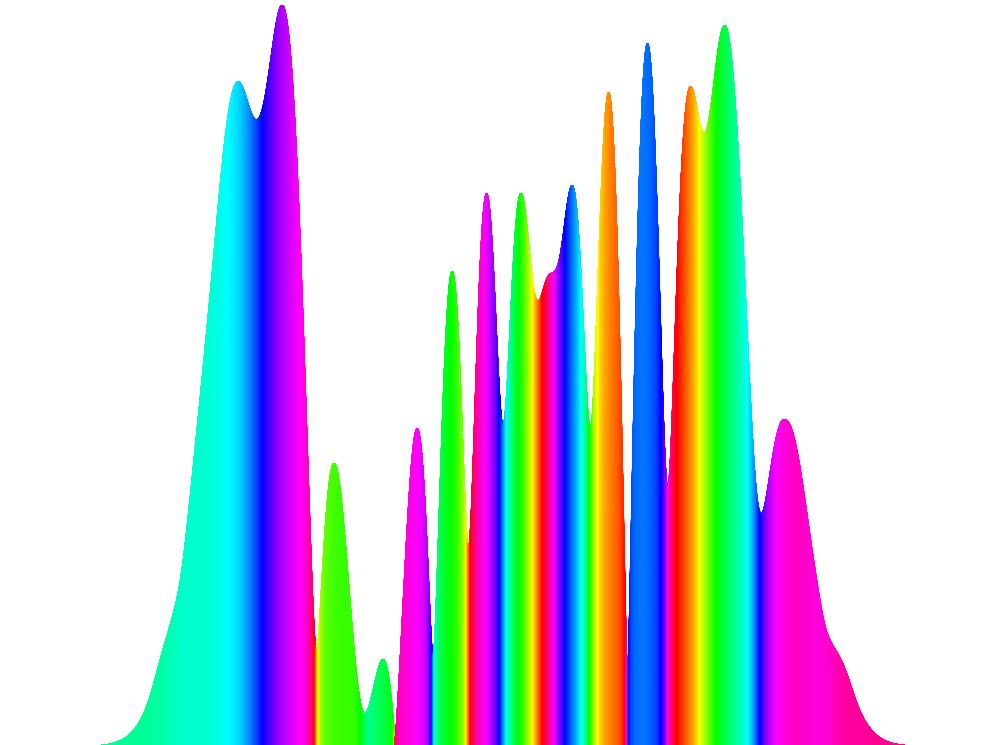


Approximating the Time Independent Schrödinger Equation

Gian Laager
November 26, 2022



Matura thesis
Kantonsschule Glarus

Supervisor: Linus Romer
Referent: Dr. Elena Borisova

Contents

Vorwort	iv
1. Introduction	1
1.1. Goals	1
2. Preliminary	2
2.1. Schrödinger Equation	2
2.2. Rust	2
2.3. Interpretation of Quantum Mechanics	3
2.4. Complex Numbers	3
2.5. GNU Plot	3
2.5.1. Usage	4
2.6. Reading Complex Plots	4
2.7. Planck Units	7
2.8. Benchmarking	7
2.8.1. perf	7
2.8.2. Rust Benchmarks	10
3. Methods	11
3.1. Program Architecture	11
3.2. Newtons Method	11
3.3. Regula Falsi with Bisection	13
3.4. Derivatives	13
3.5. Integration	14
3.6. Transition Regions	18
3.6.1. Implementation in Rust	20
4. Calculation	22
4.1. Energy Levels	22
4.1.1. Accuracy	23
4.2. Approximation Scheme	25
4.2.1. Validity	27
4.2.2. Implementation	28
4.3. Turning Points	29
4.4. Wave Function Parts	32
4.4.1. ApproxPart	32
4.4.2. PureWkb	33

4.5. Wave Function	33
4.5.1. Super Position	34
5. Program Manual	35
5.1. Installation of schroedinger_approx	35
5.1.1. Linux	35
5.1.2. macOS	36
5.1.3. Windows	36
5.2. Usage	37
5.3. WaveFunction	38
5.4. SuperPosition	39
5.5. Plotting	39
5.5.1. WaveFunction	39
5.5.2. SuperPosition	40
5.6. Potentials	41
5.6.1. Custom Potentials	43
6. Results	44
6.1. Wave Functions	44
6.1.1. Hall Example	44
6.1.2. Phase Shift	45
6.1.3. 0th Energy	46
6.2. Mexican Hat Potential	48
6.2.1. Super Position	51
6.2.2. Relationship between Energy and Mass	54
6.3. Performance	54
6.4. Accuracy	55
7. Conclusion	59
A. Detailed Calculations and Tests	60
A.1. Test Machine Specs	60
A.1.1. Machine 1	60
A.1.2. Machine 2	61
A.2. Benchmarks	62
A.2.1. Rust Benchmarks	62
A.2.2. Perf	65
A.3. Additional Plots	67
A.3.1. Mexican Hat	67
A.4. Proofs	68
A.4.1. Smoothness of Transitionfunction	68
A.5. Validity of 0 Wave Function	69
A.6. Branch Elimination	69

B. Data Files	71
B.1. Energies	71
C. Source Code	73

Vorwort

Ich habe mich entschieden, eine kleine Zusammenfassung auf deutsch zu schreiben, damit zumindest jeder die Grundlagen meiner Arbeit versteht.

Zu Beginn des 20. Jahrhunderts gab es einen Umschwung in der Physik. Die Quantenmechanik wurde entdeckt. Diese neue Theorie kann nicht mehr präzise Voraussagen machen, wie es zuvor der Fall war. Man kann nur noch sagen, mit welcher Wahrscheinlichkeit etwas passiert. Dies hat bizarre Folgen, wie zum Beispiel, dass ein Partikel an zwei Orten gleichzeitig sein kann.

Vielleicht haben Sie schon einmal von Schrödingers Katze gehört. Dies war ein Gedankenexperiment von Schrödinger um aufzuzeigen, wie absurd seine Theorie wirklich ist und dass sie nicht stimmen könne.

Stell dir vor, du schliesst eine 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 Quantenmechanik folgt und deshalb gleichzeitig bereits zerfallen ist und nicht zerfallen ist. Die einzige logische Schlussfolgerung ist deshalb, dass *die Katze gleichzeitig Tod und am Leben ist* (Schrödinger, 1935).

In der Realität funktioniert es wahrscheinlich jedoch nicht so. Das heisst, dass das Universum “entscheidet”, ob die Katze gestorben ist oder nicht. Jedoch weiss man bis heute nicht, wann das Universum “entscheidet”.

Damit die Katze gleichzeitig tot und lebendig sein kann, brauchen wir die Wellenfunktion. Sie beschreibt alles, was in unserem Universum gerade passiert und “speichert” die Wahrscheinlichkeit, dass etwas passiert. Wie zum Beispiel, dass die Katze gestorben ist.

In meiner Maturaarbeit habe ich ein Programm geschrieben, das genau diese Wellenfunktion in einem sehr vereinfachten Universum ausrechnet, weil ich schon lange mal wissen wollte, wie genau dieses bizarre Objekt aussieht. Auf der Titelseite 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 is 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 that the wave function can actually be plotted and one can look at it. Usually in our universe the wave function has more then 3 dimensions, meaning we can't really imagine nor visualize it intuitively.

1.1. Goals

The goal of this Matura thesis is to write a program, `schroeding-approx` that calculates solutions to the time independent Schrödinger equation in 1 dimension for a large variety of potentials. For the calculation it is assumed that the wave function, $\Psi(x)$ will converge to 0 as x goes to $\pm\infty$. The program should be reasonably fast, meaning that for simple potentials and low energies it should be done in under 1 minute. The architecture should be able to support improvements.

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

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

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

2. Preliminary

2.1. Schrödinger Equation

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

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

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.

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

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.

To learn more about Rust, it has excellent documentation on <https://doc.rust-lang.org/book/>.

2.3. Interpretation of Quantum Mechanics

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

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

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

2.4. Complex Numbers

In quantum mechanics it’s customary to work with complex numbers. Complex numbers are an extension to the real numbers, since Rust will do most of the calculations, you don’t have to master complex numbers.

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

i is the imaginary unit, z is the general form of a complex number where $\{a, b\} \in \mathbb{R}$, \bar{z} is the complex conjugate and $\|z\|^2$ is the norm square of z . The last equation is the Euler’s formula, it rotates a number in the complex plane by θ radians. The angle θ is also called the *argument* of a complex number and corresponds to the *color* mentioned in section 2.6.

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

2.5. GNU Plot

Gnuplot is a cross platform plotting program that has all the features needed to plot the wave functions. schroedinger-approx will output a file `data.txt` and the corresponding GNU Plot scripts to plot the wave function.

2.5.1. Usage

To plot a wave function you can type `gnuplot` in a terminal in the output directory. This will open a prompt. You can then run one of the following commands.

`call "plot.gnuplot"` This will plot the real part of the wave function.

`call "plot_im.gnuplot"` This will plot the imaginary part of the wave function.

`call "plot_3d.gnuplot"` This will plot the wave function as a 3d graph.

`call "plot_color.gnuplot"` This will plot the wave function as a color plot according to section 2.6.

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

2.6. Reading Complex Plots

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

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

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

For example on the next page the plot from the title page is plotted both as a 3D plot and color plot for comparison.

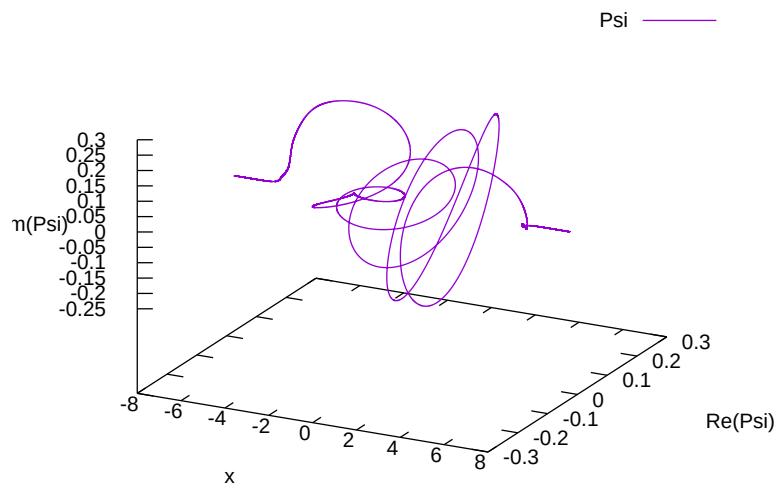


Figure 2.1.: 3D plot of super position of $V(x) = x^2$ and energies 9, 12 and 15. One can't really see anything and it's just a mess, this would be better if one could interact with the plot on the computer.

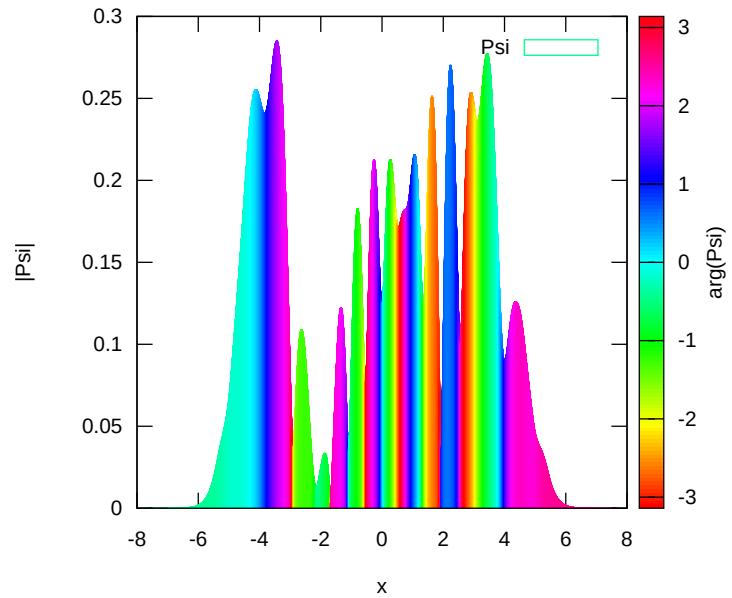


Figure 2.2.: Color plot of super position of $V(x) = x^2$ and energies 9, 12 and 15. Although it takes time to get used to, this plot is much clearer compared to figure 2.1.

As shown in figure 2.1, the 3D plot is basically unusable since there is no depth. This would usually be fixed with lighting but it would be very difficult to apply lighting on a line such that one could actually see the depth. In the color plot every thing seems to be clear. Except that the values of the phase can't be read precisely.

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

Figure 2.3 should help to read the color plots.

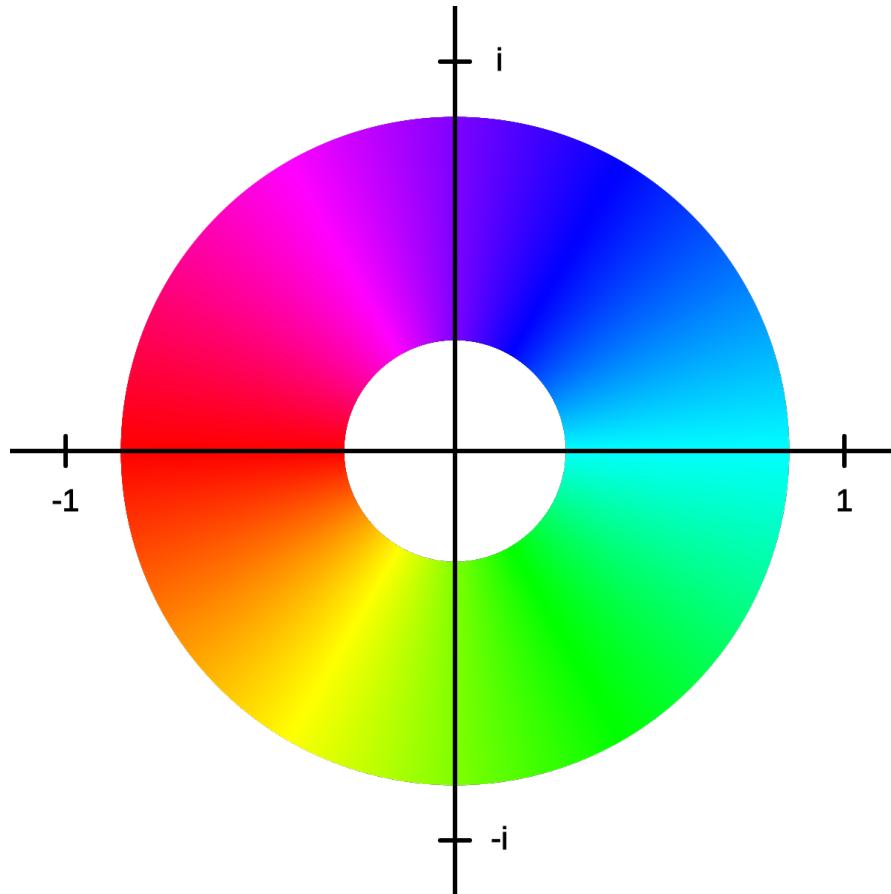


Figure 2.3.: Illustration of the mapping of an angle to color to represent complex numbers.
Every argument of a complex number is assigned to a color.

For example $1 + 0i$ would have the color cyan and $0 + 1i$ would be purple. This graph is a handy tool to read the color plots that will be used later since it's easier to associate the angle with the color if the color is actually at that angle rather than a number in radians.

2.7. Planck Units

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

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

This means that the constants will usually cancel out.

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

$$l_{\text{Planck}} = l_{\text{SI}} \sqrt{\frac{G\hbar}{c^3}} \quad 1 \text{ m}_{\text{Planck}} \approx 1.616255(18) \cdot 10^{-35} \text{ m} \quad (\text{CODATA, 2022a})$$
$$m_{\text{Planck}} = m_{\text{SI}} \sqrt{\frac{c\hbar}{G}} \quad 1 \text{ kg}_{\text{Planck}} \approx 2.176434(24) \cdot 10^{-8} \text{ kg} \quad (\text{CODATA, 2022b})$$
$$t_{\text{Planck}} = t_{\text{SI}} \sqrt{\frac{G\hbar}{c^5}} \quad 1 \text{ s}_{\text{Planck}} \approx 5.391247(60) \cdot 10^{-44} \text{ s} \quad (\text{CODATA, 2022c})$$

(Gaarder Haug, 2016, Table 1)

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

2.8. Benchmarking

Benchmarking is the process where one tests the performance of software. In the case of *schroeding_approx* the goal would be to test the individual parts and find out where improvements are possible. And in a later step if the supposed improvements actually improved the performance. For this the two tools described below will be used.

2.8.1. perf

perf is a collection of performance analysis tools for Linux (Linux Kernel Organization, 2022). It will later be used to measure the performance of *schroedinger_approx*. In its manual page (Linux Kernel Organization, 2022) it's described as:

Performance counters for Linux are a new kernel-based subsystem that provide a framework for all things performance analysis. It covers hardware level (CPU/PMU, Performance Monitoring Unit) features and software features (software counters, tracepoints) as well.

In particular `perf stat` and `perf record` will be used.

The outputs produced by `perf stat` are fairly technical can measure performance on the

CPU level. Most of the metrics produced *will not* be used. Below there's an example output of a benchmark of *schroedinger_approx*.

```

1      727,840.51 msec task-clock          # 15.452 CPUs utilized
2      404,803    context-switches       # 556.170 /sec
3      5,210     cpu-migrations        # 7.158 /sec
4      30,529    page-faults           # 41.945 /sec
5      2,482,025,824,794   cycles        # 3.410 GHz
6      1,855,377,839    stalled-cycles-frontend # 0.07% frontend
7          cycles idle                # 0.14% backend cycles
8      3,495,430,962    stalled-cycles-backend
9          idle
10     4,704,001,143,909   instructions   # 1.90 insn per cycle
11
12
13     566,687,255,541    branches       # 778.587 M/sec
14     252,663,255    branch-misses  # 0.04% of all branches
15
16     47.103191946 seconds time elapsed
17
18     722.481191000 seconds user
19     5.147335000 seconds sys

```

CPUs utilized This states how many “percent” of the *CPU* were actually used, important to note is that “CPU” in this case means hardware thread. In the case of the test machine, it has 8 cores with 2 threads each, therefor in theory this number could be as high as 16.0. The goal would be to use as much of the CPU as possible but one has to be cautious because other processes besides *schroedinger_approx* are also running and use some part of the CPU.

context-switches This factor is handled by the Linux kernel. The kernel *switches* between processes that can run on a thread of the CPU. In this case most of the context switches were happening inside *schroedinger_approx* itself because behind the scene it uses a thread pool. In general it is an indicator of how efficiently the kernel can handle the multi threading in a program.

page-faults This is an “error” that happens inside the CPU when a process is trying to access memory that is not yet loaded into the cache of the CPU. When the number of page faults per second is high, the process’s memory layout is not good and should be optimized. These kind of optimizations are very hard to do.

branch-misses Modern CPUs don’t actually execute the assembly directly but they perform *speculative execution*. This means the CPU “guesses” ahead if for example an if-statement will be true and then already do the calculations before the instruction pointer actually reached the branch. Even though the CPU is pretty good at “guessing” if it’s wrong it has to throw away all those calculation, this is called a *branch miss*. The code can be made faster if the percentage of branch misses is low (< 0.05%), this can be done by writing “predictable” code which requires intensive testing.

time elapsed This is the time the program was running. This is similar to the `time` command.

2.8.2. Rust Benchmarks

The nightly version of Rust contains benchmarks that can be run with the command `cargo bench`. The benchmarks are the functions marked with the `#[bench]` macro. All those functions take a `test::Bencher` as an argument. This struct will measure the time it takes the code to complete. It will also run the fragments multiple times. Finally it will print the time measurements for each benchmark to the terminal.

While `perf` captures the big picture, it's possible to *zoom* in to the individual components of the system with Rust benchmarks.

3. Methods

This chapter is about the implementation details and which algorithms and mathematics are used behind the scenes. Some smaller details of the code will not be discussed.

3.1. Program Architecture

The program has multiple interfaces, or traits as they are called in Rust, that give the program some abstraction. In Appendix C is a diagram of the architecture. Since the current version of Rust does not support manual implementations of `std::ops::Fn` a custom trait will be defined for functions `Func<A, R>` where `A` is the type of the argument and `R` is the return type. Later this trait will be used to implement functions for integration, evaluation and more utilities.

The `WaveFunction` struct 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` which will be discussed in detail in section 4.4. With the `range` function it can be checked when a part is valid that is to say when it can be evaluated without a large error.

3.2. Newtons Method

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

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

The sequence needs a first guess of where the root could be which will be the variable a , then the sequence is defined as

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

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



Figure 3.1.: Illustration of Newtons method, $f(x) = (x - 1)(x - 2)(x + 1)$. The blue line indicates the initial guess which in this case is 2.5 the black line ($g(x)$) is a tangent to $f(x)$ at $(\text{guess}, f(\text{guess}))$ the next guess will be where the tangent intersects the x-axis (solution of $g(x) = 0$). This will converge quicker than other methods such as Regula falsi

Code Snippet 3.1: Implementation of Newtons method. The function takes a closure f , the initial guess guess and a stop condition precision . The function will return if $|f(x_n) / f'(x_n)|$ is less than precision . If the derivative at any point becomes 0 the function will panic. Because of this the function `newtons_method_max_iters` provides an alternate implementation that will not panic and *always* return an `Option`.

```

1 pub fn newtons_method<F>(f: &F, mut guess: f64, precision: f64) -> f64
2 where
3     F: Fn(f64) -> f64,
4 {
5     loop {
6         let deriv = derivative(f, guess);
7
8         if deriv == 0.0 {
9             panic!("Devision_by_zero");
10        }
11
12         let step = f(guess) / deriv;
13         if step.abs() < precision {
14             return guess;

```

```

15     } else {
16         guess -= step;
17     }
18 }
19 }
```

An extension to the function in snippet 3.1 is implemented in the struct `NewtonsMethodFindNewZero` which can be used to find multiple roots.

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

3.3. Regula Falsi with Bisection

Newton's method fails if the first guess is at a maximum, since the step would go to infinity. For these cases a bisection method will be applied until the sign of the function changes. This needs to be done because Regula Falsi requires two guesses.

The algorithm itself is quite simple. To start define the parameters

$$f(x) : \mathbb{R} \rightarrow \mathbb{R} \quad (3.1)$$

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

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

Then draw a line between the two points $(a, f(a))$ and $(b, f(b))$. Then b becomes the x-value where the line intersects the x-axis, when this process is applied again with the new b the resulting value will become the new a . This process can be repeated until a threshold is crossed for the accuracy and the result will be the last intersection of the line with the x-axis.

3.4. Derivatives

Derivatives can be calculated numerically as in the C++ library Boost (John Maddock, 2022). The author implemented an analytical system for calculating derivatives in Go. This project used an interface with two methods `function` and `derivative`. Afterwards functions for operations like multiplication, addition, etc. were added. In these operations the `derivative` was built based on the operation. This process of writing functions is tedious and the performance is worse than in the numerical implementation in rust (snippet 3.2).

Code Snippet 3.2: Rewrite of the C++ library Boost's implementation (John Maddock, 2022) of numerical differentiation for Rust. `f64::epsilon().sqrt()` is approximately $1.4901161 \cdot 10^{-8}$. `f64::epsilon()` is the smallest double precision floating point number ϵ where $1 + \epsilon \neq 1$. This value has been chosen for dx because it is precise enough.

```

1 pub fn derivative<F, R>(func: &F, x: f64) -> R
2 where
3     F: Fn(f64) -> R + ?Sized,
4     R: Sub<R, Output = R> + Div<f64, Output = R> + Mul<f64, Output = R> + Add<R,
5         Output = R>,
6 {
7     let dx = f64::epsilon().sqrt();
8     let dx1 = dx;
9     let dx2 = dx1 * 2.0;
10    let dx3 = dx1 * 3.0;
11
12    let m1 = (func(x + dx1) - func(x - dx1)) / 2.0;
13    let m2 = (func(x + dx2) - func(x - dx2)) / 4.0;
14    let m3 = (func(x + dx3) - func(x - dx3)) / 6.0;
15
16    let fifteen_m1 = m1 * 15.0;
17    let six_m2 = m2 * 6.0;
18    let ten_dx1 = dx1 * 10.0;
19
20    return ((fifteen_m1 - six_m2) + m3) / ten_dx1;
21 }
```

3.5. Integration

The same principles apply to integrals as to derivatives, it wouldn't 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 into many smaller integrals that can be computed easily. Instead it would have 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, since integrals calculate areas under curves a trapeze is more efficient and accurate than the rectangle that results from the definition.

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

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

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

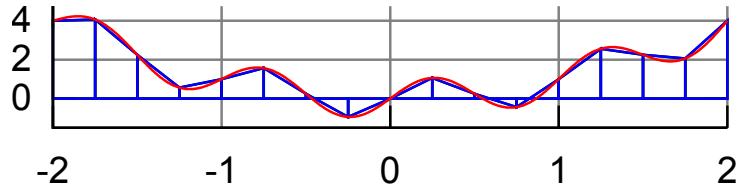


Figure 3.2.: Illustration of integration with trapeze from Wikipedia (2022).

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()` (rayon rs, 2022). This might not work in all cases because of memory safety.

```

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

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

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

```

1 pub fn evaluate_function_between<X, Y>(f: &dyn Func<X, Y>, a: X, b: X, n: usize) ->
2     Vec<Point<X, Y>>
3 where
4     X: Copy
5         + Send
6         + Sync
7         + std::cmp::PartialEq
8         + From<f64>
9         + std::ops::Add<Output = X>
10        + std::ops::Sub<Output = X>
11        + std::ops::Mul<Output = X>
12        + std::ops::Div<Output = X>,
13     Y: Send + Sync,
14 {
15     if a == b {
16         return vec![];
17     }
18     (0..n)
```

```
19     .into_par_iter()
20     .map(|i| {
21         index_to_range(
22             X::from(i as f64),
23             X::from(0.0_f64),
24             X::from((n - 1) as f64),
25             a,
26             b,
27         )
28     })
29     .map(|x: X| Point { x, y: f.eval(x) })
30     .collect()
31 }
```

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

```

1 pub fn integrate<
2     X: Sync + std::ops::Add<Output = X> + std::ops::Sub<Output = X> + Copy,
3     Y: Default
4         + Sync
5             + std::ops::AddAssign
6                 + std::ops::Div<f64, Output = Y>
7                     + std::ops::Mul<Output = Y>
8                         + std::ops::Add<Output = Y>
9                             + Send
10                                + std::iter::Sum<Y>
11                                    + Copy
12                                        + From<X>,
13 >(
14     points: Vec<Point<X, Y>>,
15     batch_size: usize,
16 ) -> Y {
17     if points.len() < 2 {
18         return Y::default();
19     }
20
21     let batches: Vec<&[Point<X, Y>]> = points.chunks(batch_size).collect();
22
23     let parallel: Y = batches
24         .par_iter()
25         .map(|batch| {
26             let mut sum = Y::default();
27             for i in 0..(batch.len() - 1) {
28                 sum += trapezoidal_approx(&batch[i], &batch[i + 1]);
29             }
30             return sum;
31         })
32         .sum();
33
34     let mut rest = Y::default();
35
36     for i in 0..batches.len() - 1 {
37         rest += trapezoidal_approx(&batches[i][batches[i].len() - 1], &batches[i + 1][0]);
38     }
39
40     return parallel + rest;
41 }
```

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, these 1000 values are called a *batch*. This parameter was chosen for the author's computer and 1000 might not be optimal for all CPUs. After all batches have been calculated the boundaries between batches also have to be considered therefor they are added in the end with `rest`.

3.6. Transition Regions

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

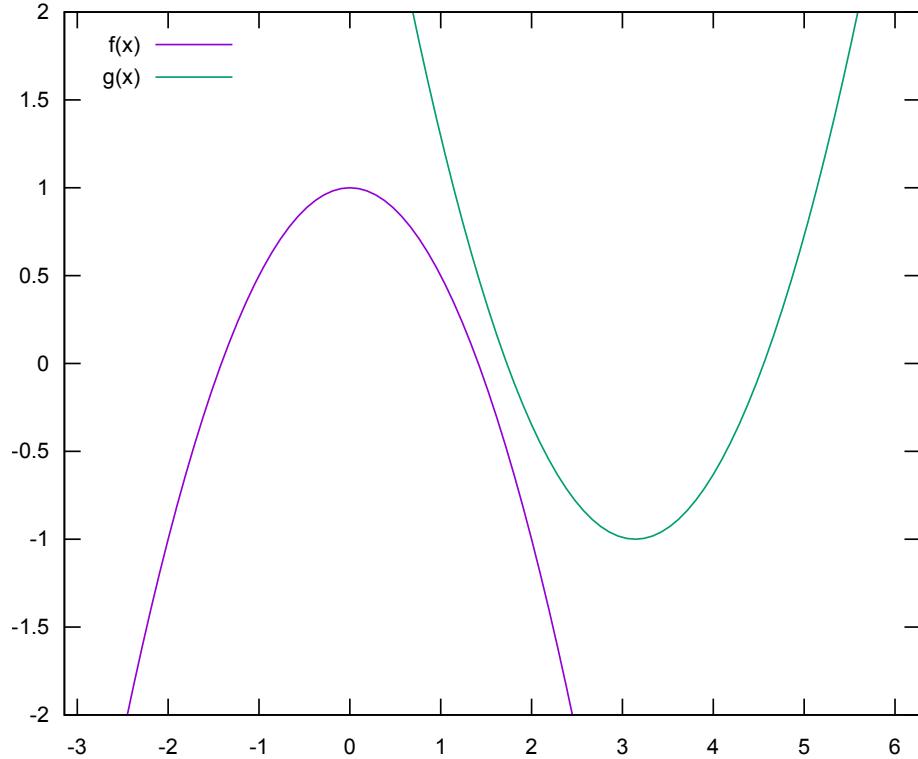


Figure 3.3.: Two example functions that should be joined at $x = \frac{\pi}{2}$. The functions are two Taylor series of cosine, they have been chosen as an example because they don't overlap and get close together at $x = \frac{\pi}{2}$.

Lets consider an example, in figure 3.3 we can see two Taylor series of cosine. Now the two functions have to be joined at $x = \pi/2$ such that its a mathematically smooth transition.

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

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

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 results in figure 3.4 which is obviously not smooth.

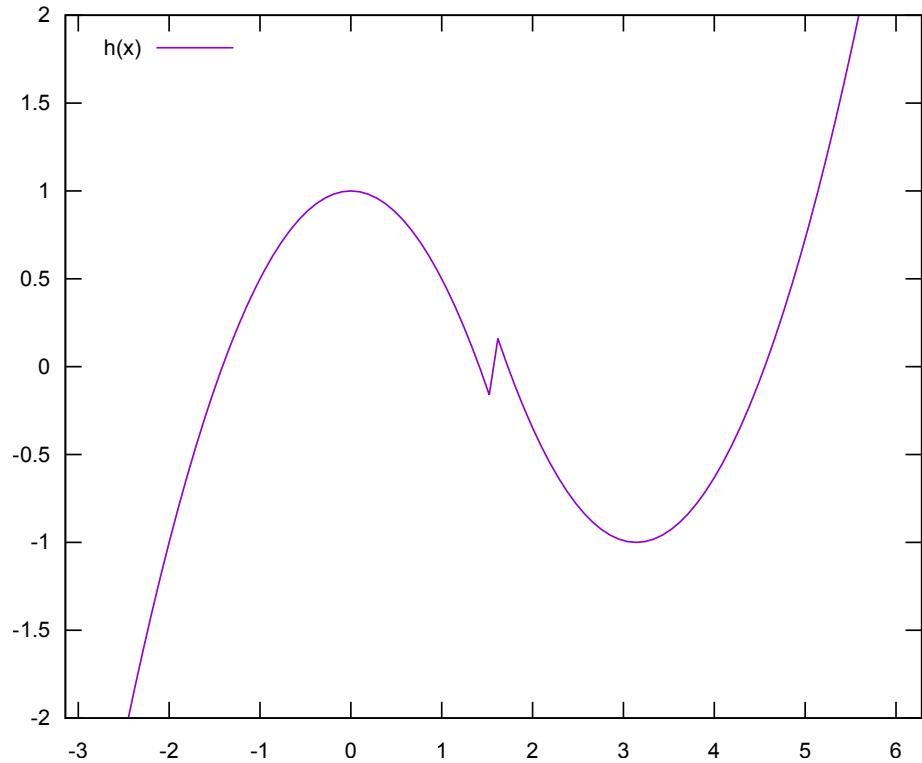


Figure 3.4.: Plot of $h(x)$ with step joint. As shown the transition is not smooth and there's a discontinuity at $x = \frac{\pi}{2}$.

Using the formula from (Hall, 2013, p. 325, section 15.6.4)

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

a much better result can be obtained

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

which is mathematically smooth as can be see in figure 3.5 (proof in Appendix A.4.1).



Figure 3.5.: Plot of $h(x)$ with Hall joint. The function is smooth and has a slight *bump* at $x = \frac{\pi}{2}$

3.6.1. Implementation in Rust

In the program the struct `Joint` implements the formula from Hall (2013). As in the example the two functions $f(x)$ and $g(x)$, which will be renamed to `left` and `right`, have to be joined at α over a range of δ . The variables α and δ are from now on called `cut` and `delta`.

