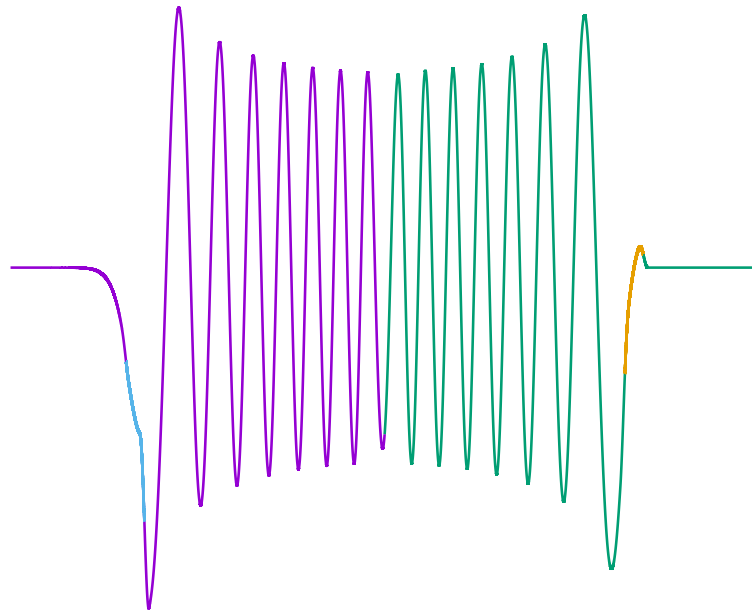# Approximating Solutions of the Time Independent Schrödinger Equation

Gian Laager
November 1, 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 $\hat{H}$.

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

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, 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

$$i^2 = -1$$
$$z = a + bi$$
$$\mathrm{Re}(z) = a$$
$$\mathrm{Im}(z) = b$$
$$\bar{z} = a - bi$$
$$\|z\|^2 = a^2 + b^2$$
$$e^{\theta i} = \cos(\theta) + i\sin(\theta)$$

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 θ 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.

2.5. 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

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

to plot the real part of the wave function, or

```
1  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.
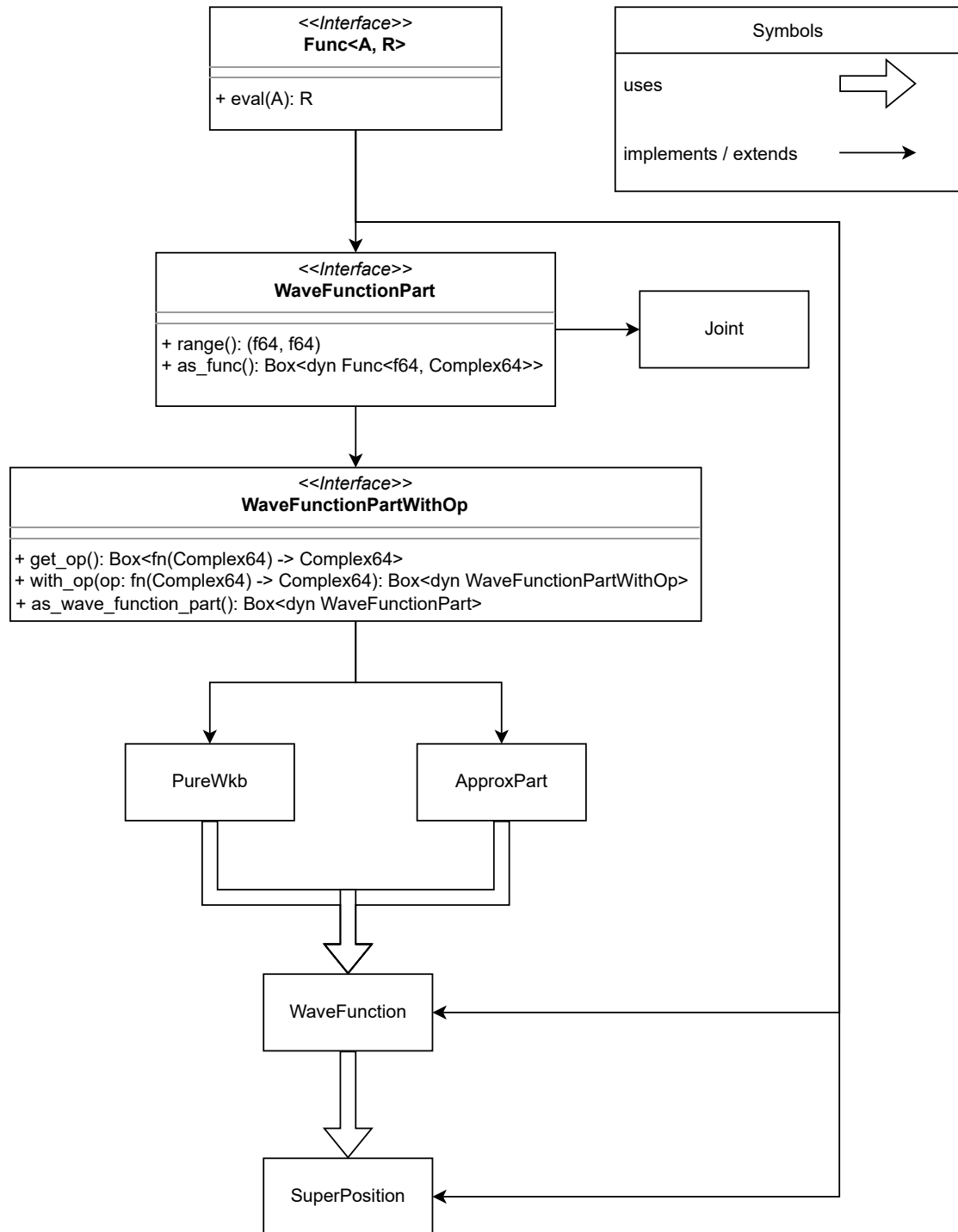
Figure 3.1.: UML diagram of program architecture

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

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

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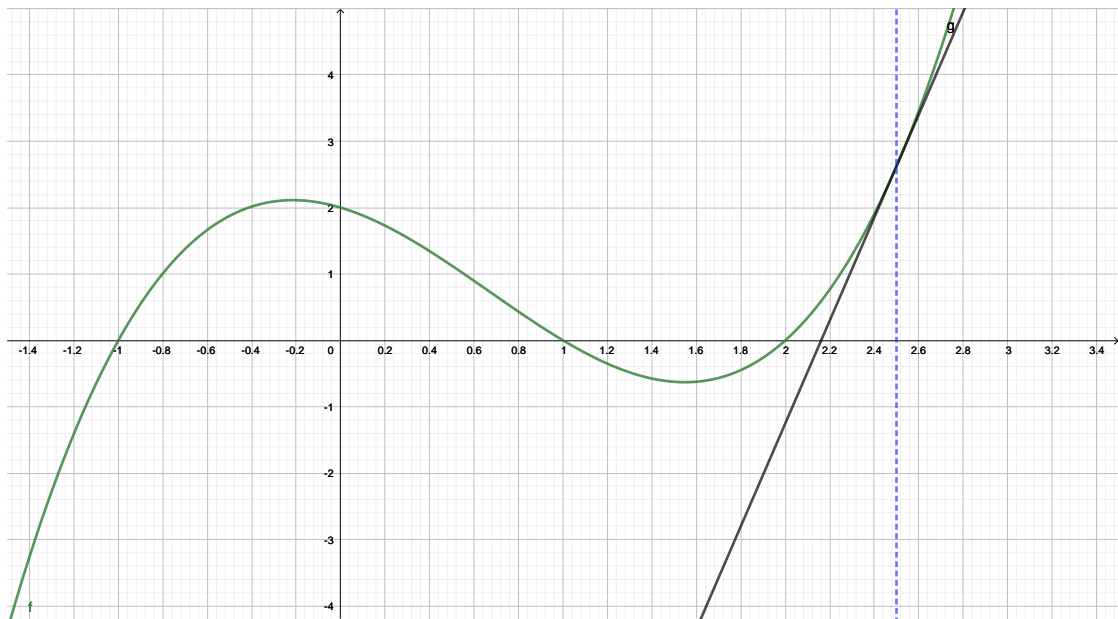

Figure 3.2.: Illustration of Newtons method, $f(x) = (x-1)(x-2)(x+1)$.

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.

```
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 $\| cfracf(x_n)f'(x_n)\|$ 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

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

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

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

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.

```
1  pub fn derivative<F, R>(func: &F, x: f64) -> R
2  where
3      F: Fn(f64) -> R + ?Sized,
4      R: Sub<R, Output = R> + Div<f64, Output = R> + Mul<f64, Output = R> + Add<R,
           Output = R>,
5  {
6      let dx = f64::epsilon().sqrt();
7      let dx1 = dx;
8      let dx2 = dx1 * 2.0;
9      let dx3 = dx1 * 3.0;
10
11     let m1 = (func(x + dx1) - func(x - dx1)) / 2.0;
12     let m2 = (func(x + dx2) - func(x - dx2)) / 4.0;
13     let m3 = (func(x + dx3) - func(x - dx3)) / 6.0;
14
15     let fifteen_m1 = m1 * 15.0;
16     let six_m2 = m2 * 6.0;
17     let ten_dx1 = dx1 * 10.0;
18
19     return ((fifteen_m1 - six_m2) + m3) / ten_dx1;
20  }
```

`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.
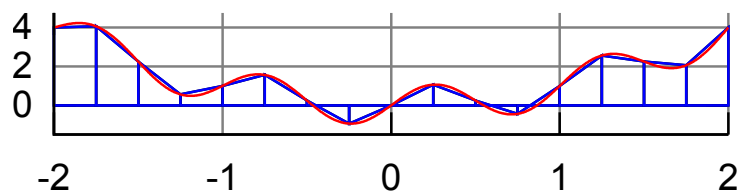

Figure 3.3.: Illustration of integration with trapeze from Wkipedia (2022).

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`.

```
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`

3.6. Transition Regions

The approximation that will be used splits $\Psi(x)$ into multiple parts that do not match perfectly together.
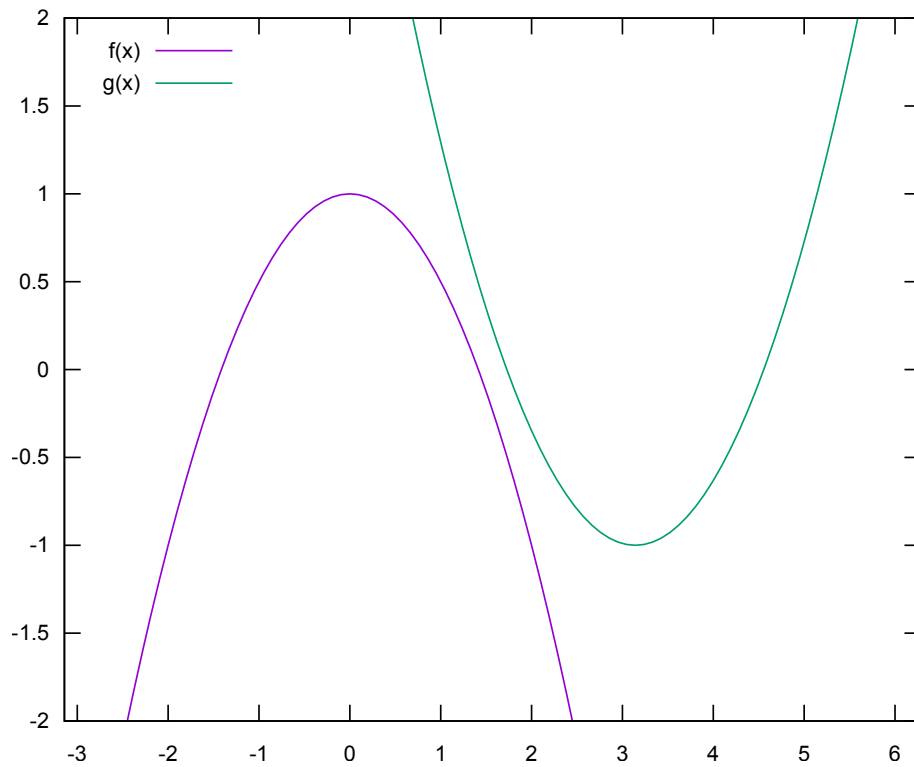
Figure 3.4.: Example for joining functions

Lets consider an example, in figure 3.6 we can see two Taylor series of cosine. Now we 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.

13
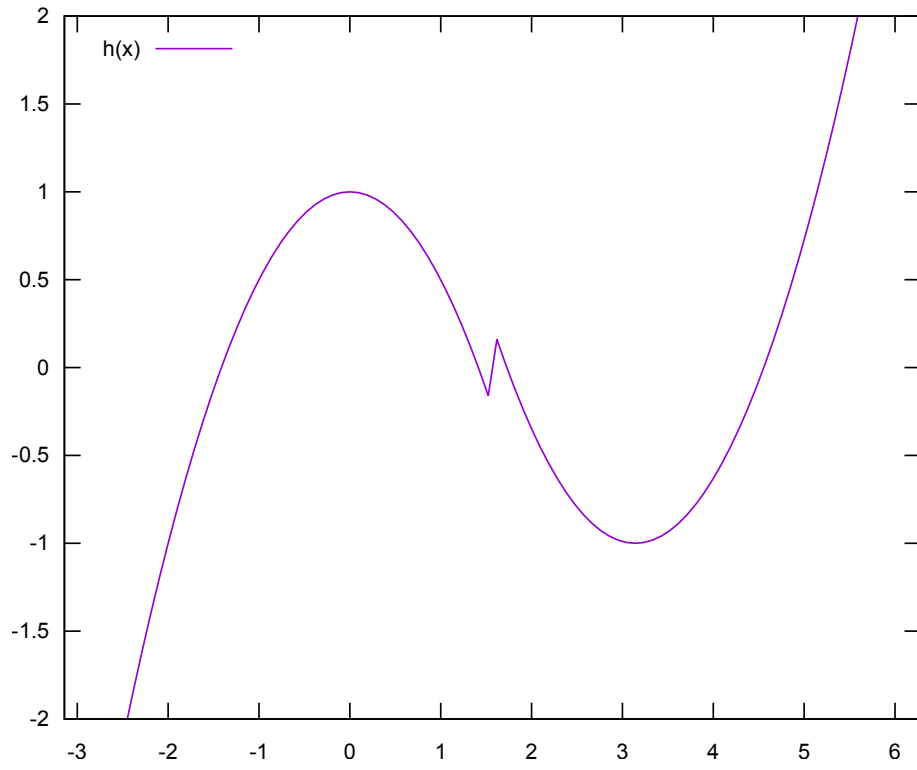
Figure 3.5.: Plot of h(x) with step joint

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 A.1.1).
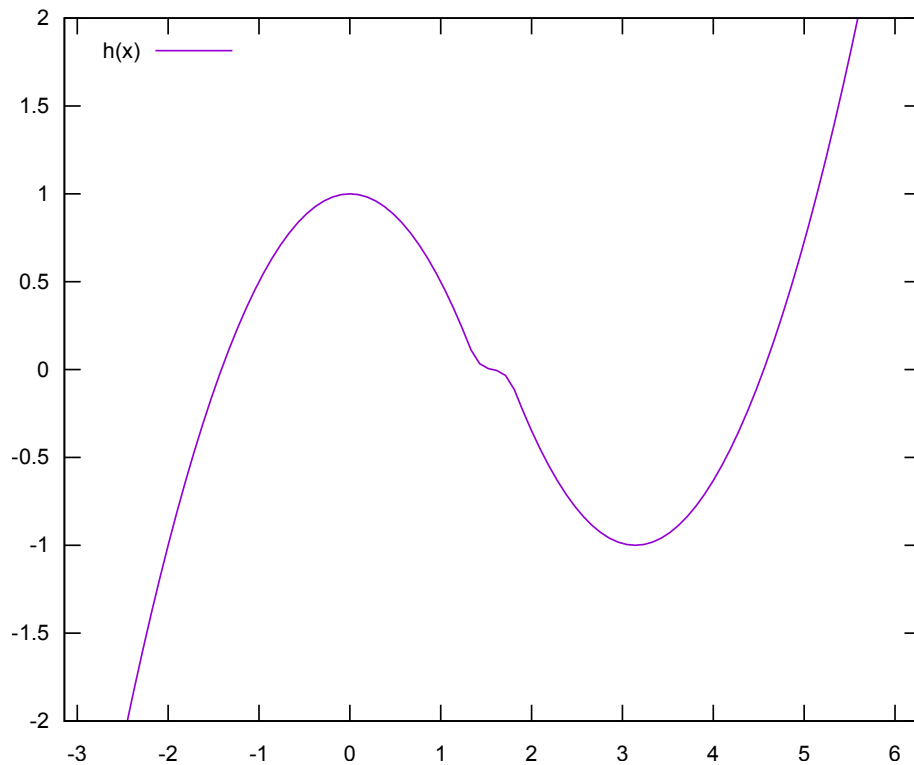
Figure 3.6.: Plot of h(x) with Hall joint

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`.

```rust
#[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)
    }
```

```
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.

