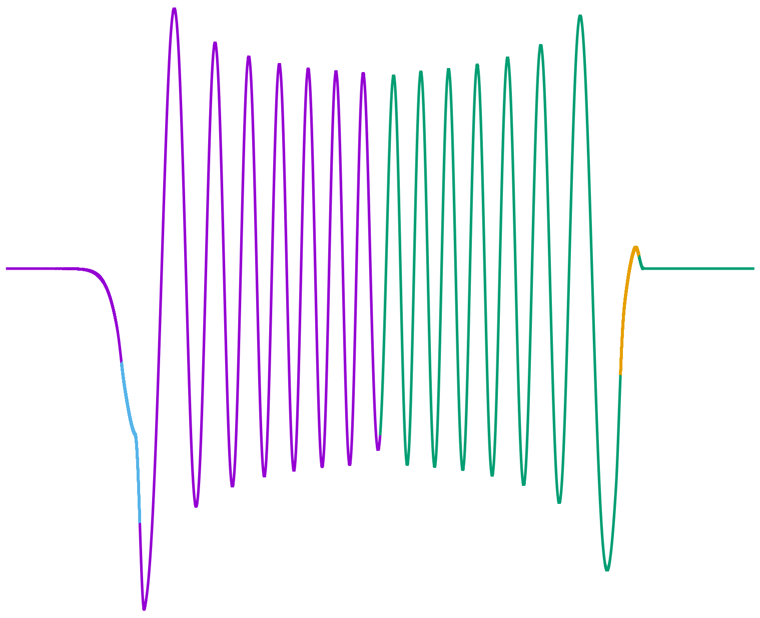
Approximating Solutions of the Time Independent Schrödinger Equation

Gian Laager

November 3, 2022



Maturaarbeit
Kantonsschule Glarus
Betreuer: Linus Romer
Referent: Elena Borisova

Contents

Vorwort

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

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

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

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

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

1 Introduction

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

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

1.1 Goals

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

2 Preliminary

2.1 Schrödinger Equation

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

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

Or more general

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

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

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

or

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

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

2.2 Rust

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

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

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

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

2.3 Interpretation of Quantum Mechanics

The author believes in the many worlds interpretation of Hugh Everett. *"The wave interpretation. This is the position proposed in the present thesis, in which the wave function itself is held to be the fundamental entity, obeying at all times a deterministic wave equation."* (**?**, 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 (**?**, 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 (**?**, p. 111).

2.4 Complex Numbers

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

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

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

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

2.5 Gnuplot

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

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

to plot the real part of the wave function, or

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

to see the full complex wave function.

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

2.6 Planck Units

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

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

This means that the constants will usually cancel out.

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

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

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

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

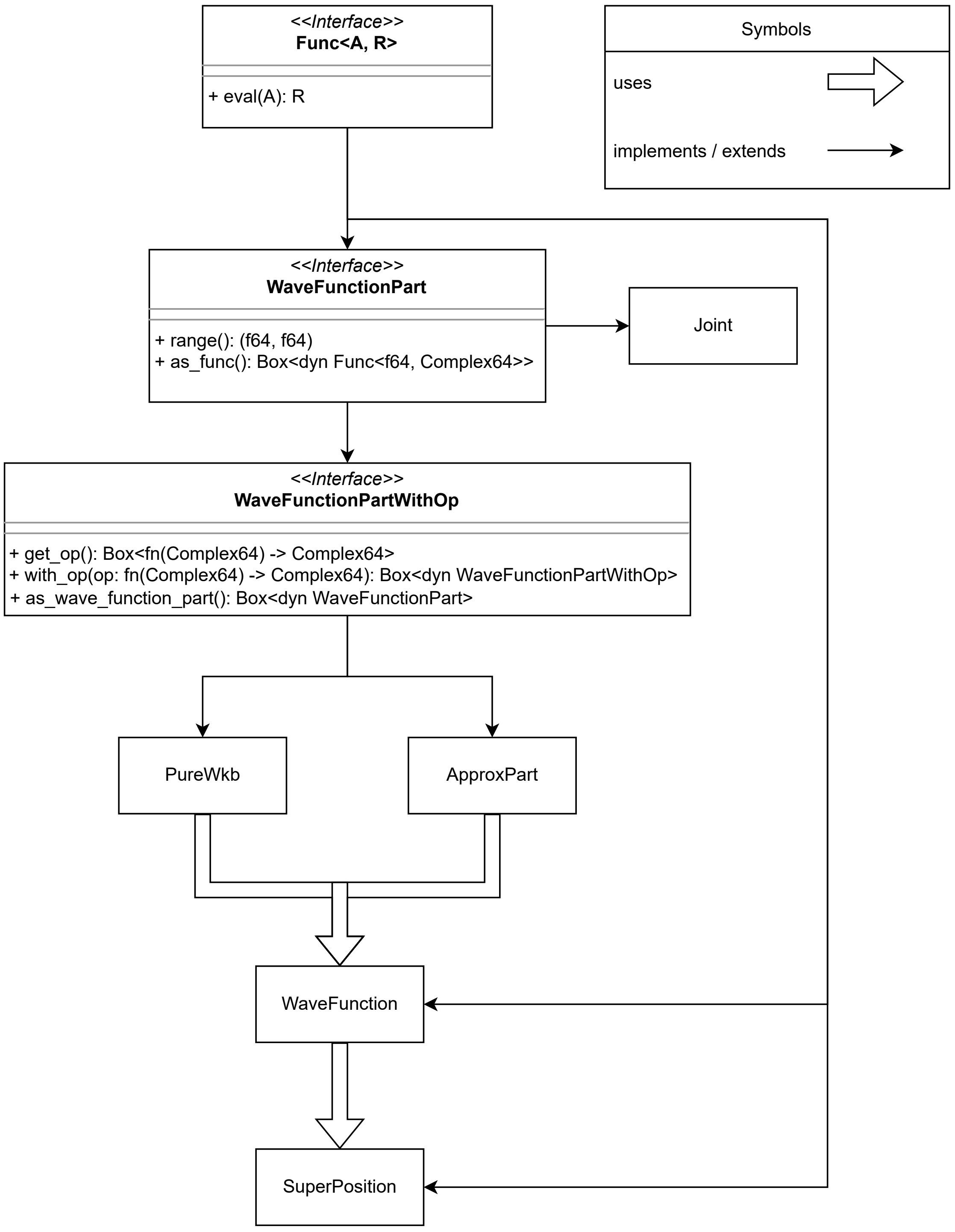
(?, Table 1)

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.



