Projet Dynamique Quantique

PUJOL Martin

4 décembre 2024

Table des matières

1 Introduction

The ability to describe the time evolution of a quantum system, that is, the evolution over time of its wave function $\Psi(\vec{r}, t)$, is one of the major results of quantum physics. Quantum mechanics is a fundamental theory of physics that has undergone considerable development in the 20th century, with numerous applications in physics, chemistry, biology, and computer science. Schrödinger's equations describe the evolution of quantum systems, and their numerical resolution is a significant challenge in theoretical physics and numerical simulation.

The numerical resolution of the time-dependent Schrödinger equations is challenging and requires adapted numerical methods, whether for one-dimensional or multidimensional systems. These methods rely on intensive computation algorithms, which demand substantial computational resources. Computational power, therefore, is a critical resource central to this study.

In this project, we implement numerical methods to solve the time-dependent Schrödinger equation for a particle of mass m subject to the action of a one-dimensional potential $V(x)$. This equation describes the time evolution of the particle's wave function $\Psi(x, t)$, given by Equation (3). The one-dimensional potential can be used to model various systems, such as particles in a potential well, electrons in a crystal, atoms in a gas, or molecules in an environment.

First, we study the stationary states of the system, corresponding to time-independent solutions of the Schrödinger equation. These stationary states are crucial in quantum physics as they determine the system's fundamental properties, such as energy levels and associated wave functions. To this end, we use the finite difference method to discretize space, replacing partial derivatives with finite differences. We also employ numerical integration methods to compute eigenvalues and eigenvectors of the matrix representing the system's Hamiltonian.

Next, we study the system's time evolution using numerical integration methods, such as Euler's method and the fourth-order Runge-Kutta method (RK4). These methods calculate the wave function $\Psi(x, t)$ at different times t, using the initial wave function at $t = 0$ as a starting condition. We also analyze the convergence and stability of these integration methods using numerical analysis techniques.

Finally, we compare numerical results with analytical results for simple systems, allowing us to conclude the importance of spatial and temporal discretization.

2 Materials and Methods

2.1 Resolution Methods : Part 1

First, we aim to determine the eigenstates of the Hamiltonian. Let us recall its expression :

$$-\frac{\hbar^2}{2m}\frac{d^2\psi}{dx^2} + V(x)\psi = E\psi$$

where $\hbar$ is the reduced Planck constant, m is the particle mass, $V(x)$ is the potential, and E is the eigenvalue associated with the stationary state ψ.

We seek to solve this equation to find the eigenvalues of the Hamiltonian. To do so, we discretize the spatial domain using a finite difference method. By discretizing the space, the second derivative is replaced with a centered finite difference, leading to the following equation :

$$-\frac{\hbar^2}{2m}\frac{\psi_{i+1} - 2\psi_i + \psi_{i-1}}{\Delta x^2} + V_i\psi_i = E\psi_i$$

where i is the discrete index representing the spatial position along the x direction, and Δx is the distance between discrete points. This equation can be rewritten in matrix form as :

$$\mathbf{H}\psi = E\psi$$

where $\mathbf{H}$ is the Hamiltonian matrix, ψ is the discretized wavefunction vector, and E is the discretized eigenvalue.

Using the finite difference method, the matrix $\mathbf{H}$ becomes tridiagonal with specific diagonal and off-diagonal elements. The main diagonal is given by :

$$\mathbf{H}_{ii} = \frac{\hbar^2}{m\Delta x^2} + V_i$$

and the off-diagonal terms are :

$$\mathbf{H}_{i,i+1} = \mathbf{H}_{i+1,i} = -\frac{\hbar^2}{2m\Delta x^2}$$

Next, to find the eigenvalues of the Hamiltonian matrix, we apply a tridiagonal diagonalization method.

This method allows us to diagonalize the Hamiltonian matrix to obtain discrete eigenvalues and their corresponding eigenvectors. These eigenvalues and eigenvectors represent the discrete energy levels and wavefunctions, respectively.

In summary, the finite difference method enables us to discretize the stationary wave equation and find the eigenfunctions of the Hamiltonian in a discrete framework.

2.2 Resolution Methods : Part 2

In the second part, we studied the time evolution of the wavefunction of a particle in a one-dimensional potential by solving the time-dependent Schrödinger equation :

$$i\hbar\frac{\partial}{\partial t}\Psi(x,t) = \hat{H}\Psi(x,t)$$

where $\hbar$ is the reduced Planck constant, and $\hat{H}$ is the Hamiltonian operator of the system. To derive a differential relationship for the wavefunction $\Psi(x,t)$, we used the Hamiltonian operator for a particle in a one-dimensional potential :

$$\hat{H} = -\frac{\hbar^2}{2m}\frac{\partial^2}{\partial x^2} + V(x)$$

where m is the particle mass, and $V(x)$ is the one-dimensional potential.

By substituting the Hamiltonian operator into the Schrödinger equation, we obtain :

$$i\hbar\frac{\partial}{\partial t}\Psi(x,t) = -\frac{\hbar^2}{2m}\frac{\partial^2}{\partial x^2}\Psi(x,t) + V(x)\Psi(x,t)$$

We then discretized the spatial domain, replacing the continuous wavefunction with discrete values Ψ_j^n, where j represents the spatial position, and n represents the time step. Partial derivatives are replaced with finite differences to derive a recurrence relation for the discrete wavefunction values :

$$i\hbar\frac{\Psi_j^{n+1} - \Psi_j^n}{\Delta t} = -\frac{\hbar^2}{2m}\frac{\Psi_{j+1}^n - 2\Psi_j^n + \Psi_{j-1}^n}{\Delta x^2} + V_j\Psi_j^n$$

where Δx and Δt are the spatial and temporal discretization steps, respectively, and V_j is the potential value at position $j\Delta x$. This recurrence relation is known as the explicit Euler method for solving the time-dependent Schrödinger equation.

