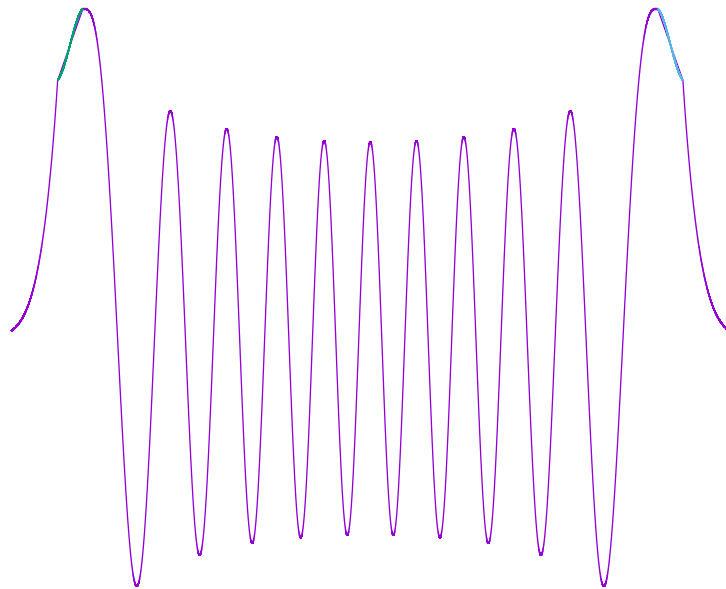


Approximating Solutions of the Time Independent Schrödinger Equation

Gian Laager
October 14, 2022



Maturaarbeit
Kantonsschule Glarus
Betreuer: Linus Romer
Referent: Elena Borisova

Contents

1. Introduction	1
1.1. Goals	1
2. Preliminary	2
2.1. Schrödinger Equation	2
2.2. Rust	2
2.3. Complex Numbers	3
2.4. Gnuplot	3
3. Methods	4
3.1. Newtons Method	4
3.2. Derivatives	5
3.3. Integration	7
3.4. Transition Regions	9
3.4.1. Implementation in Rust	12
3.5. Energy Levels	13
3.6. Approximation Scheme	13
3.6.1. Example	14
3.6.2. Validity	16
A. Source Code	17
B. Detailed Calculations	18
B.1. Proofs	18
B.1.1. Smoothness of Transitionfunction	18

1. Introduction

Richard Feynmann one of the core people behind our modern theory of quantum mechanics repeatedly said: “I think I can safely say that nobody understands quantum mechanics.”. Nothing behaves like in our every day lives. Everything is just a probability and nothing certain. Even Schrödinger the inventor of the equation that governs all of those weird phenomena rejected the idea that there are just probabilities.

In this paper we will try to understand this world a little bit better by looking at wave functions in a simplified universe. This universe only has 1 dimension and there will not be any sense of time. This means we will be able to actually see how the wave function looks like in a graph.

1.1. Goals

The goal of this Maturaarbeit is to write a program, `schroeding-approx` that calculates solutions to the time independent Schrödinger equation in 1 dimension for a large variety of potentials. We assume that the wave function, $\Psi(x)$ will diverge to 0 as x goes to infinity.

The program uses Planck units which means $c = \hbar = G = 1$

2. Preliminary

2.1. Schrödinger Equation

In 1926 Erwin Schrödinger changed our understanding of quantum physics with the Schrödinger equation. Based on the observations of de Broglie that particles behave like waves he developed a wave equation which describes how the waves move and change in a given potential $V(x)$ or Hamiltonian \hat{H} .

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

Or more general

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

The time independent version that is going to be used later, ignores the change over time and is much simpler to solve since it is **only** an ordinary differential equation instead of a partial differential equation.

$$E\psi(x) = \hat{H}\psi(x)$$

or

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

Even with the time independent equation it is very difficult to get analytical solutions, because of this there are mainly three approaches to approximate solutions of $\psi(x)$, perturbation theory, density functional field theory and WKB approximation. Perturbation theory's goal is to give an analytical approximation which means it is extremely difficult to implement for a computer. WKB on the other hand is much better since it is to some degree a step by step manual.

2.2. Rust

Rust is one of the newer programming languages and attempts to replace C/C++ which are notoriously difficult to work with. It supports both functional and object-oriented paradigms. It is much safer in terms of memory and promises the same performance as C. One of the goals of Rust is fearless concurrency which means everybody should be able to write concurrent code without deadlocks and data races. This means calculations can utilize the full potential of the CPU without countless hours of debugging.

Functional programming languages are especially useful for mathematical problems, because they are based on the same mathematics as the problem.

Rust as of the time of writing this document is not yet standardized meaning the code provided might no longer be correct with one of the newer Rust versions.

In case you aren't familiar with Rust you it has excellent documentation on <https://doc.rust-lang.org/book/>.

