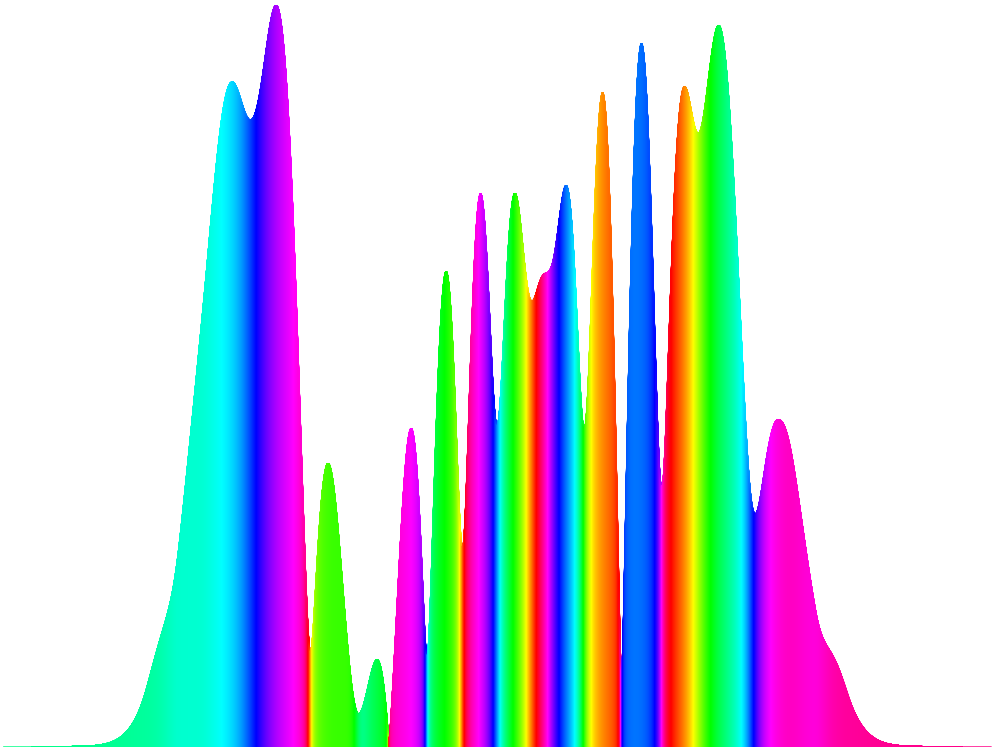
Approximating Solutions of the Time Independent Schrödinger Equation

Gian Laager
November 14, 2022



Maturaarbeit
Kantonsschule Glarus

Supervisor: 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, Quantenmechanik 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* (Schrödinger, 1935).

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 converge to 0 as x goes to $\pm\infty$. The program should be reasonably fast, meaning that for simple potentials and low energies it should be done in under 1 minute. The architecture should be able to support improvements.

Making the program user friendly is not a main focus. Meaning that a clear and simple API that can be extended in the future is enough. Even dough the user will have to edit the code to for example change between energies.

The program should also follow the UNIX philosophy, “do one thing and one thing well”. As a consequence the program will only do the calculations and not the plotting. But it provides a simple and clear interface for a plotting program such as GNU Plot.

The main focus will be to balance performance and accuracy. Accuracy manly meaning that the visualizations should be visually accurate and give some insight into quantum mechanics. The user should also be able to tune the balance between performance and accuracy to some degree.

2. Preliminary

2.1. Schrödinger Equation

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

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

Or more general

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

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

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

or

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

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

2.2. Rust

Rust is one of the newer programming languages and attempts to replace C/C++ which are notoriously difficult to work with. It supports both functional and object-oriented paradigms. It is much safer in terms of memory and promises the same performance as C. One of the goals of Rust is fearless concurrency which means everybody should be able to write concurrent

code without deadlocks and data races. This means calculations can utilize the full potential of the CPU without countless hours of debugging.

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

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

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

2.3. Interpretation of Quantum Mechanics

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

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

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

2.4. Complex Numbers

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

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

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

The complex plane is similar to the real number line, every complex number can be represented on this plane where Re(z) is the x-coordinate and 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. Reading Complex Plots

In the case of this paper we will try to plot a function

$$f : \mathbb{R} \to \mathbb{C}.$$

This means that the resulting plot will be 3 dimensional. The two main visualizations would be just to plot a 3D surface or add the phase of the complex number as a color. The first method works better if one can interact with the structure. However this is not really possible in a PDF or on a paper. This is why both options will be implemented.

For example on the next page you have the plot from the title page as a 3D plot in comparison to the color plot.

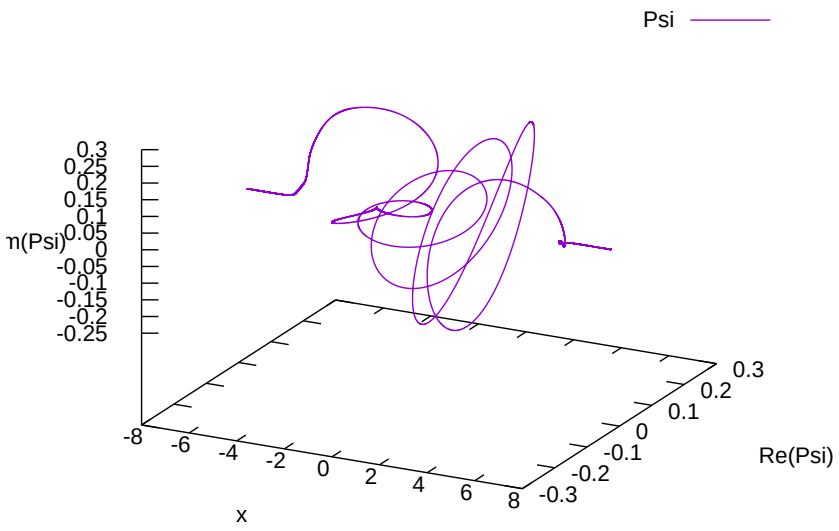


Figure 2.1.: 3D plot of super position of $V(x) = x^2$ and energies 9, 12 and 15

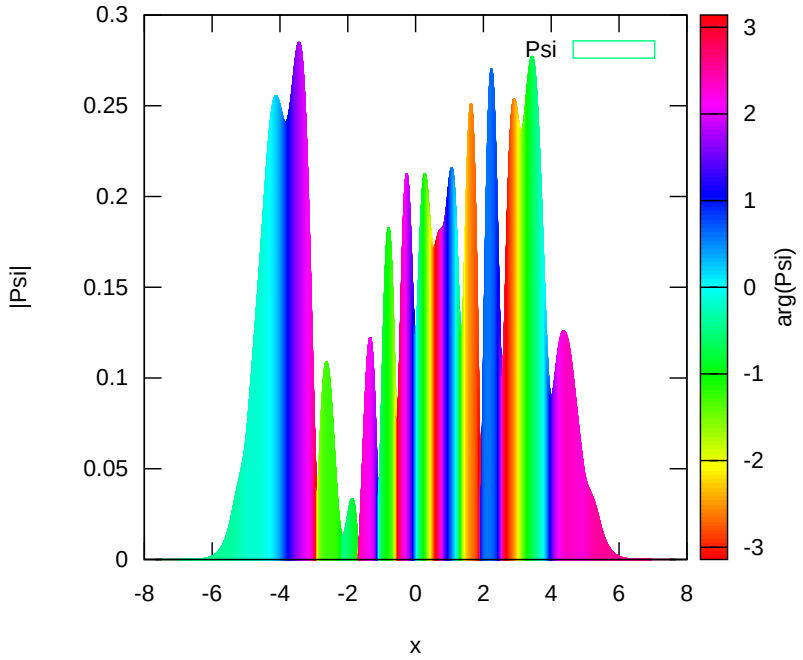


Figure 2.2.: Color plot of super position of $V(x) = x^2$ and energies 9, 12 and 15

As we can see the 3D plot is basically unusable since there is no depth and lighting would also be very difficult to do on a line such that we could actually see the depth. In the color plot every thing seems to be clear. Except that the values of the phase can't be read exactly.

When plotting the function yourself I would still recommend the 3D plot because it's clearer when you can move it around.

To get used this representation, have a look at figure 2.6.

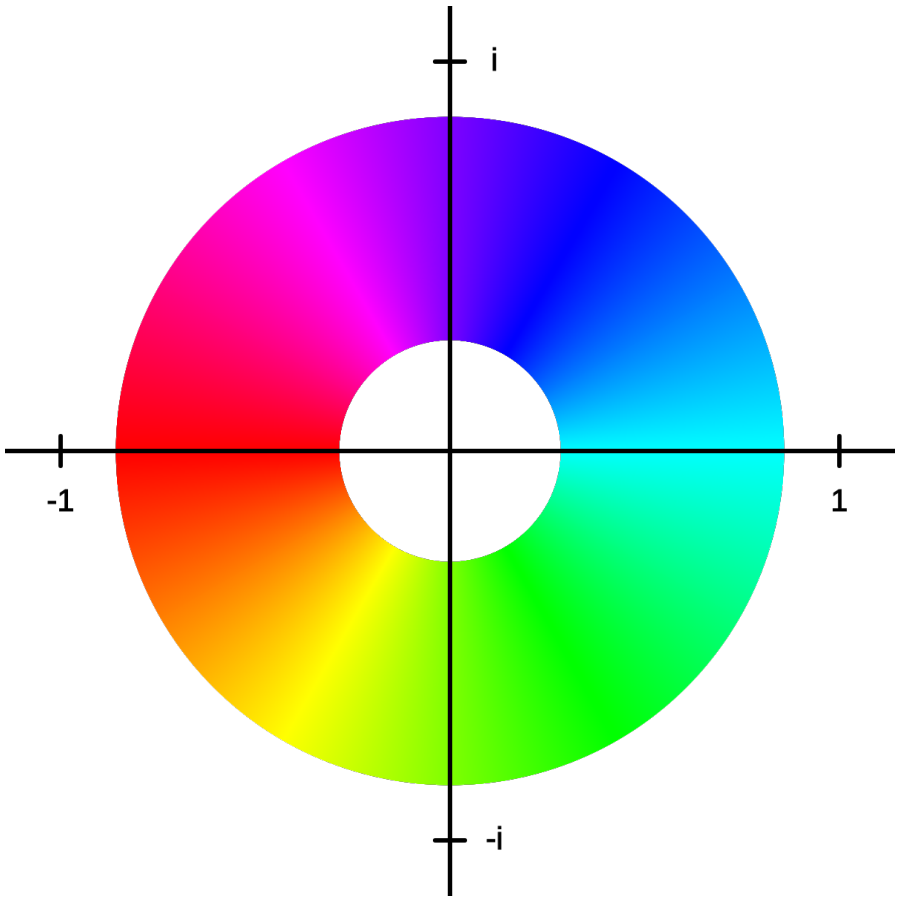


Figure 2.3.: Illustration of the mapping of an angle to color to represent complex numbers

Here we can see that we can assign a color to every argument of a complex number can be assigned a distinct color. For example $1+0i$ would have the color cyan and $0+1i$ would be purple. This graph is a handy tool to read the color plots that will be done later since it's easier to associate the angle with the color if the color is actually at that angle rather then displaying it radians.

2.7. 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. In Appendix C is a UML diagram of the architecture. Since current version of Rust does not support manual implementations of `std::ops::Fn` we have to define our own trait for functions `Func<A, R>` where `A` is the type of the argument and `R` is the return type. Later we will use this trait to implement functions for integration,evaluation and more useful utilities.

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

3.2. Newtons Method

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

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

The sequence is defined as

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

Visually this looks like figure 3.1 where $f(x) = (x-2)(x-1)(x+1)$ was taken as an example.

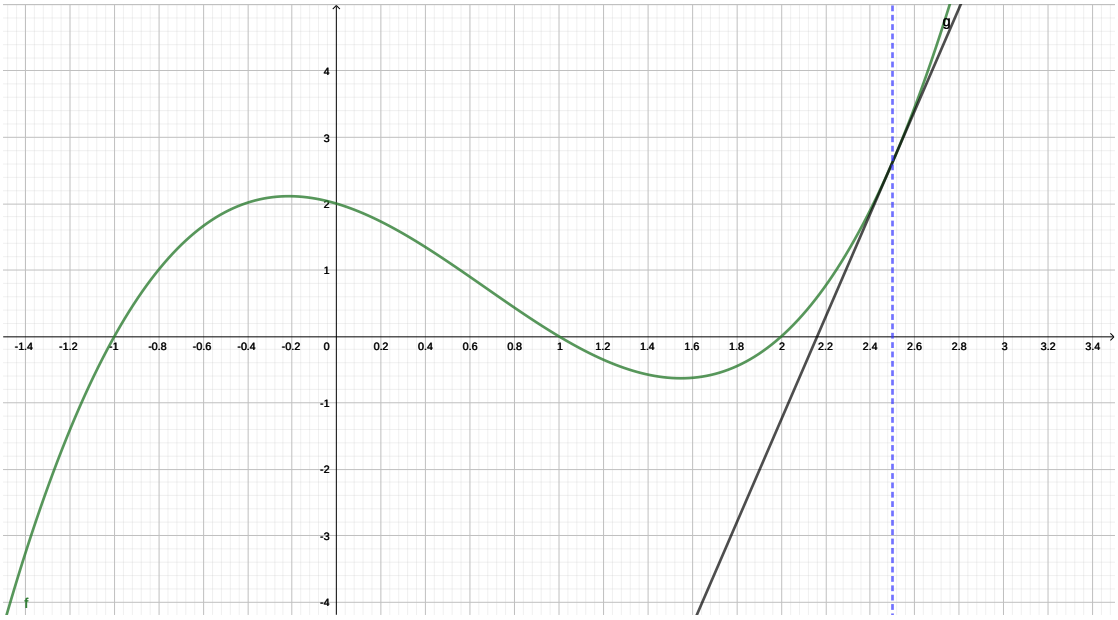


Figure 3.1.: Illustration of Newtons method, $f(x) = (x-1)(x-2)(x+1)$.

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.

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

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

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

3.3. Regula Falsi with Bisection

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

The algorithm itself is quite simple. To start we need

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

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

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

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

3.4. Derivatives

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

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

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

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

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

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

3.5. Integration

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

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

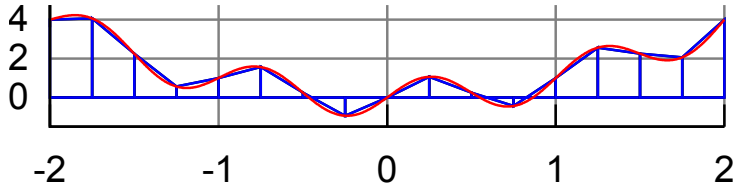


Figure 3.2.: Illustration of integration with trapeze from Wkipedia (2022).