Additionally, we used more precise numerical integration methods, such as the 4th-order Runge-Kutta method (RK4). In this case, a more complex recurrence relation

2.3 Python Tools

In this project, various Python libraries were utilized to simulate the dynamics of a Gaussian wave packet in different potentials. First, the NumPy library was employed to manipulate multidimensional arrays, enabling efficient vectorized calculations and optimizing performance.

Next, the Matplotlib library was used for plotting graphs and creating animations. Matplotlib.pyplot was leveraged to plot graphs of the temporal evolution of the wavefunction and the associated particle's mean position. Additionally, Matplotlib.animation was used to create corresponding animations.

The Numba library was also utilized to optimize the performance of calculations.

In summary, these Python libraries allowed for effective and precise simulation of wavefunction dynamics in various potentials, combining code optimization, graphical visualization, and numerical resolution of differential equations.

2.4 Python Codes (See Appendix)

The main program for the first part is "resol.py." This program solves the wave equation by discretizing space into n points with a total length L. The 'resol' function takes as input the total length L, the number of points n, the potential, and a boolean parameter 'Norm' indicating whether the wavefunction should be normalized. The function calculates the eigenvalues 'w' and eigenvectors 'v' of the tridiagonal matrix associated with the discretized wave equation and returns the results as NumPy arrays. Additionally, the 'normalize' function normalizes the eigenvectors if the 'Norm' parameter is set to 'True'.

The potential functions are stored in the file "potentiels.py."

The script "potentiel stationnaire quantitatif.py" is a Python program designed to evaluate the numerical resolution of the Schrödinger equation for various common potentials and recover the typical results, such as energy quantization and often sinusoidal wavefunctions.

It uses computation and visualization modules, such as Matplotlib, and specific functions to solve the Schrödinger equation for various potentials, including the infinite potential well, potential barrier, double-well potential, periodic potential, and others. The program uses simulation parameters to determine the wavefunctions and energy eigenvalues of the systems. For each potential, it plots the first ten wavefunctions, representing the probability density and the energy eigenvalues of each wavefunction. The results are saved as image files for each potential.

For the second part, implementing two numerical methods to solve the wavefunction's time evolution was necessary.

The program "euler.py" contains several functions. The primary function, 'euler', implements the Euler method to solve a differential equation. This function takes as input a function 'f', the bounds 'a' and 'b' of the interval on which the equation is solved, the number N of discretization steps, the initial condition 'y0', and an optional parameter vector. The function returns an array of N values taken by y with a step size h. The Euler method is a simple numerical method for solving differential equations, approximating the function's value at each step h using its current value and derivative.

The second function in "euler.py" is 'normalize', which normalizes a matrix column-wise by dividing each column by its norm and then divides the matrix by the square root of the discretization step size. This ensures that the sum of the probabilities is equal to 1.

The second implemented numerical method is the 4th-order Runge-Kutta method. The program "rk4.py" begins by defining a function 'rk4', which takes as input the time step, the function values at time $t(i)$, a derivative function, and parameters. It returns the new function values for $t(i + 1)$ using the 4th-order Runge-Kutta method.

Next, the program defines a function 'RK4', which takes as input a derivative function, lower and upper bounds a and b, the number of points N, the function's initial value 'y0', and parameters. It returns an array of N function values using the 4th-order Runge-Kutta method.

With these two methods defined, introducing the function that computes the time evolution of a wavefunction becomes essential. The ‘deriv‘ function calculates the derivative of the wavefunction ‘y‘ using the finite difference method.

The programs "evolution RK4.py" and "evolution euler.py" evaluate the respective quality of these methods. These programs record some values and plot the evolution of the error on the stationary state for several Δt values. Errors are calculated for $\Delta t = 1e{-}2, 1e{-}3$, and $1e{-}4$, and are stored in ‘error1‘, ‘error2‘, and ‘error3‘, respectively. The corresponding times are stored in the arrays ‘temps1‘, ‘temps2‘, and ‘temps3‘.

Finally, the program plots the error evolution for the three Δt values. These graphs are saved as "RK4$_{P}$*recision.png.*”

Finally, several programs were created to highlight the dynamics of a Gaussian wave packet in different potentials. Each of these programs includes an animation and ten graphs tracing the wavefunction's temporal evolution. The graphs also trace the mean position of the particle associated with the wavefunction. This simulation is useful for visualizing and understanding the behavior of wave packets in various potential environments.

Here's the continuation and completion of the translation and enhancement of the text :

