_type=BdryType.PERIODIC
        )
        self.set_boundary_condition(
            BoundarySide.BOTTOM, BdryType.PERIODIC, mpv_type=BdryType.PERIODIC
        )
        self.set_boundary_condition(
            BoundarySide.TOP, BdryType.PERIODIC, mpv_type=BdryType.PERIODIC
        )

        ###################################### Temporal Setting
##########################################################
        temporal_updates = {
            "CFL": 0.8,
            "dtfixed": 0.005,
            "dtfixed0": 0.005,
            "tout": np.array([10.0]),
            "stepmax": 101,
            "use_acoustic_cfl": False, # If True adds max_sound_speed to the
speed, therefore smaller dt in dynamic
        }
        self.set_temporal(temporal_updates)

        ###################################### Physics Settings
##########################################################
        physics_updates = {
            "wind_speed": [self.u0, self.v0, self.w0],
```

```
            "gravity_strength": (0.0, 0.0, 0.0),  # Zero gravity case
            "coriolis_strength": (0.0, 0.0, 0.0),
            "stratification": traveling_vortex_stratification,  # Isothermal
background
        }
        self.set_physics(physics_updates)

        ################################### Model Regimes
#####################################################################
        regime_updates = {
            "is_nongeostrophic": 1,
            "is_nonhydrostatic": 1,
            "is_compressible": 1,
        }
        self.set_model_regimes(regime_updates)  # This also updates Msq

        ################################### Numerics
#####################################################################
        numerics_updates = {
            "limiter_scalars": LimiterType.VAN_LEER,
            "first_order_advection_routine": AdvectionRoutines.FIRST_ORDER_RK,
            "second_order_advection_routine": AdvectionRoutines.STRANG_SPLIT,
            "initial_projection": True,
        }
        self.set_numerics(numerics_updates)

        ################################### Outputs
#####################################################################
        output_updates = {
            "output_type": "test",
            "output_folder": "traveling_vortex",
            "output_base_name": "_traveling_vortex",
            "output_timesteps": True,
            # output_suffix is updated automatically based on grid
        }
        self.set_outputs(output_updates)

        ################################### Diagnostics
#####################################################################
        diag_updates = {
            "diag": True,
            "diag_current_run": "atmpy_travelling_vortex",
        }
        self.set_diagnostics(diag_updates)

        ################################### Global Constants
#####################################################################

        constants_updates = {
            "gamma": 1.4,
            "R_gas": 287.4,
            "p_ref": 1.0e5,
            "T_ref": 300.0,
            "h_ref": 10_000.0,
            "t_ref": 100.0,
            "grav": 0.0,
        }
        self.set_global_constants(constants_updates)

        # Final check/update of Msq after constants are set
        self._update_Msq()
        # Final check/update of output suffix
        self._update_output_suffix()
```

```
        print(f"Configuration complete. Msq = {self.config.model_regimes.Msq}")
        print(
            f"Output files: {self.config.outputs.output_base_name}
{self.config.outputs.output_suffix}"
        )

    def initialize_solution(self, variables: "Variables", mpv: "MPV"):
        """Initialize density, momentum, potential temperature, and pressure
fields."""
        print("Initializing solution for Traveling Vortex...")

        grid = self.config.grid
        thermo = Thermodynamics()
        thermo.update(self.config.global_constants.gamma)
        Msq = self.config.model_regimes.Msq

        if Msq <= 0 and self.config.model_regimes.is_compressible:
            print(
                "Warning: Msq is zero or negative, but is_compressible=1.
Pressure perturbation might be zero."
            )

        # Get slices for inner domain (excluding ghost cells)
        inner_slice = grid.get_inner_slice()

        # --- Calculate Hydrostatic Base State ---
        gravity = self.config.physics.gravity_strength
        mpv.state(gravity, Msq)

        # --- Get Cell-Centered Coordinates ---
        # Assuming grid object provides meshgrid or similar functionality
        if grid.ndim == 2:
            # Use meshgrid based on cell centers
            XC, YC = np.meshgrid(
                grid.x_cells[inner_slice[0]],
                grid.y_cells[inner_slice[1]],
                indexing="ij",
            )
        else:  # Handle 1D or 3D if necessary
            raise NotImplementedError(
                "Traveling vortex initialization only implemented for 2D"
            )

        # --- Calculate Distance from Vortex Center (Handling Periodicity) ---
        Lx = grid.x_end - grid.x_start
        Ly = grid.y_end - grid.y_start

        dx = XC - self.xc
        dy = YC - self.yc

        # Account for periodicity: find the closest image
        dx = dx - Lx * np.round(dx / Lx)
        dy = dy - Ly * np.round(dy / Ly)

        r_cell = np.sqrt(dx**2 + dy**2)
        r_over_R0_cell = np.divide(r_cell, self.R0, where=self.R0 != 0)