<<Interface>>
Func<A, R>
+ eval(A): R
Symbols
uses
implements / extends
<<Interface>>
WaveFunctionPart
+ range(): (f64, f64)
+ as_func(): Box<dyn Func<f64, Complex64>>
Joint
<<Interface>>
WaveFunctionPartWithOp
+ get_op(): Box<fn(Complex64) -> Complex64>
+ with_op(op: fn(Complex64) -> Complex64): Box<dyn WaveFunctionPartWithOp>
+ as_wave_function_part(): Box<dyn WaveFunctionPart>
PureWkb
ApproxPart
WaveFunction
SuperPosition

Figure 3.1: UML diagram of program architecture

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

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

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

3.2 Newtons Method

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

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

The sequence is defined as

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

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

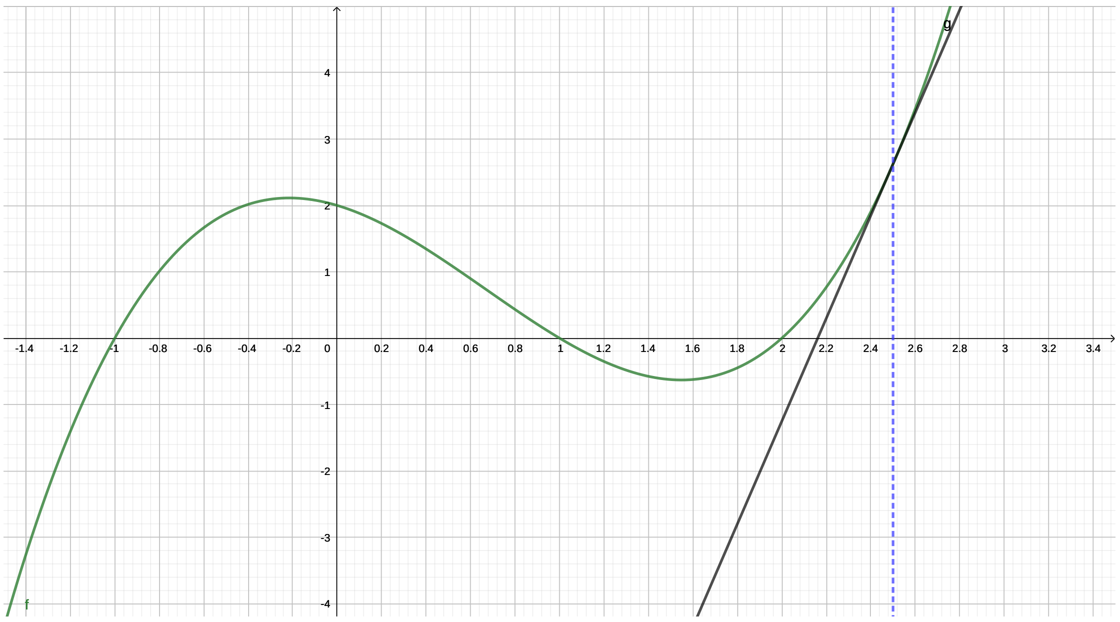


Figure 3.2: Illustration of Newtons method, $f(x) = (x-1)(x-2)(x+1)$.

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

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

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

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

3.3 Regula Falsi with Bisection

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

The algorithm itself is quite simple. To start we need

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

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

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

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

3.4 Derivatives

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

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

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

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

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

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

3.5 Integration

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

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

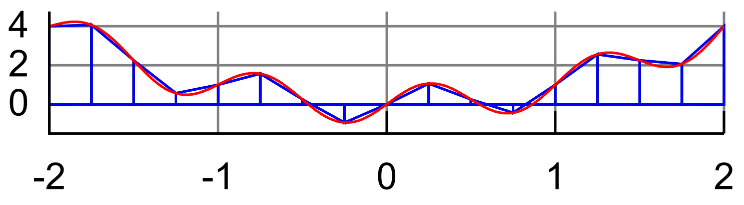


Figure 3.3: Illustration of integration with trapeze from **?**.

Figure 3.3 shows visually how the methods work, each blue trapeze from start (a) to end (b) has an area of

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

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

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

```rust
pub trait ReToC: Sync {
    fn eval(&self, x: &f64) -> Complex64;
}

pub struct Point {
    pub x: f64,
    pub y: Complex64,
}
```

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

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

```rust
pub fn evaluate_function_between(f: &dyn ReToC, a: f64, b: f64, n: usize) -> Vec<
    Point> {
    if a == b {
        return vec![];
    }

    (0..n)
        .into_par_iter()
        .map(|i| index_to_range(i as f64, 0.0, n as f64 - 1.0, a, b))
        .map(|x| Point { x, y: f.eval(&x) })
        .collect()
}
```

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

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

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

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

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

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

    let mut rest = complex(0.0, 0.0);