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

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

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

The general structure of the algorithm can very easily be run in parallel since it doesn't matter in which order the segments are added together and the segments also don't dependent on one another. In Rust this is implemented using rayon. Rayon is an implementation for parallel iterators meaning that normal data structures that implement `std::iter` can be run in parallel *just* by changing `::iter()` to `::par_iter()`. This might not work in all cases because of memory safety but in this case the borrow checker will throw an error and the code wont compile.

```
1 pub trait Func<A, R>: Sync + Send {
2     fn eval(&self, x: A) -> R;
3 }
4
5 pub struct Point {
6     pub x: f64,
```

```
    pub y: Complex64,
}
```

Such that functions with states, like wave functions that store parameters, can be integrated there is a trait `Func<A, R>`.

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

```
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![];
    }

    (0..n)
        .into_par_iter()
        .map(|i| {
            index_to_range(
                X::from(i as f64),
                X::from(0.0_f64),
                X::from((n - 1) as f64),
                a,
                b,
            )
        })
        .map(|x: X| Point { x, y: f.eval(x) })
        .collect()
}
```

`Func<X, Y>` can be passed to `evaluate_function_between` it calculates `n` points between an interval from `a` to `b` and returns a vector of `Point`. `X` and `Y` are general data types such that it supports as many types of numbers as possible.

```rust
pub fn integrate<
    X: Sync + std::ops::Add<Output = X> + std::ops::Sub<Output = X> + Copy,
    Y: Default
        + Sync
        + std::ops::AddAssign
        + std::ops::Div<f64, Output = Y>
        + std::ops::Mul<Output = Y>
        + std::ops::Add<Output = Y>
        + Send
        + std::iter::Sum<Y>
        + 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;
}
```

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.

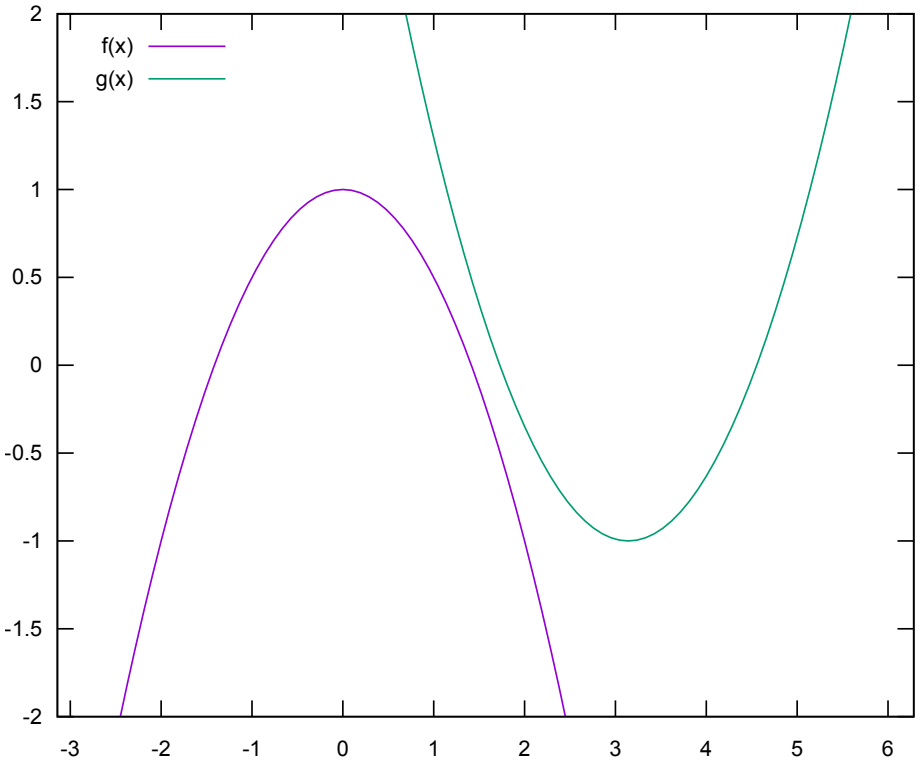


Figure 3.3.: Example for joining functions

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.

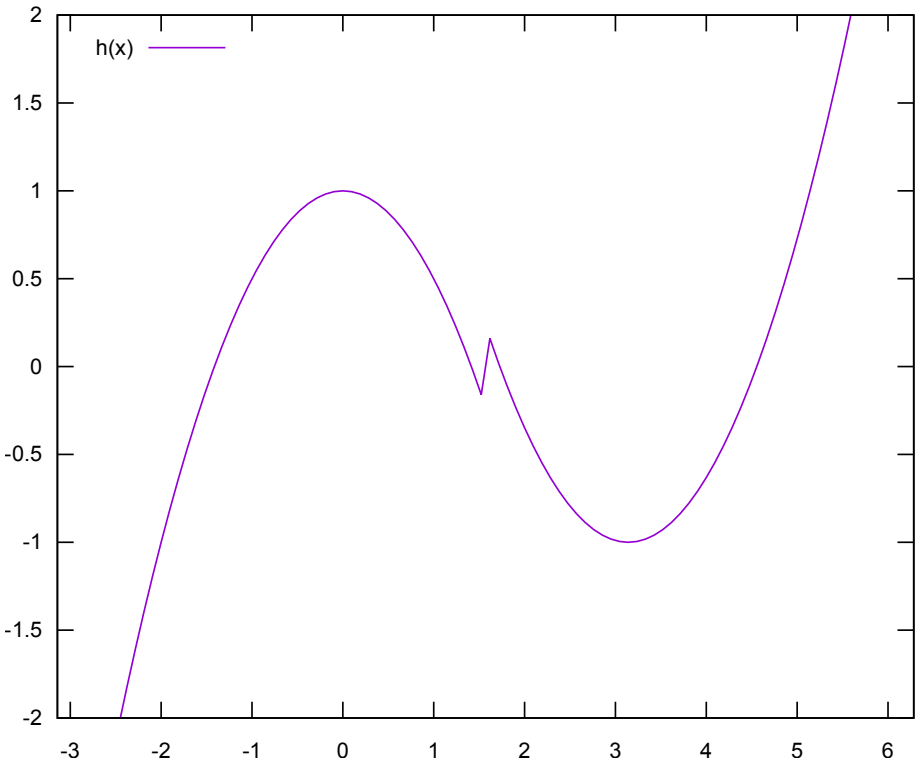


Figure 3.4.: Plot of h(x) with step joint

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

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

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).

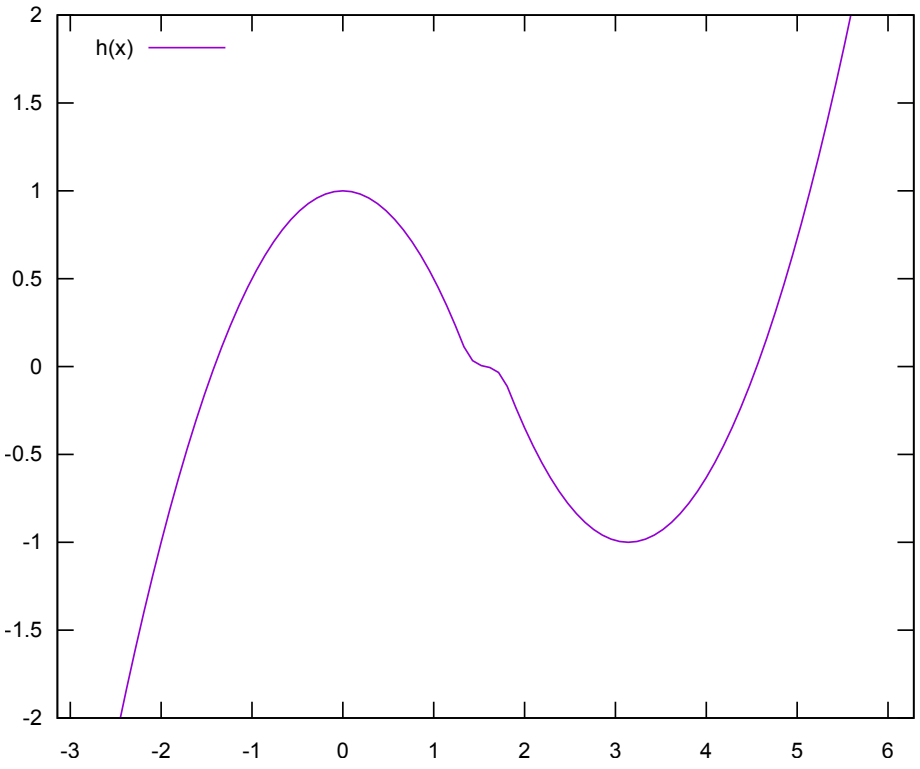


Figure 3.5.: Plot of h(x) with Hall joint

3.6.1. Implementation in Rust

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

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

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

```
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.

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

```
8
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

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

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

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

```
    let values = (0..=50)
        .into_iter()
        .map(|n: usize| Point::<usize, f64> {
            x: n,
            y: energy::nth_energy(n, 1.0, &potentials::square, APPROX_INF),
        })
        .collect::<Vec<Point<usize, f64>>>();

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

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

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

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

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

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

These programs will output two files `energies_approx.dat` (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 $\pm\frac{10}{64000} \approx \pm 1.56 \cdot 10^{-4}$, because the program checks for energies with that step size.

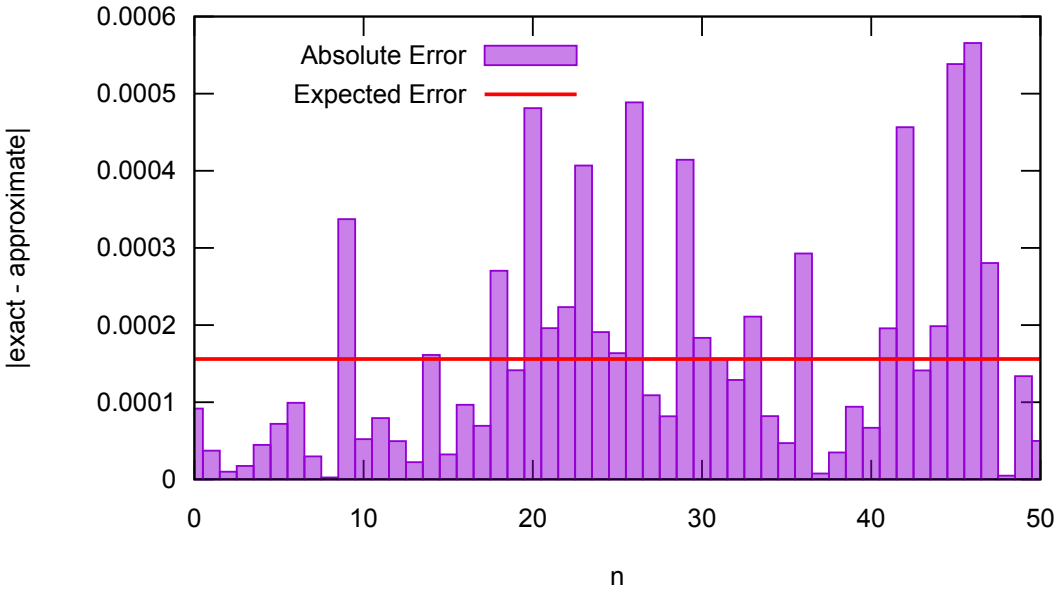


Figure 4.1.: Absolute error of energy levels in square potential

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

4.2. Approximation Scheme

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

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

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

p. 305).

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

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

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

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

$$\psi_{osc}^{WKB}(x) = \frac{c_1}{\sqrt{p(x)}} \cos\left(\int_x^t p(y)dy + \delta \right) \tag{4.9}$$

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

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

Since equation 4.7 might have more than one solution for turning points t, we have to consider each one of them individually and in the end join them into one function.

The factor of $1/2$ in equation 4.8 is analogous to (Littlejohn, 2020, eq. 92). This means that it's only valid if the turning points aren't "too close together" (Littlejohn, 2020). This will be a problem later when we look at some solutions. Littlejohn (2020) also mansions that there are extensions to WKB that can handle these cases. It would be interesting to add those to the program in the future.

Unfortunately there seems to be some kind of error in equation 4.9 when two different turning points are used the result at least according to Hall (2013) should be the same. But there functions did not join nicely in the middle of the two turning points. To maintain smoothness only one turning point was there for used. This issue will be discussed later in section 4.5.

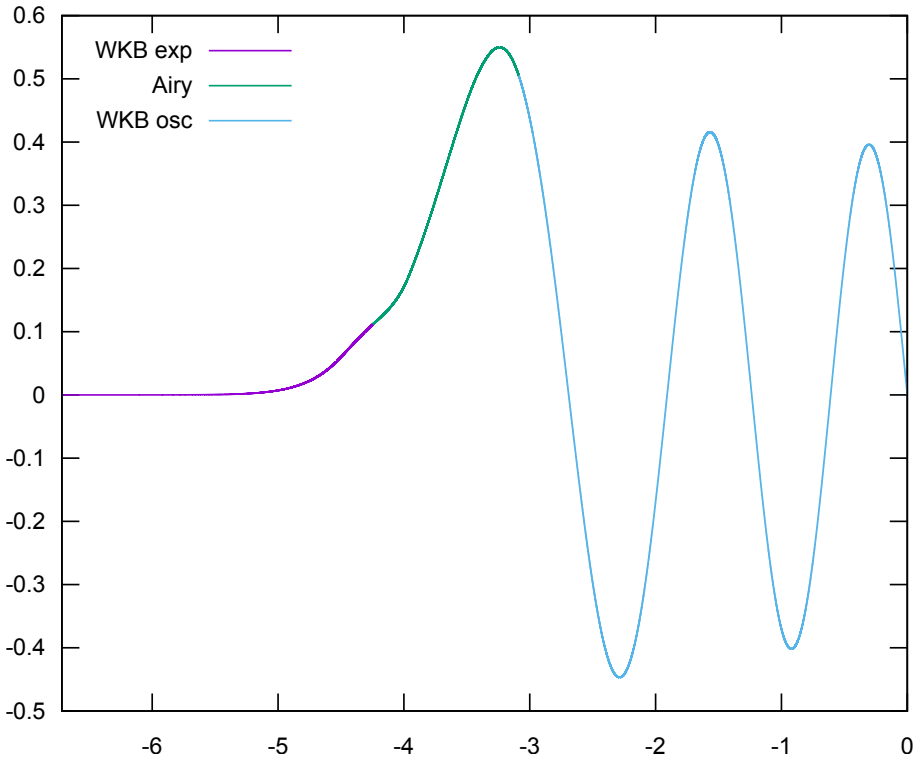


Figure 4.2.: Left half of wave function with $N_{Energy} = 9 \Rightarrow E \approx 13.4$, $m = 2$, $V(x) = x^2$

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 calculated according to (Hall, 2013, p. 317, Claim 15.7) 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 $x > t$.

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

WKB

`WkbWaveFunction` implements equations 4.8 and 4.9. For this we will create two functions `psi_osc` and `psi_exp`. We will use `psi_osc` if x is inside the classically allowed region and otherwise we will use `psi_exp`.