“‘latex

3 Results

3.1 Part One

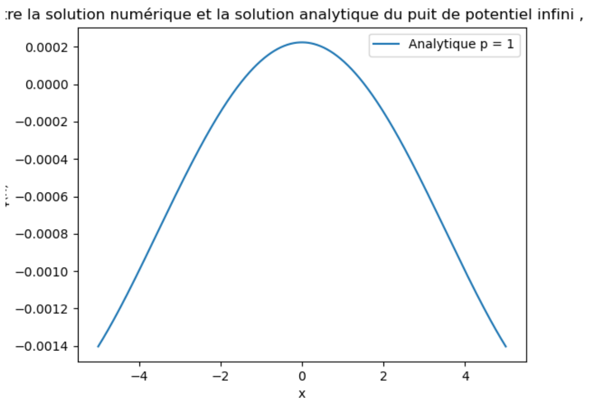

FIGURE 1 – Error for $\Delta t = 1e{-}2$

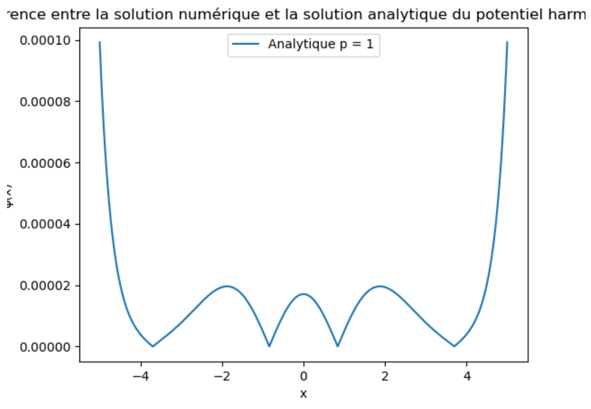

FIGURE 2 – Error for $\Delta t = 1e{-}3$

Initially, our objective was to determine the optimal value of Δt. While smaller values of Δt are expected to reduce the error, it was equally important to limit the computation time to keep simulations feasible. The graphs above illustrate the error for Δt values of $1e{-}2$ and $1e{-}3$.

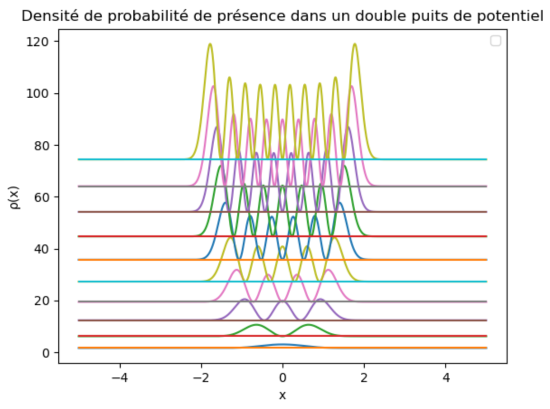

FIGURE 3 – Double-well potential

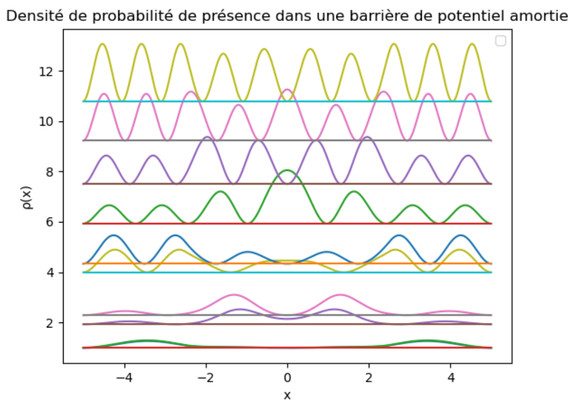

FIGURE 4 – Damped potential barrier

FIGURE 5 – Wavefunctions for various potentials

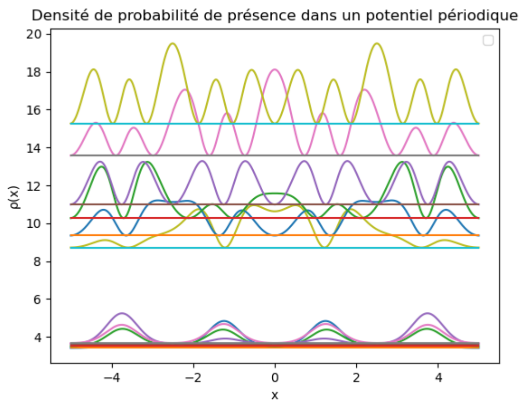

FIGURE 6 – Periodic potential

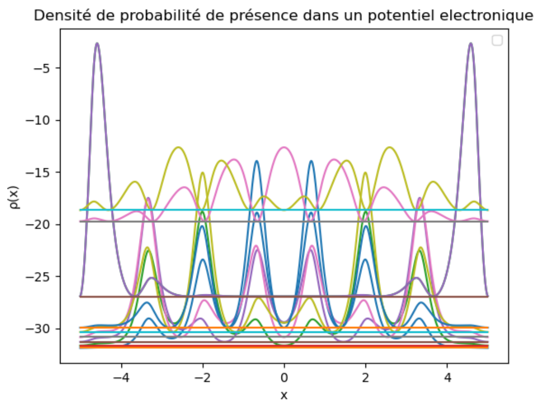

FIGURE 7 – Electronic potential

Figures 3 to 7 illustrate the computed wavefunctions for different potentials, such as double wells, damped potential barriers, periodic potentials, and electronic potentials. The probability densities show highly satisfactory shapes, reminiscent of patterns commonly observed in physics. The results also exhibit the conservation of system symmetry properties and energy level quantization. Each graph depicts ten probability densities, offset by their respective eigenenergies.

3.2 Part Two

First, it was necessary to determine the optimal values for Δt and Δx for simulations. To minimize spatial discretization, a minimum value of $\Delta x = 0.1$ was chosen for a domain of length 10. This limit was essential due to the iterative nature of the calculations. Considering the differential equation, it was expected that the ratio $\Delta t/(\Delta x)^2$ should not be too large. To confirm this hypothesis, a graph of the error over time was plotted for three different Δt values.