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

    return parallel + rest;
}
```

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

3.6 Transition Regions

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

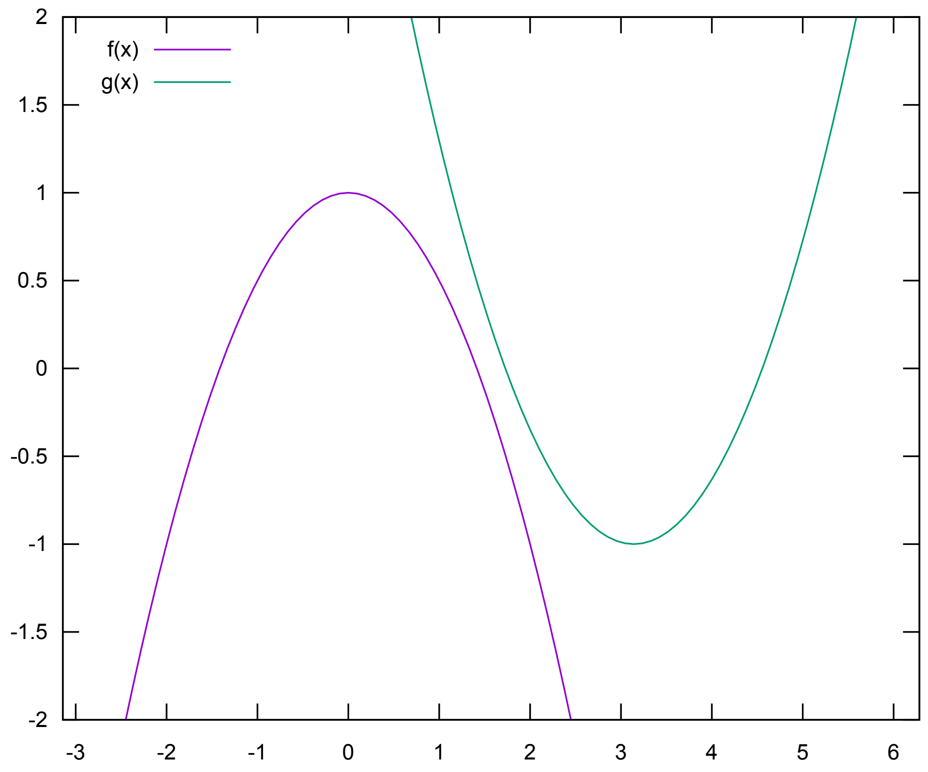


Figure 3.4: Example for joining functions

Lets consider an example, in figure 3.6 we can see two Taylor series of cosine. Now we have to join the two functions at $x = \pi/2$ such that its a mathematically smooth transition.

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

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

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

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

This gives us 3.6 which is obviously not smooth.

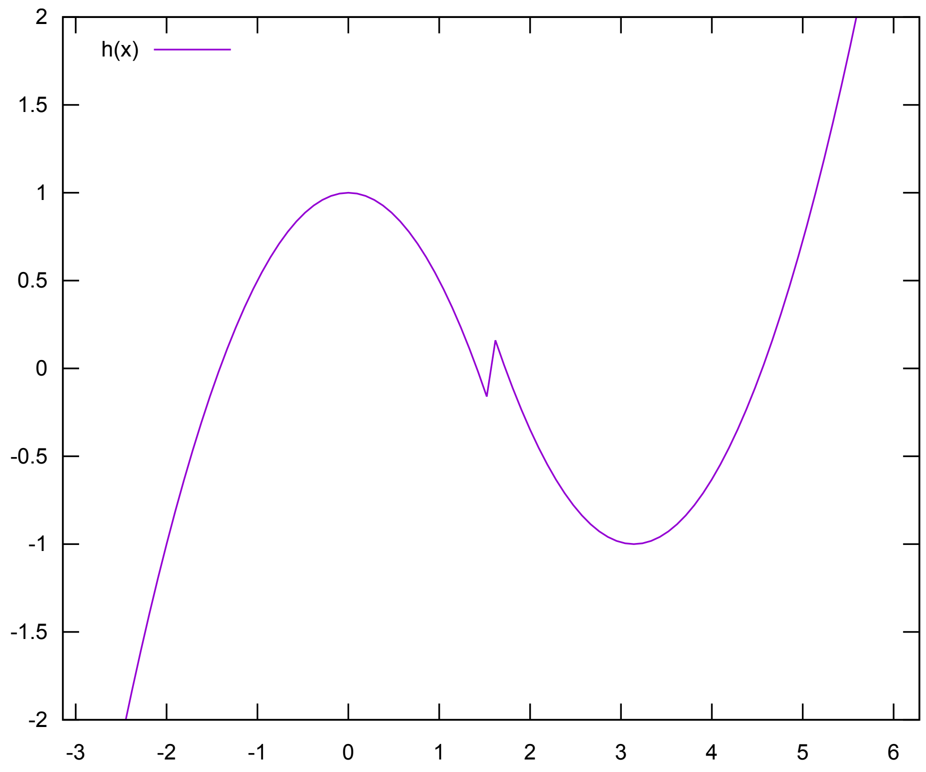


Figure 3.5: Plot of h(x) with step joint

If we use the formula from (**?**, 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 **??**).

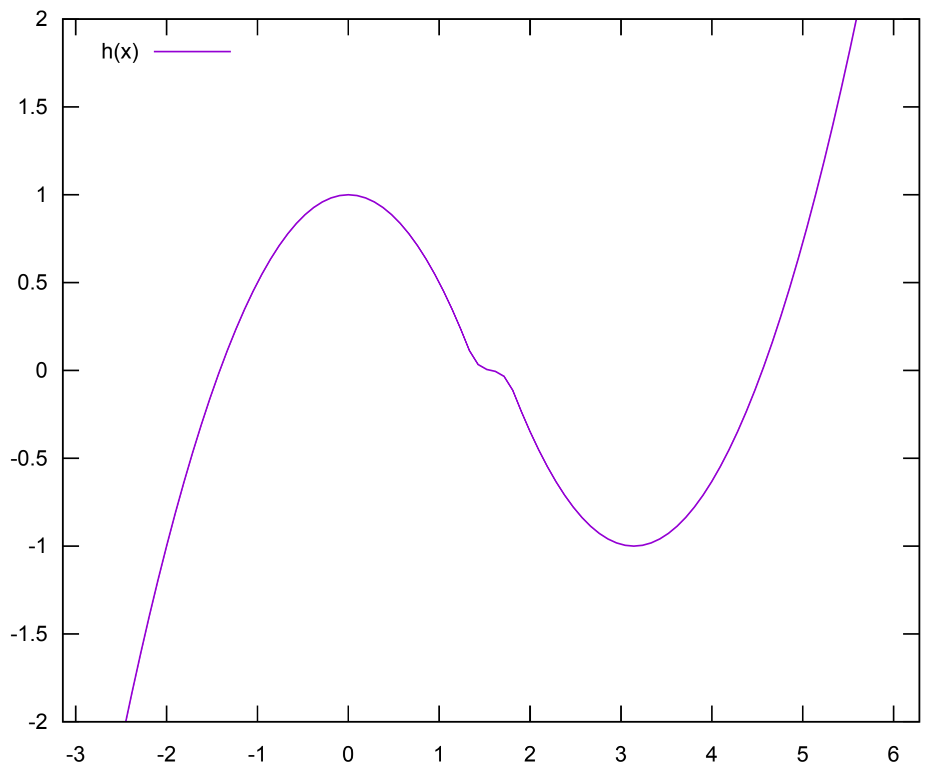


Figure 3.6: Plot of h(x) with Hall joint

3.6.1 Implementation in Rust

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

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

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

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 (**?**). It can be interpreted such that the oscillating part of the wave function has to complete all half oscillation.

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

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

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

Unfortunately 4.5 is not continuous which means that Newtons method can't be applied. Further on the bounds of integration have to be finite, this means the user of the program will have to specify a value for the constant `APPROX_INF` where any value for x out side of that range should satisfy $V(x) > E$. But it shouldn't be to big since the `integrate` function can only evaluate a relatively small number (default 64000) of trapezes before the performance will suffer enormously. The default value for `APPROX_INF` is `(-200.0, 200.0)`.

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

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

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

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

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

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

4.1.1 Accuracy

For a benchmark we will use

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

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

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

```

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

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

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

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

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

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

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

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

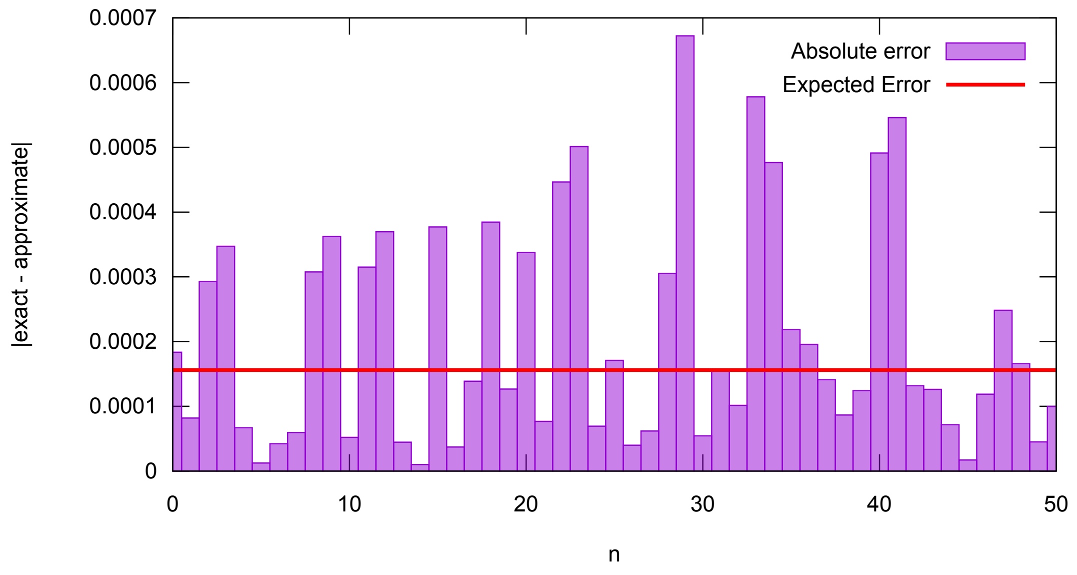


Figure 4.1: Absolute error of energy levels in square potential

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

4.2 Approximation Scheme

There are mainly three approximation methods used to solve for the actual wave function itself. There is perturbation theory which breaks the problem down in to ever smaller sub-problems that then can be solved exactly. This can be achieved by adding something to the Hamiltonian operator $\hat{H}$ which can then be solved exactly. But *perturbation theory is inefficient compared to other approximation methods when calculated on a computer* (**?**, 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 (**?**, p. 305).