```

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

```

```
14     }
15 }
```

In the proof it has been assumed that $f(x)$ and $g(x)$ are continuous of first order in the interval $(\alpha, \alpha + \delta)$. In the code this assumption will not be checked, 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 condition described by (Hall, 2013, eq. 15.31).

$$n \in \mathbb{N}_0 \quad (4.1)$$

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

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

It can be interpreted such that the oscillating part of the wave function has to complete all half oscillations. Hall (2013) also states that the equations from section 4.2 have to agree, up to a multiplication by a constant which will be the case iff equation 4.3 is satisfied.

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

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

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

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 outside 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, first equation 4.5 is evaluated for a number of energies and then checked for discontinuities.

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

```

7   let mut i = 0;
8
9   loop {
10     let vals = evaluate_function_between(
11       &sommerfeld_cond,
12       energy,
13       energy + ENERGY_STEP,
14       CHECKS_PER_ENERGY_STEP,
15     );
16     let mut int_solutions = vals
17       .iter()
18       .zip(vals.iter().skip(1))
19       .collect::<Vec<(&Point<f64, f64>, &Point<f64, f64>)>>>()
20       .par_iter()
21       .filter(|(p1, p2)| (p1.y - p2.y).abs() > 0.5 || p1.y.signum() != p2.y.
22         signum())
23       .map(|ps| ps.1)
24       .collect::<Vec<&Point<f64, f64>>>();
25     int_solutions.sort_by(|p1, p2| cmp_f64(&p1.x, &p2.x));
26     if i + int_solutions.len() > n {
27       return int_solutions[n - i].x;
28     }
29     energy += ENERGY_STEP - (ENERGY_STEP / (CHECKS_PER_ENERGY_STEP as f64 + 1.0))
30     ;
31     i += int_solutions.len();
32   }
33 }
```

First the interval $(0.0, \text{ENERGY_STEP})$ is checked, if there are not enough zeros the next interval is checked. This is repeated until n zeros have been found. It's also possible that equation 4.5 is negative before the 0th energy therefor it is also checked for sign changes.

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

4.1.1. Accuracy

For a benchmark the values mentioned below, will be used.

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

To get the actual values the Wolfram Language with WolframScript will be used. WolframScript is a programming language similar to Wolframalpha that can calculate the integral analytically and precisely. In Rust `main` can be rewritten to

```

1 fn main() {
2   let output_dir = Path::new("output");
3
4   let values = (0..=50)
```

```

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

```

this will output all energy levels from $n = 0$ to $n = 50$. The same procedure is implemented in WolframScript.

```

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

```

These programs will output two files `energies_approx.dat` (Appendix B.1) for the implementation in Rust and `energies_exact.dat` (Appendix B.1) for WolframScript. As a ruff estimate an error of $\pm \frac{10}{64000} \approx \pm 1.56 \cdot 10^{-4}$ is expected, because the program checks for energies with that step size.

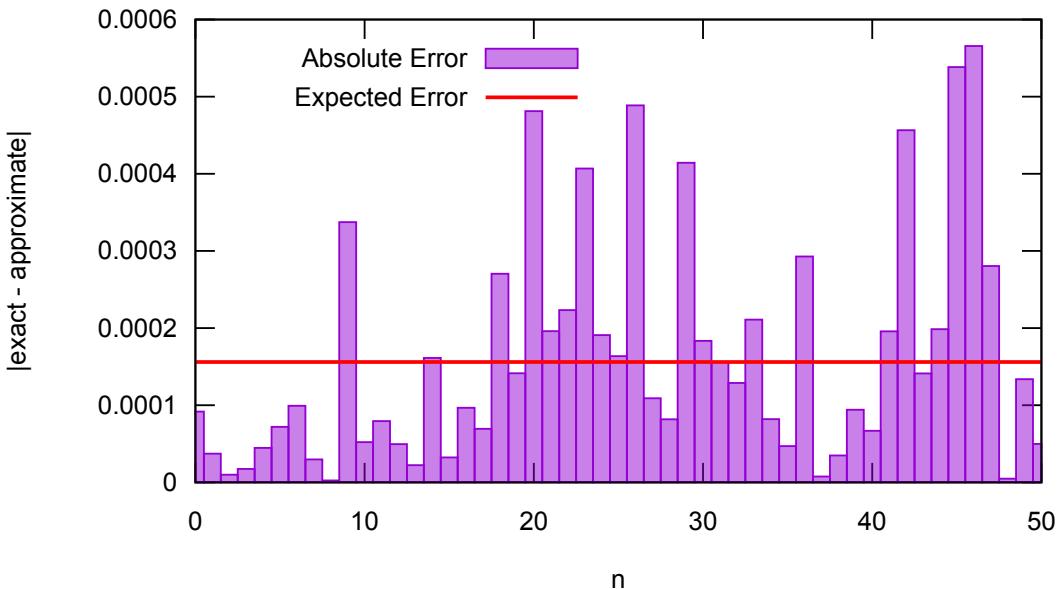


Figure 4.1.: Absolute error of energy levels in square potential

The absolute error is plotted in figure 4.1. The error is a little higher than expected which is probably due to errors in the integral. Still the algorithm should be precise enough. If you'd like you could pick a lower value for `ENERGY_STEP` in `src/energy.rs:49`, but this will impact the performance for calculating energies with higher numbers for n .

4.2. Approximation Scheme

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

The second is Density functional field theory would be interesting to add to the program in the future, but it would require major changes in the architecture.

The program uses the third method WKB approximation, it is applicable to a wide verity of linear differential equations and works very well in the case of the Schrödinger equation. Originally it was developed by Wentzel, Kramers and Brillouin in 1926. It gives an approximation to the eigenfunctions of the Hamiltonian \hat{H} in one dimension. The approximation is best understood as applying to a fixed range of energies as \hbar tends to zero (Hall, 2013, p. 305). Even though in Planck units $\hbar = 1$ the approximation is still valid because it actually

actually assumes that other terms independent to \hbar tend to 0.

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

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

$$V(t) - E = 0 \quad (4.7)$$

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

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

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

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

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

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

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

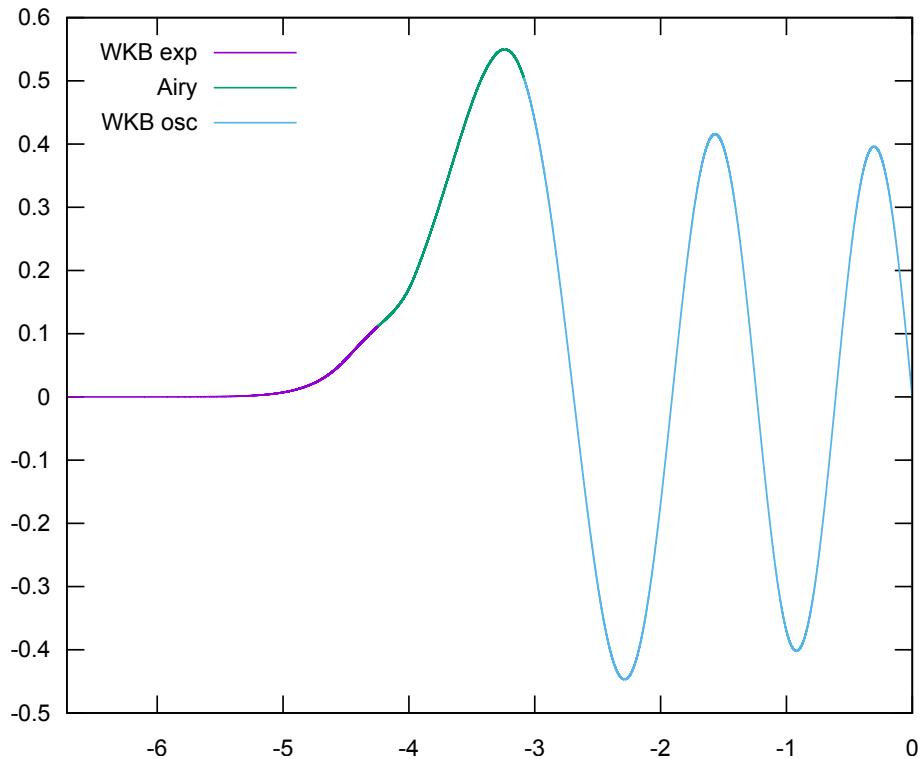


Figure 4.2.: Left half of wave function with $N_{Energy} = 9 \Rightarrow E \approx 13.4$, $m = 2$, $V(x) = x^2$

In figure 4.2 the three parts are visualized. The purple section on the left is the exponential decaying part $\psi_{exp}^{WKB}(x)$, equation 4.8 is calculated according to (Hall, 2013, p. 317, Claim 15.7) where b and a are different solutions for t of equation 4.7. The absolute symbol makes it possible to not differentiate between the case where $x < t$ and $x > t$.

4.2.1. Validity

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

$$p(x) = \sqrt{V(x) - E}$$

$$\left| \frac{dp}{dx}(x) \right| \ll p^2(x)$$

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

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

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

4.2.2. Implementation

WKB

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

```

1 fn eval(&self, x: f64) -> Complex64 {
2     let val = if self.phase.energy < (self.phase.potential)(x) {
3         self.psi_exp(x)
4     } else {
5         self.psi_osc(x)
6     };
7
8     return (self.op)(val);
9 }
```

The term `self.op` has been an attempt to use both turning points for the oscillating region. It wasn't removed because the current method is not perfect either and it can be used in the future to improve the accuracy.

In the exponential part we will always use the corresponding turning point and because we're working with two separate turning points in the same function it is possible that the sign of the exponential part doesn't match with the sign of the oscillating part. To fix this, we can define the `get_exp_sign` function.

```

1 pub fn get_exp_sign(&self) -> f64 {
2     let limit_sign = if self.turning_point_exp == self.turning_point_osc {
3         1.0
4     } else {
5         -1.0
6     };
7
8     (self.psi_osc(self.turning_point_exp + limit_sign * f64::EPSILON.sqrt()) / self.c
9      )
10    .re
11    .signum()
```

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

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

Airy

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

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

4.3. Turning Points

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

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

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

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

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

The left boundary point must have a positive and the right a negative derivative. This means we can solve for roots and group them together by there derivatives.

In order to find all roots we will use a modification of Newtons method. When we find a solution, x_0 we can divide the original function by $(x - x_0)$ this means that Newtons method wont be able to find x_0 again.

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

$$\begin{aligned} t_l &< t_r \\ (t_l - f_{view}(t_r - t_l), t_r - f_{view}(t_r - t_l)) \end{aligned}$$

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

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

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

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

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

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

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

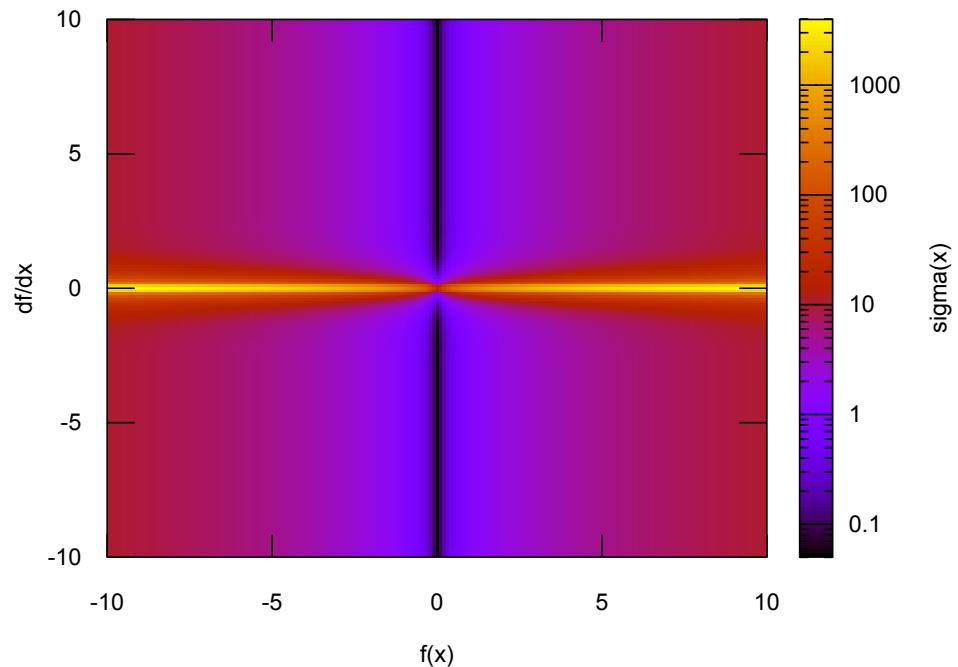


Figure 4.3.: Logarithmic heat diagram of $\sigma(x)$, darker/bluer is better

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

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

```

1 fn find_zeros(phase: &Phase, view: (f64, f64)) -> Vec<f64> {
2     let phase_clone = phase.clone();
3     let validity_func = Arc::new(move |x: f64| {
4         1.0 / (2.0 * phase_clone.mass).sqrt() * derivative(&|t| (phase_clone.
5             potential)(t), x).abs()
6         - ((phase_clone.potential)(x) - phase_clone.energy).pow(2)
7     });
8     let mut zeros = NewtonsMethodFindNewZero::new(validity_func, ACCURACY, 1e4 as
9         usize);
10
11    (0..MAX_TURNING_POINTS).into_iter().for_each(|_| {
12        let modified_func = |x| zeros.modified_func(x);
13
14        let guess = make_guess(&modified_func, view, 1000);
15        guess.map(|g| zeros.next_zero(g));
16    });
17
18    let view = if view.0 < view.1 {
19        view
20    } else {
21        (view.1, view.0)
22    };
23    let unique_zeros = zeros
24        .get_previous_zeros()
25        .iter()
26        .filter(|x| **x > view.0 && **x < view.1)
27        .map(|x| *x)
28        .collect::<Vec<f64>>();
29
30    return unique_zeros;
31 }

```

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

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

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 `NewtonMethodFindNewZero`. Unfortunately this procedure can't be implemented asynchronously since you have to know all previous zeros before you can find a new one.

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

4.4. Wave Function Parts

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

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

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

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

4.4.1. ApproxPart

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

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

```
1 fn eval(&self, x: f64) -> Complex64 {
2     if is_in_range(self.airy_join_l.range(), x) && ENABLE_AIRY_JOINTS {
3         return self.airy_join_l.eval(x);
4     } else if is_in_range(self.airy_join_r.range(), x) && ENABLE_AIRY_JOINTS {
5         return self.airy_join_r.eval(x);
6     } else if is_in_range(self.airy.ts, x) {
7         return self.airy.eval(x);
8     } else {
9         return self.wkb.eval(x);
10    }
11 }
```

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

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

(see Appendix A.6). This means that in theory the program should run a little faster if `ENABLE_AIRY_JOINTS` is disabled. This has to be taken into account when benchmarking.

4.4.2. PureWkb

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

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

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

4.5. Wave Function

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

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

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

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

```
1 (
2     lower_bound * (upper_bound - lower_bound) * view_factor,
3     upper_bound * (upper_bound - lower_bound) * view_factor,
4 )
```

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

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

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

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

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

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

None The solution wont be multiplied by anything.

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

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

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

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

4.5.1. Super Position

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

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

5. Program Manual

5.1. Installation of schroedinger_approx

This section is a guide on how to install *schroedinger_approx* on your computer. Please follow the section for your operating system.

5.1.1. Linux

Ubuntu

Run the following command to install the dependencies

```
1 sudo apt-get install gnuplot build-essential git libclang-dev
2 sudo snap go --classic
3 curl --proto '=https' --tlsv1.2 -sSf https://sh.rustup.rs | sh
4 source ~/.profile
5 rustup toolchain add nightly
```

The curl command that installs Rust will prompt you to choose between installation options, you can choose number 1.

After you've run the script above to install all the dependencies, you can go to the directory where you'd like to install *schroedinger_approx*. Then run the command

```
1 git clone https://github.com/Gian-Laager/Schroedinger-Approximation.git
```

to download the code from GitHub.

Arch Linux

Run the following command to install all the dependencies

```
1 sudo pacman -Sy gnuplot go git clang
2 curl --proto '=https' --tlsv1.2 -sSf https://sh.rustup.rs | sh
3 source ~/.profile
4 rustup toolchain add nightly
```

The curl command that installs Rust will prompt you to choose between installation options, you can choose number 1.

After you've run the script above to install all the dependencies, you can go to the directory where you'd like to install *schroedinger_approx*. Then run the command

```
1 git clone https://github.com/Gian-Laager/Schroedinger-Approximation.git
```

to download the code from GitHub.

5.1.2. macOS

The following instructions have only been tested under *macOS Ventura 13.0.1* on an Intel CPU. This means the instructions might not work for Apple silicon.

First you'll need to install the package manager *Homebrew*, it can be used to install software without downloading all the installers manually. To install it open the *Terminal* program, paste the command below and press enter.

```
1 /bin/bash -c "$(curl -fsSL https://raw.githubusercontent.com/Homebrew/install/HEAD/install.sh)"
```

Then you should be asked for your password, type your password (no text will be written while you type your password) and press enter. Next it will ask you to continue the installation, press enter. This step will take a while because it has to download *Xcode*.

After we've installed Homebrew, we can install the dependencies of *schroedinger_approx* with the command

```
1 brew install gnuplot go gcc
```

Next we need to install Rust with the command

```
1 curl --proto '=https' --tlsv1.2 -sSf https://sh.rustup.rs | sh
2 source ~/.profile
3 rustup toolchain add nightly
```

This will ask you to confirm the installation, type *1* and enter.

After we've installed all the dependencies you can go to <https://github.com/Gian-Laager/Schroedinger-Approximation> and click the green *Code* button, where you can download the code as a ZIP file.

5.1.3. Windows

Unfortunately *schroedinger_approx* can't be installed on Windows directly, but since Windows already supports Linux out of the box in WSL, you can download the latest version of *Ubuntu* from the Microsoft Store (<https://www.microsoft.com/store/productId/9PDXGNCFSCZV>).

But first the *WSL optional component* has to be enabled. To do this open the *CMD* as an administrator and run the command

```
1 wsl --install
```

and reboot once it's finished.

After rebooting open *Ubuntu*, this will start the installation automatically. Then follow the instructions of the terminal. Once you're logged in run the commands

```
1 sudo apt-get update
2 sudo apt-get install build-essential libclang-dev
3 sudo snap install go --classic
4 curl --proto '=https' --tlsv1.2 -sSf https://sh.rustup.rs | sh
5 source ~/.profile
6 rustup toolchain add nightly
```

To install *gnuplot* open the link <http://gnuplot.info> (you might get a warning that the site is not secure, ignore it and proceed). There follow the link in the gray box with the title *Version [...] (current)* that says *Release [...] ([Date])*. The current version [...] at the time of writing this document is 5.4.5 but you might have a newer version available that should also work.

After downloading the latest version on *sourceforge* run the installer. In the section *Select Additional Tasks* set the option

Add application directory to your PATH environment variable
to true.

Now you can download the *schroedinger_approx* from <https://github.com/Gian-Laager/Schroedinger-Approximation>. Click the green *Code* button and download the code as a ZIP file and extract it to a directory of your choice. Then you can *drag and drop* the directory to the Ubuntu terminal. This should look something like this

```
1 C:\Users\gianl\OneDrive\Desktop\Schroedinger-Approximation-master
```

You then need to replace all the \ to / and change C: to /mnt/c (if you have another letter use it's lowercase at /mnt/<your letter>) and add a cd at the front. For the example above it would look like this

```
1 cd /mnt/c/Users/gianl/OneDrive/Desktop/Schroedinger-Approximation-master
```

To run the program, run the command

```
1 cargo run --release
```

5.2. Usage

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 than INTEG_STEPS.

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

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

Visual Configurations

Adjust the width of joints

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

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

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

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

5.3. WaveFunction

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

```
1 let wave_function = wave_function_builder::WaveFunction::new(
2     /*potential*/,
3     /*mass*/,
4     /*nth energy*/,
5     APPROX_INF,
6     1.5,
7     ScalingType::/*Scaling*/,
8 );
```

The example above has to be placed right after the `fn main()` line. You have to replace all the commentaries (`/*...*/`) with the values you want. For the first you can choose a potential from section 5.6 for this you can type `potentials::/*potential*/`.

For the Mass you can just use a normal float.

“nth energy” must be a positive integer (including 0) and is the nth energy level of the potential.

And as for the scaling type, choose one of the options described at the end of section ??.

5.4. SuperPosition

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

```
1 let wave_function = wave_function_builder::SuperPosition::new(
2     & /*potential*/,
3     /*mass*/,
4     &[
5         /*nth energy*/, /*phase*/,
6         /*nth energy*/, /*phase*/,
7         // ...
8     ],
9     APPROX_INF,
10    1.5, // view factor
11    ScalingType::/*scaling*/),
12 );
```

Just like in section 5.3 you have to replace all the commentaries (`/*...*/`) with the values you want.

“potential” you have to choose a potential from section 5.6.

“mass” your mass as a float.

“nth energy ” must be a positive integer (including 0) and is the nth energy level of the potential.

“phase” a complex number that the wave function with the corresponding energy will be multiplied by. To make a complex number you can use `complex(/*Re*/, /*Im*/)`.

“// ...” you can add as many energies as your computer can handle.

And as for the scaling type, choose one of the options described at the end of section ??.

5.5. Plotting

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

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

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

x	Re	Im
1.0	2.718	3.141
2.0	1.414	1.465

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

5.5.1. WaveFunction

For a `WaveFunction` as we’ve seen in section 5.3 you have three options.

plot_wavefunction

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

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

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

plot_wavefunction_parts

Each pair of ψ_{osc}^{WKB} and ψ_{exp}^{WKB} will be plotted in different colors and the Airy functions will also be plotted separately.

plot_probability

This function will plot the probability $\|\Psi(x)\|^2$ of the wave function.

5.5.2. SuperPosition

plot_superposition

This function plots the superposition analogous to `plot_wavefunction` for super positions.

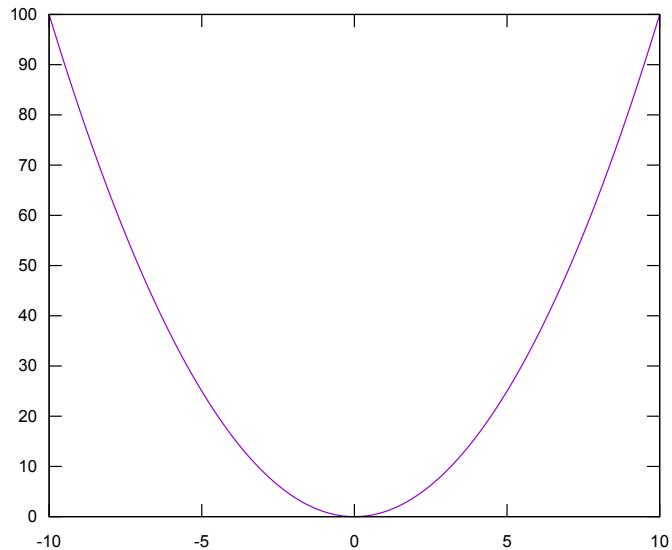
plot_probability_superposition

Plots the probability $\|Psi(x)\|^2$ of the super position analogous to `plot_probability`.

5.6. Potentials

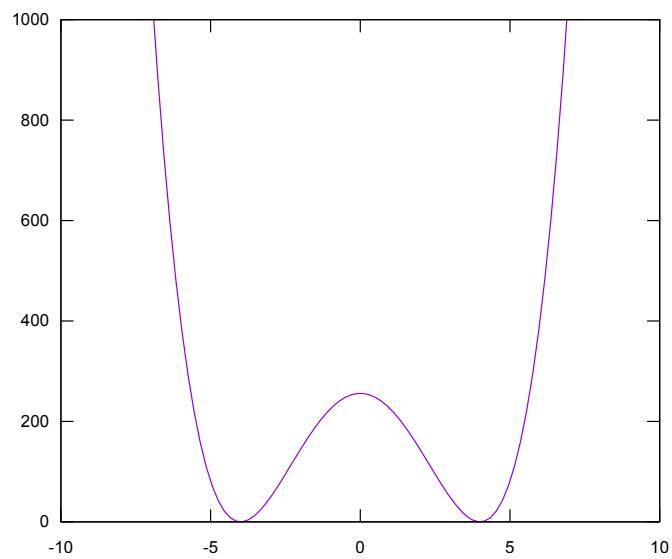
square Normal square potential as used in Hall (2013).

$$x^2$$



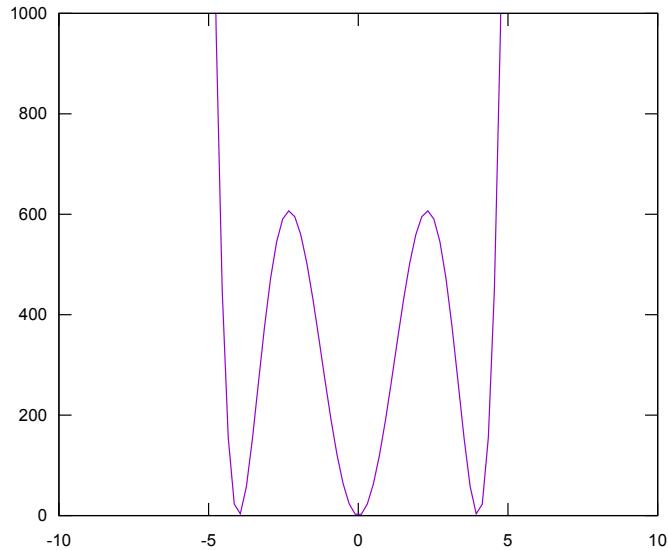
mexican_hat 4th degree polynomial that looks like a mexican hat, with 2 minima.

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



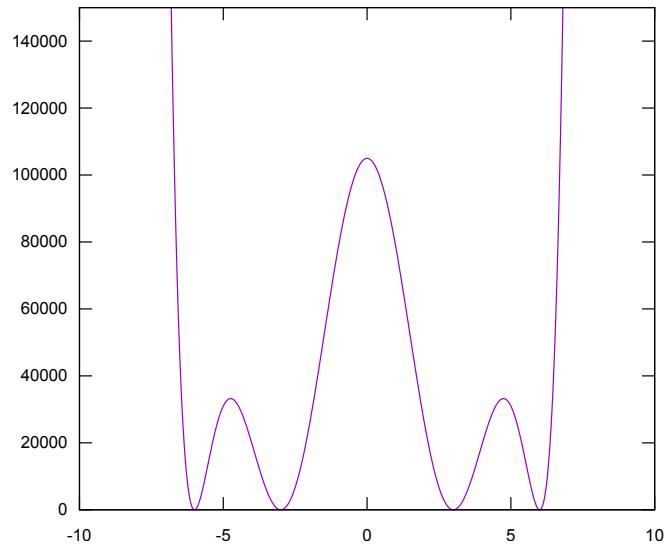
double_mexican_hat 6th degree polynomial that has 3 minima.

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



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.

5.6.1. Custom Potentials

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

