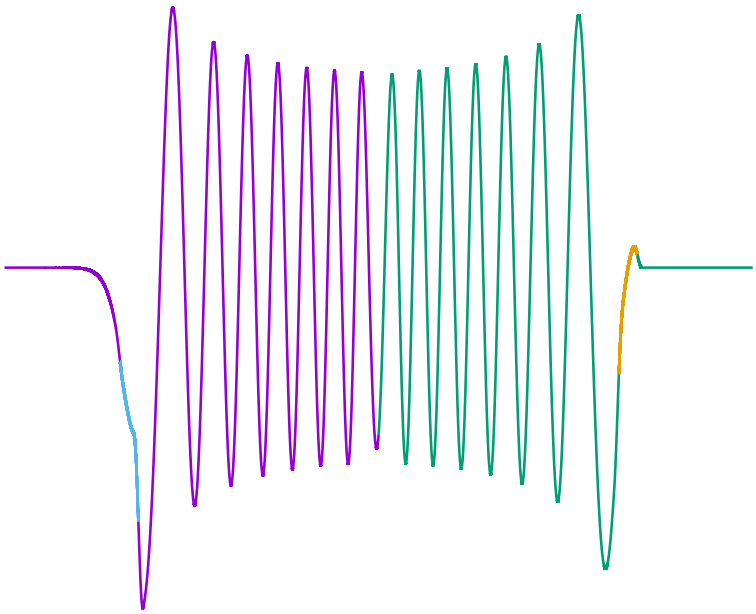
Approximating Solutions of the Time Independent Schrödinger Equation

Gian Laager
October 31, 2022



Maturaarbeit
Kantonsschule Glarus
Betreuer: Linus Romer
Referent: Elena Borisova

Contents

Vorwort

Der Rest der Arbeit wird in Englisch sein aber ich habe mich entschieden eine kleine Zusammenfassung zu schreiben, so dass jeder zumindest die Grundlagen meiner Arbeit versteht. Zu begin des 20. Jahrhunderts gab es einen Umschwung in der Physik, Quanten Mechanik wurde entdeckt. Diese neue Theorie kann nicht mehr präzise voraussagen machen wie es zuvor der Fall war. Man kann nur noch sagen mit welcher Wahrscheinlichkeit etwas passiert und ein Partikel kann an zwei Orten gleichzeitig sein.

Vielleicht haben Sie schon einmal von Schrödingers Katze gehört. Dies war ein Gedankenexperiment von Schrödinger um auf zu zeigen wie absurd seine Theorie wirklich ist und dass sie nicht stimmen könne. Stell dir vor du schliesst deine Katze in eine Box ein. In dieser Box ist ein Atom das entweder zerfallen kann oder nicht. Dazu gibt es einen Detektor der misst ob das Atom zerfallen ist, in diesem Fall wird ein Gift frei gelassen und die Katze stirbt. Das Problem ist jetzt aber, dass dieses Atom den Regeln der Quanten Mechanik folgt und deshalb gleichzeitig bereits zerfallen ist und nicht zerfallen ist, die einzig logische Schlussfolgerung ist deshalb, dass *die Katze gleichzeitig Tod und am leben ist*.

In der Realität funktioniert es wahrscheinlich jedoch nicht so. Heisst das Universum "entscheidet" ob die Katze gestorben ist oder nicht, jedoch weiss man bis Heute nicht wann das Universum "entscheidet".

Damit die Katze gleichzeitig Tod und Lebendig sein kann brauchen wir die Wellenfunktion. Sie beschreibt alles was in unserem Universum gerade passiert und "speichert" wie wahrscheinlich es ist, dass die Katze tot ist.

In meiner Maturaarbeit habe ich ein Programm geschrieben das genau diese Wellenfunktion ausrechnet in einem sehr vereinfachten Universum. Weil ich schon lange mal wissen möchte wie genau dieses bizarre Objekt aussieht. Auf der Titel Seite ist eine dieser Wellenfunktionen abgebildet.

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 verity of potentials. We assume that the wave function, $\Psi(x)$ will converges to 0 as x goes to $\pm\infty$.

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 $\hat{H}$.

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

Or more general

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

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, it has excellent documentation on `https://doc.rust-lang.org/book/`.

2.3. Interpretation of Quantum Mechanics

The author believes in the many worlds interpretation of Hugh Everett. *"The wave interpretation. This is the position proposed in the present thesis, in which the wave function itself is held to be the fundamental entity, obeying at all times a deterministic wave equation."* (DeWitt und Graham, 2015, p. 115). This means that the observer is also quantum mechanical and gets entangled with one particular state of the system that is being measured (DeWitt und Graham, 2015, p. 116). This is some what different to the popular explanation of many worlds but has the same results and is, at least to the author more reasonable.

An important point for the author also was that the theory accepts quantum mechanics as it is and doesn't make unreasonable assumption such as that the observer plays an important role.

On top of that this interpretation also discards the need for an "observation" in the program which would also be mathematically impossible (DeWitt und Graham, 2015, p. 111).

2.4. 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

$$\begin{aligned} \mathrm{i}^2 &= -1 \\ z &= a + b\mathrm{i} \\ \mathrm{Re}(z) &= a \\ \mathrm{Im}(z) &= b \\ \overline{z} &= a - b\mathrm{i} \\ \|z\|^2 &= a^2 + b^2 \\ e^{\theta \mathrm{i}} &= \cos(\theta) + \mathrm{i}\sin(\theta) \end{aligned}$$

i is the imaginary unit, z is the general form of a complex number where $\{a, b\} \in \mathbb{R}$, $\overline{z}$ 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 θ radians.

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

2.5. 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/`

2.6. Planck Units

By using Planck units the equations get a little bit easier. Working in Planck units means that all fundamental constants are equal to 1.

$$c = k_B = G = \hbar = 1.$$

This means that the constants will usually cancel out.

To convert to SI units we can just multiply powers of the constants such that there unit results in one of the base units.

$$l_{\text{Planck}} = l_{\text{SI}} \sqrt{\frac{G\hbar}{c^3}} \qquad 1\ \text{m}_{\text{Planck}} \approx 1.616255(18) \cdot 10^{-35}\ \text{m} \qquad \text{(CODATA, 2022a)}$$

$$m_{\text{Planck}} = m_{\text{SI}} \sqrt{\frac{c\hbar}{G}} \qquad 1\ \text{kg}_{\text{Planck}} \approx 2.176434(24) \cdot 10^{-8}\ \text{kg} \qquad \text{(CODATA, 2022b)}$$

$$t_{\text{Planck}} = t_{\text{SI}} \sqrt{\frac{G\hbar}{c^5}} \qquad 1\ \text{s}_{\text{Planck}} \approx 5.391247(60) \cdot 10^{-44}\ \text{s} \qquad \text{(CODATA, 2022c)}$$

(Gaarder Haug, 2016, Table 1)

The program will take all of its in- and outputs in Planck units.

