"""

Backreaction Effects in Temporal Flow Theory

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

"""

class BackreactionComputer:

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

self.scale\_hierarchy = {

'local': 1e-3, # Mpc

'cluster': 1.0, # Mpc

'cosmic': 1e3 # Mpc

}

def compute\_backreaction(self, state, scale):

"""

Compute backreaction effects across scales

"""

# Initialize averaging procedure

averager = CovariantAverager(scale)

# Compute local geometric quantities

local\_geometry = self.compute\_local\_geometry(state)

# Average over domain

averaged\_geometry = averager.average(local\_geometry)

# Compute backreaction terms

Q\_D = self.compute\_kinematical\_backreaction(state, averaged\_geometry)

P\_D = self.compute\_dynamical\_backreaction(state, averaged\_geometry)

return {

'kinematical': Q\_D,

'dynamical': P\_D,

'effective\_dynamics': self.compute\_effective\_dynamics(Q\_D, P\_D)

}

def compute\_effective\_dynamics(self, Q\_D, P\_D):

"""

Compute effective dynamics including backreaction

"""

# Modified Friedmann equations

H\_D = self.compute\_effective\_hubble(Q\_D, P\_D)

# Modified acceleration equation

a\_D = self.compute\_effective\_acceleration(Q\_D, P\_D)

# Effective equation of state

w\_eff = self.compute\_effective\_eos(H\_D, a\_D)

return {

'H\_D': H\_D,

'a\_D': a\_D,

'w\_eff': w\_eff

}

def compute\_structure\_formation(self, initial\_state, time\_span):

"""

Compute structure formation with backreaction

"""

# Multi-scale decomposition

scales = self.decompose\_scales(initial\_state)

# Evolution with backreaction

for time in range(time\_span):

# Update each scale

for scale in scales:

# Standard evolution

scale.evolve\_standard()

# Compute backreaction

backreaction = self.compute\_backreaction(scale.state, scale.size)

# Update with backreaction effects

scale.apply\_backreaction(backreaction)

# Couple scales

self.couple\_scales(scales)

return scales

class CovariantAverager:

"""

Implement covariant averaging procedure

"""

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

self.scale = scale

self.window\_function = self.initialize\_window()

def average(self, tensor\_field):

"""

Perform covariant averaging of tensor fields

"""

# Parallel transport

transported = self.parallel\_transport(tensor\_field)

# Window function averaging

averaged = self.window\_average(transported)

# Ensure covariance

return self.ensure\_covariance(averaged)