```
from __future__ import annotations

import time
import logging
from typing import TYPE_CHECKING, Dict, Any, Optional

import numpy as np

# Assuming output_writer.py is in atmpy.io (Not directly used here, but by
solver_saver)
# from atmpy.io.output_writers import NetCDFWriter
# from atmpy.infrastructure.enums import ( # Not directly used here, but by
solver_saver
#     VariableIndices as VI,
#     PrimitiveVariableIndices as PVI,
# )
# from atmpy.physics.eos import ExnerBasedEOS # Not directly used here, but by
solver_saver
from atmpy.solver.utility import calculate_dynamic_dt
import atmpy.io.saver as solver_saver

# Basic logging setup
logging.basicConfig(
    level=logging.INFO, format="%(asctime)s - %(levelname)s - %(message)s"
)

if TYPE_CHECKING:
    from atmpy.configuration.simulation_configuration import SimulationConfig
    from atmpy.grid.kgrid import Grid
    from atmpy.variables.variables import Variables
    from atmpy.variables.multiple_pressure_variables import MPV
    from atmpy.time_integrators.abstract_time_integrator import
AbstractTimeIntegrator

MACHINE_EPSILON = np.finfo(float).eps


class Solver:
    """
    Orchestrates the atmospheric simulation loop, including output and
checkpointing
    by delegating to the solver_saver module.
    """

    def __init__(
        self,
        config: "SimulationConfig",
        grid: "Grid",
        variables: "Variables",
        mpv: "MPV",
        time_integrator: "AbstractTimeIntegrator",
        initial_t: float = 0.0,
        initial_step: int = 0,
    ):
        self.config = config
        self.grid = grid
        self.variables = variables
        self.mpv = mpv
        self.time_integrator = time_integrator

        self.current_t: float = initial_t
        self.current_step: int = initial_step
        self.start_time_wall: float = 0.0
```

```
        self.dt: float = config.temporal.dtfixed
        if self.dt <= 0:
            raise ValueError("Solver requires positive dt.")
        self.tmax: float = config.temporal.tmax
        self.stepmax: int = config.temporal.stepmax

        # Attributes used by solver_saver
        self.tout: np.ndarray = np.array(config.temporal.tout)
        self._output_times_saved: np.ndarray = np.zeros_like(self.tout,
dtype=bool)
        self._output_time_tolerance: float = (
            self.dt * 0.51
        )  # Tolerance for hitting output times

        self.output_frequency_steps: int = config.outputs.output_frequency_steps
        self.output_filename: str = self.config.outputs.output_filename

        self.enable_checkpointing: bool = getattr(
            config.outputs, "enable_checkpointing", True
        )
        self.checkpoint_frequency_steps: int = getattr(
            config.outputs, "checkpoint_frequency_steps", 100
        )
        self.checkpoint_filename: str = getattr(
            config.outputs, "checkpoint_filename", "checkpoint_base"
        )  # Default base
        self.checkpoint_keep_n: int = getattr(config.outputs,
"checkpoint_keep_n", 2)

        logging.info(
            f"Solver initialized. Starting at t={self.current_t:.4f},
step={self.current_step}"
        )
        logging.info(
            f"  Output file: {self.output_filename} (freq:
{self.output_frequency_steps} steps, specific times: {self.tout})"
        )

        analysis_flag = self.config.diagnostics.analysis

        logging.info(f"  Analysis output mode: {analysis_flag}")
        logging.info(
            f"  Checkpointing: {self.enable_checkpointing} (freq:
{self.checkpoint_frequency_steps} steps, base: {self.checkpoint_filename}, keep:
{self.checkpoint_keep_n})"
        )

    def run(self) -> None:
        """Executes the main simulation loop."""
