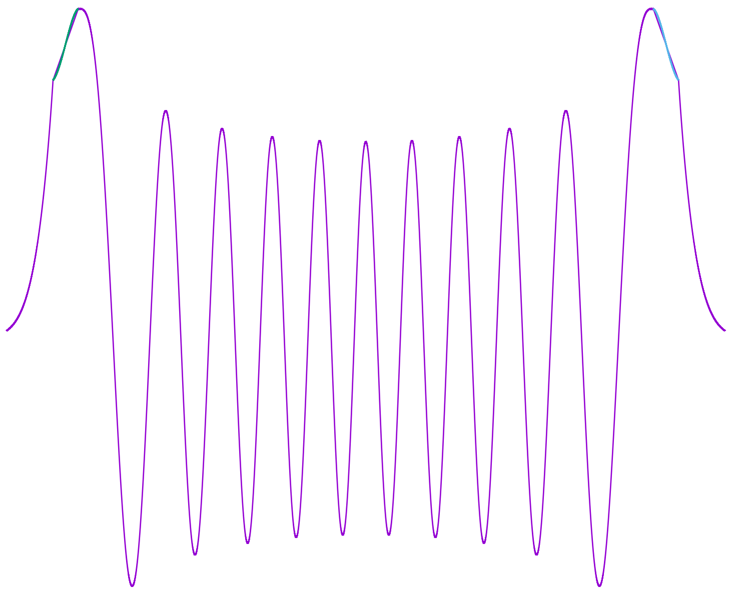
Approximating Solutions of the time independent Schrödinger equation

Gian Laager

September 7, 2022



Maturaarbeit

Kantonsschule Glarus

Betreuer: Linus Romer

Referent: Beat Temperli

Contents

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 certaint. Even Schrödinger the inventor of the equation that governs all of those weird phenomena rejected the idea that there a just probabilities.

In this paper we will try to understand this world a little bit better by looking at wave functions in a simplified universe. This universe only has 1 dimension and there will not be any sense of time. This means we will be able to actually see how the wave function looks like in a graph. In our universe it’s a little bit more complicated since humans unfortunately can’t think about 6 dimension.

2. Preliminary

2.1. Schrödinger Equation

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

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

Or more general

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

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

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

or

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

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

2.2. Rust

Rust is one of the newer programming languages and attempts to replace C/C++ which are notoriously difficult to work with. It supports both functional and object-oriented para dimes. 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.

Personally I like programming languages that support functional para dimes when I'm programming something that uses a lot of math since functional programming languages are designed according to mathematical concepts that also govern the problem. This usually results in rather neat solutions.

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.

3. Methods

3.1. Approximation Scheme

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

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

The program uses the third method WKB approximation, it is applicable to a wide verity of linear differential equations and works very well in the case of the Schrödinger equation. Originally it was developed by Wentzel, Kramers and Brillouin in 1926. It gives an approximation to the eigenfunctions of the Hamiltonian $\hat{H}$ in one dimension. The approximation is best understood as applying to a fixed range of energies as $\hbar$ tends to zero [**?** , p. 305]. This is not a physically correct explanation because one can not assume that $\hbar$ is small since it has units, but there is another interpretation that is physically more valid that will be discussed in section 3.1.2.

3.1.1. Example

This example is from [**?** , An example].

Lets solve the ordinary differential equation

$$\epsilon^2 \frac{d^2 y}{dx^2} = Q(x) y$$

where $Q(x)$ is an arbitrary function that is not $Q(x) = 0$ and ϵ is small. Note that this example relates to the Schrödinger equation where $\epsilon = -\frac{\hbar^2}{2m}$ and $Q(x) = E - V(x)$.

Replace y with

$$y(x) = \exp\left(\frac{1}{\delta} \sum_{n=0}^{\infty} \delta^n S_n(x)\right)$$

resulting in the equation

$$\epsilon^2 \frac{1}{\delta^2}\left(\sum_{n=0}^{\infty} \delta^n S_n'(x) + \frac{1}{\delta}\left(\sum_{n=0}^{\infty} \delta^n S_n''(x)\right)\right) = Q(x) \tag{3.1}$$

As an approximation set the upper bound of the sum to be 2 instead of ∞, then 3.1 can be written as

$$\frac{\epsilon^2}{\delta^2} S_0'^2 + \frac{2\epsilon^2}{\delta} S_0' S_1' + \frac{\epsilon^2}{\delta} S_0'' = Q(x) \tag{3.2}$$

If we go further and solve for S_0 and S_1 we will get that

$$S_0(x) = \pm \int_{x_0}^{x} \sqrt{Q(t)}\, dt \tag{3.3}$$

$$S_1(x) = -\frac{1}{4} \ln Q(x) + k_1 \tag{3.4}$$

this means that our approximate solution of y is

$$y(x) \approx c_1 Q^{-\frac{1}{4}}(x) \exp\left(\frac{1}{\epsilon}\int_{x_0}^{x} \sqrt{Q(t)}\, dt\right) + c_2 Q^{-\frac{1}{4}}(x) \exp\left(-\frac{1}{\epsilon}\int_{x_0}^{x} \sqrt{Q(t)}\, dt\right)$$

here c_1, c_2 depend on the boundary conditions and can also be used to renormalize the function. x_0 will later be discussed in section ***(...)***. It is important to have the right value for x_0 which depends on $V(x)$ and E although I have not yet manged to find out the connection.

3.1.2. Validity

Dieser abschnitt ist momentan nur fürs Probe Kapitel muss ich noch besser nach Rechnen und genau verstehen

The approximation that was made in equation 3.1 to 3.2 that we ignored the terms of $n > 2$ and assumed that they are small.

$$\epsilon \left| \frac{dQ}{dx} \right| \ll Q(x)^2$$

Without the assumption that $\hbar$ is infact *small* the equation can also be interpreted such that the potential $V(x)$ has to vary slowly compared to $(V(x) - E)^2$.

