



quiero un programa en python para resolver la ecuación de Schrodinger dependiente del tiempo, en un problema unidimensional



La ecuación de Schrödinger dependiente del tiempo en una dimensión se puede resolver numéricamente usando varios métodos. Uno de los métodos más comunes es el método de diferencias finitas. Aquí te proporcionaré un ejemplo de cómo puedes implementar esto en Python.

Para este ejemplo, asumiremos una partícula en una caja infinita de potencial, donde el potencial $V(x)$ es cero dentro de la caja y es infinito en las paredes de la caja. Utilizaremos condiciones de frontera en $x = 0$ y $x = L$.

Primero, importamos las librerías necesarias y definimos algunas funciones auxiliares:

```
```python
import numpy as np
import matplotlib.pyplot as plt
```

```
Parámetros del problema
L = 10.0 # Longitud de la caja
```



This conversation is archived. To continue, please unarchive it first.

Unarchive





ChatGPT



Explore GPTs

```
N = 1000 # Número de puntos espaciales
dx = L / (N - 1) # Paso espacial
x = np.linspace(0, L, N) # Puntos espaciales

Parámetros de tiempo
T = 1.0 # Tiempo total
dt = 0.001 # Paso temporal
Nt = int(T / dt) # Número de puntos temporales

Parámetros físicos
hbar = 1.0 # Constante de Planck reducida
m = 1.0 # Masa de la partícula

Condición inicial: una gaussiana centrada en L/2 con una anchura sigma
sigma = 1.0
k0 = 5.0 # Número de onda inicial
psi0 = np.exp(-(x - L/2)**2 / (2 * sigma**2)) * np.exp(1j * k0 * x)
psi0 /= np.sqrt(np.trapz(np.abs(psi0)**2, x)) # Normalización

Potencial: caja infinita
V = np.zeros_like(x)
V[0] = V[-1] = 1e10 # Paredes infinitamente altas
...
```

This conversation is archived. To continue, please unarchive it first.

[Unarchive](#)

Upgrade plan

Get GPT-4, DALL·E, and more





```
Potencial: caja infinita
```

```
V = np.zeros_like(x)
```

```
V[0] = V[-1] = 1e10 # Paredes infinitamente altas
```

```
...
```

A continuación, definimos una función para evolucionar la ecuación de Schrödinger usando el método de Crank-Nicolson:

```
```python
```

```
def evolve_schrodinger(psi0, V, x, dx, dt, Nt):
```

```
    N = len(x)
```

```
    psi = psi0.copy()
```

```
    # Matriz de Hamiltoniano discretizado
```

```
    H = np.zeros((N, N), dtype=complex)
```

```
    for i in range(1, N-1):
```

```
        H[i, i] = 1.0 / dx**2 + V[i]
```

```
        H[i, i-1] = H[i, i+1] = -0.5 / dx**2
```

```
H[0, 0] = H[-1, -1] = 1e10 # Condiciones de frontera (potencial infinito)
```

```
# Matrices A y B para el método Crank-Nicolson
```

```
I = np.eye(N, dtype=complex)
```

This conversation is archived. To continue, please unarchive it first.





ChatGPT



Explore GPTs

```

A = -1j * dx / (2 * hbar) * H
B = I + 1j * dt / (2 * hbar) * H

```

```

# Evolución temporal

```

```

for t in range(Nt):

```

```

    psi = np.linalg.solve(A, np.dot(B, psi))

```

```

    psi[0] = psi[-1] = 0 # Condiciones de frontera

```

```

return psi

```

```

...