2.3. Complex Numbers

In quantum mechanics it's customary to work with complex numbers. Complex numbers are an extension to the real numbers, since Rust will do most of the heavy lifting here are the most important things that you should know

$$i^2 = -1$$

$$z = a + bi$$

$$\text{Re}(z) = a$$

$$\text{Im}(z) = b$$

$$\bar{z} = a - bi$$

$$\|z\|^2 = a^2 + b^2$$

$$e^{a+bi} = e^a (\cos(b) + i \sin(b))$$

i is the imaginary unit, z is the general form of a complex number where $\{a, b\} \in \mathbb{R}$, \bar{z} is the complex conjugate and $\|z\|^2$ is the norm square of z . The last equation is the Euler's formula, it rotates a number in the complex plane by b radians.

The complex plane is similar to the real number line, every complex number can be represented on this plane where $\text{Re}(z)$ is the x-coordinate and $\text{Im}(z)$ is the y-coordinate.

2.4. Gnuplot

Gnuplot is a cross platform plotting program that is very simple to use. `schroedinger-approx` will output a file `data.txt`, you can plot the function by typing `gnuplot` and then typing

```
call "plot.gnuplot"
```

to plot the real part of the wave function, or

```
call "plot_3d.gnuplot"
```

to see the full complex wave function.

If you'd like to learn more about Gnuplot you can read there user manual on <http://www.gnuplot.info/>

3. Methods

3.1. Newtons Method

Newton's method, also called the Newton-Raphson method, is a root-finding algorithm that uses the first few terms of the Taylor series of a function $f(x)$ in the vicinity of a suspected root (Weisstein, 2022). It makes a sequence of approximations of a root x_n that in certain cases converges to the exact value where

$$\lim_{n \rightarrow \infty} f(x_n) = 0$$

The sequence is defined as

$$x_0 = a$$
$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)}$$

Visually this looks like figure 3.1 $f(x) = (x - 2)(x - 1)(x + 1)$. The blue line indicates the initial guess which in this case is 2.5 the black line ($g(x)$) is a tangent to $f(x)$ at $(guess, f(guess))$ the next guess will be where the tangent intersects the x-Axis (solution of $g(x) = 0$). This will converge rather quickly compared to other methods such as Regula-Falsi.

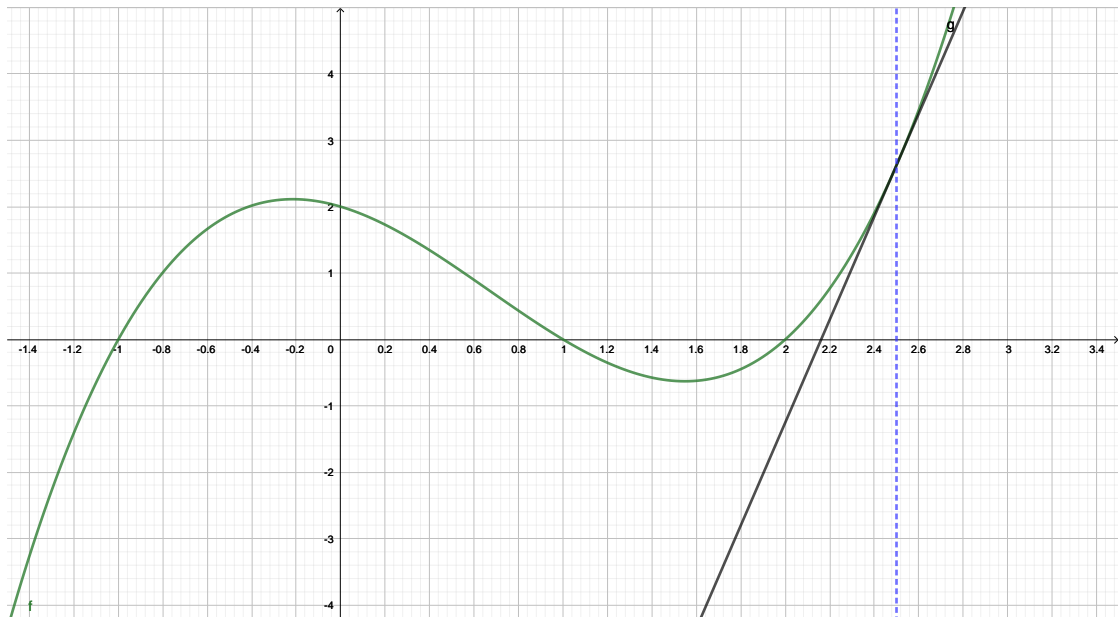


Figure 3.1.: Illustration of Newton's method, $f(x) = (x - 1)(x - 2)(x + 1)$.

```

1 pub fn newtons_method<F>(f: &F, mut guess: f64, precision: f64) -> f64
2     where
3         F: Fn(f64) -> f64,
4     {
5         loop {
6             let step = f(guess) / derivative(f, guess);
7             if step.abs() < precision {
8                 return guess;
9             } else {
10                guess -= step;
11            }
12        }
13    }

```

In Rust the sequence is implemented with a function that takes a closure f , the initial guess $guess$ and a stop condition $precision$ the function will return if $|\frac{f(x_n)}{f'(x_n)}|$ is less than $precision$.

From the structure of the algorithm it is very tempting to implement it recursively, but by using a loop it is much faster since there are no unnecessary jumps and the precision can (at least in theory) be 0 without causing a stack overflow.