During calculations I noticed that it is crucial what values x_0 has sinse some times the results can not be valid.

3.2. Newtons Method

Newton's method, also called the Newton-Raphson method, is a root-finding algorithm that uses the first few terms of the Taylor series of a function $f(x)$ in the vicinity of a suspected

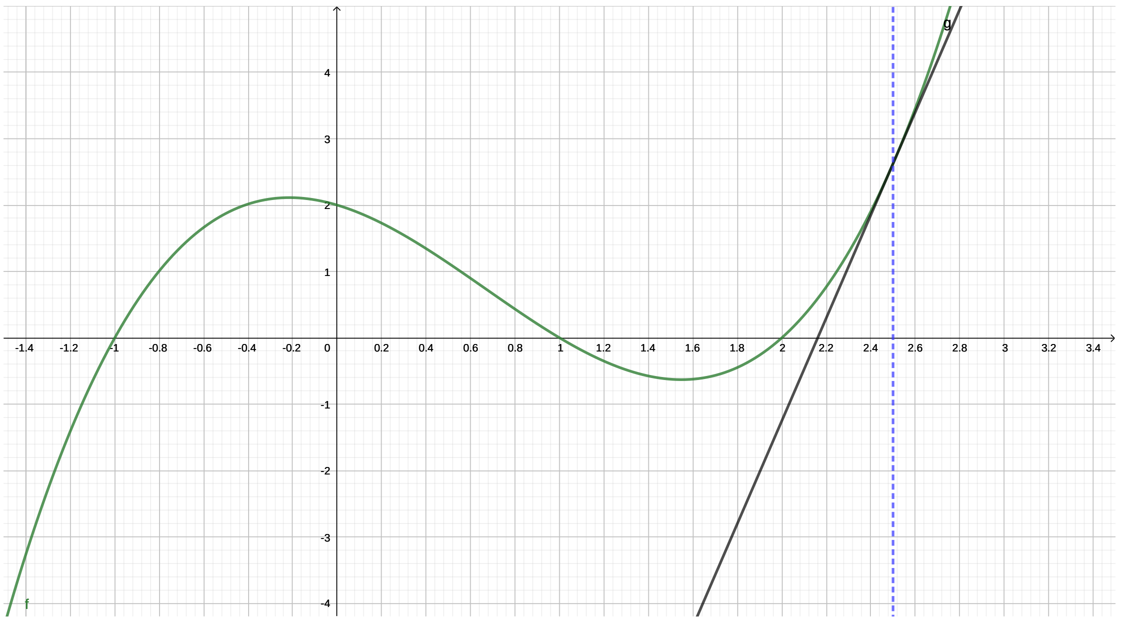


Figure 3.1.: Illustration of Newtons method, $f(x) = (x-1)(x+1)(x-2)$.

root [**?**]. It makes a sequence of approximations of a root x_n that in sure tent cases converges to the exact value where

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

The sequence is defined as

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

Visually this looks like figure 3.1 $f(x) = (x-1)(x+1)(x-2)$. The blue line indicates the initial guess which in this case is 2.5 the black line ($g(x)$) is a tangent to $f(x)$ at $(guess, f(guess))$ the next guess will be where the tangent intersects the x-Axis (solution of $g(x) = 0$). This will converge rather quickly compared to other methods such as Regula-Falsi.

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

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

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

3.3. Derivatives

The precision gained by calculating derivatives analytically would come with a massive overhead both in development and in performance, one would have to implement all the rules of differentiation and create some representation of an actual function to which those rules could be applied. I wrote an implementation of this in Go and not much accuracy can actually be gained.

A much easier approach is to approximate it. The definition of a derivative of a function $f(x)$ is

$$\frac{df}{dx} = \lim_{h \to 0} \frac{f(x+h) - f(x)}{h}$$

The limit can very easily be approximated with

$$\frac{f(x + \frac{\delta}{2}) - f(x - \frac{\delta}{2})}{\delta}$$

were δ needs to be small yet not so small that it exceeds the 16 digit precision of double. I chose $\delta = \sqrt{2^{-52}}$, because $\epsilon = 2^{-52}$ is the smallest double precision floating point number where $1 + \epsilon \neq 1$. The square root ensures that still enough precision remains. After some further research the function should be implemented the same way as described by Boost (C++ library).

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

`f64::epsilon()` this is the difference between '1.0' and the next larger representable number [?]. `derivative` is implemented not only for `f64` but for all types that support subtraction and division by `f64` this means it can also be used for closures with real inputs and complex outputs.

3.4. Integration

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

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

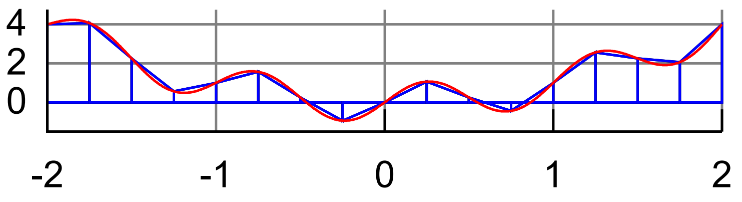


Figure 3.2.: Illustration of integration with trapeze from [?].

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

    let mut rest = complex(0.0, 0.0);

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

    return parallel + rest;
}
```

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

A. Source Code

At the moment the code is only available on `https://github.com/Gian-Laager/Schroedinger-Approximation`

B. Detailed Calculations

Bildquellen

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

Bibliography

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