```

Finalmente, ejecutamos la simulación y visualizamos los resultados:

```

```python

```

```

Ejecutar la simulación

```

```

psi_final = evolve_schrodinger(psi0, V, x, dx, dt, Nt)

```

```

Visualizar la densidad de probabilidad inicial y final

```

```

plt.plot(x, np.abs(psi0)**2, label='t=0')

```

```

plt.plot(x, np.abs(psi_final)**2, label=f't={T}')

```

```

plt.xlabel('x')

```

```

plt.ylabel('Densidad de probabilidad')

```

```

plt.legend()

```

```

plt.show()

```



This conversation is archived. To continue, please unarchive it first.



Upgrade plan

Get GPT-4, DALL·E, and more





ChatGPT



Explore GPTs

...

Este código implementa la evolución temporal de la ecuación de Schrödinger en una dimensión para una partícula en una caja de potencial infinita. Utiliza el método de Crank-Nicolson, que es un método implícito y estable para la integración en el tiempo de ecuaciones diferenciales parciales.

Si deseas resolver el problema para un potencial diferente o con diferentes condiciones iniciales, puedes ajustar las definiciones de `V` y `psi0` según tus necesidades.



me podés agregar una animación?



Claro, podemos agregar una animación para visualizar cómo evoluciona la densidad de probabilidad en el tiempo. Utilizaremos `matplotlib` y su módulo `FuncAnimation` para crear la animación. Aquí tienes el código completo con la animación incluida:

```
```python
import numpy as np
import matplotlib.pyplot as plt
from matplotlib.animation import FuncAnimation
```



Parámetros del problema

This conversation is archived. To continue, please unarchive it first.

Unarchive



Upgrade plan

Get GPT-4, DALL·E, and more





ChatGPT



Explore GPTs

```
# Guardar la animación como un archivo .mp4 (opcional)
# ani.save('schrodinger_1d.mp4', fps=30, extra_args=['-vcodec', 'libx264'])
```

```
plt.show()
...

```

Este código crea una animación de la densidad de probabilidad $(|\psi(x,t)|^2)$ a medida que evoluciona en el tiempo. La función `evolve_schrodinger` ahora devuelve una lista de los estados de la función de onda en varios momentos del tiempo, y estos estados se utilizan para crear la animación con `FuncAnimation`.

Puedes ajustar los parámetros como el intervalo de tiempo `interval` en `FuncAnimation` y la frecuencia con la que se almacenan los estados en `psi_list` para mejorar la animación según tus necesidades. También puedes descomentar la línea para guardar la animación como un archivo `.mp4` si lo deseas.



me podés repetir el código, pero con el método de split operator



Claro, podemos resolver la ecuación de Schrödinger dependiente del tiempo en una dimensión utilizando el método del operador dividido (`split operator method`). Este método aprovecha la forma particular de la ecuación de Schrödinger para separar la evolución temporal en dos pasos:

This conversation is archived. To continue, please unarchive it first.

Unarchive



Upgrade plan

Get GPT-4, DALL·E, and more





.mp4 si lo deseas.

me podés repetir el código, pero con el método de split operator



Claro, podemos resolver la ecuación de Schrödinger dependiente del tiempo en una dimensión utilizando el método del operador dividido (split operator method). Este método aprovecha la forma particular de la ecuación de Schrödinger para separar la evolución temporal en dos pasos: uno que involucra solo el operador cinético y otro que involucra solo el operador potencial. Aquí tienes el código completo, incluyendo la animación:

python

Copy code

```
import numpy as np
import matplotlib.pyplot as plt
from matplotlib.animation import FuncAnimation

# Parámetros del problema
L = 10.0 # Longitud de la caja
N = 1000 # Número de puntos espaciales
dx = L / (N - 1) # Paso espacial
x = np.linspace(0, L, N) # Puntos espaciales
```

This conversation is archived. To continue, please unarchive it first.