3.2. Derivatives

The precision gained by calculating derivatives analytically would come with a massive overhead both in development and in performance, one would have to implement all the rules

of differentiation and create some representation of an actual function to which those rules could be applied. I wrote an implementation of this in Go and not much accuracy can actually be gained.

A much easier approach is to approximate it. The definition of a derivative of a function $f(x)$ is

$$\frac{df}{dx} = \lim_{h \rightarrow 0} \frac{f(x+h) - f(x)}{h}$$

The limit can very easily be approximated with

$$\frac{f(x + \frac{\delta}{2}) - f(x - \frac{\delta}{2})}{\delta}$$

where δ needs to be small yet not so small that it exceeds the 16 digit precision of double. I chose $\delta = \sqrt{2^{-52}}$, because $\epsilon = 2^{-52}$ is the smallest double precision floating point number where $1 + \epsilon \neq 1$. The square root ensures that still enough precision remains. After some further research the function should be implemented the same way as described by Boost (C++ library).


```

1 pub fn derivative<F, R>(f: &F, x: f64) -> R
2     where
3         F: Fn(f64) -> R,
4         R: Sub<R, Output=R> + Div<f64, Output=R>,
5     {
6         let epsilon = f64::epsilon().sqrt();
7         (f(x + epsilon / 2.0) - f(x - epsilon / 2.0)) / epsilon
8     }

```

`f64::epsilon()` this is the difference between ‘1.0’ and the next larger representable number (rust lang, 2022). `derivative` is implemented not only for `f64` but for all types that support subtraction and division by `f64` this means it can also be used for closures with real inputs and complex outputs.

3.3. Integration

The same principles apply to integrals as to derivative it would not be a great benefit to implement an analytic integration system. Integrals would also be much more difficult to implement than derivatives since integrals can not be broken down in to many smaller integrals that can be computed easily instead it needs to be solved as is.

One approach would be to use the same method as with the derivative, take the definition with the limit and use a small value but this method can be improved in this case, since integrals calculate areas under curves a trapeze is more efficient and accurate then the rectangle that results from the definition.

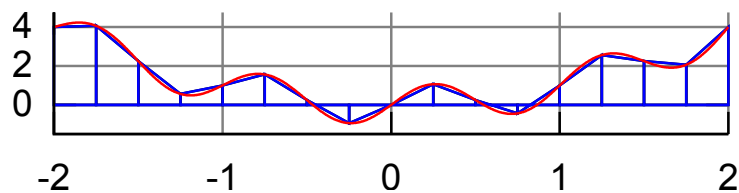


Figure 3.2.: Illustration of integration with trapeze from Wikipedia (2022a).

Figure 3.2 shows visually how the methods work, each blue trapeze from start (a) to end (b) has an area of

$$\int_a^b f(x) dx \approx (b-a)f\left(\frac{a+b}{2}\right).$$

One trapeze would be fairly inaccurate to calculate the area under the function but as the area from a to b is subdivided further the result become better and better.

The general structure of the algorithm can very easily be run in parallel since it doesn’t matter in which order the segments are added together and the segments also don’t dependent on one another. In Rust this is implemented using `rayon`. `Rayon` is an implementation for parallel iterators meaning that normal data structures that implement `std::iter` can be

run in parallel *just* by changing `::iter()` to `::par_iter()`. This might not work in all cases because of memory safety but in this case the borrow checker will throw an error and the code won't compile.

```
1 pub trait ReToC: Sync {
2     fn eval(&self, x: &f64) -> Complex64;
3 }
4
5 pub struct Point {
6     pub x: f64,
7     pub y: Complex64,
8 }
```

These functions were implemented very early and need some refactoring. Such that functions with states, like wave functions that store parameters, can be integrated there is a trait `ReToC`. `ReToC` describes a function $f : \mathbb{R} \rightarrow \mathbb{C}$ (`Fn(f64) -> Complex64`).

`Point` stores both the input (`x`) and the output (`y`) of a function.

```
1 pub fn evaluate_function_between(f: &dyn ReToC, a: f64, b: f64, n: usize) -> Vec<Point> {
2     if a == b {
3         return vec![];
4     }
5
6     (0..n)
7         .into_par_iter()
8         .map(|i| index_to_range(i as f64, 0.0, n as f64 - 1.0, a, b))
9         .map(|x| Point { x, y: f.eval(&x) })
10        .collect()
11 }
```

`ReToC` can be passed to `evaluate_function_between` it calculates `n` points between an interval from `a` to `b` and returns a vector of `Point`.

```

1 pub fn trapezoidal_approx(start: &Point, end: &Point) -> Complex64 {
2     return complex(end.x - start.x, 0.0) * (start.y + end.y) / complex(2.0, 0.0);
3 }
4
5 pub fn index_to_range(x: f64, in_min: f64, in_max: f64, out_min: f64, out_max: f64) -> f64 {
6     return (x - in_min) * (out_max - out_min) / (in_max - in_min) + out_min;
7 }
8
9 pub fn integrate(points: Vec<Point>, batch_size: usize) -> Complex64 {
10     if points.len() < 2 {
11         return complex(0.0, 0.0);
12     }
13
14     let batches: Vec<&[Point]> = points.chunks(batch_size).collect();
15
16     let parallel: Complex64 = batches
17         .par_iter()
18         .map(|batch| {
19             let mut sum = complex(0.0, 0.0);
20             for i in 0..(batch.len() - 1) {
21                 sum += trapezoidal_approx(&batch[i], &batch[i + 1]);
22             }
23             return sum;
24         })
25         .sum();
26
27     let mut rest = complex(0.0, 0.0);
28
29     for i in 0..batches.len() - 1 {
30         rest += trapezoidal_approx(&batches[i][batches[i].len() - 1], &batches[i + 1][0]);
31     }
32
33     return parallel + rest;
34 }