        # --- Calculate Tangential Velocity ---
        uth_cell = np.zeros_like(r_cell)
        mask_cell = (r_cell < self.R0) & (
            r_cell > 1e-9
        )  # Avoid division by zero at center
        uth_cell[mask_cell] = (
            self.rotdir
```

```
            * self.fac
            * (1.0 - r_over_R0_cell[mask_cell]) ** 6
            * r_over_R0_cell[mask_cell] ** 6
        )

        # --- Calculate Velocity Components ---
        u_pert = np.zeros_like(uth_cell)
        v_pert = np.zeros_like(uth_cell)
        u_pert[mask_cell] = uth_cell[mask_cell] * (-dy[mask_cell] /
r_cell[mask_cell])
        v_pert[mask_cell] = uth_cell[mask_cell] * (+dx[mask_cell] /
r_cell[mask_cell])

        u_total = self.u0 + u_pert
        v_total = self.v0 + v_pert
        w_total = self.w0  # Remains zero for 2D

        # --- Calculate Density ---
        rho_total = np.full_like(r_cell, self.rho0)
        mask_rho = r_cell < self.R0
        rho_total[mask_rho] += self.del_rho * (1.0 - r_over_R0_cell[mask_rho] **
2) ** 6

        # --- Calculate Pressure Perturbation (dp2c) ---
        dp2c = np.zeros_like(r_cell)
        if Msq > 1e-10:  # Only calculate if compressible and vortex has
strength
            for ip, c in enumerate(self.coe):
                term = c * (r_over_R0_cell ** (12 + ip) - 1.0) * self.rotdir**2
                dp2c[mask_rho] += term[mask_rho]

            dp2c_const = np.zeros_like(r_cell)
            for ip, c in enumerate(self.const_coe):
                term = c * (r_over_R0_cell ** (12 + ip) - 1.0) * self.rotdir**2
                dp2c_const[mask_rho] += term[mask_rho]

            dp2c = self.alpha * dp2c + self.alpha_const * dp2c_const
            # Scale pressure perturbation by Msq * fac^2
            dp2c *= Msq * self.fac**2
        else:
            print("Msq is near zero, pressure perturbation dp2c set to zero.")

        # --- Assign to Cell Variables (Inner Domain Only) ---
        variables.cell_vars[inner_slice + (VI.RHO,)] = rho_total
        variables.cell_vars[inner_slice + (VI.RHOU,)] = rho_total * u_total
        # variables.cell_vars[inner_slice + (VI.RHOU,)] = 0
        variables.cell_vars[inner_slice + (VI.RHOV,)] = rho_total * v_total
        # variables.cell_vars[inner_slice + (VI.RHOV,)] = 0
        variables.cell_vars[inner_slice + (VI.RHOW,)] = rho_total * w_total

        rhoY0_cells = mpv.hydrostate.cell_vars[..., HI.RHOY0]
        # Calculate rhoY (Potential Temperature * Density)
        # if self.config.model_regimes.is_compressible:
        if True:
            p_total = self.p0 + dp2c  # Add perturbation to base pressure
            # Ensure pressure is positive before taking power
            p_total_safe = np.maximum(p_total, 1e-9)
            variables.cell_vars[inner_slice + (VI.RHOY,)] =
p_total_safe**thermo.gamminv
        else:
            variables.cell_vars[inner_slice + (VI.RHOY,)] = rho_total *
rhoY0_cells[inner_slice[1]]

        # Calculate rhoX (Tracers) - Set to zero if not used
```

```
        if VI.RHOX < variables.num_vars_cell:
            variables.cell_vars[inner_slice + (VI.RHOX,)] = 0.0

        # --- Assign Pressure Perturbation p2 to MPV (Cells, Inner Domain Only)
---
        p2c_unscaled = np.zeros_like(r_cell)
        for ip, c in enumerate(self.coe):
            term = c * (r_over_R0_cell ** (12 + ip) - 1.0) * self.rotdir**2
            p2c_unscaled[mask_rho] += term[mask_rho]
        p2c_const_unscaled = np.zeros_like(r_cell)
        for ip, c in enumerate(self.const_coe):
            term = c * (r_over_R0_cell ** (12 + ip) - 1.0) * self.rotdir**2
            p2c_const_unscaled[mask_rho] += term[mask_rho]

        p2c_unscaled = self.alpha * p2c_unscaled + self.alpha_const *
p2c_const_unscaled

        # Divide by background rhoY0, multiply by Gamma * fac^2
        mpv.p2_cells[inner_slice] = (
            thermo.Gamma
            * self.fac**2
            * np.divide(
                p2c_unscaled,
                rhoY0_cells[inner_slice[1]],
                where=rhoY0_cells[inner_slice[1]] != 0,
            )
        )