3. Methods

3.1. Program Architecture

The program has multiple interfaces or traits as they are called in Rust that give the program some abstraction.

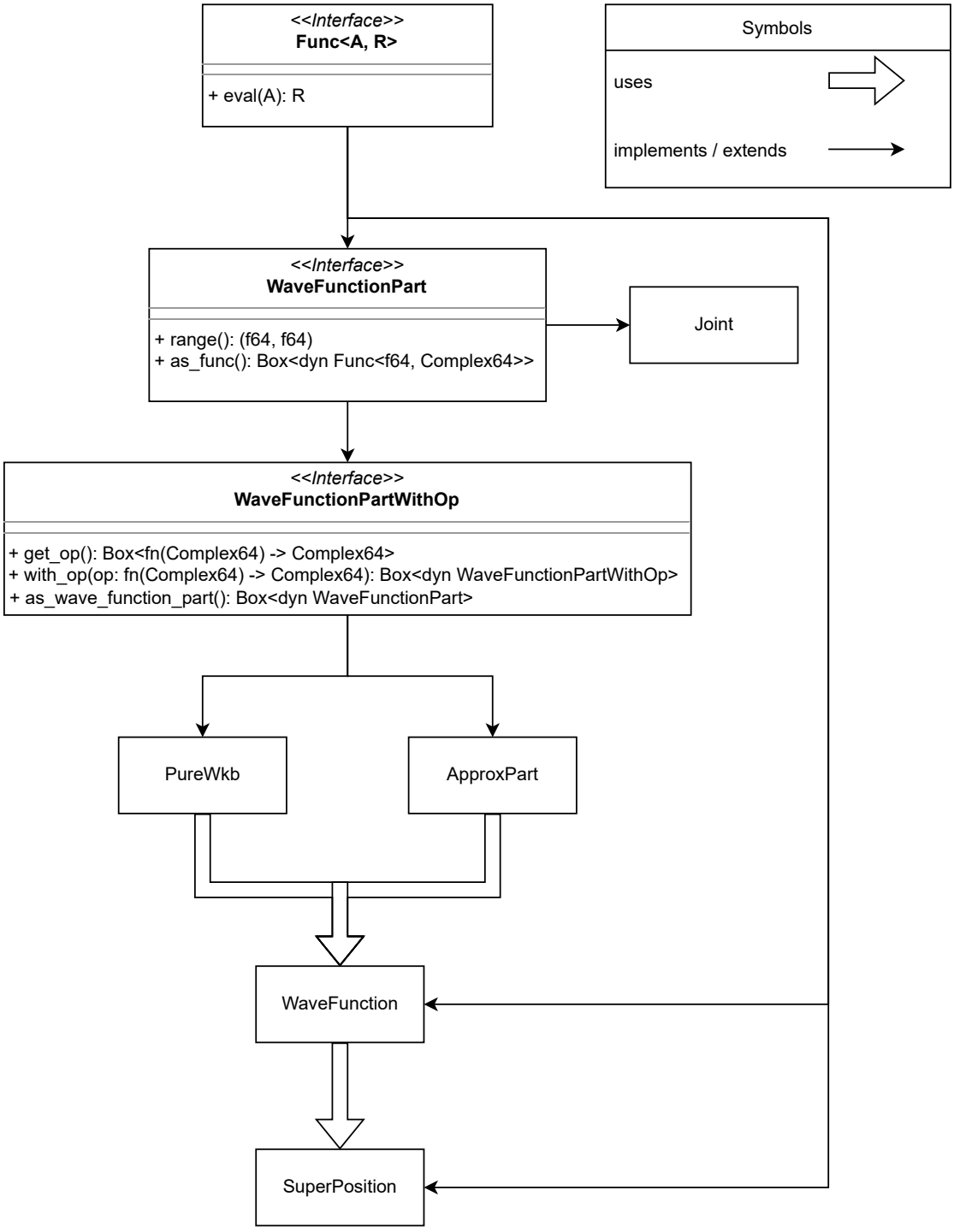


Figure 3.1.: UML diagram of program architecture

Since current version of Rust does not support manual implementations of `std::ops::Fn`

we have to define our own trait for functions `Func<A, R>` where `A` is the type of the argument and `R` is the return type. Later we will use this trait to implement functions for integration,evaluation and more useful utilities.

`WaveFunction` is at the heart of the program, it contains all the functionality to build wave functions. It is composed of `WaveFunctionPart` which represent either a `Joint`, `PureWkb` or an `ApproxPart`. With the `range` function we can check when they are valid.

3.2. 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

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

The sequence is defined as

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

Visually this looks like figure 3.2 $f(x) = (x-2)(x-1)(x+1)$.