```
1 fn my_potential(x: f64) -> f64 {  
2     return /*some calculation*/;  
3 }
```

`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 ($-e^{-x^2} + 1$)

```
1 fn neg_bell(x: f64) -> f64 {  
2     return -(-x.powi(2)).exp();  
3 }
```

General polynomial (might not work for all configurations)

```
1 const COEFFICIENTS: [f64;4] = [a, b, c, d]  
2 fn polynom(x: f64) -> f64 {  
3     let mut result = 0.0;  
4     for n in 0..COEFFICIENTS.len() {  
5         result += x.powi(n) * COEFFICIENTS[n];  
6     }  
7     return result;  
8 }
```

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 $a + bx + cx^2 + dx^3$.

6. Results

6.1. Wave Functions

6.1.1. Hall Example

As a first result lets replicate the example from hall with the 39th energy of a square potential.

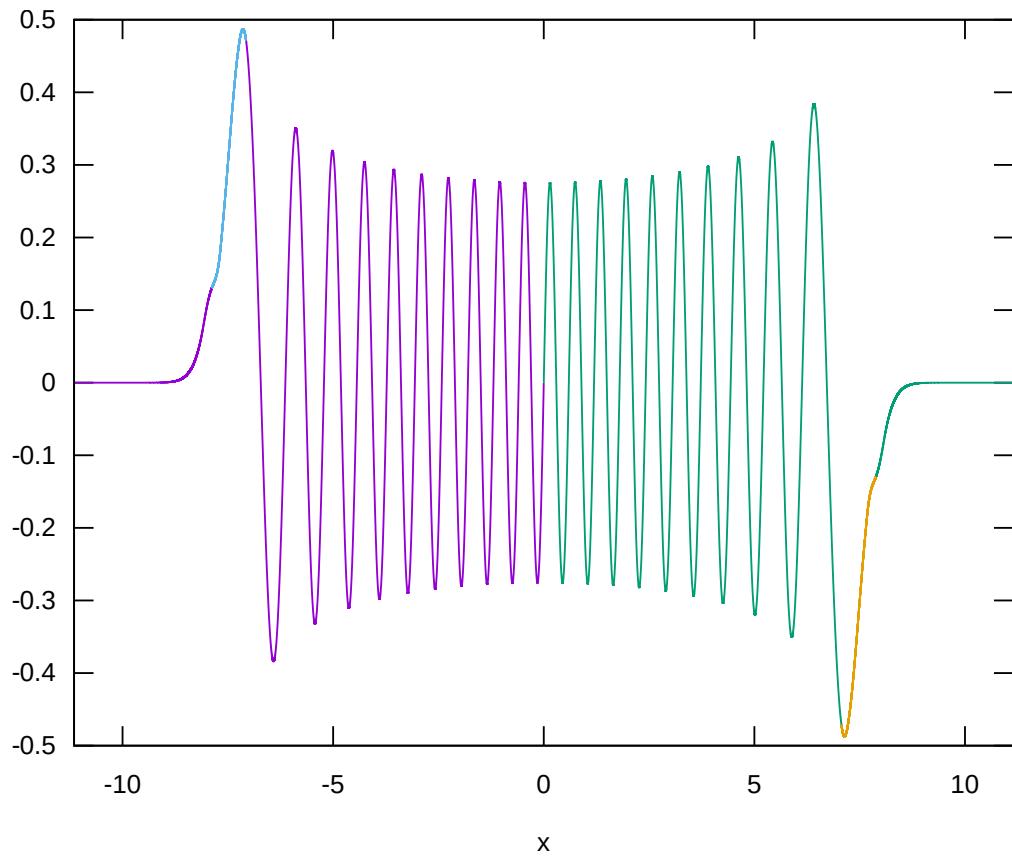


Figure 6.1.: Wave function of 39th energy with $V(x) = x^2$, $m = 1$ and $f_{view} = 0.1$.

This result is very similar to the plot (Hall, 2013, fig. 15.5). The only difference is the joint between the Airy function and the exponential WKB part. In our case the two functions don't meat as nicely. Overall the most important thing ought to be the number of maxima and minima which do match.

6.1.2. Phase Shift

Because the Schrödinger equation is linear we can rotate the wave function in the complex plane. For an example we will use a phase of $e^{i\frac{\pi}{4}}$ on the wave function of a square potential.

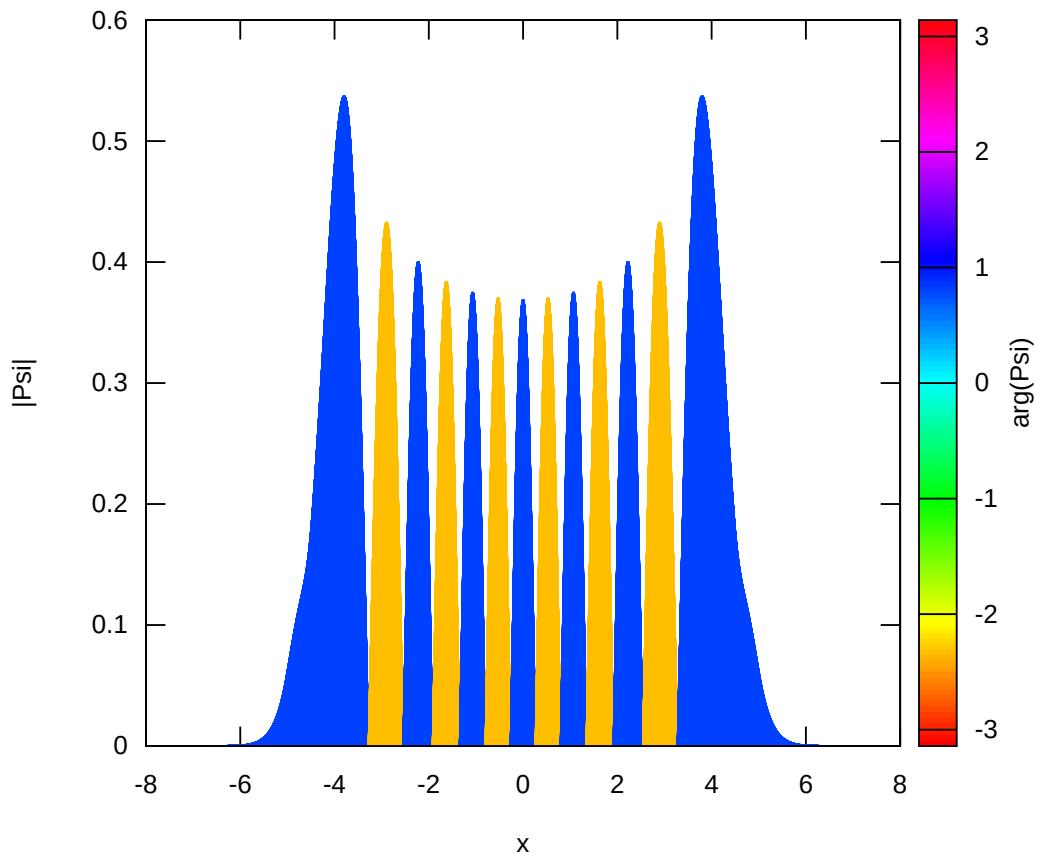


Figure 6.2.: Wave function of 12th energy with $V(x) = x^2$, $m = 1$ rotated by $\frac{\pi}{4}$.

6.1.3. 0th Energy

In quantum mechanics the 0th energy is not always 0. As an example we will take the 0th energy of a square potential.

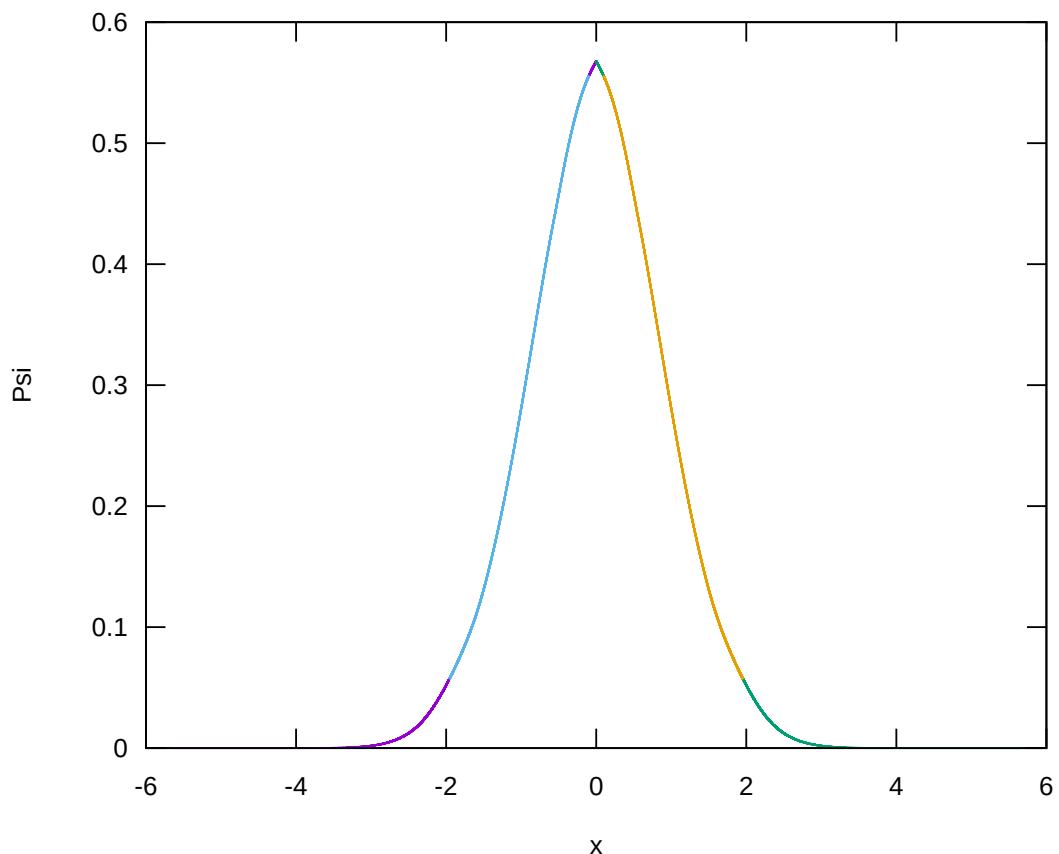


Figure 6.3.: Wave function of 0th energy with $V(x) = x^2$.

Unfortunately there's a discontinuity at $x = 0$. This shouldn't happen when only using one turning point and it only seems to be a problem with the 0th energy

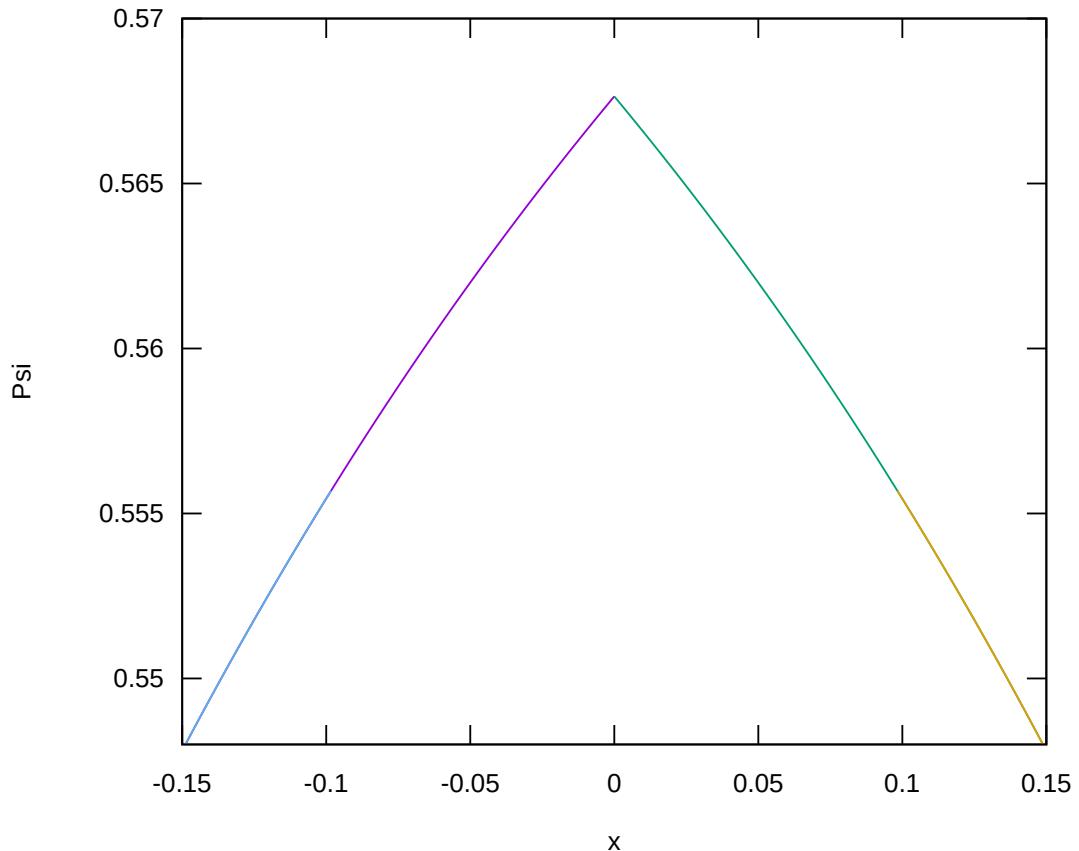


Figure 6.4.: Zoom of wave function of 0th energy with $V(x) = x^2$.

As far as the program is concerned $\Psi(x) = 0$ is a valid solution but this has to be done in a super position of the same energy with destructive interference.

$$\Psi_{super}(x) = 1 \cdot \Psi(x) - 1 \cdot \Psi(x) = 0$$

In theory this is possible but can't be physically valid because the Schrödinger equation does not show the full picture. In quantum field theory the wave function would always oscillate in some way.

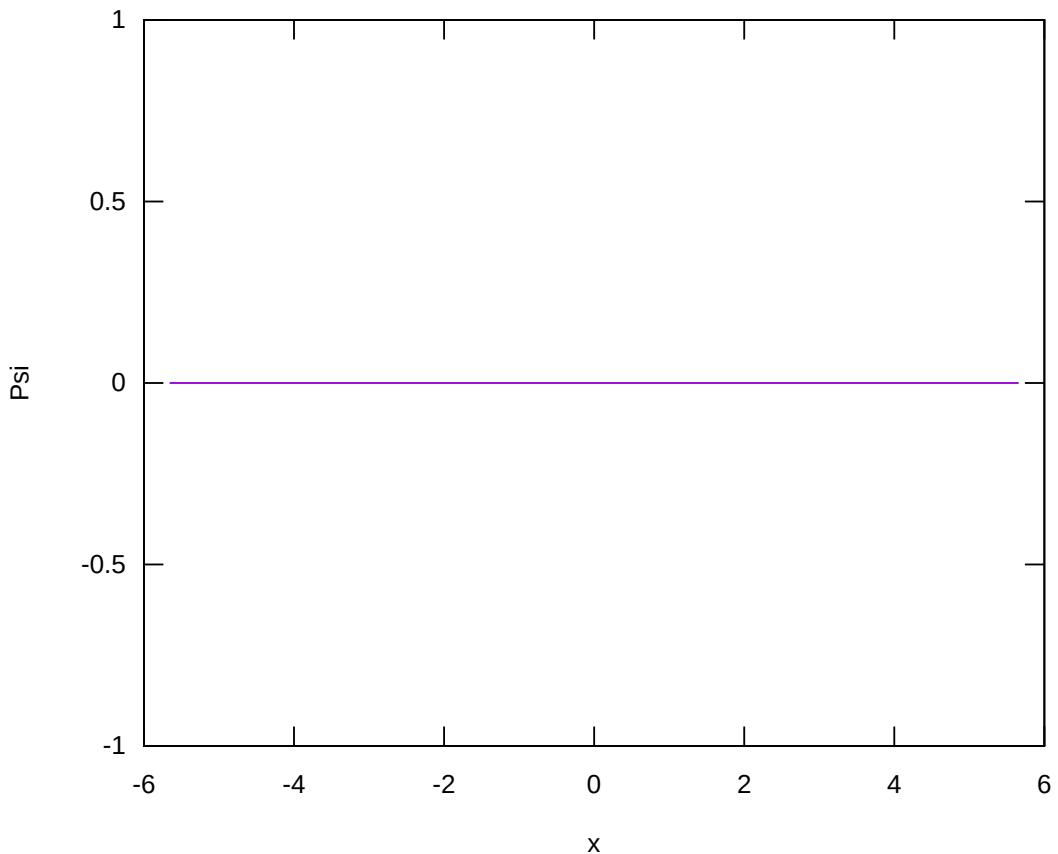


Figure 6.5.: Destructive interference of 0th energy with $V(x) = x^2$.

$\Psi(x) = 0$ would still contain energy we just can't tell because energy usually only emerges in the time dependent Schrödinger equation.

6.2. Mexican Hat Potential

For this example we will use the “mexican hat” potential.

$$V(x) = (x - 4.0)^2(x + 4.0)^2$$

It is particularly interesting because it has a maxima. This means that at low engineries it will form two oscillations around the two minima.

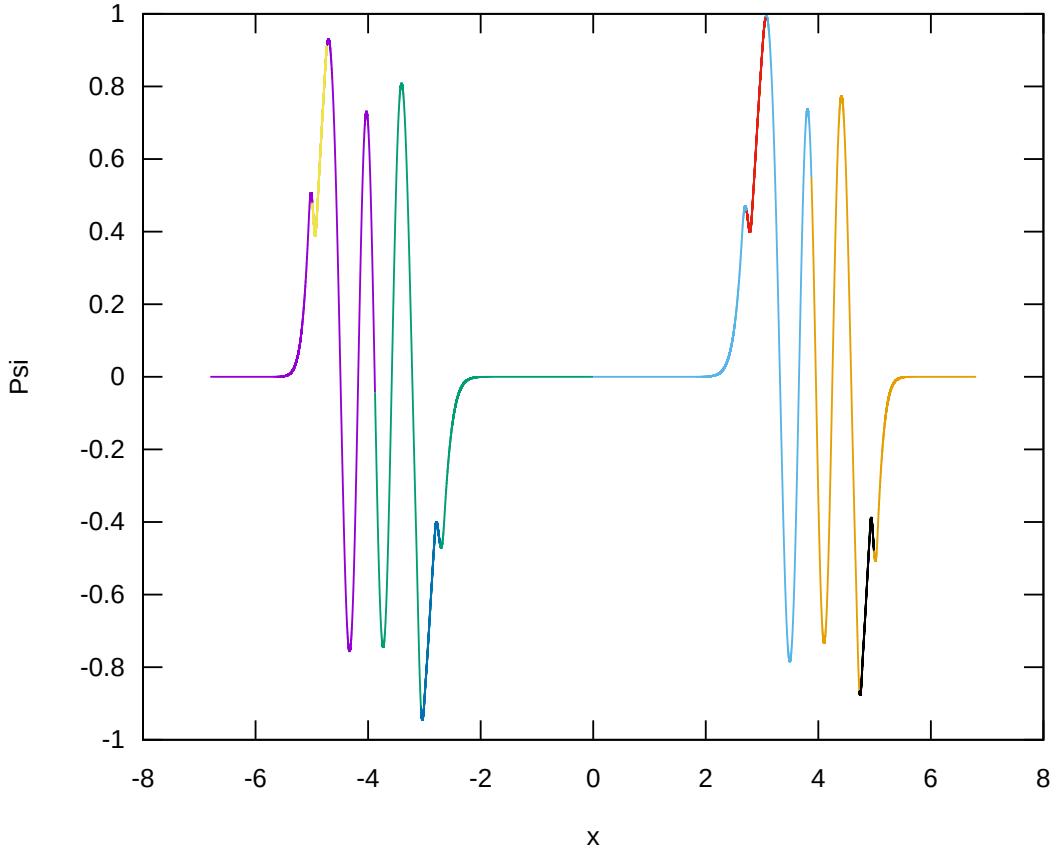


Figure 6.6.: Wave function of a mexican hat potential, with n th energy 10 and $m = 1$. The wave function oscillates in two intervals $(-4.85, -2.89)$ and $(2.89, 4.85)$. Between these intervals function decease exponentially. The transition region from exponential WKB to Airy appears to have a bigger dependency then with a square potential. Even though the two functions should meat at the same point the exponential WKB part seems to have a larger magnitude.

If the energy is high enough, the two oscillations will eventually merge into one as shown in figure 6.7.

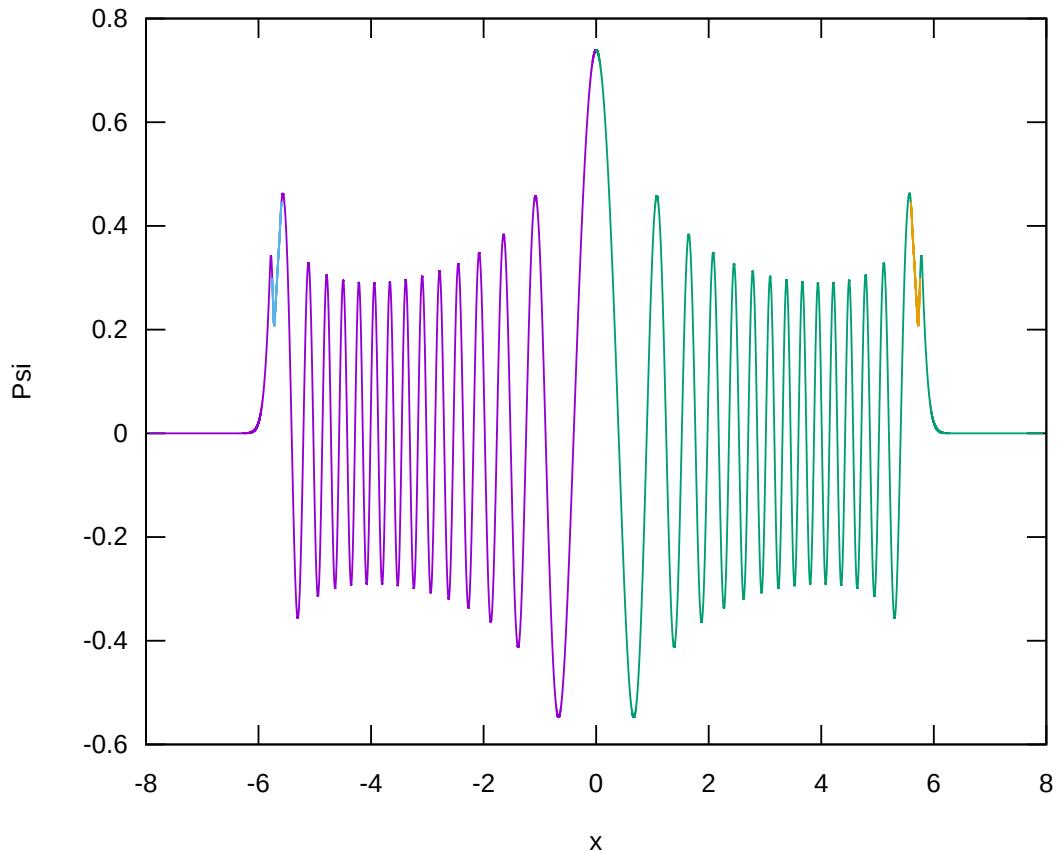


Figure 6.7.: Wave function of mexican hat potential, with n th energy 56 and $m = 1$. This is the first energy where the oscillating parts combine. The middle of the function has a rather low frequency compared to the other oscillating parts.

Unfortunately the program is not able to calculate the 55th energy because Newtons method fails to find the turning points in the middle. And with 54th energy the program doesn't generate any errors but the region around $x = 0$ can't be correct (Figure A.1), this occurs because the program fails to detect all the turning points.

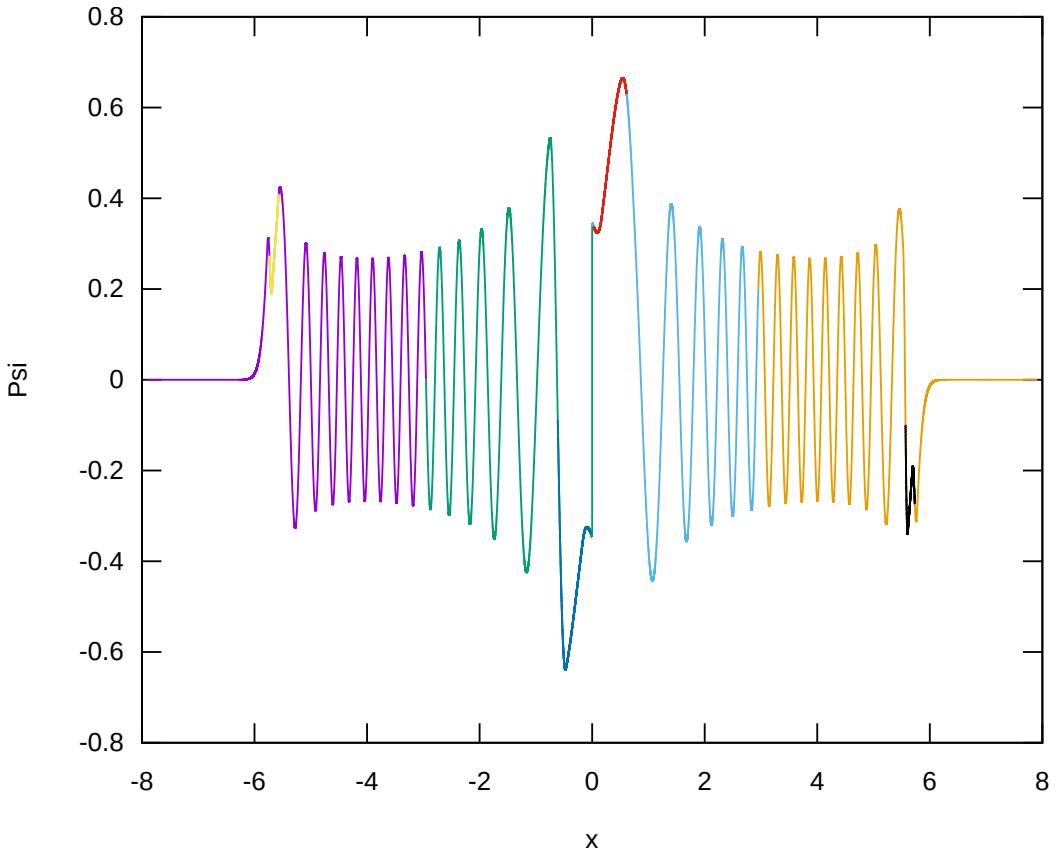


Figure 6.8.: Wave function of mexican hat potential, with 53rd energy and $m = 1$. Because the oscillating parts are right at the boundary to merge, the wave function has a discontinuity that has been generated by the program at $x = 0$.

But the 53rd energy can be calculated. However there's a discontinuity at $x = 0$ as shown in figure 6.8. This happens because there would be an extra term in the approximation that handles these cases. When this extension has been implemented, there were problems that most of the wave function would diverge to infinity.

6.2.1. Super Position

While some wave functions with super position of energy seem to be chaotic, others are in some way beautiful because of there symmetries. Unfortunately the color plots usually don't make the symmetries that arise when plotted in 3D obvious.

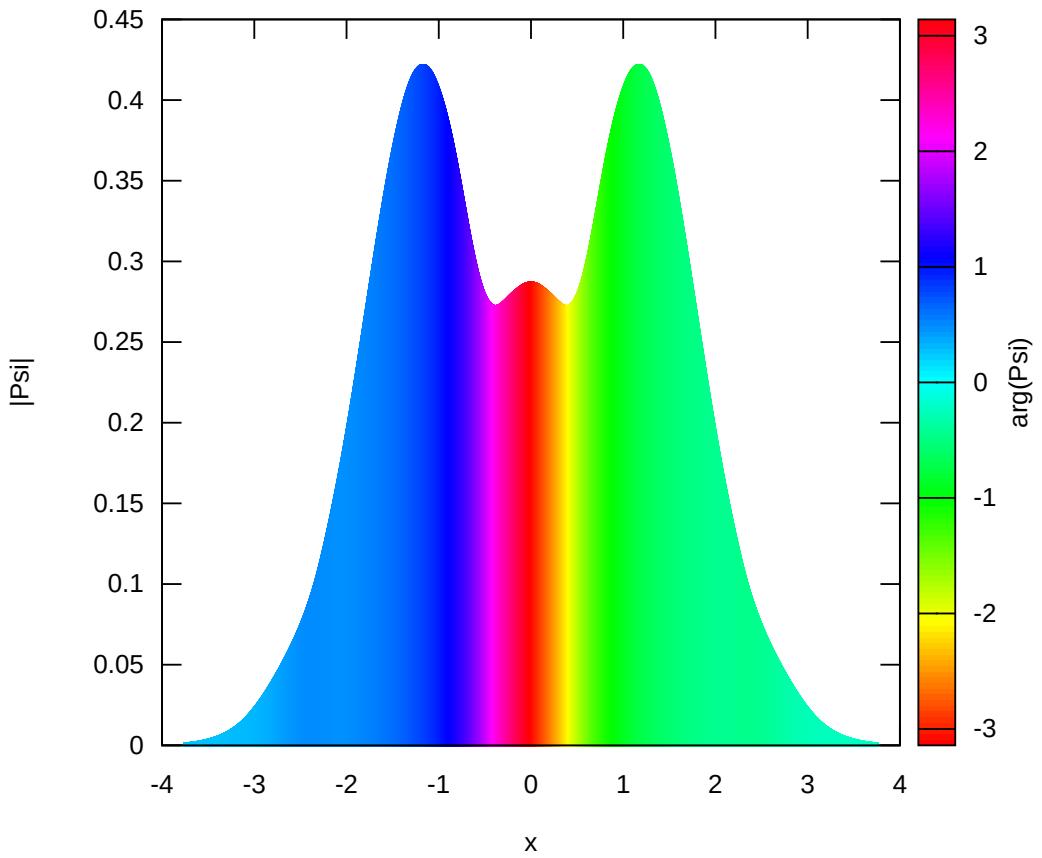


Figure 6.9.: Wave function of square potential, super position of 1 times 1st energy and i times 2nd energy. The function has three local maxim, it takes a full “turn” around the complex plane because all the colors only occurs once. Further on the two maxima at $x \approx \pm 1.17$ are in opposite directions concerning the angle of the complex plane.

This is probably one of the simplest super position that can be made. In 3D it looks just like a single loop around the x-axis that emerges out of nowhere and disappears again by exponentially decaying.

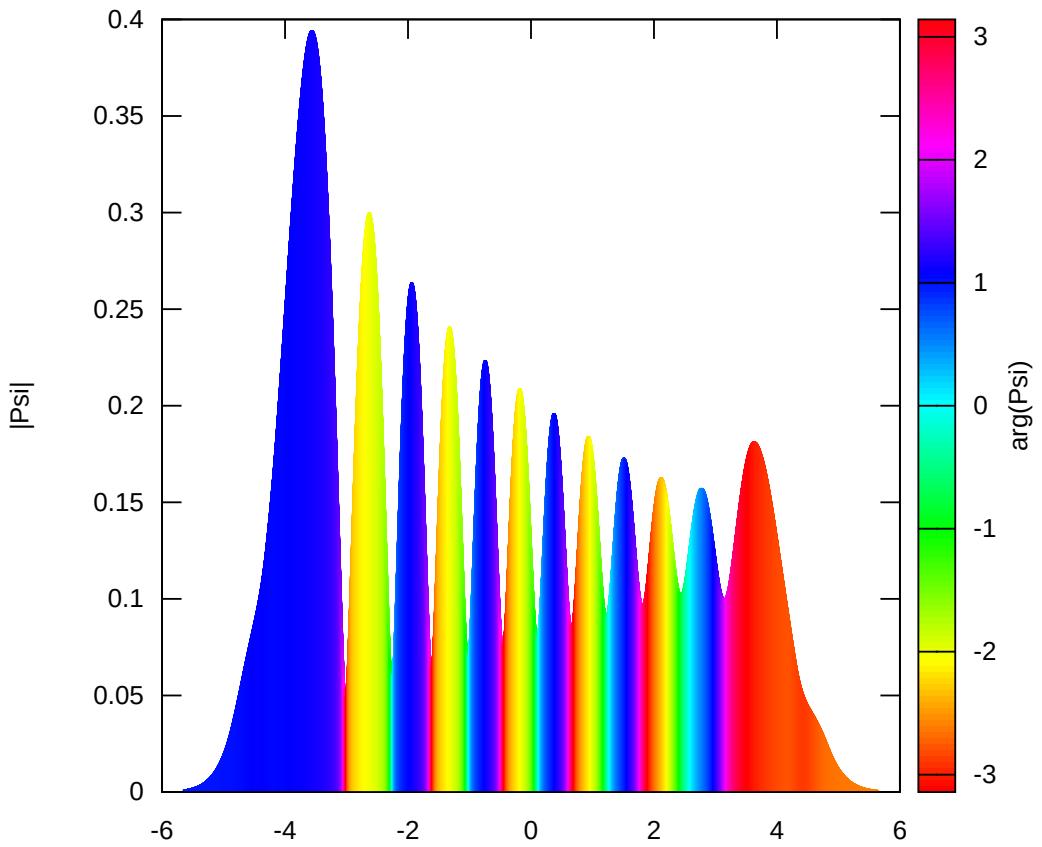


Figure 6.10.: Wave function of square potential, super position of 1 times 10th energy and $1 + i$ times 11th energy. The function has a higher probability towards the left which is interesting because both the wave functions for the 10th and 11th energy are symmetrical with respect to the y-axis. Therefore this is an example of quantum interference. The two wave functions interact in such a way that the state on the left is far more likely.

The same quantum interference as shown in figure 6.10 can be inverted such that a position on the right is more likely by changing the factor of $1 + i$ to $-1 + i$ as shown in figure 6.11.

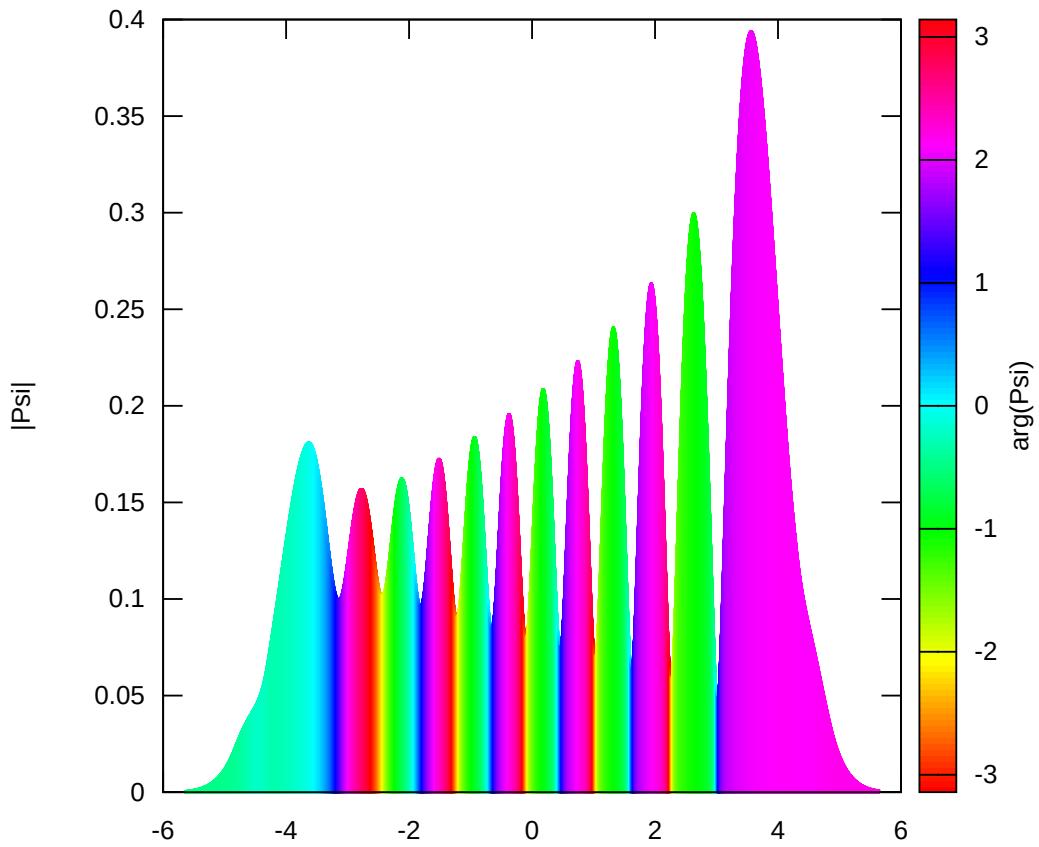


Figure 6.11.: Wave function of square potential, super position of 1 times 10th energy and $-1 + i$ times 11th energy. Just like figure 6.10 interference occurs, but in this superposition the probability on the right side of the plot is higher.

6.2.2. Relationship between Energy and Mass

6.3. Performance

In the appendix there's a detailed description of the test device A.1.1.

The overall “felt performance” by the user is from the experience of the author quite good. As expected higher energies generally take longer to run. The same also applies to potentials where simple potentials such as x^2 are quite fast and complex potentials such as *triple mexican hat* take longer. Overall the goal that the program should give a quick feed back loop for simple inputs has been achieved because the results were done in under a minute. Particularly the energies 1-6 of x^2 can on the test machine be calculated in under 10 s, energies 1-57 in under 30 s and 1-155 in under 1 min. The 155th energy isn't a particularly low energy yet the program still managed to complete in under a minute.

Further on despite the “felt” performance of the user the benchmarks also show good results. As expected the most time is spent on integration. In particular finding energies. Even though energies in an interval of 10 units are found in almost the same time frame (output A.3). This is expected due to the algorithm described in section 4.1. These could be fixed by using a grater search area by increasing the variable ENERGY_STEP in `src/energy.rs:47`. The overall evaluation process is fast with 100 samples in 10 ms. One concern also was that finding turning points would be slow but the measurements indicate that these calculations take less then a millisecond. Even though the bench mark for renormalization is fast, when applied to actual wave functions, it’s really slow because thousands of points have to be calculated.

Performance statistics from perf suggest that the program utilizes the CPU well. The memory layout also seems to be good.

All the benchmark results can be found in appendix A.2.

6.4. Accuracy

To determine the accuracy, the mathematically exact solutions to the Schrödinger equation has to be calculated. This can simple potentials be done with WolframScript. In the `exact.wsl` file a exact solutions for the potential x^2 can be calculated. Note that the energies are still calculated with the Maslov-corrected Bohr–Sommerfeld condition is used to calculate the energies which is an approximation (Hall, 2013, p. 307).

When running the script for the 5th energy level, two linearly independent solution `psi1` and `psi2` are calculated.

