"""

Temporal Flow Theory - Numerical Implementation Guide

===================================================

1. Initialization and Setup

"""

import numpy as np

from scipy import sparse

from numba import jit

import torch # For GPU acceleration

class TemporalFlowSolver:

def \_\_init\_\_(self, config):

"""

Initialize the numerical solver with configuration parameters

Parameters:

-----------

config : dict

Contains simulation parameters including:

- spatial\_resolution

- temporal\_resolution

- domain\_size

- boundary\_conditions

- numerical\_scheme

"""

self.dx = config['spatial\_resolution']

self.dt = config['temporal\_resolution']

self.L = config['domain\_size']

self.scheme = config['numerical\_scheme']

# Initialize grids

self.x = np.linspace(0, self.L, int(self.L/self.dx))

self.t = 0.0

# Initialize fields

self.W = np.zeros\_like(self.x) # Temporal flow field

self.rho = np.zeros\_like(self.x) # Density field

# Set up numerical schemes

self.setup\_numerical\_schemes()

def setup\_numerical\_schemes(self):

"""Setup numerical discretization schemes"""

if self.scheme == 'spectral':

self.setup\_spectral\_method()

elif self.scheme == 'finite\_difference':

self.setup\_finite\_difference()

else:

raise ValueError(f"Unknown numerical scheme: {self.scheme}")

@jit(nopython=True)

def compute\_spatial\_derivatives(self, field):

"""

Compute spatial derivatives using chosen numerical scheme

Parameters:

-----------

field : ndarray

Field to compute derivatives for

Returns:

--------

dx\_field : ndarray

First spatial derivative

dx2\_field : ndarray

Second spatial derivative

"""

# Fourth-order central difference for first derivative

dx\_field = (-field[4:] + 8\*field[3:-1] - 8\*field[1:-3] + field[:-4])/(12\*self.dx)

# Fourth-order central difference for second derivative

dx2\_field = (-field[2:] + 16\*field[1:-1] - 30\*field[:-2])/(12\*self.dx\*\*2)

return dx\_field, dx2\_field

def time\_step(self):

"""

Perform one time step using RK4 method

"""

k1 = self.dt \* self.compute\_temporal\_derivative(self.W)

k2 = self.dt \* self.compute\_temporal\_derivative(self.W + 0.5\*k1)

k3 = self.dt \* self.compute\_temporal\_derivative(self.W + 0.5\*k2)

k4 = self.dt \* self.compute\_temporal\_derivative(self.W + k3)

self.W += (k1 + 2\*k2 + 2\*k3 + k4)/6

self.t += self.dt

@jit(nopython=True)

def compute\_temporal\_derivative(self, W):

"""

Compute right-hand side of temporal flow equation

"""

# Compute basic terms

dW\_dx, d2W\_dx2 = self.compute\_spatial\_derivatives(W)

# Scale-dependent coupling

g\_r = 1.0/(1.0 + (self.x/self.r\_c)\*\*self.n)

# Compute quantum and gravitational forces

F\_q = self.compute\_quantum\_force(W)

F\_g = self.compute\_gravitational\_force(W)

# Full temporal derivative

dW\_dt = -g\_r \* W \* dW\_dx - self.grad\_P\_t/self.rho\_t + self.nu\_t \* d2W\_dx2 + F\_q + F\_g

return dW\_dt

class AdaptiveMeshRefinement:

"""

Implement adaptive mesh refinement for regions requiring higher resolution

"""

def \_\_init\_\_(self, base\_resolution, refinement\_criteria):

self.base\_resolution = base\_resolution

self.refinement\_criteria = refinement\_criteria

self.mesh\_levels = []

def refine\_mesh(self, field):

"""

Refine mesh based on field properties

"""

gradients = np.gradient(field)

curvature = np.gradient(gradients)

# Identify regions needing refinement

refinement\_regions = np.where(np.abs(curvature) > self.refinement\_criteria)

# Create refined mesh

refined\_mesh = self.create\_refined\_mesh(refinement\_regions)

return refined\_mesh

class ErrorAnalysis:

"""

Implement error analysis and convergence testing

"""

def \_\_init\_\_(self):

self.error\_metrics = {}

def compute\_convergence\_rate(self, solutions, resolutions):

"""

Compute numerical convergence rate

"""

errors = [np.linalg.norm(sol - solutions[-1]) for sol in solutions[:-1]]

rates = np.log(errors[1:]/errors[:-1]) / np.log(resolutions[1:]/resolutions[:-1])

return rates

def estimate\_numerical\_error(self, solution, exact\_solution=None):

"""

Estimate numerical error using Richardson extrapolation

"""

if exact\_solution is not None:

return np.linalg.norm(solution - exact\_solution)

else:

# Implement Richardson extrapolation

return self.richardson\_extrapolation(solution)

"""

Usage Example:

-------------

# Initialize solver

config = {

'spatial\_resolution': 0.01,

'temporal\_resolution': 0.001,

'domain\_size': 10.0,

'boundary\_conditions': 'periodic',

'numerical\_scheme': 'spectral'

}

solver = TemporalFlowSolver(config)

# Run simulation

for i in range(1000):

solver.time\_step()

# Analyze results

error\_analysis = ErrorAnalysis()

convergence\_rate = error\_analysis.compute\_convergence\_rate(solutions, resolutions)

"""