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

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

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

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

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

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

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

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

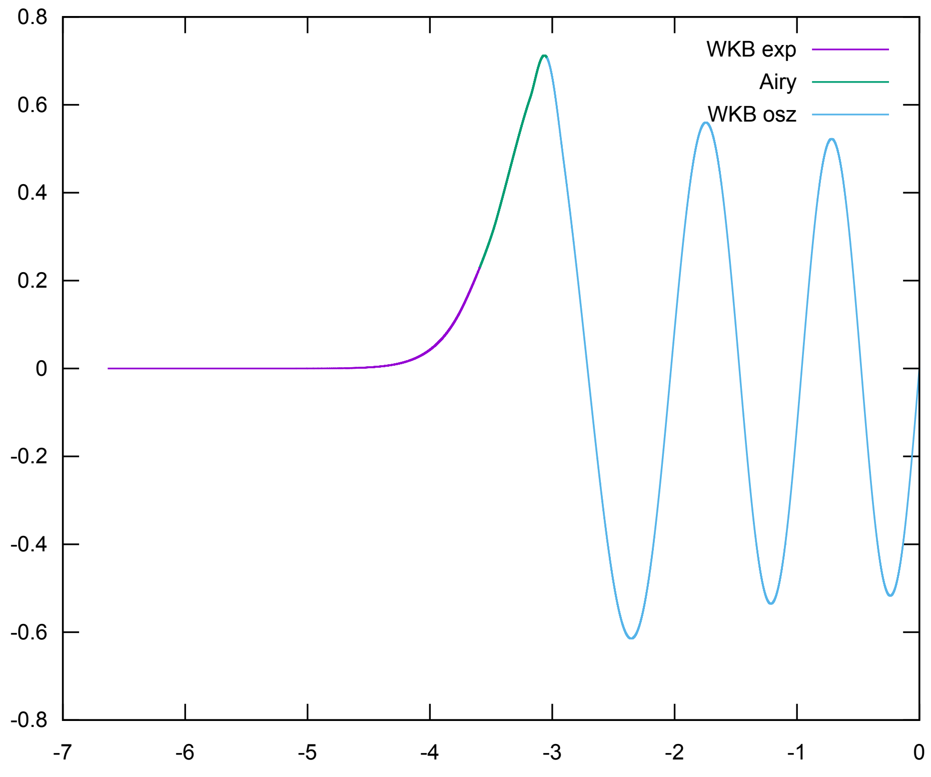


Figure 4.2: Left half of wave function with $N_{Energy} = 5 \Rightarrow E = 11.0$, $m = 2$, $V(x) = x^2$

In figure 4.2 the three parts are visualized. The purple section on the left is the exponential decaying part $\psi_{exp}^{WKB}(x)$, equation 4.8 is a modified version of the original version as described in (**?**, p. 317, eq. 15.25) where b and a are different solutions for t of equation 4.7. The absolute symbol makes it possible to not differentiate between the case where $x < t$ and $x > t$. Further on a factor of $e^{\delta \mathrm{i}}$ was added such that the imaginary part of $\psi_{exp}^{WKB}(x)$ is the same as in $\psi_{osz}^{WKB}(x)$.

The blue part on the right is $\psi_{osz}^{WKB}(x)$. Again equation 4.9 was expanded to result in the more general complex solution and it also works for both ψ_1 and ψ_2 in (**?**, p. 316-317, Claim 15.7). **?** assumes that $\delta = \pi/4$ which doesn't work in the simple case of $V(x) = x^2$, in figure 4.2 $\delta = 0$ was used. This will be further discussed in section **??**.

4.2.1 Validity

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

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

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

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

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

4.2.2 Implementation

4.3 Turning Points

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

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

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

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

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

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

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

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

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

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

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

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

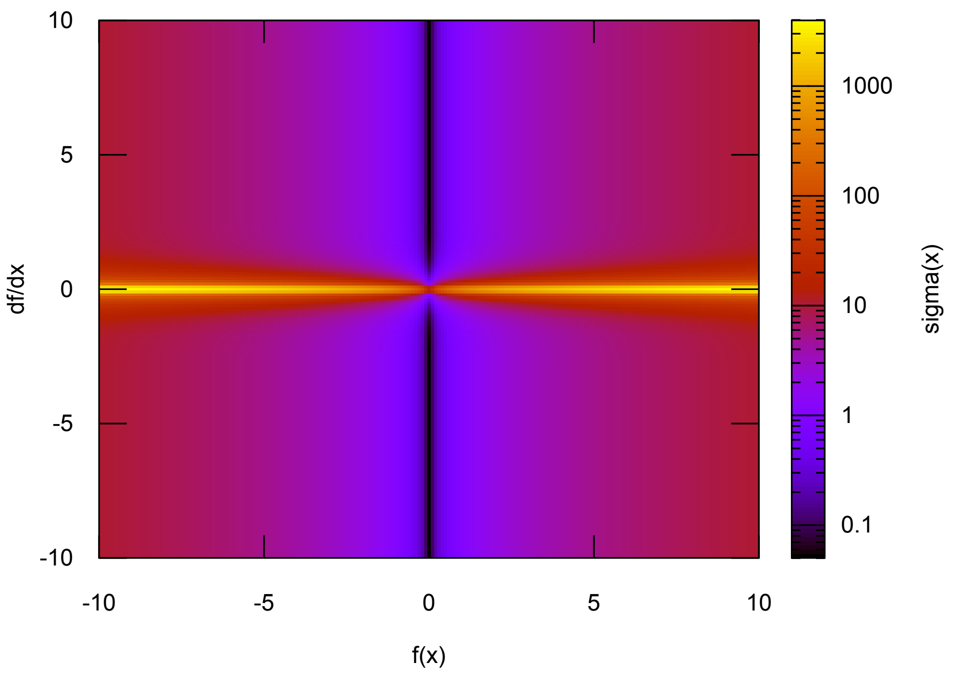


Figure 4.3: Logarithmic heat diagram of $\sigma(x)$, darker/bluer is better

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

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

```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

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

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

All that we need is a range in which the function is valid. Also (?, p. 317) states that the oscillating parts of WKB might have to negated, because of this we will also introduce the

`WaveFunctionPartWithOp` trait. In addition to the wave function itself it will also store and operation as negate for example.