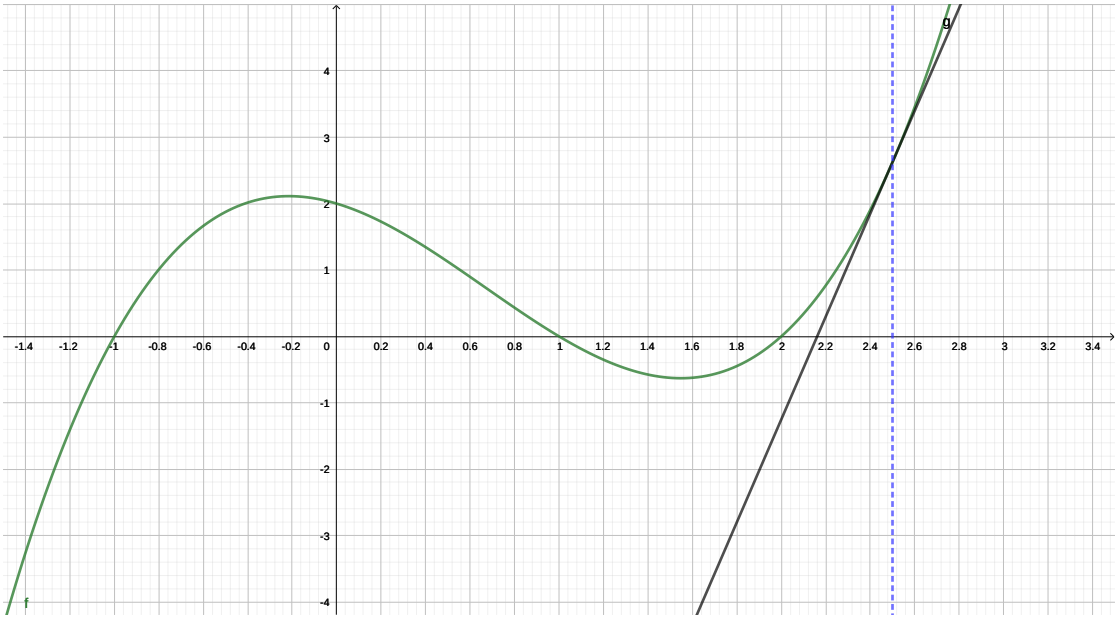


Figure 3.2.: Illustration of Newtons method, $f(x) = (x-1)(x-2)(x+1)$.

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.

```rust
pub fn newtons_method<F>(f: &F, mut guess: f64, precision: f64) -> f64
    where
        F: Fn(f64) -> f64,
{
    loop {
        let step = f(guess) / derivative(f, guess);
        if step.abs() < precision {
            return guess;
        } else {
            guess -= step;
        }
    }
}
```

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 $||cfracf(x_n)f'(x_n)|$ 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.

3.3. Regula Falsi with Bisection

Newtons method fails if the first guess is at a maximum, since the step would go to infinity. For this case we can first use a bisection search to detect a sign change. We need to do a bisection search since Regula falsi requires two guesses.

The algorithm itself is quite simple. To start we need

$$f(x) : \mathbb{R} \to \mathbb{R} \tag{3.1}$$

$$\{a \in \mathbb{R} \mid f(a) \leq 0\} \tag{3.2}$$

$$\{b \in \mathbb{R} \mid f(b) \geq 0\}. \tag{3.3}$$

Then we can draw a line between the two points $(a, f(a))$ and $(b, f(b))$. Then a becomes the x-value where the line intersects the x-axis becomes the new b, when we do the process again with the new b we will get our new value for a. We can repeat this process until we cross a fresh hold for the accuracy and the result will be the last inter section of the line with the x-axis.

3.4. Derivatives

Derivatives can be calculated numerically as in the C++ library Boost (John Maddock, 2022). The author implemented a analytical system for derivatives in Go. From that experience the benefit is negligible compared to the increase in performance and in development time since every function is a special object.

```rust
pub fn derivative<F, R>(func: &F, x: f64) -> R
where
    F: Fn(f64) -> R + ?Sized,
    R: Sub<R, Output = R> + Div<f64, Output = R> + Mul<f64, Output = R> + Add<R, Output = R>,
{
    let dx = f64::epsilon().sqrt();
    let dx1 = dx;
    let dx2 = dx1 * 2.0;
    let dx3 = dx1 * 3.0;

    let m1 = (func(x + dx1) - func(x - dx1)) / 2.0;
    let m2 = (func(x + dx2) - func(x - dx2)) / 4.0;
    let m3 = (func(x + dx3) - func(x - dx3)) / 6.0;

    let fifteen_m1 = m1 * 15.0;
    let six_m2 = m2 * 6.0;
    let ten_dx1 = dx1 * 10.0;

    return ((fifteen_m1 - six_m2) + m3) / ten_dx1;
}
```

`f64::epsilon().sqrt()` is approximately 0.000000014901161. `f64::epsilon()` is the smallest double precision floating point number where $1 + \epsilon \neq 1$. this has been chosen for dx because it should be fairly precise.

3.5. 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.

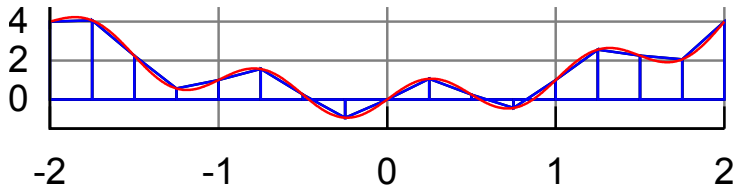


Figure 3.3.: Illustration of integration with trapeze from Wkipedia (2022).

Figure 3.3 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 wont 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 refractory. Such that functions with states, like wave functions that store parameters, can be integrated there is a trait `ReToC`. `ReToC` describes a function $f : \mathbb{R} \to \mathbb{C}$ (`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`.

```rust
pub fn trapezoidal_approx(start: &Point, end: &Point) -> Complex64 {
    return complex(end.x - start.x, 0.0) * (start.y + end.y) / complex(2.0, 0.0);
}

pub fn index_to_range(x: f64, in_min: f64, in_max: f64, out_min: f64, out_max: f64) -> f64 {
    return (x - in_min) * (out_max - out_min) / (in_max - in_min) + out_min;
}

pub fn integrate(points: Vec<Point>, batch_size: usize) -> Complex64 {
    if points.len() < 2 {
        return complex(0.0, 0.0);
    }

    let batches: Vec<&[Point]> = points.chunks(batch_size).collect();

    let parallel: Complex64 = batches
        .par_iter()
        .map(|batch| {
            let mut sum = complex(0.0, 0.0);
            for i in 0..(batch.len() - 1) {
                sum += trapezoidal_approx(&batch[i], &batch[i + 1]);
            }
            return sum;
        })
        .sum();

    let mut rest = complex(0.0, 0.0);

    for i in 0..batches.len() - 1 {
        rest += trapezoidal_approx(&batches[i][batches[i].len() - 1], &batches[i + 1][0]);
    }

    return parallel + rest;
}
```

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`

3.6. 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 3.6 we can see two Taylor series of cosine. Now we

