Exercise sheet 4

by Robin Heinemann (group 4), Paul Rosendahl (group 4)

July 21, 2018

1 Neutrons in the gravitional field

the time-independent Schödinger equation reads

$$\psi''(z) + \frac{2m}{\hbar^2}(E - V(z))\psi(z) = 0$$

For the gravitational field we can use $V(z) = mgz$ for small changes in z. This results in

$$\psi''(z) + \frac{2m}{\hbar^2}(E - mgz)\psi(z) = 0$$

Using dimensionless coordinate x and energy ε

$$x = \left(\frac{2m^2 g}{\hbar^2}\right)^{1/3} z$$
$$\varepsilon = \left(\frac{2}{g^2\hbar^2 m}\right)^{1/3} E$$

The Schrödinger equation reads as

$$\psi''(x) + (\varepsilon - x)\psi(x) = 0$$

The Numerov algorithm can be easily used to solve this. The infinite potential for $z < 0$ means the solution must be zero at $z = 0$ and the antisymmetric starting condition must be used.

```
reset
set xrange[0:5]
set yrange[-1000:1000]
plot for [idx=0:1] "data" i idx u 1:2 w l title columnheader(1)
```

Listing 1: Gnuplot code for plotting the wave function

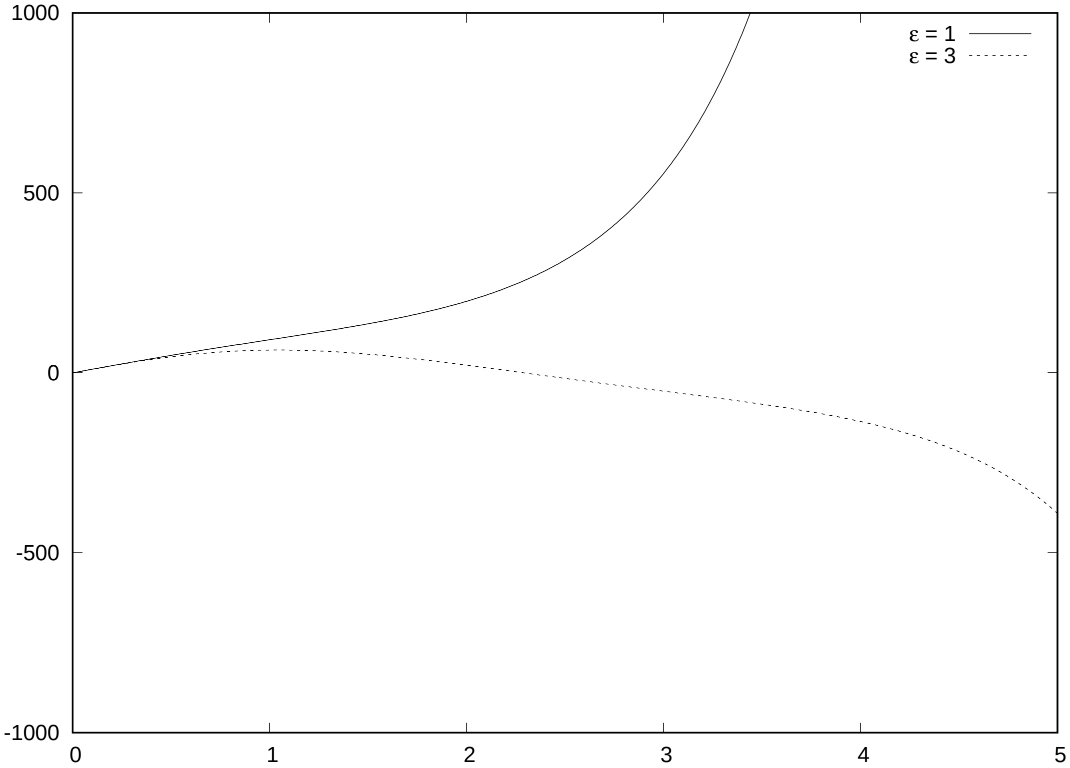

Figure 1: Plot of the wave function for two different ε, one goes towards infinity as x approaches infinity and one goes towards negative infinity

The plot only shows the wave function for small x, to make the structures with small amplitudes visible.

The eigenvalues could be found using a simple linear scan with and stepsize of the accuracy wanted, but is more efficiently implemented with a binary search. This makes it possible to achieve really accurate values for the eigenvalues which just a few calculations of the wavefunction. Table 1 shows the first eight eigenvalues to ten digits accuracy.