```rust
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>;
}
```

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

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

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

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

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

```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()
}
```

This will be done in the `find_best_op` function where we choose the operation that has the minimal error both with respect to the derivative and the value of the wave function.

Mathematically we choose the operation $o(x)$ for which

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

is minimal, $\psi_0(x)$ is the wave function to the left and $\psi_1(x)$ is the wave function to the right of the middle of the view. The next step is to introduce joints in the middle of the view. After this we end up with the whole wave function.

In the other case where there actually are turning points, we will iterate over all of them and construct a `ApproxPart` from them. In order to define a range for the `ApproxPart` we will add two more turning points at `APPROX_INF.0` and `APPROX_INF.1` Then we can iterate over all the turning points in triplets and construct a `WkbWaveFunction` at the middle turning point. Once we've constructed all the `WkbWaveFunctions` we can combine them with the `AiryWaveFunctions` to construct the `ApproxParts` Just like in the case with no turning points we have to find an operation for each `ApproxPart` using the `find_best_op_wave_func_part` function.

In case `ENABLE_WKB_JOINTS` is `true` we also have to introduce joints between all the `ApproxPart` s.

It has previously been mentioned that $\delta = \pi/4$ does not work for all potentials. An attempt to correct this is implemented in the `calc_phase_offset` function. When we consider the point where two `WkbWaveFunction` meet the condition

$$\exp\left(\mathrm{i}\int_x^{t_0} \sqrt{2m(|E-V(y)|)}dy + \mathrm{i}\delta\right) = \exp\left(\mathrm{i}\int_x^{t_1} \sqrt{2m(|E-V(y)|)}dy - \mathrm{i}\delta\right)$$

has to hold for both turning points t_0 and t_1. To make it simpler lets use

$$a = \int_x^{t_0} \sqrt{2m(|E-V(y)|)}dy \tag{4.15}$$

$$b = \int_x^{t_1} \sqrt{2m(|E-V(y)|)}dy \tag{4.16}$$

therefor

$$e^{(a+\delta)\mathrm{i}} = e^{(b-\delta)\mathrm{i}}.$$

When we solve for δ we get

$$\delta = \frac{b-a}{2}.$$

Unfortunately according to **?** this solution might not be valid in terms of the Airy functions and it should be further investigated why $\delta = \pi/4$ does not work in some cases. But at the time of writing this document no explanation has been found yet.

Finally since the Schrödinger equation is linear the user can choose from 3 types of scaling. 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.

4.5 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 $V(x) > E$.

Joint configuration Adjust the width of joints

- **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.
- **WKB_TRANSITION_FRACTION (E)** Same as the previous option just at the boundary between two WKB parts. It takes the width of the whole WKB part and multiplies it by this number to get the width of the joint.
- **ENABLE_WKB_JOINTS** If set to `true` joints will be added between WKB wave function parts. If set to `false` no joints will be added at this boundary. `false` is recommended when plotting probabilities.
- **ENABLE_AIRY_JOINTS** If set to `true` joints will be added at the boundary between Airy and WKB functions. `false` no joints will be added at these boundaries.

Complex Results Since **?** does not work with complex numbers you might get better results when setting all options to `false`.

- **COMPLEX_AIRY** If set to `true` the airy function will output complex numbers. This setting is recommended when using `plot_3d.gnuplot`. `false` only real values will be outputted. Recommended when using `plot.gnuplot`.
- **COMPLEX_EXP_WKB** If set to `true` the exponential WKB part will output complex numbers. This setting is recommended when using `plot_3d.gnuplot`. `false` only real values will be outputted. Recommended when using `plot.gnuplot`.

COMPLEX_OSZ_WKB If set to `true` the oscillating WKB part will output complex numbers. This setting is recommended when using `plot_3d.gnuplot`. `false` only real values will be outputted. Recommended when using `plot.gnuplot`.

5.1 Wave Function

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

```
    let wave_function = wave_function_builder::WaveFunction::new(
        &/*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 **??** 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 4.4.

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 **??**.
"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 4.4.

5.3 Plotting

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

```
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.

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

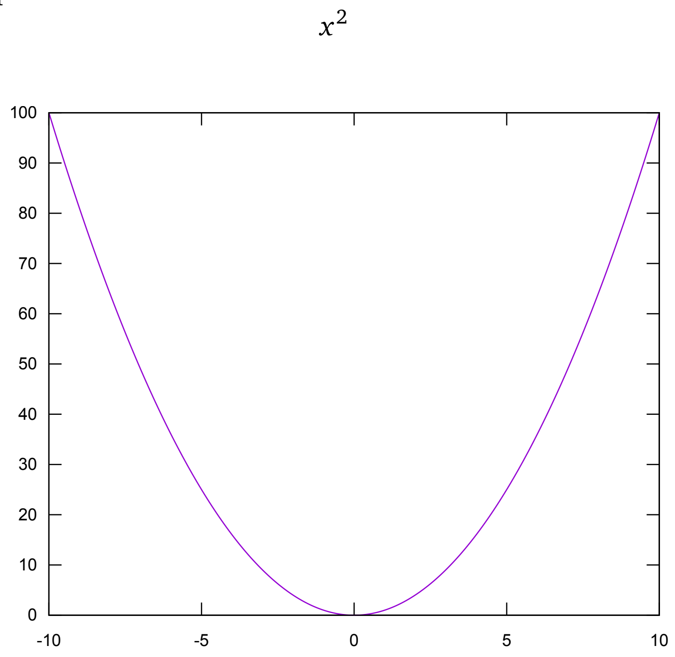
You can replace *data.txt* with another file name.

plot_wavefunction_parts

plot_probability

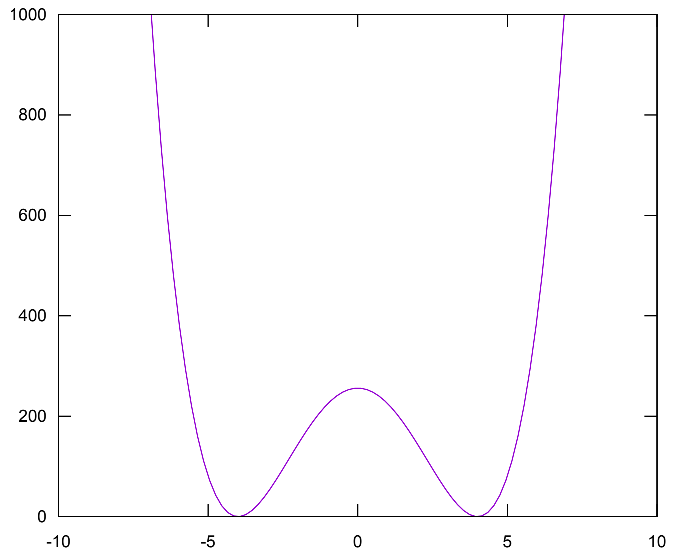
5.4 Potentials

square Normal square potential as used in **?**.



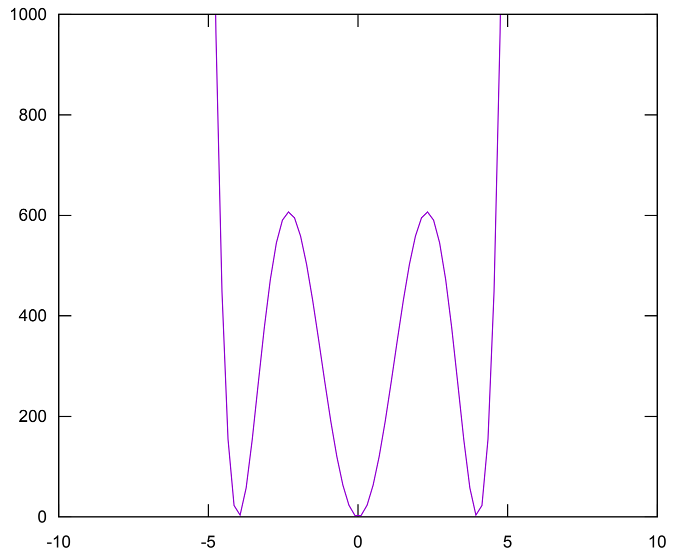
mexican_hat 4th degree polynomial that looks like a mexican hat, with 2 minima.

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



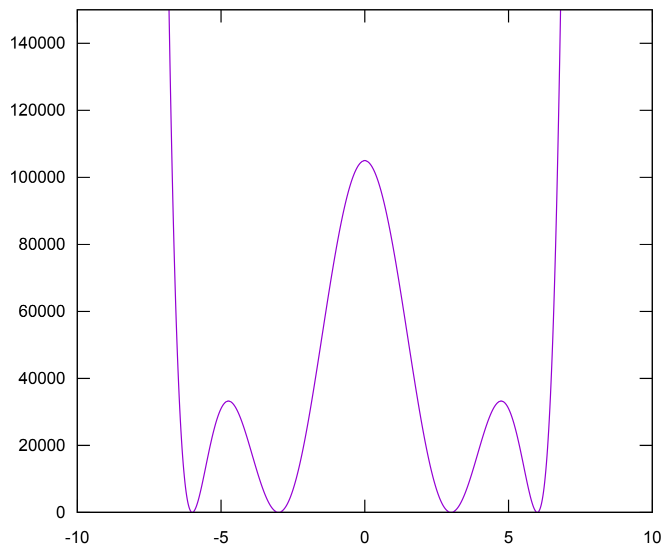
double_mexican_hat 6th degree polynomial that has 3 minima.

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



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

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 **(?)**

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

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

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

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

B Data Files

B.1 Energies

energy.txt

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

energies_exact.dat

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