```
fn eval(&self, x: f64) -> Complex64 {
    let val = if self.phase.energy < (self.phase.potential)(x) {
        self.psi_exp(x)
    } else {
        self.psi_osc(x)
    };

    return (self.op)(val);
}
```

The term `self.op` has been an attempt to use both turning points for the oscillating region. It wasn't removed because the current method is not perfect either and it can be used in the future to improve the accuracy.
In the exponential part we will always use the corresponding turning point and because we're working with two separate turning points in the same function it is possible that the sign of the exponential part doesn't match with the sign of the exponential part. To fix this, we can define the `get_exp_sign` function.

```
pub fn get_exp_sign(&self) -> f64 {
    let limit_sign = if self.turning_point_exp == self.turning_point_osc {
        1.0
    } else {
        -1.0
    };

    (self.psi_osc(self.turning_point_exp + limit_sign * f64::EPSILON.sqrt()) / self.c
        )
        .re
        .signum()
}
```

It calculates the limit of the sign of the oscillating region as x approaches the turning point.

$$\operatorname{sgn}\left(\lim_{x\to t^{\pm}} \psi_{osc}^{WKB}(x)\right)$$

Airy

The constructor`AiryWaveFunction::new` calculates all the turning points in the view and then creates an `AiryWaveFunction` for each of them. These functions are then returned as a pair of the instance and the corresponding turning point.

Just like in the case of the WKB functions, the Airy implementation also implements the `self.op` which can be used to implement the osculating region with two turning points

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

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

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 factor of a is used to emulate the behavior of $\ll$.

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.
To later plot the wave function we will define the so called "view". This is the interval which the user will see in the end. It is defined to be

$$t_l < t_r$$
$$(t_l - f_{view}(t_r - t_l), t_r - f_{view}(t_r - t_l))$$

where t_l is the left and t_r the right most turning point. f_{view} is a user defined constant. These two points will be calculated by applying Newtons method to $V(x)-E$ with initial guesses at `APPROX_INF`.

Further on since we check for roots inside the interval of the view, 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

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

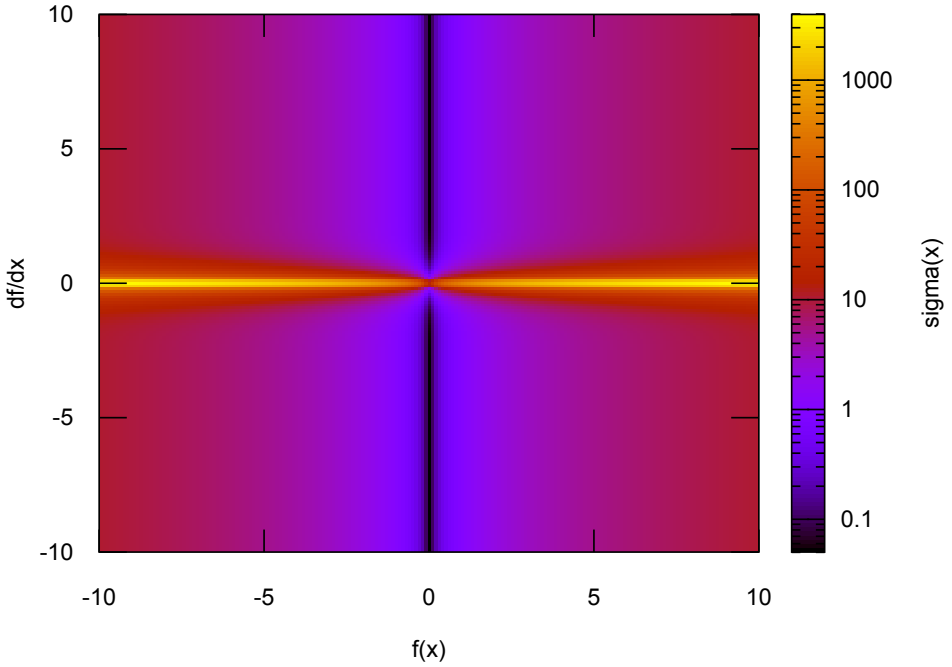


Figure 4.3.: Logarithmic heat diagram of $\sigma(x)$, darker/bluer is better

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.

```rust
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));
    });

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

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

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

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

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

4.4. Wave Function Parts

All the equations of the WKB approximation split into multiple parts. This is also reflected in the program architecture. The trait `WaveFunctionPart` represents one of these sections.

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

These parts all need to implement the `Func<_>` trait and are only valid in the range returned by `WaveFunctionPart::range`.

As previously mentioned the architecture has originally been designed around the assumption that both turning points will be used in the oscillating region. Because of this there is a specialization of this structs that can work with so called "operations". Operations were used to make the transition between the two parts of the osculating regions smoother. An operation is just a function $f : \mathbb{C} \to \mathbb{C}$ that will be applied over the whole function. The author decided not to change the architecture to the new method because the program could in theory be extended further. The wave function parts that support operations implement the `WaveFunctionPartWithOp` trait.

4.4.1. ApproxPart

An `ApproxPart` is the function around a turning point. This includes the Airy, oscillating WKB and exponential WKB part. At the same time it also handles the joints between the Airy and WKB functions.

Two joints are constructed and they have the highest "priority" when evaluating an ApproxPart for a given x.

```
fn eval(&self, x: f64) -> Complex64 {
    if is_in_range(self.airy_join_l.range(), x) && ENABLE_AIRY_JOINTS {
        return self.airy_join_l.eval(x);
    } else if is_in_range(self.airy_join_r.range(), x) && ENABLE_AIRY_JOINTS {
        return self.airy_join_r.eval(x);
    } else if is_in_range(self.airy.ts, x) {
        return self.airy.eval(x);
    } else {
        return self.wkb.eval(x);
    }
}
```

The term "priority" is used to say how far up the if statement the function is. Or in other words, functions with a higher priority are preferred. Because the joints overlap with both the ranges of the Airy and WKB function is important that they are given a higher priority. Further on the range of the Airy part is also included in the WKB range because of this the WKB part has the least priority.

As we can see, the check if the joints are even enabled happens here. Because `ENABLE_AIRY_JOINTS` is a constant. The compiler will remove the branches that are always false automatically

(see Appendix A.2). This means that in theory the program should run a little faster if `ENABLE_AIRY_JOINTS` is disabled. This has to be taken into account when benchmarking.

4.4.2. PureWkb

In the case that there are no turning points or none were found. The program will still try to calculate a wave function. This is done by taking `APPROX_INF` as the turning points. This can be done because no Airy functions will be used. In this case the turning points just act as a bound of integration.

From experience the results are inaccurate but still usable. At least in the case where the turning points were missed by Newtons method the WKB parts were fairly accurate, but unsurprisingly diverged at the turning points because there were no Airy functions.

The struct `PureWkb` works the same as `ApproxPart` but only implements the WKB functions It does not contain any Airy functions or joints.

4.5. Wave Function

To combine all the `WaveFunctionPart` structs, we will define the `WaveFunction` struct. Under the hood it will also calculate all the variables and construct all the `WaveFunctionPart` structs.

First we need to calculate the energy for the given parameters that are passed to the constructor. Note the this energy will also be printed to the terminal.

```
let energy = energy::nth_energy(n_energy, mass, &potential, approx_inf);
println!("{}␣Energy:␣{:.9}", Ordinal(n_energy).to_string(), energy);
```

Using the energy we can calculate the view as described in section 4.3.

```
(
    lower_bound * (upper_bound - lower_bound) * view_factor,
    upper_bound * (upper_bound - lower_bound) * view_factor,
)
```

Once we've got the view, we can calculate all the turning points and there Airy functions along with them, using `AiryWaveFunction::new()`. In the case that there are turning points we can then go through each turning point and also copy it's neighbors. For the outer most turning points we will take `approx_inf` as its neighbor.

With these groups of 3 we can construct a `WkbWaveFunction` for each of the turning points. However there were issues when dividing the oscillating part of the wave function was split into two parts with different turning points. As previously mentioned according to Hall (2013) it should be mathematically indistinguishable when using either of the turning points, but there arise discontinuities at the transition region. Because of that it has been decided that only the left turning point will be used.

Unfortunately in this method even though the function is continuous it will not be symmetric about the mid point of the oscillating region. This has the effect that the probabilities will be lower on the right none the less they should have the same probability. Because of the architecture of the program the oscillating part will still be split into two distinct regions.

While iterating over the turning points we can also calculate the ranges in which the functions are valid.

Once we have all the `WkbWaveFunction` instances we need to group them with the `AiryWaveFunction` instances. Using those pairs we can finally construct all the `ApproxPart` instances.

Finally we need to apply the `scaling` which may be one of the following options (where $a \in \mathbb{C}$):

None The solution wont be multiplied by anything.

Mul(a) The solution will be multiplied by a.

Renormalize(a) $\Psi(x)$ will be renormalized such that $\int_{-\infty}^{\infty} |a\Psi(x)|^2 dx = 1$. This can be useful to add a phase to the wave function.

In the case that no turning points are found WKB will be inaccurate. But for completeness we will assume that `approx_inf` is a turning point. Then we can insert two `WkbWaveFunction` instances without the Airy functions. This behavior is implemented in `PureWkb`. Afterwards we apply the same scaling procedure (4.5) as if there were turning points.

In this case you'll also get a warning in the terminal that no turning points were found. Because the results can be inaccurate.

4.5.1. Super Position

Because the super position principal is also applicable to energies it is possible that $\Psi(x)$ is a sum of wave functions with different energies.

On the implementation side this means that we can create a struct `SuperPosition` that is constructed with a list of energy levels and `ScalingType` that can be used to construct the previously discussed `WaveFunction`. Its implementation of `Func<f64, Complex64>` will then sum over all the results of the individual `WaveFunction` structs.

5. Program Manual

In the `src` directory you will find the `main.rs` file. After the imports (lines with `use`) you can find all the constants that can be configured. In the description below, (E) stands for "expert" and means that you should use the default unless you really know what you're doing.

Concurrency Configurations Tune accuracy and performance

INTEG_STEPS The number of steps that will be used to integrate over an interval

TRAPEZE_PER_THREAD (E) The number of trapezes that are calculated on a thread in sequence. This number must be smaller then `INTEG_STEPS`.

NUMBER_OF_POINTS The number of points that will be written to the output file.

APPROX_INF This are the values for "$\pm\infty$". Where the first number is $-\infty$ and the second number is ∞. Most importantly outside of this interval $V(x) > E$.

Visual Configurations Adjust the width of joints

VIEW_FACTOR This factor is used in 4.3 as f_{view}. It determines in which range the output will be calculated. This depends heavily on the potential and the energy and you probably will have to change it. If the wave function is two small and most of the plot is close to 0 then this factor has to be lowered. If the wave function is not nearly 0 at the boundary of the view, this factor should be increased. Note this factor does not influence the calculation itself.

ENABLE_WKB_JOINTS If set to `true` joints will be added between Airy and WKB wave function parts. If set to `false` no joints will be added at this boundary.

AIRY_TRANSITION_FRACTION (E) When a joint between an Airy and a WKB function has to be added, we have to know how wide the joint should be. The width is calculated by taking the distance between the turning point boundaries and multiplying it by this number.

VALIDITY_LL_FACTOR (E) This factor gets used as a in 4.13. Higher values will create larger ranges for Airy functions.

5.1. WaveFunction

When you only have one energy level you should use `WaveFunction::new`.

```
1     let wave_function = wave_function_builder::WaveFunction::new(
2         &/*potential*/,
```

```
        /*mass*/,
        /*nth energy*/,
        APPROX_INF,
        1.5,
        ScalingType::/*Scaling*/,
    );
```

The example above has to placed right after the `fn main()` line. You have to replace all the commentaries (`/*...*/`) with the values you want. For the first you can choose a potential from section 5.4 for this you can type `potentials::/*potential*/`.
For the Mass you can just use a normal float.
“nth energy ” must be a positive integer (including 0) and is the nth energy level of the potential.
And as for the scaling type, choose one of the options described at the end of section **??**.

5.2. SuperPosition

To construct a super position you can add this to your main function

```
let wave_function = wave_function_builder::SuperPosition::new(
    &/*potential*/,
    /*mass*/,
    &[
        (/*nth energy*/, /*phase*/),
        (/*nth energy*/, /*phase*/),
        // ...
    ],
    APPROX_INF,
    1.5, // view factor
    ScalingType::/*scaling*/),
);
```

Just like in section 5.1 you have to replace all the commentaries (`/*...*/`) with the values you want.
“potential” you have to choose a potential from section 5.4.
“mass” your mass as a float.
“nth energy ” must be a positive integer (including 0) and is the nth energy level of the potential.
“phase” a complex number that the wave function with the corresponding energy will be multiplied by. To make a complex number you can use `complex(/*Re*/, /*Im*/)`.
“// ...” you can add as many energies as your computer can handle.
And as for the scaling type, choose one of the options described at the end of section **??**.

5.3. Plotting

For all the plotting methods mentioned below you’ll need an output directory in which the files will be placed.

```
1 let output_dir = Path::new("output");
```

The default is *output*, you can choose any directory name that you'd like. The folder will be located where you ran the program. The data calculated by the program will be stored as space separated values like in the example below (the first line will not be in the output file).

```
x   Re    Im
1.0 2.718 3.141
2.0 1.414 1.465
```

Every line is a data point where the first number is the x-coordinate, the second the real part of $\Psi(x)$ and the third the imaginary part of $\Psi(x)$

5.3.1. WaveFunction

For a `WaveFunction` as we've seen in section 5.1 you have three options.

plot_wavefunction

With `plot::plot_wavefunction` the result will be plotted as one function in gnuplot.

```
1 plot::plot_wavefunction(&wave_function, output_dir, "data.txt");
```

You can replace *data.txt* with another file name.

