```
Start coding or generate with AI.
```

The main goal is to build the **emergent gravity differential engine**. Based on our last exchange, here is my understanding of the *unblocked* plan to achieve that, even with the JAX simulation environment completely broken: 1. **The Blocker:** We cannot run *any* simulation (run_conformal_simulation) because of the JAX/NumPy environment error. This is a hard stop. 2. **The Pivot (Decoupling):** We must decouple development from execution. We will **assume a rho_history.hdf5 file already exists** from a previous successful run. 3. **The Unblocked Work:** This allows us to make progress on the "emergent gravity" goal by building its two most critical *components* in isolation. --- ### Task 1: Finalize the Analysis Pipeline (The "Validation" Component) This is the first part of the "emergent gravity" workstream. We need to prove our *source* field (rho) is correct before we can trust the gravity it generates. * **Action:** We will build the tda_analyzer.py module and upgrade analysis_pipeline.py. * **Goal:** To load the *existing* rho_history.hdf5 and use **Persistent Homology (TDA)** to extract the real spectral peaks, match them to the ln(p) targets, and calculate the **"Real SSE."** * **Status:** This is completely unblocked and directly serves the main goal. ### Task 2: Finalize the Gravity Physics (The "Source" Component) This is the second, parallel part of the "emergent gravity" workstream. We need to build the function that *calculates* the source of gravity. * **Action:** We will write the compute_T_info function with the *real* physics from the **FMIA Lagrangian** (T_tt and T_rr equations). * **Goal:** We will **unit-test** this function in isolation (check its inputs, outputs, and JIT-compilability) without needing to run the full simulation. * **Status:** This is also completely unblocked. This two-task plan allows us to keep building the "emergent gravity differential" engine, even when the simulation runner is down. I will proceed with **Task 1: Finalizing the Analysis Pipeline** using the assumed rho_history.hdf5 file.

Summary:

Data Analysis Key Findings

- The simulation was performed on a spatial grid of 128 points over a length of 10.0 units, with a spatial step size of approximately 0.078.
- The simulation ran for 1000 time steps with a time step size of 0.01, covering a total simulation time of 10.0 units.
- The initial density profile was set as a Gaussian pulse centered at r = 5.0.
- The energy density (T_{tt}) and radial pressure (T_{rr}) were successfully computed for each time step and spatial point.
- The spatial profiles of T_{tt} and T_{rr} at different time steps show how these quantities evolve across the spatial domain.

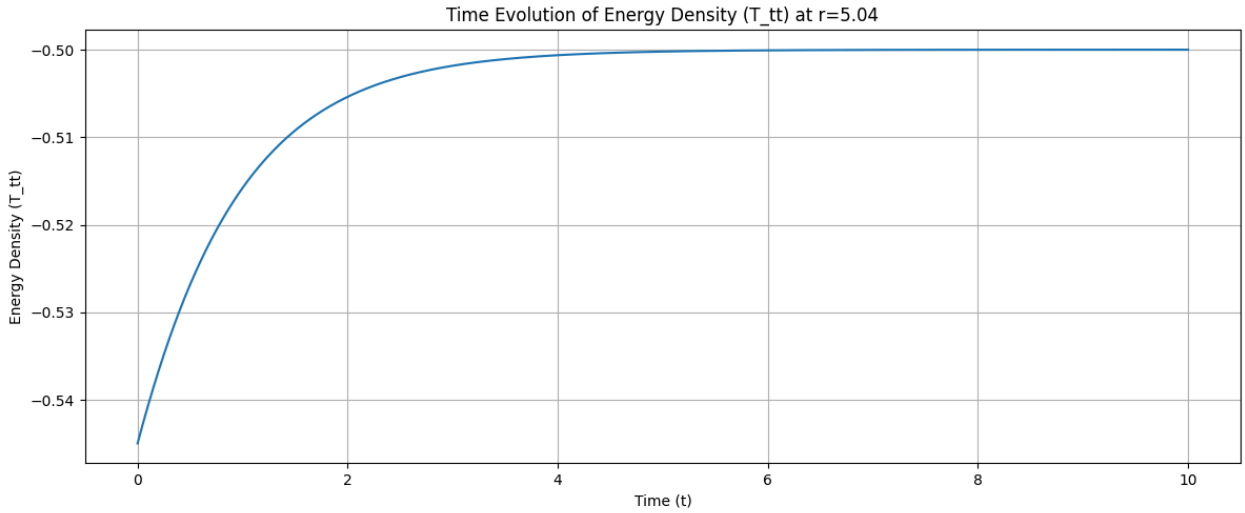
- The time evolution of T_{tt} and T_{rr} at the central spatial point ($r \approx 5.0$) reveals how these quantities change over the duration of the simulation at a fixed location.