```
38 108.89435777316754
39 111.72299572551829
40 114.55178992542766
41 117.38027162045414
42 120.2082845630391
43 123.0364537531827
44 125.86493544820918
45 128.69341714323565
46 131.52174259070352
47 134.35053679061292
48 137.17854972831506
49 140.0071876806658
50 142.83566937569228
```

```
38 108.89444430272833
39 111.72287142747452
40 114.5512985522207
41 117.3797256769669
42 120.20815280171308
43 123.03657992645928
44 125.86500705120547
45 128.69343417595167
46 131.52186130069785
47 134.35028842544403
48 137.17871555019022
49 140.00714267493643
50 142.83556979968262
```

C Source Code

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

src/main.rs

```rust
mod airy;
mod airy_wave_func;
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::collections::HashMap;
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 WKB_TRANSITION_FRACTION: f64 = 0.05;

const ENABLE_WKB_JOINTS: bool = false;
const ENABLE_AIRY_JOINTS: bool = true;

const COMPLEX_AIRY: bool = false;
const COMPLEX_EXP_WKB: bool = false;
const COMPLEX_OSZ_WKB: bool = false;

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

fn main() {
    // let wave_function = wave_function_builder::SuperPosition::new(
    //     &potentials::mexican_hat,
    //     1.0,
    //     &[
    //         (5, 1.0.into()),
    //         (12, 1.0.into()),
    //         (40, 1.0.into()),
    //     ],
    //     APPROX_INF,
    //     1.5,
    //     ScalingType::Renormalize(complex(1.0, 0.0)),
    // );

    let wave_function = wave_function_builder::WaveFunction::new(
        &potentials::square,
        1.0,
        5,
        APPROX_INF,
        1.5,
        ScalingType::Renormalize(1.0.into()),
    );

    let output_dir = Path::new("output");
    plot::plot_wavefunction(&wave_function, output_dir, "data.txt");
}
```

src/airy.rs

```rust
/* automatically generated by rust-bindgen 0.59.2 */

#[derive(PartialEq, Copy, Clone, Hash, Debug, Default)]
#[repr(C)]
pub struct __BindgenComplex<T> {
    pub re: T,
    pub im: T,
}
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)
        )
    );
}
#[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;
pub type GoUintptr = ::std::os::raw::c_ulong;
```

```rust
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)]
#[derive(Debug, Copy, Clone)]
```

```rust
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)
        )
    );
}
#[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

```
use crate::newtons_method::newtons_method;
use crate::newtons_method::*;
use crate::turning_points::*;
```

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

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

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

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

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

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

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

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

    pub fn with_phase_off(&self, phase_off: f64) -> AiryWaveFunction {
        AiryWaveFunction {
```

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

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

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

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

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

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

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

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

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

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

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

src/check.rs

```rust
use crate::*;

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

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

src/energy.rs

```
use crate::*;

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

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,
            view: self.view,
            energy,
        };
        let integral = integrate(
            evaluate_function_between(&integrand, self.view.0, self.view.1,
                INTEG_STEPS),
```

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

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

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

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

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

    // #[test]
    // fn square() {
    //     let pot = |x| x * x;
    //     assert!((nth_energy(0, 1.0, &pot, (-100.0, 100.0)) - 0.707107).abs() < 1e
        -7);
    // }
}
```

src/integrals.rs

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

#[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;
}

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>
```

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

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

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

    let mut rest = Y::default();

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

    return parallel + rest;
}

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

    (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);
    }

    #[tokio::test(flavor = "multi_thread")]
    async fn integral_of_square() {
        let square_func: Function<f64, Complex64> = Function::new(square);
        for i in 0..100 {
            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(&square_func, a, b, INTEG_STEPS
                                ),
                            TRAPEZE_PER_THREAD,
                        ),
                        complex(0.0, 0.0)
                    );
                    continue;
                }