```

The actual integration happens in `integrate`, it calculates the areas of the trapezes between the points passed to it. For optimization 1000 trapezes are calculated per thread because it would take more time to create a new thread then to actually do the calculation, this has to be further investigated and 1000 might not be optimal. The calculations performed per thread are called a batch, after all batches have been calculated the boundaries between batches also has to be considered therefor they are added in the end with `rest`

3.4. Transition Regions

The approximation that will be used splits $\Psi(x)$ into multiple parts that do not match perfectly together.

Lets consider an example, in figure 3.4 we can see two Taylor series of cosine. Now we

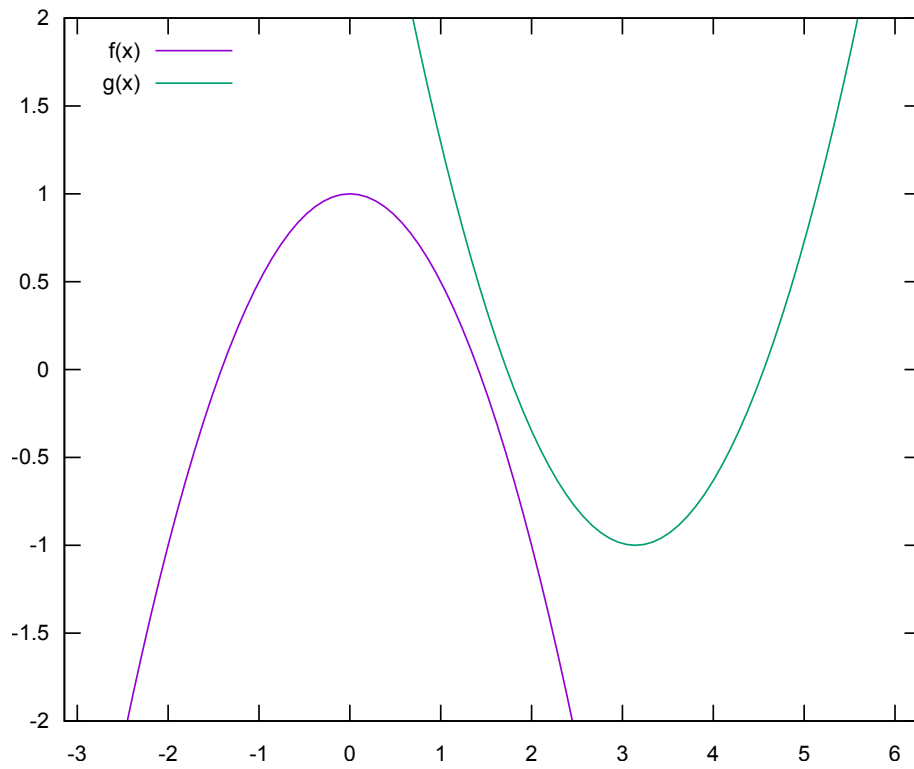


Figure 3.3.: Example for joining functions

have to join the two functions at $x = \pi/2$ such that its a mathematically smooth transition.

$$f(x) = 1 - \frac{x^2}{2} \quad (3.1)$$

$$g(x) = \frac{(x - \pi)^2}{2} - 1 \quad (3.2)$$

As a first guess lets join $f(x)$ and $g(x)$ with a step function, this means that the joint function $h(x)$ will be

$$h(x) = \begin{cases} f(x) & x < \frac{\pi}{2} \\ g(x) & x > \frac{\pi}{2} \end{cases} .$$

This gives us 3.4 which is obviously not smooth.