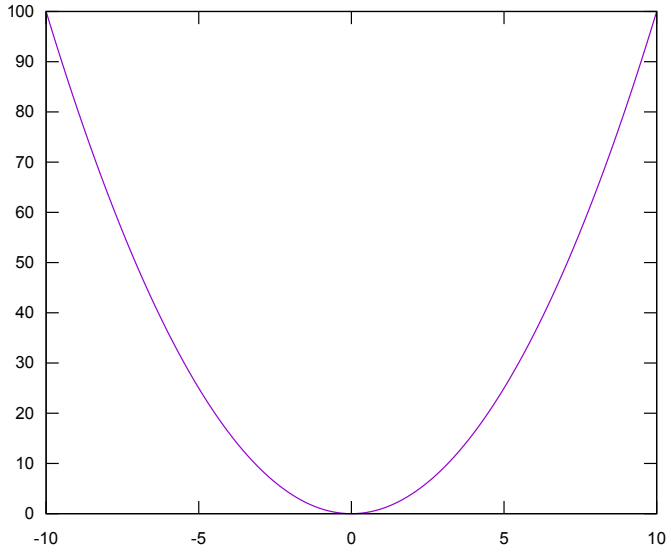
plot_wavefunction_parts

plot_probability

5.4. Potentials

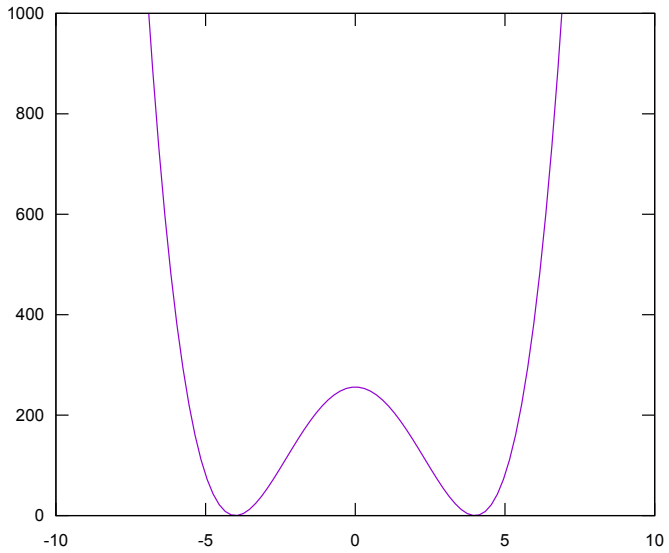
square Normal square potential as used in Hall (2013).

$$x^2$$



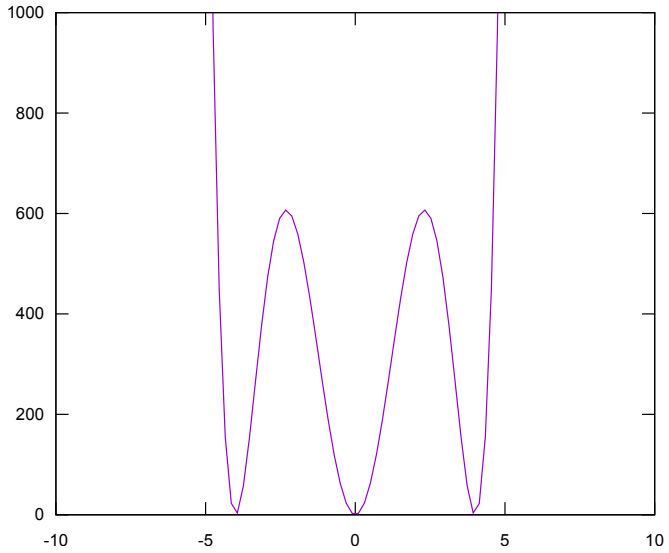
mexican_hat 4th degree polynomial that looks like a mexican hat, with 2 minima.

$$(x-4)^2(x+4)^2$$



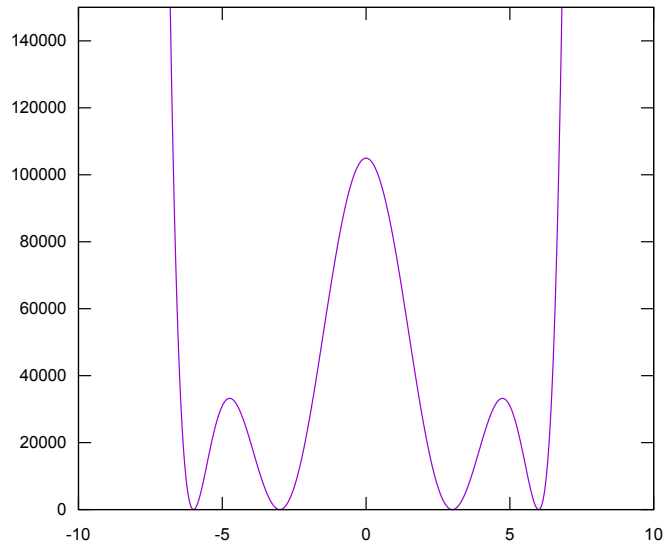
double_mexican_hat 6th degree polynomial that has 3 minima.

$$(x-4)^2x^2(x+4)^2$$



triple_mexican_hat 8th degree polynomial that has 4 minima.

$$(x-6)^2(x-3)^2(x+3)^2(x+6)^2$$



smooth_step Step function that goes to `ENERGY_INF` outside the interval $(-5, 5)$. Joints were added at ± 5 to make the function differentiable.

5.4.1. Custom Potentials

To create a custom potential you'll have to define a function like shown below.

```
fn my_potential(x: f64) -> f64 {
    return /*some calculation*/;
}
```

`my_potential` is the name that you can choose and have to use later when you're passing it to `WaveFunction::new`. `/*some calculation*/` can be any Rust code that results in a `f64`.

Examples

Negative bell curve $(-e^{-x^2} + 1)$

```
fn neg_bell(x: f64) -> f64 {
    return -(-x.powi(2)).exp();
}
```

General polynomial (might not work for all configurations)

```
const COEFFICIENTS: [f64;4] = [a, b, c, d]
fn polynom(x: f64) -> f64 {
```

```
    let mut result = 0.0;
    for n in 0..COEFFICIENTS.len() {
        result += x.powi(n) * COEFFICIENTS[n];
    }
    return result;
}
```

You need to set values for `a`, `b`, etc. and they need to be floating point numbers or you'll get error E0308. For example `1` would cause an error but `1.0` or `3.141` are correct. You can add even more coefficients if you'd like. The 4 in the square brackets is the degree of the polynomial plus 1. The potential above would mathematically be $a + bx + cx^2 + dx^3$.

A. Detailed Calculations and Tests

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

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

and

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

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

A.2. Branch Elimination

To check if branches with a constant condition of `false` gets removed we will use "Compiler Explorer" on `https://godbolt.org/`. This is an online tool to generate the assembly of source code. The settings used for this test are "Rust" for the language, "rustc 1.65.0" for the compiler with flags "-O".

Rust Code

```
const COND: bool = true;

pub fn test(x: f64) -> f64 {
    if x > 5.0 && COND {
        return x % 5.0;
    } else {
        return x*x;
    }
}
```

Assembly

```
.LCPI0_0:
        .quad   0x4014000000000000
example::test:
        push    rax
        ucomisd xmm0, qword ptr [rip + .
            LCPI0_0]
        jbe     .LBB0_1
        movsd   xmm1, qword ptr [rip + .
            LCPI0_0]
        call    qword ptr [rip +
            fmod@GOTPCREL]
        pop     rax
        ret
.LBB0_1:
        mulsd   xmm0, xmm0
        pop     rax
        ret
```

As we can see in the `true` case the `.LBB0_1` label was inserted which means the code will branch.

Rust Code

```
const COND: bool = false;

pub fn test(x: f64) -> f64 {
    if x > 5.0 && COND {
```

Assembly

```
example::test:
        mulsd   xmm0, xmm0
        ret
```

```
        return x % 5.0;
    } else {
        return x*x;
    }
}
```

In the `false` case the compiler directly calculates x^2 directly without any checks since the first condition is always `false`.

B. Data Files

B.1. Energies

energies_approx.dat

```
0 0.7071985499773434
1 2.121283145049141
2 3.535523992562384
3 4.949764840075626
4 6.364005687588868
5 7.778246535102111
6 9.192487382615353
7 10.606571982569964
8 12.020812830083207
9 13.435366182479338
10 14.849294525109691
11 16.263535372622933
12 17.67761996769473
13 19.091860815207973
14 20.506257920045474
15 21.92034251511727
16 23.33442711018907
17 24.74866795770231
18 26.163221310098443
19 27.577305905170242
20 28.99185925756637
21 30.40578760507954
22 31.82002845259278
23 33.23442555254747
24 34.648041390294935
25 36.062282237808176
26 37.477148095087195
27 38.89076393283466
28 40.305004785230715
29 41.71971439006829
30 43.133330227815755
31 44.547571075328996
32 45.96181192284224
33 47.37636527523837
34 48.79044987031017
35 50.204534470264775
36 51.61908782266091
37 53.03301616529126
```

energies_exact.dat

```
0 0.7071067811865475
1 2.1213203435596424
2 3.5355339059327373
3 4.949747468305832
4 6.363961030678928
5 7.778174593052022
6 9.192388155425117
7 10.606601717798211
8 12.020815280171307
9 13.435028842544401
10 14.849242404917497
11 16.263455967290593
12 17.677669529663685
13 19.09188309203678
14 20.506096654409877
15 21.920310216782973
16 23.334523779156065
17 24.74873734152916
18 26.162950903902257
19 27.577164466275352
20 28.991378028648445
21 30.40559159102154
22 31.819805153394636
23 33.23401871576773
24 34.648232278140824
25 36.062445840513924
26 37.476659402887016
27 38.89087296526011
28 40.30508652763321
29 41.7193000900063
30 43.13351365237939
31 44.54772721475249
32 45.961940777125584
33 47.37615433949868
34 48.790367901871775
35 50.20458146424487
36 51.61879502661797
37 53.03300858899106
```

```
38 54.4472570128045
39 55.8613416078763
40 57.27558245538954
41 58.68966705046134
42 60.10453291262317
43 61.518148750370635
44 62.93270210276677
45 64.3472554551629
46 65.76149630267614
47 67.17542464530649
48 68.58935298793685
49 70.00343758789145
50 71.41783468784614
```

```
38 54.44722215136415
39 55.86143571373725
40 57.27564927611034
41 58.68986283848344
42 60.104076400856535
43 61.51828996322963
44 62.932503525602726
45 64.34671708797582
46 65.76093065034891
47 67.175144212722
48 68.58935777509511
49 70.0035713374682
50 71.4177848998413
```

C. Source Code

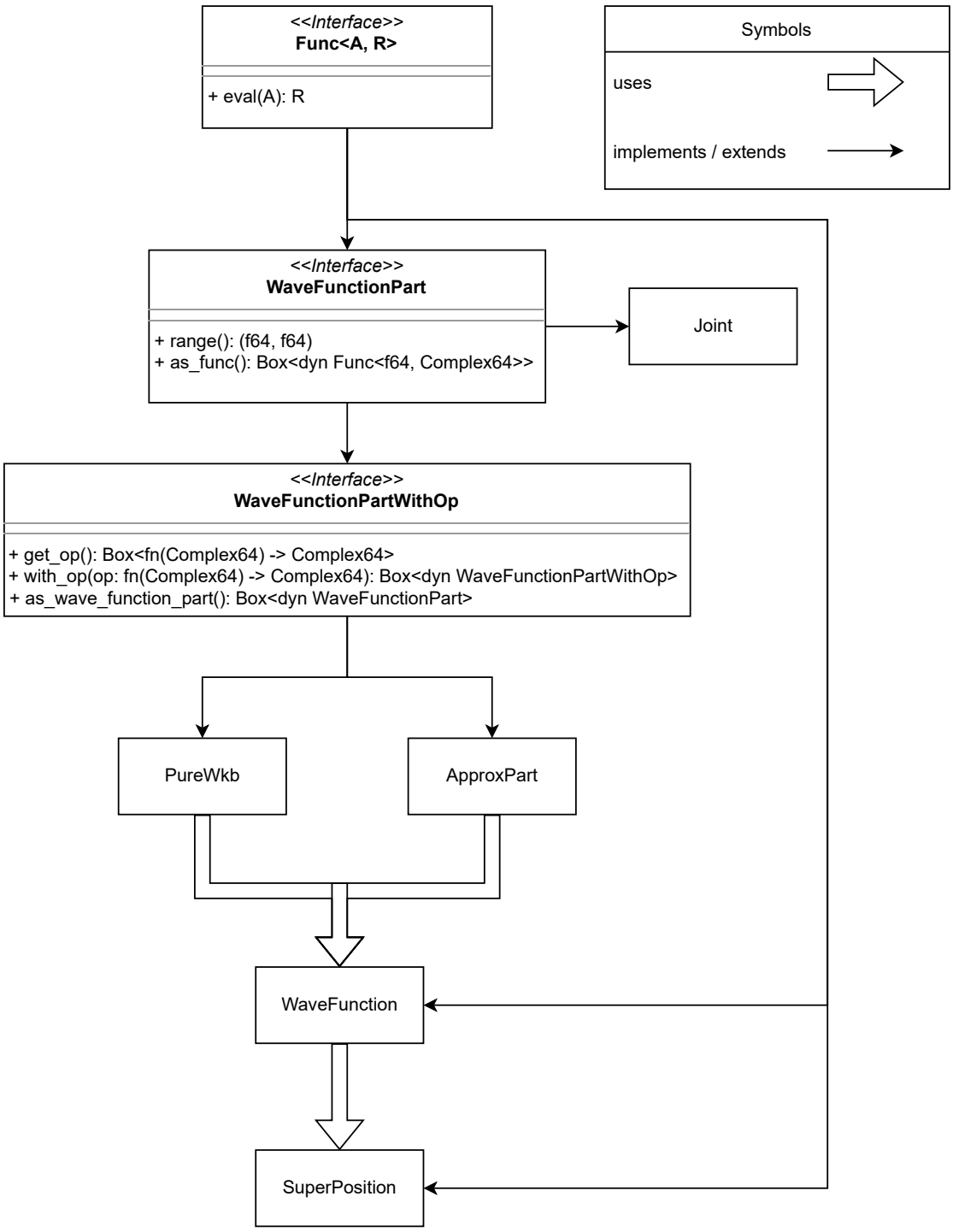


Figure C.1.: UML diagram of program architecture

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

src/main.rs

```rust
#![allow(dead_code)]

mod airy;
mod airy_wave_func;
mod check;
mod energy;
mod integrals;
mod newtons_method;
mod plot;
mod potentials;
mod tui;
mod turning_points;
mod utils;
mod wave_function_builder;
mod wkb_wave_func;

use crate::airy::airy_ai;
use crate::airy_wave_func::AiryWaveFunction;
use crate::integrals::*;
use crate::newtons_method::derivative;
use crate::utils::Func;
use crate::utils::*;
use crate::wave_function_builder::*;
use crate::wkb_wave_func::WkbWaveFunction;
use num::complex::Complex64;
use num::pow::Pow;
use rayon::iter::*;
use std::f64;
use std::fs::File;
use std::io::Write;
use std::path::Path;
use std::sync::Arc;

const INTEG_STEPS: usize = 64000;
const TRAPEZE_PER_THREAD: usize = 1000;
const NUMBER_OF_POINTS: usize = 100000;

const AIRY_TRANSITION_FRACTION: f64 = 0.5;
const ENABLE_AIRY_JOINTS: bool = true;

const VALIDITY_LL_FACTOR: f64 = 3.5;

const APPROX_INF: (f64, f64) = (-200.0, 200.0);
```

```rust

fn main() {
    // let wave_function = wave_function_builder::WaveFunction::new(
    //     &potentials::square,
    //     1.0,
    //     17,
    //     APPROX_INF,
    //     0.15,
    //     ScalingType::Renormalize(1.0.into()),
    // );

    let wave_function = wave_function_builder::SuperPosition::new(
        &potentials::square,
        1.0,
        &[
            (15, 1.0.into()),
            (16, complex(0.0, 1.0 * f64::consts::PI / 8.0).exp()),
            // (15, complex(0.0, 2.0 * f64::consts::PI / 3.0).exp()),
        ],
        APPROX_INF,
        0.15,
        ScalingType::Renormalize(complex(1.0, 0.0)),
    );

    let output_dir = Path::new("output");

    // For WaveFunction
    // plot::plot_wavefunction(&wave_function, output_dir, "data.txt");
    // plot::plot_wavefunction_parts(&wave_function, output_dir, "data.txt");
    // plot::plot_probability(&wave_function, output_dir, "data.txt");

    // For SuperPosition
    // plot::plot_superposition(&wave_function, output_dir, "data.txt");
    plot::plot_probability_super_pos(&wave_function, output_dir, "data.txt");
}
```