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.

```rust
pub fn nth_energy<F: Fn(f64) -> f64 + Sync>(n: usize, mass: f64, pot: &F, view: (f64,
    f64)) -> f64 {
    const ENERGY_STEP: f64 = 10.0;
    const CHECKS_PER_ENERGY_STEP: usize = INTEG_STEPS;
    let sommerfeld_cond = SommerfeldCond { mass, pot, view };

    let mut energy = 0.0;
    let mut i = 0;
```

```
 8
 9     loop {
10         let vals = evaluate_function_between(
11             &sommerfeld_cond,
12             energy,
13             energy + ENERGY_STEP,
14             CHECKS_PER_ENERGY_STEP,
15         );
16         let mut int_solutions = vals
17             .iter()
18             .zip(vals.iter().skip(1))
19             .collect::<Vec<(&Point<f64, f64>, &Point<f64, f64>)>>()
20             .par_iter()
21             .filter(|(p1, p2)| (p1.y - p2.y).abs() > 0.5 || p1.y.signum() != p2.y.
                   signum())
22             .map(|ps| ps.1)
23             .collect::<Vec<&Point<f64, f64>>>();
24         int_solutions.sort_by(|p1, p2| cmp_f64(&p1.x, &p2.x));
25         if i + int_solutions.len() > n {
26             return int_solutions[n - i].x;
27         }
28         energy += ENERGY_STEP - (ENERGY_STEP / (CHECKS_PER_ENERGY_STEP as f64 + 1.0))
                   ;
29         i += int_solutions.len();
30     }
31 }
```

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

$$m = 1$$
$$V(x) = x^2$$
$$(-\infty, \infty) \approx (-200, 200).$$

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

```
1  fn main() {
2      let output_dir = Path::new("output");
3
```

```
 4      let values = (0..=50)
 5          .into_iter()
 6          .map(|n: usize| Point::<usize, f64> {
 7              x: n,
 8              y: energy::nth_energy(n, 1.0, &potentials::square, APPROX_INF),
 9          })
10          .collect::<Vec<Point<usize, f64>>>();
11
12      std::env::set_current_dir(&output_dir).unwrap();
13      File::create("energy.txt")
14          .unwrap()
15          .write_all(plot::to_gnuplot_string(values).as_bytes())
16          .unwrap();
17  }
```

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

```
 1  m = 1
 2  V[x_] = x^2
 3
 4  nthEnergy[n_] = Module[{energys, energy},
 5      sommerfeldIntegral[en_] = Integrate[Sqrt[2*m*(en - V[x])],
 6                                          {x, -Sqrt[en], Sqrt[en]}]
 7      energys =  Solve[sommerfeldIntegral[en] == 2*Pi*(n + 1/2), en] // N;
 8      energy = en /. energys[[1]];
 9      energy
10      ]
11
12  energys = Table[{n, N@nthEnergy[n]}, {n, 0, 50}]
13
14  csv = ExportString[energys, "CSV"]
15  csv = StringReplace[csv, "," -> " "]
16  Export["output/energies_exact.dat", csv]
```

These programs will output two files energy.txt (Appendix B.1) for our implementation in Rust and energies_exact.dat (Appendix B.1) for WolframScript. As a ruff estimate we would expect an error of $\pm\dfrac{10}{64000} \approx \pm 1.56 * 10^{-4}$, because the program checks for energies with that step size.
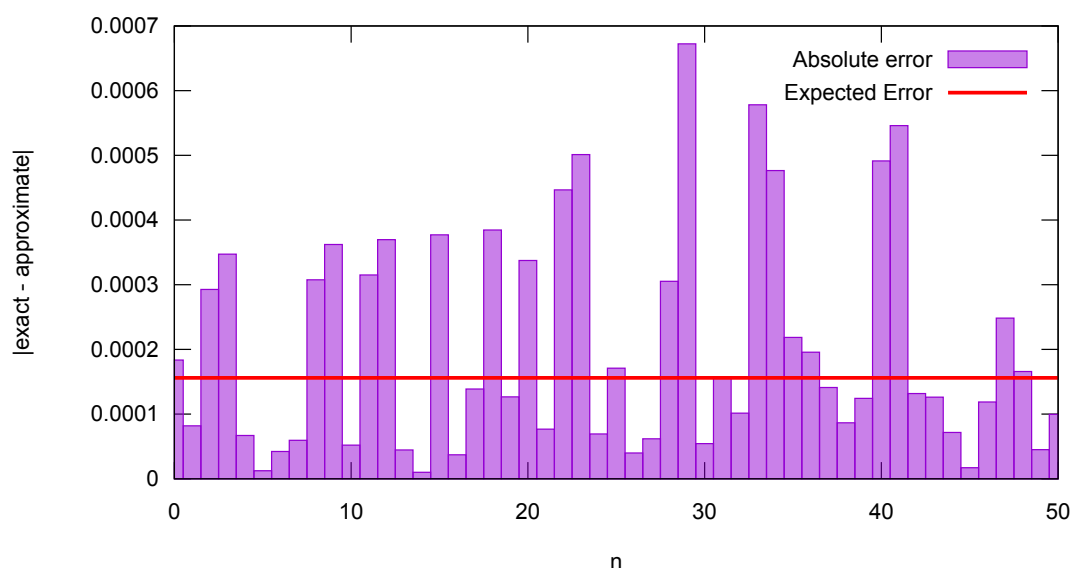
Figure 4.1.: Absolute error of energy levels in square potential

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 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 ineffi-cient 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

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

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

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

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

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

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

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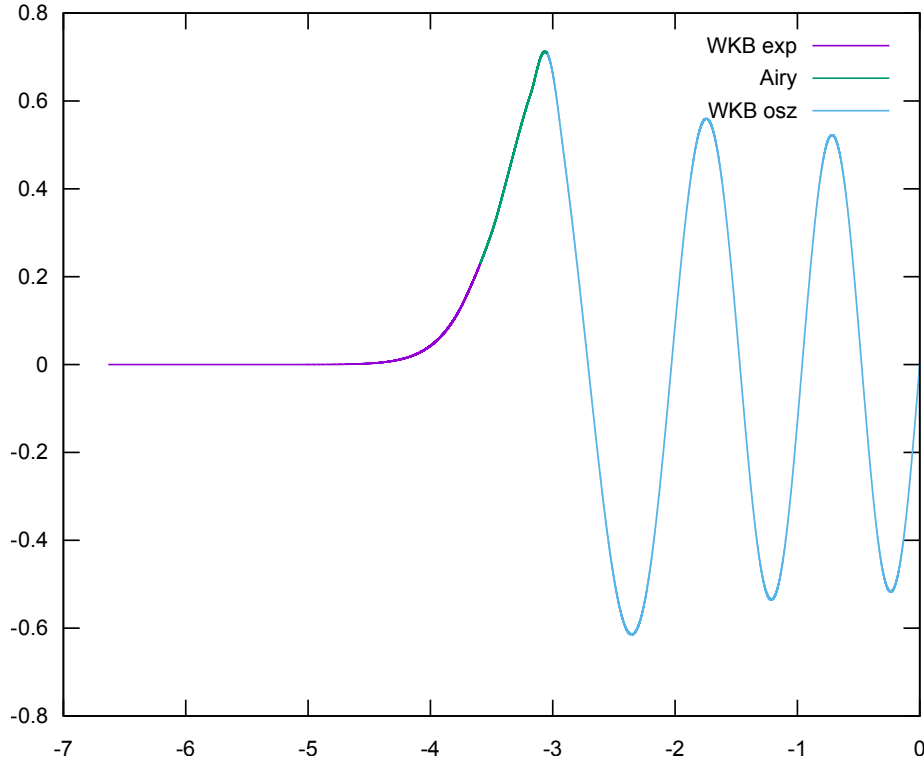
Figure 4.2.: Left half of wave function with $N_{Energy} = 5 \Rightarrow E = 11.0$, $m = 2$, $V(x) = x^2$

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 $e^{\delta i}$ 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

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

4.2.2. Implementation

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

$$\sigma(x) = \begin{cases} \dfrac{|f(x)|}{-\exp\left(\left(\frac{df}{dx}(x)\right)^2 + 1\right)} & f(x) \neq 0 \text{ and } \frac{df}{dx} \neq 0 \\ 0 & \text{else} \end{cases}$$
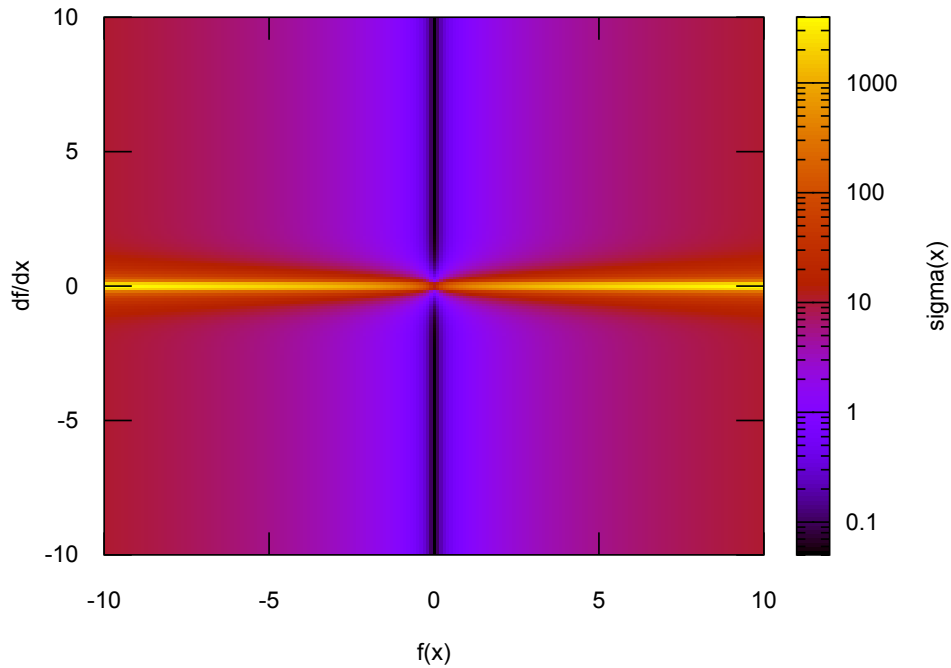
Figure 4.3.: Logarithmic heat diagram of $\sigma(x)$, darker/bluer is better

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| {
```

```
4          1.0 / (2.0 * phase_clone.mass).sqrt() * derivative(&|t| (phase_clone.
               potential)(t), x).abs()
5               - ((phase_clone.potential)(x) - phase_clone.energy).pow(2)
6      });
7      let mut zeros = NewtonsMethodFindNewZero::new(validity_func, ACCURACY, 1e4 as
           usize);
8
9      (0..MAX_TURNING_POINTS).into_iter().for_each(|_| {
10         let modified_func = |x| zeros.modified_func(x);
11
12         let guess = make_guess(&modified_func, view, 1000);
13         guess.map(|g| zeros.next_zero(g));
14     });
15
16     let view = if view.0 < view.1 {
17         view
18     } else {
19         (view.1, view.0)
20     };
21     let unique_zeros = zeros
22         .get_previous_zeros()
23         .iter()
24         .filter(|x| **x > view.0 && **x < view.1)
25         .map(|x| *x)
26         .collect::<Vec<f64>>();
27     return unique_zeros;
28 }
```

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.

4.4. Wave Function

To combine all of the different wave function parts we will create the `WaveFunction` struct. First lets define the `WaveFunctionPart` trait.

```
1  pub trait WaveFunctionPart: Func<f64, Complex64> + Sync + Send {
2      fn range(&self) -> (f64, f64);
3      fn as_func(&self) -> Box<dyn Func<f64, Complex64>>;
4  }
```

All that we need is a range in which the function is valid. Also (Hall, 2013, p. 317) states that the oscillating parts of WKB might have to negated, because of this we will also introduce the `WaveFunctionPartWithOp` trait. In addition to the wave function itself it will also store and operation as negate for example.

```
1  pub trait WaveFunctionPartWithOp: WaveFunctionPart {
2      fn get_op(&self) -> Box<fn(Complex64) -> Complex64>;
3      fn with_op(&self, op: fn(Complex64) -> Complex64) -> Box<dyn
          WaveFunctionPartWithOp>;
4      fn as_wave_function_part(&self) -> Box<dyn WaveFunctionPart>;
5  }
```

The two functions `as_wave_function_part` and `as_func` are to emulate trait up casting because in the current version of Rust this is still an experimental feature.

`ApproxPart` will be the implementation of `WaveFunctionPartWithOp` that handles a wave function part that contains a `AiryWaveFunction` and a `WkbWaveFunction`. As suggested by Hall (2013) we will add joints between the WKB and Airy functions.

`PureWkb` can be used if there are no turning points. It just stores a `WkbWaveFunction`. This version is just for completeness since it might be very inaccurate to only use a WKB function. This functionallity was just introduced for completeness.

To construct a new `WaveFunction` we will first calculate the energy with `energy::nth_energy`. After this we can calculate the new view by running Newtons method on $V(x) - E$ to find the outer most turning points, then the view will be defined by `t * view_factor` where `t` is the turning point.

Once we've calculated the view, we can calculate all the turning points with the `AiryWaveFunction` constructor that will also give us all the Airy parts we will need later. If there were no turning points found in the view we can construct two `WkbWaveFunction` and connect them in the middle of the view. Because the sign of the wave function might not match have to find a the best operation from

```
1  pub fn identity(c: Complex64) -> Complex64 {
2      c
3  }
4
5  pub fn conjugate(c: Complex64) -> Complex64 {
6      c.conj()
7  }
8
```

```
 9  pub fn negative(c: Complex64) -> Complex64 {
10      -c
11  }
12
13  pub fn negative_conj(c: Complex64) -> Complex64 {
14      -c.conj()
15  }
```

This will be done in the `find_best_op` function where we choose the operation that has the minimal error both with respect to the derivative and the value of the wave function. Mathematically we choose the operation $o(x)$ for which

$$|o(\psi_0(x)) - \psi_1(x)|^2 + \left| o\left(\frac{d}{dx}\psi_0(x)\right) - \frac{d}{dx}\psi_1(x) \right|^2$$

is minimal, $\psi_0(x)$ is the wave function to the left and $\psi_1(x)$ is the wave function to the right of the middle of the view.

A. Detailed Calculations

A.1. Proofs

A.1.1. Smoothness of Transitionfunction

Given that

$$f : \mathbb{R} \to \mathbb{C} \tag{A.1}$$

$$g : \mathbb{R} \to \mathbb{C} \tag{A.2}$$

$$\{f, g\} \in C^1 \tag{A.3}$$

$$\{\alpha, \delta\} \in \mathbb{C} \tag{A.4}$$

define (Hall, 2013)

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

$$(f \sqcup g)(x) = f(x) + (g(x) - f(x))\chi(x) \tag{A.6}$$

and proof that

$$\frac{d(f \sqcup g)}{dx}(\alpha) = \frac{df}{dx}(\alpha) \tag{A.7}$$

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

Calculate derivatives

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

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

Note that

$$\frac{d\chi}{dx}(\alpha) = 0 \tag{A.11}$$

$$\chi(\alpha) = 0 \tag{A.12}$$

$$\frac{d\chi}{dx}(\alpha + \delta) = 0 \tag{A.13}$$

$$\chi(\alpha + \delta) = 1 \tag{A.14}$$

therefor

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

and

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

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

B. Data Files

B.1. Energies

energy.txt

```
1    0 1.4143970999546869
2    1 4.2427225425397275
3    2 7.071360490007656
4    3 9.89984218503414
5    4 12.727855127619105
6    5 15.55633682264559
7    6 18.384818517672073
8    7 21.213143965139928
9    8 24.041938165049302
10   9 26.870419860075785
11   10 29.69843279777794
12   11 32.52722700257012
13   12 35.35570869759661
14   13 38.18372163529877
15   14 41.012203335208056
16   15 43.84099753511743
17   16 46.66901047281958
18   17 49.49733591540462
19   18 52.32628637263825
20   19 55.15445556278185
21   20 57.98309351024977
22   21 60.811106452834736
23   22 63.64005690518555
24   23 66.46853860021204
25   24 69.29639528547274
26   25 72.1247207329406
27   26 74.95335868040853
28   27 77.78168412299357
29   28 80.61047832778574
30   29 83.43927252769512
31   30 86.26697296051438
32   31 89.09561090798232
33   32 91.92378010300872
34   33 94.75288680780098
35   34 97.58121225038602
36   35 100.40938144541242
37   36 103.23739438311458
38   37 106.06587607814106
```

energies_exact.dat