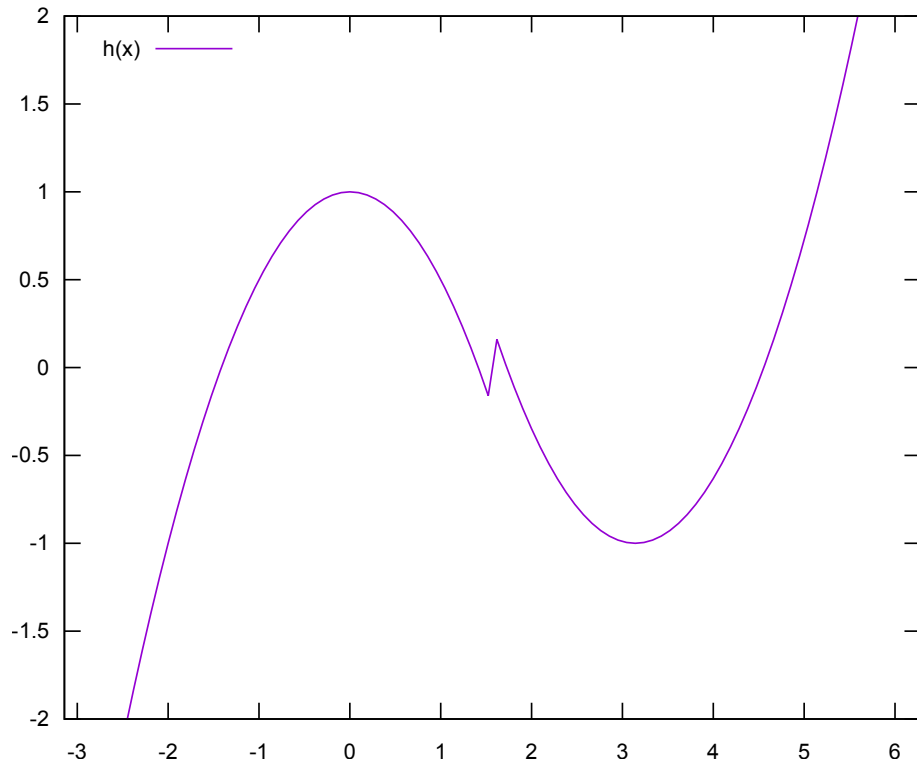


Figure 3.4.: Plot of $h(x)$ with step joint

If we use the formula from (Hall, 2013, p. 325, section 15.6.4) with

$$\begin{aligned}\delta &= 0.5 \\ \alpha &= \frac{\pi}{2} - \frac{\delta}{2} \\ \chi(x) &= \sin^2\left(x \frac{\pi}{2}\right)\end{aligned}$$

this results in

$$h(x) = \begin{cases} f(x) & x < \alpha \\ g(x) & x > \alpha + \delta \\ f(x) + (g(x) - f(x))\chi\left(\frac{x-\alpha}{\delta}\right) & \text{else} \end{cases}$$

which is mathematically smooth as we can see in figure 3.4 (proof in Appendix B.1.1).

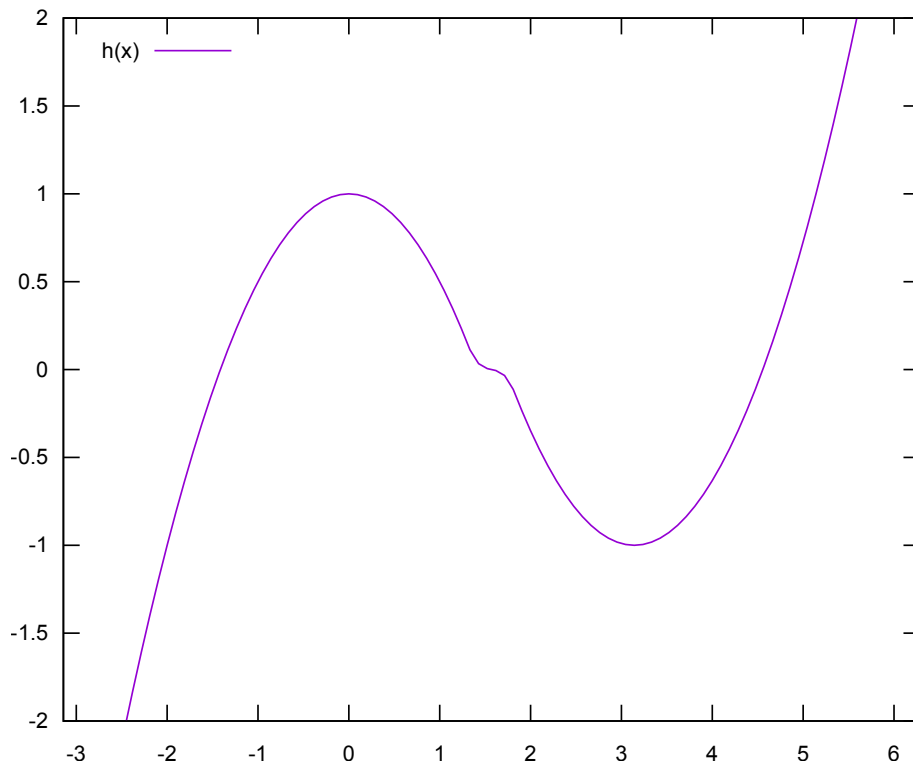


Figure 3.5.: Plot of $h(x)$ with Hall joint

3.4.1. Implementation in Rust

In the program we can define a struct `Joint` that implements `Func<f64, Complex64>`. As in the example we need two functions $f(x)$ and $g(x)$ which we will rename to `left` and `right`. We will also need a variable α and δ which will be named `cut` and `delta`.

```

1  #[derive(Clone)]
2  pub struct Joint {
3      pub left: Arc<dyn Func<f64, Complex64>>,
4      pub right: Arc<dyn Func<f64, Complex64>>,
5      pub cut: f64,
6      pub delta: f64,
7  }
8
9  impl Func<f64, Complex64> for Joint {
10     fn eval(&self, x: f64) -> Complex64 {
11         let chi = |x: f64| f64::sin(x * f64::consts::PI / 2.0).powi(2);
12         let left_val = left.eval(x);
13         return left_val + (right.eval(x) - left_val) * chi((x - self.cut) / self.delta)
14     }
15 }

```

In the proof we assume that $f(x)$ and $g(x)$ are continuous of first order in the interval $(\alpha, \alpha + \delta)$. In the code we will not check this requirement since it would have a major impact on performance to check the derivative on every point.