        logging.info(
            f"--- Starting simulation run from t={self.current_t:.4f},
step={self.current_step} ---"
        )
        self.start_time_wall = time.time()

        # --- Initial Output/Checkpoint (at initial state) ---
        solver_saver.handle_saving(self)

        ######################################### BEGIN OF THE SOLVER LOOP
#####################################################
        while True:
            if self.current_step > 0:
                logging.info(
                    f" Current time t={self.current_t:.4f},
```

```
step={self.current_step}"
                )

            if (
                self.current_t >= self.tmax - 1e-9
            ):  # Using 1e-9 as a small epsilon for float comparison
                logging.info(f"Reached tmax = {self.tmax:.4f}. Stopping.")
                break
            if self.current_step >= self.stepmax:
                logging.info(f"Reached stepmax = {self.stepmax}. Stopping.")
                break

            pending_output_times =
self.tout[np.logical_not(self._output_times_saved)]
            future_output_times = pending_output_times[
                pending_output_times > self.current_t + MACHINE_EPSILON
            ]

            if future_output_times.size > 0:
                next_target_output_t = np.min(future_output_times)
            else:
                next_target_output_t = self.tmax

            ################################################ COMPUTE TIMESTEP
######################################################
            current_dt: float
            if (
                self.config.temporal.dtfixed0 is not None and self.current_step
== 0
            ):  # Use dtfixed0 for the very first step if specified
                current_dt = self.config.temporal.dtfixed0
            elif (
                self.config.temporal.dtfixed0 is not None
                and self.config.temporal.dtfixed0 > 0
            ):  # Original line was "if self.config.temporal.dtfixed0 is not
None:"
                current_dt = (
                    self.config.temporal.dtfixed
                )  # After first step, use dtfixed if dtfixed0 was present
            else:  # Dynamic dt calculation
                current_dt = calculate_dynamic_dt(
                    self.variables,
                    self.grid,
                    self.config,
                    self.current_t,
                    next_target_output_t,
                    self.current_step,
                )

            # Ensure dt does not overshoot next_target_output_t if dynamic dt is
very flexible
            # This is a common strategy to hit tout precisely.
            if not (
                self.config.temporal.dtfixed0 is not None
                and self.config.temporal.dtfixed0 > 0
            ):  # if not using fixed dt
                if (
                    self.current_t + current_dt
                    > next_target_output_t + self._output_time_tolerance
                ):  # If current_dt overshoots
                    # Check if we are very close, then just take the step to
next_target_output_t
                    if (
                        next_target_output_t - self.current_t > MACHINE_EPSILON
```

```
                    ):  # Ensure positive step
                        current_dt = next_target_output_t - self.current_t
            ################################### TIME INTEGRATION STEP
##############################################################
            self.time_integrator.dt = current_dt  # Update dt for the time
integrator

            step_start_time = time.time()
            try:
                self.time_integrator.step(global_step=self.current_step)
            except Exception as e:
                logging.error(
                    f"Error during step {self.current_step + 1}
(t={self.current_t + current_dt:.4f}): {e}",
                    exc_info=True,
                )
                solver_saver.save_checkpoint(self, force_save=True)
                raise e

            self.current_step += 1
            self.current_t += current_dt

            log_interval_steps = max(1, self.stepmax // 20) if self.stepmax > 0
else 50
            if self.current_step % log_interval_steps == 0:
                logging.info(
                    f"Step: {self.current_step}/{self.stepmax}, Time:
{self.current_t:.4f}/{self.tmax:.4f}, dt: {current_dt:.2e}, Step Time:
{time.time() - step_start_time:.4f}s"
                )

            solver_saver.handle_saving(self)

        end_time_wall = time.time()
        total_time = end_time_wall - self.start_time_wall
        logging.info(f"--- Simulation finished ---")
        logging.info(
            f"Final state: Time={self.current_t:.4f}, Step={self.current_step}"
        )
        logging.info(f"Total wall clock time for this run: {total_time:.2f}
seconds")

    @staticmethod
    def load_checkpoint_data(filename: str) -> Dict[str, Any]:
        """
        Loads simulation state data from a checkpoint file using the
solver_saver module.
        """
        return solver_saver.load_checkpoint_data(filename)
```