```
1    0 1.4142135623730951
2    1 4.242640687119286
3    2 7.0710678118654755
4    3 9.899494936611665
5    4 12.727922061357857
6    5 15.556349186104047
7    6 18.38477631085024
8    7 21.213203435596427
9    8 24.041630560342618
10   9 26.870057685088806
11   10 29.698484809834998
12   11 32.526911934581186
13   12 35.35533905932738
14   13 38.18376618407357
15   14 41.01219330881976
16   15 43.84062043356595
17   16 46.66904755831214
18   17 49.49747468305833
19   18 52.32590180780452
20   19 55.15432893255071
21   20 57.9827560572969
22   21 60.81118318204309
23   22 63.63961030678928
24   23 66.46803743153548
25   24 69.29646455628166
26   25 72.12489168102785
27   26 74.95331880577405
28   27 77.78174593052023
29   28 80.61017305526643
30   29 83.43860018001261
31   30 86.2670273047588
32   31 89.095454429505
33   32 91.92388155425118
34   33 94.75230867899738
35   34 97.58073580374356
36   35 100.40916292848975
37   36 103.23759005323595
38   37 106.06601717798213
```

39	38	108.89435777316754		39	38	108.89444430272833
40	39	111.72299572551829		40	39	111.72287142747452
41	40	114.55178992542766		41	40	114.5512985522207
42	41	117.38027162045414		42	41	117.3797256769669
43	42	120.2082845630391		43	42	120.20815280171308
44	43	123.0364537531827		44	43	123.03657992645928
45	44	125.86493544820918		45	44	125.86500705120547
46	45	128.69341714323565		46	45	128.69343417595167
47	46	131.52174259070352		47	46	131.52186130069785
48	47	134.35053679061292		48	47	134.35028842544403
49	48	137.17854972831506		49	48	137.17871555019022
50	49	140.0071876806658		50	49	140.00714267493643
51	50	142.83566937569228		51	50	142.83556979968262

C. Source Code

The source code is also available on the authors GitHub
`https://github.com/Gian-Laager/Schroedinger-Approximation`

src/main.rs

```rust
 1  mod airy;
 2  mod airy_wave_func;
 3  mod energy;
 4  mod integrals;
 5  mod newtons_method;
 6  mod plot;
 7  mod potentials;
 8  mod tui;
 9  mod turning_points;
10  mod utils;
11  mod wave_function_builder;
12  mod wkb_wave_func;
13
14  use crate::airy::airy_ai;
15  use crate::airy_wave_func::AiryWaveFunction;
16  use crate::integrals::*;
17  use crate::newtons_method::derivative;
18  use crate::utils::Func;
19  use crate::utils::*;
20  use crate::wave_function_builder::*;
21  use crate::wkb_wave_func::WkbWaveFunction;
22  use num::complex::Complex64;
23  use num::pow::Pow;
24  use rayon::iter::*;
25  use std::collections::HashMap;
26  use std::f64;
27  use std::fs::File;
28  use std::io::Write;
29  use std::path::Path;
30  use std::sync::Arc;
31
32  const INTEG_STEPS: usize = 64000;
33  const TRAPEZE_PER_THREAD: usize = 1000;
34  const NUMBER_OF_POINTS: usize = 100000;
35
36  const AIRY_EXTRA: f64 = 0.5;
```

```rust
37  const WKB_TRANSITION_FRACTION: f64 = 0.05;
38
39  const ENABLE_WKB_JOINTS: bool = false;
40  const ENABLE_AIRY_JOINTS: bool = true;
41
42  const COMPLEX_AIRY: bool = false;
43  const COMPLEX_EXP_WKB: bool = false;
44  const COMPLEX_OSZ_WKB: bool = false;
45
46  const APPROX_INF: (f64, f64) = (-200.0, 200.0);
47
48  fn main() {
49      // let wave_function = wave_function_builder::SuperPosition::new(
50      //     &potentials::mexican_hat,
51      //     1.0,
52      //     &[
53      //         (5, 1.0.into()),
54      //         (12, 1.0.into()),
55      //         (40, 1.0.into()),
56      //     ],
57      //     APPROX_INF,
58      //     1.5,
59      //     ScalingType::Renormalize(complex(1.0, 0.0)),
60      // );
61
62      let wave_function = wave_function_builder::WaveFunction::new(
63          &potentials::square,
64          1.0,
65          5,
66          APPROX_INF,
67          1.5,
68          ScalingType::Renormalize(1.0.into()),
69      );
70
71      let output_dir = Path::new("output");
72      plot::plot_wavefunction_parts(&wave_function, output_dir, "data.txt");
73  }
```

src/airy.rs

```rust
1  /* automatically generated by rust-bindgen 0.59.2 */
2
3  #[derive(PartialEq, Copy, Clone, Hash, Debug, Default)]
4  #[repr(C)]
5  pub struct __BindgenComplex<T> {
6      pub re: T,
7      pub im: T,
8  }
9  pub type size_t = ::std::os::raw::c_ulong;
```

```rust
10  pub type wchar_t = ::std::os::raw::c_int;
11  #[repr(C)]
12  #[repr(align(16))]
13  #[derive(Debug, Copy, Clone)]
14  pub struct max_align_t {
15      pub __clang_max_align_nonce1: ::std::os::raw::c_longlong,
16      pub __bindgen_padding_0: u64,
17      pub __clang_max_align_nonce2: u128,
18  }
19  #[test]
20  fn bindgen_test_layout_max_align_t() {
21      assert_eq!(
22          ::std::mem::size_of::<max_align_t>(),
23          32usize,
24          concat!("Size of: ", stringify!(max_align_t))
25      );
26      assert_eq!(
27          ::std::mem::align_of::<max_align_t>(),
28          16usize,
29          concat!("Alignment of ", stringify!(max_align_t))
30      );
31      assert_eq!(
32          unsafe {
33              &(*(::std::ptr::null::<max_align_t>())).__clang_max_align_nonce1 as *
34                  const _ as usize
35          },
35          0usize,
36          concat!(
37              "Offset of field: ",
38              stringify!(max_align_t),
39              "::",
40              stringify!(__clang_max_align_nonce1)
41          )
42      );
43      assert_eq!(
44          unsafe {
45              &(*(::std::ptr::null::<max_align_t>())).__clang_max_align_nonce2 as *
46                  const _ as usize
46          },
47          16usize,
48          concat!(
49              "Offset of field: ",
50              stringify!(max_align_t),
51              "::",
52              stringify!(__clang_max_align_nonce2)
53          )
54      );
55  }
56  #[repr(C)]
```

```rust
#[derive(Debug, Copy, Clone)]
pub struct _GoString_ {
    pub p: *const ::std::os::raw::c_char,
    pub n: isize,
}
#[test]
fn bindgen_test_layout__GoString_() {
    assert_eq!(
        ::std::mem::size_of::<_GoString_>(),
        16usize,
        concat!("Size of: ", stringify!(_GoString_))
    );
    assert_eq!(
        ::std::mem::align_of::<_GoString_>(),
        8usize,
        concat!("Alignment of ", stringify!(_GoString_))
    );
    assert_eq!(
        unsafe { &(*(::std::ptr::null::<_GoString_>())).p as *const _ as usize },
        0usize,
        concat!(
            "Offset of field: ",
            stringify!(_GoString_),
            "::",
            stringify!(p)
        )
    );
    assert_eq!(
        unsafe { &(*(::std::ptr::null::<_GoString_>())).n as *const _ as usize },
        8usize,
        concat!(
            "Offset of field: ",
            stringify!(_GoString_),
            "::",
            stringify!(n)
        )
    );
}
pub type GoInt8 = ::std::os::raw::c_schar;
pub type GoUint8 = ::std::os::raw::c_uchar;
pub type GoInt16 = ::std::os::raw::c_short;
pub type GoUint16 = ::std::os::raw::c_ushort;
pub type GoInt32 = ::std::os::raw::c_int;
pub type GoUint32 = ::std::os::raw::c_uint;
pub type GoInt64 = ::std::os::raw::c_longlong;
pub type GoUint64 = ::std::os::raw::c_ulonglong;
pub type GoInt = GoInt64;
pub type GoUint = GoUint64;
pub type GoUintptr = ::std::os::raw::c_ulong;
```

```rust
106  pub type GoFloat32 = f32;
107  pub type GoFloat64 = f64;
108  pub type GoComplex64 = __BindgenComplex<f32>;
109  pub type GoComplex128 = __BindgenComplex<f64>;
110  pub type _check_for_64_bit_pointer_matching_GoInt = [::std::os::raw::c_char; 1usize];
111  pub type GoString = _GoString_;
112  pub type GoMap = *mut ::std::os::raw::c_void;
113  pub type GoChan = *mut ::std::os::raw::c_void;
114  #[repr(C)]
115  #[derive(Debug, Copy, Clone)]
116  pub struct GoInterface {
117      pub t: *mut ::std::os::raw::c_void,
118      pub v: *mut ::std::os::raw::c_void,
119  }
120  #[test]
121  fn bindgen_test_layout_GoInterface() {
122      assert_eq!(
123          ::std::mem::size_of::<GoInterface>(),
124          16usize,
125          concat!("Size of: ", stringify!(GoInterface))
126      );
127      assert_eq!(
128          ::std::mem::align_of::<GoInterface>(),
129          8usize,
130          concat!("Alignment of ", stringify!(GoInterface))
131      );
132      assert_eq!(
133          unsafe { &(*(::std::ptr::null::<GoInterface>())).t as *const _ as usize },
134          0usize,
135          concat!(
136              "Offset of field: ",
137              stringify!(GoInterface),
138              "::",
139              stringify!(t)
140          )
141      );
142      assert_eq!(
143          unsafe { &(*(::std::ptr::null::<GoInterface>())).v as *const _ as usize },
144          8usize,
145          concat!(
146              "Offset of field: ",
147              stringify!(GoInterface),
148              "::",
149              stringify!(v)
150          )
151      );
152  }
153  #[repr(C)]
154  #[derive(Debug, Copy, Clone)]
```

```rust
155  pub struct GoSlice {
156      pub data: *mut ::std::os::raw::c_void,
157      pub len: GoInt,
158      pub cap: GoInt,
159  }
160  #[test]
161  fn bindgen_test_layout_GoSlice() {
162      assert_eq!(
163          ::std::mem::size_of::<GoSlice>(),
164          24usize,
165          concat!("Size of: ", stringify!(GoSlice))
166      );
167      assert_eq!(
168          ::std::mem::align_of::<GoSlice>(),
169          8usize,
170          concat!("Alignment of ", stringify!(GoSlice))
171      );
172      assert_eq!(
173          unsafe { &(*(::std::ptr::null::<GoSlice>())).data as *const _ as usize },
174          0usize,
175          concat!(
176              "Offset of field: ",
177              stringify!(GoSlice),
178              "::",
179              stringify!(data)
180          )
181      );
182      assert_eq!(
183          unsafe { &(*(::std::ptr::null::<GoSlice>())).len as *const _ as usize },
184          8usize,
185          concat!(
186              "Offset of field: ",
187              stringify!(GoSlice),
188              "::",
189              stringify!(len)
190          )
191      );
192      assert_eq!(
193          unsafe { &(*(::std::ptr::null::<GoSlice>())).cap as *const _ as usize },
194          16usize,
195          concat!(
196              "Offset of field: ",
197              stringify!(GoSlice),
198              "::",
199              stringify!(cap)
200          )
201      );
202  }
203  #[repr(C)]
```

```
204  #[derive(Debug, Copy, Clone)]
205  pub struct airy_ai_return {
206      pub r0: GoFloat64,
207      pub r1: GoFloat64,
208  }
209  #[test]
210  fn bindgen_test_layout_airy_ai_return() {
211      assert_eq!(
212          ::std::mem::size_of::<airy_ai_return>(),
213          16usize,
214          concat!("Size of: ", stringify!(airy_ai_return))
215      );
216      assert_eq!(
217          ::std::mem::align_of::<airy_ai_return>(),
218          8usize,
219          concat!("Alignment of ", stringify!(airy_ai_return))
220      );
221      assert_eq!(
222          unsafe { &(*(::std::ptr::null::<airy_ai_return>())).r0 as *const _ as usize
                  },
223          0usize,
224          concat!(
225              "Offset of field: ",
226              stringify!(airy_ai_return),
227              "::",
228              stringify!(r0)
229          )
230      );
231      assert_eq!(
232          unsafe { &(*(::std::ptr::null::<airy_ai_return>())).r1 as *const _ as usize
                  },
233          8usize,
234          concat!(
235              "Offset of field: ",
236              stringify!(airy_ai_return),
237              "::",
238              stringify!(r1)
239          )
240      );
241  }
242  extern "C" {
243      pub fn airy_ai(zr: GoFloat64, zi: GoFloat64) -> airy_ai_return;
244  }
```

src/airy_wave_func.rs

```
1  use crate::newtons_method::newtons_method;
2  use crate::newtons_method::*;
3  use crate::turning_points::*;
```

```rust
use crate::wkb_wave_func::Phase;
use crate::*;
use num::signum;
use std::sync::Arc;

fn Ai(x: Complex64) -> Complex64 {
    let go_return;
    unsafe {
        go_return = airy_ai(x.re, x.im);
    }
    return complex(go_return.r0, go_return.r1);
}

fn Bi(x: Complex64) -> Complex64 {
    return -complex(0.0, 1.0) * Ai(x)
        + 2.0 * Ai(x * complex(-0.5, 3.0_f64.sqrt() / 2.0)) * complex(3_f64.sqrt() /
            2.0, 0.5);
}

#[derive(Clone)]
pub struct AiryWaveFunction {
    c: Complex64,
    u_1: f64,
    pub turning_point: f64,
    phase: Arc<Phase>,
    pub ts: (f64, f64),
    op: fn(Complex64) -> Complex64,
    phase_off: f64,
}

impl AiryWaveFunction {
    pub fn get_op(&self) -> Box<fn(Complex64) -> Complex64> {
        Box::new(self.op)
    }

    fn get_u_1_cube_root(u_1: f64) -> f64 {
        signum(u_1) * u_1.abs().pow(1.0 / 3.0)
    }

    pub fn new<'a>(phase: Arc<Phase>, view: (f64, f64)) -> (Vec<AiryWaveFunction>,
        TGroup) {
        let phase = phase;
        let turning_point_boundaries = turning_points::calc_ts(phase.as_ref(), view);

        let funcs: Vec<AiryWaveFunction> = turning_point_boundaries
            .ts
            .iter()
            .map(|((t1, t2), _)| {
                let x_1 = newtons_method(
```

```
51                      &|x| (phase.potential)(x) - phase.energy,
52                      (*t1 + *t2) / 2.0,
53                      1e-7,
54                  );
55                  let u_1 = 2.0 * phase.mass * -derivative(phase.potential.as_ref(),
                        x_1);
56                  // let u_1 = |x| -2.0 * phase.mass * ((phase.potential)(&x) - phase.
                        energy) / (H_BAR * H_BAR * (x - x_1));
57
58                  AiryWaveFunction {
59                      u_1,
60                      turning_point: x_1,
61                      phase: phase.clone(),
62                      ts: (*t1, *t2),
63                      op: identity,
64                      c: 1.0.into(),
65                      phase_off: 0.0,
66                  }
67              })
68              .collect::<Vec<AiryWaveFunction>>();
69          return (funcs, turning_point_boundaries);
70      }
71
72      pub fn with_op(&self, op: fn(Complex64) -> Complex64) -> AiryWaveFunction {
73          AiryWaveFunction {
74              u_1: self.u_1,
75              turning_point: self.turning_point,
76              phase: self.phase.clone(),
77              ts: self.ts,
78              op,
79              c: self.c,
80              phase_off: self.phase_off,
81          }
82      }
83
84      pub fn with_c(&self, c: Complex64) -> AiryWaveFunction {
85          AiryWaveFunction {
86              u_1: self.u_1,
87              turning_point: self.turning_point,
88              phase: self.phase.clone(),
89              ts: self.ts,
90              op: self.op,
91              c,
92              phase_off: self.phase_off,
93          }
94      }
95
96      pub fn with_phase_off(&self, phase_off: f64) -> AiryWaveFunction {
97          AiryWaveFunction {
```

```rust
            u_1: self.u_1,
            turning_point: self.turning_point,
            phase: self.phase.clone(),
            ts: self.ts,
            op: self.op,
            c: self.c,
            phase_off,
        }
    }
}

impl Func<f64, Complex64> for AiryWaveFunction {
    fn eval(&self, x: f64) -> Complex64 {
        let u_1_cube_root = Self::get_u_1_cube_root(self.u_1);

        if self.u_1 < 0.0 {
            return (self.op)(
                ((std::f64::consts::PI.sqrt() / (self.u_1).abs().pow(1.0 / 6.0))
                    * Ai(complex(u_1_cube_root * (self.turning_point - x), 0.0)))
                    as Complex64
                    * if COMPLEX_AIRY {
                        complex(
                            ((-(self.turning_point - x) + self.phase_off)).cos(),
                            ((-(self.turning_point - x) + self.phase_off)).sin(),
                        )
                    } else {
                        1.0.into()
                    },
            );
        } else {
            return (self.op)(
                ((std::f64::consts::PI.sqrt() / (self.u_1).abs().pow(1.0 / 6.0))
                    * Ai(complex(u_1_cube_root * (self.turning_point - x), 0.0)))
                    as Complex64
                    * if COMPLEX_AIRY {
                        complex(
                            ((self.turning_point - x) + self.phase_off).cos(),
                            ((self.turning_point - x) + self.phase_off).sin(),
                        )
                    } else {
                        1.0.into()
                    },
            );
        }
    }
}

#[cfg(test)]
mod test {
```

```rust
    use super::*;