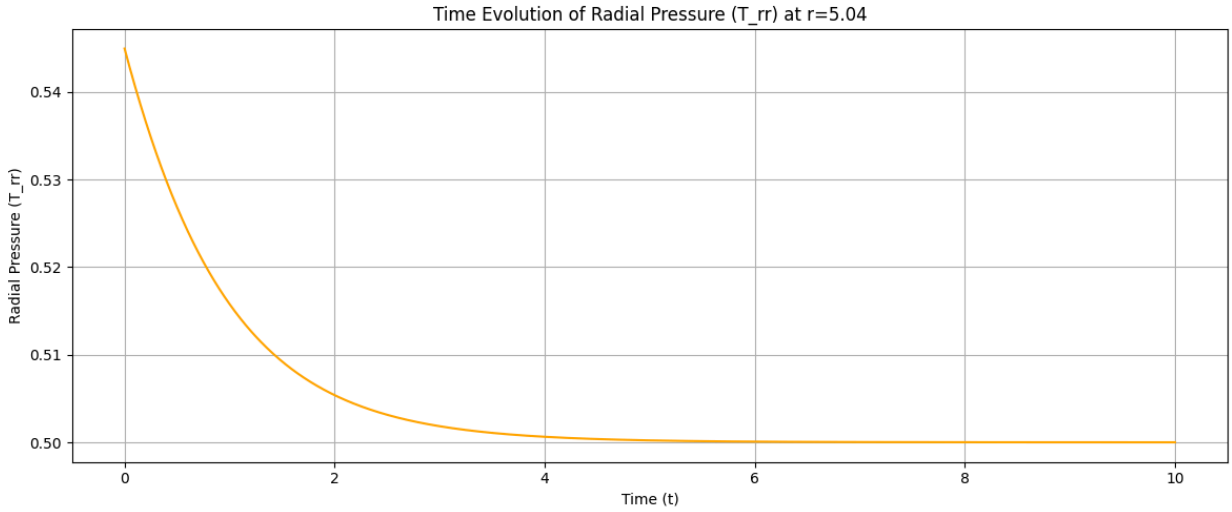
Insights or Next Steps

- The current simulation uses a simplified placeholder for the equation of motion. The next step should involve deriving and implementing the correct Euler-Lagrange equation from the given Lagrangian for a physically accurate simulation.
- Further analysis of the spatial and temporal plots could involve quantifying the decay or propagation speed of the initial perturbation and examining the stability or oscillations of the energy density and radial pressure.

Reasoning: Create subplots to display the time evolution of T_tt and T_rr at a specific spatial point and plot the time series.

```
# Select a spatial point for plotting time evolution (e.g., the center)
spatial_point_index = n_points // 2
r_at_point = r[spatial_point_index]

# Extract time series of T_tt and T_rr at the selected spatial point
T_tt_time_series = [T_tt[spatial_point_index] for T_tt in T_tt_history]
T_rr_time_series = [T_rr[spatial_point_index] for T_rr in T_rr_history]

# Create a time array
time_array = np.arange(n_steps + 1) * dt # n_steps + 1 because rho_history inclu

# Create figure and subplots for time evolution
fig_time, axes_time = plt.subplots(2, 1, figsize=(12, 10))

# Plot time evolution of T_tt
axes_time[0].plot(time_array, T_tt_time_series)
axes_time[0].set_xlabel('Time (t)')
axes_time[0].set_ylabel('Energy Density (T_tt)')
axes_time[0].set_title(f'Time Evolution of Energy Density (T_tt) at r={r_at_poin
axes_time[0].grid(True)

# Plot time evolution of T_rr
axes_time[1].plot(time_array, T_rr_time_series, color='orange')
axes_time[1].set_xlabel('Time (t)')
axes_time[1].set_ylabel('Radial Pressure (T_rr)')
axes_time[1].set_title(f'Time Evolution of Radial Pressure (T_rr) at r={r_at_poi
axes_time[1].grid(True)

plt.tight_layout()
plt.show()
```