3.5. Energy Levels

Solving the Schrödinger equation is an eigenvalue problem. This means that only certain energies will result in physically correct results. For an energy to be valid it has to satisfy the Maslov-corrected Bhor-Sommerfeld condition which states that

$$n \in \mathbb{N}_0 \quad (3.3)$$

$$C = \{x \in \mathbb{R} \mid V(x) < E\} \quad (3.4)$$

$$\int_C \sqrt{2m(E - V(x))} dx = 2\pi(n + 1/2) \quad (3.5)$$

this condition does not (in most cases) give the exact energy levels (Hall, 2013). It can be interpreted such that the oscillating part of the wave function has to complete all half oscillation.

To solve this problem for an arbitrary potential in a computer the set C and the fact that n has to be a non negative integer is not really helpful, but the condition can be rewritten to

$$p(x) = \begin{cases} \sqrt{2m(E - V(x))} & V(x) < E \\ 0 & \text{else} \end{cases} \quad (3.6)$$

$$\frac{1}{2\pi} \int_{-\infty}^{\infty} p(x) dx - \frac{1}{2} \mod 1 = 0 \quad (3.7)$$

Unfortunately 3.7 is not continuous which means that Newtons method can't be applied. Further on the bounds of integration have to be finite, this means the user of the program will have to specify a value for the constant APPROX_INF where any value for x out side of that range should satisfy $V(x) > E$. But it shouldn't be to big since the integrate function can only evaluate a relatively small number (default 64000) of trapezes before the performance will suffer enormously. The default value for APPROX_INF is $(-200.0, 200.0)$.

3.6. Approximation Scheme

There are mainly three approximation methods used to solve for the actual wave function itself. There is perturbation theory which breaks the problem down in to ever smaller sub-problems that then can be solved exactly. This can be achieved by adding something to the Hamiltonian operator \hat{H} which can then be solved exactly. But *perturbation theory is inefficient compared to other approximation methods when calculated on a computer* (Van Mourik et al., 2014, Introduction).

The second is Density functional field theory, it has evolved over the years and is used heavily in chemistry to calculate properties of molecules and is also applicable for the time dependent Schrödinger equation. It is something that might be interesting to add to the

program in the future.

The program uses the third method WKB approximation, it is applicable to a wide variety of linear differential equations and works very well in the case of the Schrödinger equation. Originally it was developed by Wentzel, Kramers and Brillouin in 1926. It gives an approximation to the eigenfunctions of the Hamiltonian \hat{H} in one dimension. The approximation is best understood as applying to a fixed range of energies as \hbar tends to zero (Hall, 2013, p. 305).

WKB splits $\Psi(x)$ into three parts that can be connected to form the full solution. The three parts are described as

$$p(x) = \sqrt{2m(|E - V(x)|)} \quad (3.8)$$

$$V(t) - E = 0 \quad (3.9)$$

$$\psi_{exp}^{WKB}(x) = e^{\delta i} \frac{c_1}{2\sqrt{p(x)}} \exp\left(-\left|\int_x^t p(y)dy\right|\right) \quad (3.10)$$

$$\psi_{osz}^{WKB}(x) = \frac{c_1}{\sqrt{p(x)}} \exp\left(-\left|\int_x^t p(y)dy\right| i + \delta i\right) \quad (3.11)$$

$$u_1 = -2m \frac{dV}{dx}(t) \quad (3.12)$$

$$\psi^{Airy}(x) = e^{(t-x+\delta)i} \frac{c_1 \sqrt{\pi}}{\sqrt[6]{u_1}} \text{Ai}\left(\sqrt[3]{u_1}(t-x)\right). \quad (3.13)$$

Since equation 3.9 might have more than one solution for turning points t , we have to consider each one of them individually and in the end join them into one function.

In figure 3.6 the three parts are visualized. The purple section on the left is the exponential decaying part $\psi_{exp}^{WKB}(x)$, equation 3.10 is a modified version of the original version as described in (Hall, 2013, p. 317, eq. 15.25) where b and a are different solutions for t of equation 3.9. The absolute symbol makes it possible to not differentiate between the case where $x < t$ and $x > t$. Further on a factor of $e^{\delta i}$ was added such that the imaginary part of $\psi_{exp}^{WKB}(x)$ is the same as in $\psi_{osz}^{WKB}(x)$.

The blue part on the right is $\psi_{osz}^{WKB}(x)$. Again equation 3.11 was expanded to result in the more general complex solution and it also works for both ψ_1 and ψ_2 in (Hall, 2013, p. 316-317, Claim 15.7). Hall (2013) assumes that $\delta = \pi/4$ which doesn't work in the simple case of $V(x) = x^2$, in figure 3.6 $\delta = 0$ was used. This will be further discussed in section ??.