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

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

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

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

    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

```rust
mod airy;
mod airy_wave_func;
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::collections::HashMap;
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 WKB_TRANSITION_FRACTION: f64 = 0.05;

const ENABLE_WKB_JOINTS: bool = false;
const ENABLE_AIRY_JOINTS: bool = true;

const COMPLEX_AIRY: bool = false;
const COMPLEX_EXP_WKB: bool = false;
const COMPLEX_OSZ_WKB: bool = false;

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

fn main() {
    // let wave_function = wave_function_builder::SuperPosition::new(
    //     &potentials::mexican_hat,
    //     1.0,
    //     &[
    //         (5, 1.0.into()),
    //         (12, 1.0.into()),
    //         (40, 1.0.into()),
    //     ],
    //     APPROX_INF,
    //     1.5,
    //     ScalingType::Renormalize(complex(1.0, 0.0)),
    // );

    let wave_function = wave_function_builder::WaveFunction::new(
        &potentials::square,
        1.0,
        5,
        APPROX_INF,
        1.5,
        ScalingType::Renormalize(1.0.into()),
    );

    let output_dir = Path::new("output");
    plot::plot_wavefunction(&wave_function, output_dir, "data.txt");
}
```

src/newtons_method.rs

```
use crate::integrals::*;
use crate::utils::cmp_f64;
use num::traits::FloatConst;
use num::{signum, Float};
use rayon::prelude::*;
use std::cmp::Ordering;
use std::fmt::Debug;
use std::ops::*;
use std::rc::Rc;
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)))
```

```rust
    }
}

impl Add for Vec2 {
    type Output = Vec2;

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

impl Sub for Vec2 {
    type Output = Vec2;

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

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

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

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

impl Add for CoVec2 {
    type Output = CoVec2;

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

impl Sub for CoVec2 {
    type Output = CoVec2;

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

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

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

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

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

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

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

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

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

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

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

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

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

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

        return zero;
    }

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

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

```rust
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,
    mut 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)
}

pub fn inverse<F, A, R>(f: &F) -> Box<dyn Fn(R) -> Vec<A>>
where
    F: Fn(A) -> R,
{
    todo!();
}

#[cfg(test)]
mod test {
    use super::*;
    use crate::integrals::*;
    use crate::utils::cmp_f64;
    use num::zero;

    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 {
```

```rust
            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()) {
```

```rust
                            assert!((*expected - *actual).abs() < 1e-10);
                        }
                    }
                }
            }
        }
    }

    #[test]
    fn newtons_method_find_next_test() {
        use std::f64::consts;
        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

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

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

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

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

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

    let airy_values = wave_function
        .get_airy_ranges()
        .iter()
        .map(|range| {
            evaluate_function_between(
                wave_function,
                f64::max(wave_function.get_view().0, range.0),
                f64::min(wave_function.get_view().1, range.1),
                NUMBER_OF_POINTS,
            )
        })
        .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
        .write_all(format!("plot␣\"{}\"␣u␣1:2␣t␣\"Re({})\"␣w␣l", output_file, title).
            as_bytes())
        .unwrap();

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

```
        .write_all(format!("plot␣\"{}\"␣u␣1:3␣t␣\"Im({})\"␣w␣l", output_file, title).as_bytes())
        .unwrap();
}

pub fn plot_wavefunction(wave_function: &WaveFunction, output_dir: &Path, output_file: &str) {
    plot_complex_function(
        wave_function,
        wave_function.get_view(),
        "Psi",
        output_dir,
        output_file,
    );
}

pub fn plot_superposition(wave_function: &SuperPosition, output_dir: &Path, output_file: &str) {
    plot_complex_function(
        wave_function,
        wave_function.get_view(),
        "Psi",
        output_dir,
        output_file,
    );
}

pub fn plot_probability(wave_function: &WaveFunction, 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();
}

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 {
```

```rust
                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;
        }
```

```rust

    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

```rust
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

```rust
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()
            - ((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 {
    // return TGroup{ts:vec![(-4.692, -4.255), (4.255, 4.692)]};
    let zeros = find_zeros(phase, view);
    return group_ts(&zeros, phase);
}

fn find_zeros(phase: &Phase, view: (f64, f64)) -> Vec<f64> {
    let phase_clone = phase.clone();
    let validity_func = Arc::new(move |x: f64| {
        1.0 / (2.0 * phase_clone.mass).sqrt() * derivative(&|t| (phase_clone.
            potential)(t), x).abs()
            - ((phase_clone.potential)(x) - phase_clone.energy).pow(2)
    });
    let mut zeros = NewtonsMethodFindNewZero::new(validity_func, ACCURACY, 1e4 as
        usize);

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

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

```rust
    });

    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;
}

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

src/utils.rs

```rust
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())
}

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)
    }
}
```

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

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

src/wave_function_builder.rs

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

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

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

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

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

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

impl WaveFunctionPart for Joint {
    fn range(&self) -> (f64, f64) {
        if self.delta > 0.0 {
            (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())
    }
}

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);
}

fn calc_phase_offset(phase: Arc<Phase>, (turn_left, turn_right): (f64, f64)) ->
    Option<f64> {
    // return Some(f64::consts::PI / 4.0);
    let critical_x = (turn_left + turn_right) / 2.0;
    if (phase.potential)(critical_x) > phase.energy {
        return None;
    }

    let int_left = integrate(
        evaluate_function_between(phase.as_ref(), critical_x, turn_left, INTEG_STEPS)
            ,
        TRAPEZE_PER_THREAD,
    );
    let int_right = -integrate(
        evaluate_function_between(phase.as_ref(), critical_x, turn_right, INTEG_STEPS
            ),
        TRAPEZE_PER_THREAD,
    );

    println!("left:␣{},␣right:␣{}", int_left, int_right);
    let phase_off = ((-int_left - int_right) / 2.0) % (2.0 * f64::consts::PI);

    println!(
        "phase_off␣/␣PI:␣{:.12}",
        phase_off / f64::consts::PI
    );

    Some(phase_off)
}

pub fn find_best_op_wave_func_part(
    phase: Arc<Phase>,
    previous: &dyn WaveFunctionPartWithOp,
    current: &dyn WaveFunctionPartWithOp,
) -> fn(Complex64) -> Complex64 {
    if !float_compare(current.range().0, previous.range().1, 1e-3) {
        println!("current:␣({},␣{})", current.range().0, current.range().1);
        println!("previous:␣({},␣{})", previous.range().0, previous.range().1);
    }
    assert!(float_compare(current.range().0, previous.range().1, 1e-3));
    let boundary = current.range().0;

    let deriv_prev = derivative(&|x| previous.eval(x), current.range().0);
    let val_prev = previous.eval(current.range().0);
    let deriv = derivative(&|x| current.eval(x), current.range().0);
    let val = current.eval(boundary);

    return if (phase.potential)(boundary) >= phase.energy {
        *previous.get_op()
    } else {
        let conj_deriv = conjugate(deriv);
        let conj_val = conjugate(val);
        let neg_conj_deriv = negative_conj(deriv);
        let neg_conj_val = negative_conj(val);
        let neg_deriv = negative(deriv);
        let neg_val = negative(val);

        let conj_mse = (conj_deriv - deriv_prev).norm_sqr() + (conj_val - val_prev).
            norm_sqr();
        let neg_conj_mse =
            (neg_conj_deriv - deriv_prev).norm_sqr() + (neg_conj_val - val_prev).
                norm_sqr();
        let neg_mse = (neg_deriv - deriv_prev).norm_sqr() + (neg_val - val_prev).
            norm_sqr();
        let id_mse = (deriv - deriv_prev).norm_sqr() + (val - val_prev).norm_sqr();