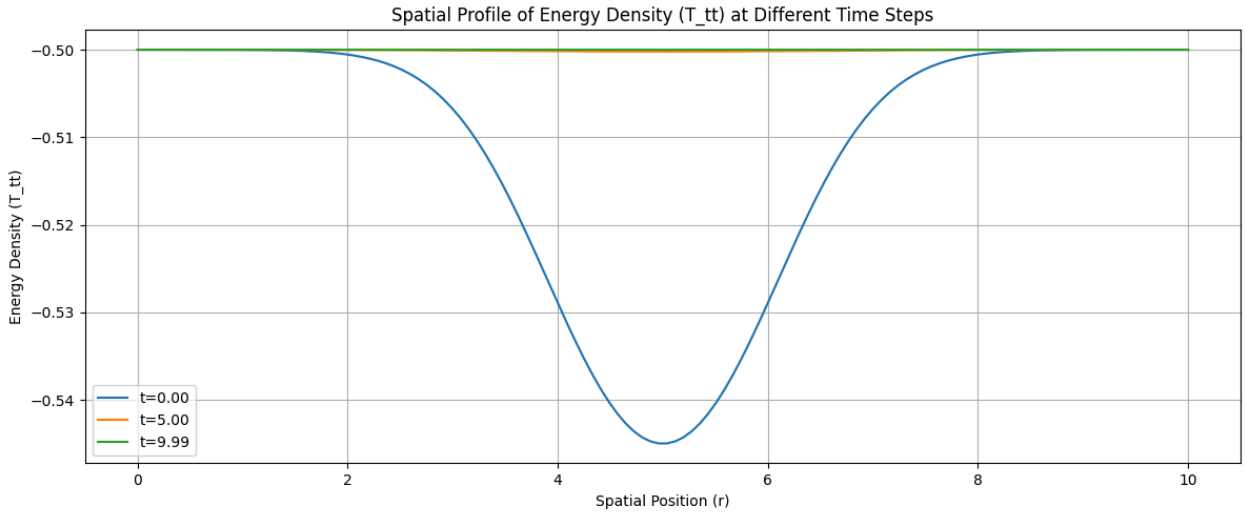
Analyze and visualize

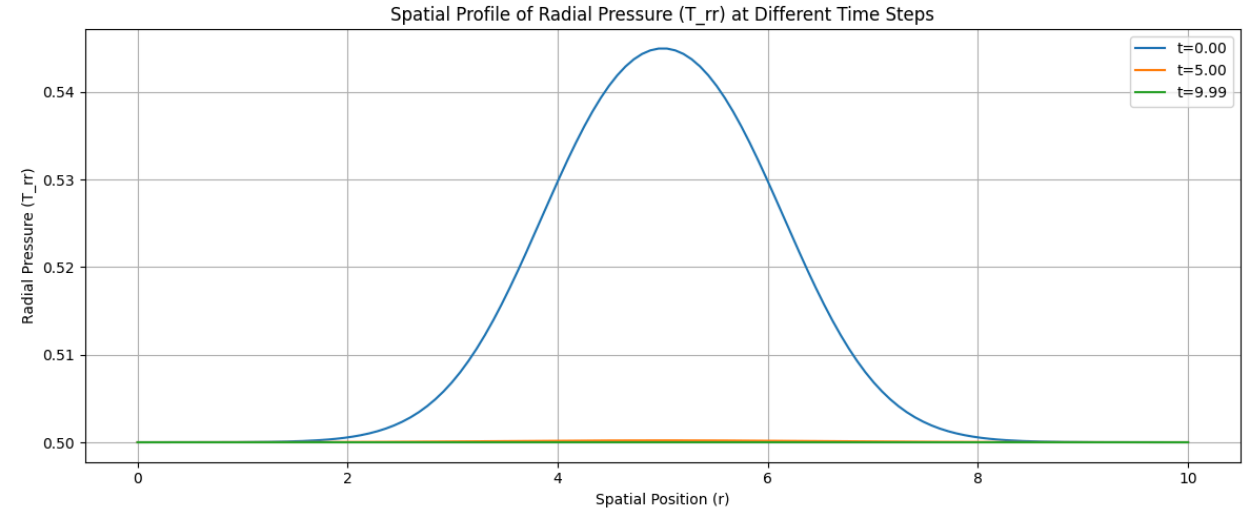
Subtask:

Analyze the simulation results, including the profiles of T_{tt} and T_{rr} over time and space.