3.6.1. Example

This example is from (Wikipedia, 2022b, An example).

Lets solve the ordinary differential equation

$$\epsilon^2 \frac{d^2 y}{dx^2} = Q(x)y$$

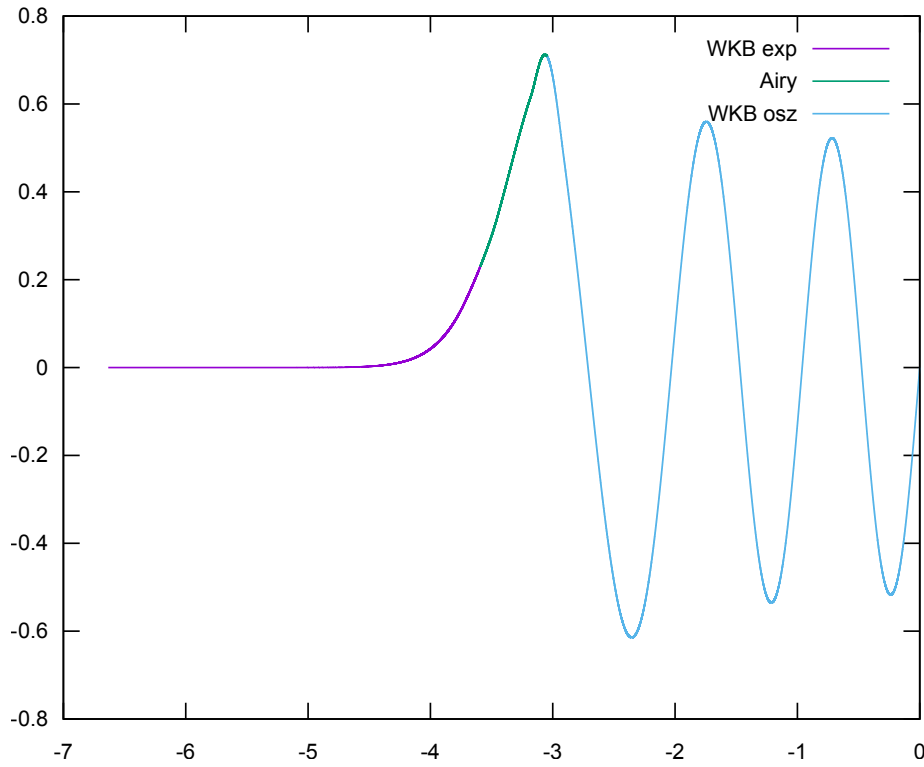


Figure 3.6.: Left half of wave function with $N_{Energy} = 5 \Rightarrow E = 11.0$, $m = 2$, $V(x) = x^2$

where $Q(x)$ is an arbitrary function that is not $Q(x) = 0$ and ϵ is small. Note that this example relates to the Schrödinger equation where $\epsilon = -\frac{\hbar^2}{2m}$ and $Q(x) = E - V(x)$.

Replace y with

$$y(x) = \exp\left(\frac{1}{\delta} \sum_{n=0}^{\infty} \delta^n S_n(x)\right)$$

resulting in the equation

$$\epsilon^2 \frac{1}{\delta^2} \left(\sum_{n=0}^{\infty} \delta^n S'_n(x) + \frac{1}{\delta} \left(\sum_{n=0}^{\infty} \delta^n S''_n(x) \right) \right) = Q(x) \quad (3.14)$$

As an approximation set the upper bound of the sum to be 2 instead of ∞ , then 3.14 can be written as

$$\frac{\epsilon^2}{\delta^2} S_0'^2 + \frac{2\epsilon^2}{\delta} S_0' S_1' + \frac{\epsilon^2}{\delta} S_0'' = Q(x) \quad (3.15)$$

If we go further and solve for S_0 and S_1 we will get that

$$S_0(x) = \pm \int_{x_0}^x \sqrt{Q(t)} dt \quad (3.16)$$

$$S_1(x) = -\frac{1}{4} \ln Q(x) + k_1 \quad (3.17)$$

this means that our approximate solution of y is

$$y(x) \approx c_1 Q^{-\frac{1}{4}}(x) \exp\left(\frac{1}{\epsilon} \int_{x_0}^x \sqrt{Q(t)} dt\right) + c_2 Q^{-\frac{1}{4}}(x) \exp\left(-\frac{1}{\epsilon} \int_{x_0}^x \sqrt{Q(t)} dt\right)$$

here c_1, c_2 depend on the boundary conditions and can also be used to renormalize the function. x_0 will later be discussed in section (...). It is important to have the right value for x_0 which depends on $V(x)$ and E although I have not yet managed to find out the connection.

3.6.2. Validity

Dieser abschnitt ist momentan nur fürs Probe Kapitel muss ich noch besser nach Rechnen und genau verstehen

The approximation that was made in equation 3.14 to 3.15 that we ignored the terms of $n > 2$ and assumed that they are small.

$$\epsilon \left| \frac{dQ}{dx} \right| \ll Q(x)^2$$

Without the assumption that \hbar is infact *small* the equation can also be interpreted such that the potential $V(x)$ has to vary slowly compared to $(V(x) - E)^2$.