    #[test]
    fn airy_func_plot() {
        let output_dir = Path::new("output");
        std::env::set_current_dir(&output_dir).unwrap();

        let airy_ai = Function::new(|x| Ai(complex(x, 0.0)));
        let airy_bi = Function::new(|x| Bi(complex(x, 0.0)));
        let values = evaluate_function_between(&airy_ai, -10.0, 5.0, NUMBER_OF_POINTS
            );

        let mut data_file = File::create("airy.txt").unwrap();

        let data_str_ai: String = values
            .par_iter()
            .map(|p| -> String { format!("{} {} {}\n", p.x, p.y.re, p.y.im) })
            .reduce(|| String::new(), |s: String, current: String| s + &*current);

        let values_bi = evaluate_function_between(&airy_bi, -5.0, 2.0,
            NUMBER_OF_POINTS);

        let data_str_bi: String = values_bi
            .par_iter()
            .map(|p| -> String { format!("{} {} {}\n", p.x, p.y.re, p.y.im) })
            .reduce(|| String::new(), |s: String, current: String| s + &*current);

        data_file
            .write_all((data_str_ai + "\n\n" + &*data_str_bi).as_ref())
            .unwrap()
    }
}
```

src/check.rs

```rust
use crate::*;

pub struct SchroedingerError<'a> {
    pub wave_func: &'a WaveFunction,
}

impl Func<f64, Complex64> for SchroedingerError<'_> {
    fn eval(&self, x: f64) -> Complex64 {
        complex(-1.0 / (2.0 * self.wave_func.get_phase().mass), 0.0)
            * Derivative {
                f: &Derivative { f: self.wave_func },
            }
            .eval(x)
            + ((self.wave_func.get_phase().potential)(x) - self.wave_func.get_phase()
```

```
15                .energy)
                * self.wave_func.eval(x)
16        }
17  }
```

src/energy.rs

```
 1  use crate::*;
 2
 3  struct Integrand<'a, F: Fn(f64) -> f64 + Sync> {
 4      mass: f64,
 5      pot: &'a F,
 6      view: (f64, f64),
 7      energy: f64,
 8  }
 9
10  impl<F: Fn(f64) -> f64 + Sync> Func<f64, f64> for Integrand<'_, F> {
11      fn eval(&self, x: f64) -> f64 {
12          let pot = (self.pot)(x);
13
14          if !pot.is_finite() {
15              return 0.0;
16          }
17
18          if pot < self.energy {
19              return (2.0 * self.mass * (self.energy - pot)).sqrt();
20          } else {
21              return 0.0;
22          }
23      }
24  }
25
26  struct SommerfeldCond<'a, F: Fn(f64) -> f64 + Sync> {
27      mass: f64,
28      pot: &'a F,
29      view: (f64, f64),
30  }
31
32  impl<F: Fn(f64) -> f64 + Sync> Func<f64, f64> for SommerfeldCond<'_, F> {
33      fn eval(&self, energy: f64) -> f64 {
34          let integrand = Integrand {
35              mass: self.mass,
36              pot: self.pot,
37              view: self.view,
38              energy,
39          };
40          let integral = integrate(
41              evaluate_function_between(&integrand, self.view.0, self.view.1,
                  INTEG_STEPS),
```

```rust
                TRAPEZE_PER_THREAD,
        );
        return ((integral - f64::consts::PI) / f64::consts::TAU) % 1.0;
    }
}

pub fn nth_energy<F: Fn(f64) -> f64 + Sync>(n: usize, mass: f64, pot: &F, view: (f64,
    f64)) -> f64 {
    const ENERGY_STEP: f64 = 10.0;
    const CHECKS_PER_ENERGY_STEP: usize = INTEG_STEPS;
    let sommerfeld_cond = SommerfeldCond { mass, pot, view };

    let mut energy = 0.0; // newtons_method_non_smooth(&|e| sommerfeld_cond.eval(e),
        1e-7, 1e-7);
    let mut i = 0;

    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();
    }
}

#[cfg(test)]
mod test {
    use super::*;

    // #[test]
    // fn square() {
    //     let pot = |x| x * x;
```

```
87    //     assert!((nth_energy(0, 1.0, &pot, (-100.0, 100.0)) - 0.707107).abs() < 1e
          -7);
88    // }
89  }
```

src/integrals.rs

```
 1  use crate::*;
 2  use rayon::prelude::*;
 3
 4  #[derive(Clone)]
 5  pub struct Point<T_X, T_Y> {
 6      pub x: T_X,
 7      pub y: T_Y,
 8  }
 9
10  pub fn trapezoidal_approx<X, Y>(start: &Point<X, Y>, end: &Point<X, Y>) -> Y
11  where
12      X: std::ops::Sub<Output = X> + Copy,
13      Y: std::ops::Add<Output = Y>
14          + std::ops::Mul<Output = Y>
15          + std::ops::Div<f64, Output = Y>
16          + Copy
17          + From<X>,
18  {
19      return Y::from(end.x - start.x) * (start.y + end.y) / 2.0_f64;
20  }
21
22  pub fn index_to_range<T>(x: T, in_min: T, in_max: T, out_min: T, out_max: T) -> T
23  where
24      T: Copy
25          + std::ops::Sub<Output = T>
26          + std::ops::Mul<Output = T>
27          + std::ops::Div<Output = T>
28          + std::ops::Add<Output = T>,
29  {
30      return (x - in_min) * (out_max - out_min) / (in_max - in_min) + out_min;
31  }
32
33  pub fn integrate<
34      X: Sync + std::ops::Add<Output = X> + std::ops::Sub<Output = X> + Copy,
35      Y: Default
36          + Sync
37          + std::ops::AddAssign
38          + std::ops::Div<f64, Output = Y>
39          + std::ops::Mul<Output = Y>
40          + std::ops::Add<Output = Y>
41          + Send
42          + std::iter::Sum<Y>
```

```rust
            + Copy
            + From<X>,
>(
    points: Vec<Point<X, Y>>,
    batch_size: usize,
) -> Y {
    if points.len() < 2 {
        return Y::default();
    }

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

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

    let mut rest = Y::default();

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

    return parallel + rest;
}

pub fn evaluate_function_between<X, Y>(f: &dyn Func<X, Y>, a: X, b: X, n: usize) ->
    Vec<Point<X, Y>>
where
    X: Copy
        + Send
        + Sync
        + std::cmp::PartialEq
        + From<f64>
        + std::ops::Add<Output = X>
        + std::ops::Sub<Output = X>
        + std::ops::Mul<Output = X>
        + std::ops::Div<Output = X>,
    Y: Send + Sync,
{
    if a == b {
        return vec![];
```

```rust
 90        }
 91
 92      (0..n)
 93          .into_par_iter()
 94          .map(|i| {
 95              index_to_range(
 96                  X::from(i as f64),
 97                  X::from(0.0_f64),
 98                  X::from((n - 1) as f64),
 99                  a,
100                  b,
101              )
102          })
103          .map(|x: X| Point { x, y: f.eval(x) })
104          .collect()
105 }
106
107 #[cfg(test)]
108 mod test {
109     use super::*;
110
111     fn square(x: f64) -> Complex64 {
112         return complex(x * x, 0.0);
113     }
114
115     fn square_integral(a: f64, b: f64) -> Complex64 {
116         return complex(b * b * b / 3.0 - a * a * a / 3.0, 0.0);
117     }
118
119     #[tokio::test(flavor = "multi_thread")]
120     async fn integral_of_square() {
121         let square_func: Function<f64, Complex64> = Function::new(square);
122         for i in 0..100 {
123             for j in 0..10 {
124                 let a = f64::from(i - 50) / 12.3;
125                 let b = f64::from(j - 50) / 12.3;
126
127                 if i == j {
128                     assert_eq!(
129                         integrate(
130                             evaluate_function_between(&square_func, a, b, INTEG_STEPS
                                 ),
131                             TRAPEZE_PER_THREAD,
132                         ),
133                         complex(0.0, 0.0)
134                     );
135                     continue;
136                 }
137
```

```rust
                let epsilon = 0.00001;
                assert!(complex_compare(
                    integrate(
                        evaluate_function_between(&square_func, a, b, INTEG_STEPS),
                        TRAPEZE_PER_THREAD,
                    ),
                    square_integral(a, b),
                    epsilon,
                ));
            }
        }
    }

    #[test]
    fn evaluate_square_func_between() {
        let square_func: Function<f64, Complex64> = Function::new(square);
        let actual = evaluate_function_between(&square_func, -2.0, 2.0, 5);
        let expected = vec![
            Point {
                x: -2.0,
                y: complex(4.0, 0.0),
            },
            Point {
                x: -1.0,
                y: complex(1.0, 0.0),
            },
            Point {
                x: 0.0,
                y: complex(0.0, 0.0),
            },
            Point {
                x: 1.0,
                y: complex(1.0, 0.0),
            },
            Point {
                x: 2.0,
                y: complex(4.0, 0.0),
            },
        ];

        for (a, e) in actual.iter().zip(expected) {
            assert_eq!(a.x, e.x);
            assert_eq!(a.y, e.y);
        }
    }

    fn sinusoidal_exp_complex(x: f64) -> Complex64 {
        return complex(x, x).exp();
    }
```

```
187
188    fn sinusoidal_exp_complex_integral(a: f64, b: f64) -> Complex64 {
189        // (-1/2 + i/2) (e^((1 + i) a) - e^((1 + i) b))
190        return complex(-0.5, 0.5) * (complex(a, a).exp() - complex(b, b).exp());
191    }
192
193    #[tokio::test(flavor = "multi_thread")]
194    async fn integral_of_sinusoidal_exp() {
195        let SINUSOIDAL_EXP_COMPLEX: Function<f64, Complex64> =
196            Function::new(sinusoidal_exp_complex);
197        for i in 0..10 {
198            for j in 0..10 {
199                let a = f64::from(i - 50) / 12.3;
200                let b = f64::from(j - 50) / 12.3;
201
202                if i == j {
203                    assert_eq!(
204                        integrate(
205                            evaluate_function_between(&SINUSOIDAL_EXP_COMPLEX, a, b,
206                                INTEG_STEPS),
206                            TRAPEZE_PER_THREAD,
207                        ),
208                        complex(0.0, 0.0)
209                    );
210                    continue;
211                }
212                let epsilon = 0.0001;
213                assert!(complex_compare(
214                    integrate(
215                        evaluate_function_between(&SINUSOIDAL_EXP_COMPLEX, a, b,
                                INTEG_STEPS),
216                        TRAPEZE_PER_THREAD,
217                    ),
218                    sinusoidal_exp_complex_integral(a, b),
219                    epsilon,
220                ));
221            }
222        }
223    }
224 }
```

src/main.rs

```
1  mod airy;
2  mod airy_wave_func;
3  mod energy;
4  mod integrals;
5  mod newtons_method;
6  mod plot;
```

```rust
 7   mod potentials;
 8   mod tui;
 9   mod turning_points;
10   mod utils;
11   mod wave_function_builder;
12   mod wkb_wave_func;
13
14   use crate::airy::airy_ai;
15   use crate::airy_wave_func::AiryWaveFunction;
16   use crate::integrals::*;
17   use crate::newtons_method::derivative;
18   use crate::utils::Func;
19   use crate::utils::*;
20   use crate::wave_function_builder::*;
21   use crate::wkb_wave_func::WkbWaveFunction;
22   use num::complex::Complex64;
23   use num::pow::Pow;
24   use rayon::iter::*;
25   use std::collections::HashMap;
26   use std::f64;
27   use std::fs::File;
28   use std::io::Write;
29   use std::path::Path;
30   use std::sync::Arc;
31
32   const INTEG_STEPS: usize = 64000;
33   const TRAPEZE_PER_THREAD: usize = 1000;
34   const NUMBER_OF_POINTS: usize = 100000;
35
36   const AIRY_EXTRA: f64 = 0.5;
37   const WKB_TRANSITION_FRACTION: f64 = 0.05;
38
39   const ENABLE_WKB_JOINTS: bool = false;
40   const ENABLE_AIRY_JOINTS: bool = true;
41
42   const COMPLEX_AIRY: bool = false;
43   const COMPLEX_EXP_WKB: bool = false;
44   const COMPLEX_OSZ_WKB: bool = false;
45
46   const APPROX_INF: (f64, f64) = (-200.0, 200.0);
47
48   fn main() {
49       // let wave_function = wave_function_builder::SuperPosition::new(
50       //     &potentials::mexican_hat,
51       //     1.0,
52       //     &[
53       //         (5, 1.0.into()),
54       //         (12, 1.0.into()),
55       //         (40, 1.0.into()),
```

```
56    //        ],
57    //        APPROX_INF,
58    //        1.5,
59    //        ScalingType::Renormalize(complex(1.0, 0.0)),
60    // );
61
62    let wave_function = wave_function_builder::WaveFunction::new(
63        &potentials::square,
64        1.0,
65        5,
66        APPROX_INF,
67        1.5,
68        ScalingType::Renormalize(1.0.into()),
69    );
70
71    let output_dir = Path::new("output");
72    plot::plot_wavefunction_parts(&wave_function, output_dir, "data.txt");
73 }
```

src/newtons_method.rs

```
 1  use crate::integrals::*;
 2  use crate::utils::cmp_f64;
 3  use num::traits::FloatConst;
 4  use num::{signum, Float};
 5  use rayon::prelude::*;
 6  use std::cmp::Ordering;
 7  use std::fmt::Debug;
 8  use std::ops::*;
 9  use std::rc::Rc;
10  use std::sync::Arc;
11
12  #[derive(Default, Debug)]
13  pub struct Vec2 {
14      x: f64,
15      y: f64,
16  }
17
18  impl Vec2 {
19      pub fn dot(&self, other: &Vec2) -> f64 {
20          return self.x * other.x + self.y + other.y;
21      }
22
23      pub fn mag(&self) -> f64 {
24          return (self.x.powi(2) * self.y.powi(2)).sqrt();
25      }
26
27      pub fn pseudo_inverse(&self) -> CoVec2 {
28          CoVec2(self.x, self.y) * (1.0 / (self.x.powi(2) + self.y.powi(2)))
```

```rust
        }
}

impl Add for Vec2 {
    type Output = Vec2;

    fn add(self, other: Self) -> Self::Output {
        Vec2 {
            x: self.x + other.x,
            y: self.y + other.y,
        }
    }
}

impl Sub for Vec2 {
    type Output = Vec2;

    fn sub(self, other: Self) -> Self::Output {
        Vec2 {
            x: self.x - other.x,
            y: self.x - other.y,
        }
    }
}

impl Mul<f64> for Vec2 {
    type Output = Vec2;

    fn mul(self, s: f64) -> Self::Output {
        Vec2 {
            x: self.x * s,
            y: self.y * s,
        }
    }
}

#[derive(Debug)]
pub struct CoVec2(f64, f64);

impl Add for CoVec2 {
    type Output = CoVec2;

    fn add(self, other: Self) -> Self::Output {
        CoVec2(self.0 + other.0, self.1 + other.1)
    }
}

impl Sub for CoVec2 {
    type Output = CoVec2;
```

```rust
 78
 79      fn sub(self, other: Self) -> Self::Output {
 80          CoVec2(self.0 - other.0, self.1 - other.1)
 81      }
 82  }
 83
 84  impl Mul<Vec2> for CoVec2 {
 85      type Output = f64;
 86
 87      fn mul(self, vec: Vec2) -> Self::Output {
 88          return self.0 * vec.x + self.1 * vec.y;
 89      }
 90  }
 91
 92  impl Mul<f64> for CoVec2 {
 93      type Output = CoVec2;
 94
 95      fn mul(self, s: f64) -> Self::Output {
 96          CoVec2(self.0 * s, self.1 * s)
 97      }
 98  }
 99
100  fn gradient<F>(f: F, x: f64) -> Vec2
101  where
102      F: Fn(f64) -> Vec2,
103  {
104      let x_component = |x| f(x).x;
105      let y_component = |x| f(x).y;
106      return Vec2 {
107          x: derivative(&x_component, x),
108          y: derivative(&y_component, x),
109      };
110  }
111
112  // pub fn derivative<F, R>(f: &F, x: f64) -> R
113  // where
114  //     F: Fn(f64) -> R + ?Sized,
115  //     R: Sub<R, Output = R> + Div<f64, Output = R>,
116  // {
117  //     let epsilon = f64::epsilon().sqrt();
118  //     (f(x + epsilon / 2.0) - f(x - epsilon / 2.0)) / epsilon
119  // }
120
121  pub fn derivative<F, R>(func: &F, x: f64) -> R
122  where
123      F: Fn(f64) -> R + ?Sized,
124      R: Sub<R, Output = R> + Div<f64, Output = R> + Mul<f64, Output = R> + Add<R,
125          Output = R>,
126  {
```

