"""

Non-linear Cluster Evolution with Temporal Flow Effects

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

"""

class NonlinearClusterEvolution:

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

self.simulation\_params = {

'spatial\_resolution': 1.0, # kpc

'time\_resolution': 1e6, # years

'max\_refinement\_level': 8

}

def evolve\_nonlinear\_system(self, initial\_state, time\_span):

"""

Evolve full non-linear cluster dynamics

"""

# Initialize adaptive mesh

mesh = AdaptiveMesh(

base\_resolution=self.simulation\_params['spatial\_resolution'],

max\_level=self.simulation\_params['max\_refinement\_level']

)

# Set up non-linear solver

solver = NonlinearSolver(

temporal\_coupling=self.compute\_temporal\_coupling,

shock\_capture=True,

turbulence\_model='large\_eddy'

)

# Evolution loop with adaptive timesteps

state = initial\_state

time = 0

while time < time\_span:

# Compute timestep based on local conditions

dt = self.compute\_adaptive\_timestep(state)

# Solve non-linear system

state = solver.step(state, dt)

# Handle shock waves and discontinuities

state = self.handle\_shocks(state)

# Update mesh refinement

mesh.adapt(state)

time += dt

return state

def handle\_violent\_relaxation(self, state):

"""

Handle violent relaxation during mergers

"""

# Compute phase-space distribution

f = self.compute\_phase\_space\_distribution(state)

# Temporal flow modifications to violent relaxation

modified\_potential = self.compute\_modified\_potential(state)

# Solve modified Vlasov-Poisson system

f\_evolved = self.solve\_modified\_vlasov\_poisson(f, modified\_potential)

return f\_evolved

def compute\_turbulent\_cascade(self, state):

"""

Compute turbulent energy cascade with temporal effects

"""

# Initialize energy spectrum

E\_k = self.initialize\_energy\_spectrum(state)

# Modified Kolmogorov cascade

for k in self.wavenumbers:

# Standard turbulent terms

cascade\_term = self.compute\_cascade\_term(E\_k, k)

# Temporal flow modifications

temporal\_mod = self.compute\_temporal\_turbulence\_mod(k)

# Combined evolution

E\_k[k] = cascade\_term \* (1.0 + temporal\_mod)

return E\_k

class ShockHandler:

"""

Handle shock waves and discontinuities

"""

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

self.shock\_detector = ShockDetector()

self.riemann\_solver = RiemannSolver()

def process\_shocks(self, state):

"""

Process shock waves with temporal flow effects

"""

# Detect shocks

shock\_locations = self.shock\_detector.find\_shocks(state)

# Modified Rankine-Hugoniot conditions

for shock in shock\_locations:

# Standard jump conditions

jump\_conditions = self.compute\_jump\_conditions(shock)

# Temporal flow modifications

modified\_jump = self.apply\_temporal\_modifications(jump\_conditions)

# Solve modified Riemann problem

solution = self.riemann\_solver.solve(modified\_jump)

state = self.update\_state\_with\_shock(state, solution)

return state