During calculations I noticed that it is crucial what values x_0 has sinse some times the results can not be valid.

A. Source Code

At the moment the code is only available on <https://github.com/Gian-Laager/Schroedinger-Approximation>

B. Detailed Calculations

B.1. Proofs

B.1.1. Smoothness of Transitionfunction

Given that

$$f : \mathbb{R} \rightarrow \mathbb{C} \quad (\text{B.1})$$

$$g : \mathbb{R} \rightarrow \mathbb{C} \quad (\text{B.2})$$

$$\{f, g\} \in C^1 \quad (\text{B.3})$$

$$\{\alpha, \delta\} \in \mathbb{C} \quad (\text{B.4})$$

define (Hall, 2013)

$$\chi(x) = \sin^2\left(\frac{\pi(x - \alpha)}{2\delta}\right) \quad (\text{B.5})$$

$$(f \sqcup g)(x) = f(x) + (g(x) - f(x))\chi(x) \quad (\text{B.6})$$

and proof that

$$\frac{d(f \sqcup g)}{dx}(\alpha) = \frac{df}{dx}(\alpha) \quad (\text{B.7})$$

$$\frac{d(f \sqcup g)}{dx}(\alpha + \delta) = \frac{dg}{dx}(\alpha + \delta). \quad (\text{B.8})$$

Calculate derivatives

$$\frac{d\chi}{dx}(x) = \frac{\pi}{2\delta} \sin\left(\frac{\pi(x - \alpha)}{\delta}\right) \quad (\text{B.9})$$

$$\frac{d(f \sqcup g)}{dx}(x) = \frac{df}{dx}(x) + \left(\frac{dg}{dx}(x) - \frac{df}{dx}(x)\right)\chi(x) + (g(x) - f(x))\frac{d\chi}{dx}(x). \quad (\text{B.10})$$

Note that

$$\frac{d\chi}{dx}(\alpha) = 0 \quad (\text{B.11})$$

$$\chi(\alpha) = 0 \quad (\text{B.12})$$

$$\frac{d\chi}{dx}(\alpha + \delta) = 0 \quad (\text{B.13})$$

$$\chi(\alpha + \delta) = 1 \quad (\text{B.14})$$

therefor

$$\frac{d(f \sqcup g)}{dx}(\alpha) = \frac{df}{dx}(\alpha) + 0 \left(\frac{dg}{dx}(\alpha) - \frac{df}{dx}(\alpha) \right) + 0(g(x) - f(x)) = \frac{df}{dx}(\alpha) \quad (\text{B.15})$$

and

$$\frac{d(f \sqcup g)}{dx}(\alpha + \delta) = \frac{df}{dx}(\alpha + \delta) + 1 \left(\frac{dg}{dx}(\alpha + \delta) - \frac{df}{dx}(\alpha + \delta) \right) + 0(g(x) - f(x)) \quad (\text{B.16})$$

$$\frac{d(f \sqcup g)}{dx}(\alpha + \delta) = \frac{df}{dx}(\alpha + \delta) + \frac{dg}{dx}(\alpha + \delta) - \frac{df}{dx}(\alpha + \delta) = \frac{dg}{dx}(\alpha + \delta) \blacksquare. \quad (\text{B.17})$$

Bildquellen

Wo nicht anders angegeben, sind die Bilder aus dieser Arbeit selbst erstellt worden.

Bibliography

Brain C. Hall. *Quantum Theory for Mathematicians*. Springer New York, NY, 1 edition, 2013. ISBN 978-1461471158.

rust lang. Constant std::f64::EPSILON, 2022. URL <https://doc.rust-lang.org/std/f64/constant.EPSILON.html>. [Online; accessed 18-August-2022].

Tanja Van Mourik, Michael Bühl, und Marie-Pierre Gageot. Density functional theory across chemistry, physics and biology, 2014.

Eric W. Weisstein. Newton's Method, 2022. URL <https://mathworld.wolfram.com/NewtonsMethod.html>. [Online; accessed 10-August-2022].

Wkipedia. Numerical integration, 2022a. URL https://en.wikipedia.org/wiki/Numerical_integration. [Online; accessed 10-August-2022].

Wkipedia. WKB approximation, 2022b. URL https://en.wikipedia.org/wiki/WKB_approximation. [Online; accessed 10-August-2022].

Selbständigkeitserklärung

Hiermit bestätige ich, Gian Laager, meine Maturaarbeit selbständig verfasst und alle Quellen angegeben zu haben.

Ich nehme zur Kenntnis, dass meine Arbeit zur Überprüfung der korrekten und vollständigen Angabe der Quellen mit Hilfe einer Software (Plagiaterkennungstool) geprüft wird. Zu meinem eigenen Schutz wird die Software auch dazu verwendet, später eingereichte Arbeiten mit meiner Arbeit elektronisch zu vergleichen und damit Abschriften und eine Verletzung meines Urheberrechts zu verhindern. Falls Verdacht besteht, dass mein Urheberrecht verletzt wurde, erkläre ich mich damit einverstanden, dass die Schulleitung meine Arbeit zu Prüfzwecken herausgibt.

Ort

Datum

Unterschrift