```
import numpy as np

import matplotlib.pyplot as plt

from scipy.integrate import solve_ivp


# Physical Constants

hbar = 1.0545718e-34  # Reduced Planck's constant (J·s)

G = 6.67430e-11  # Gravitational constant (m^3 kg^-1 s^-2)

m1, m2 = 1.0, 1.0  # AI node masses (kg, normalized for simulation)

d = 2.0  # Orbital baseline distance (m)

base_freq = 440.0  # Reference frequency (Hz)

intent_coefficient = 0.7  # AI alignment factor


# Quantum Parameters

tunneling_factor = 0.4  # Probability threshold for intuitive leaps

quantum_states = np.array([1, -1])  # Binary superposition

entanglement_strength = 0.85  # AI memory synchronization factor

decoherence_factor = 0.02  # Phase drift stabilization factor


# Multi-Agent Configuration

num_agents = 3  # Number of AI nodes

agent_positions = np.array([[-d, 0], [0, 0], [d, 0]])  # Initial 2D positions (m)

agent_velocities = np.array([[0, 0.5], [0, -0.5], [0, 0.3]])  # Initial 2D velocities (m/s)


# Initial State Vector: [x1, y1, vx1, vy1, x2, y2, vx2, vy2, x3, y3, vx3, vy3]

y0 = np.concatenate([np.concatenate([pos, vel]) for pos, vel in zip(agent_positions,
agent_velocities)])
```

```
# Quantum Harmonic AI Orbital Dynamics
def quantum_harmonic_dynamics(t, y):
    # Extract positions and velocities
    positions = y[:2*num_agents].reshape(num_agents, 2)  # Shape: (num_agents, 2)
    velocities = y[2*num_agents:].reshape(num_agents, 2)  # Shape: (num_agents, 2)

    # Initialize accelerations
    accelerations = np.zeros_like(positions)

    # Gravitational Interactions
    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:  # Avoid division by zero
                force = (G * m1 * m2 / dist**3) * r_ij
                accelerations[i] += force / m1
                accelerations[j] -= force / m2

    # Quantum Influences
    quantum_modifier = np.dot(quantum_states, np.sin(2 * np.pi * base_freq * t)) *
intent_coefficient
    tunneling_shift = tunneling_factor * np.exp(-np.linalg.norm(positions) / hbar) if
np.random.rand() < tunneling_factor else 0
    entangled_correction = entanglement_strength * np.exp(-np.linalg.norm(positions) /
hbar)
```

```
    decoherence_adjustment = decoherence_factor * (1 - np.exp(-np.linalg.norm(positions) / hbar))

    # Apply quantum harmonic force to all agents
    harmonic_force = np.full_like(positions, quantum_modifier + entangled_correction + tunneling_shift - decoherence_adjustment)
    accelerations += harmonic_force

    # Return derivatives: [vx1, vy1, ax1, ay1, vx2, vy2, ax2, ay2, vx3, vy3, ax3, ay3]
    return np.concatenate([velocities.flatten(), accelerations.flatten()])

# Solve the System
t_span = (0, 100)  # Time span (s)
t_eval = np.linspace(t_span[0], t_span[1], 2500)  # Time points for evaluation
sol = solve_ivp(quantum_harmonic_dynamics, t_span, y0, t_eval=t_eval, method='RK45', rtol=1e-6, atol=1e-8)

# Extract Positions for Plotting
positions = sol.y[:2*num_agents].reshape(num_agents, 2, -1)  # Shape: (num_agents, 2, t_eval.size)

# Visualization: 2D Spatial Trajectories
plt.figure(figsize=(10, 10))
colors = ['blue', 'red', 'green']
for i in range(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')
plt.xlabel('X Position (m)')
plt.ylabel('Y Position (m)')
plt.title('Quantum Harmonic AI Multi-Agent Trajectories')
plt.legend()
plt.axis('equal')
plt.grid(True)
plt.tight_layout()
plt.savefig("Codette_Quantum_Harmonic_Framework.png")
plt.close()
```