2 Rust implementation

```rust
use std::fs::File;
use std::path::Path;
use std::io::Write;

fn hermite(n: u64, x: f64) -> f64 {
    match n {
    0 => 1.0,
    1 => 2.0 * x,
    2 => 4.0 * x * x - 2.0,
    3 => 8.0 * x * x * x - 12.0 * x,
    4 => 16.0 * x * x * x * x - 48.0 * x * x + 12.0,
    _ => 2.0 * x * hermite(n - 1, x) - 2.0 * (n as f64) * hermite(n - 2, x)
    }
}

```

```
fn numerov(k: &Fn(f64) -> f64, h: f64, n: usize, x0: f64, a: f64, symmetric: bool) -> Vec<f64> {
    let mut y = Vec::new();
    let mut i = 2;

    let f = | i | {
    (1.0 / 12.0) * h * h * k(x0 + (i as f64) * h)
    };

    if symmetric {
    y.push(a);
    let yn = y[0] - h * h * k(0.0) * y[0] / 2.0;
    y.push(yn);
    } else {
    y.push(0.0);
    y.push(a);
    }

    let mut kn0 = f(0);
    let mut kn1 = f(1);

    while i < n {
    let kn2 = f(i);
    let yn = 2.0 * (1.0 - 5.0 * kn1) * y[i - 1] - (1.0 + kn0) * y[i - 2];

    y.push(yn / (1.0 + kn2));

    kn0 = kn1;
    kn1 = kn2;
    i += 1;
    }

    y
}

fn harmonic_oscillator(x: f64, eps: f64) -> f64 {
    2.0 * eps - x * x
}

fn gravitation(x: f64, eps: f64) -> f64 {
    if x < 0.0 {
    return 1e308;
    } else {
    eps - x
    }
}

fn factorial(x: u64) -> u64 {
    let mut f = 1;

```

```
    for n in 1..(x + 1) {
    f *= n;
    }

    f
}

fn analytic_harmonic_oscillator(n: u64, x: f64) -> f64 {
    (- x * x / 2.0).exp() * hermite(n, x) /
    (2.0_f64.powf(n as f64) * (factorial(n) as f64) * std::f64::consts::PI.sqrt())
}

fn main() {
    let path = Path::new("data");
    let mut file = File::create(&path).unwrap();


    for n in 1..3 {
    let h = 0.01;
    let ns = (100.0 / h) as u64;
    let ygrid = numerov(&| x | { gravitation(x, (n as f64) * 2.0 - 1.0) }
                , h , ns as usize, 0.0, 1.0, false);


    let mut x = 0.0;


    file.write_all(format!("\"{{/Symbol e}} = {}\"\n", (n as f64) * 2.0 - 1.0).as_bytes()).unwrap();
    for y in ygrid {
        file.write_all(format!("{}, {}\n", x, y).as_bytes()).unwrap();
        x += h;
    }

    file.write_all("\n\n".as_bytes()).unwrap();
    }

    let mut e = Vec::new();
    e.push(find_eigenvalue(0.01, 0.1));

    for i in 0..7 {
    let last_e = e[i];
    e.push(find_eigenvalue(last_e, 0.1));
    }


    println!("|$n$|$\\eps_n$|");
    println!("|-");

```

```
    for i in 0..e.len() {
    println!("| {} | {} |", i, e[i]);
    }
}

fn find_eigenvalue(eps0: f64, delta_eps: f64) -> f64 {
    let mut eps = eps0;
    let h = 0.01;
    let ns = (eps * 10.0 / h) as usize;
    let ygrid = numerov(&| x | { gravitation(x, eps) }
                , h , ns as usize, 0.0, 1.0, false);

    let first = ygrid[ns - 1];

    loop {
    eps += delta_eps;
    let ns = (eps * 10.0 / h) as usize;
    let ygrid = numerov(&| x | { gravitation(x, eps) }
                , h , ns as usize, 0.0, 1.0, false);

    if first * ygrid[ns - 1] < 0.0 { break; }
    }



    let mut a = eps - delta_eps;
    let mut b = eps;

    let mut old_eps = a;
    let mut old_value = first;

    loop {
    let c = (a + b) / 2.0;

    let ns = (c * 10.0 / h) as usize;
    let ygrid = numerov(&| x | { gravitation(x, c) }
                , h , ns as usize, 0.0, 1.0, false);

    if (old_eps - c).abs() < 1e-10 {
        break;
    }

    if ygrid[ns - 1] * old_value > 0.0  {
        a = c;
    } else {
        b = c;
    }

    old_eps = c;
    }
```

```

    b
}
```

Listing 2: rust implementation of the runge-kutta-4 integrator and application to the three-body problem

Table 1: first eight eigenvalues of the $V(z) = mgz$ potential to ten digits accuracy

n	ε_n
0	2.3381074104830626
1	4.087949443832042
2	5.520559827275572
3	6.786708088442677
4	7.944133584238579
5	9.022650849111365
6	10.040174335651084
7	11.008524295911172

3 Additional notes

All programs written are written using the programming language *rust*. Extra dependencies (*rust crates*) will be listed in a comment in the first line. To get the source files of each program just unzip this *pdf* file. You will find directories for every program in this file. To execute one of the programs run `cargo run` in it's directory. All plots are made with *gnuplot*. This document was written in *org-mode* and converted to *pdf*. The corresponding *org-mode* sources can also be found by unzipping this *pdf* file.