        # --- Calculate Nodal Pressure Perturbation p2 (Nodes, Inner Domain
Only) ---
        if grid.ndim == 2:
            XN, YN = np.meshgrid(
                grid.x_nodes[inner_slice[0]],
                grid.y_nodes[inner_slice[1]],
                indexing="ij",
            )
        else:
            raise NotImplementedError("Nodal calculation only for 2D")

        dx_node = XN - self.xc
        dy_node = YN - self.yc
        dx_node = dx_node - Lx * np.round(dx_node / Lx)
        dy_node = dy_node - Ly * np.round(dy_node / Ly)

        r_node = np.sqrt(dx_node**2 + dy_node**2)
        r_over_R0_node = np.divide(r_node, self.R0, where=self.R0 != 0)
        mask_node = r_node < self.R0

        p2_nodes_unscaled = np.zeros_like(r_node)
        for ip, c in enumerate(self.coe):
            term = c * (r_over_R0_node ** (12 + ip) - 1.0) * self.rotdir**2
            p2_nodes_unscaled[mask_node] += term[mask_node]
        p2n_const_unscaled = np.zeros_like(r_node)
        for ip, c in enumerate(self.const_coe):
            term = c * (r_over_R0_node ** (12 + ip) - 1.0) * self.rotdir**2
            p2n_const_unscaled[mask_node] += term[mask_node]

        p2_nodes_unscaled = (
            self.alpha * p2_nodes_unscaled + self.alpha_const *
p2n_const_unscaled
        )

        ngy = self.config.grid.ng[1][0]  # get number of ghost cells in y
direction.
```

```
        mpv.p2_nodes[inner_slice] = (
            thermo.Gamma
            * self.fac**2
            * np.divide(p2_nodes_unscaled, rhoY0_cells[ngy : -ngy + 1])  #
Divide by 1.0
        )

        mpv.p2_nodes /= 14

        # x = mpv.hydrostate.node_vars[..., HI.P0] /
self.config.model_regimes.Msq
        # mpv.p2_nodes = np.repeat(x.reshape(1, -1),
self.config.spatial_grid.grid.nshape[0], axis=0)


        # --- Set dp2_nodes (used in pressure solver) ---
        # PyBella sets dp2_nodes = p2_nodes initially after
hydrostatics.initial_pressure
        # Here, we can initialize it similarly or set to zero. Let's initialize.
        mpv.p0 = self.p0
        mpv.dp2_nodes[...] = 0.0

        # --- Initial Projection ---
        # The PyBella code calls lm_sp.euler_backward_non_advective_impl_part
here.
        # we *do not* include this logic here. It belongs in the
        # time integrator or a separate initialization step *after* the test
case
        # setup and variable initialization. The solver calling this test case
        # would be responsible for performing the initial projection if
        # config.numerics.initial_projection is True.

        logging.info("Solution initialization complete.")


if __name__ == "__main__":
    from atmpy.boundary_conditions.boundary_manager import BoundaryManager
    import matplotlib.pyplot as plt
    from atmpy.variables.multiple_pressure_variables import MPV
    from atmpy.variables.variables import Variables

    case = TravelingVortex()
    # case = RisingBubble()
    config = case.config  # The SimulationConfig object is now held by the case

    # Modify config if needed (e.g., simulation time)
    config.temporal.tmax = 3
    config.temporal.stepmax = 101000  # Limit steps
    config.outputs.output_frequency_steps = 3  # Output every 2 steps
    config.temporal.tout = [0]  # Also output at a specific time

    grid = config.grid
    gravity_vec = config.physics.gravity_strength
    Msq = config.model_regimes.Msq
    th = Thermodynamics()
    th.update(config.global_constants.gamma)

    ###################################### MPV
########################################################################
    mpv = MPV(grid)

    ###################################### Variables
#######################################################################
    num_vars = 6
```

```
    variables = Variables(
        grid, num_vars_cell=num_vars, num_vars_node=1
    )  # Adjust num_vars if needed