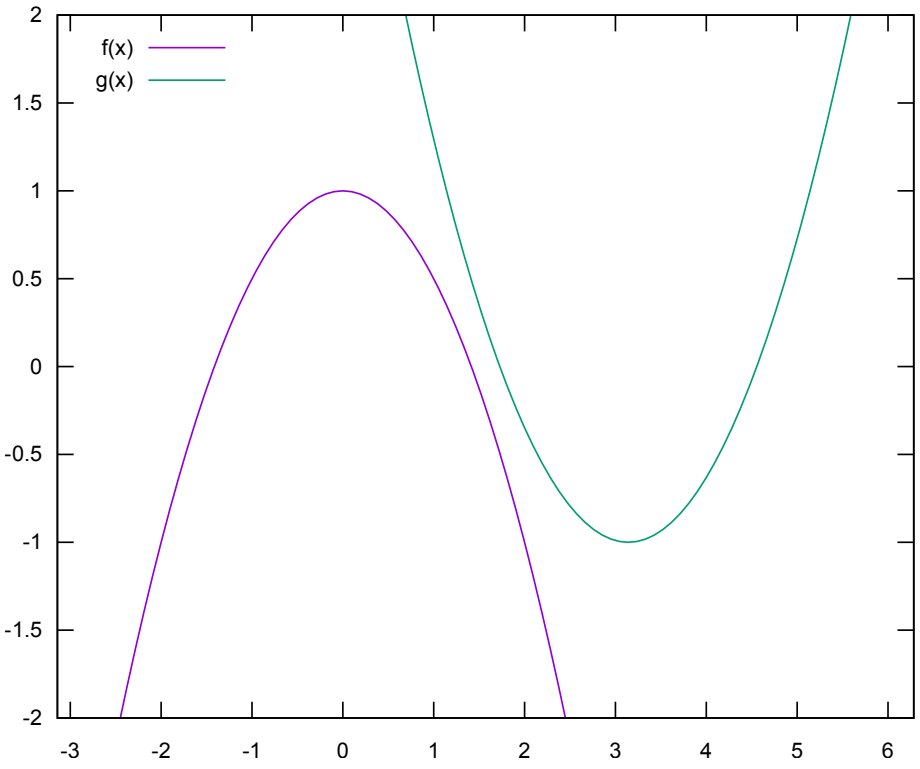


Figure 3.4.: Example for joining functions

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

$$f(x) = 1 - \frac{x^2}{2} \tag{3.4}$$

$$g(x) = \frac{(x-\pi)^2}{2} - 1 \tag{3.5}$$

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

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

This gives us 3.6 which is obviously not smooth.

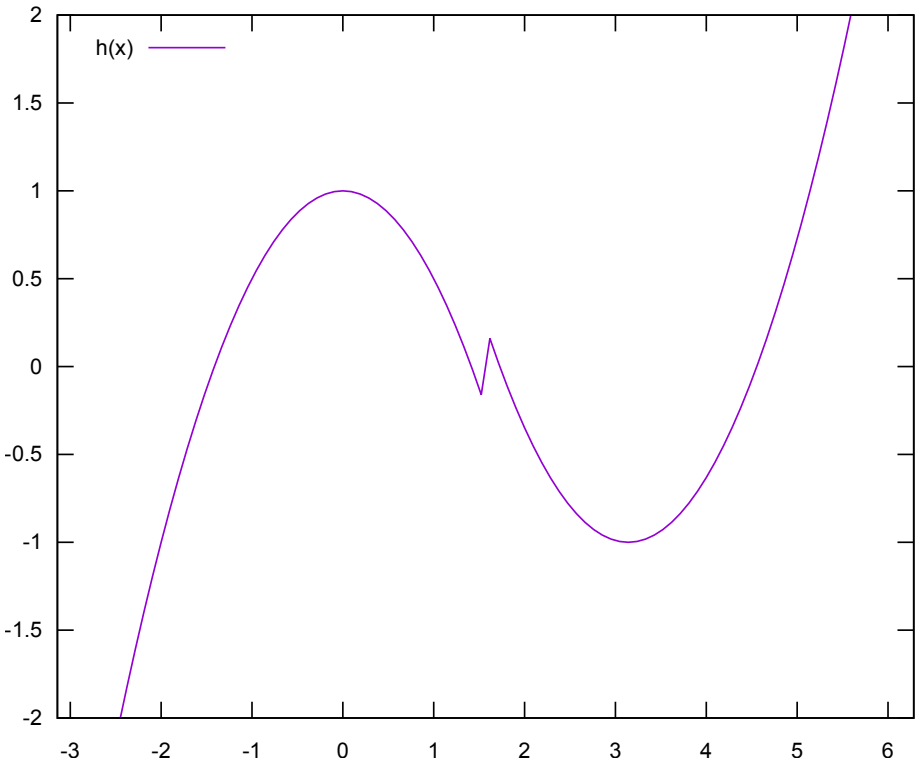


Figure 3.5.: Plot of h(x) with step joint

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

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

this results in

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

which is mathematically smooth as we can see in figure 3.6 (proof in Appendix B.1.1).