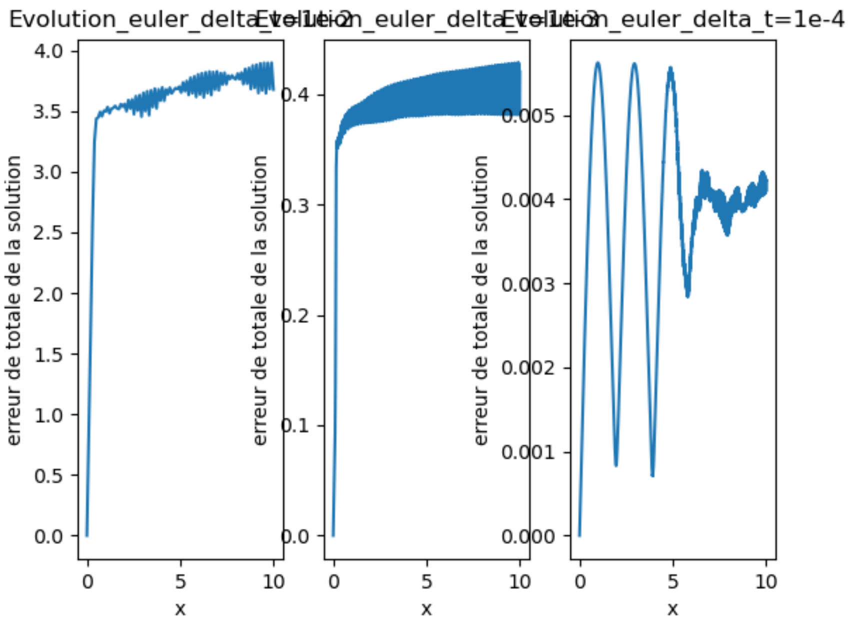

FIGURE 8 – Error evolution for the Euler method

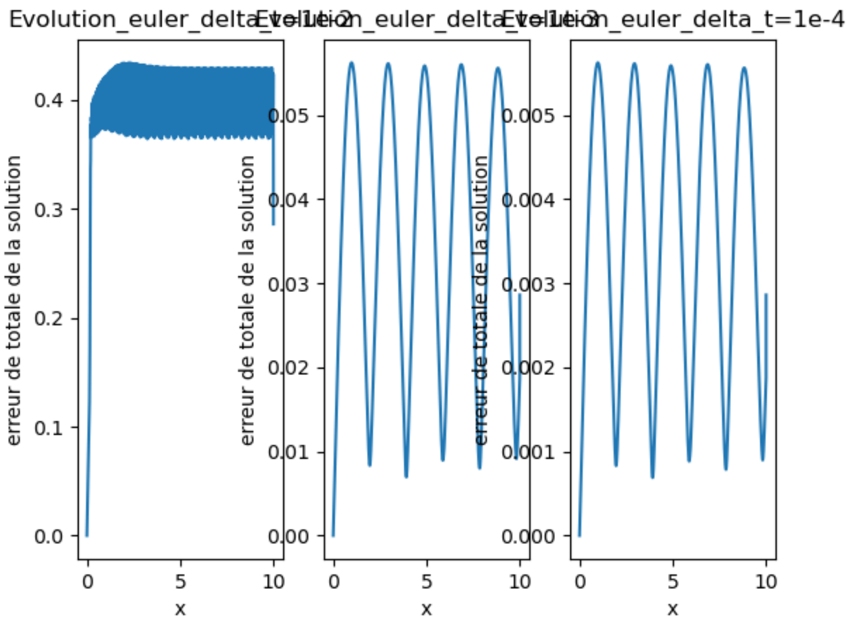

FIGURE 9 – Error evolution for the Runge-Kutta 4 method

The graphs show the error evolution for the Euler method and the Runge-Kutta 4 (RK4) method over time for different Δt values. It can be observed that, for a large Δt, the error increases rapidly, whereas for a sufficiently small Δt, the error remains low and stable. Overall, the RK4 method is more accurate than the Euler method for all tested Δt values. Therefore, simulations were continued with the RK4 method and $\Delta t = 1\mathrm{e}{-3}$.

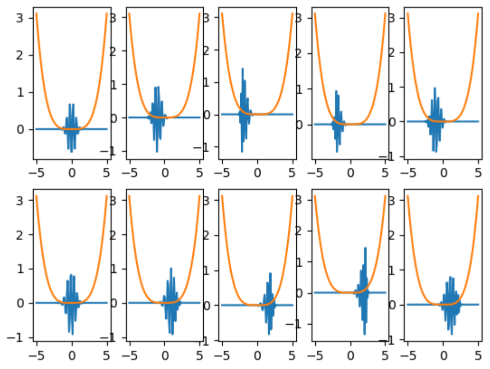
FIGURE 10 – Wave packet dynamics in a harmonic potential

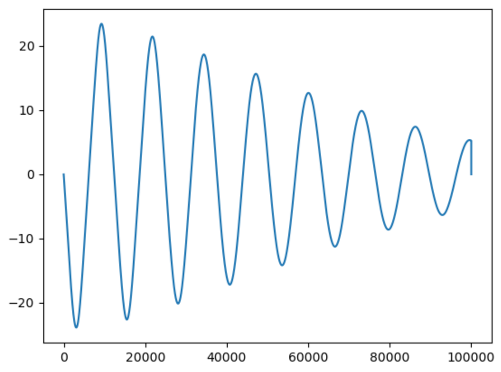
FIGURE 11 – Mean position of the particle

Figure 10 shows the oscillation of a wave packet in a harmonic potential. To achieve a significant displacement, the wave packet was assigned very high energy ($k_0 = 1\mathrm{e}10$). However, despite this high energy, the wave packet's position spreads over time, possibly due to numerical errors. Moreover, the particle's mean position exhibits a periodic oscillation, seemingly converging toward zero. This observation reflects the quasi-classical behavior of the particle in a harmonic potential.