Unarchive



Upgrade plan

Get GPT-4, DALL·E, and more





```
plt.show()
```

Este código utiliza el método del operador dividido para evolucionar la función de onda en el tiempo. Aquí está el desglose de los pasos clave:

1. **Inicialización de parámetros:** Definimos los parámetros espaciales y temporales, así como la condición inicial de la función de onda y el potencial.
2. **Operadores cinético y potencial:** Calculamos los operadores cinético y potencial en el espacio de Fourier y en el espacio real, respectivamente.
3. **Evolución temporal:** En cada paso de tiempo, aplicamos el operador potencial a medio paso, luego transformamos la función de onda al espacio de Fourier para aplicar el operador cinético y finalmente volvemos al espacio real para aplicar otro medio paso del operador potencial.
4. **Animación:** Creamos una animación para visualizar cómo evoluciona la densidad de probabilidad en el tiempo.

Puedes ajustar los parámetros del sistema y las condiciones iniciales según sea necesario. La línea para guardar la animación como un archivo  está comentada, pero puedes descomentarla si deseas guardar la animación.

This conversation is archived. To continue, please unarchive it first.

 Unarchive





para guardar la animación como un archivo `.mp4` está comentada, pero puedes descomentarla si deseas guardar la animación.

cuando haces la evolución temporal con el método Crank-Nicolson, estás resolviendo $(A, np.dot(B, \psi))$ tiempo a tiempo. Eso no es necesario cuando el potencial V es independiente del tiempo. Podrías invertir la matriz inicialmente, y luego utilizar este operador para la evolución temporal



Tienes razón. Cuando el potencial V es independiente del tiempo, podemos precomputar la matriz inversa para hacer el método Crank-Nicolson más eficiente. A continuación, ajustaremos el código para hacer esto y añadiremos la animación.

```
``python
import numpy as np
import matplotlib.pyplot as plt
from matplotlib.animation import FuncAnimation

# Parámetros del problema
L = 10.0 # Longitud de la caja
N = 1000 # Número de puntos espaciales
dx = L / (N - 1) # Paso espacial
x = np.linspace(0, L, N) # Puntos espaciales
```



This conversation is archived. To continue, please unarchive it first.

Unarchive





```
H[i, i-1] = H[i, i+1] = -0.5 / dx**2
```

```
H[0, 0] = H[-1, -1] = 1e10 # Condiciones de frontera (potencial infinito)
```

```
# Matrices A y B para el método Crank-Nicolson
```

```
I = np.eye(N, dtype=complex)
```

```
A = I - 1j * dt / (2 * hbar) * H
```

```
B = I + 1j * dt / (2 * hbar) * H
```

```
# Precompute the inverse of A
```

```
A_inv = np.linalg.inv(A)
```

```
return A_inv, B
```

```
def evolve_schrodinger(psi0, A_inv, B, Nt):
```

```
    psi = psi0.copy()
```

```
    psi_list = [psi.copy()]
```

```
# Evolución temporal
```

```
for t in range(Nt):
```

```
    psi = np.dot(A_inv, np.dot(B, psi))
```

```
    psi[0] = psi[-1] = 0 # Condiciones de frontera
```

```
if t % 10 == 0: # Almacenar cada 10 pasos para reducir el tamaño de la lista
```

```
    psi_list.append(psi.copy())
```

This conversation is archived. To continue, please unarchive it first.

 Unarchive



Upgrade plan

Get GPT-4, DALL·E, and more





ChatGPT

Explore GPTs

```
if t % 10 == 0: # Actualizar cada 10 pasos para reducir el tamaño de la lista
    psi_list.append(psi.copy())

return psi_list

# Precompute matrices
A_inv, B = precompute_matrices(dx, dt, hbar, V)

# Ejecutar la simulación
psi_list = evolve_schrodinger(psi0, A_inv, B, Nt)

# Configurar la animación
fig, ax = plt.subplots()
line, = ax.plot(x, np.abs(psi0)**2)
ax.set_xlim(0, L)
ax.set_ylim(0, 1)
ax.set_xlabel('x')
ax.set_ylabel('Densidad de probabilidad')

def animate(i):
    psi = psi_list[i]
    line.set_ydata(np.abs(psi)**2)
    return line,

ani = FuncAnimation(fig, animate, frames=len(psi_list), interval=50, blit=True)
```

This conversation is archived. To continue, please unarchive it first.