src/airy.rs

```rust
/* automatically generated by rust-bindgen 0.59.2 */
#![allow(non_snake_case)]
#![allow(deref_nullptr)]
#![allow(non_camel_case_types)]

#[derive(PartialEq, Copy, Clone, Hash, Debug, Default)]
#[repr(C)]
pub struct __BindgenComplex<T> {
    pub re: T,
    pub im: T,
}
```

```rust
pub type size_t = ::std::os::raw::c_ulong;
pub type wchar_t = ::std::os::raw::c_int;
#[repr(C)]
#[repr(align(16))]
#[derive(Debug, Copy, Clone)]
pub struct max_align_t {
    pub __clang_max_align_nonce1: ::std::os::raw::c_longlong,
    pub __bindgen_padding_0: u64,
    pub __clang_max_align_nonce2: u128,
}
#[test]
fn bindgen_test_layout_max_align_t() {
    assert_eq!(
        ::std::mem::size_of::<max_align_t>(),
        32usize,
        concat!("Size␣of:␣", stringify!(max_align_t))
    );
    assert_eq!(
        ::std::mem::align_of::<max_align_t>(),
        16usize,
        concat!("Alignment␣of␣", stringify!(max_align_t))
    );
    assert_eq!(
        unsafe {
            &(*(::std::ptr::null::<max_align_t>())).__clang_max_align_nonce1 as *
                const _ as usize
        },
        0usize,
        concat!(
            "Offset␣of␣field:␣",
            stringify!(max_align_t),
            "::",
            stringify!(__clang_max_align_nonce1)
        )
    );
    assert_eq!(
        unsafe {
            &(*(::std::ptr::null::<max_align_t>())).__clang_max_align_nonce2 as *
                const _ as usize
        },
        16usize,
        concat!(
            "Offset␣of␣field:␣",
            stringify!(max_align_t),
            "::",
            stringify!(__clang_max_align_nonce2)
        )
    );
}
```

```rust
#[repr(C)]
#[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;
```

```rust
pub type GoUintptr = ::std::os::raw::c_ulong;
pub type GoFloat32 = f32;
pub type GoFloat64 = f64;
pub type GoComplex64 = __BindgenComplex<f32>;
pub type GoComplex128 = __BindgenComplex<f64>;
pub type _check_for_64_bit_pointer_matching_GoInt = [::std::os::raw::c_char; 1usize];
pub type GoString = _GoString_;
pub type GoMap = *mut ::std::os::raw::c_void;
pub type GoChan = *mut ::std::os::raw::c_void;
#[repr(C)]
#[derive(Debug, Copy, Clone)]
pub struct GoInterface {
    pub t: *mut ::std::os::raw::c_void,
    pub v: *mut ::std::os::raw::c_void,
}
#[test]
fn bindgen_test_layout_GoInterface() {
    assert_eq!(
        ::std::mem::size_of::<GoInterface>(),
        16usize,
        concat!("Size␣of:␣", stringify!(GoInterface))
    );
    assert_eq!(
        ::std::mem::align_of::<GoInterface>(),
        8usize,
        concat!("Alignment␣of␣", stringify!(GoInterface))
    );
    assert_eq!(
        unsafe { &(*(::std::ptr::null::<GoInterface>())).t as *const _ as usize },
        0usize,
        concat!(
            "Offset␣of␣field:␣",
            stringify!(GoInterface),
            "::",
            stringify!(t)
        )
    );
    assert_eq!(
        unsafe { &(*(::std::ptr::null::<GoInterface>())).v as *const _ as usize },
        8usize,
        concat!(
            "Offset␣of␣field:␣",
            stringify!(GoInterface),
            "::",
            stringify!(v)
        )
    );
}
#[repr(C)]
```

```rust
#[derive(Debug, Copy, Clone)]
pub struct GoSlice {
    pub data: *mut ::std::os::raw::c_void,
    pub len: GoInt,
    pub cap: GoInt,
}
#[test]
fn bindgen_test_layout_GoSlice() {
    assert_eq!(
        ::std::mem::size_of::<GoSlice>(),
        24usize,
        concat!("Size␣of:␣", stringify!(GoSlice))
    );
    assert_eq!(
        ::std::mem::align_of::<GoSlice>(),
        8usize,
        concat!("Alignment␣of␣", stringify!(GoSlice))
    );
    assert_eq!(
        unsafe { &(*(::std::ptr::null::<GoSlice>())).data as *const _ as usize },
        0usize,
        concat!(
            "Offset␣of␣field:␣",
            stringify!(GoSlice),
            "::",
            stringify!(data)
        )
    );
    assert_eq!(
        unsafe { &(*(::std::ptr::null::<GoSlice>())).len as *const _ as usize },
        8usize,
        concat!(
            "Offset␣of␣field:␣",
            stringify!(GoSlice),
            "::",
            stringify!(len)
        )
    );
    assert_eq!(
        unsafe { &(*(::std::ptr::null::<GoSlice>())).cap as *const _ as usize },
        16usize,
        concat!(
            "Offset␣of␣field:␣",
            stringify!(GoSlice),
            "::",
            stringify!(cap)
        )
    );
}
```

```rust
#[repr(C)]
#[derive(Debug, Copy, Clone)]
pub struct airy_ai_return {
    pub r0: GoFloat64,
    pub r1: GoFloat64,
}
#[test]
fn bindgen_test_layout_airy_ai_return() {
    assert_eq!(
        ::std::mem::size_of::<airy_ai_return>(),
        16usize,
        concat!("Size␣of:␣", stringify!(airy_ai_return))
    );
    assert_eq!(
        ::std::mem::align_of::<airy_ai_return>(),
        8usize,
        concat!("Alignment␣of␣", stringify!(airy_ai_return))
    );
    assert_eq!(
        unsafe { &(*(::std::ptr::null::<airy_ai_return>())).r0 as *const _ as usize
            },
        0usize,
        concat!(
            "Offset␣of␣field:␣",
            stringify!(airy_ai_return),
            "::",
            stringify!(r0)
        )
    );
    assert_eq!(
        unsafe { &(*(::std::ptr::null::<airy_ai_return>())).r1 as *const _ as usize
            },
        8usize,
        concat!(
            "Offset␣of␣field:␣",
            stringify!(airy_ai_return),
            "::",
            stringify!(r1)
        )
    );
}
extern "C" {
    pub fn airy_ai(zr: GoFloat64, zi: GoFloat64) -> airy_ai_return;
}
```

src/airy_wave_func.rs

```rust
use crate::newtons_method::newtons_method;
use crate::newtons_method::*;
```

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

#[allow(non_snake_case)]
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);
}

#[allow(non_snake_case)]
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(
                    &|x| (phase.potential)(x) - phase.energy,
                    (*t1 + *t2) / 2.0,
                    1e-7,
                );
                let u_1 = 2.0 * phase.mass * -derivative(phase.potential.as_ref(),
                    x_1);
                // let u_1 = |x| -2.0 * phase.mass * ((phase.potential)(&x) - phase.
                    energy) / (H_BAR * H_BAR * (x - x_1));

                AiryWaveFunction {
                    u_1,
                    turning_point: x_1,
                    phase: phase.clone(),
                    ts: (*t1, *t2),
                    op: identity,
                    c: 1.0.into(),
                    phase_off: 0.0,
                }
            })
            .collect::<Vec<AiryWaveFunction>>();
        return (funcs, turning_point_boundaries);
    }

    pub fn with_op(&self, op: fn(Complex64) -> Complex64) -> AiryWaveFunction {
        AiryWaveFunction {
            u_1: self.u_1,
            turning_point: self.turning_point,
            phase: self.phase.clone(),
            ts: self.ts,
            op,
            c: self.c,
            phase_off: self.phase_off,
        }
    }

    pub fn with_c(&self, c: Complex64) -> AiryWaveFunction {
        AiryWaveFunction {
            u_1: self.u_1,
            turning_point: self.turning_point,
            phase: self.phase.clone(),
            ts: self.ts,
            op: self.op,
            c,
            phase_off: self.phase_off,
        }
    }
```

```
97
98      pub fn with_phase_off(&self, phase_off: f64) -> AiryWaveFunction {
99          AiryWaveFunction {
100             u_1: self.u_1,
101             turning_point: self.turning_point,
102             phase: self.phase.clone(),
103             ts: self.ts,
104             op: self.op,
105             c: self.c,
106             phase_off,
107         }
108     }
109 }
110
111 impl Func<f64, Complex64> for AiryWaveFunction {
112     fn eval(&self, x: f64) -> Complex64 {
113         let u_1_cube_root = Self::get_u_1_cube_root(self.u_1);
114
115         if self.u_1 < 0.0 {
116             return self.c
117                 * ((std::f64::consts::PI.sqrt() / (self.u_1).abs().pow(1.0 / 6.0))
118                     * Ai(complex(u_1_cube_root * (self.turning_point - x), 0.0)))
119                     as Complex64;
120         } else {
121             return self.c
122                 * ((std::f64::consts::PI.sqrt() / (self.u_1).abs().pow(1.0 / 6.0))
123                     * Ai(complex(u_1_cube_root * (self.turning_point - x), 0.0)))
124                     as Complex64;
125         }
126     }
127 }
128
129 #[cfg(test)]
130 mod test {
131     use super::*;
132
133     #[test]
134     fn airy_func_plot() {
135         let output_dir = Path::new("output");
136         std::env::set_current_dir(&output_dir).unwrap();
137
138         let airy_ai = Function::new(|x| Ai(complex(x, 0.0)));
139         let airy_bi = Function::new(|x| Bi(complex(x, 0.0)));
140         let values = evaluate_function_between(&airy_ai, -10.0, 5.0, NUMBER_OF_POINTS
            );
141
142         let mut data_file = File::create("airy.txt").unwrap();
143
144         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

```
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()
                .energy)
                * self.wave_func.eval(x)
    }
}
```

src/energy.rs

```
use crate::*;

struct Integrand<'a, F: Fn(f64) -> f64 + Sync> {
    mass: f64,
    pot: &'a F,
    energy: f64,
}

```

```rust
impl<F: Fn(f64) -> f64 + Sync> Func<f64, f64> for Integrand<'_, F> {
    fn eval(&self, x: f64) -> f64 {
        let pot = (self.pot)(x);

        if !pot.is_finite() {
            return 0.0;
        }

        if pot < self.energy {
            return (2.0 * self.mass * (self.energy - pot)).sqrt();
        } else {
            return 0.0;
        }
    }
}

struct SommerfeldCond<'a, F: Fn(f64) -> f64 + Sync> {
    mass: f64,
    pot: &'a F,
    view: (f64, f64),
}

impl<F: Fn(f64) -> f64 + Sync> Func<f64, f64> for SommerfeldCond<'_, F> {
    fn eval(&self, energy: f64) -> f64 {
        let integrand = Integrand {
            mass: self.mass,
            pot: self.pot,
            energy,
        };
        let integral = integrate(
            evaluate_function_between(&integrand, self.view.0, self.view.1,
                INTEG_STEPS),
            TRAPEZE_PER_THREAD,
        );
        return ((2.0 * 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();
    }
}
```

src/integrals.rs

```
use crate::*;
use rayon::prelude::*;

#[allow(non_camel_case_types)]
#[derive(Clone)]
pub struct Point<T_X, T_Y> {
    pub x: T_X,
    pub y: T_Y,
}

pub fn trapezoidal_approx<X, Y>(start: &Point<X, Y>, end: &Point<X, Y>) -> Y
where
    X: std::ops::Sub<Output = X> + Copy,
    Y: std::ops::Add<Output = Y>
        + std::ops::Mul<Output = Y>
        + std::ops::Div<f64, Output = Y>
        + Copy
        + From<X>,
{
    return Y::from(end.x - start.x) * (start.y + end.y) / 2.0_f64;
}

```

```rust
pub fn index_to_range<T>(x: T, in_min: T, in_max: T, out_min: T, out_max: T) -> T
where
    T: Copy
        + std::ops::Sub<Output = T>
        + std::ops::Mul<Output = T>
        + std::ops::Div<Output = T>
        + std::ops::Add<Output = T>,
{
    return (x - in_min) * (out_max - out_min) / (in_max - in_min) + out_min;
}

pub fn integrate<
    X: Sync + std::ops::Add<Output = X> + std::ops::Sub<Output = X> + Copy,
    Y: Default
        + Sync
        + std::ops::AddAssign
        + std::ops::Div<f64, Output = Y>
        + std::ops::Mul<Output = Y>
        + std::ops::Add<Output = Y>
        + Send
        + std::iter::Sum<Y>
        + 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]);
```

```rust
    }

    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![];
    }

    (0..n)
        .into_par_iter()
        .map(|i| {
            index_to_range(
                X::from(i as f64),
                X::from(0.0_f64),
                X::from((n - 1) as f64),
                a,
                b,
            )
        })
        .map(|x: X| Point { x, y: f.eval(x) })
        .collect()
}

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

    fn square(x: f64) -> Complex64 {
        return complex(x * x, 0.0);
    }

    fn square_integral(a: f64, b: f64) -> Complex64 {
        return complex(b * b * b / 3.0 - a * a * a / 3.0, 0.0);
    }
```

```
119
120     #[tokio::test(flavor = "multi_thread")]
121     async fn integral_of_square() {
122         let square_func: Function<f64, Complex64> = Function::new(square);
123         for i in 0..100 {
124             for j in 0..10 {
125                 let a = f64::from(i - 50) / 12.3;
126                 let b = f64::from(j - 50) / 12.3;
127
128                 if i == j {
129                     assert_eq!(
130                         integrate(
131                             evaluate_function_between(&square_func, a, b, INTEG_STEPS
                                    ),
132                             TRAPEZE_PER_THREAD,
133                         ),
134                         complex(0.0, 0.0)
135                     );
136                     continue;
137                 }
138
139                 let epsilon = 0.00001;
140                 assert!(complex_compare(
141                     integrate(
142                         evaluate_function_between(&square_func, a, b, INTEG_STEPS),
143                         TRAPEZE_PER_THREAD,
144                     ),
145                     square_integral(a, b),
146                     epsilon,
147                 ));
148             }
149         }
150     }
151
152     #[test]
153     fn evaluate_square_func_between() {
154         let square_func: Function<f64, Complex64> = Function::new(square);
155         let actual = evaluate_function_between(&square_func, -2.0, 2.0, 5);
156         let expected = vec![
157             Point {
158                 x: -2.0,
159                 y: complex(4.0, 0.0),
160             },
161             Point {
162                 x: -1.0,
163                 y: complex(1.0, 0.0),
164             },
165             Point {
166                 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();
    }

    fn sinusoidal_exp_complex_integral(a: f64, b: f64) -> Complex64 {
        // (-1/2 + i/2) (e^((1 + i) a) - e^((1 + i) b))
        return complex(-0.5, 0.5) * (complex(a, a).exp() - complex(b, b).exp());
    }

    #[tokio::test(flavor = "multi_thread")]
    async fn integral_of_sinusoidal_exp() {
        let sinusoidal_exp_complex: Function<f64, Complex64> =
            Function::new(sinusoidal_exp_complex);
        for i in 0..10 {
            for j in 0..10 {
                let a = f64::from(i - 50) / 12.3;
                let b = f64::from(j - 50) / 12.3;

                if i == j {
                    assert_eq!(
                        integrate(
                            evaluate_function_between(&sinusoidal_exp_complex, a, b,
                                INTEG_STEPS),
                            TRAPEZE_PER_THREAD,
                        ),
                        complex(0.0, 0.0)
                    );
                    continue;
                }
                let epsilon = 0.0001;
                assert!(complex_compare(
```

```
                    integrate(
                        evaluate_function_between(&sinusoidal_exp_complex, a, b,
                            INTEG_STEPS),
                        TRAPEZE_PER_THREAD,
                    ),
                    sinusoidal_exp_complex_integral(a, b),
                    epsilon,
                ));
            }
        }
    }
}
```