Figures 12 and 13 extend the previous simulations, showing a wave packet traversing a potential field composed of six uniformly spaced electrons. It can be observed that the wave packet appears trapped at the center, where the potential is highest. Figure 13 depicts the same potential as Figure 12 but with higher energy assigned to each electron.

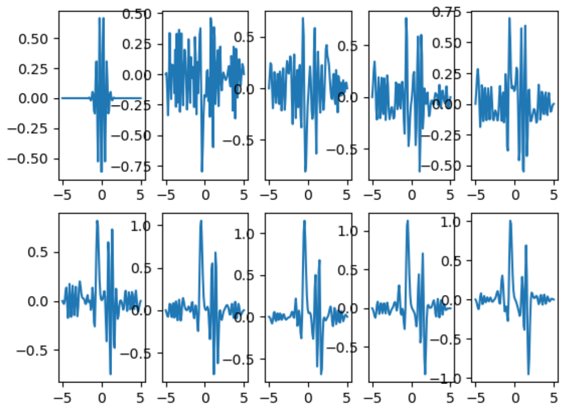

FIGURE 12 – Wave packet dynamics in an electronic potential

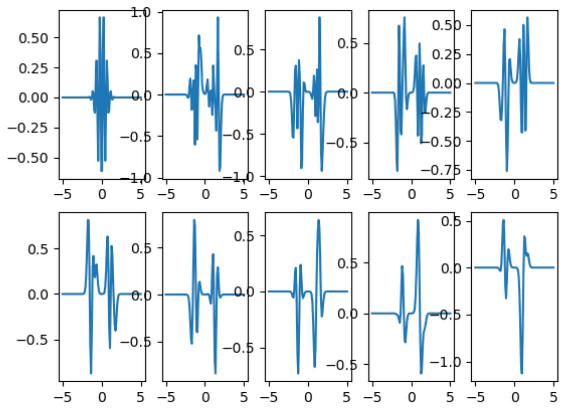

FIGURE 13 – Wave packet dynamics in a higher electronic potential

4 Discussion of Results

The discussion can be structured into two main parts : numerical methods and observed results.

Numerical Methods

The first part of the study focused on solving the eigenstates of the Hamiltonian for various potentials. The initial objective was to determine the optimal value of Δt to minimize the error while limiting computation time. Graphs depicting errors for $\Delta t = 1\text{e}{-2}$ and $\Delta t = 1\text{e}{-3}$ showed a significant reduction in error for smaller Δt values but at the cost of increased computation time. Thus, $\Delta t = 1\text{e}{-3}$ was chosen as a reasonable compromise.

The following graphs presented the computed eigenfunctions for various potentials. The resulting probability densities were highly satisfactory, showing patterns consistent with the expected physical properties for each potential, while preserving system symmetry and energy quantization.

In the second part, we examined the temporal evolution of wavefunctions. Optimal Δt and Δx values were determined, ensuring that $\Delta t/(\Delta x)^2$ remained small. The error evolution graphs for different Δt values validated this approach.

Observed Results

Regarding the results, it was reassuring to find quantized energy levels and correct analytical solutions for the initial energy levels in the infinite square well and harmonic potential. The shapes of the probability densities were also consistent with expectations. A key confirmation of these calculations was the invariance of wavefunctions over time, in contrast to their linear combinations or wave packets.

Finally, the dynamics of wavefunctions revealed further insights. For example, the periodic oscillation of a Gaussian wave packet in a harmonic potential suggested quasi-classical behavior. Additionally, the trapping of a wave packet by multiple electrons highlighted the quantum mechanical phenomenon of localization.

5 Conclusion

This project successfully implemented numerical methods to solve the time-dependent Schrödinger equation based on the system Hamiltonian. The results demonstrated the importance of carefully selecting Δt and Δx to balance accuracy and computational feasibility. The study highlighted the precision of the RK4 method over the Euler method, albeit at a higher computational cost.

The investigation revealed quantized energy levels, consistent analytical predictions, and preserved system symmetry. Additionally, the dynamics of Gaussian wave packets in harmonic and electronic potentials provided valuable insights into quasi-classical behavior and quantum localization.

In summary, the project

advanced the understanding and application of numerical methods for solving the time-dependent Schrödinger equation, showcasing the quantum properties and dynamics of different systems.

6 Appendix

6.1 Python Codes

Below are the primary Python scripts used in the project. See the code listing in the respective files :

```python
import numpy as np
from scipy.linalg import eigh_tridiagonal
import matplotlib.pyplot as plt

#On sait que la solution de l'equation d'onde doit respecter des conditions
    de normalisation
#Elle doit etre de carré sommable égale à 1
#Ayant discrétisé l'espace, on traduit la condition de normalisation avec la
    méthode suivante  :
def normalize(m,delta_x):
    m /= np.linalg.norm(m, axis=0) # Normaliser les colonnes
    m /= np.sqrt(delta_x) # diviser par 1/sqrt(dx)
    return m



def resol(L,n, potentiel, Norm = False) :
    delta  = L/n
    x = np.linspace(-L/2, L/2, n)
    d = potentiel(x) + 2/(delta**2) # diagonale principale de la matrice
    e = np.ones(n-1)*(-1/(delta**2)) # diagonale juste au-dessus
    w, v = eigh_tridiagonal(d,e)
    if Norm :
        v = normalize(v, delta)
    return x, w, v
```

```python
import numpy as np
import math


def barrière_de_potentiel(x, d = 5, U0 = 10000000):
    res = np.zeros(np.size(x))
    for i in range(np.size(x)):
        if x[i] >= -d/2 and x[i] <= d/2:
            res[i] = U0
    return res

def barrière_de_potentiel_amortie(x, d = 5, U0 = 4):
    res = np.zeros(np.size(x))
    for i in range(np.size(x)):
        if x[i] >= -d/2 and x[i] <= d/2:
            res[i] = U0*((np.cos(2*np.pi*x[i]/d))**2)
    return res


def double_puits_de_potentiel(x, a = 5):
    res = np.zeros(np.size(x))
    for i in range(np.size(x)):
        res[i] = a*(x[i]**4) -(x[i]**2)/2
    return res

def potentiel_periodique(x,U0 = 10, d = 5):
    return U0*(np.cos(2*np.pi*x/d))**2

