Felix Krone Friedrich-Dessauer-Gymnasium 20/07/2023

# Numerical solution of the Schrödinger equation

## Summary:

The numerical solution of the Schrödinger equation with an arbitrary potential for a Gauss-packet in one and two dimensions using the Crank-Nicolson and Gauss-Seidel methods as well as the visualization of the solution.

## Goal:

Implementing a numerical solution and visualization of the Schrödinger equation with an arbitrary potential and arbitrary initial conditions as an introduction to numerical to the numerical solution of partial differential equations, in particular heat equations, and their visualization.

## Methods:

### Crank-Nicolson Method:

A finite difference method for the solution of partial differential equations constructs the average of the Euler-forwards and Euler-backwards methods for the solution of partial second derivatives.

$$\frac{u_i^{n+1} - u_i^n}{\Delta t} = \frac{1}{2}\left(F_i^{n+1}\left(u, x, t, \frac{\partial u}{\partial x}, \frac{\partial^2 u}{\partial x^2}\right) + F_i^n\left(u, x, t, \frac{\partial u}{\partial x}, \frac{\partial^2 u}{\partial x^2}\right)\right) \qquad \text{(Crank-Nicolson)}$$

Fig. 1: Crank-Nicolson method

To solve a partial differential equation with this method it is necessary to solve a system of linear equations. In the case of partial differential equations with a similar form to heat equations, like the Schrödinger equation in this case, the solution of this system of linear equations is only dependent on a tridiagonal matrix which drastically lowers the number of computational steps necessary to solve it.

### Gauss-Seidel Method:

An iterative method for solving systems of linear equations. The advantage of using an iterative method over a direct method is that we have more control over the numerical error which could occur.

$$x_k^{(m+1)} := \frac{1}{a_{kk}}\left(b_k - \sum_{i=1}^{k-1} a_{ki} \cdot x_i^{(m+1)} - \sum_{i=k+1}^{n} a_{ki} \cdot x_i^{(m)}\right),\ k = 1, \dots, n$$

Fig. 2: Gauß-Seidel method

Using this iterative method has a downside that the kth term is dependent on the k-1st term. This means that the algorithm cannot necessarily be parallelized.
An additional requirement for this method is that the matrix is strictly diagonally dominant. This requirement imposes a condition on our spatial and temporal resolutions of the solution due to the form of the diagonal elements of the tridiagonal matrix we receive from using the Crank-Nicolson method. A big upside and speed-up on the other hand is that due to the matrix being tridiagonal, the solution of the linear equations boils down to three terms.

### Programming Libraries:

For the implementation in C++, I used the linear algebra library “Eigen”.

For visualizing the data in Python I used Matplotlib, mainly for simple line plots, and Vispy for plotting 3d data due to its vastly better performance than Python.

### Boundary Conditions:

The boundary conditions are set so that the wave function always has a probability of zero of being at the edge of our space, so infinitely tall potential barriers on the sides imitating a closed box.

## The one-dimensional implementation:

I started by solving the Schrödinger equation in one dimension. I chose to do this to get a feeling for the methods and libraries I was later gonna use for the two-dimensional case.

First, I implemented both the Crank-Nicolson and Gauss-Seidel algorithms in C++ as well as set up the initial conditions of the system: a Gaussian wave packet with wave number k and a certain mean and standard deviation.

$$\Psi(x) = \frac{1}{\sqrt{2\pi}\sigma} \cdot e^{\frac{(x-\mu_x)^2}{\sigma^2}} \cdot e^{ikx}$$

Eq. 3: Gaussian wave packet with wave number k

As a first simple system, I set up the potential to be a finitely tall potential barrier. Solving this potential using the algorithms I implemented yielded these results:

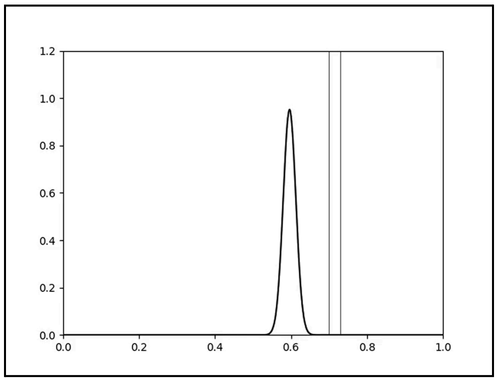
Fig. 1: Wave packet before impact

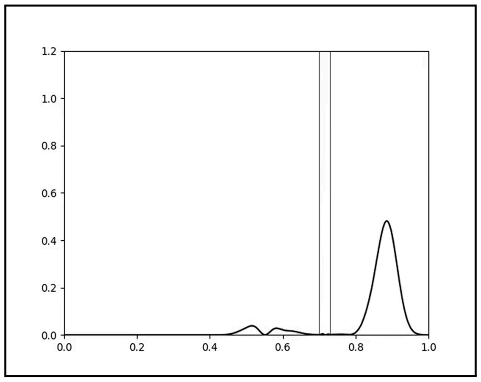
Fig 2: Wave packet after the impact

You can see a very clear reflection of the wave packet at the potential wall, as well as the remaining wave packet continuing behind the potential wall. This is exactly what you would expect to see in the case of a Gaussian wave packet interacting with a finitely tall potential barrier.

For the one-dimensional case of the finitely tall potential barrier, I also transformed the wave packet from position-space to k-space using a Fourier transformation and a very simple integration method

$$\Phi(k) = \frac{1}{\sqrt{2\pi}} \int_V \Psi(x) \cdot e^{-ikx} dx$$

Eq. 4: Fourier Transformation

$$\int_a^b f(x)\,dx \approx \frac{b-a}{n} \left( \frac{f(a)}{2} + \sum_{k=1}^{n-1} \left( f\left( a + k\frac{b-a}{n} \right) \right) + \frac{f(b)}{2} \right)$$

Eq. 5: Trapezoidal integration method

yielding these results:

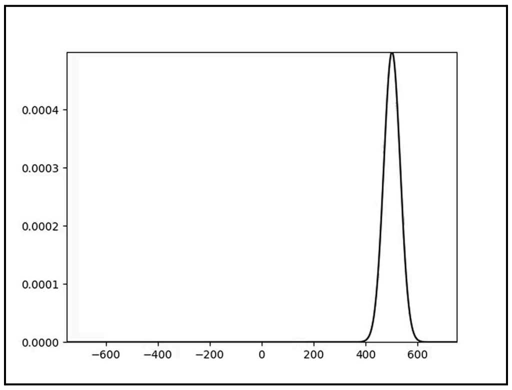
Fig 3: Before interacting with the barrier

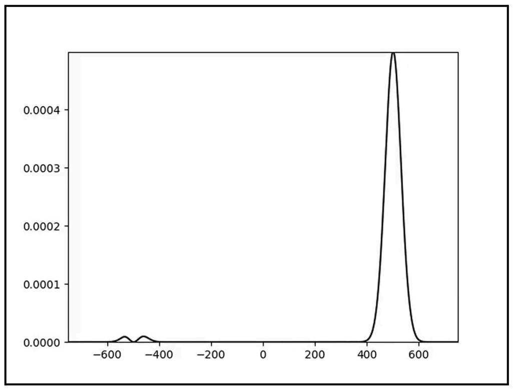
Fig 4: After interacting with the barrier

When solving this I did not normalize the initial wave packet which leads to very obviously incorrect orders of magnitude. Nevertheless, you can very clearly see the split of the wave packet at the potential barrier in k-space (Fig. 3 and Fig. 4) as well, resulting in the original Gaussian packet at around k = 500 and two new smaller wave packets at k = -400 and k = -600.