```
1 psi1[x] = ParabolicCylinderD[4.99999999999998, 1.6817928305074292*x]
2 psi2[x] = ParabolicCylinderD[-5.99999999999998, (0. + 1.6817928305074292*I)*x]
```

`psi2` will be ignored because it diverges to infinity. Afterwards because `psi1[x]` is not normalized, it is multiplied such that the maxima of the exact solution matches the maxima of the solutions calculated by `schroedinger_approx`. In figure 6.12 the two solutions are plotted together.

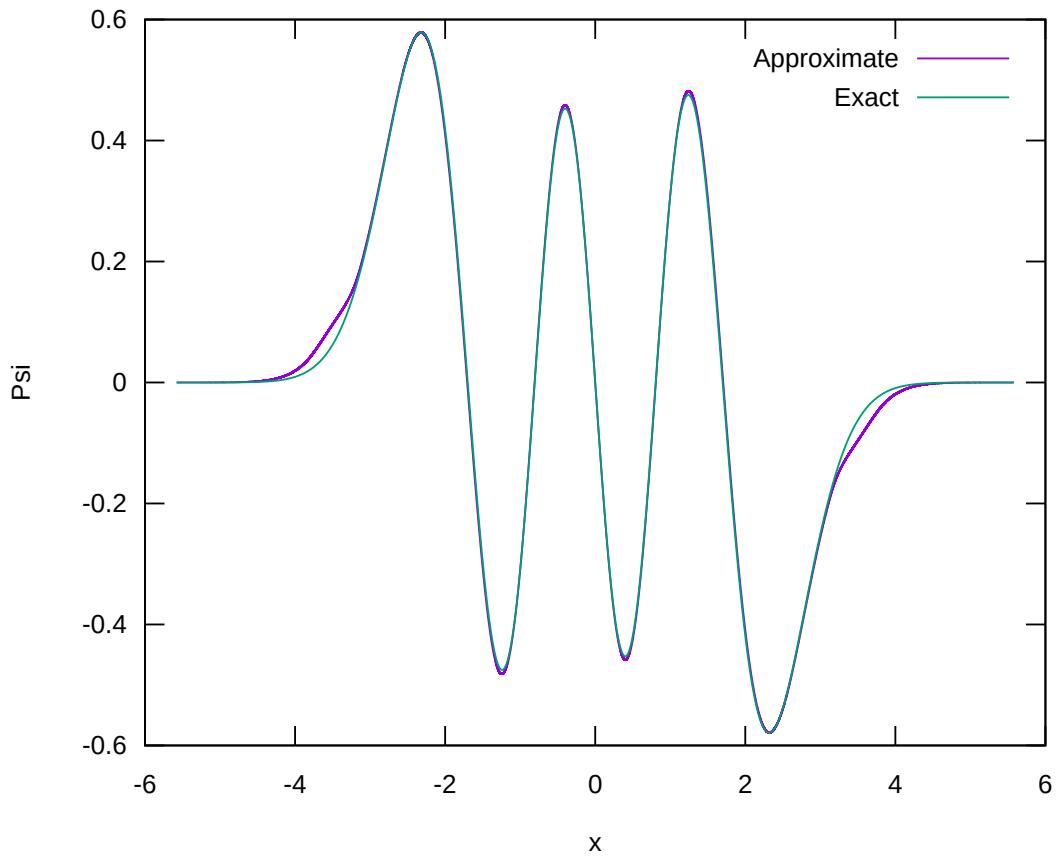


Figure 6.12.: The exact and approximate solution of the 5th energy of the potential x^2 . As expected the grates error can be seen where the joint between the Airy and exponential part was inserted. Otherwise the approximate solution is so close to the real solution that the difference is barely noticeable.

The absolute error is plotted in figure 6.13. Even though the exact solution to the mexican hat potential was not calculated, the error at the transition regions gets worse because one would expected a smooth exponential decay.

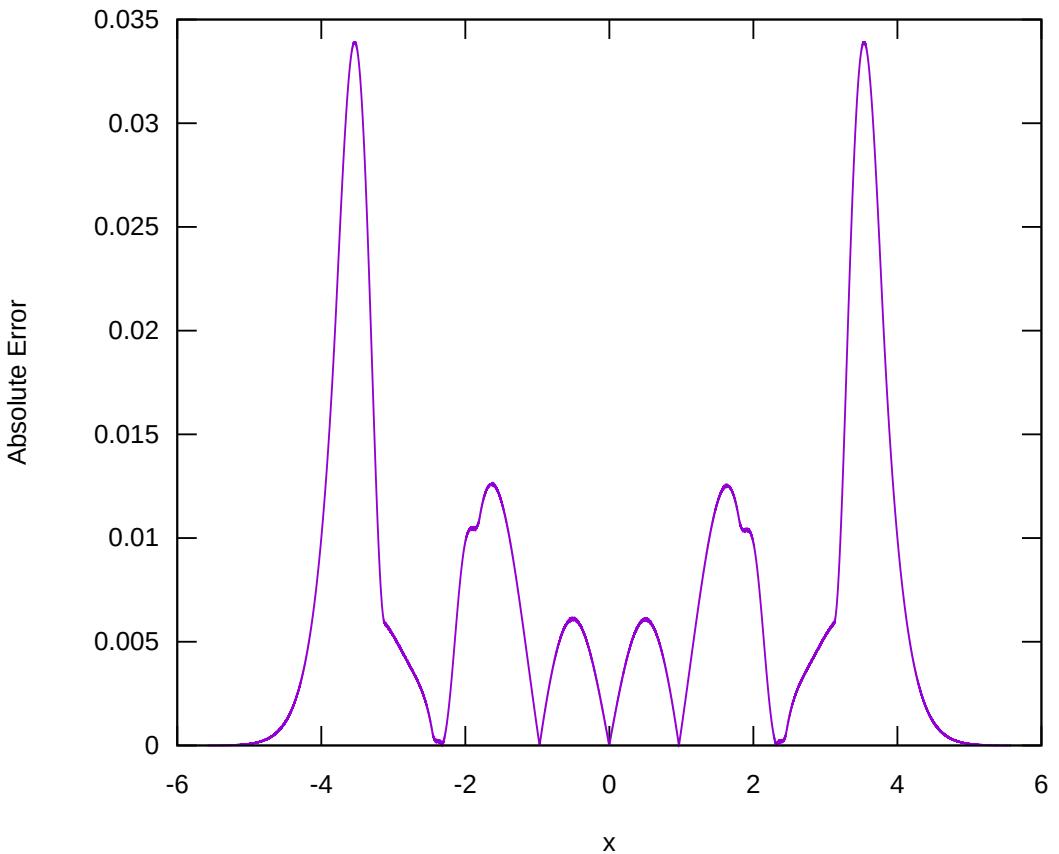


Figure 6.13.: Absolute error of the exact and approximate solution of the 5th energy of the potential x^2 . The error in the semiclassical region is low around 0.01-0.015, this meets the goal that the wave function should be “visually accurate”. At the transition regions the error reaches its maximum at around 0.034.

The error at the local maxima, as shown in figure 6.14, of the wave function could potentially be lower because the scaling factor of the exact solution is not optimal. This could be achieved by solving for a minimal area under the curve of the absolute error.

For other energies then 5 the same pattern occurred, the error is greatest at the local maxima of the wave function and even bigger at the transition region.

This error can be minimized by tuning the `VALIDITY_LL_FACTOR` constant that makes the Airy functions larger. But this has to be done for each potential separately and the default value of 3.5 should give reasonably good results for most potentials.

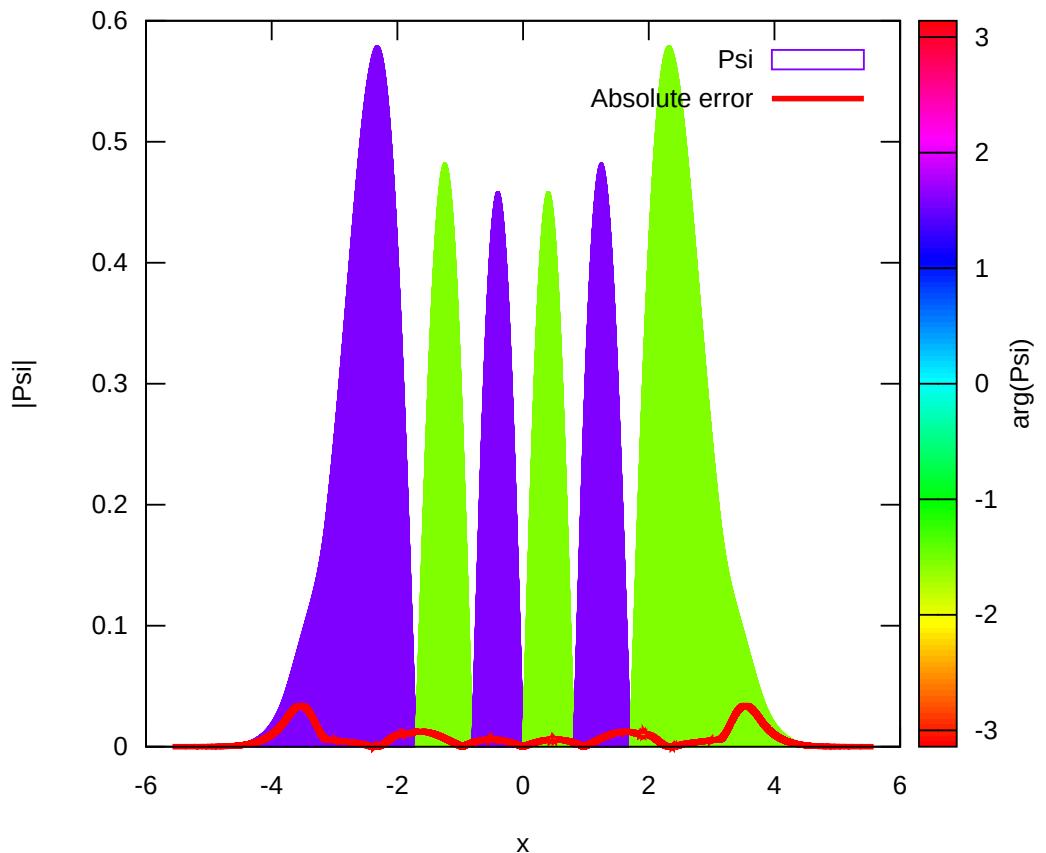


Figure 6.14.: The absolute from figure 6.13 is plotted with the approximate wave function.
As the red line indicates the error reaches a local maxima at the maxima of the
wave function itself.

7. Conclusion

In the end the program could be extended and improved almost indefinitely. Unfortunately I had to stop improving it at some point to actually write this paper about how the program works. Looking back at the goals, the program is quite fast for low and even “medium” energies while still being able to calculate solutions for complex potentials. The accuracy can be tuned a little bit with the constants, but they don’t offer a great flexibility.

One of the core design decisions was to follow the UNIX philosophy to some degree and personally I think this has been achieved with the flexible architecture. But as usual the UNIX philosophy has an impact on the user interface which is probably the worse part of the program. It is only accessible to programmers and can’t be used to its full potential when the user doesn’t know Rust, a systems programming language that is not very similar to popular programming languages like Python. If I have time I will definitely improve the user experience without taking away the freedom to edit the code if someone would like to.

I might also add a time dependent wave function that uses something simple like Euler’s method on the initial WKB solution. In general there are still many things for which there are already utilities inside the current version of the program. For example, there’s an implementation for multidimensional variant of Newton’s method that could be used to extend the program to more than one dimension.

In the end I gained a deeper understanding of this weird world we’re living in and the beauty of the things that can arise from complex mathematics.

A. Detailed Calculations and Tests

A.1. Test Machine Specs

This output was generated by `garuda-inxi`. “[...]” Means the data was removed because it is not necessary.

A.1.1. Machine 1

Machine 1 is the laptop of the author.

System:

```
Kernel: 6.0.9-zn1-1-zn arch: x86_64 bits: 64 compiler: gcc v: 12.2.0
parameters: BOOT_IMAGE=/@/boot/vmlinuz-linux-zn
root=UUID=308b5626-2597-4f46-9f16-a4d882bf2bf0 rw rootflags=subvol=@
quiet splash rd.udev.log_priority=3 vt.global_cursor_default=0
systemd.unified_cgroup_hierarchy=1 loglevel=3 ibt=off
Console: pty pts/0 DM: SDDM Distro: Garuda Linux base: Arch Linux
```

Machine:

```
Type: Laptop System: ASUSTeK product: ROG Zephyrus G14 GA401QM_GA401QM
v: 1.0 serial: <filter>
Mobo: ASUSTeK model: GA401QM v: 1.0 serial: <filter> UEFI: American
Megatrends LLC. v: GA401QM.412 date: 08/30/2022
```

[...]

CPU:

```
Info: model: AMD Ryzen 9 5900HS with Radeon Graphics socket: FP6 bits: 64
type: MT MCP arch: Zen 3 gen: 4 level: v3 note: check built: 2021-22
process: TSMC n7 (7nm) family: 0x19 (25) model-id: 0x50 (80) stepping: 0
microcode: 0xA50000C
```

```
Topology: cpus: 1x cores: 8 tpc: 2 threads: 16 smt: enabled cache:
L1: 512 KiB desc: d-8x32 KiB; i-8x32 KiB L2: 4 MiB desc: 8x512 KiB
L3: 16 MiB desc: 1x16 MiB
```

```
Speed (MHz): avg: 3300 min/max: 1200/4679 boost: enabled
base/boost: 3300/4650 scaling: driver: acpi-cpufreq governor: performance
volts: 1.2 V ext-clock: 100 MHz cores: 1: 3300 2: 3300 3: 3300 4: 3300
5: 3300 6: 3300 7: 3300 8: 3300 9: 3300 10: 3300 11: 3300 12: 3300
13: 3300 14: 3300 15: 3300 16: 3300 bogomips: 105400
```

```
Flags: avx avx2 ht lm nx pae sse sse2 sse3 sse4_1 sse4_2 sse4a ssse3 svm
Vulnerabilities:
Type: itlb_multihit status: Not affected
Type: l1tf status: Not affected
Type: mds status: Not affected
Type: meltdown status: Not affected
Type: mmio_stale_data status: Not affected
Type: retbleed status: Not affected
Type: spec_store_bypass mitigation: Speculative Store Bypass disabled via
    prctl
Type: spectre_v1 mitigation: usercopy/swapgs barriers and __user pointer
    sanitization
Type: spectre_v2 mitigation: Retpolines, IBPB: conditional, IBRS_FW,
    STIBP: always-on, RSB filling, PBRSB-eIBRS: Not affected
Type: srbds status: Not affected
Type: tsx_async_abort status: Not affected
```

[...]

Garuda (2.6.9-1):

```
System install date: 2022-01-06
Last full system update: 2022-11-19
Is partially upgraded: No
Relevant software: NetworkManager
Windows dual boot: Yes
Snapshots: Snapper
Failed units: shadow.service snapper-cleanup.service
```

A.1.2. Machine 2

Machine 2 is a VM running Ubuntu 20.04.1 LTS on the server of Fridolins Robotik (FRC team 6417).

System:

```
Kernel: 5.4.0-132-generic x86_64 bits: 64 compiler: gcc v: 9.4.0
parameters: BOOT_IMAGE=/vmlinuz-5.4.0-132-generic
root=/dev/mapper/ubuntu--vg-ubuntu--lv ro maybe-ubiquity
Console: tty 0 dm: N/A Distro: Ubuntu 20.04.1 LTS (Focal Fossa)
```

Machine:

```
Type: Kvm System: QEMU product: Standard PC (i440FX + PIIX, 1996)
v: pc-i440fx-6.1 serial: <filter> Chassis: type: 1 v: pc-i440fx-6.1
serial: <filter>
Mobo: N/A model: N/A serial: N/A BIOS: SeaBIOS
v: rel-1.14.0-0-g155821a1990b-prebuilt.qemu.org date: 04/01/2014
```

```

CPU:
Topology: 24-Core model: Common KVM bits: 64 type: MCP arch: K8
family: F (15) model-id: 6 stepping: 1 microcode: 1000065
L2 cache: 12.0 MiB
flags: lm nx pae sse sse2 sse3 bogomips: 138938
Speed: 2895 MHz min/max: N/A Core speeds (MHz): 1: 2895 2: 2895 3: 2895
4: 2895 5: 2895 6: 2895 7: 2895 8: 2895 9: 2895 10: 2895 11: 2895 12: 2895
13: 2895 14: 2895 15: 2895 16: 2895 17: 2895 18: 2895 19: 2895 20: 2895
21: 2895 22: 2895 23: 2895 24: 2895
Vulnerabilities: Type: itlb_multihit status: Not affected
Type: l1tf status: Not affected
Type: mds status: Not affected
Type: meltdown status: Not affected
Type: mmio_stale_data status: Not affected
Type: retbleed status: Not affected
Type: spec_store_bypass status: Not affected
Type: spectre_v1
mitigation: usercopy/swapgs barriers and __user pointer sanitization
Type: spectre_v2 mitigation: Retpolines, STIBP: disabled, RSB filling,
PBRSB-eIBRS: Not affected
Type: srbds status: Not affected
Type: tsx_async_abort status: Not affected

```

[...]

Info:

```

Processes: 288 Uptime: 1m Memory: 3.83 GiB used: 345.0 MiB (8.8%)
Init: systemd v: 245 runlevel: 5 Compilers: gcc: 9.4.0 alt: 9
Shell: garuda-inxi running in: tty 0 (SSH) inxi: 3.0.38

```

A.2. Benchmarks

A.2.1. Rust Benchmarks

These benchmarks were performed after a reboot of test machine 1 (specs A.1.1) in a TTY to minimize the CPU usage of other processes.

Output A.1: Test run on machine 1, the energy tests were ignored because they take about 2 hours to run.

```

1 test benchmarks::test::energy_bench_nenergy_1      ... ignored
2 test benchmarks::test::energy_bench_nenergy_2      ... ignored
3 test benchmarks::test::energy_bench_nenergy_3      ... ignored
4 test benchmarks::test::energy_bench_nenergy_4      ... ignored
5 test benchmarks::test::energy_bench_nenergy_5      ... ignored

```

```

6 test benchmarks::test::energy_bench_nenergy_6 ... ignored
7 test benchmarks::test::energy_bench_nenergy_7 ... ignored
8 test benchmarks::test::energy_bench_nenergy_8 ... ignored
9 test benchmarks::test::energy_bench_nenergy_9 ... ignored
10 test benchmarks::test::evaluate_bench_nenergy_1 ... bench: 11,106,464 ns/
    iter (+/- 1,746,488)
11 test benchmarks::test::evaluate_bench_nenergy_2 ... bench: 10,952,637 ns/
    iter (+/- 3,015,099)
12 test benchmarks::test::evaluate_bench_nenergy_3 ... bench: 10,779,924 ns/
    iter (+/- 1,115,300)
13 test benchmarks::test::evaluate_bench_nenergy_4 ... bench: 10,674,015 ns/
    iter (+/- 1,218,173)
14 test benchmarks::test::evaluate_bench_nenergy_5 ... bench: 10,597,315 ns/
    iter (+/- 1,565,714)
15 test benchmarks::test::evaluate_bench_nenergy_6 ... bench: 10,394,542 ns/
    iter (+/- 1,370,995)
16 test benchmarks::test::evaluate_bench_nenergy_7 ... bench: 10,459,327 ns/
    iter (+/- 3,031,317)
17 test benchmarks::test::evaluate_bench_nenergy_8 ... bench: 10,426,418 ns/
    iter (+/- 1,183,630)
18 test benchmarks::test::evaluate_bench_nenergy_9 ... bench: 10,168,316 ns/
    iter (+/- 1,286,419)
19 test turning_points::test::turning_point_square_nenergy_1 ... bench: 352,416 ns/
    iter (+/- 14,520)
20 test turning_points::test::turning_point_square_nenergy_2 ... bench: 345,360 ns/
    iter (+/- 18,186)
21 test turning_points::test::turning_point_square_nenergy_3 ... bench: 334,762 ns/
    iter (+/- 10,151)
22 test turning_points::test::turning_point_square_nenergy_4 ... bench: 320,085 ns/
    iter (+/- 17,957)
23 test turning_points::test::turning_point_square_nenergy_5 ... bench: 337,889 ns/
    iter (+/- 16,637)
24 test turning_points::test::turning_point_square_nenergy_6 ... bench: 319,445 ns/
    iter (+/- 140,482)
25 test turning_points::test::turning_point_square_nenergy_7 ... bench: 318,446 ns/
    iter (+/- 19,353)
26 test turning_points::test::turning_point_square_nenergy_8 ... bench: 321,418 ns/
    iter (+/- 135,708)
27 test turning_points::test::turning_point_square_nenergy_9 ... bench: 325,621 ns/
    iter (+/- 18,741)
28 test wave_function_builder::test::renormalize_square ... bench: 147,407 ns/
    iter (+/- 11,088)

```

Output A.2: Test on machine 2 (specs A.1.2).