src/main.rs

```
#![allow(dead_code)]

mod airy;
mod airy_wave_func;
mod check;
mod energy;
mod integrals;
mod newtons_method;
mod plot;
mod potentials;
mod tui;
mod turning_points;
mod utils;
mod wave_function_builder;
mod wkb_wave_func;

use crate::airy::airy_ai;
use crate::airy_wave_func::AiryWaveFunction;
use crate::integrals::*;
use crate::newtons_method::derivative;
use crate::utils::Func;
use crate::utils::*;
use crate::wave_function_builder::*;
use crate::wkb_wave_func::WkbWaveFunction;
use num::complex::Complex64;
use num::pow::Pow;
use rayon::iter::*;
use std::f64;
use std::fs::File;
use std::io::Write;
use std::path::Path;
use std::sync::Arc;

const INTEG_STEPS: usize = 64000;
```

```rust
const TRAPEZE_PER_THREAD: usize = 1000;
const NUMBER_OF_POINTS: usize = 100000;

const AIRY_TRANSITION_FRACTION: f64 = 0.5;
const ENABLE_AIRY_JOINTS: bool = true;

const VALIDITY_LL_FACTOR: f64 = 3.5;

const APPROX_INF: (f64, f64) = (-200.0, 200.0);

fn main() {
    // let wave_function = wave_function_builder::WaveFunction::new(
    //     &potentials::square,
    //     1.0,
    //     17,
    //     APPROX_INF,
    //     0.15,
    //     ScalingType::Renormalize(1.0.into()),
    // );

    let wave_function = wave_function_builder::SuperPosition::new(
        &potentials::square,
        1.0,
        &[
            (15, 1.0.into()),
            (16, complex(0.0, 1.0 * f64::consts::PI / 8.0).exp()),
            // (15, complex(0.0, 2.0 * f64::consts::PI / 3.0).exp()),
        ],
        APPROX_INF,
        0.15,
        ScalingType::Renormalize(complex(1.0, 0.0)),
    );

    let output_dir = Path::new("output");

    // For WaveFunction
    // plot::plot_wavefunction(&wave_function, output_dir, "data.txt");
    // plot::plot_wavefunction_parts(&wave_function, output_dir, "data.txt");
    // plot::plot_probability(&wave_function, output_dir, "data.txt");

    // For SuperPosition
    // plot::plot_superposition(&wave_function, output_dir, "data.txt");
    plot::plot_probability_super_pos(&wave_function, output_dir, "data.txt");
}
```

src/newtons_method.rs

```rust
use crate::integrals::*;
use crate::utils::cmp_f64;
```

```rust
use num::Float;
use rayon::prelude::*;
use std::cmp::Ordering;
use std::fmt::Debug;
use std::ops::*;
use std::sync::Arc;

#[derive(Default, Debug)]
pub struct Vec2 {
    x: f64,
    y: f64,
}

impl Vec2 {
    pub fn dot(&self, other: &Vec2) -> f64 {
        return self.x * other.x + self.y + other.y;
    }

    pub fn mag(&self) -> f64 {
        return (self.x.powi(2) * self.y.powi(2)).sqrt();
    }

    pub fn pseudo_inverse(&self) -> CoVec2 {
        CoVec2(self.x, self.y) * (1.0 / (self.x.powi(2) + self.y.powi(2)))
    }
}

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,
        }
    }
}

```

```rust
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;

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

impl Mul<Vec2> for CoVec2 {
    type Output = f64;

    fn mul(self, vec: Vec2) -> Self::Output {
        return self.0 * vec.x + self.1 * vec.y;
    }
}

impl Mul<f64> for CoVec2 {
    type Output = CoVec2;

    fn mul(self, s: f64) -> Self::Output {
        CoVec2(self.0 * s, self.1 * s)
    }
}

fn gradient<F>(f: F, x: f64) -> Vec2
where
    F: Fn(f64) -> Vec2,
```

```
{
    let x_component = |x| f(x).x;
    let y_component = |x| f(x).y;
    return Vec2 {
        x: derivative(&x_component, x),
        y: derivative(&y_component, x),
    };
}

// pub fn derivative<F, R>(f: &F, x: f64) -> R
// where
//     F: Fn(f64) -> R + ?Sized,
//     R: Sub<R, Output = R> + Div<f64, Output = R>,
// {
//     let epsilon = f64::epsilon().sqrt();
//     (f(x + epsilon / 2.0) - f(x - epsilon / 2.0)) / epsilon
// }

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

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

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

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

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,
    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 check_sign(initial: f64, new: f64) -> bool {
    if initial == new {
        return false;
    }
```

```rust
    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
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);
}
```

```rust

#[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 {
            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 {
```

```rust
                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>
where
    F: Fn(f64) -> f64 + Sync,
{
    let sort_func = |(_, y1): &(f64, f64), (_, y2): &(f64, f64)| -> Ordering {
        cmp_f64(&y1, &y2) };
    let mut points: Vec<(f64, f64)> = (0..n)
        .into_par_iter()
        .map(|i| index_to_range(i as f64, 0.0, n as f64, start, end))
        .map(move |x| {
            let der = derivative(f, x);
            (x, f(x) / (-(-der * der).exp() + 1.0))
        })
        .map(|(x, y)| (x, y.abs()))
        .collect();
    points.sort_by(sort_func);
    points.get(0).map(|point| point.0)
}

pub fn newtons_method_find_new_zero<F>(
    f: &F,
    guess: f64,
    precision: f64,
    max_iters: usize,
    known_zeros: &Vec<f64>,
) -> Option<f64>
where
    F: Fn(f64) -> f64,
{
    let f_modified = |x| f(x) / known_zeros.iter().fold(0.0, |acc, &z| acc * (x - z))
        ;
    newtons_method_max_iters(&f_modified, guess, precision, max_iters)
}

```

```rust
#[cfg(test)]
mod test {
    use super::*;
    use crate::utils::cmp_f64;

    fn float_compare(expect: f64, actual: f64, epsilon: f64) -> bool {
        let average = (expect.abs() + actual.abs()) / 2.0;
        if average != 0.0 {
            (expect - actual).abs() / average < epsilon
        } else {
            (expect - actual).abs() < epsilon
        }
    }

    #[test]
    fn derivative_square_test() {
        let square = |x| x * x;
        let actual = |x| 2.0 * x;

        for i in 0..100 {
            let x = index_to_range(i as f64, 0.0, 100.0, -20.0, 20.0);
            assert!(float_compare(derivative(&square, x), actual(x), 1e-4));
        }
    }

    #[test]
    fn derivative_exp_test() {
        let exp = |x: f64| x.exp();

        for i in 0..100 {
            let x = index_to_range(i as f64, 0.0, 100.0, -20.0, 20.0);
            assert!(float_compare(derivative(&exp, x), exp(x), 1e-4));
        }
    }

    #[test]
    fn newtons_method_square() {
        for i in 0..100 {
            let zero = index_to_range(i as f64, 0.0, 100.0, 0.1, 10.0);
            let func = |x| x * x - zero * zero;
            assert!(float_compare(
                newtons_method(&func, 100.0, 1e-7),
                zero,
                1e-4,
            ));
            assert!(float_compare(
                newtons_method(&func, -100.0, 1e-7),
                -zero,
                1e-4,
```

```
            ));
        }
    }

    #[test]
    fn newtons_method_cube() {
        for i in 0..100 {
            let zero = index_to_range(i as f64, 0.0, 100.0, 0.1, 10.0);
            let func = |x| (x - zero) * (x + zero) * (x - zero / 2.0);
            assert!(float_compare(
                newtons_method(&func, 100.0, 1e-7),
                zero,
                1e-4,
            ));
            assert!(float_compare(
                newtons_method(&func, -100.0, 1e-7),
                -zero,
                1e-4,
            ));
            assert!(float_compare(
                newtons_method(&func, 0.0, 1e-7),
                zero / 2.0,
                1e-4,
            ));
        }
    }

    #[test]
    fn newtons_method_find_next_polynomial() {
        for i in 0..10 {
            for j in 0..10 {
                for k in 0..10 {
                    let a = index_to_range(i as f64, 0.0, 10.0, -10.0, 10.0);
                    let b = index_to_range(j as f64, 0.0, 10.0, -100.0, 0.0);
                    let c = index_to_range(k as f64, 0.0, 10.0, -1.0, 20.0);
                    let test_func = |x: f64| (x - a) * (x - b) * (x - c);

                    for _guess in [a, b, c] {
                        let mut finder =
                            NewtonsMethodFindNewZero::new(Arc::new(test_func), 1e-15,
                                 10000000);

                        finder.next_zero(1.0);
                        finder.next_zero(1.0);
                        finder.next_zero(1.0);

                        let mut zeros_expected = [a, b, c];
                        let mut zeros_actual = finder.get_previous_zeros().clone();

```

```
                        zeros_expected.sort_by(cmp_f64);
                        zeros_actual.sort_by(cmp_f64);

                        assert_eq!(zeros_actual.len(), 3);

                        for (expected, actual) in zeros_expected.iter().zip(
                            zeros_actual.iter()) {
                            assert!((*expected - *actual).abs() < 1e-10);
                        }
                    }
                }
            }
        }
    }

    #[test]
    fn newtons_method_find_next_test() {
        let interval = (-10.0, 10.0);

        let test_func = |x: f64| 5.0 * (3.0 * x + 1.0).abs() - (1.5 * x.powi(2) + x -
             50.0).powi(2);

        let mut finder = NewtonsMethodFindNewZero::new(Arc::new(test_func), 1e-11,
            100000000);

        for _i in 0..4 {
            let guess = make_guess(&|x| finder.modified_func(x), interval, 1000);
            finder.next_zero(guess.unwrap());
        }

        let mut zeros = finder.get_previous_zeros().clone();
        zeros.sort_by(cmp_f64);
        let expected = [-6.65276132415, -5.58024707627, 4.91358040961,
            5.98609465748];

        println!("zeros:␣{:#?}", zeros);

        assert_eq!(zeros.len(), expected.len());

        for (expected, actual) in expected.iter().zip(zeros.iter()) {
            assert!((*expected - *actual).abs() < 1e-10);
        }
    }

    #[test]
    fn regula_falsi_bisection_test() {
        let func = |x: f64| x * (x - 2.0) * (x + 2.0);

        let actual = regula_falsi_bisection(&func, -1e-3, -1e-3, 1e-5);
```

```
        let expected = -2.0;

        println!("expected: {}, actual {}", expected, actual);
        assert!(float_compare(expected, actual, 1e-3));
    }
}
```

src/plot.rs

```
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,
            )
        })
        .collect::<Vec<Vec<Point<f64, Complex64>>>>();

    let wkb_values_str = wkb_values
        .par_iter()
        .map(|values| to_gnuplot_string_complex(values.to_vec()))
        .reduce(
            || String::new(),
            |s: String, current: String| s + "\n\n" + &*current,
        );

    let airy_values_str = airy_values
        .par_iter()
        .map(|values| to_gnuplot_string_complex(values.to_vec()))
        .reduce(
            || String::new(),
            |s: String, current: String| s + "\n\n" + &*current,
        );

    let mut data_full = File::create(output_file).unwrap();
    data_full.write_all(wkb_values_str.as_ref()).unwrap();
    data_full.write_all("\n\n".as_bytes()).unwrap();
    data_full.write_all(airy_values_str.as_ref()).unwrap();

    let mut plot_3d_file = File::create("plot_3d.gnuplot").unwrap();

    let wkb_3d_cmd = (1..=wkb_values.len())
        .into_iter()
        .map(|n| {
            format!(
                "\"{}\" u 1:2:3 i {} t \"WKB {}\" w l",
                output_file,
                n - 1,
                n
            )
        })
        .collect::<Vec<String>>()
        .join(", ");

    let airy_3d_cmd = (1..=airy_values.len())
        .into_iter()
        .map(|n| {
            format!(
                "\"{}\" u 1:2:3 i {} t \"Airy {}\" w l",
```

```
                output_file,
                n + wkb_values.len() - 1,
                n
            )
        })
        .collect::<Vec<String>>()
        .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(",␣");

    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
```

```
183         .write_all(format!("plot␣\"{}\"␣u␣1:2␣t␣\"Re({})\"␣w␣l", output_file, title).
                as_bytes())
184         .unwrap();
185
186     let mut plot_im_file = File::create("plot_im.gnuplot").unwrap();
187     plot_im_file
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
```

```
    let values_str = to_gnuplot_string(values);

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

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

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

pub fn plot_probability_super_pos(
    wave_function: &SuperPosition,
    output_dir: &Path,
    output_file: &str,
) {
    std::env::set_current_dir(&output_dir).unwrap();
    let values = evaluate_function_between(
        wave_function,
        wave_function.get_view().0,
        wave_function.get_view().1,
        NUMBER_OF_POINTS,
    )
    .par_iter()
    .map(|p| Point {
        x: p.x,
        y: p.y.norm_sqr(),
    })
    .collect();

    let values_str = to_gnuplot_string(values);

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

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

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

src/potentials.rs

```
use crate::*;

```

```
const ENERGY_INF: f64 = 1e6;

#[allow(unused)]
pub fn smooth_step(x: f64) -> f64 {
    const TRANSITION: f64 = 0.5;
    let step = Arc::new(Function::new(|x: f64| -> Complex64 {
        if x.abs() < 2.0 {
            complex(10.0, 0.0)
        } else {
            complex(0.0, 0.0)
        }
    }));
    let zero = Arc::new(Function::new(|_: f64| -> Complex64 { complex(0.0, 0.0) }));
    let inf = Arc::new(Function::new(|x: f64| -> Complex64 {
        if x.abs() > 5.0 {
            complex(ENERGY_INF, 0.0)
        } else {
            complex(0.0, 0.0)
        }
    }));

    let joint_inf_zero_l = wave_function_builder::Joint {
        left: inf.clone(),
        right: zero.clone(),
        cut: -5.0 + TRANSITION / 2.0,
        delta: TRANSITION,
    };

    let joint_zero_step_l = wave_function_builder::Joint {
        left: zero.clone(),
        right: step.clone(),
        cut: -2.0 + TRANSITION / 2.0,
        delta: TRANSITION,
    };

    let joint_zero_inf_r = wave_function_builder::Joint {
        left: zero.clone(),
        right: inf.clone(),
        cut: 5.0 - TRANSITION / 2.0,
        delta: TRANSITION,
    };

    let joint_step_zero_r = wave_function_builder::Joint {
        left: step.clone(),
        right: zero.clone(),
        cut: 2.0 - TRANSITION / 2.0,
        delta: TRANSITION,
    };

```