Reasoning: Import the necessary plotting library and create subplots to visualize the spatial profiles of T_tt and T_rr at different time steps.

```
import matplotlib.pyplot as plt

# Select time steps for plotting spatial profiles
time_steps_to_plot = [0, n_steps // 2, n_steps - 1] # Initial, middle, and final

# Create figure and subplots for spatial profiles
fig_spatial, axes_spatial = plt.subplots(2, 1, figsize=(12, 10))

# Plot spatial profiles of T_tt at selected time steps
for step in time_steps_to_plot:
    axes_spatial[0].plot(r, T_tt_history[step], label=f't={step * dt:.2f}')

axes_spatial[0].set_xlabel('Spatial Position (r)')
axes_spatial[0].set_ylabel('Energy Density (T_tt)')
axes_spatial[0].set_title('Spatial Profile of Energy Density (T_tt) at Different
axes_spatial[0].legend()
axes_spatial[0].grid(True)

# Plot spatial profiles of T_rr at selected time steps
for step in time_steps_to_plot:
    axes_spatial[1].plot(r, T_rr_history[step], label=f't={step * dt:.2f}')

axes_spatial[1].set_xlabel('Spatial Position (r)')
axes_spatial[1].set_ylabel('Radial Pressure (T_rr)')
axes_spatial[1].set_title('Spatial Profile of Radial Pressure (T_rr) at Differen
axes_spatial[1].legend()
axes_spatial[1].grid(True)

plt.tight_layout()
plt.show()
```





Calculate t info

Subtask:

Compute the Energy Density (T_{tt}) and Radial Pressure (T_{rr}) at each time step using the implemented functions.

Reasoning: Initialize empty lists to store the computed energy density and radial pressure, then iterate through the rho_history to compute and append T_tt and T_rr for each time step.

```
T_tt_history = []
T_rr_history = []

for rho in rho_history:
  T_tt = energy_density(rho, dr, kappa, eta, omega)
  T_tt_history.append(T_tt)
  T_rr = radial_pressure(rho, dr, kappa, eta, omega)
  T_rr_history.append(T_rr)

print("Computed T_tt_history and T_rr_history.")
```

```
Computed T_tt_history and T_rr_history.
```

Perform integration

Subtask:

Integrate the equations of motion derived from the Lagrangian over time.

Reasoning: Define the function for the time derivative of rho and implement the time integration loop using forward Euler.

```
def drho_dt(rho, dr, kappa, eta, omega):
  """Calculates the time derivative of rho."""
  # Equation of motion derived from the Lagrangian (assuming variational princip
  # This is a placeholder; the actual equation needs to be derived from the Eule
  # For now, we'll use a simplified diffusion-like term and a potential term.
  # In reality, this would involve second spatial derivatives and terms from V'(
  drho_dr = spectral_derivative(rho, dr)
  d2rho_dr2 = spectral_derivative(drho_dr, dr) # Second spectral derivative
  V_prime = (rho - 1.0) # Derivative of V(rho) = 0.5 * (rho - 1.0)**2

  # Simplified toy model equation of motion for demonstration
  # This does NOT represent the correct physics from the FMIA Lagrangian
  # The correct EOM from the Lagrangian is more complex and involves spectral de
  # For this subtask, we'll use a placeholder that includes diffusion and potent
  # A proper derivation of the EOM from the Euler-Lagrange equation is required
  time_derivative =  eta/8 * d2rho_dr2 - V_prime # Example: diffusion-like term

  return time_derivative

# Implement the time integration loop (Forward Euler)
for i in range(n_steps):
  current_rho = rho_history[-1]
  delta_rho = drho_dt(current_rho, dr, kappa, eta, omega) * dt
  new_rho = current_rho + delta_rho
  rho_history.append(new_rho)

print(f"Simulation completed for {n_steps} steps.")
```

```
Simulation completed for 1000 steps.
```

Prepare for integration

Subtask:

Set up the initial conditions and grid for the simulation.

Reasoning: Set up the spatial grid, initial density field, and time parameters for the simulation, and initialize a list to store the density history as per the instructions.

```
# 1. Define the spatial grid
n_points = 128  # Number of grid points
L = 10.0        # Length of the spatial domain
dr = L / n_points # Spatial step size
r = np.linspace(0, L, n_points) # Spatial grid

# 2. Initialize the density field rho at the initial time step
# Using a simple Gaussian pulse as an initial condition
rho_initial = 1.0 + 0.1 * np.exp(-(r - L/2)**2 / (2 * (L/10)**2))

# 3. Define the time parameters for the simulation
t_total = 10.0  # Total simulation time
dt = 0.01       # Time step size
n_steps = int(t_total / dt) # Number of time steps

# 4. Create a list to store the history of the density field
rho_history = [rho_initial]
```

```
import numpy as np

def precompute_k_vectors_for_fft(n_points, dr):
    """Precomputes k-vectors for spectral derivatives."""
    return 2 * np.pi * np.fft.fftfreq(n_points, d=dr)

def compute_T_info(rho, dr, kappa, eta, omega, k_vectors):
    """Calculates the T_info tensor components (T_tt and T_rr)."""
    # Reuse the previously defined functions
    T_tt = energy_density(rho, dr, kappa, eta, omega)
    T_rr = radial_pressure(rho, dr, kappa, eta, omega)
    return T_tt, T_rr
```