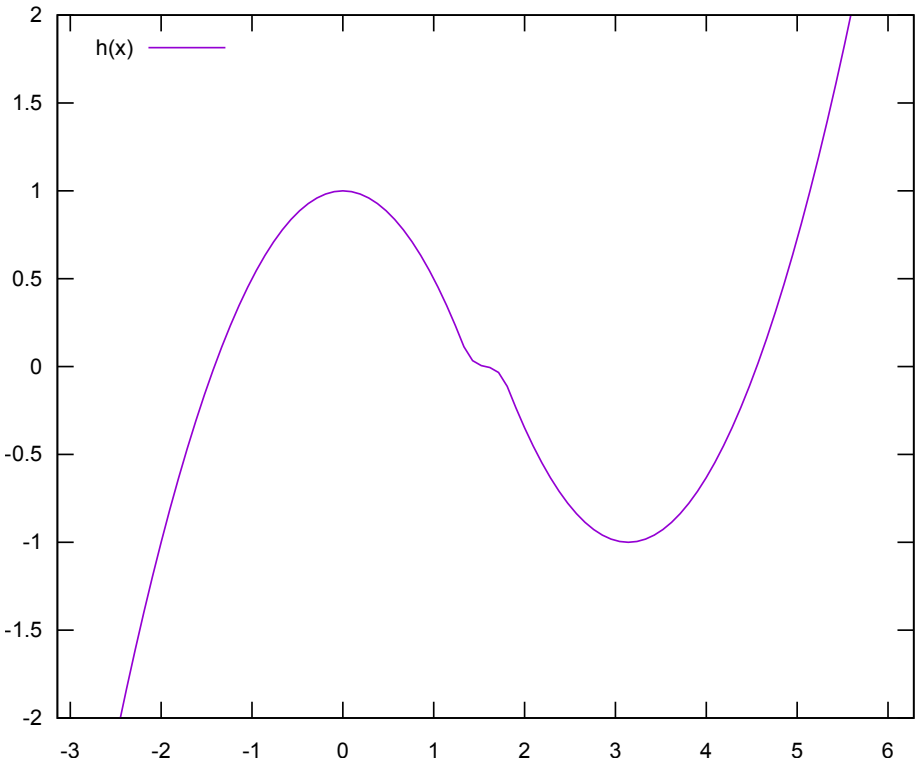


Figure 3.6.: Plot of h(x) with Hall joint

3.6.1. 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`.

```
#[derive(Clone)]
pub struct Joint {
    pub left: Arc<dyn Func<f64, Complex64>>,
    pub right: Arc<dyn Func<f64, Complex64>>,
    pub cut: f64,
    pub delta: f64,
}

impl Func<f64, Complex64> for Joint {
    fn eval(&self, x: f64) -> Complex64 {
        let chi = |x: f64| f64::sin(x * f64::consts::PI / 2.0).powi(2);
        let left_val = left.eval(x);
        return left_val + (right.eval(x) - left_val) * chi((x - self.cut) / self.delta)
    }
}
```

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.

4. Calculation

4.1. 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

$$n \in \mathbb{N}_0 \tag{4.1}$$

$$C = \{x \in \mathbb{R} \mid V(x) < E\} \tag{4.2}$$

$$\int_C \sqrt{2m(E - V(x))}dx = 2\pi(n + 1/2) \tag{4.3}$$

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

$$p(x) = \begin{cases} \sqrt{2m(E - V(x))} & V(x) < E \\ 0 & else \end{cases} \tag{4.4}$$

$$\frac{1}{2\pi}\int_{-\infty}^{\infty} p(x)dx - \frac{1}{2} \mod 1 = 0 \tag{4.5}$$

Unfortunately 4.5 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)`.

The implementation is quite strait forward we evaluate 4.5 for a number of energies and then check for discontinuities.

```
1  pub fn nth_energy<F: Fn(f64) -> f64 + Sync>(n: usize, mass: f64, pot: &F, view: (f64, f64)) -> f64 {
2      const ENERGY_STEP: f64 = 10.0;
3      const CHECKS_PER_ENERGY_STEP: usize = INTEG_STEPS;
4      let sommerfeld_cond = SommerfeldCond { mass, pot, view };
5
6      let mut energy = 0.0;
7      let mut i = 0;
8
```

```
    loop {
        let vals = evaluate_function_between(
            &sommerfeld_cond,
            energy,
            energy + ENERGY_STEP,
            CHECKS_PER_ENERGY_STEP,
        );
        let mut int_solutions = vals
            .iter()
            .zip(vals.iter().skip(1))
            .collect::<Vec<(&Point<f64, f64>, &Point<f64, f64>)>>()
            .par_iter()
            .filter(|(p1, p2)| (p1.y - p2.y).abs() > 0.5 || p1.y.signum() != p2.y.signum())
            .map(|ps| ps.1)
            .collect::<Vec<&Point<f64, f64>>>();
        int_solutions.sort_by(|p1, p2| cmp_f64(&p1.x, &p2.x));
        if i + int_solutions.len() > n {
            return int_solutions[n - i].x;
        }
        energy += ENERGY_STEP - (ENERGY_STEP / (CHECKS_PER_ENERGY_STEP as f64 + 1.0));
        i += int_solutions.len();
    }
}
```

First we check over the interval `(0.0, ENERGY_STEP)` if there are not enough zeros we check the next interval of energies and so on until we found n zeros. It's also possible that 4.5 is negative before the 0th energy there for we also have to check for normal zeros by comparing the signs of the values.

The struct `SommerfeldCond` is a `Func<f64, f64>` that evaluates 4.5.

4.1.1. Accuracy

For a benchmark we will use

$$\begin{aligned} m &= 1 \\ V(x) &= x^2 \\ (-\infty, \infty) &\approx (-200, 200). \end{aligned}$$

To get the actual values we will use Wolfram Language with WolframScript a programing language similar to Wolframalpha that can calculate the integral analytically. In Rust we can rewrite `main` to

```
fn main() {
    let output_dir = Path::new("output");

    let values = (0..=50)
        .into_iter()
        .map(|n: usize| Point::<usize, f64> {
```

```
            x: n,
            y: energy::nth_energy(n, 1.0, &potentials::square, APPROX_INF),
        })
        .collect::<Vec<Point<usize, f64>>>();

    std::env::set_current_dir(&output_dir).unwrap();
    File::create("energy.txt")
        .unwrap()
        .write_all(plot::to_gnuplot_string(values).as_bytes())
        .unwrap();
}
```

This will output all energy levels from $n = 0$ to $n = 50$. We can implement the same thing WolframScript

```
m = 1
V[x_] = x^2

nthEnergy[n_] = Module[{energys, energy},
    sommerfeldIntegral[en_] = Integrate[Sqrt[2*m*(en - V[x])],
                                        {x, -Sqrt[en], Sqrt[en]}]
    energys =  Solve[sommerfeldIntegral[en] == 2*Pi*(n + 1/2), en] // N;
    energy = en /. energys[[1]];
    energy
    ]

energys = Table[{n, N@nthEnergy[n]}, {n, 0, 50}]

csv = ExportString[energys, "CSV"]
csv = StringReplace[csv, "," -> " "]
Export["output/energies_exact.dat", csv]
```

These programs will output two files `energy.txt` (Appendix C.1) for our implementation in Rust and `energies_exact.dat` (Appendix C.1) for WolframScript. As a ruff estimate we would expect an error of $\pm\frac{10}{64000} \approx \pm 1.56 * 10^{-4}$, because the program checks for energies with that step size.