## Two-dimensional Implementation:

After solving the Schrödinger equation in one dimension and getting a feeling for the methods and libraries in use I decide to go over to solving the same problem in two dimensions. Visualizing the absolute square of the wave function in two dimensions requires a three-dimensional surface plot or a two-dimensional heat plot. I decided to go for the prior because it produced more easily readable plots. The issue with using three-dimensional surface plots is that the plotting library I had been using so far, Matplotlib, is very slow at producing 3-dimensional plots. Due to this performance issue, I decided to start looking for a more performant 3d plotting library and ended up settling for Vispy which yielded a speed-up of at least 10x or 20x.

The first system I solved in 3 dimensions was once again the finitely tall potential barrier. Solving this problem and plotting the data (here still using Matplotlib) yielded the following results:

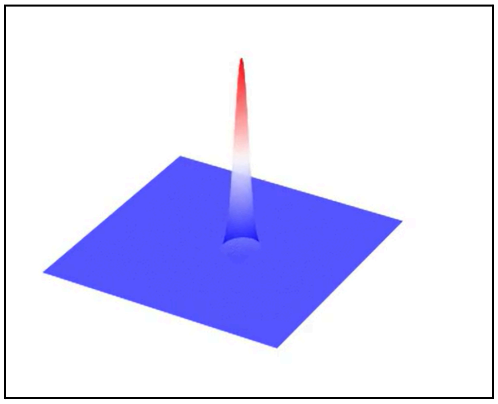
Fig 5: Wave packet before

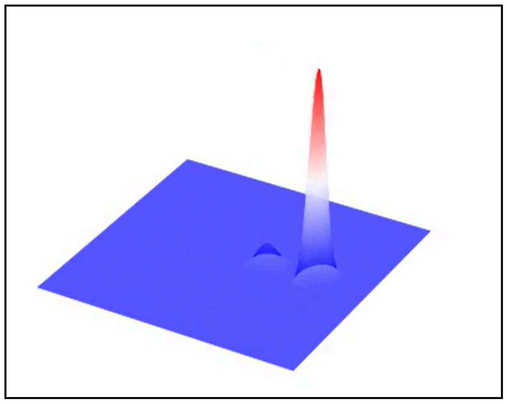
Fig 6: Wave packet after

Once again you can see a very clear reflection of the wave packet at the potential wall, these results are pretty much identical to the ones in 1 dimension.

After solving this first relatively simple potential I wanted to now start solving the double-slit potential. As a first implementation of this potential, I constructed the walls from finitely high potential barriers instead of infinite potential barriers. This led to a similar tunneling effect as in the previous solution (as can be seen in Fig. 7) and negatively influenced the resulting interference pattern.

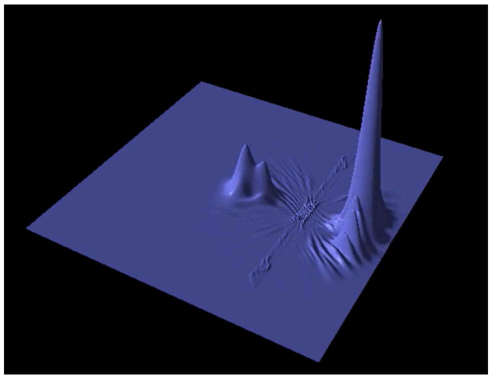

Fig 7: Gaussian wave packet after the double slit

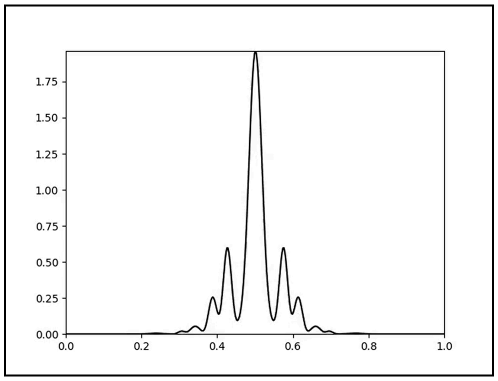

Fig 8: The resulting interference pattern

The general shape of the interference pattern already looks very promising but it is still very clearly not entirely what you would expect from the classic double slit experiment which uses infinite potential barriers as walls.
After noticing this I changed the potential from a finite potential barrier to an infinite one by directly setting the probability of the wave function within the wall to 0. Implementing this change led to far better results. To show this I solved the double-slit experiment and an equivalent single-slit experiment and overlaid their interference patterns.

