Exercise sheet 2

by Robin Heinemann (group 4), Paul Rosendahl (group 2) and Andreas Rall (group 1)

February 1, 2019

1 Numerical simulation of the two-body-problem

Let $\vec{r}$ denote the seperation of two point masses with mass M_1 and M_2. Using Newton's equations

$$\ddot{\vec{r}} = -\frac{G(M_1 + M_2)}{r^3}\frac{\vec{r}}{r}$$

is obtained. This differential equation can be solved using the Euler integration scheme. $\vec{r}, \dot{\vec{r}} = \vec{v}$ and t are transformed to characteristic scales $\vec{s}$, $\vec{\omega}$ and τ by

$$\begin{aligned}
\vec{s} &= \frac{\vec{r}}{R_0} \\
\vec{\omega} &= \vec{v}\left(\frac{R_0}{GM}\right)^{1/2} \\
\tau &= t\left(\frac{GM}{R_0^3}\right)^{1/2}
\end{aligned}$$

with R_0 being the initial seperation of the two masses. The forward Euler scheme then can be written as

$$\begin{aligned}
\vec{s}_i &= \vec{s}_{i-1} + \vec{\omega}_{i-1}h \\
\vec{\omega}_i &= \vec{\omega}_{i-1} - \frac{\vec{s}_{i-1}}{s_{i-1}^3}h
\end{aligned}$$

where $\vec{s}_i$ and $\vec{\omega}_i$ denotes the characteristic seperation and characteristic velocity at time τ_i and $h := \tau_i - \tau_{i-1}$. The velocity $\vec{0}$ for with orbit is circular can be determined from the Laplace-Runge-Lenz vector $\vec{e}$ which is related to eccentricity e by $|\vec{e}| = e$. A circular orbit means $e = 0 \implies \vec{e} = 0$. In addition to that the velocity must be perpendicular to $\vec{r}$.

$$\begin{aligned}
\vec{e} &= \frac{\vec{v} \times \vec{j}}{GM} - \frac{\vec{r}}{r} \stackrel{!}{=} 0 \\
&= \frac{\vec{v} \times (\vec{r} \times \vec{v})}{GM} - \frac{\vec{r}}{r} \\
&= \frac{(\vec{v} \cdot \vec{v})\vec{r} - (\vec{v} \cdot \vec{r})\vec{v}}{GM} - \frac{\vec{r}}{r}
\end{aligned}$$

the term $(\vec{r} \cdot \vec{r})$ must be zero because $\vec{r} \perp \vec{v}$

$$\begin{aligned}
&= \frac{v^2\vec{r}}{GM} - \frac{\vec{r}}{r} = 0 \\
\implies v &= \sqrt{\frac{GM}{r}}
\end{aligned}$$

Because $|\vec{r}| = r$ must be constant for a circular orbit, finally

$$v = \sqrt{\frac{GM}{R_0}}$$

is obtained.

```rust
// cargo-deps: rulinalg
#[macro_use]
extern crate rulinalg;

use std::fs::File;
use std::path::Path;
use std::io::Write;
use rulinalg::vector::Vector;
use rulinalg::norm::Euclidean;

const G: f64 = 1.0;


fn forward_euler(s: &Vector<f64>, w: &Vector<f64>, dt: f64) -> (Vector<f64>, Vector<f64>) {
      let sn = s + w * dt;
      let sa = s.norm(Euclidean);
      let wn = w - s * dt / (sa * sa * sa);

      (sn, wn)
}

fn normalize(M: f64, r: Vector<f64>, v: Vector<f64>, h: f64, R0: f64) -> (Vector<f64>, Vector<f64>, f64) {
      let s = r / R0;
      let V0 =  ((G * M) / R0).sqrt();
      let w = v / V0;
      let dt = h * V0 / R0;

      (s, w, dt)
}

fn cross(a: &Vector<f64>, b: &Vector<f64>) -> Vector<f64> {
      let mut c = vector![0.0, 0.0, 0.0];
      c[0] = a[1] * b[2] - a[2] * b[1];
      c[1] = a[2] * b[0] - a[0] * b[2];
      c[2] = a[0] * b[1] - a[1] * b[0];
      c
}

fn euler(dt_0: f64, v0: f64, file: &str,  interval: Option<u64>) {
      let M1: f64 = 1.0;
      let M2: f64 = 1.0;
```

```
42        let R0: f64 = 1.0;
43        let r = vector![1.0, 0.0, 0.0];
44        let v = vector![0.0, v0, 0.0];
45        let te: f64 = 100.0;
46        let h: f64 = 0.01;
47        let mut t = 0.0;
48        let interval =
49        match interval {
50            Some(i) => i,
51            None => 1
52        };
53
54        let (mut s, mut w, mut dt) = normalize(M1 + M2, r, v, h, R0);
55        dt = dt_0;
56
57        let path = Path::new(file);
58        let mut file = File::create(&path).unwrap();
59
60        let mut i = 0;
61
62        while t < te {
63            t += dt;
64            let (sn, wn) = forward_euler(&s, &w, dt);
65
66            s = sn; w = wn;
67
68            let e = (cross(&w,&cross(&s, &w)) - &s).norm(Euclidean);
69
70            if i % interval == 0 {
71            file.write_all(format!("{}, {}, {}, {}, {}\n", t, s[0], s[1], s[2],
    ↪  e).as_bytes()).unwrap();
72            }
73
74            i += 1;
75        }
76
77    }
78
79    fn main() {
80        euler(0.01, (2.0_f64).sqrt(), "forward_euler_1", None);
81        euler(0.01, (1.0_f64).sqrt(), "forward_euler_2", None);
82        euler(0.01, (2.0_f64).sqrt() * 2.0, "forward_euler_3", None);
83        euler(0.01, (2.0_f64).sqrt() / 3.0, "forward_euler_4_1", None);
84        euler(0.0001, (2.0_f64).sqrt() / 3.0, "forward_euler_4_2", Some(10));
85        euler(0.000001, (2.0_f64).sqrt() / 3.0, "forward_euler_4_3", Some(1000));
86    }
```