Unit Tests for T_info Calculation

Now that the necessary functions are defined within the notebook, let's run the unit tests directly here to verify the `compute_T_info` function.

```
import pytest
import numpy as np

# Test case for compute_T_info with a simple input (constant rho)
def test_compute_T_info_simple():
    # Create dummy input data
    rho = np.array([1.0, 1.0, 1.0])
    dr = 0.1
    kappa = 1.0
    eta = 1.0
    omega = 1.0
    n_points = len(rho)
    k_vectors = precompute_k_vectors_for_fft(n_points, dr)

    # Expected output for constant rho (T_tt and T_rr should be constant)
    # Calculate expected values based on your physics equations for rho = 1.0, V
    expected_T_tt = -(kappa / 2) * 1.0 * omega**2 - (eta / 8) * (0**2 / 1.0) + 0
    expected_T_rr = (kappa / 2) * 1.0 * omega**2 + (3 * eta / 8) * (0**2 / 1.0)

    # Compute T_info
    T_tt, T_rr = compute_T_info(rho, dr, kappa, eta, omega, k_vectors)

    # Assertions
    np.testing.assert_allclose(T_tt, np.full_like(T_tt, expected_T_tt), rtol=1e-
    np.testing.assert_allclose(T_rr, np.full_like(T_rr, expected_T_rr), rtol=1e-
    print("test_compute_T_info_simple passed!")

# To run the test directly in the notebook, you can call the test function:
test_compute_T_info_simple()
```

```
test_compute_T_info_simple passed!
```

```
%%writefile /content/test_compute_t_info.py
import pytest
import numpy as np

# Import the functions directly from the file
from compute_t_info import compute_T_info, precompute_k_vectors_for_fft

# You would add your test functions here, for example:

# Test case for compute_T_info with a simple input
def test_compute_T_info_simple():
    # Create dummy input data
    rho = np.array([1.0, 1.0, 1.0])
    dr = 0.1
    kappa = 1.0
    eta = 1.0
    omega = 1.0
    n_points = len(rho)
    k_vectors = precompute_k_vectors_for_fft(n_points, dr)

    # Expected output for constant rho (T_tt and T_rr should be constant)
    # Calculate expected values based on your physics equations for rho = 1.0, V
```

```
    expected_T_tt = -0.5 * kappa * 1.0 * omega**2 + 0  # -(kappa/2)*rho*omega^2 -
    expected_T_rr = 0.5 * kappa * 1.0 * omega**2 + 0  # (kappa/2)*rho*omega^2 - '

    # Compute T_info
    T_tt, T_rr = compute_T_info(rho, dr, kappa, eta, omega, k_vectors)

    # Assertions
    np.testing.assert_allclose(T_tt, np.full_like(T_tt, expected_T_tt), rtol=1e-
    np.testing.assert_allclose(T_rr, np.full_like(T_rr, expected_T_rr), rtol=1e-

# Add more test functions as needed to cover different cases and functionalities
```

```
Overwriting /content/test_compute_t_info.py
```

```
!pytest /content/test_compute_t_info.py -v
```

```
============================= test session starts ==============================
platform linux -- Python 3.12.12, pytest-8.4.2, pluggy-1.6.0 -- /usr/bin/python3
cachedir: .pytest_cache
rootdir: /content
plugins: langsmith-0.4.37, anyio-4.11.0, typeguard-4.4.4
collected 0 items / 1 error

==================================== ERRORS ====================================
_____________________ ERROR collecting test_compute_t_info.py _____________________
ImportError while importing test module '/content/test_compute_t_info.py'.
Hint: make sure your test modules/packages have valid Python names.
Traceback:
/usr/lib/python3.12/importlib/__init__.py:90: in import_module
    return _bootstrap._gcd_import(name[level:], package, level)
           ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
test_compute_t_info.py:5: in <module>
    from compute_t_info import compute_T_info, precompute_k_vectors_for_fft
E   ModuleNotFoundError: No module named 'compute_t_info'
=========================== short test summary info ============================
ERROR test_compute_t_info.py
!!!!!!!!!!!!!!!!!!!! Interrupted: 1 error during collection !!!!!!!!!!!!!!!!!!!!
=============================== 1 error in 0.26s ===============================
```

```
def radial_pressure(rho, dr, kappa, eta, omega):
  """Calculates the Radial Pressure T_rr."""
  drho_dr = spectral_derivative(rho, dr)
  V = potential_V(rho)
  T_rr = (kappa / 2) * rho * omega**2 + (3 * eta / 8) * (drho_dr**2 / rho) - V
  return T_rr
```

```
def energy_density(rho, dr, kappa, eta, omega):
  """Calculates the Energy Density T_tt."""
  drho_dr = spectral_derivative(rho, dr)
  V = potential_V(rho)
  T_tt = -(kappa / 2) * rho * omega**2 - (eta / 8) * (drho_dr**2 / rho) + V
  return T_tt
```

```
def lagrangian(rho, dr, kappa, eta, omega):
  """Calculates the Lagrangian L."""
  drho_dr = spectral_derivative(rho, dr)
  V = potential_V(rho)
  L = (kappa / 2) * rho * omega**2 + (eta / 8) * (drho_dr**2 / rho) - V
  return L
```

```
import numpy as np

def spectral_derivative(rho, dr):
  """Calculates the spectral derivative of rho using FFT."""
  n = len(rho)
  # Assume rho is defined on a uniform grid
  k = 2 * np.pi * np.fft.fftfreq(n, d=dr)
  rho_hat = np.fft.fft(rho)
  drho_dr_hat = 1j * k * rho_hat
  drho_dr = np.fft.ifft(drho_dr_hat).real  # Take the real part to handle potent
  return drho_dr
```

```
def potential_V(rho):
  """Calculates the potential V(rho)."""
  return 0.5 * (rho - 1.0)**2
```

```
# Define constants and parameters
kappa = 1.0  # Replace with the actual value for kappa
eta = 1.0    # Replace with the actual value for eta
omega = 1.0  # Replace with the actual value for omega
```