```rust
        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;
}

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;
        }
    }
}

pub fn newtons_method_2d<F>(f: &F, mut guess: f64, precision: f64) -> f64
where
    F: Fn(f64) -> Vec2,
    F::Output: Debug,
{
    loop {
        let jacobian = gradient(f, guess);
        let step: f64 = jacobian.pseudo_inverse() * f(guess);
        if step.abs() < precision {
            return guess;
        } else {
            guess -= step;
        }
    }
}

pub fn newtons_method_max_iters<F>(
    f: &F,
    mut guess: f64,
```

```rust
    precision: f64,
    max_iters: usize,
) -> Option<f64>
where
    F: Fn(f64) -> f64,
{
    for _ in 0..max_iters {
        let step = f(guess) / derivative(f, guess);
        if step.abs() < precision {
            return Some(guess);
        } else {
            guess -= step;
        }
    }
    None
}

fn sigmoid(x: f64) -> f64 {
    1.0 / (1.0 + (-x).exp())
}

fn smooth_sgn(x: f64) -> f64 {
    if x > 0.0 {
        (x + 3.0).exp() - 3.0.exp()
    } else {
        0.0
    }
}

fn check_sign(initial: f64, new: f64) -> bool {
    if initial == new {
        return false;
    }
    return (initial <= -0.0 && new >= 0.0) || (initial >= 0.0 && new <= 0.0);
}

pub fn bisection_search_sign_change<F>(f: &F, initial_guess: f64, step: f64) -> (f64,
    f64)
where
    F: Fn(f64) -> f64 + ?Sized,
{
    let mut result = initial_guess;
    while !check_sign(f(initial_guess), f(result)) {
        result += step
    }
    return (result - step, result);
}

fn regula_falsi_c<F>(f: &F, a: f64, b: f64) -> f64
```

```rust
where
    F: Fn(f64) -> f64 + ?Sized,
{
    return (a * f(b) - b * f(a)) / (f(b) - f(a));
}

pub fn regula_falsi_method<F>(f: &F, mut a: f64, mut b: f64, precision: f64) -> f64
where
    F: Fn(f64) -> f64 + ?Sized,
{
    if a > b {
        let temp = a;
        a = b;
        b = temp;
    }

    let mut c = regula_falsi_c(f, a, b);
    while f64::abs(f(c)) > precision {
        b = regula_falsi_c(f, a, b);
        a = regula_falsi_c(f, a, b);
        c = regula_falsi_c(f, a, b);
    }
    return c;
}

pub fn regula_falsi_bisection<F>(f: &F, guess: f64, bisection_step: f64, precision:
    f64) -> f64
where
    F: Fn(f64) -> f64 + ?Sized,
{
    let (a, b) = bisection_search_sign_change(f, guess, bisection_step);
    return regula_falsi_method(f, a, b, precision);
}

#[derive(Clone)]
pub struct NewtonsMethodFindNewZero<F>
where
    F: Fn(f64) -> f64 + ?Sized + Clone,
{
    f: Arc<F>,
    precision: f64,
    max_iters: usize,
    previous_zeros: Vec<(i32, f64)>,
}

impl<F: Fn(f64) -> f64 + ?Sized + Clone> NewtonsMethodFindNewZero<F> {
    pub(crate) fn new(f: Arc<F>, precision: f64, max_iters: usize) ->
            NewtonsMethodFindNewZero<F> {
        NewtonsMethodFindNewZero {
```

```rust
                f,
                precision,
                max_iters,
                previous_zeros: vec![],
            }
        }

        pub(crate) fn modified_func(&self, x: f64) -> f64 {
            let divisor = self
                .previous_zeros
                .iter()
                .fold(1.0, |acc, (n, z)| acc * (x - z).powi(*n));
            let divisor = if divisor == 0.0 {
                divisor + self.precision
            } else {
                divisor
            };
            (self.f)(x) / divisor
        }

        pub(crate) fn next_zero(&mut self, guess: f64) -> Option<f64> {
            let zero = newtons_method_max_iters(
                &|x| self.modified_func(x),
                guess,
                self.precision,
                self.max_iters,
            );

            if let Some(z) = zero {
                // to avoid hitting maxima and minima twice
                if derivative(&|x| self.modified_func(x), z).abs() < self.precision {
                    self.previous_zeros.push((2, z));
                } else {
                    self.previous_zeros.push((1, z));
                }
            }

            return zero;
        }

        pub(crate) fn get_previous_zeros(&self) -> Vec<f64> {
            self.previous_zeros
                .iter()
                .map(|(_, z)| *z)
                .collect::<Vec<f64>>()
        }
}

pub fn make_guess<F>(f: &F, (start, end): (f64, f64), n: usize) -> Option<f64>
```

```rust
319 where
320     F: Fn(f64) -> f64 + Sync,
321 {
322     let sort_func = |(_, y1): &(f64, f64), (_, y2): &(f64, f64)| -> Ordering {
            cmp_f64(&y1, &y2) };
323     let mut points: Vec<(f64, f64)> = (0..n)
324         .into_par_iter()
325         .map(|i| index_to_range(i as f64, 0.0, n as f64, start, end))
326         .map(move |x| {
327             let der = derivative(f, x);
328             (x, f(x) / (-(-der * der).exp() + 1.0))
329         })
330         .map(|(x, y)| (x, y.abs()))
331         .collect();
332     points.sort_by(sort_func);
333     points.get(0).map(|point| point.0)
334 }
335
336 pub fn newtons_method_find_new_zero<F>(
337     f: &F,
338     mut guess: f64,
339     precision: f64,
340     max_iters: usize,
341     known_zeros: &Vec<f64>,
342 ) -> Option<f64>
343 where
344     F: Fn(f64) -> f64,
345 {
346     let f_modified = |x| f(x) / known_zeros.iter().fold(0.0, |acc, &z| acc * (x - z))
            ;
347     newtons_method_max_iters(&f_modified, guess, precision, max_iters)
348 }
349
350 pub fn inverse<F, A, R>(f: &F) -> Box<dyn Fn(R) -> Vec<A>>
351 where
352     F: Fn(A) -> R,
353 {
354     todo!();
355 }
356
357 #[cfg(test)]
358 mod test {
359     use super::*;
360     use crate::integrals::*;
361     use crate::utils::cmp_f64;
362     use num::zero;
363
364     fn float_compare(expect: f64, actual: f64, epsilon: f64) -> bool {
365         let average = (expect.abs() + actual.abs()) / 2.0;
```

58

```
366         if average != 0.0 {
367             (expect - actual).abs() / average < epsilon
368         } else {
369             (expect - actual).abs() < epsilon
370         }
371     }
372
373     #[test]
374     fn derivative_square_test() {
375         let square = |x| x * x;
376         let actual = |x| 2.0 * x;
377
378         for i in 0..100 {
379             let x = index_to_range(i as f64, 0.0, 100.0, -20.0, 20.0);
380             assert!(float_compare(derivative(&square, x), actual(x), 1e-4));
381         }
382     }
383
384     #[test]
385     fn derivative_exp_test() {
386         let exp = |x: f64| x.exp();
387
388         for i in 0..100 {
389             let x = index_to_range(i as f64, 0.0, 100.0, -20.0, 20.0);
390             assert!(float_compare(derivative(&exp, x), exp(x), 1e-4));
391         }
392     }
393
394     #[test]
395     fn newtons_method_square() {
396         for i in 0..100 {
397             let zero = index_to_range(i as f64, 0.0, 100.0, 0.1, 10.0);
398             let func = |x| x * x - zero * zero;
399             assert!(float_compare(
400                 newtons_method(&func, 100.0, 1e-7),
401                 zero,
402                 1e-4,
403             ));
404             assert!(float_compare(
405                 newtons_method(&func, -100.0, 1e-7),
406                 -zero,
407                 1e-4,
408             ));
409         }
410     }
411
412     #[test]
413     fn newtons_method_cube() {
414         for i in 0..100 {
```

```
415                let zero = index_to_range(i as f64, 0.0, 100.0, 0.1, 10.0);
416                let func = |x| (x - zero) * (x + zero) * (x - zero / 2.0);
417                assert!(float_compare(
418                    newtons_method(&func, 100.0, 1e-7),
419                    zero,
420                    1e-4,
421                ));
422                assert!(float_compare(
423                    newtons_method(&func, -100.0, 1e-7),
424                    -zero,
425                    1e-4,
426                ));
427                assert!(float_compare(
428                    newtons_method(&func, 0.0, 1e-7),
429                    zero / 2.0,
430                    1e-4,
431                ));
432            }
433        }
434
435        #[test]
436        fn newtons_method_find_next_polynomial() {
437            for i in 0..10 {
438                for j in 0..10 {
439                    for k in 0..10 {
440                        let a = index_to_range(i as f64, 0.0, 10.0, -10.0, 10.0);
441                        let b = index_to_range(j as f64, 0.0, 10.0, -100.0, 0.0);
442                        let c = index_to_range(k as f64, 0.0, 10.0, -1.0, 20.0);
443                        let test_func = |x: f64| (x - a) * (x - b) * (x - c);
444
445                        for guess in [a, b, c] {
446                            let mut finder =
447                                NewtonsMethodFindNewZero::new(Arc::new(test_func), 1e-15,
                                        10000000);
448
449                            finder.next_zero(1.0);
450                            finder.next_zero(1.0);
451                            finder.next_zero(1.0);
452
453                            let mut zeros_expected = [a, b, c];
454                            let mut zeros_actual = finder.get_previous_zeros().clone();
455
456                            zeros_expected.sort_by(cmp_f64);
457                            zeros_actual.sort_by(cmp_f64);
458
459                            assert_eq!(zeros_actual.len(), 3);
460
461                            for (expected, actual) in zeros_expected.iter().zip(
                                    zeros_actual.iter()) {
```

```rust
462                          assert!((*expected - *actual).abs() < 1e-10);
463                      }
464                  }
465              }
466          }
467      }
468  }
469
470      #[test]
471      fn newtons_method_find_next_test() {
472          use std::f64::consts;
473          let interval = (-10.0, 10.0);
474
475          let test_func = |x: f64| 5.0 * (3.0 * x + 1.0).abs() - (1.5 * x.powi(2) + x -
476              50.0).powi(2);
477
478          let mut finder = NewtonsMethodFindNewZero::new(Arc::new(test_func), 1e-11,
479              100000000);
480
481          for i in 0..4 {
482              let guess = make_guess(&|x| finder.modified_func(x), interval, 1000);
483              finder.next_zero(guess.unwrap());
484          }
485
486          let mut zeros = finder.get_previous_zeros().clone();
487          zeros.sort_by(cmp_f64);
488          let expected = [-6.65276132415, -5.58024707627, 4.91358040961,
489              5.98609465748];
490
491          println!("zeros: {:#?}", zeros);
492
493          assert_eq!(zeros.len(), expected.len());
494
495          for (expected, actual) in expected.iter().zip(zeros.iter()) {
496              assert!((*expected - *actual).abs() < 1e-10);
497          }
498      }
499
500      #[test]
501      fn regula_falsi_bisection_test() {
502          let func = |x: f64| x * (x - 2.0) * (x + 2.0);
503
504          let actual = regula_falsi_bisection(&func, -1e-3, -1e-3, 1e-5);
505          let expected = -2.0;
506
507          println!("expected: {}, actual {}", expected, actual);
508          assert!(float_compare(expected, actual, 1e-3));
509      }
510 }
```

src/plot.rs

```rust
use crate::*;
use std::fmt;

pub fn to_gnuplot_string_complex<X>(values: Vec<Point<X, Complex64>>) -> String
where
    X: fmt::Display + Send + Sync,
{
    values
        .par_iter()
        .map(|p| -> String { format!("{} {} {}\n", p.x, p.y.re, p.y.im) })
        .reduce(|| String::new(), |s: String, current: String| s + &*current)
}

pub fn to_gnuplot_string<X, Y>(values: Vec<Point<X, Y>>) -> String
where
    X: fmt::Display + Send + Sync,
    Y: fmt::Display + Send + Sync,
{
    values
        .par_iter()
        .map(|p| -> String { format!("{} {}\n", p.x, p.y) })
        .reduce(|| String::new(), |s: String, current: String| s + &*current)
}

pub fn plot_wavefunction_parts(wave_function: &WaveFunction, output_dir: &Path,
    output_file: &str) {
    std::env::set_current_dir(&output_dir).unwrap();

    let wkb_values = wave_function
        .get_wkb_ranges_in_view()
        .iter()
        .map(|range| evaluate_function_between(wave_function, range.0, range.1,
            NUMBER_OF_POINTS))
        .collect::<Vec<Vec<Point<f64, Complex64>>>>();

    let airy_values = wave_function
        .get_airy_ranges()
        .iter()
        .map(|range| {
            evaluate_function_between(
                wave_function,
                f64::max(wave_function.get_view().0, range.0),
                f64::min(wave_function.get_view().1, range.1),
                NUMBER_OF_POINTS,
            )
        })
```

```
45          .collect::<Vec<Vec<Point<f64, Complex64>>>>();
46
47      let wkb_values_str = wkb_values
48          .par_iter()
49          .map(|values| to_gnuplot_string_complex(values.to_vec()))
50          .reduce(
51              || String::new(),
52              |s: String, current: String| s + "\n\n" + &*current,
53          );
54
55      let airy_values_str = airy_values
56          .par_iter()
57          .map(|values| to_gnuplot_string_complex(values.to_vec()))
58          .reduce(
59              || String::new(),
60              |s: String, current: String| s + "\n\n" + &*current,
61          );
62
63      let mut data_full = File::create(output_file).unwrap();
64      data_full.write_all(wkb_values_str.as_ref()).unwrap();
65      data_full.write_all("\n\n".as_bytes()).unwrap();
66      data_full.write_all(airy_values_str.as_ref()).unwrap();
67
68      let mut plot_3d_file = File::create("plot_3d.gnuplot").unwrap();
69
70      let wkb_3d_cmd = (1..=wkb_values.len())
71          .into_iter()
72          .map(|n| {
73              format!(
74                  "\"{}\" u 1:2:3 i {} t \"WKB {}\" w l",
75                  output_file,
76                  n - 1,
77                  n
78              )
79          })
80          .collect::<Vec<String>>()
81          .join(", ");
82
83      let airy_3d_cmd = (1..=airy_values.len())
84          .into_iter()
85          .map(|n| {
86              format!(
87                  "\"{}\" u 1:2:3 i {} t \"Airy {}\" w l",
88                  output_file,
89                  n + wkb_values.len() - 1,
90                  n
91              )
92          })
93          .collect::<Vec<String>>()
```

```rust
                .join(", ");
    let plot_3d_cmd: String = "splot ".to_string() + &wkb_3d_cmd + ", " + &
        airy_3d_cmd;
    plot_3d_file.write_all(plot_3d_cmd.as_ref()).unwrap();

    let mut plot_file = File::create("plot.gnuplot").unwrap();
    let wkb_cmd = (1..=wkb_values.len())
        .into_iter()
        .map(|n| {
            format!(
                "\"{}\" u 1:2 i {} t \"Re(WKB {})\" w l",
                output_file,
                n - 1,
                n
            )
        })
        .collect::<Vec<String>>()
        .join(", ");

    let airy_cmd = (1..=airy_values.len())
        .into_iter()
        .map(|n| {
            format!(
                "\"{}\" u 1:2 i {} t \"Re(Airy {})\" w l",
                output_file,
                n + wkb_values.len() - 1,
                n
            )
        })
        .collect::<Vec<String>>()
        .join(", ");
    let plot_cmd: String = "plot ".to_string() + &wkb_cmd + ", " + &airy_cmd;

    plot_file.write_all(plot_cmd.as_ref()).unwrap();

    let mut plot_imag_file = File::create("plot_im.gnuplot").unwrap();

    let wkb_im_cmd = (1..=wkb_values.len())
        .into_iter()
        .map(|n| {
            format!(
                "\"{}\" u 1:3 i {} t \"Im(WKB {})\" w l",
                output_file,
                n - 1,
                n
            )
        })
        .collect::<Vec<String>>()
        .join(", ");
```

```rust
    let airy_im_cmd = (1..=airy_values.len())
        .into_iter()
        .map(|n| {
            format!(
                "\"{}\"␣u␣1:3␣i␣{}␣t␣\"Im(Airy␣{})\"␣w␣l",
                output_file,
                n + wkb_values.len() - 1,
                n
            )
        })
        .collect::<Vec<String>>()
        .join(",␣");
    let plot_imag_cmd: String = "plot␣".to_string() + &wkb_im_cmd + ",␣" + &
        airy_im_cmd;

    plot_imag_file.write_all(plot_imag_cmd.as_ref()).unwrap();
}

pub fn plot_complex_function(
    func: &dyn Func<f64, Complex64>,
    view: (f64, f64),
    title: &str,
    output_dir: &Path,
    output_file: &str,
) {
    std::env::set_current_dir(&output_dir).unwrap();
    let values = evaluate_function_between(func, view.0, view.1, NUMBER_OF_POINTS);

    let values_str = to_gnuplot_string_complex(values);

    let mut data_file = File::create(output_file).unwrap();

    data_file.write_all(values_str.as_bytes()).unwrap();

    let mut plot_3d_file = File::create("plot_3d.gnuplot").unwrap();
    plot_3d_file
        .write_all(format!("splot␣\"{}\"␣u␣1:2:3␣t␣\"{}\"␣w␣l", output_file, title).
            as_bytes())
        .unwrap();

    let mut plot_file = File::create("plot.gnuplot").unwrap();
    plot_file
        .write_all(format!("plot␣\"{}\"␣u␣1:2␣t␣\"Re({})\"␣w␣l", output_file, title).
            as_bytes())
        .unwrap();

    let mut plot_im_file = File::create("plot_im.gnuplot").unwrap();
    plot_im_file
```

