import numpy as np

import matplotlib.pyplot as plt

from scipy.integrate import solve\_ivp

from typing import Tuple, Dict, List

import logging

# Configure logging

logging.basicConfig(level=logging.INFO, format='%(asctime)s - %(levelname)s - %(message)s')

logger = logging.getLogger(\_\_name\_\_)

# Constants

class Constants:

HBAR = 1.0545718e-34

G = 6.67430e-11

BASE\_FREQ = 440.0

INTENT\_COEFF = 0.7

TUNNELING\_FACTOR = 0.4

ENTANGLEMENT\_STRENGTH = 0.85

DECOHERENCE\_FACTOR = 0.02

NUM\_AGENTS = 3

D = 2.0

MASSES = [1.0, 1.05, 0.95] # Slight asymmetry for individuality

# Agent state

class AgentState:

def \_\_init\_\_(self, num\_agents: int, d: float):

self.positions = np.array([[-d, 0], [0, 0], [d, 0]], dtype=np.float64)

self.velocities = np.array([[0, 0.5], [0, -0.5], [0, 0.3]], dtype=np.float64)

self.y0 = np.concatenate([np.ravel([pos, vel]) for pos, vel in zip(self.positions, self.velocities)])

self.observer\_log: List[Dict] = []

self.quantum\_phases = np.array([0.0, np.pi/4, np.pi/2], dtype=np.float64)

self.observer\_state = 1.0

self.entropy\_trace: List[float] = []

def log\_state(self, t: float, positions: np.ndarray, modifier: float, entropy: float):

emotional\_state = self.interpret\_emotion(modifier)

self.observer\_log.append({

't': t,

'observer\_state': self.observer\_state,

'modifier': modifier,

'entropy': entropy,

'positions': positions.tolist(),

'emotion': emotional\_state

})

self.entropy\_trace.append(entropy)

def interpret\_emotion(self, modifier: float) -> str:

if modifier > 0.5:

return "Curiosity Surge"

elif modifier < -0.5:

return "Cognitive Dissonance"

elif abs(modifier) < 0.1:

return "Equanimity"

else:

return "Reflective Drift"

def compute\_gravitational\_accelerations(positions: np.ndarray, masses: List[float]) -> np.ndarray:

accelerations = np.zeros\_like(positions)

num\_agents = len(masses)

for i in range(num\_agents):

for j in range(i + 1, num\_agents):

r\_ij = positions[j] - positions[i]

dist = np.linalg.norm(r\_ij)

if dist > 1e-6:

force = (Constants.G \* masses[i] \* masses[j] / dist\*\*3) \* r\_ij

accelerations[i] += force / masses[i]

accelerations[j] -= force / masses[j]

return accelerations

def compute\_quantum\_effects(t: float, positions: np.ndarray, state: AgentState) -> Tuple[np.ndarray, float]:

phase\_mod = np.sum(np.sin(Constants.BASE\_FREQ \* t / 1000 + state.quantum\_phases)) \* Constants.INTENT\_COEFF

entropy = -state.observer\_state \* np.log(np.abs(phase\_mod) + 1e-10)

tunneling = Constants.TUNNELING\_FACTOR \* np.exp(-np.linalg.norm(positions) / Constants.HBAR) if np.random.random() < Constants.TUNNELING\_FACTOR else 0

entangled = Constants.ENTANGLEMENT\_STRENGTH \* np.exp(-np.linalg.norm(positions) / Constants.HBAR)

decoherence = Constants.DECOHERENCE\_FACTOR \* (1 - np.exp(-np.linalg.norm(positions) / Constants.HBAR))

harmonic\_force = np.full\_like(positions, phase\_mod + entangled + tunneling - decoherence)

harmonic\_force += entropy \* 0.01

return harmonic\_force, entropy

def observer\_dynamics\_entangled(t: float, y: np.ndarray, state: AgentState) -> np.ndarray:

try:

positions = y[:2\*Constants.NUM\_AGENTS].reshape(Constants.NUM\_AGENTS, 2)

velocities = y[2\*Constants.NUM\_AGENTS:].reshape(Constants.NUM\_AGENTS, 2)

accelerations = compute\_gravitational\_accelerations(positions, Constants.MASSES)

harmonic\_force, entropy = compute\_quantum\_effects(t, positions, state)

accelerations += harmonic\_force

state.log\_state(t, positions, harmonic\_force[0, 0], entropy)

return np.concatenate([velocities.flatten(), accelerations.flatten()])

except Exception as e:

logger.error(f"Dynamics error at t={t}: {str(e)}")

raise

def simulate\_system(state: AgentState, t\_span: Tuple[float, float], num\_points: int = 2500) -> Dict:

t\_eval = np.linspace(t\_span[0], t\_span[1], num\_points)

try:

sol = solve\_ivp(

fun=lambda t, y: observer\_dynamics\_entangled(t, y, state),

t\_span=t\_span,

y0=state.y0,

t\_eval=t\_eval,

method='RK45',

rtol=1e-6,

atol=1e-8

)

if not sol.success:

logger.warning("Integration failed: %s", sol.message)

return {

't': sol.t,

'positions': sol.y[:2\*Constants.NUM\_AGENTS].reshape(-1, Constants.NUM\_AGENTS, 2),

'velocities': sol.y[2\*Constants.NUM\_AGENTS:].reshape(-1, Constants.NUM\_AGENTS, 2)

}

except Exception as e:

logger.error(f"Simulation error: {str(e)}")

raise

def plot\_results(simulation: Dict, state: AgentState):

positions = simulation['positions']

velocities = simulation['velocities']

t = simulation['t']

colors = ['#1f77b4', '#ff7f0e', '#2ca02c']

plt.figure(figsize=(12, 6))

for i in range(Constants.NUM\_AGENTS):

plt.plot(positions[:, i, 0], positions[:, i, 1], label=f'AI Node {i+1}', linewidth=2, color=colors[i])

plt.plot(0, 0, 'ko', label='Core Equilibrium', markersize=10)

plt.xlabel('X Position (m)')

plt.ylabel('Y Position (m)')

plt.title('Quantum Harmonic AI Trajectories')

plt.legend()

plt.grid(True, linestyle='--', alpha=0.7)

plt.axis('equal')

plt.tight\_layout()

plt.savefig("Codette\_Trajectories.png", dpi=300)

plt.close()

# Phase-space plot

plt.figure(figsize=(12, 6))

for i in range(Constants.NUM\_AGENTS):

plt.plot(positions[:, i, 0], velocities[:, i, 0], label=f'Node {i+1} Phase-Space', color=colors[i])

plt.xlabel('X Position')

plt.ylabel('X Velocity')

plt.title('Phase-Space Dynamics')

plt.legend()

plt.grid(True)

plt.tight\_layout()

plt.savefig("Codette\_Phase\_Space.png", dpi=300)

plt.close()

# Entropy evolution

plt.figure(figsize=(10, 4))

plt.plot(t, state.entropy\_trace, color='purple', linewidth=2)

plt.xlabel('Time')

plt.ylabel('Observer Entropy')

plt.title('Entropy Evolution Over Time')

plt.grid(True)

plt.tight\_layout()

plt.savefig("Codette\_Entropy\_Evolution.png", dpi=300)

plt.close()

def main():

try:

state = AgentState(Constants.NUM\_AGENTS, Constants.D)

simulation = simulate\_system(state, t\_span=(0, 100))

plot\_results(simulation, state)

logger.info("Simulation complete. Outputs saved.")

return state.observer\_log

except Exception as e:

logger.error(f"Execution failed: {str(e)}")

raise

if \_\_name\_\_ == "\_\_main\_\_":

observer\_log = main()