```

```python
def potentiel_electronique(x, V0 = 40):
    pos_atomes = np.linspace(-6,6,10)
    p = 0.5
    V = 0
    for i in pos_atomes : V += -V0*np.exp(-np.abs(x-i)/p)
    return V

def puit_de_potentiel(x):
    return np.zeros(np.size(x))

def potentiel_harmonique(x, k = 1):
    return 0.5*k*(x**2)

def gauss(x, x0, sigma, k0, A =1):
    return (1/(sigma*np.sqrt(2*np.pi)))*np.exp(-((x-x0)**2)/(2*sigma**2))*np
    .exp(1j*k0*x)

def potentiel_puit(x, d = 5, U0 = 10000000):
    res = np.ones(np.size(x))*U0
    for i in range(np.size(x)):
        if x[i] >= -d/2 and x[i] <= d/2:
            res[i] = 0
    return res
```

```python
import numpy as np

def deriv(y,params ): # params = [delta_x, V]
    dy = np.zeros(np.size(y), dtype= complex)
    for k in range(1, np.size(y)-1): #Comme la particule est confiné on peut
     imposé comme condition limite que la sa probabilité de présence est
    nulle au bord
            dy[k]= complex(0,(1/(2*(params[0]**2)))*(y[k+1] + y[k-1] - 2*y[k
    ]) + params[1][k]*y[k])
            #dy[k]= complex(0,(1/(params[0]**2))*(y[k+1] + y[k-1] - 2*y[k])
    + 2*params[1][k]*y[k])
    return dy
```

```python
import numpy as np
from scipy.linalg import eigh_tridiagonal

#Méthode d'Euler
def euler( f, a, b, N, y0, params = None, Norm = True):
    h = (b-a)/N
    y = np.zeros((N, np.size(y0)), dtype=complex)
    y[0] = y0
    i = 0
    while(i< N-1):
        y[i+1] = y[i] + h*(f( y[i], params))
        i = i +1
        if Norm : y[i] /= np.sqrt(sum(np.abs(y[i])**2)*params[0])
    return y #retourne un tableau des N valeurs prise par y avec un pas h



#On sait que la solution de l'equation d'onde doit respecter des conditions
    de normalisation
#Elle doit etre de carré sommable égale à 1
#Ayant discrétisé l'espace, on traduit la condition de normalisation avec la
     méthode suivante  :


```

```python
def normalize(m,delta_x):
    m /= np.linalg.norm(m, axis=0) # normaliser les colonnes
    m /= np.sqrt(delta_x) # diviser par 1/sqrt(dx)
    return m



def resol(L,n, potentiel, Norm = False) :
    delta  = L/n
    x = np.linspace(-L/2, L/2, n)
    d = potentiel(x) + 2/(delta**2) # diagonale principale de la matrice
    e = np.ones(n-1)*(-1/(delta**2)) # diagonale juste au-dessus
    w, v = eigh_tridiagonal(d,e)
    if Norm :
        v = normalize(v, delta)
    return x, w, v
```

```python
# -*- coding: utf-8 -*-

import numpy as np
from numba import njit


def rk4(dt, y, deriv,params):
    """
      /*-----------------------------------------
      sous programme de resolution d'equations
      differentielles du premier ordre par
      la methode de Runge-Kutta d'ordre 4
      x = abscisse, une valeur scalaire, par exemple le temps
      dx = pas, par exemple le pas de temps
      y = valeurs des fonctions au temps t(i), c'est un tableau numpy de taille n
      avec n le nombre d'équations différentielles du 1er ordre

      rk4 renvoie les nouvelles valeurs de y pour t(i+1)

      deriv = variable contenant le nom du
      sous-programme qui calcule les derivees
      deriv doit avoir trois arguments: deriv(x,y,params) et renvoyer
      un tableau numpy dy de taille n
      ----------------------------------------*/
    """
    #  /* d1, d2, d3, d4 = estimations des derivees
    #    yp = estimations intermediaires des fonctions */
    #        #         /* demi-pas */
    d1 = deriv(y,params)   #       /* 1ere estimation */
    yp = y + d1*dt
    #    for  i in range(n):
    #        yp[i] = y[i] + d1[i]*ddx
    d2 = deriv(yp,params)     #/* 2eme estimat. (1/2 pas) */
    yp = y + d2*dt
    d3 = deriv(yp,params)  #/* 3eme estimat. (1/2 pas) */
    yp = y + d3*dt
    d4 = deriv(yp,params)     #  /* 4eme estimat. (1 pas) */
    #/* estimation de y pour le pas suivant en utilisant
    #  une moyenne ponderee des derivees en remarquant
    #  que : 1/6 + 1/3 + 1/3 + 1/6 = 1 */
    res = y + dt*( d1 + 2*d2 + 2*d3 + d4 )/6
    return res