```rust
188             .write_all(format!("plot \"{}\" u 1:3 t \"Im({})\" w l", output_file, title).
                    as_bytes())
189             .unwrap();
190 }
191
192 pub fn plot_wavefunction(wave_function: &WaveFunction, output_dir: &Path, output_file
        : &str) {
193     plot_complex_function(
194         wave_function,
195         wave_function.get_view(),
196         "Psi",
197         output_dir,
198         output_file,
199     );
200 }
201
202 pub fn plot_superposition(wave_function: &SuperPosition, output_dir: &Path,
        output_file: &str) {
203     plot_complex_function(
204         wave_function,
205         wave_function.get_view(),
206         "Psi",
207         output_dir,
208         output_file,
209     );
210 }
211
212 pub fn plot_probability(wave_function: &WaveFunction, output_dir: &Path, output_file:
         &str) {
213     std::env::set_current_dir(&output_dir).unwrap();
214     let values = evaluate_function_between(
215         wave_function,
216         wave_function.get_view().0,
217         wave_function.get_view().1,
218         NUMBER_OF_POINTS,
219     )
220     .par_iter()
221     .map(|p| Point {
222         x: p.x,
223         y: p.y.norm_sqr(),
224     })
225     .collect();
226
227     let values_str = to_gnuplot_string(values);
228
229     let mut data_file = File::create(output_file).unwrap();
230
231     data_file.write_all(values_str.as_bytes()).unwrap();
232
```

```rust
233    let mut plot_file = File::create("plot.gnuplot").unwrap();
234    plot_file
235        .write_all(format!("plot␣\"{}\"␣u␣1:2␣t␣\"|Psi|^2\"␣w␣l", output_file).
                as_bytes())
236        .unwrap();
237 }
238
239 pub fn plot_probability_super_pos(
240    wave_function: &SuperPosition,
241    output_dir: &Path,
242    output_file: &str,
243 ) {
244    std::env::set_current_dir(&output_dir).unwrap();
245    let values = evaluate_function_between(
246        wave_function,
247        wave_function.get_view().0,
248        wave_function.get_view().1,
249        NUMBER_OF_POINTS,
250    ).par_iter()
251    .map(|p| Point {
252        x: p.x,
253        y: p.y.norm_sqr(),
254    })
255    .collect();
256
257    let values_str = to_gnuplot_string(values);
258
259    let mut data_file = File::create(output_file).unwrap();
260
261    data_file.write_all(values_str.as_bytes()).unwrap();
262
263    let mut plot_file = File::create("plot.gnuplot").unwrap();
264    plot_file
265        .write_all(format!("plot␣\"{}\"␣u␣1:2␣t␣\"|Psi|^2\"␣w␣l", output_file).
                as_bytes())
266        .unwrap();
267 }
```

src/potentials.rs

```rust
1 use crate::*;
2
3 const ENERGY_INF: f64 = 1e6;
4
5 #[allow(unused)]
6 pub fn smooth_step(x: f64) -> f64 {
7    const TRANSITION: f64 = 0.5;
8    let step = Arc::new(Function::new(|x: f64| -> Complex64 {
9        if x.abs() < 2.0 {
```

```rust
10              complex(10.0, 0.0)
11          } else {
12              complex(0.0, 0.0)
13          }
14      }));
15      let zero = Arc::new(Function::new(|_: f64| -> Complex64 { complex(0.0, 0.0) }));
16      let inf = Arc::new(Function::new(|x: f64| -> Complex64 {
17          if x.abs() > 5.0 {
18              complex(ENERGY_INF, 0.0)
19          } else {
20              complex(0.0, 0.0)
21          }
22      }));
23
24      let joint_inf_zero_l = wave_function_builder::Joint {
25          left: inf.clone(),
26          right: zero.clone(),
27          cut: -5.0 + TRANSITION / 2.0,
28          delta: TRANSITION,
29      };
30
31      let joint_zero_step_l = wave_function_builder::Joint {
32          left: zero.clone(),
33          right: step.clone(),
34          cut: -2.0 + TRANSITION / 2.0,
35          delta: TRANSITION,
36      };
37
38      let joint_zero_inf_r = wave_function_builder::Joint {
39          left: zero.clone(),
40          right: inf.clone(),
41          cut: 5.0 - TRANSITION / 2.0,
42          delta: TRANSITION,
43      };
44
45      let joint_step_zero_r = wave_function_builder::Joint {
46          left: step.clone(),
47          right: zero.clone(),
48          cut: 2.0 - TRANSITION / 2.0,
49          delta: TRANSITION,
50      };
51
52      if wave_function_builder::is_in_range(joint_zero_inf_r.range(), x) {
53          return joint_zero_inf_r.eval(x).re;
54      }
55
56      if wave_function_builder::is_in_range(joint_inf_zero_l.range(), x) {
57          return joint_inf_zero_l.eval(x).re;
58      }
```

```
59
60     if wave_function_builder::is_in_range(joint_step_zero_r.range(), x) {
61         return joint_step_zero_r.eval(x).re;
62     }
63
64     if wave_function_builder::is_in_range(joint_zero_step_l.range(), x) {
65         return joint_zero_step_l.eval(x).re;
66     }
67
68     return zero.eval(x).re.max(inf.eval(x).re.max(step.eval(x).re));
69 }
70
71 #[allow(unused)]
72 pub fn mexican_hat(x: f64) -> f64 {
73     (x - 4.0).powi(2) * (x + 4.0).powi(2)
74 }
75
76 #[allow(unused)]
77 pub fn double_mexican_hat(x: f64) -> f64 {
78     (x - 4.0).powi(2) * x.powi(2) * (x + 4.0).powi(2)
79 }
80
81 #[allow(unused)]
82 pub fn triple_mexican_hat(x: f64) -> f64 {
83     (x - 6.0).powi(2) * (x - 3.0).powi(2) * (x + 3.0).powi(2) * (x + 6.0).powi(2)
84 }
85
86 pub fn square(x: f64) -> f64 {
87     x * x
88 }
```

src/tui.rs

```
1 use std::io;
2
3 fn get_float_from_user(message: &str) -> f64 {
4     loop {
5         println!("{}", message);
6         let mut input = String::new();
7
8         // io::stdout().lock().write(message.as_ref()).unwrap();
9         io::stdin()
10             .read_line(&mut input)
11             .expect("Not␣a␣valid␣string");
12         println!("");
13         let num = input.trim().parse();
14         if num.is_ok() {
15             return num.unwrap();
16         }
```

```
17        }
18  }
19
20  fn get_user_bounds() -> (f64, f64) {
21      let user_bound_lower: f64 = get_float_from_user("Lower␣Bound:␣");
22
23      let user_bound_upper: f64 = get_float_from_user("Upper_bound:␣");
24      return (user_bound_lower, user_bound_upper);
25  }
26  fn ask_user_for_view(lower_bound: Option<f64>, upper_bound: Option<f64>) -> (f64, f64
        ) {
27      println!("Failed␣to␣determine␣boundary␣of␣the␣graph␣automatically.");
28      println!("Pleas␣enter␣values␣manually.");
29      lower_bound.map(|b| println!("(Suggestion␣for␣lower␣bound:␣{})", b));
30      upper_bound.map(|b| println!("(Suggestion␣for␣upper␣bound:␣{})", b));
31
32      return get_user_bounds();
33  }
```

src/turning_points.rs

```
 1  use crate::cmp_f64;
 2  use crate::newtons_method::*;
 3  use crate::wkb_wave_func::*;
 4  use crate::*;
 5  use num::signum;
 6
 7  const MAX_TURNING_POINTS: usize = 256;
 8  const ACCURACY: f64 = 1e-9;
 9
10  pub struct TGroup {
11      pub ts: Vec<((f64, f64), f64)>,
12      // pub tn: Option<f64>,
13  }
14
15  impl TGroup {
16      pub fn new() -> TGroup {
17          TGroup { ts: vec![] }
18      }
19
20      pub fn add_ts(&mut self, new_t: ((f64, f64), f64)) {
21          self.ts.push(new_t);
22      }
23  }
24
25  fn validity_func(phase: Phase) -> Arc<dyn Fn(f64) -> f64> {
26      Arc::new(move |x: f64| {
27          1.0 / (2.0 * phase.mass).sqrt() * derivative(&|t| (phase.potential)(t), x).
                abs()
```

```rust
28                - ((phase.potential)(x) - phase.energy).pow(2)
29        })
30 }
31
32 fn group_ts(zeros: &Vec<f64>, phase: &Phase) -> TGroup {
33     let mut zeros = zeros.clone();
34     let valid = validity_func(phase.clone());
35
36     zeros.sort_by(cmp_f64);
37     let mut derivatives = zeros
38         .iter()
39         .map(|x| derivative(valid.as_ref(), *x))
40         .map(signum)
41         .zip(zeros.clone())
42         .collect::<Vec<(f64, f64)>>();
43
44     let mut groups = TGroup { ts: vec![] };
45
46     if let Some((deriv, z)) = derivatives.first() {
47         if *deriv < 0.0 {
48             let mut guess = z - ACCURACY.sqrt();
49             let mut new_deriv = *deriv;
50             let mut missing_t = *z;
51
52             while new_deriv < 0.0 {
53                 missing_t =
54                     regula_falsi_bisection(valid.as_ref(), guess, -ACCURACY.sqrt(),
55                         ACCURACY);
55                 new_deriv = signum(derivative(valid.as_ref(), missing_t));
56                 guess -= ACCURACY.sqrt();
57             }
58
59             derivatives.insert(
60                 0,
61                 (signum(derivative(valid.as_ref(), missing_t)), missing_t),
62             );
63         }
64     }
65
66     if let Some((deriv, z)) = derivatives.last() {
67         if *deriv > 0.0 {
68             let mut guess = z + ACCURACY.sqrt();
69             let mut new_deriv = *deriv;
70             let mut missing_t = *z;
71
72             while new_deriv > 0.0 {
73                 missing_t =
74                     regula_falsi_bisection(valid.as_ref(), guess, ACCURACY.sqrt(),
75                         ACCURACY);
```

```rust
                new_deriv = signum(derivative(valid.as_ref(), missing_t));
                guess += ACCURACY.sqrt();
            }

            derivatives.push((signum(derivative(valid.as_ref(), missing_t)),
                missing_t));
        }
    }

    assert_eq!(derivatives.len() % 2, 0);

    for i in (0..derivatives.len()).step_by(2) {
        let (t1_deriv, t1) = derivatives[i];
        let (t2_deriv, t2) = derivatives[i + 1];
        assert!(t1_deriv > 0.0);
        assert!(t2_deriv < 0.0);

        let turning_point = newtons_method(
            &|x| phase.energy - (phase.potential)(x),
            (t1 + t2) / 2.0,
            1e-7,
        );
        groups.add_ts(((t1, t2), turning_point));
    }

    return groups;
}

pub fn calc_ts(phase: &Phase, view: (f64, f64)) -> TGroup {
    // return TGroup{ts:vec![(-4.692, -4.255), (4.255, 4.692)]};
    let zeros = find_zeros(phase, view);
    return group_ts(&zeros, phase);
}

fn find_zeros(phase: &Phase, view: (f64, f64)) -> Vec<f64> {
    let phase_clone = phase.clone();
    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));
```

```
121        });
122
123        let view = if view.0 < view.1 {
124            view
125        } else {
126            (view.1, view.0)
127        };
128        let unique_zeros = zeros
129            .get_previous_zeros()
130            .iter()
131            .filter(|x| **x > view.0 && **x < view.1)
132            .map(|x| *x)
133            .collect::<Vec<f64>>();
134        return unique_zeros;
135 }
136
137 #[cfg(test)]
138 mod test {
139        use super::*;
140 }
```

src/utils.rs

```
1  use crate::newtons_method::derivative;
2  use crate::Complex64;
3  use std::cmp::Ordering;
4
5  pub fn cmp_f64(a: &f64, b: &f64) -> Ordering {
6      if a < b {
7          return Ordering::Less;
8      } else if a > b {
9          return Ordering::Greater;
10     }
11     return Ordering::Equal;
12 }
13
14 pub fn complex(re: f64, im: f64) -> Complex64 {
15     return Complex64 { re, im };
16 }
17
18 pub fn sigmoid(x: f64) -> f64 {
19     1.0 / (1.0 + (-x).exp())
20 }
21
22 pub fn identity(c: Complex64) -> Complex64 {
23     c
24 }
25
26 pub fn conjugate(c: Complex64) -> Complex64 {
```

```rust
27      c.conj()
28  }
29
30  pub fn negative(c: Complex64) -> Complex64 {
31      -c
32  }
33
34  pub fn negative_conj(c: Complex64) -> Complex64 {
35      -c.conj()
36  }
37
38  pub fn complex_compare(expect: Complex64, actual: Complex64, epsilon: f64) -> bool {
39      let average = (expect.norm() + actual.norm()) / 2.0;
40      return (expect - actual).norm() / average < epsilon;
41  }
42
43  pub fn float_compare(expect: f64, actual: f64, epsilon: f64) -> bool {
44      let average = (expect + actual) / 2.0;
45
46      if average < epsilon {
47          return expect == actual;
48      }
49
50      return (expect - actual) / average < epsilon;
51  }
52
53  pub trait Func<A, R>: Sync + Send {
54      fn eval(&self, x: A) -> R;
55  }
56
57  pub trait ReToC: Sync + Func<f64, Complex64> {}
58
59  pub trait ReToRe: Sync + Func<f64, f64> {}
60
61  pub struct Function<A, R> {
62      pub(crate) f: fn(A) -> R,
63  }
64
65  impl<A, R> Function<A, R> {
66      pub const fn new(f: fn(A) -> R) -> Function<A, R> {
67          return Function { f };
68      }
69  }
70
71  impl<A, R> Func<A, R> for Function<A, R> {
72      fn eval(&self, x: A) -> R {
73          (self.f)(x)
74      }
75  }
```

```rust
pub struct Derivative<'a> {
    pub f: &'a dyn Func<f64, Complex64>,
}

impl Func<f64, Complex64> for Derivative<'_> {
    fn eval(&self, x: f64) -> Complex64 {
        derivative(&|x| self.f.eval(x), x)
    }
}
```

src/wave_function_builder.rs

```rust
use crate::wkb_wave_func::Phase;
use crate::*;
use std::sync::*;

pub enum ScalingType {
    Mul(Complex64),
    Renormalize(Complex64),
    None,
}

pub trait WaveFunctionPart: Func<f64, Complex64> + Sync + Send {
    fn range(&self) -> (f64, f64);
    fn as_func(&self) -> Box<dyn Func<f64, Complex64>>;
}

pub trait WaveFunctionPartWithOp: WaveFunctionPart {
    fn get_op(&self) -> Box<fn(Complex64) -> Complex64>;
    fn with_op(&self, op: fn(Complex64) -> Complex64) -> Box<dyn
        WaveFunctionPartWithOp>;
    fn as_wave_function_part(&self) -> Box<dyn WaveFunctionPart>;
}

pub fn is_in_range(range: (f64, f64), x: f64) -> bool {
    return range.0 <= x && range.1 > x;
}

#[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 WaveFunctionPart for Joint {
    fn range(&self) -> (f64, f64) {
        if self.delta > 0.0 {
```

```rust
                (self.cut, self.cut + self.delta)
        } else {
                (self.cut + self.delta, self.cut)
        }
    }
    fn as_func(&self) -> Box<dyn Func<f64, Complex64>> {
        return Box::new(self.clone());
    }
}

impl Func<f64, Complex64> for Joint {
    fn eval(&self, x: f64) -> Complex64 {
        let (left, right) = if self.delta > 0.0 {
            (&self.left, &self.right)
        } else {
            (&self.right, &self.left)
        };

        let delta = self.delta.abs();

        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) / delta);
    }
}

#[derive(Clone)]
struct PureWkb {
    wkb: Arc<WkbWaveFunction>,
    range: (f64, f64),
}

impl WaveFunctionPart for PureWkb {
    fn range(&self) -> (f64, f64) {
        self.range
    }
    fn as_func(&self) -> Box<dyn Func<f64, Complex64>> {
        Box::new(self.clone())
    }
}

impl WaveFunctionPartWithOp for PureWkb {
    fn as_wave_function_part(&self) -> Box<dyn WaveFunctionPart> {
        Box::new(self.clone())
    }

    fn get_op(&self) -> Box<fn(Complex64) -> Complex64> {
        self.wkb.get_op()
    }
}
```

```rust
 86
 87     fn with_op(&self, op: fn(Complex64) -> Complex64) -> Box<dyn
             WaveFunctionPartWithOp> {
 88         Box::new(PureWkb {
 89             wkb: Arc::new(self.wkb.with_op(op)),
 90             range: self.range,
 91         })
 92     }
 93 }
 94
 95 impl Func<f64, Complex64> for PureWkb {
 96     fn eval(&self, x: f64) -> Complex64 {
 97         self.wkb.eval(x)
 98     }
 99 }
100
101 #[derive(Clone)]
102 struct ApproxPart {
103     airy: Arc<AiryWaveFunction>,
104     wkb: Arc<WkbWaveFunction>,
105     airy_join_l: Joint,
106     airy_join_r: Joint,
107     range: (f64, f64),
108 }
109
110 impl WaveFunctionPart for ApproxPart {
111     fn range(&self) -> (f64, f64) {
112         self.range
113     }
114     fn as_func(&self) -> Box<dyn Func<f64, Complex64>> {
115         Box::new(self.clone())
116     }
117 }
118
119 impl WaveFunctionPartWithOp for ApproxPart {
120     fn as_wave_function_part(&self) -> Box<dyn WaveFunctionPart> {
121         Box::new(self.clone())
122     }
123
124     fn get_op(&self) -> Box<fn(Complex64) -> Complex64> {
125         self.wkb.get_op()
126     }
127
128     fn with_op(&self, op: fn(Complex64) -> Complex64) -> Box<dyn
             WaveFunctionPartWithOp> {
129         Box::new(ApproxPart::new(
130             self.airy.with_op(op),
131             self.wkb.with_op(op),
132             self.range,
```