```

1 test benchmarks::test::energy_bench_nenergy_1 ... ignored
2 test benchmarks::test::energy_bench_nenergy_2 ... ignored
3 test benchmarks::test::energy_bench_nenergy_3 ... ignored
4 test benchmarks::test::energy_bench_nenergy_4 ... ignored

```

```

5 test benchmarks::test::energy_bench_nenergy_5 ... ignored
6 test benchmarks::test::energy_bench_nenergy_6 ... ignored
7 test benchmarks::test::energy_bench_nenergy_7 ... ignored
8 test benchmarks::test::energy_bench_nenergy_8 ... ignored
9 test benchmarks::test::energy_bench_nenergy_9 ... ignored
10 test benchmarks::test::evaluate_bench_nenergy_1 ... bench: 25,078,868 ns/
    iter (+/- 7,078,947)
11 test benchmarks::test::evaluate_bench_nenergy_2 ... bench: 24,153,512 ns/
    iter (+/- 5,987,513)
12 test benchmarks::test::evaluate_bench_nenergy_3 ... bench: 24,043,451 ns/
    iter (+/- 6,440,041)
13 test benchmarks::test::evaluate_bench_nenergy_4 ... bench: 23,665,692 ns/
    iter (+/- 6,265,762)
14 test benchmarks::test::evaluate_bench_nenergy_5 ... bench: 22,671,743 ns/
    iter (+/- 5,090,286)
15 test benchmarks::test::evaluate_bench_nenergy_6 ... bench: 23,273,191 ns/
    iter (+/- 6,871,818)
16 test benchmarks::test::evaluate_bench_nenergy_7 ... bench: 23,143,284 ns/
    iter (+/- 6,225,635)
17 test benchmarks::test::evaluate_bench_nenergy_8 ... bench: 23,470,605 ns/
    iter (+/- 6,941,491)
18 test benchmarks::test::evaluate_bench_nenergy_9 ... bench: 22,564,435 ns/
    iter (+/- 5,548,813)
19 test turning_points::test::turning_point_square_nenergy_1 ... bench: 1,871,010 ns/
    iter (+/- 826,263)
20 test turning_points::test::turning_point_square_nenergy_2 ... bench: 1,928,734 ns/
    iter (+/- 893,799)
21 test turning_points::test::turning_point_square_nenergy_3 ... bench: 1,750,410 ns/
    iter (+/- 972,254)
22 test turning_points::test::turning_point_square_nenergy_4 ... bench: 1,826,680 ns/
    iter (+/- 986,256)
23 test turning_points::test::turning_point_square_nenergy_5 ... bench: 1,943,480 ns/
    iter (+/- 1,000,387)
24 test turning_points::test::turning_point_square_nenergy_6 ... bench: 1,887,249 ns/
    iter (+/- 806,322)
25 test turning_points::test::turning_point_square_nenergy_7 ... bench: 1,781,980 ns/
    iter (+/- 846,578)
26 test turning_points::test::turning_point_square_nenergy_8 ... bench: 1,957,382 ns/
    iter (+/- 908,014)
27 test turning_points::test::turning_point_square_nenergy_9 ... bench: 1,919,896 ns/
    iter (+/- 962,210)
28 test wave_function_builder::test::renormalize_square ... bench: 813,134 ns/
    iter (+/- 675,427)

```

Output A.3: Previously ignored energy tests on machine 2

```

1 running 9 tests
2 test benchmarks::test::energy_bench_nenergy_1 ... bench: 2,066,938,596 ns
    /iter (+/- 122,637,612)
3 test benchmarks::test::energy_bench_nenergy_2 ... bench: 2,087,782,039 ns

```

```

        /iter (+/- 135,082,403)
4 test benchmarks::test::energy_bench_nenergy_3           ... bench: 2,093,995,693 ns
      /iter (+/- 156,802,393)
5 test benchmarks::test::energy_bench_nenergy_4           ... bench: 2,069,512,511 ns
      /iter (+/- 102,294,553)
6 test benchmarks::test::energy_bench_nenergy_5           ... bench: 2,079,575,086 ns
      /iter (+/- 129,997,432)
7 test benchmarks::test::energy_bench_nenergy_6           ... bench: 2,091,588,104 ns
      /iter (+/- 134,889,997)
8 test benchmarks::test::energy_bench_nenergy_7           ... bench: 4,149,571,071 ns
      /iter (+/- 225,497,112)
9 test benchmarks::test::energy_bench_nenergy_8           ... bench: 4,175,954,724 ns
      /iter (+/- 201,706,134)
10 test benchmarks::test::energy_bench_nenergy_9          ... bench: 4,175,015,897 ns
      /iter (+/- 163,725,865)
11
12 test result: ok. 0 passed; 0 failed; 0 ignored; 9 measured; 35 filtered out; finished
               in 7521.30s

```

A.2.2. Perf

`perf stat -d -d -d -repeat 100` on super position of energies 1 through 9 of x^2 .

```

1 # started on Sat Nov 19 22:21:53 2022
2
3
4 Performance counter stats for './target/release/schroedinger_approx' (100 runs):
5
6     2,446,386.86 msec task-clock                      # 15.938 CPUs utilized
     ( +- 0.07% )
7       769,294    context-switches                   # 314.455 /sec
     ( +- 0.73% )
8       18,443    cpu-migrations                     # 7.539 /sec
     ( +- 0.92% )
9       44,453    page-faults                       # 18.171 /sec
     ( +- 1.21% )
10      7,820,854,372,190   cycles                  # 3.197 GHz
     ( +- 0.05% ) (40.00%)
11      8,485,422,199   stalled-cycles-frontend    # 0.11% frontend cycles
     idle ( +- 28.66% ) (40.00%)
12      9,501,600,446   stalled-cycles-backend    # 0.12% backend cycles
     idle ( +- 1.14% ) (40.00%)
13     14,477,558,852,335   instructions          # 1.85 insn per cycle
14                               # 0.00 stalled cycles per insn
     ( +- 0.01% ) (40.00%)
15     1,384,019,071,011   branches                # 565.729 M/sec
     ( +- 0.02% ) (40.00%)
16     818,139,391    branch-misses            # 0.06% of all branches
     ( +- 0.45% ) (40.00%)

```

```

17 5,582,382,628,800    L1-dcache-loads          # 2.282 G/sec
18 115,562,637,690     L1-dcache-load-misses   # 2.07% of all L1-dcache
19      accesses ( +- 0.17% ) (40.00%)
20      <not supported> LLC-loads
21      <not supported> LLC-load-misses
22      12,523,189,958    L1-icache-loads        # 5.119 M/sec
23          ( +- 0.48% ) (40.00%)
24      80,902,476       L1-icache-load-misses   # 0.64% of all L1-icache
25          accesses ( +- 0.61% ) (40.00%)
26      2,380,836,711    dTLB-loads           # 973.186 K/sec
27          ( +- 1.10% ) (40.00%)
28      53,829,893       dTLB-load-misses      # 2.37% of all dTLB cache
29          accesses ( +- 1.68% ) (40.00%)
30      34,803,159       iTLB-loads           # 14.226 K/sec
31          ( +- 2.36% ) (40.00%)
32      17,518,263       iTLB-load-misses      # 39.70% of all iTLB cache
33          accesses ( +- 1.86% ) (40.00%)
34      106,080,681,746   L1-dcache-prefetches   # 43.361 M/sec
35          ( +- 0.00% ) (40.00%)
36      <not supported> L1-dcache-prefetch-misses
37
38      153.498 +- 0.128 seconds time elapsed ( +- 0.08% )

```

A.3. Additional Plots

A.3.1. Mexican Hat

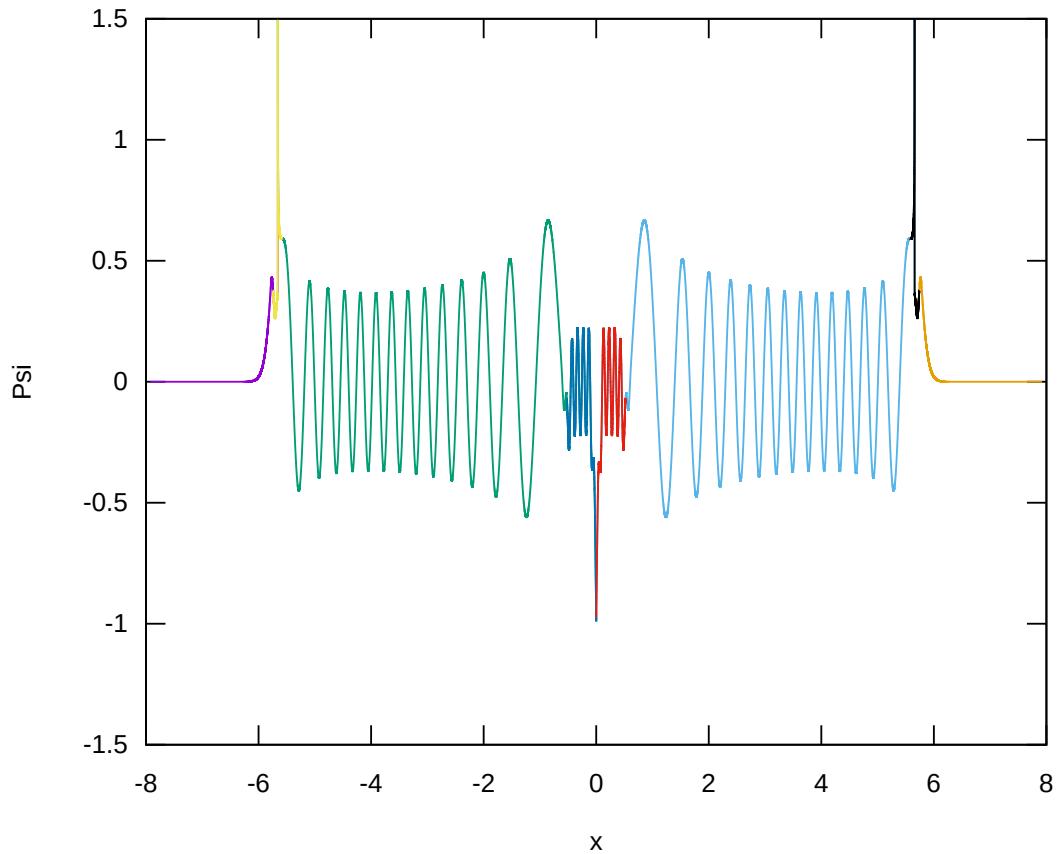


Figure A.1.: Wave function of mexican hat potential, with the 54th energy and $m = 1$. The program could not find all the turning points, that's why there are two asymptotes at the turning points. Around $x = 0$ it oscillates with a high frequency even though a small frequency was expected.

A.4. Proofs

A.4.1. Smoothness of Transitionfunction

Given that

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

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

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

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

define

(Hall, 2013)

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

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

and proof that

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

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

Calculate derivatives

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

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

Note that

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

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

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

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

therefor

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

and

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

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

A.5. Validity of 0 Wave Function

$$\Psi(x) = 0 \quad (\text{A.18})$$

$$\frac{1}{2m} \frac{d^2}{dx^2} \Psi(x) + V(x)\Psi(x) = E\Psi(x) \quad (\text{A.19})$$

$$\frac{1}{2m} \cdot 0 + V(x) \cdot 0 = E \cdot 0 \quad (\text{A.20})$$

$$0 = 0 \blacksquare \quad (\text{A.21})$$

Therefore $\Psi(x) = 0$ is a solution for all energies and all potentials.

A.6. Branch Elimination

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

Rust Code

```
const COND: bool = true;

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

Assembly

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

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

Rust Code

```
const COND: bool = false;

pub fn test(x: f64) -> f64 {
```

Assembly

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

```
if x > 5.0 && COND {  
    return x % 5.0;  
} else {  
    return x*x;  
}  
}
```

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

B. Data Files

B.1. Energies

energies_approx.dat

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

energies_exact.dat

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

38	54.4472570128045	38	54.44722215136415
39	55.8613416078763	39	55.86143571373725
40	57.27558245538954	40	57.27564927611034
41	58.68966705046134	41	58.68986283848344
42	60.10453291262317	42	60.104076400856535
43	61.518148750370635	43	61.51828996322963
44	62.93270210276677	44	62.932503525602726
45	64.3472554551629	45	64.34671708797582
46	65.76149630267614	46	65.76093065034891
47	67.17542464530649	47	67.175144212722
48	68.58935298793685	48	68.58935777509511
49	70.00343758789145	49	70.0035713374682
50	71.41783468784614	50	71.4177848998413

C. Source Code

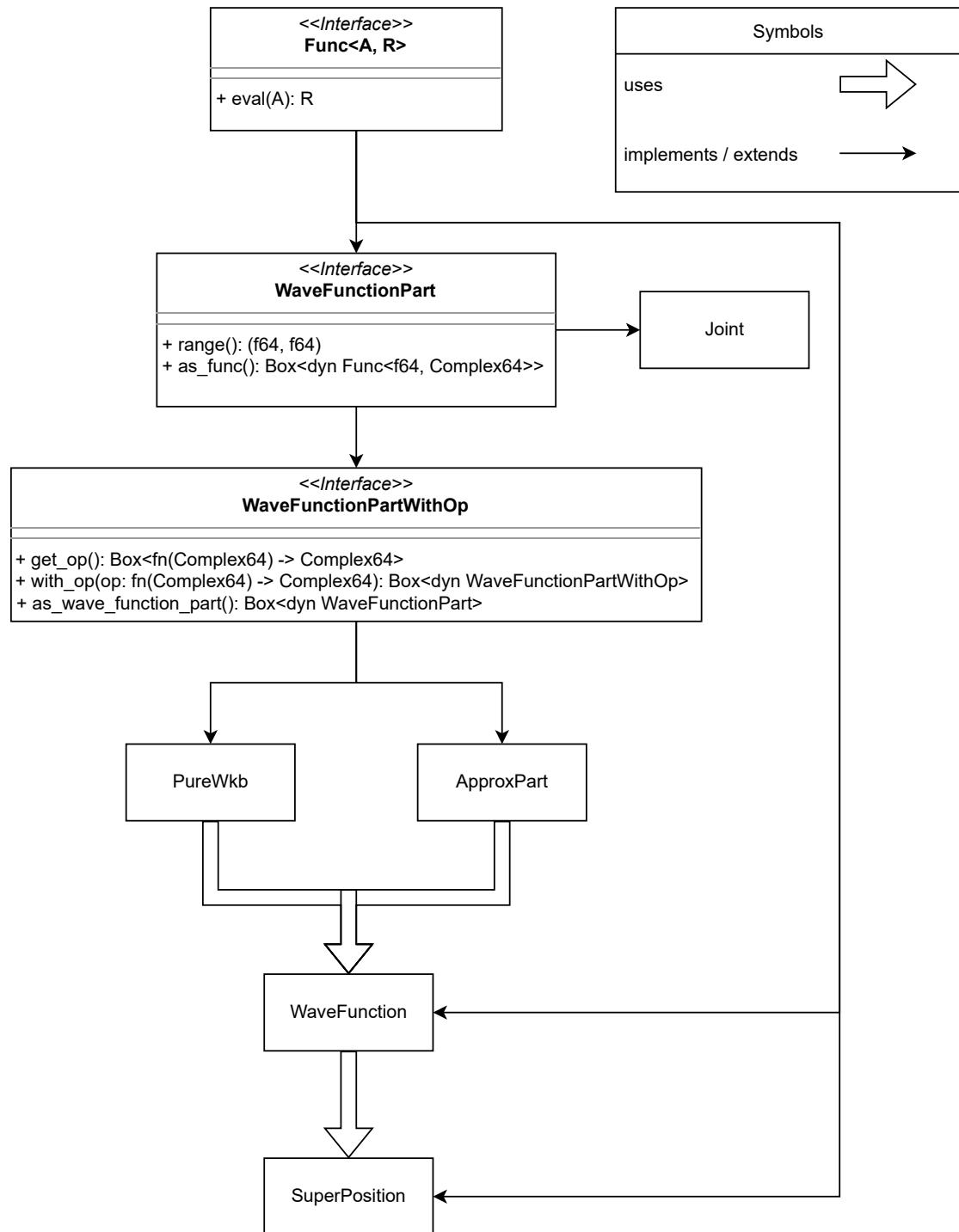


Figure C.1.: UML diagram of program architecture

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

src/main.rs

```
1 #![allow(dead_code)]
2 #![feature(test)]
3
4 mod airy;
5 mod airy_wave_func;
6 mod benchmarks;
7 mod check;
8 mod energy;
9 mod integrals;
10 mod newtons_method;
11 mod plot;
12 mod potentials;
13 mod tui;
14 mod turning_points;
15 mod utils;
16 mod wave_function_builder;
17 mod wkb_wave_func;
18
19 use crate::airy::airy_ai;
20 use crate::airy_wave_func::AiryWaveFunction;
21 use crate::integrals::*;

22 use crate::newtons_method::derivative;
23 use crate::utils::Func;
24 use crate::utils::*;

25 use crate::wave_function_builder::*;

26 use crate::wkb_wave_func::WkbWaveFunction;
27 use num::complex::Complex64;
28 use num::pow::Pow;
29 use rayon::iter::*;

30 use std::f64;
31 use std::fs::File;
32 use std::io::Write;
33 use std::path::Path;
34 use std::sync::Arc;
35
36 const INTEG_STEPS: usize = 64000;
37 const TRAPEZE_PER_THREAD: usize = 1000;
38 const NUMBER_OF_POINTS: usize = 100000;
39
40 const AIRY_TRANSITION_FRACTION: f64 = 0.5;
41 const ENABLE_AIRY_JOINTS: bool = true;
42
43 const VALIDITY_LL_FACTOR: f64 = 3.5;
```

```

44
45 const APPROX_INF: (f64, f64) = (-200.0, 200.0);
46 const VIEW_FACTOR: f64 = 0.5;
47
48 fn main() {
49     let wave_function = wave_function_builder::WaveFunction::new(
50         &potentials::mexican_hat,
51         1.0, // mass
52         5, // nth energy
53         APPROX_INF,
54         VIEW_FACTOR,
55         ScalingType::Renormalize(1.0.into()),
56     );
57
58     // let wave_function = wave_function_builder::SuperPosition::new(
59     //     &potentials::square,
60     //     1.0, // mass
61     //     &[
62     //         (1, complex(1.0, 0.0)), // (nth energy, phase)
63     //         (2, complex(2.0, 0.0)), // (nth energy, phase)
64     //         (3, complex(3.0, 0.0)), // (nth energy, phase)
65     //         (4, complex(4.0, 0.0)), // (nth energy, phase)
66     //         (5, complex(5.0, 0.0)), // (nth energy, phase)
67     //         (6, complex(6.0, 0.0)), // (nth energy, phase)
68     //         (7, complex(7.0, 0.0)), // (nth energy, phase)
69     //         (8, complex(8.0, 0.0)), // (nth energy, phase)
70     //         (9, complex(9.0, 0.0)), // (nth energy, phase)
71     //     ],
72     //     APPROX_INF,
73     //     VIEW_FACTOR,
74     //     ScalingType::Renormalize(complex(1.0, 0.0)),
75     // );
76
77     let output_dir = Path::new("output");
78
79     // For WaveFunction
80     // plot::plot_wavefunction(&wave_function, output_dir, "data.txt");
81     plot::plot_wavefunction_parts(&wave_function, output_dir, "data.txt");
82     // plot::plot_probability(&wave_function, output_dir, "data.txt");
83
84     // For SuperPosition
85     // plot::plot_superposition(&wave_function, output_dir, "data.txt");
86     // plot::plot_probability_superposition(&wave_function, output_dir, "data.txt");
87 }

```

src/airy.rs

```

1 /* automatically generated by rust-bindgen 0.59.2 */
2 #![allow(non_snake_case)]

```

```

3 #[allow(deref_nullptr)]
4 #[allow(non_camel_case_types)]
5
6 #[derive(PartialEq, Copy, Clone, Hash, Debug, Default)]
7 #[repr(C)]
8 pub struct __BindgenComplex<T> {
9     pub re: T,
10    pub im: T,
11 }
12 pub type size_t = ::std::os::raw::c_ulong;
13 pub type wchar_t = ::std::os::raw::c_int;
14 #[repr(C)]
15 #[repr(align(16))]
16 #[derive(Debug, Copy, Clone)]
17 pub struct max_align_t {
18     pub __clang_max_align_nonce1: ::std::os::raw::c_longlong,
19     pub __bindgen_padding_0: u64,
20     pub __clang_max_align_nonce2: u128,
21 }
22 #[test]
23 fn bindgen_test_layout_max_align_t() {
24     assert_eq!(
25         ::std::mem::size_of::<max_align_t>(),
26         32usize,
27         concat!("Size_of:", stringify!(max_align_t))
28     );
29     assert_eq!(
30         ::std::mem::align_of::<max_align_t>(),
31         16usize,
32         concat!("Alignment_of:", stringify!(max_align_t))
33     );
34     assert_eq!(
35         unsafe {
36             &(*(::std::ptr::null::<max_align_t>())).__clang_max_align_nonce1 as *
37             const _ as usize
38         },
39         0usize,
40         concat!(
41             "Offset_of_field:",
42             stringify!(max_align_t),
43             "::",
44             stringify!(__clang_max_align_nonce1)
45         )
46     );
47     assert_eq!(
48         unsafe {
49             &(*(::std::ptr::null::<max_align_t>())).__clang_max_align_nonce2 as *
50             const _ as usize
51         },
52         0usize,
53         concat!(
54             "Offset_of_field:",
55             stringify!(max_align_t),
56             "::",
57             stringify!(__clang_max_align_nonce2)
58         )
59     );
60 }

```

```

50     16usize,
51     concat!(
52         "Offset_of_field:",
53         stringify!(max_align_t),
54         "::",
55         stringify!(_clang_max_align_nonce2)
56     )
57 );
58 }
59 #[repr(C)]
60 #[derive(Debug, Copy, Clone)]
61 pub struct _GoString_ {
62     pub p: *const ::std::os::raw::c_char,
63     pub n: isize,
64 }
65 #[test]
66 fn bindgen_test_layout__GoString_() {
67     assert_eq!(
68         ::std::mem::size_of::<_GoString_>(),
69         16usize,
70         concat!("Size_of:", stringify!(_GoString_))
71     );
72     assert_eq!(
73         ::std::mem::align_of::<_GoString_>(),
74         8usize,
75         concat!("Alignment_of:", stringify!(_GoString_))
76     );
77     assert_eq!(
78         unsafe { &(*(::std::ptr::null::<_GoString_>())).p as *const _ as usize },
79         0usize,
80         concat!(
81             "Offset_of_field:",
82             stringify!(_GoString_),
83             "::",
84             stringify!(p)
85         )
86     );
87     assert_eq!(
88         unsafe { &(*(::std::ptr::null::<_GoString_>())).n as *const _ as usize },
89         8usize,
90         concat!(
91             "Offset_of_field:",
92             stringify!(_GoString_),
93             "::",
94             stringify!(n)
95         )
96     );
97 }
98 pub type GoInt8 = ::std::os::raw::c_schar;

```

```

99 pub type GoUint8 = ::std::os::raw::c_uchar;
100 pub type GoInt16 = ::std::os::raw::c_short;
101 pub type GoUint16 = ::std::os::raw::c_ushort;
102 pub type GoInt32 = ::std::os::raw::c_int;
103 pub type GoUint32 = ::std::os::raw::c_uint;
104 pub type GoInt64 = ::std::os::raw::c_longlong;
105 pub type GoUint64 = ::std::os::raw::c_ulonglong;
106 pub type GoInt = GoInt64;
107 pub type GoUint = GoUint64;
108 pub type GoUintptr = ::std::os::raw::c_ulong;
109 pub type GoFloat32 = f32;
110 pub type GoFloat64 = f64;
111 pub type GoComplex64 = __BindgenComplex<f32>;
112 pub type GoComplex128 = __BindgenComplex<f64>;
113 pub type _check_for_64_bit_pointer_matching_GoInt = [::std::os::raw::c_char; 1usize];
114 pub type GoString = _GoString_;
115 pub type GoMap = *mut ::std::os::raw::c_void;
116 pub type GoChan = *mut ::std::os::raw::c_void;
117 #[repr(C)]
118 #[derive(Debug, Copy, Clone)]
119 pub struct GoInterface {
120     pub t: *mut ::std::os::raw::c_void,
121     pub v: *mut ::std::os::raw::c_void,
122 }
123 #[test]
124 fn bindgen_test_layout_GoInterface() {
125     assert_eq!(
126         ::std::mem::size_of::<GoInterface>(),
127         16usize,
128         concat!("Size_of_", stringify!(GoInterface))
129     );
130     assert_eq!(
131         ::std::mem::align_of::<GoInterface>(),
132         8usize,
133         concat!("Alignment_of_", stringify!(GoInterface))
134     );
135     assert_eq!(
136         unsafe { &(*(::std::ptr::null::<GoInterface>())).t as *const _ as usize },
137         0usize,
138         concat!(
139             "Offset_of_field:_",
140             stringify!(GoInterface),
141             "::",
142             stringify!(t)
143         )
144     );
145     assert_eq!(
146         unsafe { &(*(::std::ptr::null::<GoInterface>())).v as *const _ as usize },
147         8usize,

```

```

148     concat!(
149         "Offset_of_field:",
150         stringify!(GoInterface),
151         "::",
152         stringify!(v)
153     )
154 );
155 }
156 #[repr(C)]
157 #[derive(Debug, Copy, Clone)]
158 pub struct GoSlice {
159     pub data: *mut ::std::os::raw::c_void,
160     pub len: GoInt,
161     pub cap: GoInt,
162 }
163 #[test]
164 fn bindgen_test_layout_GoSlice() {
165     assert_eq!(
166         ::std::mem::size_of::<GoSlice>(),
167         24usize,
168         concat!("Size_of:", stringify!(GoSlice))
169     );
170     assert_eq!(
171         ::std::mem::align_of::<GoSlice>(),
172         8usize,
173         concat!("Alignment_of:", stringify!(GoSlice))
174     );
175     assert_eq!(
176         unsafe { &(*(::std::ptr::null::<GoSlice>())).data as *const _ as usize },
177         0usize,
178         concat!(
179             "Offset_of_field:",
180             stringify!(GoSlice),
181             "::",
182             stringify!(data)
183         )
184     );
185     assert_eq!(
186         unsafe { &(*(::std::ptr::null::<GoSlice>())).len as *const _ as usize },
187         8usize,
188         concat!(
189             "Offset_of_field:",
190             stringify!(GoSlice),
191             "::",
192             stringify!(len)
193         )
194     );
195     assert_eq!(
196         unsafe { &(*(::std::ptr::null::<GoSlice>())).cap as *const _ as usize },

```

```

197     16usize,
198     concat!(
199         "Offset_of_field:",
200         stringify!(GoSlice),
201         "::",
202         stringify!(cap)
203     )
204 );
205 }
206 #[repr(C)]
207 #[derive(Debug, Copy, Clone)]
208 pub struct airy_ai_return {
209     pub r0: GoFloat64,
210     pub r1: GoFloat64,
211 }
212 #[test]
213 fn bindgen_test_layout_airy_ai_return() {
214     assert_eq!(
215         ::std::mem::size_of::<airy_ai_return>(),
216         16usize,
217         concat!("Size_of:", stringify!(airy_ai_return))
218     );
219     assert_eq!(
220         ::std::mem::align_of::<airy_ai_return>(),
221         8usize,
222         concat!("Alignment_of:", stringify!(airy_ai_return))
223     );
224     assert_eq!(
225         unsafe { &(*(::std::ptr::null::<airy_ai_return>())).r0 as *const _ as usize
226             },
227         0usize,
228         concat!(
229             "Offset_of_field:",
230             stringify!(airy_ai_return),
231             "::",
232             stringify!(r0)
233         )
234     );
235     assert_eq!(
236         unsafe { &(*(::std::ptr::null::<airy_ai_return>())).r1 as *const _ as usize
237             },
238         8usize,
239         concat!(
240             "Offset_of_field:",
241             stringify!(airy_ai_return),
242             "::",
243             stringify!(r1)
244         )
245     );
}

```

```

244 }
245 extern "C" {
246     pub fn airy_ai(zr: GoFloat64, zi: GoFloat64) -> airy_ai_return;
247 }
```

src/airy_wave_func.rs

```

1 use crate::newtons_method::*;
2 use crate::turning_points::*;
3 use crate::wkb_wave_func::Phase;
4 use crate::*;

5 use num::signum;
6 use std::sync::Arc;
7
8 #[allow(non_snake_case)]
9 fn Ai(x: Complex64) -> Complex64 {
10     let go_return;
11     unsafe {
12         go_return = airy_ai(x.re, x.im);
13     }
14     return complex(go_return.r0, go_return.r1);
15 }
16
17 #[allow(non_snake_case)]
18 fn Bi(x: Complex64) -> Complex64 {
19     return -complex(0.0, 1.0) * Ai(x)
20         + 2.0 * Ai(x * complex(-0.5, 3.0_f64.sqrt() / 2.0)) * complex(3_f64.sqrt() /
2.0, 0.5);
21 }
22
23 #[derive(Clone)]
24 pub struct AiryWaveFunction {
25     c: Complex64,
26     u_1: f64,
27     pub turning_point: f64,
28     phase: Arc<Phase>,
29     pub ts: (f64, f64),
30     op: fn(Complex64) -> Complex64,
31     phase_off: f64,
32 }
33
34 impl AiryWaveFunction {
35     pub fn get_op(&self) -> Box<fn(Complex64) -> Complex64> {
36         Box::new(self.op)
37     }
38
39     fn get_u_1_cube_root(u_1: f64) -> f64 {
40         signum(u_1) * u_1.abs().pow(1.0 / 3.0)
41     }