```

```python

#Methode qui retourne un tableau de N valeurs de X(t) avec X(t) la solution de l'équation différentielle en utlisant la méthode de Runge-Kutta d'ordre 4
def RK4(f, a, b, N, y0, params):
    dt = (b-a)/N
    y = np.zeros((N, np.size(y0)), dtype = complex)
    y[0] = y0
    i = 0
    while(i< N-2):
        y[i+1] = rk4(dt, y[i], f, params)
        i = i +1
        y[i] /= np.sqrt(np.sum(abs(y[i])**2)*params[0])
    return y
```

```python
'''Dans ce programme, on résout l'équation de Schrödinger par la méthode de Runge Kutta d'ordre 4
La dérivé temporelle de la fonction d'onde est contenu dans l'Hamiltonien
En discrétisant l'espace et le temps, on obtient une équation différentielle
La dérivé temporelle de la fonction au rend n+1 est calculé par la fonction deriv
Cette méthode etant plus performante, on calculera aussi l'evolution de la probabilité de présence d'une paquet d'onde gaussien'''

'''On importe les bibliothèques nécessaires'''''

from resol import *
from matplotlib.animation import FuncAnimation
from euler import *
from potentiels import *
from deriv import *

'''On définit les variables nécessaires'''
n_espace, n_temps, debut, fin, L= 100, 100000, 0, 10,  10
#x = np.linspace(-L/2, L/2, n_espace)
sigma,k0 = 0, 0

x,w, v= resol(L,n_espace, puit_de_potentiel, Norm = True)
y1 = v[:, 7]
y2 = (v[:,7]  +v[:,14] + v[:, 21])/np.sqrt(3)
params = [L/n_espace, puit_de_potentiel(x)]


res = euler(deriv, debut, fin, n_temps, y1,params)

'''On trace l'evolution de l'erreur sur l'etat stationnaire pour plusieurs valeurs de delta_t'''
temps1 =np.linspace(debut, fin, n_temps)
error1 = np.zeros(n_temps)
for i in range(n_temps): error1[i] = np.sum(np.abs(res[i]- res[0]))
error1 *= fin/n_temps


'''On définit les variables nécessaires'''
n_temps = 1000

x,w, v= resol(L,n_espace, puit_de_potentiel, Norm = True)
y1 = v[:, 7]
params = [L/n_espace, puit_de_potentiel(x)]
res = euler(deriv, debut, fin, n_temps, y1, params)

```

```
'''On trace l'evolution de l'erreur sur l'etat stationnaire'''
temps2 = np.linspace(debut, fin, n_temps)
error2 = np.zeros(n_temps)
for i in range(n_temps): error2[i] = np.sum(np.abs(res[i]- res[0]))
error2 *= fin/n_temps



'''On définit les variables nécessaires'''
n_temps = 100

x,w, v= resol(L,n_espace, puit_de_potentiel, Norm = True)
y1 = v[:, 7]
params = [L/n_espace, puit_de_potentiel(x)]
res = euler(deriv, debut, fin, n_temps, y1, params )

'''On trace l'evolution de l'erreur sur l'etat stationnaire'''
error3 = np.zeros(n_temps)
for i in range(n_temps): error3[i] = np.sum(np.abs(res[i]- res[0]))
error3 *= fin/n_temps
temps3 = np.linspace(debut, fin, n_temps)



plt.figure()
plt.subplot(1,3,1)
plt.plot(temps3, error3)
plt.xlabel('x')
plt.ylabel('erreur de totale de la solution')
plt.title('Evolution_euler_delta_t=1e-2')
plt.subplot(1,3,2)
plt.plot(temps2, error2)
plt.xlabel('x')
plt.ylabel('erreur de totale de la solution')
plt.title('Evolution_euler_delta_t=1e-3')
plt.subplot(1,3,3)
plt.plot(temps1, error1)
plt.xlabel('x')
plt.ylabel('erreur de totale de la solution')
plt.title('Evolution_euler_delta_t=1e-4')
plt.savefig('Euler_Precision')
plt.show()
plt.close()
```

```
'''Dans ce programme, on résout l'équation de Schrödinger par la méthode de
    Runge Kutta d'ordre 4
La dérivé temporelle de la fonction d'onde est contenu dans l'Hamiltonien
En discrétisant l'espace et le temps, on obtient une équation différentielle
La dérivé temporelle de la fonction au rend n+1 est calculé par la fonction
    deriv
Cette méthode etant plus performante, on calculera aussi l'evolution de la
    probabilité de présence d'une paquet d'onde gaussien'''

'''On importe les bibliothèques nécessaires'''''

from resol import *
from matplotlib.animation import FuncAnimation
from rk4 import *
from potentiels import *
from deriv import *