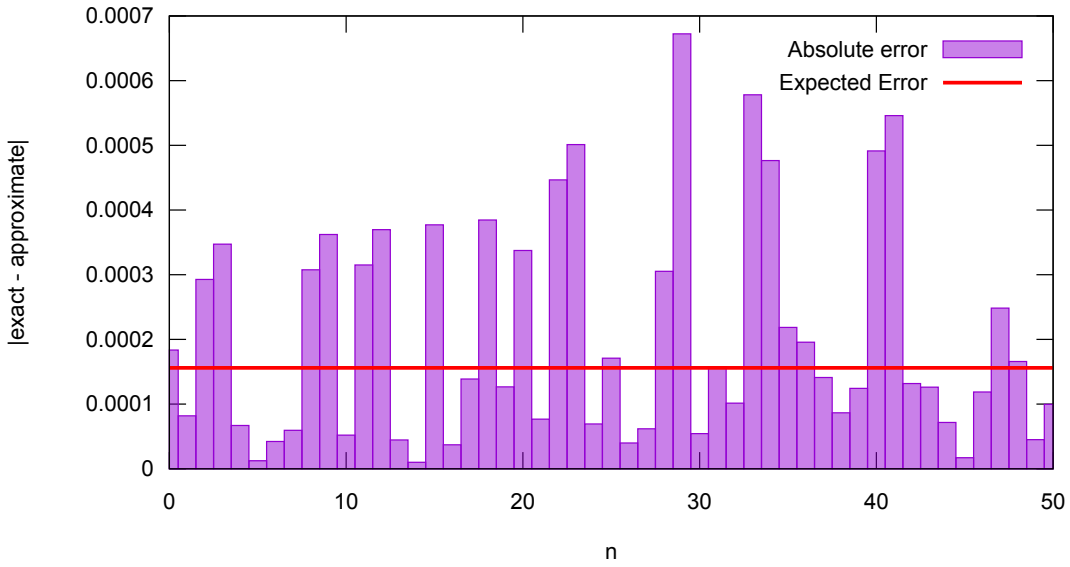


Figure 4.1.: Absolute error of energy levels in square potential

When we plot the absolute error we get figure 4.1.1. The error is a little higher than expected which is probably due to errors in the integral. Still the algorithm should be precise enough. If you'd like you could pick a lower value for `ENERGY_STEP` in `src/energy.rs:49`, but this will impact the performance for calculating energies with higher numbers for n.

4.2. 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 subproblems 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 verity 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 tree parts that can be connected to form the full solution. The tree parts are described as

$$p(x) = \sqrt{2m(|E - V(x)|)} \tag{4.6}$$

$$V(t) - E = 0 \tag{4.7}$$

$$\psi_{exp}^{WKB}(x) = e^{\delta \mathrm{i}} \frac{c_1}{2\sqrt{p(x)}} \exp\left(-\left| \int_x^t p(y)dy \right| \right) \tag{4.8}$$

$$\psi_{osz}^{WKB}(x) = \frac{c_1}{\sqrt{p(x)}} \exp\left(-\left| \int_x^t p(y)dy \right| \mathrm{i} + \delta \mathrm{i} \right) \tag{4.9}$$

$$u_1 = -2m \frac{dV}{dx}(t) \tag{4.10}$$

$$\psi^{Airy}(x) = e^{(t-x+\delta)\mathrm{i}} \frac{c_1\sqrt{\pi}}{\sqrt[6]{u_1}} \mathrm{Ai}\left(\sqrt[3]{u_1}(t-x) \right). \tag{4.11}$$

Since equation 4.7 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.

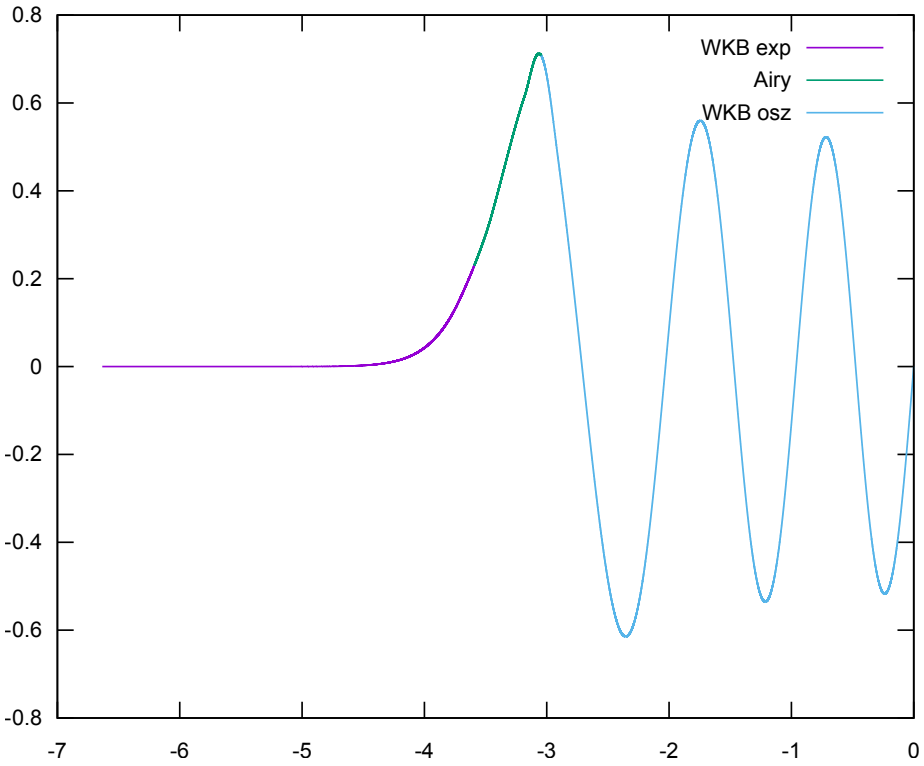


Figure 4.2.: Left half of wave function with $N_{Energy} = 5 \Rightarrow E = 11.0$, $m = 2$, $V(x) = x^2$

In figure 4.2 the three parts are visualized. The purple section on the left is the exponential decaying part $\psi_{exp}^{WKB}(x)$, equation 4.8 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 4.7. The absolute symbol makes it possible to not differentiate between the case where $x < t$ and $x > t$. Further on a factor of $e^{\delta \mathrm{i}}$ 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 4.9 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 4.2 $\delta = 0$ was used. This will be further discussed in section **??**.

4.2.1. Validity

When we look at the derivation of WKB we will see that equations 4.8 and 4.9 can only be valid if

$$p(x) = \sqrt{V(x) - E}$$
$$\left| \frac{dp}{dx}(x) \right| \ll p^2(x)$$

as Zwiebach (2018) showed in his lecture. But this would mean that WKB is only valid iff $V(x) > E$ because $p^2(x)$ would be negative otherwise. If this is the case this would imply that 4.8 can't be valid.