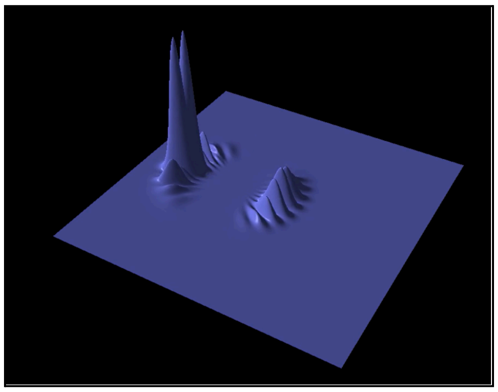

Fig 9: Gaussian wave packet after the double slit

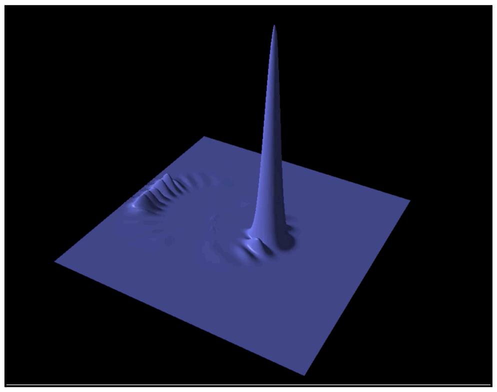

Fig 10: Gaussian wave packet after the single slit

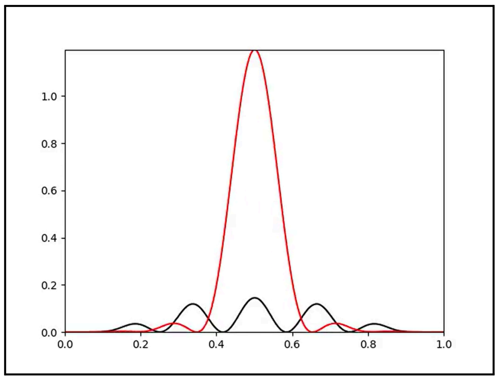

Fig 11:
Single slit distribution: Red
Double slit distribution: Black

These two distributions already reflect what we know from the two experiments a lot more closely. You can even make out the correlation of the first minimum of the single-slit distribution lining up with the first maximum of the double-slit distribution. The results of the numerical solution are limited by their spatial resolution. Solving this same problem with a higher temporal and spatial resolution would yield even more accurate results even though I did not explicitly check this.

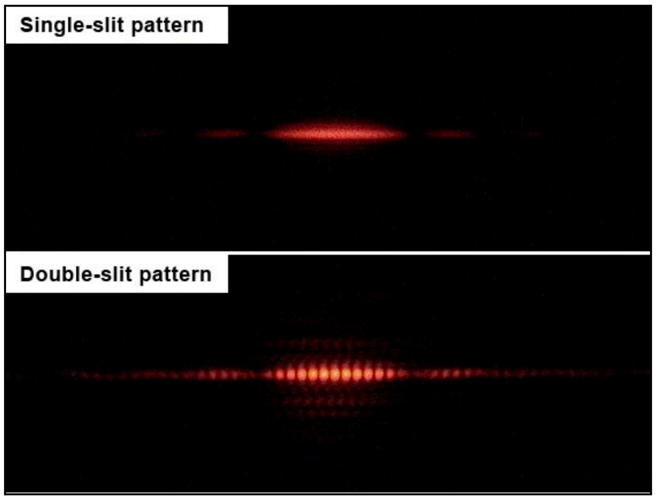

Fig 12: Experimental data of the two experiments

After having used the Gauss-Seidel method to solve the problem I initially wanted to solve, I also wanted to analyze its efficiency with different temporal and spatial resolutions. To do this I ran the program with the same initial conditions for multiple different spatial and temporal resolutions and plotted the average amount of iteration steps against the temporal resolution. This shows that the method very quickly approaches a limit. Choosing the best temporal resolution for a given spatial resolution, therefore, does not necessarily corresponds with choosing the highest possible temporal resolution but rather with choosing the optimized resolution.