```

```

42
43     pub fn new<'a>(phase: Arc<Phase>, view: (f64, f64)) -> (Vec<AiryWaveFunction>,
44         TGroup) {
45         let phase = phase;
46         let turning_point_boundaries = turning_points::calc_ts(phase.as_ref(), view);
47
48         let funcs: Vec<AiryWaveFunction> = turning_point_boundaries
49             .ts
50             .iter()
51             .map(|((tb1, tb2), t)| {
52                 let u_1 = 2.0 * phase.mass * -derivative(phase.potential.as_ref(), *t
53
54                     );
55
56                     AiryWaveFunction {
57                         u_1,
58                         turning_point: *t,
59                         phase: phase.clone(),
60                         ts: (*tb1, *tb2),
61                         op: identity,
62                         c: 1.0.into(),
63                         phase_off: 0.0,
64                     }
65
66             })
67             .collect::<Vec<AiryWaveFunction>>();
68         return (funcs, turning_point_boundaries);
69     }
70
71     pub fn with_op(&self, op: fn(Complex64) -> Complex64) -> AiryWaveFunction {
72         AiryWaveFunction {
73             u_1: self.u_1,
74             turning_point: self.turning_point,
75             phase: self.phase.clone(),
76             ts: self.ts,
77             op,
78             c: self.c,
79             phase_off: self.phase_off,
80         }
81     }
82
83     pub fn with_c(&self, c: Complex64) -> AiryWaveFunction {
84         AiryWaveFunction {
85             u_1: self.u_1,
86             turning_point: self.turning_point,
87             phase: self.phase.clone(),
88             ts: self.ts,
89             op: self.op,
90             c,
91             phase_off: self.phase_off,
92         }
93     }

```

```

89     }
90
91     pub fn with_phase_off(&self, phase_off: f64) -> AiryWaveFunction {
92         AiryWaveFunction {
93             u_1: self.u_1,
94             turning_point: self.turning_point,
95             phase: self.phase.clone(),
96             ts: self.ts,
97             op: self.op,
98             c: self.c,
99             phase_off,
100        }
101    }
102 }
103
104 impl Func<f64, Complex64> for AiryWaveFunction {
105     fn eval(&self, x: f64) -> Complex64 {
106         let u_1_cube_root = Self::get_u_1_cube_root(self.u_1);
107
108         let value = self.c
109             * ((std::f64::consts::PI.sqrt() / (self.u_1).abs().pow(1.0 / 6.0))
110                 * Ai(complex(u_1_cube_root * (self.turning_point - x), 0.0)))
111                 as Complex64;
112         return (self.op)(value);
113     }
114 }
115
116 #[cfg(test)]
117 mod test {
118     use super::*;

119     #[test]
120     fn airy_func_plot() {
121         let output_dir = Path::new("output");
122         std::env::set_current_dir(&output_dir).unwrap();
123
124         let airy_ai = Function::new(|x| Ai(complex(x, 0.0)));
125         let airy_bi = Function::new(|x| Bi(complex(x, 0.0)));
126         let values = evaluate_function_between(&airy_ai, -10.0, 5.0, NUMBER_OF_POINTS
127             );
128
129         let mut data_file = File::create("airy.txt").unwrap();
130
131         let data_str_ai: String = values
132             .par_iter()
133             .map(|p| -> String { format!("{} {} {}\n", p.x, p.y.re, p.y.im) })
134             .reduce(|| String::new(), |s: String, current: String| s + &*current);
135
136         let values_bi = evaluate_function_between(&airy_bi, -5.0, 2.0,

```

```

        NUMBER_OF_POINTS);

137     let data_str_bi: String = values_bi
138         .par_iter()
139         .map(|p| -> String { format!("{}{}{}\n", p.x, p.y.re, p.y.im) })
140         .reduce(|| String::new(), |s: String, current: String| s + &*current);
141
142     data_file
143         .write_all((data_str_ai + "\n\n" + &*data_str_bi).as_ref())
144         .unwrap()
145     }
146 }
147 }
```

src/check.rs

```

1 use crate::*;

2
3 pub struct SchroedingerError<'a> {
4     pub wave_func: &'a WaveFunction,
5 }
6
7 impl Func<f64, Complex64> for SchroedingerError<'_> {
8     fn eval(&self, x: f64) -> Complex64 {
9         complex(-1.0 / (2.0 * self.wave_func.get_phase().mass), 0.0)
10        * Derivative {
11            f: &Derivative { f: self.wave_func },
12        }
13        .eval(x)
14        + ((self.wave_func.get_phase().potential)(x) - self.wave_func.get_phase()
15            .energy)
16        * self.wave_func.eval(x)
17    }
18 }
```

src/energy.rs

```

1 use crate::*;

2
3 struct Integrand<'a, F: Fn(f64) -> f64 + Sync> {
4     mass: f64,
5     pot: &'a F,
6     energy: f64,
7 }
8
9 impl<F: Fn(f64) -> f64 + Sync> Func<f64, f64> for Integrand<'_, F> {
10    fn eval(&self, x: f64) -> f64 {
11        let pot = (self.pot)(x);
12
13        if !pot.is_finite() {
```

```

14         return 0.0;
15     }
16
17     if pot < self.energy {
18         return (2.0 * self.mass * (self.energy - pot)).sqrt();
19     } else {
20         return 0.0;
21     }
22 }
23 }
24
25 struct SommerfeldCond<'a, F: Fn(f64) -> f64 + Sync> {
26     mass: f64,
27     pot: &'a F,
28     view: (f64, f64),
29 }
30
31 impl<F: Fn(f64) -> f64 + Sync> Func<f64, f64> for SommerfeldCond<'_, F> {
32     fn eval(&self, energy: f64) -> f64 {
33         let integrand = Integrand {
34             mass: self.mass,
35             pot: self.pot,
36             energy,
37         };
38         let integral = integrate(
39             evaluate_function_between(&integrand, self.view.0, self.view.1,
40                 INTEG_STEPS),
41             TRAPEZE_PER_THREAD,
42         );
43         return ((2.0 * integral - f64::consts::PI) / f64::consts::TAU) % 1.0;
44     }
45 }
46 pub fn nth_energy<F: Fn(f64) -> f64 + Sync>(n: usize, mass: f64, pot: &F, view: (f64,
47     f64)) -> f64 {
48     const ENERGY_STEP: f64 = 10.0;
49     const CHECKS_PER_ENERGY_STEP: usize = INTEG_STEPS;
50     let sommerfeld_cond = SommerfeldCond { mass, pot, view };
51
52     let mut energy = 0.0; // newtons_method_non_smooth(&|e| sommerfeld_cond.eval(e),
53     1e-7, 1e-7);
54     let mut i = 0;
55
56     loop {
57         let vals = evaluate_function_between(
58             &sommerfeld_cond,
59             energy,
60             energy + ENERGY_STEP,
61             CHECKS_PER_ENERGY_STEP,

```

```

60     );
61     let mut int_solutions = vals
62         .iter()
63         .zip(vals.iter().skip(1))
64         .collect:::<Vec<(&Point<f64, f64>, &Point<f64, f64>)>>>()
65         .par_iter()
66         .filter(|(p1, p2)| (p1.y - p2.y).abs() > 0.5 || p1.y.signum() != p2.y.
67             signum())
68         .map(|ps| ps.1)
69         .collect:::<Vec<&Point<f64, f64>>>();
70     int_solutions.sort_by(|p1, p2| cmp_f64(&p1.x, &p2.x));
71     if i + int_solutions.len() > n {
72         return int_solutions[n - i].x;
73     }
74     energy += ENERGY_STEP - (ENERGY_STEP / (CHECKS_PER_ENERGY_STEP as f64 + 1.0))
75     ;
76     i += int_solutions.len();
77 }

```

src/integrals.rs

```

1 use crate::*;

2 use rayon::prelude::*;

3

4 #[allow(non_camel_case_types)]
5 #[derive(Clone)]
6 pub struct Point<T_X, T_Y> {
7     pub x: T_X,
8     pub y: T_Y,
9 }
10

11 pub fn trapezoidal_approx<X, Y>(start: &Point<X, Y>, end: &Point<X, Y>) -> Y
12 where
13     X: std::ops::Sub<Output = X> + Copy,
14     Y: std::ops::Add<Output = Y>
15         + std::ops::Mul<Output = Y>
16         + std::ops::Div<f64, Output = Y>
17         + Copy
18         + From<X>,
19 {
20     return Y::from(end.x - start.x) * (start.y + end.y) / 2.0_f64;
21 }
22

23 pub fn index_to_range<T>(x: T, in_min: T, in_max: T, out_min: T, out_max: T) -> T
24 where
25     T: Copy
26         + std::ops::Sub<Output = T>
27         + std::ops::Mul<Output = T>

```

```

28     + std::ops::Div<Output = T>
29     + std::ops::Add<Output = T>,
30 {
31     return (x - in_min) * (out_max - out_min) / (in_max - in_min) + out_min;
32 }
33
34 pub fn integrate<
35     X: Sync + std::ops::Add<Output = X> + std::ops::Sub<Output = X> + Copy,
36     Y: Default
37     + Sync
38     + std::ops::AddAssign
39     + std::ops::Div<f64, Output = Y>
40     + std::ops::Mul<Output = Y>
41     + std::ops::Add<Output = Y>
42     + Send
43     + std::iter::Sum<Y>
44     + Copy
45     + From<X>,
46 >(
47     points: Vec<Point<X, Y>>,
48     batch_size: usize,
49 ) -> Y {
50     if points.len() < 2 {
51         return Y::default();
52     }
53
54     let batches: Vec<&[Point<X, Y>]> = points.chunks(batch_size).collect();
55
56     let parallel: Y = batches
57         .par_iter()
58         .map(|batch| {
59             let mut sum = Y::default();
60             for i in 0..(batch.len() - 1) {
61                 sum += trapezoidal_approx(&batch[i], &batch[i + 1]);
62             }
63             return sum;
64         })
65         .sum();
66
67     let mut rest = Y::default();
68
69     for i in 0..batches.len() - 1 {
70         rest += trapezoidal_approx(&batches[i][batches[i].len() - 1], &batches[i +
71             1][0]);
72     }
73
74     return parallel + rest;
75 }
```

```

76 pub fn evaluate_function_between<X, Y>(f: &dyn Func<X, Y>, a: X, b: X, n: usize) ->
77     Vec<Point<X, Y>>
78 where
79     X: Copy
80         + Send
81         + Sync
82         + std::cmp::PartialEq
83         + From<f64>
84         + std::ops::Add<Output = X>
85         + std::ops::Sub<Output = X>
86         + std::ops::Mul<Output = X>
87         + std::ops::Div<Output = X>,
88     Y: Send + Sync,
89 {
90     if a == b {
91         return vec![];
92     }
93     (0..n)
94         .into_par_iter()
95         .map(|i| {
96             index_to_range(
97                 X::from(i as f64),
98                 X::from(0.0_f64),
99                 X::from((n - 1) as f64),
100                a,
101                b,
102            )
103        })
104        .map(|x: X| Point { x, y: f.eval(x) })
105        .collect()
106    }
107 #[cfg(test)]
108 mod test {
109     use super::*;

110     fn square(x: f64) -> Complex64 {
111         return complex(x * x, 0.0);
112     }

113     fn square_integral(a: f64, b: f64) -> Complex64 {
114         return complex(b * b * b / 3.0 - a * a * a / 3.0, 0.0);
115     }

116     fn integral_of_square() {
117         let square_func: Function<f64, Complex64> = Function::new(square);
118         for i in 0..100 {
119             for j in 0..10 {

```

```

124     let a = f64::from(i - 50) / 12.3;
125     let b = f64::from(j - 50) / 12.3;
126
127     if i == j {
128         assert_eq!(
129             integrate(
130                 evaluate_function_between(&square_func, a, b, INTEG_STEPS
131                     ),
132                     TRAPEZE_PER_THREAD,
133                     ),
134                     complex(0.0, 0.0)
135                     );
136         continue;
137     }
138
139     let epsilon = 0.00001;
140     assert!(complex_compare(
141         integrate(
142             evaluate_function_between(&square_func, a, b, INTEG_STEPS),
143             TRAPEZE_PER_THREAD,
144             ),
145             square_integral(a, b),
146             epsilon,
147             ));
148
149 }
150
151 #[test]
152 fn evaluate_square_func_between() {
153     let square_func: Function<f64, Complex64> = Function::new(square);
154     let actual = evaluate_function_between(&square_func, -2.0, 2.0, 5);
155     let expected = vec![
156         Point {
157             x: -2.0,
158             y: complex(4.0, 0.0),
159         },
160         Point {
161             x: -1.0,
162             y: complex(1.0, 0.0),
163         },
164         Point {
165             x: 0.0,
166             y: complex(0.0, 0.0),
167         },
168         Point {
169             x: 1.0,
170             y: complex(1.0, 0.0),
171         },

```

```

172     Point {
173         x: 2.0,
174         y: complex(4.0, 0.0),
175     },
176 ];
177
178     for (a, e) in actual.iter().zip(expected) {
179         assert_eq!(a.x, e.x);
180         assert_eq!(a.y, e.y);
181     }
182 }
183
184 fn sinusoidal_exp_complex(x: f64) -> Complex64 {
185     return complex(x, x).exp();
186 }
187
188 fn sinusoidal_exp_complex_integral(a: f64, b: f64) -> Complex64 {
189     // (-1/2 + i/2) (e^((1 + i) a) - e^((1 + i) b))
190     return complex(-0.5, 0.5) * (complex(a, a).exp() - complex(b, b).exp());
191 }
192
193 fn integral_of_sinusoidal_exp() {
194     let sinusoidal_exp_complex: Function<f64, Complex64> =
195         Function::new(sinusoidal_exp_complex);
196     for i in 0..10 {
197         for j in 0..10 {
198             let a = f64::from(i - 50) / 12.3;
199             let b = f64::from(j - 50) / 12.3;
200
201             if i == j {
202                 assert_eq!(
203                     integrate(
204                         evaluate_function_between(&sinusoidal_exp_complex, a, b,
205                             INTEG_STEPS),
206                         TRAPEZE_PER_THREAD,
207                         ),
208                         complex(0.0, 0.0)
209                     );
210                 continue;
211             }
212             let epsilon = 0.0001;
213             assert!(complex_compare(
214                 integrate(
215                     evaluate_function_between(&sinusoidal_exp_complex, a, b,
216                         INTEG_STEPS),
217                         TRAPEZE_PER_THREAD,
218                         ),
219                     sinusoidal_exp_complex_integral(a, b),
220                     epsilon,
221                     epsilon));
222         }
223     }
224 }
```

```
219             );
220         }
221     }
222 }
```

src/main.rs

```
1 #![allow(dead_code)]
2 #![feature(test)]
3
4 mod airy;
5 mod airy_wave_func;
6 mod benchmarks;
7 mod check;
8 mod energy;
9 mod integrals;
10 mod newtons_method;
11 mod plot;
12 mod potentials;
13 mod tui;
14 mod turning_points;
15 mod utils;
16 mod wave_function_builder;
17 mod wkb_wave_func;
18
19 use crate::airy::airy_ai;
20 use crate::airy_wave_func::AiryWaveFunction;
21 use crate::integrals::*;
22 use crate::newtons_method::derivative;
23 use crate::utils::Func;
24 use crate::utils::*;
25 use crate::wave_function_builder::*;
26 use crate::wkb_wave_func::WkbWaveFunction;
27 use num::complex::Complex64;
28 use num::pow::Pow;
29 use rayon::iter::*;
30 use std::f64;
31 use std::fs::File;
32 use std::io::Write;
33 use std::path::Path;
34 use std::sync::Arc;
35
36 const INTEG_STEPS: usize = 64000;
37 const TRAPEZE_PER_THREAD: usize = 1000;
38 const NUMBER_OF_POINTS: usize = 100000;
39
40 const AIRY_TRANSITION_FRACTION: f64 = 0.5;
41 const ENABLE_AIRY_JOINTS: bool = true;
```

```

42
43 const VALIDITY_LL_FACTOR: f64 = 3.5;
44
45 const APPROX_INF: (f64, f64) = (-200.0, 200.0);
46 const VIEW_FACTOR: f64 = 0.5;
47
48 fn main() {
49     let wave_function = wave_function_builder::WaveFunction::new(
50         &potentials::mexican_hat,
51         1.0, // mass
52         5, // nth energy
53         APPROX_INF,
54         VIEW_FACTOR,
55         ScalingType::Renormalize(1.0.into()),
56     );
57
58     // let wave_function = wave_function_builder::SuperPosition::new(
59     //     &potentials::square,
60     //     1.0, // mass
61     //     &[
62     //         (1, complex(1.0, 0.0)), // (nth energy, phase)
63     //         (2, complex(2.0, 0.0)), // (nth energy, phase)
64     //         (3, complex(3.0, 0.0)), // (nth energy, phase)
65     //         (4, complex(4.0, 0.0)), // (nth energy, phase)
66     //         (5, complex(5.0, 0.0)), // (nth energy, phase)
67     //         (6, complex(6.0, 0.0)), // (nth energy, phase)
68     //         (7, complex(7.0, 0.0)), // (nth energy, phase)
69     //         (8, complex(8.0, 0.0)), // (nth energy, phase)
70     //         (9, complex(9.0, 0.0)), // (nth energy, phase)
71     //     ],
72     //     APPROX_INF,
73     //     VIEW_FACTOR,
74     //     ScalingType::Renormalize(complex(1.0, 0.0)),
75     // );
76
77     let output_dir = Path::new("output");
78
79     // For WaveFunction
80     // plot::plot_wavefunction(&wave_function, output_dir, "data.txt");
81     plot::plot_wavefunction_parts(&wave_function, output_dir, "data.txt");
82     // plot::plot_probability(&wave_function, output_dir, "data.txt");
83
84     // For SuperPosition
85     // plot::plot_superposition(&wave_function, output_dir, "data.txt");
86     // plot::plot_probability_superposition(&wave_function, output_dir, "data.txt");
87 }

```

src/newtons_method.rs

```

1 use crate::integrals::*;
2 use crate::utils::cmp_f64;
3 use num::Float;
4 use rayon::prelude::*;
5 use std::cmp::Ordering;
6 use std::fmt::Debug;
7 use std::ops::*;
8 use std::sync::Arc;
9
10 #[derive(Default, Debug)]
11 pub struct Vec2 {
12     x: f64,
13     y: f64,
14 }
15
16 impl Vec2 {
17     pub fn dot(&self, other: &Vec2) -> f64 {
18         return self.x * other.x + self.y + other.y;
19     }
20
21     pub fn mag(&self) -> f64 {
22         return (self.x.powi(2) + self.y.powi(2)).sqrt();
23     }
24
25     pub fn pseudo_inverse(&self) -> CoVec2 {
26         CoVec2(self.x, self.y) * (1.0 / (self.x.powi(2) + self.y.powi(2)))
27     }
28 }
29
30 impl Add for Vec2 {
31     type Output = Vec2;
32
33     fn add(self, other: Self) -> Self::Output {
34         Vec2 {
35             x: self.x + other.x,
36             y: self.y + other.y,
37         }
38     }
39 }
40
41 impl Sub for Vec2 {
42     type Output = Vec2;
43
44     fn sub(self, other: Self) -> Self::Output {
45         Vec2 {
46             x: self.x - other.x,
47             y: self.y - other.y,
48         }
49     }

```

```

50 }
51
52 impl Mul<f64> for Vec2 {
53     type Output = Vec2;
54
55     fn mul(self, s: f64) -> Self::Output {
56         Vec2 {
57             x: self.x * s,
58             y: self.y * s,
59         }
60     }
61 }
62
63 #[derive(Debug)]
64 pub struct CoVec2(f64, f64);
65
66 impl Add for CoVec2 {
67     type Output = CoVec2;
68
69     fn add(self, other: Self) -> Self::Output {
70         CoVec2(self.0 + other.0, self.1 + other.1)
71     }
72 }
73
74 impl Sub for CoVec2 {
75     type Output = CoVec2;
76
77     fn sub(self, other: Self) -> Self::Output {
78         CoVec2(self.0 - other.0, self.1 - other.1)
79     }
80 }
81
82 impl Mul<Vec2> for CoVec2 {
83     type Output = f64;
84
85     fn mul(self, vec: Vec2) -> Self::Output {
86         return self.0 * vec.x + self.1 * vec.y;
87     }
88 }
89
90 impl Mul<f64> for CoVec2 {
91     type Output = CoVec2;
92
93     fn mul(self, s: f64) -> Self::Output {
94         CoVec2(self.0 * s, self.1 * s)
95     }
96 }
97
98 fn gradient<F>(f: F, x: f64) -> Vec2

```

```

99 where
100     F: Fn(f64) -> Vec2,
101 {
102     let x_component = |x| f(x).x;
103     let y_component = |x| f(x).y;
104     return Vec2 {
105         x: derivative(&x_component, x),
106         y: derivative(&y_component, x),
107     };
108 }
109
110 pub fn derivative<F, R>(func: &F, x: f64) -> R
111 where
112     F: Fn(f64) -> R + ?Sized,
113     R: Sub<R, Output = R> + Div<f64, Output = R> + Mul<f64, Output = R> + Add<R,
114         Output = R>,
115 {
116     let dx = f64::epsilon().sqrt();
117     let dx1 = dx;
118     let dx2 = dx1 * 2.0;
119     let dx3 = dx1 * 3.0;
120
121     let m1 = (func(x + dx1) - func(x - dx1)) / 2.0;
122     let m2 = (func(x + dx2) - func(x - dx2)) / 4.0;
123     let m3 = (func(x + dx3) - func(x - dx3)) / 6.0;
124
125     let fifteen_m1 = m1 * 15.0;
126     let six_m2 = m2 * 6.0;
127     let ten_dx1 = dx1 * 10.0;
128
129     return ((fifteen_m1 - six_m2) + m3) / ten_dx1;
130 }
131
132 pub fn newtons_method<F>(f: &F, mut guess: f64, precision: f64) -> f64
133 where
134     F: Fn(f64) -> f64,
135 {
136     loop {
137         let deriv = derivative(f, guess);
138
139         if deriv == 0.0 {
140             panic!("Devision_by_zero");
141         }
142
143         let step = f(guess) / deriv;
144         if step.abs() < precision {
145             return guess;
146         } else {
147             guess -= step;

```

```

147         }
148     }
149 }
150
151 pub fn newtons_method_2d<F>(f: &F, mut guess: f64, precision: f64) -> f64
152 where
153     F: Fn(f64) -> Vec2,
154     F::Output: Debug,
155 {
156     loop {
157         let jacobian = gradient(f, guess);
158         let step: f64 = jacobian.pseudo_inverse() * f(guess);
159         if step.abs() < precision {
160             return guess;
161         } else {
162             guess -= step;
163         }
164     }
165 }
166
167 pub fn newtons_method_max_iters<F>(
168     f: &F,
169     mut guess: f64,
170     precision: f64,
171     max_iters: usize,
172 ) -> Option<f64>
173 where
174     F: Fn(f64) -> f64,
175 {
176     for _ in 0..max_iters {
177         let derivative = derivative(f, guess);
178         if derivative == 0.0 {
179             return None;
180         }
181         let step = f(guess) / derivative;
182         if step.abs() < precision {
183             return Some(guess);
184         } else {
185             guess -= step;
186         }
187     }
188     None
189 }
190
191 fn sigmoid(x: f64) -> f64 {
192     1.0 / (1.0 + (-x).exp())
193 }
194
195 fn check_sign(initial: f64, new: f64) -> bool {

```

```

196     if initial == new {
197         return false;
198     }
199     return (initial <= -0.0 && new >= 0.0) || (initial >= 0.0 && new <= 0.0);
200 }
201
202 pub fn bisection_search_sign_change<F>(f: &F, initial_guess: f64, step: f64) -> (f64,
203                                         f64)
204 where
205     F: Fn(f64) -> f64 + ?Sized,
206 {
207     let mut result = initial_guess;
208     while !check_sign(f(initial_guess), f(result)) {
209         result += step
210     }
211     return (result - step, result);
212 }
213
214 fn regula_falsi_c<F>(f: &F, a: f64, b: f64) -> f64
215 where
216     F: Fn(f64) -> f64 + ?Sized,
217 {
218     return (a * f(b) - b * f(a)) / (f(b) - f(a));
219 }
220
221 pub fn regula_falsi_method<F>(f: &F, mut a: f64, mut b: f64, precision: f64) -> f64
222 where
223     F: Fn(f64) -> f64 + ?Sized,
224 {
225     if a > b {
226         let temp = a;
227         a = b;
228         b = temp;
229     }
230
231     let mut c = regula_falsi_c(f, a, b);
232     while f64::abs(f(c)) > precision {
233         b = regula_falsi_c(f, a, b);
234         a = regula_falsi_c(f, a, b);
235         c = regula_falsi_c(f, a, b);
236     }
237     return c;
238 }
239
240 pub fn regula_falsi_bisection<F>(f: &F, guess: f64, bisection_step: f64, precision:
241                                         f64) -> f64
242 where
243     F: Fn(f64) -> f64 + ?Sized,
244 {

```

```

243     let (a, b) = bisection_search_sign_change(f, guess, bisection_step);
244     return regula_falsi_method(f, a, b, precision);
245 }
246
247 #[derive(Clone)]
248 pub struct NewtonsMethodFindNewZero<F>
249 where
250     F: Fn(f64) -> f64 + ?Sized + Clone,
251 {
252     f: Arc<F>,
253     precision: f64,
254     max_iters: usize,
255     previous_zeros: Vec<(i32, f64)>,
256 }
257
258 impl<F: Fn(f64) -> f64 + ?Sized + Clone> NewtonsMethodFindNewZero<F> {
259     pub(crate) fn new(f: Arc<F>, precision: f64, max_iters: usize) ->
260         NewtonsMethodFindNewZero<F> {
261         NewtonsMethodFindNewZero {
262             f,
263             precision,
264             max_iters,
265             previous_zeros: vec![],
266         }
267     }
268
269     pub(crate) fn modified_func(&self, x: f64) -> f64 {
270         let divisor = self
271             .previous_zeros
272             .iter()
273             .fold(1.0, |acc, (n, z)| acc * (x - z).powi(*n));
274         let divisor = if divisor == 0.0 {
275             divisor + self.precision
276         } else {
277             divisor
278         };
279         (self.f)(x) / divisor
280     }
281
282     pub(crate) fn next_zero(&mut self, guess: f64) -> Option<f64> {
283         let zero = newtons_method_max_iters(
284             &|x| self.modified_func(x),
285             guess,
286             self.precision,
287             self.max_iters,
288         );
289
290         if let Some(z) = zero {
291             // to avoid hitting maxima and minima twice

```

```

291         if derivative(&|x| self.modified_func(x), z).abs() < self.precision {
292             self.previous_zeros.push((2, z));
293         } else {
294             self.previous_zeros.push((1, z));
295         }
296     }
297
298     return zero;
299 }
300
301 pub(crate) fn get_previous_zeros(&self) -> Vec<f64> {
302     self.previous_zeros
303         .iter()
304         .map(|(_ , z)| *z)
305         .collect::<Vec<f64>>()
306 }
307 }
308
309 pub fn make_guess<F>(f: &F, (start, end): (f64, f64), n: usize) -> Option<f64>
310 where
311     F: Fn(f64) -> f64 + Sync,
312 {
313     let sort_func = |( _, y1): &(f64, f64), ( _, y2): &(f64, f64)| -> Ordering {
314         cmp_f64(&y1, &y2);
315     }
316     let mut points: Vec<(f64, f64)> = (0..n)
317         .into_par_iter()
318         .map(|i| index_to_range(i as f64, 0.0, n as f64, start, end))
319         .map(move |x| {
320             let der = derivative(f, x);
321             (x, f(x) / (-(-der * der).exp() + 1.0))
322         })
323         .map(|(x, y)| (x, y.abs()))
324         .collect();
325     points.sort_by(sort_func);
326     points.get(0).map(|point| point.0)
327 }
328
329 pub fn newtons_method_find_new_zero<F>(
330     f: &F,
331     guess: f64,
332     precision: f64,
333     max_iters: usize,
334     known_zeros: &Vec<f64>,
335 ) -> Option<f64>
336 where
337     F: Fn(f64) -> f64,
338 {
339     let f_modified = |x| f(x) / known_zeros.iter().fold(0.0, |acc, &z| acc * (x - z));

```

```

338     newtons_method_max_iters(&f_modified, guess, precision, max_iters)
339 }
340
341 #[cfg(test)]
342 mod test {
343     use super::*;

344     use crate::utils::cmp_f64;
345
346     fn float_compare(expect: f64, actual: f64, epsilon: f64) -> bool {
347         let average = (expect.abs() + actual.abs()) / 2.0;
348         if average != 0.0 {
349             (expect - actual).abs() / average < epsilon
350         } else {
351             (expect - actual).abs() < epsilon
352         }
353     }
354
355     #[test]
356     fn derivative_square_test() {
357         let square = |x| x * x;
358         let actual = |x| 2.0 * x;
359
360         for i in 0..100 {
361             let x = index_to_range(i as f64, 0.0, 100.0, -20.0, 20.0);
362             assert!(float_compare(derivative(&square, x), actual(x), 1e-4));
363         }
364     }
365
366     #[test]
367     fn derivative_exp_test() {
368         let exp = |x: f64| x.exp();
369
370         for i in 0..100 {
371             let x = index_to_range(i as f64, 0.0, 100.0, -20.0, 20.0);
372             assert!(float_compare(derivative(&exp, x), exp(x), 1e-4));
373         }
374     }
375
376     #[test]
377     fn newtons_method_square() {
378         for i in 0..100 {
379             let zero = index_to_range(i as f64, 0.0, 100.0, 0.1, 10.0);
380             let func = |x| x * x - zero * zero;
381             assert!(float_compare(
382                 newtons_method(&func, 100.0, 1e-7),
383                 zero,
384                 1e-4,
385             ));
386             assert!(float_compare(