We will assume that this contradiction is wrong and assume that WKB is valid if

$$\left| \frac{d}{dx}(\sqrt{|V(x)-E|}) \right| < |V(x)-E|$$

4.3. Turning Points

A point x where $V(x) = E$ is called a turning point. We assume that the WKB function is a good approximation in the region where

$$-\frac{1}{2m}\frac{dV}{dx}(x) \ll (V(x)-E)^2. \tag{4.12}$$

In order to do the actual calculation we need a range were the Airy function is valid. From equation 4.12 we can infer that the Airy function is valid where

$$-\frac{1}{2m}\frac{dV}{dx}(x) - (V(x)-E)^2 > 0 \tag{4.13}$$

We can assume that the Airy function is only valid in a closed interval, this means that there must be at least two roots of equation 4.13. These roots will be called turning point boundaries from now on.

The left boundary point must have a positive and the right a negative derivative. This means we can solve for roots and group them together by there derivatives.
In order to find all roots we will use a modification of Newtons method. When we find a solution, x_0 we can divide the original function by $(x-x_0)$ this means that Newtons method wont be able to find x_0 again.

Further on since we check for roots inside the interval of `APPROX_INF` we don't have a good first guess where the turning point might be. Because of this we will make 1000 guesses evenly distributed over the interval and invent a system that can rate how good of a guess this point could be. Newtons method works well if the value of $f(x)$ is small and $f'(x)$ is neither to small nor to big. We will assume that $f'(x) = 1$ is optimal. As a rating we will use

$$\sigma(x) = \frac{|f(x)|}{-\exp\left(\left(\frac{df}{dx}(x)\right)^2 + 1\right)}$$

where lower is better. This function is just an educated guess, but it has to have some properties, as the derivative of f tends to 0, $\sigma(x)$ should diverge to infinity.

$$lim_{\frac{df}{dx}\to 0}\sigma(x) = \infty$$

If $f(x) = 0$ we found an actual root in the first guess meaning that $\sigma(x)$ should be 0. Formula 4.3 doesn't satisfy this property since it's undefined if $f'(x) = 0$ and $f(x) = 0$, but we can extend it's definition such that

$$\sigma(x) = \begin{cases} \dfrac{|f(x)|}{-\exp\left(\left(\frac{df}{dx}(x)\right)^2 + 1\right)} & f(x) \neq 0 \text{ and } \frac{df}{dx} \neq 0 \\ 0 & \text{else} \end{cases}$$

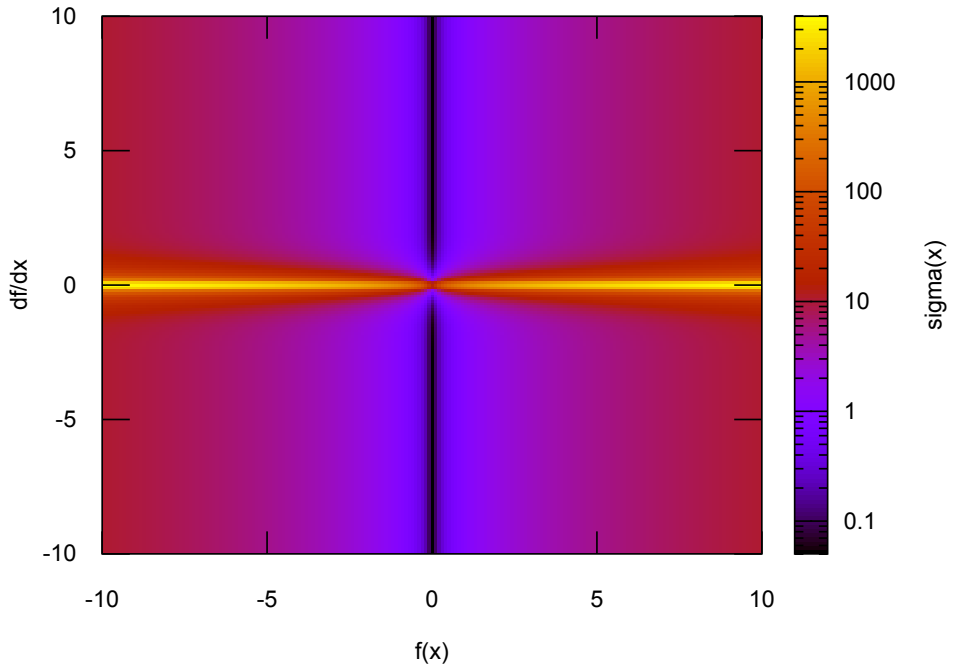


Figure 4.3.: Logarithmic heat diagram of $\sigma(x)$, darker/bluer is better

As we can see in figure 4.3 where darker/bluer values are better than yellow/red areas that $\sigma(x)$ indeed has all of the desired properties.

After we rated all of the 1000 guesses we can pick the best one as a first guess and use the modified Newtons method with it. We do this process 256 times by default. In theory we could therefor use the WKB approximation for potentials with up to 256 turning points.

```
1  fn find_zeros(phase: &Phase, view: (f64, f64)) -> Vec<f64> {
2      let phase_clone = phase.clone();
3      let validity_func = Arc::new(move |x: f64| {
```

```
        1.0 / (2.0 * phase_clone.mass).sqrt() * derivative(&|t| (phase_clone.potential)(t), x).abs()
            - ((phase_clone.potential)(x) - phase_clone.energy).pow(2)
    });
    let mut zeros = NewtonsMethodFindNewZero::new(validity_func, ACCURACY, 1e4 as usize);

    (0..MAX_TURNING_POINTS).into_iter().for_each(|_| {
        let modified_func = |x| zeros.modified_func(x);

        let guess = make_guess(&modified_func, view, 1000);
        guess.map(|g| zeros.next_zero(g));
    });

    let view = if view.0 < view.1 {
        view
    } else {
        (view.1, view.0)
    };
    let unique_zeros = zeros
        .get_previous_zeros()
        .iter()
        .filter(|x| **x > view.0 && **x < view.1)
        .map(|x| *x)
        .collect::<Vec<f64>>();
    return unique_zeros;
}
```

Here `make_guess` uses $\sigma(x)$ and returns the best guess. `NewtonsMethodFindNewZero` is the modified version of Newtons method where all the roots are stored and its implementation of `Func<f64, f64>` is just defined as

$$\frac{f(x)}{\prod_{r \in Z}(x-r)} \tag{4.14}$$

Where the set Z is the set of all the zeros that have been found previously. After the 256 iterations we filter out all the zeros that aren't in the view. Equation 4.14 is implemented in `NewtonsMethodFindNewZero`. Unfortunately this procedure can't be implement asynchronously since you have to know all previous zeros before you can find a new one.