        if conj_mse <= neg_conj_mse && conj_mse <= neg_mse && conj_mse <= id_mse {
            println!(
                "conjugate␣mse,␣conj:␣{},␣neg␣conj:␣{},␣neg:␣{},␣id:␣{}",
                conj_mse, neg_conj_mse, neg_mse, id_mse
            );
            conjugate
        } else if neg_conj_mse <= conj_mse && neg_conj_mse <= neg_mse && neg_conj_mse
             <= id_mse {
            println!(
                "negative␣conj␣mse,␣conj:␣{},␣neg␣conj:␣{},␣neg:␣{},␣id:␣{}",
                conj_mse, neg_conj_mse, neg_mse, id_mse
            );
            negative_conj
        } else if neg_mse <= conj_mse && neg_mse <= neg_conj_mse && neg_mse <= id_mse
             {
            println!(
                "negative␣mse,␣conj:␣{},␣neg␣conj:␣{},␣neg:␣{},␣id:␣{}",
                conj_mse, neg_conj_mse, neg_mse, id_mse
            );
            negative
        } else {
            println!(
                "identity␣mse,␣conj:␣{},␣neg␣conj:␣{},␣neg:␣{},␣id:␣{}",
                conj_mse, neg_conj_mse, neg_mse, id_mse
            );
            identity
        }
    };
}

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

    pub fn new<F: Fn(f64) -> f64 + Sync + Send>(
        potential: &'static F,
        mass: f64,
        n_energy: usize,
        approx_inf: (f64, f64),
        view_factor: f64,
        scaling: ScalingType,
    ) -> WaveFunction {
        let energy = energy::nth_energy(n_energy, mass, &potential, approx_inf);

        let lower_bound = newtons_method::newtons_method_max_iters(
            &|x| potential(x) - energy,
            approx_inf.0,
            1e-7,
            100000,
        );
        let upper_bound = newtons_method::newtons_method_max_iters(
```

```rust
            &|x| potential(x) - energy,
            approx_inf.1,
            1e-7,
            100000,
        );

        let view = if lower_bound.is_some() && upper_bound.is_some() {
            (
                lower_bound.unwrap() * view_factor,
                upper_bound.unwrap() * view_factor,
            )
        } else {
            println!("Failed␣to␣determine␣view␣automatically,␣using␣APPROX_INF␣as␣view");
            approx_inf.clone()
        };

        let phase = Arc::new(Phase::new(energy, mass, potential));

        let (airy_wave_funcs, boundaries) = AiryWaveFunction::new(phase.clone(), (view.0, view.1));
        let (parts, airy_ranges, wkb_ranges): (
            Vec<Arc<dyn WaveFunctionPart>>,
            Vec<(f64, f64)>,
            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,
                calc_phase_offset(phase.clone(), approx_inf).unwrap_or(f64::consts::PI / 4.0),
            );
            let wkb2 = WkbWaveFunction::new(
                phase.clone(),
                1.0.into(),
                INTEG_STEPS,
                approx_inf.1,
                calc_phase_offset(phase.clone(), approx_inf).unwrap_or(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 op = find_best_op_wave_func_part(phase.clone(), wkb1.as_ref(), wkb2.
                as_ref());

            let wkb1_range = wkb1.range();
            let wkb2 = wkb2.with_op(op);
            let delta = (view.1 - view.0) * WKB_TRANSITION_FRACTION;
            (
                if ENABLE_WKB_JOINTS {
                    vec![
                        Arc::new(Joint {
                            left: Arc::from(wkb1.as_func()),
                            right: Arc::from(wkb2.as_func()),
                            cut: (view.0 + view.1) / 2.0 - delta / 2.0,
                            delta: delta,
                        }),
                        Arc::from(wkb1.as_wave_function_part()),
                        Arc::from(wkb2.as_wave_function_part()),
                    ]
                } else {
                    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)) {
                        (
```

```rust
                            WkbWaveFunction::new(
                                phase.clone(),
                                1.0.into(),
                                INTEG_STEPS,
                                *boundary,
                                calc_phase_offset(phase.clone(), (*previous, *
                                    boundary))
                                    .unwrap_or(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)))
                .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();

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

            for i in 0..(approx_parts.len() - 1) {
                let part1 = &approx_parts[i];
                let part2 = &approx_parts[i + 1];
                let p2_with_op = part2.with_op(find_best_op_wave_func_part(
                    phase.clone(),
                    part1.as_ref(),
                    part2.as_ref(),
                ));
                approx_parts_with_op.push(Arc::from(p2_with_op));
            }
            let mut approx_parts_with_joints: Vec<Arc<dyn WaveFunctionPart>> = vec!
                [];

            if ENABLE_WKB_JOINTS {
                for (prev, curr) in approx_parts_with_op
                    .iter()
                    .zip(approx_parts_with_op.iter().skip(1))
                {
                    assert!(float_compare(prev.range().1, curr.range().0, 1e-4));

                    let distance = (f64::min(prev.range().1, view.1)
                        - f64::max(prev.range().0, view.0))
                        + (f64::min(curr.range().1, view.1) - f64::max(curr.range()
                            .0, view.0));
                    let delta = distance * WKB_TRANSITION_FRACTION;
                    let joint = Joint {
                        left: Arc::from(prev.as_func()),
                        right: Arc::from(curr.as_func()),
                        cut: f64::min(prev.range().1, view.1) - delta / 2.0,
                        delta,
                    };

                    println!("Joint␣in␣range:␣{:#?},␣delta:␣{}", joint.range(), delta
                        );

                    approx_parts_with_joints.push(Arc::new(joint));
                }
            }

            approx_parts_with_joints = vec![
                approx_parts_with_joints,
                approx_parts_with_op
                    .iter()
                    .map(|p| Arc::from(p.as_wave_function_part()))
                    .collect(),
            ]
            .concat();

            (approx_parts_with_joints, 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", *e);
                return wave;
            })
            .collect();

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

```rust

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 {
    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,
```

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

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

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

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

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

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

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

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

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

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

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

```rust
                            / self.phase.sqrt_momentum(x)
                    } else {
                        1.0.into()
                    }
            } else {
                (self.c * 0.5 * (-integral.abs()).exp())
                    * if COMPLEX_EXP_WKB {
                        complex((self.phase_off).cos(), (self.phase_off).sin())
                            / self.phase.sqrt_momentum(x)
                    } else {
                        1.0.into()
                    }
            }
        } else {
            if x < self.turning_point {
                self.c
                    * complex(
                        (-integral + self.phase_off).cos(),
                        if COMPLEX_OSZ_WKB {
                            (-integral + self.phase_off).sin()
                        } else {
                            0.0
                        },
                    )
                    / self.phase.sqrt_momentum(x)
            } else {
                self.c
                    * complex(
                        (integral + self.phase_off).cos(),
                        if COMPLEX_OSZ_WKB {
                            (integral + self.phase_off).sin()
                        } else {
                            0.0
                        },
                    )
                    / self.phase.sqrt_momentum(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
```

```go

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

```rust
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

```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"

[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]}] == 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/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.

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