```

```python
'''On définit les variables nécessaires'''
n_espace, n_temps, debut, fin, L= 100, 100000, 0, 10,  10
#x = np.linspace(-L/2, L/2, n_espace)
sigma,k0 = 0, 0

x,w, v= resol(L,n_espace, puit_de_potentiel, Norm = True)
y1 = v[:, 7]
y2 = (v[:,7]  +v[:,14] + v[:, 21])/np.sqrt(3)
params = [L/n_espace, puit_de_potentiel(x)]
res1 = RK4(deriv, debut, fin, n_temps, y1, params = params)
res2 = RK4(deriv, debut, fin, n_temps, y2, params = params)

'''
def animate(i):
    etat1.set_data(x,abs(res1[i*100])**2)
    etat2.set_data(x,abs(res2[i*100])**2)
    potentiel.set_data(x, puit_de_potentiel(x))
    return  etat1, etat2, potentiel,

fig=plt.figure()
plt.xlim(-5, 5)
plt.ylim(0, 1)
etat1, etat2 ,potentiel = plt.plot([], [], [], [], [], [])
anim = FuncAnimation(fig,animate,blit=False)
plt.show()
'''

'''J'enregistre quelques valeurs'''
plt.figure()
for i in range(1,9):
    plt.subplot(2,4,i)
    plt.plot(x, np.abs(res2[i*1000])**2)
    plt.plot(x, puit_de_potentiel(x))
    plt.title('t = %s'%(round(i*1000*(fin/n_temps),3)))
plt.savefig('Evolution_euler_combinaison_puit_infini.png')
#plt.show()
plt.close()


'''On trace l'evolution de l'erreur sur l'etat stationnaire pour plusieurs
    valeurs de delta_t'''
temps1 =np.linspace(debut, fin, n_temps)
error1 = np.zeros(n_temps)
for i in range(n_temps): error1[i] = np.sum(np.abs(res1[i]- res1[0]))
error1 *= fin/n_temps


'''On définit les variables nécessaires'''
n_temps = 10000

x,w, v= resol(L,n_espace, puit_de_potentiel, Norm = True)
y1 = v[:, 7]
params = [L/n_espace, puit_de_potentiel(x)]
res1 = RK4(deriv, debut, fin, n_temps, y1, params = params)

'''On trace l'evolution de l'erreur sur l'etat stationnaire'''
temps2 = np.linspace(debut, fin, n_temps)
error2 = np.zeros(n_temps)
for i in range(n_temps): error2[i] = np.sum(np.abs(res1[i]- res1[0]))
error2 *= fin/n_temps
```

```python



'''On définit les variables nécessaires'''
n_temps = 1000

x,w, v= resol(L,n_espace, puit_de_potentiel, Norm = True)
y1 = v[:, 7]
params = [L/n_espace, puit_de_potentiel(x)]
res1 = RK4(deriv, debut, fin, n_temps, y1, params = params)

'''On trace l'evolution de l'erreur sur l'etat stationnaire'''
error3 = np.zeros(n_temps)
for i in range(n_temps): error3[i] = np.sum(np.abs(res1[i]- res1[0]))
error3 *= fin/n_temps
temps3 = np.linspace(debut, fin, n_temps)



plt.figure()
plt.subplot(1,3,1)
plt.plot(temps3, error3)
plt.xlabel('x')
plt.ylabel('erreur de totale de la solution')
plt.title('Evolution_euler_delta_t=1e-2')
plt.subplot(1,3,2)
plt.plot(temps2, error2)
plt.xlabel('x')
plt.ylabel('erreur de totale de la solution')
plt.title('Evolution_euler_delta_t=1e-3')
plt.subplot(1,3,3)
plt.plot(temps1, error1)
plt.xlabel('x')
plt.ylabel('erreur de totale de la solution')
plt.title('Evolution_euler_delta_t=1e-4')
plt.savefig('RK4_Precision')
plt.show()
plt.close()
```

```python
'''Dans ce programme, on résout l'équation de Schrödinger par la méthode de
    Runge Kutta d'ordre 4
La dérivé temporelle de la fonction d'onde est contenu dans l'Hamiltonien
En discrétisant l'espace et le temps, on obtient une équation différentielle
La dérivé temporelle de la fonction au rend n+1 est calculé par la fonction
    deriv
Cette méthode etant plus performante, on calculera aussi l'evolution de la
    probabilité de présence d'une paquet d'onde gaussien'''

'''On importe les bibliothèques nécessaires'''''

from resol import *
from matplotlib.animation import FuncAnimation
from rk4 import *
from potentiels import *
from deriv import *
import matplotlib.animation as animation

'''On définit les variables nécessaires'''
n_espace, n_temps, debut, fin, L= 100,10000, 0, 1,  10
x = np.linspace(-L/2, L/2, n_espace)
sigma,k0, x0 = 0.5, 1e10, 0
```

```python



#x,w, v= resol(L,n_espace, double_puits_de_potentiel, Norm = True)
y1 = gauss(x, x0, sigma, k0)
params = [L/n_espace, potentiel_harmonique(x)]
res1 = RK4(deriv, debut, fin, n_temps, y1, params = params)


'''
def animate(i):
    etat1.set_data(x,abs(res1[i*10])**2)
    potentiel.set_data(x, potentiel_harmonique(x)*1e-3)

    return  etat1, potentiel,

fig=plt.figure()
plt.xlim(-5, 5)
plt.ylim(0, 1)
etat1 ,potentiel = plt.plot([], [],  [], [])
anim = FuncAnimation(fig,animate,blit=False, interval = 1)
plt.show()
'''


'''Je trace la valeur moyenne de la position dans le temps'''

plt.figure()
for i in range(10):
    plt.subplot(2,5, i +1)
    plt.plot(x, res1[int(i*(n_temps/10))])
    plt.plot(x, potentiel_harmonique(x)*1e-3)
plt.savefig("Dynamique_Paquet_Onde_Potentiel_Harmonique.png")
plt.show()
plt.close()

pos_moyenne = []
for i in range(n_temps) :
    pos_moyenne += [np.sum((np.abs(res1[i])**2) * x)]
plt.figure()
plt.plot(range(n_temps), pos_moyenne)
#plt.savefig("Position_Moyenne_Paquet_Onde_Potentiel_Harmonique.png")
plt.show()
plt.plot()
```