```
    if wave_function_builder::is_in_range(joint_zero_inf_r.range(), x) {
        return joint_zero_inf_r.eval(x).re;
    }

    if wave_function_builder::is_in_range(joint_inf_zero_l.range(), x) {
        return joint_inf_zero_l.eval(x).re;
    }

    if wave_function_builder::is_in_range(joint_step_zero_r.range(), x) {
        return joint_step_zero_r.eval(x).re;
    }

    if wave_function_builder::is_in_range(joint_zero_step_l.range(), x) {
        return joint_zero_step_l.eval(x).re;
    }

    return zero.eval(x).re.max(inf.eval(x).re.max(step.eval(x).re));
}

#[allow(unused)]
pub fn mexican_hat(x: f64) -> f64 {
    (x - 4.0).powi(2) * (x + 4.0).powi(2)
}

#[allow(unused)]
pub fn double_mexican_hat(x: f64) -> f64 {
    (x - 4.0).powi(2) * x.powi(2) * (x + 4.0).powi(2)
}

#[allow(unused)]
pub fn triple_mexican_hat(x: f64) -> f64 {
    (x - 6.0).powi(2) * (x - 3.0).powi(2) * (x + 3.0).powi(2) * (x + 6.0).powi(2)
}

pub fn square(x: f64) -> f64 {
    x * x
}
```

src/tui.rs

```
use std::io;

fn get_float_from_user(message: &str) -> f64 {
    loop {
        println!("{}", message);
        let mut input = String::new();

        // io::stdout().lock().write(message.as_ref()).unwrap();
        io::stdin()
```

```
            .read_line(&mut input)
            .expect("Not a valid string");
        println!("");
        let num = input.trim().parse();
        if num.is_ok() {
            return num.unwrap();
        }
    }
}

fn get_user_bounds() -> (f64, f64) {
    let user_bound_lower: f64 = get_float_from_user("Lower Bound: ");

    let user_bound_upper: f64 = get_float_from_user("Upper bound: ");
    return (user_bound_lower, user_bound_upper);
}
fn ask_user_for_view(lower_bound: Option<f64>, upper_bound: Option<f64>) -> (f64, f64
    ) {
    println!("Failed to determine boundary of the graph automatically.");
    println!("Pleas enter values manually.");
    lower_bound.map(|b| println!("(Suggestion for lower bound: {})", b));
    upper_bound.map(|b| println!("(Suggestion for upper bound: {})", b));

    return get_user_bounds();
}
```

src/turning_points.rs

```
use crate::cmp_f64;
use crate::newtons_method::*;
use crate::wkb_wave_func::*;
use crate::*;
use num::signum;

const MAX_TURNING_POINTS: usize = 256;
const ACCURACY: f64 = 1e-9;

pub struct TGroup {
    pub ts: Vec<((f64, f64), f64)>,
    // pub tn: Option<f64>,
}

impl TGroup {
    pub fn new() -> TGroup {
        TGroup { ts: vec![] }
    }

    pub fn add_ts(&mut self, new_t: ((f64, f64), f64)) {
        self.ts.push(new_t);
```

```
    }
}

fn validity_func(phase: Phase) -> Arc<dyn Fn(f64) -> f64> {
    Arc::new(move |x: f64| {
        1.0 / (2.0 * phase.mass).sqrt() * derivative(&|t| (phase.potential)(t), x).
            abs() * VALIDITY_LL_FACTOR
            - ((phase.potential)(x) - phase.energy).pow(2)
    })
}

fn group_ts(zeros: &Vec<f64>, phase: &Phase) -> TGroup {
    let mut zeros = zeros.clone();
    let valid = validity_func(phase.clone());

    zeros.sort_by(cmp_f64);
    let mut derivatives = zeros
        .iter()
        .map(|x| derivative(valid.as_ref(), *x))
        .map(signum)
        .zip(zeros.clone())
        .collect::<Vec<(f64, f64)>>();

    let mut groups = TGroup { ts: vec![] };

    if let Some((deriv, z)) = derivatives.first() {
        if *deriv < 0.0 {
            let mut guess = z - ACCURACY.sqrt();
            let mut new_deriv = *deriv;
            let mut missing_t = *z;

            while new_deriv < 0.0 {
                missing_t =
                    regula_falsi_bisection(valid.as_ref(), guess, -ACCURACY.sqrt(),
                        ACCURACY);
                new_deriv = signum(derivative(valid.as_ref(), missing_t));
                guess -= ACCURACY.sqrt();
            }

            derivatives.insert(
                0,
                (signum(derivative(valid.as_ref(), missing_t)), missing_t),
            );
        }
    }

    if let Some((deriv, z)) = derivatives.last() {
        if *deriv > 0.0 {
            let mut guess = z + ACCURACY.sqrt();
```

```
            let mut new_deriv = *deriv;
            let mut missing_t = *z;

            while new_deriv > 0.0 {
                missing_t =
                    regula_falsi_bisection(valid.as_ref(), guess, ACCURACY.sqrt(),
                        ACCURACY);
                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 {
    let zeros = find_zeros(phase, view);
    let groups = group_ts(&zeros, phase);
    println!(
        "Turning␣Points:␣{:.7?}",
        groups.ts.iter().map(|(_, t)| *t).collect::<Vec<f64>>()
    );
    return groups;
}

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()
            * VALIDITY_LL_FACTOR
            - ((phase_clone.potential)(x) - phase_clone.energy).pow(2)
    });
    let mut zeros = NewtonsMethodFindNewZero::new(validity_func, ACCURACY, 1e4 as
        usize);

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

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

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

src/utils.rs

```
use crate::newtons_method::derivative;
use crate::Complex64;
use std::cmp::Ordering;

pub fn cmp_f64(a: &f64, b: &f64) -> Ordering {
    if a < b {
        return Ordering::Less;
    } else if a > b {
        return Ordering::Greater;
    }
    return Ordering::Equal;
}

pub fn complex(re: f64, im: f64) -> Complex64 {
    return Complex64 { re, im };
}

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

```rust
}

pub fn identity(c: Complex64) -> Complex64 {
    c
}

pub fn conjugate(c: Complex64) -> Complex64 {
    c.conj()
}

pub fn negative(c: Complex64) -> Complex64 {
    -c
}

pub fn negative_conj(c: Complex64) -> Complex64 {
    -c.conj()
}

pub fn complex_compare(expect: Complex64, actual: Complex64, epsilon: f64) -> bool {
    let average = (expect.norm() + actual.norm()) / 2.0;
    return (expect - actual).norm() / average < epsilon;
}

pub fn float_compare(expect: f64, actual: f64, epsilon: f64) -> bool {
    let average = (expect + actual) / 2.0;

    if average < epsilon {
        return expect == actual;
    }

    return (expect - actual) / average < epsilon;
}

pub trait Func<A, R>: Sync + Send {
    fn eval(&self, x: A) -> R;
}

pub trait ReToC: Sync + Func<f64, Complex64> {}

pub trait ReToRe: Sync + Func<f64, f64> {}

pub struct Function<A, R> {
    pub(crate) f: fn(A) -> R,
}

impl<A, R> Function<A, R> {
    pub const fn new(f: fn(A) -> R) -> Function<A, R> {
        return Function { f };
    }
```

```
}

impl<A, R> Func<A, R> for Function<A, R> {
    fn eval(&self, x: A) -> R {
        (self.f)(x)
    }
}
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)
    }
}

fn get_wavefunc_exp_sign(x: f64) -> f64 {
    if -0.5 <= x && x <= 0.5 {
        return 1.0;
    } else {
        return -1.0;
    }
}
```

src/wave_function_builder.rs

```
use crate::wkb_wave_func::Phase;
use crate::*;
use ordinal::Ordinal;
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>;
}
```

```rust

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 {
            (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()
    }

    fn with_op(&self, op: fn(Complex64) -> Complex64) -> Box<dyn
        WaveFunctionPartWithOp> {
        Box::new(PureWkb {
            wkb: Arc::new(self.wkb.with_op(op)),
            range: self.range,
        })
    }
}

impl Func<f64, Complex64> for PureWkb {
    fn eval(&self, x: f64) -> Complex64 {
        self.wkb.eval(x)
    }
}

#[derive(Clone)]
struct ApproxPart {
    airy: Arc<AiryWaveFunction>,
    wkb: Arc<WkbWaveFunction>,
    airy_join_l: Joint,
    airy_join_r: Joint,
    range: (f64, f64),
}