```

```

387             newtons_method(&func, -100.0, 1e-7),
388             -zero,
389             1e-4,
390         ));
391     }
392 }
393
394 #[test]
395 fn newtons_method_cube() {
396     for i in 0..100 {
397         let zero = index_to_range(i as f64, 0.0, 100.0, 0.1, 10.0);
398         let func = |x| (x - zero) * (x + zero) * (x - zero / 2.0);
399         assert!(float_compare(
400             newtons_method(&func, 100.0, 1e-7),
401             zero,
402             1e-4,
403         ));
404         assert!(float_compare(
405             newtons_method(&func, -100.0, 1e-7),
406             -zero,
407             1e-4,
408         ));
409         assert!(float_compare(
410             newtons_method(&func, 0.0, 1e-7),
411             zero / 2.0,
412             1e-4,
413         ));
414     }
415 }
416
417 #[test]
418 fn newtons_method_find_next_polynomial() {
419     for i in 0..10 {
420         for j in 0..10 {
421             for k in 0..10 {
422                 let a = index_to_range(i as f64, 0.0, 10.0, -10.0, 10.0);
423                 let b = index_to_range(j as f64, 0.0, 10.0, -100.0, 0.0);
424                 let c = index_to_range(k as f64, 0.0, 10.0, -1.0, 20.0);
425                 let test_func = |x: f64| (x - a) * (x - b) * (x - c);

426                 for _guess in [a, b, c] {
427                     let mut finder =
428                         NewtonsMethodFindNewZero::new(Arc::new(test_func), 1e-15,
429                                         10000000);
430
431                     finder.next_zero(1.0);
432                     finder.next_zero(1.0);
433                     finder.next_zero(1.0);
434

```

```

435     let mut zeros_expected = [a, b, c];
436     let mut zeros_actual = finder.get_previous_zeros().clone();
437
438     zeros_expected.sort_by(cmp_f64);
439     zeros_actual.sort_by(cmp_f64);
440
441     assert_eq!(zeros_actual.len(), 3);
442
443     for (expected, actual) in zeros_expected.iter().zip(
444         zeros_actual.iter()) {
445         assert!((*expected - *actual).abs() < 1e-10);
446     }
447 }
448 }
449 }
450 }
451
452 #[test]
453 fn newtons_method_find_next_test() {
454     let interval = (-10.0, 10.0);
455
456     let test_func = |x: f64| 5.0 * (3.0 * x + 1.0).abs() - (1.5 * x.powi(2) + x -
457         50.0).powi(2);
458
459     let mut finder = NewtonsMethodFindNewZero::new(Arc::new(test_func), 1e-11,
460         100000000);
461
462     for _i in 0..4 {
463         let guess = make_guess(&|x| finder.modified_func(x), interval, 1000);
464         finder.next_zero(guess.unwrap());
465     }
466
467     let mut zeros = finder.get_previous_zeros().clone();
468     zeros.sort_by(cmp_f64);
469     let expected = [-6.65276132415, -5.58024707627, 4.91358040961,
470         5.98609465748];
471
472     println!("zeros:{:#?}", zeros);
473
474     assert_eq!(zeros.len(), expected.len());
475
476     for (expected, actual) in expected.iter().zip(zeros.iter()) {
477         assert!((*expected - *actual).abs() < 1e-10);
478     }
479
480 #[test]
481 fn regula_falsi_bisection_test() {

```

```

480     let func = |x: f64| x * (x - 2.0) * (x + 2.0);
481
482     let actual = regula_falsi_bisection(&func, -1e-3, -1e-3, 1e-5);
483     let expected = -2.0;
484
485     println!("expected:{}{}, actual:{}", expected, actual);
486     assert!(float_compare(expected, actual, 1e-3));
487 }
488 }
```

src/plot.rs

```

1 use crate::*;

2 use std::fmt;

3

4 pub fn to_gnuplot_string_complex<X>(values: Vec<Point<X, Complex64>>) -> String
5 where
6     X: fmt::Display + Send + Sync,
7 {
8     values
9         .par_iter()
10        .map(|p| -> String { format!("{}{}{}\n", p.x, p.y.re, p.y.im) })
11        .reduce(|| String::new(), |s: String, current: String| s + &*current)
12 }
13

14 pub fn to_gnuplot_string<X, Y>(values: Vec<Point<X, Y>>) -> String
15 where
16     X: fmt::Display + Send + Sync,
17     Y: fmt::Display + Send + Sync,
18 {
19     values
20         .par_iter()
21        .map(|p| -> String { format!("{}{}\n", p.x, p.y) })
22        .reduce(|| String::new(), |s: String, current: String| s + &*current)
23 }
24

25 pub fn plot_wavefunction_parts(wave_function: &WaveFunction, output_dir: &Path,
26                                 output_file: &str) {
27     std::env::set_current_dir(&output_dir).unwrap();
28
29     let wkb_values = wave_function
30         .get_wkb_ranges_in_view()
31         .iter()
32         .map(|range| evaluate_function_between(wave_function, range.0, range.1,
33                                              NUMBER_OF_POINTS))
34         .collect::<Vec<Vec<Point<f64, Complex64>>>();
35
36     let airy_values = wave_function
37         .get_airy_ranges()
```

```

36     .iter()
37     .map(|range| {
38         evaluate_function_between(
39             wave_function,
40             f64::max(wave_function.get_view().0, range.0),
41             f64::min(wave_function.get_view().1, range.1),
42             NUMBER_OF_POINTS,
43         )
44     })
45     .collect::<Vec<Vec<Point<f64, Complex64>>>();
46
47 let wkb_values_str = wkb_values
48     .par_iter()
49     .map(|values| to_gnuplot_string_complex(values.to_vec()))
50     .reduce(
51         || String::new(),
52         |s: String, current: String| s + "\n\n" + &*current,
53     );
54
55 let airy_values_str = airy_values
56     .par_iter()
57     .map(|values| to_gnuplot_string_complex(values.to_vec()))
58     .reduce(
59         || String::new(),
60         |s: String, current: String| s + "\n\n" + &*current,
61     );
62
63 let mut data_full = File::create(output_file).unwrap();
64 data_full.write_all(wkb_values_str.as_ref()).unwrap();
65 data_full.write_all("\n\n".as_bytes()).unwrap();
66 data_full.write_all(airy_values_str.as_ref()).unwrap();
67
68 let mut plot_3d_file = File::create("plot_3d.gnuplot").unwrap();
69
70 let wkb_3d_cmd = (1..=wkb_values.len())
71     .into_iter()
72     .map(|n| {
73         format!(
74             "\">\u{1d3f}1:2:3\u{1d3f}\t\"WKB\u{1d3f}\">\u{1d3f}l",
75             output_file,
76             n - 1,
77             n
78         )
79     })
80     .collect::<Vec<String>>()
81     .join(",");
82
83 let airy_3d_cmd = (1..=airy_values.len())
84     .into_iter()

```

```

85     .map(|n| {
86         format!(
87             "\"{}\"{}1:2{}{}\"Airy{}\"{}",
88             output_file,
89             n + wkb_values.len() - 1,
90             n
91         )
92     })
93     .collect::<Vec<String>>()
94     .join(",");
95     let plot_3d_cmd: String = "splot".to_string() + &wkb_3d_cmd + "," + &
96         airy_3d_cmd;
97     plot_3d_file.write_all(plot_3d_cmd.as_ref()).unwrap();
98
99     let mut plot_file = File::create("plot.gnuplot").unwrap();
100    let wkb_cmd = (1..=wkb_values.len())
101        .into_iter()
102        .map(|n| {
103            format!(
104                "\"{}\"{}1:2{}{}\"Re(WKB{})\"{}",
105                output_file,
106                n - 1,
107                n
108            )
109        })
110        .collect::<Vec<String>>()
111        .join(",");
112
113    let airy_cmd = (1..=airy_values.len())
114        .into_iter()
115        .map(|n| {
116            format!(
117                "\"{}\"{}1:2{}{}\"Re(Airy{})\"{}",
118                output_file,
119                n + wkb_values.len() - 1,
120                n
121            )
122        })
123        .collect::<Vec<String>>()
124        .join(",");
125    let plot_cmd: String = "plot".to_string() + &wkb_cmd + "," + &airy_cmd;
126
127    plot_file.write_all(plot_cmd.as_ref()).unwrap();
128
129    let mut plot_imag_file = File::create("plot_im.gnuplot").unwrap();
130
131    let wkb_im_cmd = (1..=wkb_values.len())
132        .into_iter()
133        .map(|n| {

```

```

133     format!(
134         "\\"{}\"_u_1:3_u_i_u{}_t_u\"Im(WKB_u{})\"_w_u_l",
135         output_file,
136         n - 1,
137         n
138     )
139 }
140 .collect::<Vec<String>>()
141 .join(",");
142
143 let airy_im_cmd = (1..=airy_values.len())
144 .into_iter()
145 .map(|n| {
146     format!(
147         "\\"{}\"_u_1:3_u_i_u{}_t_u\"Im(Airy_u{})\"_w_u_l",
148         output_file,
149         n + wkb_values.len() - 1,
150         n
151     )
152 })
153 .collect::<Vec<String>>()
154 .join(",");
155 let plot_imag_cmd: String = "plot_u".to_string() + &wkb_im_cmd + ",_u" + &
156     airy_im_cmd;
157
158 plot_imag_file.write_all(plot_imag_cmd.as_ref()).unwrap();
159 }
160
161 pub fn plot_complex_function(
162     func: &dyn Func<f64, Complex64>,
163     view: (f64, f64),
164     title: &str,
165     output_dir: &Path,
166     output_file: &str,
167     color_plot: bool,
168 ) {
169     std::env::set_current_dir(&output_dir).unwrap();
170     let values = evaluate_function_between(func, view.0, view.1, NUMBER_OF_POINTS);
171
172     let values_str = to_gnuplot_string_complex(values);
173
174     let mut data_file = File::create(output_file).unwrap();
175
176     data_file.write_all(values_str.as_bytes()).unwrap();
177
178     let mut plot_3d_file = File::create("plot_3d.gnuplot").unwrap();
179     plot_3d_file
180         .write_all(format!("splot_u\"{}\"_u_1:2:3_u_t_u\"{}\"_w_u_l", output_file, title).
181             as_bytes())

```

```

180     .unwrap();
181
182     let mut plot_file = File::create("plot.gnuplot").unwrap();
183     plot_file
184         .write_all(format!("plot \"{}\" u 1:2 t \"Re({})\" w l", output_file, title).
185                     as_bytes())
186         .unwrap();
187
188     let mut plot_im_file = File::create("plot_im.gnuplot").unwrap();
189     plot_im_file
190         .write_all(format!("plot \"{}\" u 1:3 t \"Im({})\" w l", output_file, title).
191                     as_bytes())
192         .unwrap();
193     if color_plot {
194         let mut plot_color_file = File::create("plot_color.gnuplot").unwrap();
195         plot_color_file
196             .write_all(
197                 format!(
198                     "set cbrange [-pi:pi]\nset xlabel \"arg({})\"\nset ylabel \"|{}|\"\n
199                     set palette model HSV defined (0,0,1,1,1,1)\nplot \"{}\" u
200                     1:(sqrt($2**2+$3**2)):(atan2($2,$3)) w boxes t \"{}\" lc
201                     palette z",
202                     title, title, output_file, title
203                 )
204             .as_bytes(),
205         )
206         .unwrap();
207     }
208 }
209
210 pub fn plot_wavefunction(wave_function: &WaveFunction, output_dir: &Path, output_file:
211 : &str) {
212     plot_complex_function(
213         wave_function,
214         wave_function.get_view(),
215         "Psi",
216         output_dir,
217         output_file,
218         true,
219     );
220 }
221
222 pub fn plot_superposition(wave_function: &SuperPosition, output_dir: &Path,
223 output_file: &str) {
224     plot_complex_function(
225         wave_function,
226         wave_function.get_view(),
227         "Psi",
228         output_dir,
229     );
230 }
```

```

222         output_file,
223         true,
224     );
225 }
226
227 pub fn plot_probability(wave_function: &WaveFunction, output_dir: &Path, output_file: &str) {
228     std::env::set_current_dir(&output_dir).unwrap();
229     let values = evaluate_function_between(
230         wave_function,
231         wave_function.get_view().0,
232         wave_function.get_view().1,
233         NUMBER_OF_POINTS,
234     )
235     .par_iter()
236     .map(|p| Point {
237         x: p.x,
238         y: p.y.norm_sqr(),
239     })
240     .collect();
241
242     let values_str = to_gnuplot_string(values);
243
244     let mut data_file = File::create(output_file).unwrap();
245
246     data_file.write_all(values_str.as_bytes()).unwrap();
247
248     let mut plot_file = File::create("plot.gnuplot").unwrap();
249     plot_file
250         .write_all(format!("plot \"{}\" u 1:2 t \"|Psi|^2\" w l", output_file).
251             as_bytes())
252         .unwrap();
253 }
254
255 pub fn plot_probability_superposition(
256     wave_function: &SuperPosition,
257     output_dir: &Path,
258     output_file: &str,
259 ) {
260     std::env::set_current_dir(&output_dir).unwrap();
261     let values = evaluate_function_between(
262         wave_function,
263         wave_function.get_view().0,
264         wave_function.get_view().1,
265         NUMBER_OF_POINTS,
266     )
267     .par_iter()
268     .map(|p| Point {
         x: p.x,

```

```

269         y: p.y.norm_sqr(),
270     })
271     .collect();
272
273     let values_str = to_gnuplot_string(values);
274
275     let mut data_file = File::create(output_file).unwrap();
276
277     data_file.write_all(values_str.as_bytes()).unwrap();
278
279     let mut plot_file = File::create("plot.gnuplot").unwrap();
280     plot_file
281         .write_all(format!("plot \\"{}\" u 1:2 t \"|Psi|^2\" w l", output_file).
282             as_bytes())
283         .unwrap();
284 }

```

src/potentials.rs

```

1 use crate::*;

2

3 const ENERGY_INF: f64 = 1e6;

4

5 #[allow(unused)]
6 pub fn smooth_step(x: f64) -> f64 {
7     const TRANSITION: f64 = 0.5;
8     let step = Arc::new(Function::new(|x: f64| -> Complex64 {
9         if x.abs() < 2.0 {
10             complex(10.0, 0.0)
11         } else {
12             complex(0.0, 0.0)
13         }
14     }));
15     let zero = Arc::new(Function::new(|_: f64| -> Complex64 { complex(0.0, 0.0) }));
16     let inf = Arc::new(Function::new(|x: f64| -> Complex64 {
17         if x.abs() > 5.0 {
18             complex(ENERGY_INF, 0.0)
19         } else {
20             complex(0.0, 0.0)
21         }
22     }));
23
24     let joint_inf_zero_l = wave_function_builder::Joint {
25         left: inf.clone(),
26         right: zero.clone(),
27         cut: -5.0 + TRANSITION / 2.0,
28         delta: TRANSITION,
29     };
30

```

```

31 let joint_zero_step_l = wave_function_builder::Joint {
32     left: zero.clone(),
33     right: step.clone(),
34     cut: -2.0 + TRANSITION / 2.0,
35     delta: TRANSITION,
36 };
37
38 let joint_zero_inf_r = wave_function_builder::Joint {
39     left: zero.clone(),
40     right: inf.clone(),
41     cut: 5.0 - TRANSITION / 2.0,
42     delta: TRANSITION,
43 };
44
45 let joint_step_zero_r = wave_function_builder::Joint {
46     left: step.clone(),
47     right: zero.clone(),
48     cut: 2.0 - TRANSITION / 2.0,
49     delta: TRANSITION,
50 };
51
52 if wave_function_builder::is_in_range(joint_zero_inf_r.range(), x) {
53     return joint_zero_inf_r.eval(x).re;
54 }
55
56 if wave_function_builder::is_in_range(joint_inf_zero_l.range(), x) {
57     return joint_inf_zero_l.eval(x).re;
58 }
59
60 if wave_function_builder::is_in_range(joint_step_zero_r.range(), x) {
61     return joint_step_zero_r.eval(x).re;
62 }
63
64 if wave_function_builder::is_in_range(joint_zero_step_l.range(), x) {
65     return joint_zero_step_l.eval(x).re;
66 }
67
68 return zero.eval(x).re.max(inf.eval(x).re.max(step.eval(x).re));
69 }
70
71 #[allow(unused)]
72 pub fn mexican_hat(x: f64) -> f64 {
73     (x - 4.0).powi(2) * (x + 4.0).powi(2)
74 }
75
76 #[allow(unused)]
77 pub fn double_mexican_hat(x: f64) -> f64 {
78     (x - 4.0).powi(2) * x.powi(2) * (x + 4.0).powi(2)
79 }
```

```

80
81 #[allow(unused)]
82 pub fn triple_mexican_hat(x: f64) -> f64 {
83     (x - 6.0).powi(2) * (x - 3.0).powi(2) * (x + 3.0).powi(2) * (x + 6.0).powi(2)
84 }
85
86 pub fn square(x: f64) -> f64 {
87     x * x
88 }
```

src/tui.rs

```

1 use std::io;
2
3 fn get_float_from_user(message: &str) -> f64 {
4     loop {
5         println!("{}: ", message);
6         let mut input = String::new();
7
8         // io::stdout().lock().write(message.as_ref()).unwrap();
9         io::stdin()
10            .read_line(&mut input)
11            .expect("Not a valid string");
12         println!("");
13         let num = input.trim().parse();
14         if num.is_ok() {
15             return num.unwrap();
16         }
17     }
18 }
19
20 fn get_user_bounds() -> (f64, f64) {
21     let user_bound_lower: f64 = get_float_from_user("Lower bound: ");
22
23     let user_bound_upper: f64 = get_float_from_user("Upper bound: ");
24     return (user_bound_lower, user_bound_upper);
25 }
26 fn ask_user_for_view(lower_bound: Option<f64>, upper_bound: Option<f64>) -> (f64, f64)
27 {
28     println!("Failed to determine boundary of the graph automatically.");
29     println!("Please enter values manually.");
30     lower_bound.map(|b| println!("(Suggestion for lower bound: {})", b));
31     upper_bound.map(|b| println!("(Suggestion for upper bound: {})", b));
32
33     return get_user_bounds();
34 }
```

src/turning_points.rs

```

1 use crate::cmp_f64;
2 use crate::newtons_method::*;
3 use crate::wkb_wave_func::*;
4 use crate::*;
5 use num::signum;
6
7 const MAX_TURNING_POINTS: usize = 2048;
8 const ACCURACY: f64 = 1e-9;
9
10 pub struct TGroup {
11     pub ts: Vec<((f64, f64), f64)>,
12     // pub tn: Option<f64>,
13 }
14
15 impl TGroup {
16     pub fn new() -> TGroup {
17         TGroup { ts: vec![] }
18     }
19
20     pub fn add_ts(&mut self, new_t: ((f64, f64), f64)) {
21         self.ts.push(new_t);
22     }
23 }
24
25 fn validity_func(phase: Phase) -> Arc<dyn Fn(f64) -> f64> {
26     Arc::new(move |x: f64| {
27         1.0 / (2.0 * phase.mass).sqrt()
28         * derivative(&|t| (phase.potential)(t), x).abs()
29         * VALIDITY_LL_FACTOR
30         - ((phase.potential)(x) - phase.energy).pow(2)
31     })
32 }
33
34 fn group_ts(zeros: &Vec<f64>, phase: &Phase) -> TGroup {
35     let mut zeros = zeros.clone();
36     let valid = validity_func(phase.clone());
37
38     zeros.sort_by(cmp_f64);
39     let mut derivatives = zeros
40         .iter()
41         .map(|x| derivative(valid.as_ref(), *x))
42         .map(signum)
43         .zip(zeros.clone())
44         .collect::<Vec<(f64, f64)>>();
45
46     let mut groups = TGroup { ts: vec![] };
47
48     if let Some((deriv, z)) = derivatives.first() {
49         if *deriv < 0.0 {

```

```

50     let mut guess = z - ACCURACY.sqrt();
51     let mut new_deriv = *deriv;
52     let mut missing_t = *z;
53
54     while new_deriv < 0.0 {
55         missing_t =
56             regula_falsi_bisection(valid.as_ref(), guess, -ACCURACY.sqrt(),
57                                     ACCURACY);
58         new_deriv = signum(derivative(valid.as_ref(), missing_t));
59         guess -= ACCURACY.sqrt();
60     }
61
62     derivatives.insert(
63         0,
64         (signum(derivative(valid.as_ref(), missing_t)), missing_t),
65     );
66 }
67
68 if let Some((deriv, z)) = derivatives.last() {
69     if *deriv > 0.0 {
70         let mut guess = z + ACCURACY.sqrt();
71         let mut new_deriv = *deriv;
72         let mut missing_t = *z;
73
74         while new_deriv > 0.0 {
75             missing_t =
76                 regula_falsi_bisection(valid.as_ref(), guess, ACCURACY.sqrt(),
77                                         ACCURACY);
78             new_deriv = signum(derivative(valid.as_ref(), missing_t));
79             guess += ACCURACY.sqrt();
80         }
81
82         derivatives.push((signum(derivative(valid.as_ref(), missing_t)),
83                           missing_t));
84     }
85 }
86
87 assert_eq!(derivatives.len() % 2, 0);
88
89 for i in (0..derivatives.len()).step_by(2) {
90     let (t1_deriv, t1) = derivatives[i];
91     let (t2_deriv, t2) = derivatives[i + 1];
92     assert!(t1_deriv > 0.0);
93     assert!(t2_deriv < 0.0);
94
95     let turning_point = newtons_method(
96         &|x| phase.energy - (phase.potential)(x),
97         (t1 + t2) / 2.0,

```

```

96         1e-7,
97     );
98     groups.add_ts(((t1, t2), turning_point));
99 }
100
101    return groups;
102 }
103
104 pub fn calc_ts(phase: &Phase, view: (f64, f64)) -> TGroup {
105     let zeros = find_zeros(phase, view);
106     let groups = group_ts(&zeros, phase);
107     return groups;
108 }
109
110 fn find_zeros(phase: &Phase, view: (f64, f64)) -> Vec<f64> {
111     let phase_clone = phase.clone();
112     let validity_func = Arc::new(move |x: f64| {
113         1.0 / (2.0 * phase_clone.mass).sqrt()
114             * derivative(&|t| (phase_clone.potential)(t), x).abs()
115             * VALIDITY_LL_FACTOR
116             - ((phase_clone.potential)(x) - phase_clone.energy).pow(2)
117     });
118     let mut zeros = NewtonsMethodFindNewZero::new(validity_func, ACCURACY, 1e4 as
119         usize);
120
121     for _ in 0..MAX_TURNING_POINTS {
122         let modified_func = |x| zeros.modified_func(x);
123
124         let guess = make_guess(&modified_func, view, 1000);
125         let result = guess.map(|g| zeros.next_zero(g)).flatten();
126         if result.is_none() {
127             break;
128         }
129
130         let view = if view.0 < view.1 {
131             view
132         } else {
133             (view.1, view.0)
134         };
135         let unique_zeros = zeros
136             .get_previous_zeros()
137             .iter()
138             .filter(|x| **x > view.0 && **x < view.1)
139             .map(|x| *x)
140             .collect::<Vec<f64>>();
141         return unique_zeros;
142     }
143 }
```

```

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

147     extern crate test;
148
149     use test::Bencher;
150
151     use duplicate::duplicate_item;
152     use paste::paste;
153     use std::sync::Arc;
154
155     #[duplicate_item(
156         num,
157         [1];
158         [2];
159         [3];
160         [4];
161         [5];
162         [6];
163         [7];
164         [8];
165         [9];
166     )]
167     ]
168     paste! {
169         #[bench]
170         fn [< turning_point_square_nenergy_ num >](b: &mut Bencher) {
171             let potential = &potentials::square;
172             let mass = 1.0;
173
174             let energy = energy::nth_energy(num, mass, potential, APPROX_INF);
175             let lower_bound = newtons_method::newtons_method(
176                 &|x| potential(x) - energy,
177                 APPROX_INF.0,
178                 1e-7,
179             );
180             let upper_bound = newtons_method::newtons_method(
181                 &|x| potential(x) - energy,
182                 APPROX_INF.1,
183                 1e-7,
184             );
185             let phase = Arc::new(Phase::new(energy, mass, potential));
186             let view =
187                 (
188                     lower_bound - (upper_bound - lower_bound) * VIEW_FACTOR,
189                     upper_bound + (upper_bound - lower_bound) * VIEW_FACTOR,
190                 );
191             b.iter(|| {
192                 let _result = AiryWaveFunction::new(phase.clone(), test::black_box(

```

```
        view.0, view.1)));
193     });
194   }
195 }
196 }
```

src/utils.rs

```
1 use crate::newtons_method::derivative;
2 use crate::Complex64;
3 use std::cmp::Ordering;
4
5 pub fn cmp_f64(a: &f64, b: &f64) -> Ordering {
6     if a < b {
7         return Ordering::Less;
8     } else if a > b {
9         return Ordering::Greater;
10    }
11    return Ordering::Equal;
12 }
13
14 pub fn complex(re: f64, im: f64) -> Complex64 {
15     return Complex64 { re, im };
16 }
17
18 pub fn sigmoid(x: f64) -> f64 {
19     1.0 / (1.0 + (-x).exp())
20 }
21
22 pub fn identity(c: Complex64) -> Complex64 {
23     c
24 }
25
26 pub fn conjugate(c: Complex64) -> Complex64 {
27     c.conj()
28 }
29
30 pub fn negative(c: Complex64) -> Complex64 {
31     -c
32 }
33
34 pub fn negative_conj(c: Complex64) -> Complex64 {
35     -c.conj()
36 }
37
38 pub fn complex_compare(expect: Complex64, actual: Complex64, epsilon: f64) -> bool {
39     let average = (expect.norm() + actual.norm()) / 2.0;
40     return (expect - actual).norm() / average < epsilon;
41 }
```

```

42
43 pub fn float_compare(expect: f64, actual: f64, epsilon: f64) -> bool {
44     let average = (expect + actual) / 2.0;
45
46     if average < epsilon {
47         return expect == actual;
48     }
49
50     return (expect - actual) / average < epsilon;
51 }
52
53 pub trait Func<A, R>: Sync + Send {
54     fn eval(&self, x: A) -> R;
55 }
56
57 pub trait ReToC: Sync + Func<f64, Complex64> {}
58
59 pub trait ReToRe: Sync + Func<f64, f64> {}
60
61 pub struct Function<A, R> {
62     pub(crate) f: fn(A) -> R,
63 }
64
65 impl<A, R> Function<A, R> {
66     pub const fn new(f: fn(A) -> R) -> Function<A, R> {
67         return Function { f };
68     }
69 }
70
71 impl<A, R> Func<A, R> for Function<A, R> {
72     fn eval(&self, x: A) -> R {
73         (self.f)(x)
74     }
75 }
76
77 pub struct NormSquare<'a> {
78     pub f: &'a dyn Func<f64, Complex64>,
79 }
80
81 impl Func<f64, f64> for NormSquare<'_> {
82     fn eval(&self, x: f64) -> f64 {
83         self.f.eval(x).norm_sqr()
84     }
85 }
86
87 pub struct Derivative<'a> {
88     pub f: &'a dyn Func<f64, Complex64>,
89 }
90

```

```

91 impl Func<f64, Complex64> for Derivative<'_> {
92     fn eval(&self, x: f64) -> Complex64 {
93         derivative(&|x| self.f.eval(x), x)
94     }
95 }
```

src/wave_function_builder.rs

```

1 use crate::wkb_wave_func::Phase;
2 use crate::*;

3 use ordinal::Ordinal;
4 use std::sync::*;

5

6 pub enum ScalingType {
7     Mul(Complex64),
8     Renormalize(Complex64),
9     None,
10 }

11

12 pub trait WaveFunctionPart: Func<f64, Complex64> + Sync + Send {
13     fn range(&self) -> (f64, f64);
14     fn as_func(&self) -> Box<dyn Func<f64, Complex64>>;
15 }

16

17 pub trait WaveFunctionPartWithOp: WaveFunctionPart {
18     fn get_op(&self) -> Box<fn(Complex64) -> Complex64>;
19     fn with_op(&self, op: fn(Complex64) -> Complex64) -> Box<dyn
20         WaveFunctionPartWithOp>;
21     fn as_wave_function_part(&self) -> Box<dyn WaveFunctionPart>;
22 }

23 pub fn is_in_range(range: (f64, f64), x: f64) -> bool {
24     return range.0 <= x && range.1 > x;
25 }

26

27 #[derive(Clone)]
28 pub struct Joint {
29     pub left: Arc<dyn Func<f64, Complex64>>,
30     pub right: Arc<dyn Func<f64, Complex64>>,
31     pub cut: f64,
32     pub delta: f64,
33 }

34

35 impl WaveFunctionPart for Joint {
36     fn range(&self) -> (f64, f64) {
37         if self.delta > 0.0 {
38             (self.cut, self.cut + self.delta)
39         } else {
40             (self.cut + self.delta, self.cut)
41         }
42     }
43 }
```

```

41         }
42     }
43     fn as_func(&self) -> Box<dyn Func<f64, Complex64>> {
44         return Box::new(self.clone());
45     }
46 }
47
48 impl Func<f64, Complex64> for Joint {
49     fn eval(&self, x: f64) -> Complex64 {
50         let (left, right) = if self.delta > 0.0 {
51             (&self.left, &self.right)
52         } else {
53             (&self.right, &self.left)
54         };
55
56         let delta = self.delta.abs();
57
58         let chi = |x: f64| f64::sin(x * f64::consts::PI / 2.0).powi(2);
59         let left_val = left.eval(x);
60         return left_val + (right.eval(x) - left_val) * chi((x - self.cut) / delta);
61     }
62 }
63
64 #[derive(Clone)]
65 struct PureWkb {
66     wkb: Arc<WkbWaveFunction>,
67     range: (f64, f64),
68 }
69
70 impl WaveFunctionPart for PureWkb {
71     fn range(&self) -> (f64, f64) {
72         self.range
73     }
74     fn as_func(&self) -> Box<dyn Func<f64, Complex64>> {
75         Box::new(self.clone())
76     }
77 }
78
79 impl WaveFunctionPartWithOp for PureWkb {
80     fn as_wave_function_part(&self) -> Box<dyn WaveFunctionPart> {
81         Box::new(self.clone())
82     }
83
84     fn get_op(&self) -> Box<fn(Complex64) -> Complex64> {
85         self.wkb.get_op()
86     }
87
88     fn with_op(&self, op: fn(Complex64) -> Complex64) -> Box<dyn
        WaveFunctionPartWithOp> {