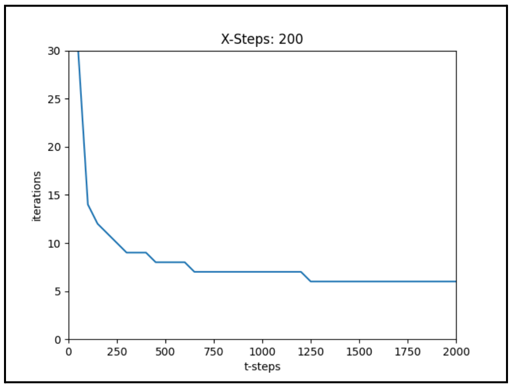

Fig 13

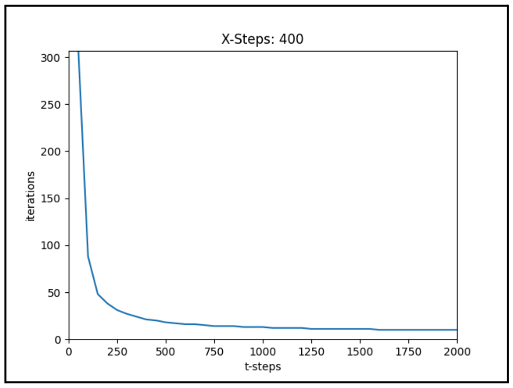

Fig 14

Multiplying these distributions with the number of computations every iteration takes yields graphs with obviously visible maxima which represent the “sweet spot”.

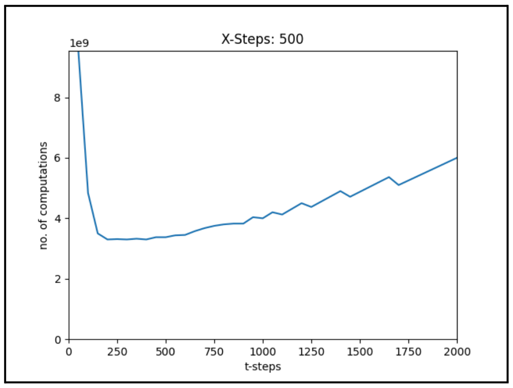

Fig 15

## Conclusion:

I numerically solved the Schrödinger equation in two dimensions for the double slit experiment. Solving this problem gave me a good first look into the numerical solution of partial differential equations and the visualization of the solutions. In particular the solution of simple quantum systems and their visualization.

## Sources:

https://en.wikipedia.org/wiki/Crank%E2%80%93Nicolson_method
https://en.wikipedia.org/wiki/Gauss%E2%80%93Seidel_method
http://hyperphysics.phy-astr.gsu.edu/hbase/phyopt/sindoub.html
https://en.wikipedia.org/wiki/Fourier_transform
https://en.wikipedia.org/wiki/Numerical_integration
https://en.wikipedia.org/wiki/Double-slit_experiment
Lecture notes by Prof. Engel

## Programming libraries/ software:

https://matplotlib.org/
https://vispy.org/
https://gitlab.com/libeigen/eigen

## Figures:

Eq 1: https://de.wikipedia.org/wiki/Crank-Nicolson-Verfahren
Eq 2: https://de.wikipedia.org/wiki/Gau%C3%9F-Seidel-Verfahren
Eq 3 and Eq 4: self-made using Latex
Fig 12: https://en.wikipedia.org/wiki/Double-slit_experiment
All other figures were created by me using Matplotlib or Vispy

## Code:

https://github.com/Xilaeth/phys

All links work as of 20/07/2023