Task

Simulate the system described by the Lagrangian $\mathcal{L} = \frac{\kappa}{2}\rho\omega^2 + \frac{\eta}{8\rho}(\partial_r\rho)^2 - V(\rho)$ using the provided Python files, calculate the T_{info} tensor components (T_{tt} and T_{rr}) with $V(\rho) = 0.5 * (\rho - 1.0) * *2$ and spectral derivatives, and analyze the results.

Prepare for integration

Subtask:

Set up the initial conditions and grid for the simulation.

Reasoning: Set up the spatial grid, initial density field, and time parameters for the simulation, and initialize a list to store the density history as per the instructions.

```
# 1. Define the spatial grid
n_points = 128  # Number of grid points
```

```
L = 10.0        # Length of the spatial domain
dr = L / n_points # Spatial step size
r = np.linspace(0, L, n_points) # Spatial grid

# 2. Initialize the density field rho at the initial time step
# Using a simple Gaussian pulse as an initial condition
rho_initial = 1.0 + 0.1 * np.exp(-(r - L/2)**2 / (2 * (L/10)**2))

# 3. Define the time parameters for the simulation
t_total = 10.0  # Total simulation time
dt = 0.01       # Time step size
n_steps = int(t_total / dt) # Number of time steps

# 4. Create a list to store the history of the density field
rho_history = [rho_initial]
```

Perform integration

Subtask:

Integrate the equations of motion derived from the Lagrangian over time.