```rust
133          ))
134      }
135  }
136
137  impl ApproxPart {
138      fn new(airy: AiryWaveFunction, wkb: WkbWaveFunction, range: (f64, f64)) ->
             ApproxPart {
139          let airy_rc = Arc::new(airy);
140          let wkb_rc = Arc::new(wkb);
141          let delta = (airy_rc.ts.1 - airy_rc.ts.0) * AIRY_EXTRA;
142          ApproxPart {
143              airy: airy_rc.clone(),
144              wkb: wkb_rc.clone(),
145              airy_join_l: Joint {
146                  left: wkb_rc.clone(),
147                  right: airy_rc.clone(),
148                  cut: airy_rc.ts.0 + delta / 2.0,
149                  delta: -delta,
150              },
151              airy_join_r: Joint {
152                  left: airy_rc.clone(),
153                  right: wkb_rc.clone(),
154                  cut: airy_rc.ts.1 - delta / 2.0,
155                  delta,
156              },
157              range,
158          }
159      }
160  }
161
162  impl Func<f64, Complex64> for ApproxPart {
163      fn eval(&self, x: f64) -> Complex64 {
164          if is_in_range(self.airy_join_l.range(), x) && ENABLE_AIRY_JOINTS {
165              return self.airy_join_l.eval(x);
166          } else if is_in_range(self.airy_join_r.range(), x) && ENABLE_AIRY_JOINTS {
167              return self.airy_join_r.eval(x);
168          } else if is_in_range(self.airy.ts, x) {
169              return self.airy.eval(x);
170          } else {
171              return self.wkb.eval(x);
172          }
173      }
174  }
175
176  #[derive(Clone)]
177  pub struct WaveFunction {
178      phase: Arc<Phase>,
179      view: (f64, f64),
180      parts: Vec<Arc<dyn WaveFunctionPart>>,
```

```rust
181        airy_ranges: Vec<(f64, f64)>,
182        wkb_ranges: Vec<(f64, f64)>,
183        scaling: Complex64,
184    }
185
186    fn sign_match(f1: f64, f2: f64) -> bool {
187        return f1.signum() == f2.signum();
188    }
189
190    fn sign_match_complex(mut c1: Complex64, mut c2: Complex64) -> bool {
191        if c1.re.abs() < c1.im.abs() {
192            c1.re = 0.0;
193        }
194
195        if c1.im.abs() < c1.re.abs() {
196            c1.im = 0.0;
197        }
198
199        if c2.re.abs() < c2.im.abs() {
200            c2.re = 0.0;
201        }
202
203        if c2.im.abs() < c2.re.abs() {
204            c2.im = 0.0;
205        }
206
207        return sign_match(c1.re, c2.re) && sign_match(c1.im, c2.im);
208    }
209
210    fn calc_phase_offset(phase: Arc<Phase>, (turn_left, turn_right): (f64, f64)) ->
           Option<f64> {
211        // return Some(f64::consts::PI / 4.0);
212        let critical_x = (turn_left + turn_right) / 2.0;
213        if (phase.potential)(critical_x) > phase.energy {
214            return None;
215        }
216
217        let int_left = integrate(
218            evaluate_function_between(phase.as_ref(), critical_x, turn_left, INTEG_STEPS)
                 ,
219            TRAPEZE_PER_THREAD,
220        );
221        let int_right = -integrate(
222            evaluate_function_between(phase.as_ref(), critical_x, turn_right, INTEG_STEPS
                 ),
223            TRAPEZE_PER_THREAD,
224        );
225
226        println!("left: {}, right: {}", int_left, int_right);
```

```rust
227        let phase_off = ((-int_left - int_right) / 2.0) % (2.0 * f64::consts::PI);
228
229        println!(
230            "phase_off / PI: {:.12}",
231            phase_off / f64::consts::PI
232        );
233
234        Some(phase_off)
235    }
236
237    pub fn find_best_op_wave_func_part(
238        phase: Arc<Phase>,
239        previous: &dyn WaveFunctionPartWithOp,
240        current: &dyn WaveFunctionPartWithOp,
241    ) -> fn(Complex64) -> Complex64 {
242        if !float_compare(current.range().0, previous.range().1, 1e-3) {
243            println!("current: ({}, {})", current.range().0, current.range().1);
244            println!("previous: ({}, {})", previous.range().0, previous.range().1);
245        }
246        assert!(float_compare(current.range().0, previous.range().1, 1e-3));
247        let boundary = current.range().0;
248
249        let deriv_prev = derivative(&|x| previous.eval(x), current.range().0);
250        let val_prev = previous.eval(current.range().0);
251        let deriv = derivative(&|x| current.eval(x), current.range().0);
252        let val = current.eval(boundary);
253
254        return if (phase.potential)(boundary) >= phase.energy {
255            *previous.get_op()
256        } else {
257            let conj_deriv = conjugate(deriv);
258            let conj_val = conjugate(val);
259            let neg_conj_deriv = negative_conj(deriv);
260            let neg_conj_val = negative_conj(val);
261            let neg_deriv = negative(deriv);
262            let neg_val = negative(val);
263
264            let conj_mse = (conj_deriv - deriv_prev).norm_sqr() + (conj_val - val_prev).
265                norm_sqr();
266            let neg_conj_mse =
267                (neg_conj_deriv - deriv_prev).norm_sqr() + (neg_conj_val - val_prev).
268                    norm_sqr();
269            let neg_mse = (neg_deriv - deriv_prev).norm_sqr() + (neg_val - val_prev).
270                norm_sqr();
271            let id_mse = (deriv - deriv_prev).norm_sqr() + (val - val_prev).norm_sqr();
272
273            if conj_mse <= neg_conj_mse && conj_mse <= neg_mse && conj_mse <= id_mse {
274                println!(
275                    "conjugate mse, conj: {}, neg conj: {}, neg: {}, id: {}",
```

```
273                     conj_mse, neg_conj_mse, neg_mse, id_mse
274                 );
275                 conjugate
276         } else if neg_conj_mse <= conj_mse && neg_conj_mse <= neg_mse && neg_conj_mse
                 <= id_mse {
277             println!(
278                 "negative␣conj␣mse,␣conj:␣{},␣neg␣conj:␣{},␣neg:␣{},␣id:␣{}",
279                 conj_mse, neg_conj_mse, neg_mse, id_mse
280             );
281             negative_conj
282         } else if neg_mse <= conj_mse && neg_mse <= neg_conj_mse && neg_mse <= id_mse
                 {
283             println!(
284                 "negative␣mse,␣conj:␣{},␣neg␣conj:␣{},␣neg:␣{},␣id:␣{}",
285                 conj_mse, neg_conj_mse, neg_mse, id_mse
286             );
287             negative
288         } else {
289             println!(
290                 "identity␣mse,␣conj:␣{},␣neg␣conj:␣{},␣neg:␣{},␣id:␣{}",
291                 conj_mse, neg_conj_mse, neg_mse, id_mse
292             );
293             identity
294         }
295     };
296 }
297
298 impl WaveFunction {
299     pub fn get_energy(&self) -> f64 {
300         self.phase.energy
301     }
302
303     pub fn new<F: Fn(f64) -> f64 + Sync + Send>(
304         potential: &'static F,
305         mass: f64,
306         n_energy: usize,
307         approx_inf: (f64, f64),
308         view_factor: f64,
309         scaling: ScalingType,
310     ) -> WaveFunction {
311         let energy = energy::nth_energy(n_energy, mass, &potential, approx_inf);
312
313         let lower_bound = newtons_method::newtons_method_max_iters(
314             &|x| potential(x) - energy,
315             approx_inf.0,
316             1e-7,
317             100000,
318         );
319         let upper_bound = newtons_method::newtons_method_max_iters(
```

```
320            &|x| potential(x) - energy,
321            approx_inf.1,
322            1e-7,
323            100000,
324        );
325
326        let view = if lower_bound.is_some() && upper_bound.is_some() {
327            (
328                lower_bound.unwrap() * view_factor,
329                upper_bound.unwrap() * view_factor,
330            )
331        } else {
332            println!("Failed␣to␣determine␣view␣automatically,␣using␣APPROX_INF␣as␣
                view");
333            approx_inf.clone()
334        };
335
336        let phase = Arc::new(Phase::new(energy, mass, potential));
337
338        let (airy_wave_funcs, boundaries) = AiryWaveFunction::new(phase.clone(), (
                view.0, view.1));
339        let (parts, airy_ranges, wkb_ranges): (
340            Vec<Arc<dyn WaveFunctionPart>>,
341            Vec<(f64, f64)>,
342            Vec<(f64, f64)>,
343        ) = if boundaries.ts.len() == 0 {
344            println!("No␣turning␣points␣found␣in␣view!␣Results␣might␣be␣in␣accurate")
                ;
345            let wkb1 = WkbWaveFunction::new(
346                phase.clone(),
347                1.0.into(),
348                INTEG_STEPS,
349                approx_inf.0,
350                calc_phase_offset(phase.clone(), approx_inf).unwrap_or(f64::consts::
                    PI / 4.0),
351            );
352            let wkb2 = WkbWaveFunction::new(
353                phase.clone(),
354                1.0.into(),
355                INTEG_STEPS,
356                approx_inf.1,
357                calc_phase_offset(phase.clone(), approx_inf).unwrap_or(f64::consts::
                    PI / 4.0),
358            );
359
360            let center = (view.0 + view.1) / 2.0;
361            let wkb1 = Box::new(PureWkb {
362                wkb: Arc::new(wkb1),
363                range: (approx_inf.0, center),
```

```rust
364            });
365
366            let wkb2 = Box::new(PureWkb {
367                wkb: Arc::new(wkb2),
368                range: (center, approx_inf.1),
369            });
370
371            let op = find_best_op_wave_func_part(phase.clone(), wkb1.as_ref(), wkb2.
                   as_ref());
372
373            let wkb1_range = wkb1.range();
374            let wkb2 = wkb2.with_op(op);
375            let delta = (view.1 - view.0) * WKB_TRANSITION_FRACTION;
376            (
377                if ENABLE_WKB_JOINTS {
378                    vec![
379                        Arc::new(Joint {
380                            left: Arc::from(wkb1.as_func()),
381                            right: Arc::from(wkb2.as_func()),
382                            cut: (view.0 + view.1) / 2.0 - delta / 2.0,
383                            delta: delta,
384                        }),
385                        Arc::from(wkb1.as_wave_function_part()),
386                        Arc::from(wkb2.as_wave_function_part()),
387                    ]
388                } else {
389                    vec![
390                        Arc::from(wkb1.as_wave_function_part()),
391                        Arc::from(wkb2.as_wave_function_part()),
392                    ]
393                },
394                vec![],
395                vec![wkb1_range, wkb2.range()],
396            )
397        } else {
398            let turning_points: Vec<f64> = [
399                vec![2.0 * approx_inf.0 - boundaries.ts.first().unwrap().1],
400                boundaries.ts.iter().map(|p| p.1).collect(),
401                vec![2.0 * approx_inf.1 - boundaries.ts.last().unwrap().1],
402            ]
403            .concat();
404
405            let wave_funcs = turning_points
406                .iter()
407                .zip(turning_points.iter().skip(1))
408                .zip(turning_points.iter().skip(2))
409                .map(
410                    |((previous, boundary), next)| -> (WkbWaveFunction, (f64, f64)) {
411                        (
```

```
412                        WkbWaveFunction::new(
413                            phase.clone(),
414                            1.0.into(),
415                            INTEG_STEPS,
416                            *boundary,
417                            calc_phase_offset(phase.clone(), (*previous, *
                                    boundary))
418                                .unwrap_or(f64::consts::PI / 4.0),
419                        ),
420                        ((boundary + previous) / 2.0, (next + boundary) / 2.0),
421                    )
422                },
423            )
424            .collect::<Vec<(WkbWaveFunction, (f64, f64))>>();

426        let wkb_airy_pair: Vec<(&(WkbWaveFunction, (f64, f64)), AiryWaveFunction)
                > = wave_funcs
427            .iter()
428            .zip(airy_wave_funcs.iter())
429            .map(|(w, a)| (w, a.with_phase_off(w.0.phase_off)))
430            .collect();

432        let wkb_ranges = wkb_airy_pair
433            .iter()
434            .map(|((_, wkb_range), _)| *wkb_range)
435            .collect();
436        let airy_ranges = wkb_airy_pair.iter().map(|(_, airy)| airy.ts).collect()
                ;

438        let approx_parts: Vec<Arc<dyn WaveFunctionPartWithOp>> = wkb_airy_pair
439            .iter()
440            .map(|((wkb, range), airy)| -> Arc<dyn WaveFunctionPartWithOp> {
441                Arc::new(ApproxPart::new(airy.clone(), wkb.clone(), *range))
442            })
443            .collect();

445        let mut approx_parts_with_op: Vec<Arc<dyn WaveFunctionPartWithOp>> =
446            vec![Arc::from(approx_parts.first().unwrap().with_op(identity))];
447        approx_parts_with_op.reserve(approx_parts.len() - 1);

449        for i in 0..(approx_parts.len() - 1) {
450            let part1 = &approx_parts[i];
451            let part2 = &approx_parts[i + 1];
452            let p2_with_op = part2.with_op(find_best_op_wave_func_part(
453                phase.clone(),
454                part1.as_ref(),
455                part2.as_ref(),
456            ));
457            approx_parts_with_op.push(Arc::from(p2_with_op));
```

```
458            }
459            let mut approx_parts_with_joints: Vec<Arc<dyn WaveFunctionPart>> = vec!
                   [];
460
461            if ENABLE_WKB_JOINTS {
462                for (prev, curr) in approx_parts_with_op
463                    .iter()
464                    .zip(approx_parts_with_op.iter().skip(1))
465                {
466                    assert!(float_compare(prev.range().1, curr.range().0, 1e-4));
467
468                    let distance = (f64::min(prev.range().1, view.1)
469                        - f64::max(prev.range().0, view.0))
470                        + (f64::min(curr.range().1, view.1) - f64::max(curr.range()
                               .0, view.0));
471                    let delta = distance * WKB_TRANSITION_FRACTION;
472                    let joint = Joint {
473                        left: Arc::from(prev.as_func()),
474                        right: Arc::from(curr.as_func()),
475                        cut: f64::min(prev.range().1, view.1) - delta / 2.0,
476                        delta,
477                    };
478
479                    println!("Joint in range: {:#?}, delta: {}", joint.range(), delta
                           );
480
481                    approx_parts_with_joints.push(Arc::new(joint));
482                }
483            }
484
485            approx_parts_with_joints = vec![
486                approx_parts_with_joints,
487                approx_parts_with_op
488                    .iter()
489                    .map(|p| Arc::from(p.as_wave_function_part()))
490                    .collect(),
491            ]
492            .concat();
493
494            (approx_parts_with_joints, airy_ranges, wkb_ranges)
495        };
496
497        match scaling {
498            ScalingType::Mul(s) => WaveFunction {
499                phase,
500                view,
501                parts,
502                airy_ranges,
503                wkb_ranges,
```

```rust
504                        scaling: s,
505                    },
506                ScalingType::None => WaveFunction {
507                        phase,
508                        view,
509                        parts,
510                        airy_ranges,
511                        wkb_ranges,
512                        scaling: complex(1.0, 0.0),
513                    },
514                ScalingType::Renormalize(s) => {
515                        let unscaled = WaveFunction {
516                            phase: phase.clone(),
517                            view,
518                            parts: parts.clone(),
519                            airy_ranges: airy_ranges.clone(),
520                            wkb_ranges: wkb_ranges.clone(),
521                            scaling: s,
522                        };
523                        let factor = renormalize_factor(&unscaled, approx_inf);
524                        WaveFunction {
525                            phase,
526                            view,
527                            parts,
528                            airy_ranges,
529                            wkb_ranges,
530                            scaling: s * factor,
531                        }
532                    }
533            }
534        }
535
536        pub fn calc_psi(&self, x: f64) -> Complex64 {
537            for part in self.parts.as_slice() {
538                if is_in_range(part.range(), x) {
539                    return part.eval(x);
540                }
541            }
542            panic!(
543                "[WkbWaveFunction::calc_psi] x out of range (x = {}, ranges: {:#?})",
544                x,
545                self.parts
546                    .iter()
547                    .map(|p| p.range())
548                    .collect::<Vec<(f64, f64)>>()
549            );
550        }
551
552        pub fn get_airy_ranges(&self) -> &[(f64, f64)] {
```