```

```

89     Box::new(PureWkb {
90         wkb: Arc::new(self.wkb.with_op(op)),
91         range: self.range,
92     })
93 }
94 }
95
96 impl Func<f64, Complex64> for PureWkb {
97     fn eval(&self, x: f64) -> Complex64 {
98         self.wkb.eval(x)
99     }
100 }
101
102 #[derive(Clone)]
103 struct ApproxPart {
104     airy: Arc<AiryWaveFunction>,
105     wkb: Arc<WkbWaveFunction>,
106     airy_join_l: Joint,
107     airy_join_r: Joint,
108     range: (f64, f64),
109 }
110
111 impl WaveFunctionPart for ApproxPart {
112     fn range(&self) -> (f64, f64) {
113         self.range
114     }
115     fn as_func(&self) -> Box<dyn Func<f64, Complex64>> {
116         Box::new(self.clone())
117     }
118 }
119
120 impl WaveFunctionPartWithOp for ApproxPart {
121     fn as_wave_function_part(&self) -> Box<dyn WaveFunctionPart> {
122         Box::new(self.clone())
123     }
124
125     fn get_op(&self) -> Box<fn(Complex64) -> Complex64> {
126         self.wkb.get_op()
127     }
128
129     fn with_op(&self, op: fn(Complex64) -> Complex64) -> Box<dyn
130         WaveFunctionPartWithOp> {
131         Box::new(ApproxPart::new(
132             self.airy.with_op(op),
133             self.wkb.with_op(op),
134             self.range,
135         ))
136     }
137 }

```

```

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

```

```

185 }
186
187 fn sign_match(f1: f64, f2: f64) -> bool {
188     return f1.signum() == f2.signum();
189 }
190
191 fn sign_match_complex(mut c1: Complex64, mut c2: Complex64) -> bool {
192     if c1.re.abs() < c1.im.abs() {
193         c1.re = 0.0;
194     }
195
196     if c1.im.abs() < c1.re.abs() {
197         c1.im = 0.0;
198     }
199
200     if c2.re.abs() < c2.im.abs() {
201         c2.re = 0.0;
202     }
203
204     if c2.im.abs() < c2.re.abs() {
205         c2.im = 0.0;
206     }
207
208     return sign_match(c1.re, c2.re) && sign_match(c1.im, c2.im);
209 }
210
211 impl WaveFunction {
212     pub fn get_energy(&self) -> f64 {
213         self.phase.energy
214     }
215
216     pub fn new<F: Fn(f64) -> f64 + Sync + Send>(
217         potential: &'static F,
218         mass: f64,
219         n_energy: usize,
220         approx_inf: (f64, f64),
221         view_factor: f64,
222         scaling: ScalingType,
223     ) -> WaveFunction {
224         let energy = energy::nth_energy(n_energy, mass, &potential, approx_inf);
225         println!("{}Energy: {:.9}", Ordinal(n_energy).to_string(), energy);
226
227         let lower_bound = newtons_method::newtons_method_max_iters(
228             &|x| potential(x) - energy,
229             approx_inf.0,
230             1e-7,
231             100000,
232         );
233         let upper_bound = newtons_method::newtons_method_max_iters(

```

```

234         &|x| potential(x) - energy,
235         approx_inf.1,
236         1e-7,
237         100000,
238     );
239
240     let view = if lower_bound.is_some() && upper_bound.is_some() {
241         (
242             lower_bound.unwrap() - (upper_bound.unwrap() - lower_bound.unwrap())
243                 * view_factor,
244             upper_bound.unwrap() + (upper_bound.unwrap() - lower_bound.unwrap())
245                 * view_factor,
246         )
247     } else {
248         println!("Failed to determine view automatically, using APPROX_INF as view");
249         (
250             approx_inf.0 - f64::EPSILON.sqrt(),
251             approx_inf.1 + f64::EPSILON.sqrt(),
252         )
253     };
254
255     let phase = Arc::new(Phase::new(energy, mass, potential));
256
257     let (airy_wave_funcs, boundaries) = AiryWaveFunction::new(phase.clone(), (
258         view.0, view.1));
259     let (parts, airy_ranges, wkb_ranges): (
260         Vec<Arc<dyn WaveFunctionPart>>,
261         Vec<(f64, f64)>,
262         Vec<(f64, f64)>,
263     ) = if boundaries.ts.len() == 0 {
264         println!("No turning points found in view! Results might be inaccurate")
265         ;
266         let wkb1 = WkbWaveFunction::new(
267             phase.clone(),
268             1.0.into(),
269             INTEG_STEPS,
270             approx_inf.0,
271             approx_inf.0,
272             f64::consts::PI / 4.0,
273         );
274         let wkb2 = WkbWaveFunction::new(
275             phase.clone(),
276             1.0.into(),
277             INTEG_STEPS,
278             approx_inf.0,
279             approx_inf.1,
280             f64::consts::PI / 4.0,
281         );

```

```

278
279     let center = (view.0 + view.1) / 2.0;
280     let wkb1 = Box::new(PureWkb {
281         wkb: Arc::new(wkb1),
282         range: (approx_inf.0, center),
283     });
284
285     let wkb2 = Box::new(PureWkb {
286         wkb: Arc::new(wkb2),
287         range: (center, approx_inf.1),
288     });
289
290     let wkb1_range = wkb1.range();
291     (
292         vec![
293             Arc::from(wkb1.as_wave_function_part()),
294             Arc::from(wkb2.as_wave_function_part()),
295         ],
296         vec![],
297         vec![wkb1_range, wkb2.range()],
298     )
299 } else {
300     let turning_points: Vec<f64> = [
301         vec![2.0 * approx_inf.0 - boundaries.ts.first().unwrap().1],
302         boundaries.ts.iter().map(|p| p.1).collect(),
303         vec![2.0 * approx_inf.1 - boundaries.ts.last().unwrap().1],
304     ]
305     .concat();
306
307     let wave_funcs = turning_points
308         .iter()
309         .zip(turning_points.iter().skip(1))
310         .zip(turning_points.iter().skip(2))
311         .map(
312             |((previous, boundary), next)| -> (WkbWaveFunction, (f64, f64)) {
313                 (
314                     if derivative(phase.potential.as_ref(), *boundary) > 0.0
315                     {
316                         WkbWaveFunction::new(
317                             phase.clone(),
318                             1.0.into(),
319                             INTEG_STEPS,
320                             *boundary,
321                             *previous,
322                             f64::consts::PI / 4.0,
323                         )
324                     } else {
325                         WkbWaveFunction::new(
326                             phase.clone(),

```

```

326                               1.0.into(),
327                               INTEG_STEPS,
328                               *boundary,
329                               *boundary,
330                               f64::consts::PI / 4.0,
331                               )
332                               },
333                               ((boundary + previous) / 2.0, (next + boundary) / 2.0),
334                               )
335                               },
336                               )
337                               .collect::<Vec<(WkbWaveFunction, (f64, f64))>>();
338
339 let wkb_airy_pair: Vec<(&WkbWaveFunction, (f64, f64)), AiryWaveFunction>
340 = wave_funcs
341     .iter()
342     .zip(airy_wave_funcs.iter())
343     .map(|(w, a)| {
344         (
345             w,
346             a.with_phase_off(w.0.phase_off)
347                 .with_c(w.0.get_exp_sign().into()),
348         )
349     })
350     .collect();
351
351 let wkb_ranges = wkb_airy_pair
352     .iter()
353     .map(|((_, wkb_range), _)| *wkb_range)
354     .collect();
355 let airy_ranges = wkb_airy_pair.iter().map(|(_, airy)| airy.ts).collect()
356     ;
357
357 let approx_parts: Vec<Arc<dyn WaveFunctionPartWithOp>> = wkb_airy_pair
358     .iter()
359     .map(|((wkb, range), airy)| -> Arc<dyn WaveFunctionPartWithOp> {
360         Arc::new(ApproxPart::new(airy.clone(), wkb.clone(), *range))
361     })
362     .collect();
363
364     (
365         approx_parts
366             .iter()
367             .map(|p| Arc::from(p.as_wave_function_part()))
368             .collect(),
369         airy_ranges,
370         wkb_ranges,
371     )
372 };

```

```

373
374     match scaling {
375         ScalingType::Mul(s) => WaveFunction {
376             phase,
377             view,
378             parts,
379             airy_ranges,
380             wkb_ranges,
381             scaling: s,
382         },
383         ScalingType::None => WaveFunction {
384             phase,
385             view,
386             parts,
387             airy_ranges,
388             wkb_ranges,
389             scaling: complex(1.0, 0.0),
390         },
391         ScalingType::Renormalize(s) => {
392             let unscaled = WaveFunction {
393                 phase: phase.clone(),
394                 view,
395                 parts: parts.clone(),
396                 airy_ranges: airy_ranges.clone(),
397                 wkb_ranges: wkb_ranges.clone(),
398                 scaling: s,
399             };
400             let factor = renormalize_factor(&unscaled, approx_inf);
401             WaveFunction {
402                 phase,
403                 view,
404                 parts,
405                 airy_ranges,
406                 wkb_ranges,
407                 scaling: s * factor,
408             }
409         }
410     }
411 }
412
413 pub fn calc_psi(&self, x: f64) -> Complex64 {
414     for part in self.parts.as_slice() {
415         if is_in_range(part.range(), x) {
416             return part.eval(x);
417         }
418     }
419     panic!(
420         "[WkbWaveFunction::calc_psi] x_out_of_range(x={:?}", x,
421         x,

```

```

422     self.parts
423         .iter()
424         .map(|p| p.range())
425         .collect::<Vec<(f64, f64)>>()
426     );
427 }
428
429 pub fn get_airy_ranges(&self) -> &[(f64, f64)] {
430     self.airy_ranges.as_slice()
431 }
432
433 pub fn get_wkb_ranges(&self) -> &[(f64, f64)] {
434     self.wkb_ranges.as_slice()
435 }
436
437 pub fn get_wkb_ranges_in_view(&self) -> Vec<(f64, f64> {
438     self.wkb_ranges
439         .iter()
440         .map(|range| {
441             (
442                 f64::max(self.get_view().0, range.0),
443                 f64::min(self.get_view().1, range.1),
444             )
445         })
446         .collect::<Vec<(f64, f64)>>()
447 }
448
449 pub fn is_wkb(&self, x: f64) -> bool {
450     self.wkb_ranges
451         .iter()
452         .map(|r| is_in_range(*r, x))
453         .collect::<Vec<bool>>()
454         .contains(&true)
455 }
456
457 pub fn is_airy(&self, x: f64) -> bool {
458     self.airy_ranges
459         .iter()
460         .map(|r| is_in_range(*r, x))
461         .collect::<Vec<bool>>()
462         .contains(&true)
463 }
464
465 pub fn get_view(&self) -> (f64, f64) {
466     self.view
467 }
468
469 pub fn set_view(&mut self, view: (f64, f64)) {
470     self.view = view

```

```

471     }
472
473     pub fn get_phase(&self) -> Arc<Phase> {
474         self.phase.clone()
475     }
476 }
477
478 impl Func<f64, Complex64> for WaveFunction {
479     fn eval(&self, x: f64) -> Complex64 {
480         self.scaling * self.calc_psi(x)
481     }
482 }
483
484 pub struct SuperPosition {
485     wave_funcs: Vec<WaveFunction>,
486     scaling: Complex64,
487 }
488
489 impl SuperPosition {
490     pub fn new<F: Fn(f64) -> f64 + Send + Sync>(
491         potential: &'static F,
492         mass: f64,
493         n_energies_scaling: &[(usize, Complex64)],
494         approx_inf: (f64, f64),
495         view_factor: f64,
496         scaling: ScalingType,
497     ) -> SuperPosition {
498         let wave_funcs = n_energies_scaling
499             .par_iter()
500             .map(|(e, scale)| {
501                 let wave = WaveFunction::new(
502                     potential,
503                     mass,
504                     *e,
505                     approx_inf,
506                     view_factor,
507                     ScalingType::Mul(*scale),
508                 );
509                 println!("Calculated {} Energy\n", Ordinal(*e).to_string());
510                 return wave;
511             })
512             .collect();
513
514         match scaling {
515             ScalingType::Mul(s) => SuperPosition {
516                 wave_funcs,
517                 scaling: s,
518             },
519             ScalingType::None => SuperPosition {

```

```

520         wave_funcs,
521         scaling: 1.0.into(),
522     },
523     ScalingType::Renormalize(s) => {
524         let unscaled = SuperPosition {
525             wave_funcs: wave_funcs.clone(),
526             scaling: s,
527         };
528         let factor = renormalize_factor(&unscaled, approx_inf);
529         println!("factor:{} {}", factor);
530         SuperPosition {
531             wave_funcs,
532             scaling: s * factor,
533         }
534     }
535 }
536 }
537
538 pub fn get_view(&self) -> (f64, f64) {
539     let view_a = self
540         .wave_funcs
541         .iter()
542         .map(|w| w.get_view().0)
543         .min_by(cmp_f64)
544         .unwrap();
545     let view_b = self
546         .wave_funcs
547         .iter()
548         .map(|w| w.get_view().1)
549         .max_by(cmp_f64)
550         .unwrap();
551     (view_a, view_b)
552 }
553 }
554
555 impl Func<f64, Complex64> for SuperPosition {
556     fn eval(&self, x: f64) -> Complex64 {
557         self.scaling * self.wave_funcs.iter().map(|w| w.eval(x)).sum::<Complex64>()
558     }
559 }
560
561 struct Scaled<A, R>
562 where
563     R: std::ops::Mul<R, Output = R> + Sync + Send + Clone,
564 {
565     scale: R,
566     func: Box<dyn Func<A, R>>,
567 }
568

```

```

569 impl<A, R> Func<A, R> for Scaled<A, R>
570 where
571     R: std::ops::Mul<R, Output = R> + Sync + Send + Clone,
572 {
573     fn eval(&self, x: A) -> R {
574         self.func.eval(x) * self.scale.clone()
575     }
576 }
577
578 fn renormalize_factor(wave_func: &dyn Func<f64, Complex64>, approx_inf: (f64, f64))
579     -> f64 {
580     let area = integrate(
581         evaluate_function_between(
582             wave_func,
583             approx_inf.0 * (1.0 - f64::EPSILON),
584             approx_inf.1 * (1.0 - f64::EPSILON),
585             INTEG_STEPS,
586         )
587         .par_iter()
588         .map(|p| Point {
589             x: p.x,
590             y: p.y.norm_sqr(),
591         })
592         .collect(),
593         TRAPEZE_PER_THREAD,
594     );
595
596     let area = if area == 0.0 {
597         println!("Can't renormalize, area under Psi is 0.");
598         1.0
599     } else {
600         area
601     };
602
603     1.0 / area
604 }
605
606 pub fn renormalize(
607     wave_func: Box<dyn Func<f64, Complex64>>,
608     approx_inf: (f64, f64),
609 ) -> Box<dyn Func<f64, Complex64>> {
610     let area = renormalize_factor(wave_func.as_ref(), approx_inf);
611     return Box::new(Scaled::<f64, Complex64> {
612         scale: area.into(),
613         func: wave_func,
614     });
615 }
616 #[cfg(test)]

```

```

617 mod test {
618     use super::*;

619
620     extern crate test;
621     use test::Bencher;
622
623     #[test]
624     fn sign_check_complex_test() {
625         let range = (-50.0, 50.0);
626         let n = 100000;
627         for ril in 0..n {
628             for ii1 in 0..n {
629                 for ri2 in 0..n {
630                     for ii2 in 0..n {
631                         let re1 = index_to_range(ril as f64, 0.0, n as f64, range.0,
632                                         range.1);
633                         let im1 = index_to_range(ii1 as f64, 0.0, n as f64, range.0,
634                                         range.1);
635                         let re2 = index_to_range(ri2 as f64, 0.0, n as f64, range.0,
636                                         range.1);
637                         let im2 = index_to_range(ii2 as f64, 0.0, n as f64, range.0,
638                                         range.1);

639                         assert_eq!(
640                             sign_match_complex(complex(re1, im1), complex(re2, im2)),
641                             sign_match_complex(complex(re2, im2), complex(re1, im1))
642                         );
643                     }
644                 }
645             }
646         }
647         #[bench]
648         fn renormalize_square(b: &mut Bencher) {
649             let square = Function::new(|x: f64| complex(x*x, 0.0));
650
651             b.iter(||{
652                 let bounds = test::black_box((-10.0, 10.0));
653                 let _ = test::black_box(renormalize_factor(&square, bounds));
654             });
655         }
656     }

```

src/wkb_wave_func.rs

```

1 use crate::*;
2 use std::fmt::Display;
3 use std::sync::Arc;

```

```

4
5 #[derive(Clone)]
6 pub struct Phase {
7     pub energy: f64,
8     pub mass: f64,
9     pub potential: Arc<dyn Fn(f64) -> f64 + Send + Sync>,
10 }
11
12 impl Display for Phase {
13     fn fmt(&self, f: &mut std::fmt::Formatter<'_>) -> std::fmt::Result {
14         write!(
15             f,
16             "Phase{{energy:{}, mass:{}, potential:{}{}}}",
17             self.energy, self.mass
18         )
19     }
20 }
21
22 impl Phase {
23     fn default() -> Phase {
24         Phase {
25             energy: 0.0,
26             mass: 0.0,
27             potential: Arc::new(|x| 0.0),
28         }
29     }
30 }
31 pub fn new<F: Fn(f64) -> f64 + Sync + Send>(
32     energy: f64,
33     mass: f64,
34     potential: &'static F,
35 ) -> Phase {
36     return Phase {
37         energy,
38         mass,
39         potential: Arc::new(potential),
40     };
41 }
42
43     fn sqrt_momentum(&self, x: f64) -> f64 {
44         self.eval(x).abs().sqrt()
45     }
46 }
47
48 impl Func<f64, f64> for Phase {
49     fn eval(&self, x: f64) -> f64 {
50         (2.0 * self.mass * ((self.potential)(x) - self.energy))
51             .abs()
52             .sqrt()

```

```

53     }
54 }
55
56 #[derive(Clone)]
57 pub struct WkbWaveFunction {
58     pub c: Complex64,
59     pub turning_point_exp: f64,
60     pub turning_point_osc: f64,
61     pub phase: Arc<Phase>,
62     integration_steps: usize,
63     op: fn(Complex64) -> Complex64,
64     pub phase_off: f64,
65 }
66
67 impl WkbWaveFunction {
68     pub fn get_c(&self) -> Complex64 {
69         self.c
70     }
71
72     pub fn with_c(&self, c: Complex64) -> WkbWaveFunction {
73         WkbWaveFunction {
74             c,
75             turning_point_exp: self.turning_point_exp,
76             turning_point_osc: self.turning_point_osc,
77             phase: self.phase.clone(),
78             integration_steps: self.integration_steps,
79             op: self.op,
80             phase_off: self.phase_off,
81         }
82     }
83
84     pub fn new(
85         phase: Arc<Phase>,
86         c: Complex64,
87         integration_steps: usize,
88         turning_point_exp: f64,
89         turning_point_osc: f64,
90         phase_off: f64,
91     ) -> WkbWaveFunction {
92         return WkbWaveFunction {
93             c,
94             turning_point_exp,
95             turning_point_osc,
96             phase: phase.clone(),
97             integration_steps,
98             op: identity,
99             phase_off,
100        };
101    }

```

```

102
103 pub fn with_op(&self, op: fn(Complex64) -> Complex64) -> WkbWaveFunction {
104     return WkbWaveFunction {
105         c: self.c,
106         turning_point_exp: self.turning_point_exp,
107         turning_point_osc: self.turning_point_osc,
108         phase: self.phase.clone(),
109         integration_steps: self.integration_steps,
110         op,
111         phase_off: self.phase_off,
112     };
113 }
114
115 pub fn get_op(&self) -> Box<fn(Complex64) -> Complex64> {
116     Box::new(self.op)
117 }
118
119 pub fn get_exp_sign(&self) -> f64 {
120     let limit_sign = if self.turning_point_exp == self.turning_point_osc {
121         1.0
122     } else {
123         -1.0
124     };
125
126     (self.psi_osc(self.turning_point_exp + limit_sign * f64::EPSILON.sqrt()) /
127      self.c)
128         .re
129         .signum()
130 }
131
132 fn psi_osc(&self, x: f64) -> Complex64 {
133     let integral = integrate(
134         evaluate_function_between(
135             self.phase.as_ref(),
136             x,
137             self.turning_point_osc,
138             self.integration_steps,
139         ),
140         TRAPEZE_PER_THREAD,
141     );
142     self.c * complex((integral + self.phase_off).cos(), 0.0) / self.phase.
143         sqrt_momentum(x)
144 }
145
146 fn psi_exp(&self, x: f64) -> Complex64 {
147     let integral = integrate(
148         evaluate_function_between(

```

```

149         self.turning_point_exp,
150         self.integration_steps,
151     ),
152     TRAPEZE_PER_THREAD,
153 );
154 let exp_sign = self.get_exp_sign();
155
156     exp_sign * (self.c * 0.5 * (-integral.abs()).exp())
157 }
158 }
159
160 impl Func<f64, Complex64> for WkbWaveFunction {
161     fn eval(&self, x: f64) -> Complex64 {
162         let val = if self.phase.energy < (self.phase.potential)(x) {
163             self.psi_exp(x)
164         } else {
165             self.psi_osc(x)
166         };
167
168         return (self.op)(val);
169     }
170 }
171
172 #[cfg(test)]
173 mod test {
174     use super::*;

175     use std::cmp::Ordering;
176
177     fn pot(x: f64) -> f64 {
178         1.0 / (x * x)
179     }
180
181     fn pot_in(x: f64) -> f64 {
182         1.0 / x.sqrt()
183     }
184
185     #[test]
186     fn phase_off() {
187         let energy_cond = |e: f64| -> f64 { (0.5 * (e - 0.5)) % 1.0 };
188
189         let integ = Function::<f64, f64>::new(energy_cond);
190         let mut values = evaluate_function_between(&integ, 0.0, 5.0, NUMBER_OF_POINTS
191             );
192         let sort_func =
193             |p1: &Point<f64, f64>, p2: &Point<f64, f64>| -> Ordering { cmp_f64(&p1.x,
194                 &p2.x) };
195         values.sort_by(sort_func);
196
197         let mut data_file = File::create("energy.txt").unwrap();

```

```
196     let data_str: String = values
197         .par_iter()
198         .map(|p| -> String { format!("{}{}{}\n", p.x, p.y) })
199         .reduce(|| String::new(), |s: String, current: String| s + &*current);
200
201     data_file.write_all((data_str).as_ref()).unwrap()
202 }
203 }
```

lib/build.sh

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

lib/go.mod

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

lib/main.go

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

build.rs

```
1 use std::env;
2 use std::path::PathBuf;
3 use std::process::Command;
4
```

```

5 fn main() {
6     Command::new("sh")
7         .arg("build.sh")
8         .current_dir("./lib/")
9         .status()
10        .unwrap();
11
12    let path = "./lib";
13    let lib = "airy";
14
15    println!("cargo:rustc-link-search=native={}, path");
16    println!("cargo:rustc-link-lib=static={}, lib");
17
18    // The bindgen::Builder is the main entry point
19    // to bindgen, and lets you build up options for
20    // the resulting bindings.
21    let bindings = bindgen::Builder::default()
22        // The input header we would like to generate
23        // bindings for.
24        .header("lib/libairy.h")
25        // Tell cargo to invalidate the built crate whenever any of the
26        // included header files changed.
27        .parse_callbacks(Box::new(bindgen::CargoCallbacks))
28        // Finish the builder and generate the bindings.
29        .generate()
30        // Unwrap the Result and panic on failure.
31        .expect("Unable to generate bindings");
32
33    // Write the bindings to the $OUT_DIR/bindings.rs file.
34    let out_path = PathBuf::from(env::var("OUT_DIR").unwrap());
35    bindings
36        .write_to_file(out_path.join("bindings.rs"))
37        .expect("Couldn't write bindings!");
38 }

```

Cargo.toml

```

1 [package]
2 name = "schroedinger_approx"
3 version = "0.1.0"
4 edition = "2021"
5
6 # See more keys and their definitions at https://doc.rust-lang.org/cargo/reference/manifest.html
7
8 [dependencies]
9 num = "0.4.0"
10 rayon = "1.5.3"
11 ordinal = "0.3.1"

```

```

12 duplicate = "0.4.1"
13 paste = "1.0.9"
14
15 [build-dependencies]
16 bindgen = "0.60.1"

```

energy.wsl

```

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

```

exact.wsl

```

1 c1 = 1.0
2 c2 = 0.0
3 numberofPoints = 10000
4 m = 1
5 n = 5
6 viewFactor = 1.5
7
8 V[x_] := x^2
9
10 energys = Solve[Integrate[Sqrt[2*m*(en - V[x])], {x, -Sqrt[en], Sqrt[en]}] == Pi*(n +
11   1/2), en] // N
12 energy = en /. energys[[1]]
13
14 view = Solve[energy == V[x], x]
15 view = Function[l, x /. l] /@ view
16 view = Function[x, x*viewFactor] /@ view
17
18 Print["Energy = ", energy]
19 Print["view = ", view]
20
21
22 solution := DSolve[{V[x] psi[x] - psi''[x]/(2 m) == energy psi[x]}, psi[x], x]

```

```

23 psi[x_] = psi[x] /. solution[[1]] /. C[1] -> c1 /. C[2] -> c2
24
25 Print["psi[x] = ", psi[x]]
26
27 (*psi[x_] = c2*ParabolicCylinderD[(-1 - 50*.Sqrt[m])/2, *)
28 (*I*2^(3/4)*m^(1/4)*x] + c1*ParabolicCylinderD[(-1 + 50*.Sqrt[m])/2, *)
29 (*2^(3/4)*m^(1/4)*x]*)
30
31
32
33 step = (Abs[view[[1]]] + Abs[view[[2]]]) / numberOfPoints
34
35
36 vals = Table[{x, N@psi[x]}, {x, view[[1]], view[[2]], step}]
37 vals = Function[p, {p[[1]], Re[p[[2]]], Im[p[[2]]]}] /@ vals
38 Print["psi[0] = ", psi[0]]
39
40 total = N@Integrate[Re[psi[x]]^2 + Im[psi[x]]^2, {x, -Sqrt[energy], Sqrt[energy]}]
41
42 Print["area under solution = ", total]
43 total = N@Integrate[Abs[psi[x]], {x, -Sqrt[energy], Sqrt[energy]}]
44 Print["area under solution after renormalization = ", N@Integrate[Re[psi[x]]^2 + Im[
45     psi[x]]^2, {x, -Sqrt[energy], Sqrt[energy]}]]
46
47 vals = Function[p, {p[[1]], p[[2]] / total, p[[3]] / total}] /@ vals
48 csv = ExportString[vals, "CSV"]
49 csv = StringReplace[csv, "," -> " "]
50 Export["output/exact.dat", csv]

```

Bildquellen

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

Bibliography

- CODATA. CODATA Value: Planck Length. <https://physics.nist.gov/cgi-bin/cuu/Value?plkl>, 2022a.
- CODATA. CODATA Value: Planck Mass. <https://physics.nist.gov/cgi-bin/cuu/Value?plkm>, 2022b.
- CODATA. CODATA Value: Planck Time. <https://physics.nist.gov/cgi-bin/cuu/Value?plkt>, 2022c.
- Bryce Seligman DeWitt und Neill Graham. *The many-worlds interpretation of quantum mechanics*, volume 63. Princeton University Press, 2015.
- Espen Gaarder Haug. The gravitational constant and the Planck units. A simplification of the quantum realm. *Physics Essays*, 29(4):558–561, 2016.
- Brain C. Hall. *Quantum Theory for Mathematicians*. Springer New York, NY, 1 edition, 2013. ISBN 978-1461471158.
- Christopher Kormanyos John Maddock. Calculating a Derivative - 1.58.0. https://www.boost.org/doc/libs/1_58_0/libs/multiprecision/doc/html/boost_multiprecision/tut/floats/fp_2022.html.
- Linux Kernel Organization. Perf Manual Page, April 2022. URL [url{https://man.archlinux.org/man/perf.1}](https://man.archlinux.org/man/perf.1). [Copy, Online; accessed 20-November-2022].
- Robert G. Littlejohn. Physics 221A, 2020. URL [url{https://www.pas.rochester.edu/~passage/resources/prelim/Quantum/UCB%20Notes/7%20wkb.pdf}](https://www.pas.rochester.edu/~passage/resources/prelim/Quantum/UCB%20Notes/7%20wkb.pdf).
- rayon rs. Rayon, November 2022. URL [url{https://github.com/rayon-rs/rayon}](https://github.com/rayon-rs/rayon). [Online; accessed 23-November-2022].
- Erwin Schrödinger. Die gegenwärtige Situation in der Quantenmechanik. *Naturwissenschaften*, 23, 1935.
- Tanja Van Mourik, Michael Bühl, und Marie-Pierre Gaigeot. Density functional theory across chemistry, physics and biology, 2014.
- Eric W. Weisstein. Newton's Method, 2022. URL <https://mathworld.wolfram.com/NewtonMethod.html>. [Online; accessed 10-August-2022].
- Wikipedia. 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/}](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