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

```rust

impl WaveFunctionPartWithOp for ApproxPart {
    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()
    }

    fn with_op(&self, op: fn(Complex64) -> Complex64) -> Box<dyn
        WaveFunctionPartWithOp> {
        Box::new(ApproxPart::new(
            self.airy.with_op(op),
            self.wkb.with_op(op),
            self.range,
        ))
    }
}

impl ApproxPart {
    fn new(airy: AiryWaveFunction, wkb: WkbWaveFunction, range: (f64, f64)) ->
        ApproxPart {
        let airy_rc = Arc::new(airy);
        let wkb_rc = Arc::new(wkb);
        let delta = (airy_rc.ts.1 - airy_rc.ts.0) * AIRY_TRANSITION_FRACTION;
        ApproxPart {
            airy: airy_rc.clone(),
            wkb: wkb_rc.clone(),
            airy_join_l: Joint {
                left: wkb_rc.clone(),
                right: airy_rc.clone(),
                cut: airy_rc.ts.0 + delta / 2.0,
                delta: -delta,
            },
            airy_join_r: Joint {
                left: airy_rc.clone(),
                right: wkb_rc.clone(),
                cut: airy_rc.ts.1 - delta / 2.0,
                delta,
            },
            range,
        }
    }
}

impl Func<f64, Complex64> for ApproxPart {
    fn eval(&self, x: f64) -> Complex64 {
        if is_in_range(self.airy_join_l.range(), x) && ENABLE_AIRY_JOINTS {
```

```
            return self.airy_join_l.eval(x);
        } else if is_in_range(self.airy_join_r.range(), x) && ENABLE_AIRY_JOINTS {
            return self.airy_join_r.eval(x);
        } else if is_in_range(self.airy.ts, x) {
            return self.airy.eval(x);
        } else {
            return self.wkb.eval(x);
        }
    }
}

#[derive(Clone)]
pub struct WaveFunction {
    phase: Arc<Phase>,
    view: (f64, f64),
    parts: Vec<Arc<dyn WaveFunctionPart>>,
    airy_ranges: Vec<(f64, f64)>,
    wkb_ranges: Vec<(f64, f64)>,
    scaling: Complex64,
}

fn sign_match(f1: f64, f2: f64) -> bool {
    return f1.signum() == f2.signum();
}

fn sign_match_complex(mut c1: Complex64, mut c2: Complex64) -> bool {
    if c1.re.abs() < c1.im.abs() {
        c1.re = 0.0;
    }

    if c1.im.abs() < c1.re.abs() {
        c1.im = 0.0;
    }

    if c2.re.abs() < c2.im.abs() {
        c2.re = 0.0;
    }

    if c2.im.abs() < c2.re.abs() {
        c2.im = 0.0;
    }

    return sign_match(c1.re, c2.re) && sign_match(c1.im, c2.im);
}

impl WaveFunction {
    pub fn get_energy(&self) -> f64 {
        self.phase.energy
    }
```

```
215
216     pub fn new<F: Fn(f64) -> f64 + Sync + Send>(
217         potential: &'static F,
218         mass: f64,
219         n_energy: usize,
220         approx_inf: (f64, f64),
221         view_factor: f64,
222         scaling: ScalingType,
223     ) -> WaveFunction {
224         let energy = energy::nth_energy(n_energy, mass, &potential, approx_inf);
225         println!("{} Energy: {:.9}", Ordinal(n_energy).to_string(), energy);
226
227         let lower_bound = newtons_method::newtons_method_max_iters(
228             &|x| potential(x) - energy,
229             approx_inf.0,
230             1e-7,
231             100000,
232         );
233         let upper_bound = newtons_method::newtons_method_max_iters(
234             &|x| potential(x) - energy,
235             approx_inf.1,
236             1e-7,
237             100000,
238         );
239
240         let view = if lower_bound.is_some() && upper_bound.is_some() {
241             (
242                 lower_bound.unwrap() * (upper_bound.unwrap() - lower_bound.unwrap())
                        * view_factor,
243                 upper_bound.unwrap() * (upper_bound.unwrap() - lower_bound.unwrap())
                        * view_factor,
244             )
245         } else {
246             println!("Failed to determine view automatically, using APPROX_INF as
                    view");
247             (
248                 approx_inf.0 - f64::EPSILON.sqrt(),
249                 approx_inf.1 + f64::EPSILON.sqrt(),
250             )
251         };
252
253         let phase = Arc::new(Phase::new(energy, mass, potential));
254
255         let (airy_wave_funcs, boundaries) = AiryWaveFunction::new(phase.clone(), (
                view.0, view.1));
256         let (parts, airy_ranges, wkb_ranges): (
257             Vec<Arc<dyn WaveFunctionPart>>,
258             Vec<(f64, f64)>,
259             Vec<(f64, f64)>,
```

```
        ) = if boundaries.ts.len() == 0 {
            println!("No turning points found in view! Results might be in accurate")
                ;
            let wkb1 = WkbWaveFunction::new(
                phase.clone(),
                1.0.into(),
                INTEG_STEPS,
                approx_inf.0,
                approx_inf.0,
                f64::consts::PI / 4.0,
            );
            let wkb2 = WkbWaveFunction::new(
                phase.clone(),
                1.0.into(),
                INTEG_STEPS,
                approx_inf.0,
                approx_inf.1,
                f64::consts::PI / 4.0,
            );

            let center = (view.0 + view.1) / 2.0;
            let wkb1 = Box::new(PureWkb {
                wkb: Arc::new(wkb1),
                range: (approx_inf.0, center),
            });

            let wkb2 = Box::new(PureWkb {
                wkb: Arc::new(wkb2),
                range: (center, approx_inf.1),
            });

            let wkb1_range = wkb1.range();
            (
                vec![
                    Arc::from(wkb1.as_wave_function_part()),
                    Arc::from(wkb2.as_wave_function_part()),
                ],
                vec![],
                vec![wkb1_range, wkb2.range()],
            )
        } else {
            let turning_points: Vec<f64> = [
                vec![2.0 * approx_inf.0 - boundaries.ts.first().unwrap().1],
                boundaries.ts.iter().map(|p| p.1).collect(),
                vec![2.0 * approx_inf.1 - boundaries.ts.last().unwrap().1],
            ]
            .concat();

            let wave_funcs = turning_points
```

```
                .iter()
                .zip(turning_points.iter().skip(1))
                .zip(turning_points.iter().skip(2))
                .map(
                    |((previous, boundary), next)| -> (WkbWaveFunction, (f64, f64)) {
                        (
                            if derivative(phase.potential.as_ref(), *boundary) > 0.0
                                {
                                WkbWaveFunction::new(
                                    phase.clone(),
                                    1.0.into(),
                                    INTEG_STEPS,
                                    *boundary,
                                    *previous,
                                    f64::consts::PI / 4.0,
                                )
                            } else {
                                WkbWaveFunction::new(
                                    phase.clone(),
                                    1.0.into(),
                                    INTEG_STEPS,
                                    *boundary,
                                    *boundary,
                                    f64::consts::PI / 4.0,
                                )
                            },
                            ((boundary + previous) / 2.0, (next + boundary) / 2.0),
                        )
                    },
                )
                .collect::<Vec<(WkbWaveFunction, (f64, f64))>>();

            let wkb_airy_pair: Vec<(&(WkbWaveFunction, (f64, f64)), AiryWaveFunction)
                > = wave_funcs
                .iter()
                .zip(airy_wave_funcs.iter())
                .map(|(w, a)| {
                    (
                        w,
                        a.with_phase_off(w.0.phase_off)
                            .with_c(w.0.get_exp_sign().into()),
                    )
                })
                .collect();

            let wkb_ranges = wkb_airy_pair
                .iter()
                .map(|((_, wkb_range), _)| *wkb_range)
                .collect();
```

```
            let airy_ranges = wkb_airy_pair.iter().map(|(_, airy)| airy.ts).collect()
                ;

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

            (
                approx_parts
                    .iter()
                    .map(|p| Arc::from(p.as_wave_function_part()))
                    .collect(),
                airy_ranges,
                wkb_ranges,
            )
        };

        match scaling {
            ScalingType::Mul(s) => WaveFunction {
                phase,
                view,
                parts,
                airy_ranges,
                wkb_ranges,
                scaling: s,
            },
            ScalingType::None => WaveFunction {
                phase,
                view,
                parts,
                airy_ranges,
                wkb_ranges,
                scaling: complex(1.0, 0.0),
            },
            ScalingType::Renormalize(s) => {
                let unscaled = WaveFunction {
                    phase: phase.clone(),
                    view,
                    parts: parts.clone(),
                    airy_ranges: airy_ranges.clone(),
                    wkb_ranges: wkb_ranges.clone(),
                    scaling: s,
                };
                let factor = renormalize_factor(&unscaled, approx_inf);
                WaveFunction {
                    phase,
```

```
                    view,
                    parts,
                    airy_ranges,
                    wkb_ranges,
                    scaling: s * factor,
                }
            }
        }
    }

    pub fn calc_psi(&self, x: f64) -> Complex64 {
        for part in self.parts.as_slice() {
            if is_in_range(part.range(), x) {
                return part.eval(x);
            }
        }
        panic!(
            "[WkbWaveFunction::calc_psi] x out of range (x = {}, ranges: {:#?})",
            x,
            self.parts
                .iter()
                .map(|p| p.range())
                .collect::<Vec<(f64, f64)>>()
        );
    }

    pub fn get_airy_ranges(&self) -> &[(f64, f64)] {
        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 {
    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", Ordinal(*e).to_string());
                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,
                };
                let factor = renormalize_factor(&unscaled, approx_inf);
                println!("factor:␣{}", factor);
                SuperPosition {
                    wave_funcs,
                    scaling: s * factor,
                }
            }
        }
    }

    pub fn get_view(&self) -> (f64, f64) {
        let view_a = self
            .wave_funcs
            .iter()
            .map(|w| w.get_view().0)
            .min_by(cmp_f64)
            .unwrap();
        let view_b = self
            .wave_funcs
            .iter()
            .map(|w| w.get_view().1)
            .max_by(cmp_f64)
```

```
            .unwrap();
        (view_a, view_b)
    }
}

impl Func<f64, Complex64> for SuperPosition {
    fn eval(&self, x: f64) -> Complex64 {
        self.scaling * self.wave_funcs.iter().map(|w| w.eval(x)).sum::<Complex64>()
    }
}

struct Scaled<A, R>
where
    R: std::ops::Mul<R, Output = R> + Sync + Send + Clone,
{
    scale: R,
    func: Box<dyn Func<A, R>>,
}

impl<A, R> Func<A, R> for Scaled<A, R>
where
    R: std::ops::Mul<R, Output = R> + Sync + Send + Clone,
{
    fn eval(&self, x: A) -> R {
        self.func.eval(x) * self.scale.clone()
    }
}

fn renormalize_factor(wave_func: &dyn Func<f64, Complex64>, approx_inf: (f64, f64))
    -> f64 {
    1.0 / integrate(
        evaluate_function_between(
            wave_func,
            approx_inf.0 * (1.0 - f64::EPSILON),
            approx_inf.1 * (1.0 - f64::EPSILON),
            INTEG_STEPS,
        )
        .par_iter()
        .map(|p| Point {
            x: p.x,
            y: p.y.norm_sqr(),
        })
        .collect(),
        TRAPEZE_PER_THREAD,
    )
}

pub fn renormalize(
    wave_func: Box<dyn Func<f64, Complex64>>,
```

```
    approx_inf: (f64, f64),
) -> Box<dyn Func<f64, Complex64>> {
    let area = renormalize_factor(wave_func.as_ref(), approx_inf);
    return Box::new(Scaled::<f64, Complex64> {
        scale: area.into(),
        func: wave_func,
    });
}

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

    #[test]
    fn sign_check_complex_test() {
        let range = (-50.0, 50.0);
        let n = 100000;
        for ri1 in 0..n {
            for ii1 in 0..n {
                for ri2 in 0..n {
                    for ii2 in 0..n {
                        let re1 = index_to_range(ri1 as f64, 0.0, n as f64, range.0,
                            range.1);
                        let im1 = index_to_range(ii1 as f64, 0.0, n as f64, range.0,
                            range.1);
                        let re2 = index_to_range(ri2 as f64, 0.0, n as f64, range.0,
                            range.1);
                        let im2 = index_to_range(ii2 as f64, 0.0, n as f64, range.0,
                            range.1);

                        assert_eq!(
                            sign_match_complex(complex(re1, im1), complex(re2, im2)),
                            sign_match_complex(complex(re2, im2), complex(re1, im1))
                        );
                    }
                }
            }
        }
    }
}
```

src/wkb_wave_func.rs

```
use crate::*;
use std::fmt::Display;
use std::sync::Arc;

#[derive(Clone)]
pub struct Phase {
```

```rust
    pub energy: f64,
    pub mass: f64,
    pub potential: Arc<dyn Fn(f64) -> f64 + Send + Sync>,
}

impl Display for Phase {
    fn fmt(&self, f: &mut std::fmt::Formatter<'_>) -> std::fmt::Result {
        write!(
            f,
            "Phase␣{{energy:␣{},␣mass:␣{},␣potential:␣[func]}}",
            self.energy, self.mass
        )
    }
}

impl Phase {
    fn default() -> Phase {
        Phase {
            energy: 0.0,
            mass: 0.0,
            potential: Arc::new(|_x| 0.0),
        }
    }

    pub fn new<F: Fn(f64) -> f64 + Sync + Send>(
        energy: f64,
        mass: f64,
        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()
    }
}

```

```rust
#[derive(Clone)]
pub struct WkbWaveFunction {
    pub c: Complex64,
    pub turning_point_exp: f64,
    pub turning_point_osc: 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_exp: self.turning_point_exp,
            turning_point_osc: self.turning_point_osc,
            phase: self.phase.clone(),
            integration_steps: self.integration_steps,
            op: self.op,
            phase_off: self.phase_off,
        }
    }

    pub fn new(
        phase: Arc<Phase>,
        c: Complex64,
        integration_steps: usize,
        turning_point_exp: f64,
        turning_point_osc: f64,
        phase_off: f64,
    ) -> WkbWaveFunction {
        return WkbWaveFunction {
            c,
            turning_point_exp,
            turning_point_osc,
            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_exp: self.turning_point_exp,
            turning_point_osc: self.turning_point_osc,
            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)
    }

    pub fn get_exp_sign(&self) -> f64 {
        let limit_sign = if self.turning_point_exp == self.turning_point_osc {
            1.0
        } else {
            -1.0
        };

        self.psi_osc(self.turning_point_exp + limit_sign * f64::EPSILON.sqrt())
            .re
            .signum()
    }

    fn psi_osc(&self, x: f64) -> Complex64 {
        let integral = integrate(
            evaluate_function_between(
                self.phase.as_ref(),
                x,
                self.turning_point_osc,
                self.integration_steps,
            ),
            TRAPEZE_PER_THREAD,
        );
        -self.c * complex((integral - self.phase_off).cos(), 0.0) / self.phase.
            sqrt_momentum(x)
    }

    fn psi_exp(&self, x: f64) -> Complex64 {
        let integral = integrate(
            evaluate_function_between(
                self.phase.as_ref(),
                x,
                self.turning_point_exp,
                self.integration_steps,
            ),
            TRAPEZE_PER_THREAD,
```

```
        );
        let exp_sign = self.get_exp_sign();

        exp_sign * (self.c * 0.5 * (-integral.abs()).exp())
    }
}

impl Func<f64, Complex64> for WkbWaveFunction {
    fn eval(&self, x: f64) -> Complex64 {
        let val = if self.phase.energy < (self.phase.potential)(x) {
            self.psi_exp(x)
        } else {
            self.psi_osc(x)
        };

        return (self.op)(val);
    }
}

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

    fn pot(x: f64) -> f64 {
        1.0 / (x * x)
    }

    fn pot_in(x: f64) -> f64 {
        1.0 / x.sqrt()
    }

    #[test]
    fn phase_off() {
        let energy_cond = |e: f64| -> f64 { (0.5 * (e - 0.5)) % 1.0 };

        let integ = Function::<f64, f64>::new(energy_cond);
        let mut values = evaluate_function_between(&integ, 0.0, 5.0, NUMBER_OF_POINTS
            );
        let sort_func =
            |p1: &Point<f64, f64>, p2: &Point<f64, f64>| -> Ordering { cmp_f64(&p1.x,
                &p2.x) };
        values.sort_by(sort_func);

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

        let data_str: String = values
            .par_iter()
            .map(|p| -> String { format!("{}␣{}\n", p.x, p.y) })
```

```
            .reduce(|| String::new(), |s: String, current: String| s + &*current);

        data_file.write_all((data_str).as_ref()).unwrap()
    }
}
```

lib/build.sh

```
#! /bin/bash

go get main
go build -o libairy.a -buildmode=c-archive main.go
```

lib/go.mod

```
module main

go 1.18

require gonum.org/v1/gonum v0.11.0
```

lib/main.go

```
package main

import "C"
import "gonum.org/v1/gonum/mathext"

//export airy_ai
func airy_ai(zr float64, zi float64) (float64, float64) {
    z := mathext.AiryAi(complex(zr, zi))
    return real(z), imag(z)
}

func main() {

}
```

build.rs

```
use std::env;
use std::path::PathBuf;
use std::process::Command;

fn main() {
    Command::new("sh")
        .arg("build.sh")
        .current_dir("./lib/")
```

```
        .status()
        .unwrap();

    let path = "./lib";
    let lib = "airy";

    println!("cargo:rustc-link-search=native={}", path);
    println!("cargo:rustc-link-lib=static={}", lib);

    // The bindgen::Builder is the main entry point
    // to bindgen, and lets you build up options for
    // the resulting bindings.
    let bindings = bindgen::Builder::default()
        // The input header we would like to generate
        // bindings for.
        .header("lib/libairy.h")
        // Tell cargo to invalidate the built crate whenever any of the
        // included header files changed.
        .parse_callbacks(Box::new(bindgen::CargoCallbacks))
        // Finish the builder and generate the bindings.
        .generate()
        // Unwrap the Result and panic on failure.
        .expect("Unable␣to␣generate␣bindings");

    // Write the bindings to the $OUT_DIR/bindings.rs file.
    let out_path = PathBuf::from(env::var("OUT_DIR").unwrap());
    bindings
        .write_to_file(out_path.join("bindings.rs"))
        .expect("Couldn't␣write␣bindings!");
}
```

Cargo.toml

```
[package]
name = "shroedinger_approx"
version = "0.1.0"
edition = "2021"

# See more keys and their definitions at https://doc.rust-lang.org/cargo/reference/
    manifest.html

[dependencies]
num = "0.4.0"
tokio = { version = "1.0.3", features = ["full"] }
rayon = "1.5.3"
scilib = "0.5.0"
ordinal = "0.3.1"

[build-dependencies]
```

```
bindgen = "0.60.1"
```

energy.wsl

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

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

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

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

exact.wsl

```
c1 = -5.0
c2 = 1.0
numberOfPoints = 1000
m = 2
n = 5
viewFactor = 1.5

V[x_] := x^2

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

view = Solve[energy == V[x], x]
view = Function[l, x /. l] /@ view
view = Function[x, x*viewFactor] /@ view


Print["Energy = ", energy]
Print["view = ", view]


solution := DSolve[{V[x] psi[x] - psi''[x]/(2 m) == energy psi[x]}, psi[x], x]
psi[x_] = psi[x] /. solution[[1]] /. C[1] -> c1 /. C[2] -> c2

Print["psi[x] = ", psi[x]]

```

```
(*psi[x_] = c2*ParabolicCylinderD[(-1 - 50*Sqrt[m])/2, *)
          (*I*2^(3/4)*m^(1/4)*x] + c1*ParabolicCylinderD[(-1 + 50*Sqrt[m])/2, *)
          (*2^(3/4)*m^(1/4)*x]*)



step = (Abs[view[[1]]] + Abs[view[[2]]]) / numberOfPoints


vals = Table[{x, N@psi[x]}, {x, view[[1]], view[[2]], step}]
vals = Function[p, {p[[1]], Re[p[[2]]], Im[p[[2]]]}] /@ vals
Print["psi[0] = ", psi[0]]

total = N@Integrate[Re[psi[x]]^2 + Im[psi[x]]^2, {x, -Sqrt[energy], Sqrt[energy]}]

Print["area under solution = ", total]
total = N@Integrate[Abs[psi[x]], {x, -Sqrt[energy], Sqrt[energy]}]
Print["area under solution after renormalization = ", N@Integrate[Re[psi[x]]^2 + Im[
    psi[x]]^2, {x, -Sqrt[energy], Sqrt[energy]}]]

vals = Function[p, {p[[1]], p[[2]] / total, p[[3]] / total}] /@ vals

csv = ExportString[vals, "CSV"]
csv = StringReplace[csv, "," -> " "]
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.

Robert G. Littlejohn. Physics 221A, 2020. URL `url{https://www.pas.rochester.edu/~passage/resources/prelim/Quantum/UCB%20Notes/7%20wkb.pdf}`.

Erwin Schrödinger. Die gegenwärtige Situation in der Quantenmechanik. *Naturwissenschaften*, 23, 1935.

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