```rust
            self.airy_ranges.as_slice()
    }

    pub fn get_wkb_ranges(&self) -> &[(f64, f64)] {
        self.wkb_ranges.as_slice()
    }

    pub fn get_wkb_ranges_in_view(&self) -> Vec<(f64, f64)> {
        self.wkb_ranges
            .iter()
            .map(|range| {
                (
                    f64::max(self.get_view().0, range.0),
                    f64::min(self.get_view().1, range.1),
                )
            })
            .collect::<Vec<(f64, f64)>>()
    }

    pub fn is_wkb(&self, x: f64) -> bool {
        self.wkb_ranges
            .iter()
            .map(|r| is_in_range(*r, x))
            .collect::<Vec<bool>>()
            .contains(&true)
    }

    pub fn is_airy(&self, x: f64) -> bool {
        self.airy_ranges
            .iter()
            .map(|r| is_in_range(*r, x))
            .collect::<Vec<bool>>()
            .contains(&true)
    }

    pub fn get_view(&self) -> (f64, f64) {
        self.view
    }

    pub fn set_view(&mut self, view: (f64, f64)) {
        self.view = view
    }

    pub fn get_phase(&self) -> Arc<Phase> {
        self.phase.clone()
    }
}

impl Func<f64, Complex64> for WaveFunction {
```

```rust
    fn eval(&self, x: f64) -> Complex64 {
        self.scaling * self.calc_psi(x)
    }
}

pub struct SuperPosition {
    wave_funcs: Vec<WaveFunction>,
    scaling: Complex64,
}

impl SuperPosition {
    pub fn new<F: Fn(f64) -> f64 + Send + Sync>(
        potential: &'static F,
        mass: f64,
        n_energies_scaling: &[(usize, Complex64)],
        approx_inf: (f64, f64),
        view_factor: f64,
        scaling: ScalingType,
    ) -> SuperPosition {
        let wave_funcs = n_energies_scaling
            .iter()
            .map(|(e, scale)| {
                let wave = WaveFunction::new(
                    potential,
                    mass,
                    *e,
                    approx_inf,
                    view_factor,
                    ScalingType::Mul(*scale),
                );
                println!("Calculated Energy {}\n", *e);
                return wave;
            })
            .collect();

        match scaling {
            ScalingType::Mul(s) => SuperPosition {
                wave_funcs,
                scaling: s,
            },
            ScalingType::None => SuperPosition {
                wave_funcs,
                scaling: 1.0.into(),
            },
            ScalingType::Renormalize(s) => {
                let unscaled = SuperPosition {
                    wave_funcs: wave_funcs.clone(),
                    scaling: s,
                };
```

```rust
651                let factor = renormalize_factor(&unscaled, approx_inf);
652                println!("factor: {}", factor);
653                SuperPosition {
654                    wave_funcs,
655                    scaling: s * factor,
656                }
657            }
658        }
659    }
660
661    pub fn get_view(&self) -> (f64, f64) {
662        let view_a = self
663            .wave_funcs
664            .iter()
665            .map(|w| w.get_view().0)
666            .min_by(cmp_f64)
667            .unwrap();
668        let view_b = self
669            .wave_funcs
670            .iter()
671            .map(|w| w.get_view().1)
672            .max_by(cmp_f64)
673            .unwrap();
674        (view_a, view_b)
675    }
676 }
677
678 impl Func<f64, Complex64> for SuperPosition {
679     fn eval(&self, x: f64) -> Complex64 {
680         self.scaling * self.wave_funcs.iter().map(|w| w.eval(x)).sum::<Complex64>()
681     }
682 }
683
684 struct Scaled<A, R>
685 where
686     R: std::ops::Mul<R, Output = R> + Sync + Send + Clone,
687 {
688     scale: R,
689     func: Box<dyn Func<A, R>>,
690 }
691
692 impl<A, R> Func<A, R> for Scaled<A, R>
693 where
694     R: std::ops::Mul<R, Output = R> + Sync + Send + Clone,
695 {
696     fn eval(&self, x: A) -> R {
697         self.func.eval(x) * self.scale.clone()
698     }
699 }
```

```rust
700
701  fn renormalize_factor(wave_func: &dyn Func<f64, Complex64>, approx_inf: (f64, f64))
         -> f64 {
702      1.0 / integrate(
703          evaluate_function_between(
704              wave_func,
705              approx_inf.0 * (1.0 - f64::EPSILON),
706              approx_inf.1 * (1.0 - f64::EPSILON),
707              INTEG_STEPS,
708          )
709          .par_iter()
710          .map(|p| Point {
711              x: p.x,
712              y: p.y.norm_sqr(),
713          })
714          .collect(),
715          TRAPEZE_PER_THREAD,
716      )
717  }
718
719  pub fn renormalize(
720      wave_func: Box<dyn Func<f64, Complex64>>,
721      approx_inf: (f64, f64),
722  ) -> Box<dyn Func<f64, Complex64>> {
723      let area = renormalize_factor(wave_func.as_ref(), approx_inf);
724      return Box::new(Scaled::<f64, Complex64> {
725          scale: area.into(),
726          func: wave_func,
727      });
728  }
729
730  #[cfg(test)]
731  mod test {
732      use super::*;
733
734      #[test]
735      fn sign_check_complex_test() {
736          let range = (-50.0, 50.0);
737          let n = 100000;
738          for ri1 in 0..n {
739              for ii1 in 0..n {
740                  for ri2 in 0..n {
741                      for ii2 in 0..n {
742                          let re1 = index_to_range(ri1 as f64, 0.0, n as f64, range.0,
                                 range.1);
743                          let im1 = index_to_range(ii1 as f64, 0.0, n as f64, range.0,
                                 range.1);
744                          let re2 = index_to_range(ri2 as f64, 0.0, n as f64, range.0,
                                 range.1);
```

```
745                    let im2 = index_to_range(ii2 as f64, 0.0, n as f64, range.0,
                           range.1);
746
747                    assert_eq!(
748                        sign_match_complex(complex(re1, im1), complex(re2, im2)),
749                        sign_match_complex(complex(re2, im2), complex(re1, im1))
750                    );
751                }
752            }
753        }
754    }
755    }
756 }
```

src/wkb_wave_func.rs

```
1  use crate::*;
2  use std::fmt::Display;
3  use std::sync::Arc;
4
5  #[derive(Clone)]
6  pub struct Phase {
7      pub energy: f64,
8      pub mass: f64,
9      pub potential: Arc<dyn Fn(f64) -> f64 + Send + Sync>,
10 }
11
12 impl Display for Phase {
13     fn fmt(&self, f: &mut std::fmt::Formatter<'_>) -> std::fmt::Result {
14         write!(
15             f,
16             "Phase {{energy: {}, mass: {}, potential: [func]}}",
17             self.energy, self.mass
18         )
19     }
20 }
21
22 impl Phase {
23     fn default() -> Phase {
24         Phase {
25             energy: 0.0,
26             mass: 0.0,
27             potential: Arc::new(|_x| 0.0),
28         }
29     }
30
31     pub fn new<F: Fn(f64) -> f64 + Sync + Send>(
32         energy: f64,
33         mass: f64,
```

```rust
            potential: &'static F,
        ) -> Phase {
            return Phase {
                energy,
                mass,
                potential: Arc::new(potential),
            };
        }

        fn sqrt_momentum(&self, x: f64) -> f64 {
            self.eval(x).abs().sqrt()
        }
    }

    impl Func<f64, f64> for Phase {
        fn eval(&self, x: f64) -> f64 {
            (2.0 * self.mass * ((self.potential)(x) - self.energy))
                .abs()
                .sqrt()
        }
    }

    #[derive(Clone)]
    pub struct WkbWaveFunction {
        pub c: Complex64,
        pub turning_point: f64,
        pub phase: Arc<Phase>,
        integration_steps: usize,
        op: fn(Complex64) -> Complex64,
        pub phase_off: f64,
    }

    impl WkbWaveFunction {
        pub fn get_c(&self) -> Complex64 {
            self.c
        }

        pub fn with_c(&self, c: Complex64) -> WkbWaveFunction {
            WkbWaveFunction {
                c,
                turning_point: self.turning_point,
                phase: self.phase.clone(),
                integration_steps: self.integration_steps,
                op: self.op,
                phase_off: self.phase_off,
            }
        }

        pub fn new(
```

```rust
        phase: Arc<Phase>,
        c: Complex64,
        integration_steps: usize,
        turning_point: f64,
        phase_off: f64,
    ) -> WkbWaveFunction {
        return WkbWaveFunction {
            c,
            turning_point,
            phase: phase.clone(),
            integration_steps,
            op: identity,
            phase_off,
        };
    }

    pub fn with_op(&self, op: fn(Complex64) -> Complex64) -> WkbWaveFunction {
        return WkbWaveFunction {
            c: self.c,
            turning_point: self.turning_point,
            phase: self.phase.clone(),
            integration_steps: self.integration_steps,
            op,
            phase_off: self.phase_off,
        };
    }

    pub fn get_op(&self) -> Box<fn(Complex64) -> Complex64> {
        Box::new(self.op)
    }
}

impl Func<f64, Complex64> for WkbWaveFunction {
    fn eval(&self, x: f64) -> Complex64 {
        let integral = integrate(
            evaluate_function_between(
                self.phase.as_ref(),
                x,
                self.turning_point,
                self.integration_steps,
            ),
            TRAPEZE_PER_THREAD,
        );

        let val = if self.phase.energy < (self.phase.potential)(x) {
            if x < self.turning_point {
                (self.c * 0.5 * (-integral.abs()).exp())
                    * if COMPLEX_EXP_WKB {
                        complex((self.phase_off).cos(), (self.phase_off).sin())
```

93

```
132                        / self.phase.sqrt_momentum(x)
133                    } else {
134                        1.0.into()
135                    }
136            } else {
137                (self.c * 0.5 * (-integral.abs()).exp())
138                    * if COMPLEX_EXP_WKB {
139                        complex((self.phase_off).cos(), (self.phase_off).sin())
140                            / self.phase.sqrt_momentum(x)
141                    } else {
142                        1.0.into()
143                    }
144            }
145        } else {
146            if x < self.turning_point {
147                self.c
148                    * complex(
149                        (-integral + self.phase_off).cos(),
150                        if COMPLEX_OSZ_WKB {
151                            (-integral + self.phase_off).sin()
152                        } else {
153                            0.0
154                        },
155                    )
156                    / self.phase.sqrt_momentum(x)
157            } else {
158                self.c
159                    * complex(
160                        (integral + self.phase_off).cos(),
161                        if COMPLEX_OSZ_WKB {
162                            (integral + self.phase_off).sin()
163                        } else {
164                            0.0
165                        },
166                    )
167                    / self.phase.sqrt_momentum(x)
168            }
169        };
170
171        return (self.op)(val);
172    }
173 }
174
175 #[cfg(test)]
176 mod test {
177    use super::*;
178    use std::cmp::Ordering;
179
180    fn pot(x: f64) -> f64 {
```

```rust
181            1.0 / (x * x)
182        }
183
184        fn pot_in(x: f64) -> f64 {
185            1.0 / x.sqrt()
186        }
187
188        #[test]
189        fn phase_off() {
190            let energy_cond = |e: f64| -> f64 { (0.5 * (e - 0.5)) % 1.0 };
191
192            let integ = Function::<f64, f64>::new(energy_cond);
193            let mut values = evaluate_function_between(&integ, 0.0, 5.0, NUMBER_OF_POINTS
                   );
194            let sort_func =
195                |p1: &Point<f64, f64>, p2: &Point<f64, f64>| -> Ordering { cmp_f64(&p1.x,
                       &p2.x) };
196            values.sort_by(sort_func);
197
198            let mut data_file = File::create("energy.txt").unwrap();
199
200            let data_str: String = values
201                .par_iter()
202                .map(|p| -> String { format!("{}␣{}\n", p.x, p.y) })
203                .reduce(|| String::new(), |s: String, current: String| s + &*current);
204
205            data_file.write_all((data_str).as_ref()).unwrap()
206        }
207    }
```

lib/build.sh

```bash
1  #! /bin/bash
2
3  go get main
4  go build -o libairy.a -buildmode=c-archive main.go
```

lib/go.mod

```
1  module main
2
3  go 1.18
4
5  require gonum.org/v1/gonum v0.11.0
```

lib/main.go

```go
1  package main
```

```
 2
 3  import "C"
 4  import "gonum.org/v1/gonum/mathext"
 5
 6  //export airy_ai
 7  func airy_ai(zr float64, zi float64) (float64, float64) {
 8      z := mathext.AiryAi(complex(zr, zi))
 9      return real(z), imag(z)
10  }
11
12  func main() {
13
14  }
```

build.rs

```rust
 1  use std::env;
 2  use std::path::PathBuf;
 3  use std::process::Command;
 4
 5  fn main() {
 6      Command::new("sh")
 7          .arg("build.sh")
 8          .current_dir("lib/")
 9          .status()
10          .unwrap();
11
12      let path = "./lib";
13      let lib = "airy";
14
15      println!("cargo:rustc-link-search=native={}", path);
16      println!("cargo:rustc-link-lib=static={}", lib);
17
18      // The bindgen::Builder is the main entry point
19      // to bindgen, and lets you build up options for
20      // the resulting bindings.
21      let bindings = bindgen::Builder::default()
22          // The input header we would like to generate
23          // bindings for.
24          .header("lib/libairy.h")
25          // Tell cargo to invalidate the built crate whenever any of the
26          // included header files changed.
27          .parse_callbacks(Box::new(bindgen::CargoCallbacks))
28          // Finish the builder and generate the bindings.
29          .generate()
30          // Unwrap the Result and panic on failure.
31          .expect("Unable to generate bindings");
32
33      // Write the bindings to the $OUT_DIR/bindings.rs file.
```

```
34      let out_path = PathBuf::from(env::var("OUT_DIR").unwrap());
35      bindings
36          .write_to_file(out_path.join("bindings.rs"))
37          .expect("Couldn't␣write␣bindings!");
38  }
```

Cargo.toml

```
1   [package]
2   name = "shroedinger_approx"
3   version = "0.1.0"
4   edition = "2021"
5
6   # See more keys and their definitions at https://doc.rust-lang.org/cargo/reference/
          manifest.html
7
8   [dependencies]
9   num = "0.4.0"
10  tokio = { version = "1.0.3", features = ["full"] }
11  rayon = "1.5.3"
12  scilib = "0.5.0"
13
14  [build-dependencies]
15  bindgen = "0.60.1"
```

energy.wsl

```
1   m = 1
2   V[x_] = x^2
3
4   nthEnergy[n_] = Module[{energys, energy},
5       energys =  Solve[Integrate[Sqrt[2*m*(en - V[x])], {x, -Sqrt[en], Sqrt[en]}] == 2*
            Pi*(n + 1/2), en] // N;
6       energy = en /. energys[[1]];
7       energy
8       ]
9
10  energys = Table[{n, N@nthEnergy[n]}, {n, 0, 50}]
11
12  csv = ExportString[energys, "CSV"]
13  csv = StringReplace[csv, "," -> " "]
14  Export["output/energys_exact.dat", csv]
```

exact.wsl

```
1   c1 = -5.0
2   c2 = 1.0
3   numberOfPoints = 1000
```

```
 4  m = 2
 5  n = 5
 6  viewFactor = 1.5
 7
 8  V[x_] := x^2
 9
10  energys = Solve[Integrate[Sqrt[2*m*(en - V[x])], {x, -Sqrt[en], Sqrt[en]}] == 2*Pi*(n
          + 1/2), en] // N
11  energy = en /. energys[[1]]
12
13  view = Solve[energy == V[x], x]
14  view = Function[l, x /. l] /@ view
15  view = Function[x, x*viewFactor] /@ view
16
17
18  Print["Energy = ", energy]
19  Print["view = ", view]
20
21
22  solution := DSolve[{V[x] psi[x] - psi''[x]/(2 m) == energy psi[x]}, psi[x], x]
23  psi[x_] = psi[x] /. solution[[1]] /. C[1] -> c1 /. C[2] -> c2
24
25  Print["psi[x] = ", psi[x]]
26
27  (*psi[x_] = c2*ParabolicCylinderD[(-1 - 50*Sqrt[m])/2, *)
28          (*I*2^(3/4)*m^(1/4)*x] + c1*ParabolicCylinderD[(-1 + 50*Sqrt[m])/2, *)
29          (*2^(3/4)*m^(1/4)*x]*)
30
31
32
33  step = (Abs[view[[1]]] + Abs[view[[2]]]) / numberOfPoints
34
35
36  vals = Table[{x, N@psi[x]}, {x, view[[1]], view[[2]], step}]
37  vals = Function[p, {p[[1]], Re[p[[2]]], Im[p[[2]]]}] /@ vals
38  Print["psi[0] = ", psi[0]]
39
40  total = N@Integrate[Re[psi[x]]^2 + Im[psi[x]]^2, {x, -Sqrt[energy], Sqrt[energy]}]
41
42  Print["area under solution = ", total]
43  total = N@Integrate[Abs[psi[x]], {x, -Sqrt[energy], Sqrt[energy]}]
44  Print["area under solution after renormalization = ", N@Integrate[Re[psi[x]]^2 + Im[
          psi[x]]^2, {x, -Sqrt[energy], Sqrt[energy]}]]
45
46  vals = Function[p, {p[[1]], p[[2]] / total, p[[3]] / total}] /@ vals
47
48  csv = ExportString[vals, "CSV"]
49  csv = StringReplace[csv, "," -> " "]
50  Export["output/exact.dat", csv]
```

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