Once we found the zeros we need to group them as previously mentioned the derivative of the validity function (4.13) must be positive if the boundary point is on the left and negative when its on the right side of the turning point. It could be the case that if the turning point is in the view that one of the boundary points is actually outside the view. For this we can use Regula falsi combined with bisection. We will do this for both the left and right most turning point if there was only one boundary found.

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

$$f : \mathbb{R} \to \mathbb{C} \tag{B.1}$$
$$g : \mathbb{R} \to \mathbb{C} \tag{B.2}$$
$$\{f, g\} \in C^1 \tag{B.3}$$
$$\{\alpha, \delta\} \in \mathbb{C} \tag{B.4}$$

define (Hall, 2013)

$$\chi(x) = \sin^2\left(\frac{\pi(x-\alpha)}{2\delta}\right) \tag{B.5}$$
$$(f \sqcup g)(x) = f(x) + (g(x) - f(x))\chi(x) \tag{B.6}$$

and proof that

$$\frac{d(f \sqcup g)}{dx}(\alpha) = \frac{df}{dx}(\alpha) \tag{B.7}$$
$$\frac{d(f \sqcup g)}{dx}(\alpha + \delta) = \frac{dg}{dx}(\alpha + \delta). \tag{B.8}$$

Calculate derivatives

$$\frac{d\chi}{dx}(x) = \frac{\pi}{2\delta}\sin\left(\frac{\pi(x-\alpha)}{\delta}\right) \tag{B.9}$$
$$\frac{d(f \sqcup g)}{dx}(x) = \frac{df}{dx}(x) + \left(\frac{dg}{dx}(x) - \frac{df}{dx}(x)\right)\chi(x) + (g(x) - f(x))\frac{d\chi}{dx}(x). \tag{B.10}$$

Note that

$$\frac{d\chi}{dx}(\alpha) = 0 \tag{B.11}$$
$$\chi(\alpha) = 0 \tag{B.12}$$
$$\frac{d\chi}{dx}(\alpha + \delta) = 0 \tag{B.13}$$
$$\chi(\alpha + \delta) = 1 \tag{B.14}$$

therefor

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

and

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

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

C. Data Files

C.1. Energies

`energy.txt`

```
0  1.4143970999546869
1  4.2427225425397275
2  7.071360490007656
3  9.89984218503414
4  12.727855127619105
5  15.55633682264559
6  18.384818517672073
7  21.213143965139928
8  24.041938165049302
9  26.870419860075785
10  29.69843279777794
11  32.52722700257012
12  35.35570869759661
13  38.18372163529877
14  41.012203335208056
15  43.84099753511743
16  46.66901047281958
17  49.49733591540462
18  52.32628637263825
19  55.15445556278185
20  57.98309351024977
21  60.811106452834736
22  63.64005690518555
23  66.46853860021204
24  69.29639528547274
25  72.1247207329406
26  74.95335868040853
27  77.78168412299357
28  80.61047832778574
29  83.43927252769512
30  86.26697296051438
31  89.09561090798232
32  91.92378010300872
```

`energies_exact.dat`

```
0  1.4142135623730951
1  4.242640687119286
2  7.0710678118654755
3  9.899494936611665
4  12.727922061357857
5  15.556349186104047
6  18.38477631085024
7  21.213203435596427
8  24.041630560342618
9  26.870057685088806
10  29.698484809834998
11  32.526911934581186
12  35.35533905932738
13  38.18376618407357
14  41.01219330881976
15  43.84062043356595
16  46.66904755831214
17  49.49747468305833
18  52.32590180780452
19  55.15432893255071
20  57.9827560572969
21  60.81118318204309
22  63.63961030678928
23  66.46803743153548
24  69.29646455628166
25  72.12489168102785
26  74.95331880577405
27  77.78174593052023
28  80.61017305526643
29  83.43860018001261
30  86.2670273047588
31  89.095454429505
32  91.92388155425118
```

33	94.75288680780098	33	94.75230867899738
34	97.58121225038602	34	97.58073580374356
35	100.40938144541242	35	100.40916292848975
36	103.23739438311458	36	103.23759005323595
37	106.06587607814106	37	106.06601717798213
38	108.89435777316754	38	108.89444430272833
39	111.72299572551829	39	111.72287142747452
40	114.55178992542766	40	114.5512985522207
41	117.38027162045414	41	117.3797256769669
42	120.2082845630391	42	120.20815280171308
43	123.0364537531827	43	123.03657992645928
44	125.86493544820918	44	125.86500705120547
45	128.69341714323565	45	128.69343417595167
46	131.52174259070352	46	131.52186130069785
47	134.35053679061292	47	134.35028842544403
48	137.17854972831506	48	137.17871555019022
49	140.0071876806658	49	140.00714267493643
50	142.83566937569228	50	142.83556979968262

Bildquellen

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

Bibliography

CODATA. CODATA Value: Planck Length. https://physics.nist.gov/cgi-bin/cuu/Value?plkl, 2022a.

CODATA. CODATA Value: Planck Mass. https://physics.nist.gov/cgi-bin/cuu/Value?plkm, 2022b.

CODATA. CODATA Value: Planck Time. https://physics.nist.gov/cgi-bin/cuu/Value?plkt, 2022c.

Bryce Seligman DeWitt und Neill Graham. *The many-worlds interpretation of quantum mechanics*, volume 63. Princeton University Press, 2015.

Espen Gaarder Haug. The gravitational constant and the Planck units. A simplification of the quantum realm. *Physics Essays*, 29(4):558–561, 2016.

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

Christopher Kormanyos John Maddock. Calculating a Derivative - 1.58.0. https://www.boost.org/doc/libs/1_58_0/libs/multiprecision/doc/html/boost_multiprecision/tut/floats/fp_ 2022.

Tanja Van Mourik, Michael Bühl, und Marie-Pierre Gaigeot. 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, 2022. URL `https://en.wikipedia.org/wiki/Numerical_integration`. [Online; accessed 10-August-2022].

Barton Zwiebach. MIT 8.06 Quantum Physics III, 2018. URL `url{https://ocw.mit.edu/courses/8-06-quantum-physics-iii-spring-2018/resources/l7-3/}`.

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