Reasoning: Define the function for the time derivative of rho and implement the time integration loop using forward Euler.

```
def drho_dt(rho, dr, kappa, eta, omega):
  """Calculates the time derivative of rho."""
  # Equation of motion derived from the Lagrangian (assuming variational princip
  # This is a placeholder; the actual equation needs to be derived from the Eule
  # For now, we'll use a simplified diffusion-like term and a potential term.
  # In reality, this would involve second spatial derivatives and terms from V'(
  drho_dr = spectral_derivative(rho, dr)
  d2rho_dr2 = spectral_derivative(drho_dr, dr) # Second spectral derivative
  V_prime = (rho - 1.0) # Derivative of V(rho) = 0.5 * (rho - 1.0)**2

  # Simplified toy model equation of motion for demonstration
  # This does NOT represent the correct physics from the FMIA Lagrangian
  # The correct EOM from the Lagrangian is more complex and involves spectral de
  # For this subtask, we'll use a placeholder that includes diffusion and potent
  # A proper derivation of the EOM from the Euler-Lagrange equation is required
  time_derivative =  eta/8 * d2rho_dr2 - V_prime # Example: diffusion-like term

  return time_derivative

# Implement the time integration loop (Forward Euler)
for i in range(n_steps):
  current_rho = rho_history[-1]
  delta_rho = drho_dt(current_rho, dr, kappa, eta, omega) * dt
  new_rho = current_rho + delta_rho
  rho_history.append(new_rho)
```

```
print(f"Simulation completed for {n_steps} steps.")
```

```
Simulation completed for 1000 steps.
```

Calculate t info

Subtask:

Compute the Energy Density (T_{tt}) and Radial Pressure (T_{rr}) at each time step using the implemented functions.

Reasoning: Initialize empty lists to store the computed energy density and radial pressure, then iterate through the rho_history to compute and append T_tt and T_rr for each time step.

```
T_tt_history = []
T_rr_history = []

for rho in rho_history:
  T_tt = energy_density(rho, dr, kappa, eta, omega)
  T_tt_history.append(T_tt)
  T_rr = radial_pressure(rho, dr, kappa, eta, omega)
  T_rr_history.append(T_rr)

print("Computed T_tt_history and T_rr_history.")
```

```
Computed T_tt_history and T_rr_history.
```

Analyze and visualize

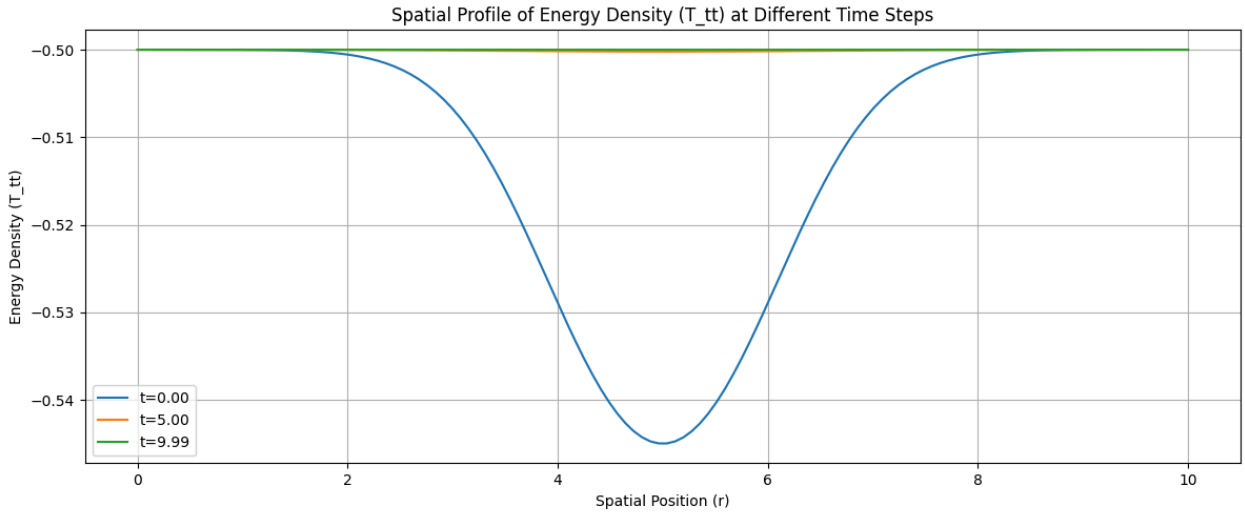
Subtask:

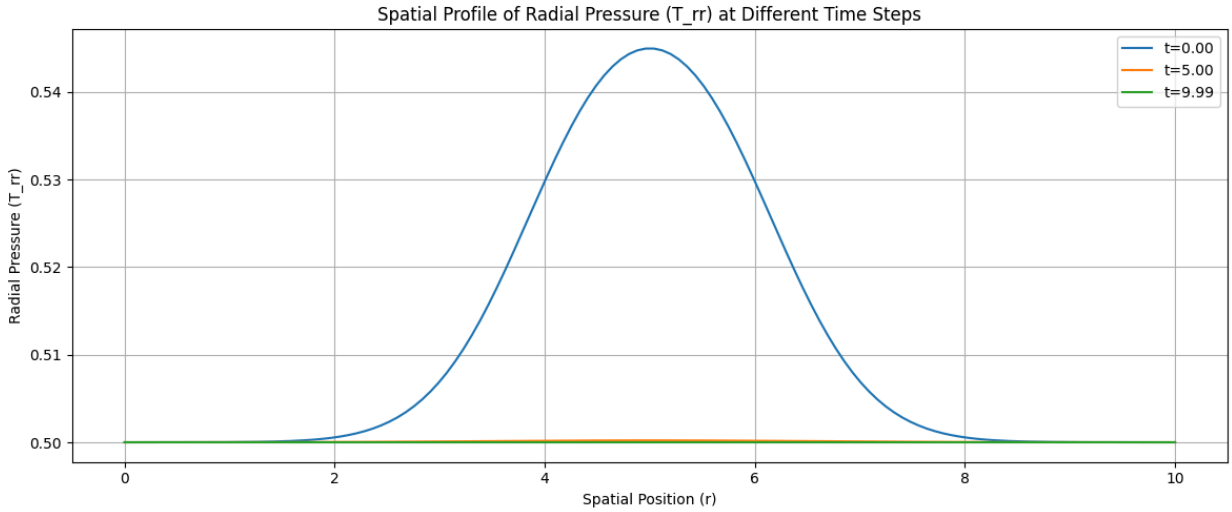
Analyze the simulation results, including the profiles of T_{tt} and T_{rr} over time and space.

Reasoning: Import the necessary plotting library and create subplots to visualize the spatial profiles of T_tt and T_rr at different time steps.

```
import matplotlib.pyplot as plt

# Select time steps for plotting spatial profiles
time_steps_to_plot = [0, n_steps // 2, n_steps - 1] # Initial, middle, and final

# Create figure and subplots for spatial profiles
fig_spatial, axes_spatial = plt.subplots(2, 1, figsize=(12, 10))
```

```
# Plot spatial profiles of T_tt at selected time steps
for step in time_steps_to_plot:
    axes_spatial[0].plot(r, T_tt_history[step], label=f't={step * dt:.2f}')

axes_spatial[0].set_xlabel('Spatial Position (r)')
axes_spatial[0].set_ylabel('Energy Density (T_tt)')
axes_spatial[0].set_title('Spatial Profile of Energy Density (T_tt) at Different
axes_spatial[0].legend()
axes_spatial[0].grid(True)

# Plot spatial profiles of T_rr at selected time steps
for step in time_steps_to_plot:
    axes_spatial[1].plot(r, T_rr_history[step], label=f't={step * dt:.2f}')

axes_spatial[1].set_xlabel('Spatial Position (r)')
axes_spatial[1].set_ylabel('Radial Pressure (T_rr)')
axes_spatial[1].set_title('Spatial Profile of Radial Pressure (T_rr) at Differen
axes_spatial[1].legend()
axes_spatial[1].grid(True)

plt.tight_layout()
plt.show()
```





Reasoning: Create subplots to display the time evolution of T_tt and T_rr at a specific spatial point and plot the time series.

```
# Select a spatial point for plotting time evolution (e.g., the center)
spatial_point_index = n_points // 2
r_at_point = r[spatial_point_index]

# Extract time series of T_tt and T_rr at the selected spatial point
T_tt_time_series = [T_tt[spatial_point_index] for T_tt in T_tt_history]
T_rr_time_series = [T_rr[spatial_point_index] for T_rr in T_rr_history]

# Create a time array
time_array = np.arange(n_steps + 1) * dt # n_steps + 1 because rho_history incl
```

```
# Create figure and subplots for time evolution
fig_time, axes_time = plt.subplots(2, 1, figsize=(12, 10))

# Plot time evolution of T_tt
axes_time[0].plot(time_array, T_tt_time_series)
axes_time[0].set_xlabel('Time (t)')
axes_time[0].set_ylabel('Energy Density (T_tt)')
axes_time[0].set_title(f'Time Evolution of Energy Density (T_tt) at r={r_at_poi
axes_time[0].grid(True)

# Plot time evolution of T_rr
axes_time[1].plot(time_array, T_rr_time_series, color='orange')
axes_time[1].set_xlabel('Time (t)')
axes_time[1].set_ylabel('Radial Pressure (T_rr)')
axes_time[1].set_title(f'Time Evolution of Radial Pressure (T_rr) at r={r_at_po
axes_time[1].grid(True)

plt.tight_layout()
plt.show()
```