[Unarchive](#)

Upgrade plan

Get GPT-4, DALL-E, and more





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

En este código, se han realizado las siguientes modificaciones:

1. **Precomputación de matrices**: La función `precompute_matrices` calcula y devuelve la matriz inversa (A^{-1}) y la matriz (B) antes de la evolución temporal.
2. **Evolución temporal**: La función `evolve_schrodinger` utiliza la matriz precomputada (A^{-1}) para la evolución temporal, lo que hace el cálculo más eficiente.
3. **Animación**: La animación se genera de la misma manera que antes, utilizando `FuncAnimation` para visualizar la densidad de probabilidad en el tiempo.

Este enfoque mejora la eficiencia al evitar la inversión de la matriz en cada paso de tiempo.



no conviene multiplicar A_{inv} con B y utilizar ese operador en la evolución?



Sí, es una excelente idea. En lugar de multiplicar (A^{-1}) por (B) en cada paso temporal, podemos precomputar esta multiplicación y usar el operador resultante en la evolución temporal. Esto reducirá significativamente el costo computacional durante la evolución temporal.

A continuación, se muestra el código actualizado ↓ que realiza esta optimización:

This conversation is archived. To continue, please unarchive it first.

Unarchive





```
H[0, 0] = H[-1, -1] = 1e10 # Condiciones de frontera (potencial infinito)
```

```
# Matrices A y B para el método Crank-Nicolson
```

```
I = np.eye(N, dtype=complex)
```

```
A = I - 1j * dt / (2 * hbar) * H
```

```
B = I + 1j * dt / (2 * hbar) * H
```

```
# Precompute the combined operator
```

```
A_inv = np.linalg.inv(A)
```

```
operator = np.dot(A_inv, B)
```

```
return operator
```

```
def evolve_schrodinger(psi0, operator, Nt):
```

```
    psi = psi0.copy()
```

```
    psi_list = [psi.copy()]
```

```
    # Evolución temporal
```

```
    for t in range(Nt):
```

```
        psi = np.dot(operator, psi)
```

```
        psi[0] = psi[-1] = 0 # Condiciones de frontera
```

```
    if t % 10 == 0: # Almacenar cada 10 pasos para reducir el tamaño de la lista
```

```
        psi_list.append(psi.copy())
```

```
    return psi_list
```



This conversation is archived. To continue, please unarchive it first.

 Unarchive





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

Explicación de los cambios:

1. **Precomputación del operador combinado**:

- La función `precompute_operator` calcula las matrices $\{(A)\}$ y $\{(B)\}$ para el método Crank-Nicolson y luego precomputa el operador combinado $\{(A^{-1}B)\}$.
- Este operador combinado se devuelve y se utiliza para la evolución temporal.

2. **Evolución temporal optimizada**:

- En la función `evolve_schrodinger`, utilizamos directamente el operador precomputado para actualizar el estado de la función de onda en cada paso temporal.
- Esto elimina la necesidad de realizar una multiplicación de matrices en cada paso temporal, lo que mejora la eficiencia.

3. **Animación**:

- La animación se configura y se ejecuta de la misma manera que antes, visualizando la densidad de probabilidad en función del tiempo.

Este enfoque optimizado reduce significativamente el costo computacional de la evolución temporal al reutilizar el operador precomputado en lugar de calcularlo en cada paso.

¡Eres una genia!



¡Gracias! Me alegra saber que encontraste útil la solución. Si tienes alguna otra pregunta o necesitas más ayuda con algo, no dudes en decírmelo. ¡Estoy aquí para ayudarte!



me codes hacer la implementación con Pune-Kutta

This conversation is archived. To continue, please unarchive it first.

Unarchive