    # --- Initialize Solution using Test Case
    case.initialize_solution(variables, mpv)  # This now uses the case object

    bm_config = config.get_boundary_manager_config()
    manager = BoundaryManager(bm_config)
    # Apply initial BCs (though initialize_solution might handle inner domain,
ghosts need update)
    manager.apply_boundary_on_all_sides(variables.cell_vars)
    # nodal_bc_contexts = ([manager.get_context_for_side(i, is_nodal=True) for i
in range(grid.ndim * 2)]) # Get contexts for p2
    # manager.apply_boundary_on_single_var_all_sides(mpv.p2_nodes,
nodal_bc_contexts)

    fig, ax = plt.subplots()
    x_coords = grid.x_cells
    y_coords = grid.y_cells
    cmap = plt.cm.viridis
    rho = variables.cell_vars[..., VI.RHO]
    rhou = variables.cell_vars[..., VI.RHOU]
    u = rhou / rho
    arg = u
    data_min = arg.min().item()
    data_max = arg.max().item()

    contour = ax.contourf(
        x_coords,
        y_coords,
        arg.T,
        cmap=cmap,
        levels=np.linspace(data_min, data_max, 15),
    )

    cbar = fig.colorbar(contour, ax=ax)
    ax.set_xlabel("X (m)")
    ax.set_ylabel("Y (m)")

    # contour = ax.contourf(
    #     x_coords[2:-2],
    #     y_coords[2:-2],
    #     u.T,
    #     cmap=cmap,
    #     levels=np.linspace(data_min, data_max, 15),
    # )

    plt.show()

    case = TravelingVortex()
    config = case.config

    grid = config.grid
    mpv = MPV(grid)
    num_vars = 6
    variables = Variables(grid, num_vars_cell=num_vars, num_vars_node=1)
    case.initialize_solution(variables, mpv)

    bm_config = (
        config.get_boundary_manager_config()
    )  # Not strictly needed for this debug
    manager = BoundaryManager(bm_config)
    manager.apply_boundary_on_all_sides(variables.cell_vars)
```

```
    fig, ax = plt.subplots()
    x_coords = grid.x_cells
    y_coords = grid.y_cells
    cmap = plt.cm.viridis

    rho = variables.cell_vars[..., VI.RHO]
    rhou = variables.cell_vars[..., VI.RHOU]
    #
    # print(f"--- Debugging Initial U in traveling_vortex.py ---")
    # print(f"Full rho array min: {rho.min()}, max: {rho.max()}")
    # print(f"Full rhou array min: {rhou.min()}, max: {rhou.max()}")

    # Check for zeros in rho BEFORE division
    # if np.any(rho == 0):
    #     print("WARNING: Zeros found in rho array BEFORE division!")
    #     zero_rho_indices = np.where(rho == 0)
    #     print(f"Indices of zero rho: {zero_rho_indices}")
    #     if len(zero_rho_indices[0]) > 0:  # If any zeros found
    #         print(f"rhou values at zero rho locations:
{rhou[zero_rho_indices]}")

    # Perform division carefully
    u_initial = np.zeros_like(rhou, dtype=float)  # Ensure float type
    # Create a mask for non-zero rho to avoid division by zero warnings/errors
    # Use a small epsilon to also catch near-zero values if they are problematic
    epsilon = 1e-12
    valid_rho_mask = np.abs(rho) > epsilon

    # Calculate u_initial only where rho is valid
    u_initial[valid_rho_mask] = rhou[valid_rho_mask] / rho[valid_rho_mask]

    # Set u_initial to NaN where rho was not valid (or keep as 0, depending on
desired plot)
    # Setting to NaN will make matplotlib skip these points (often appearing
white)
    u_initial[~valid_rho_mask] = np.nan

    print(
        f"u_initial min (finite): {np.nanmin(u_initial)}, max (finite):
{np.nanmax(u_initial)}"
    )
    if np.any(np.isinf(u_initial)):
        print("WARNING: Inf values found in u_initial array AFTER division!")
    if np.any(np.isnan(u_initial)):
        print(
            "WARNING: NaN values found in u_initial array AFTER division (could
be from invalid rho)!"
        )
        nan_indices = np.where(np.isnan(u_initial))
        # Optional: print x,y coordinates of NaNs if helpful
        # print(f"X-coords of NaNs: {x_coords[nan_indices[0]]}") # This indexing
might need adjustment based on array structure
        # print(f"Y-coords of NaNs: {y_coords[nan_indices[1]]}")

    # Determine plot range from finite values
    data_min_plot = np.nanmin(u_initial)
    data_max_plot = np.nanmax(u_initial)

    # If after all this, min/max are still problematic, manually set them to
expected theoretical range
    # data_min_plot = 0.70 # Example
    # data_max_plot = 1.30 # Example
```

```
print(f"Plotting u_initial with min={data_min_plot}, max={data_max_plot}")

contour = ax.contourf(
    x_coords,
    y_coords,
    u_initial.T,  # Plotting u_initial
    cmap=cmap,
    levels=np.linspace(data_min_plot, data_max_plot, 15),
    # extend='both' # Useful if data goes outside levels
)

cbar = fig.colorbar(contour, ax=ax)
ax.set_xlabel("X (m)")
ax.set_ylabel("Y (m)")
ax.set_title("Initial u = rhou/rho from traveling_vortex.py")

plt.show()
```