Listing 1: forward euler integration of the two body problem

```
reset
plot "forward_euler_1" using 2:3 with lines title "forward euler with v_0"
```

Listing 2: gnuplot code for plotting the orbit

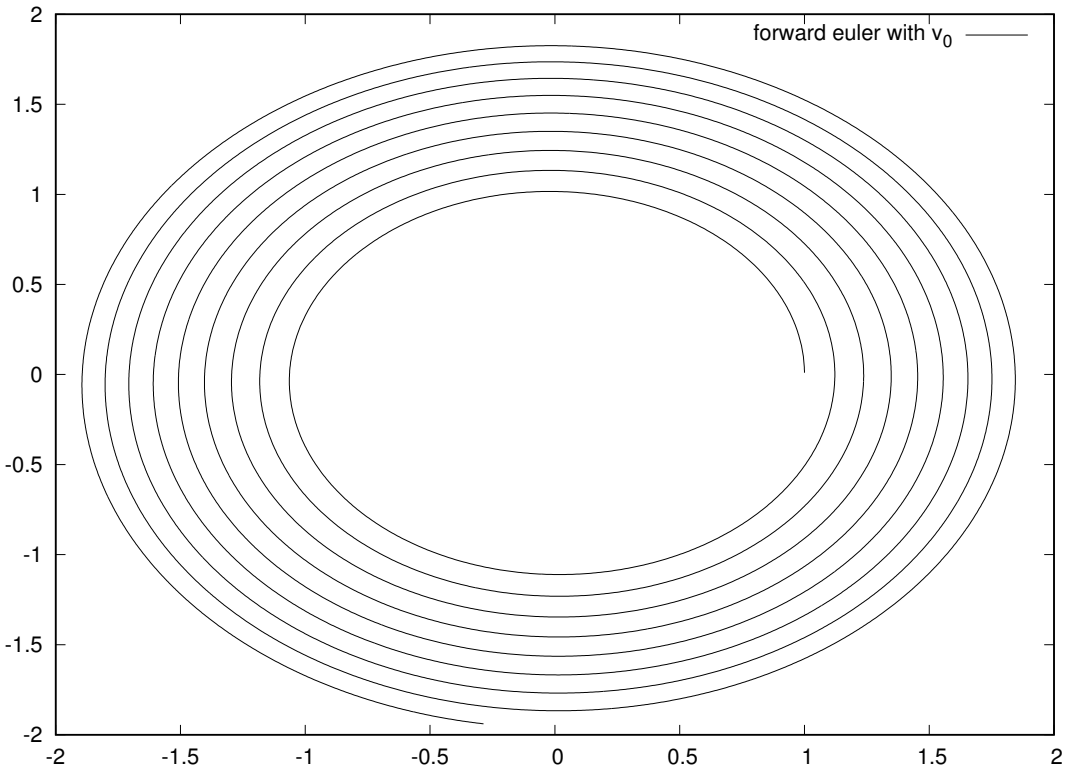

Figure 1: plot of x and y component of $\vec{s}$ with v_0

```
reset
plot "forward_euler_2" using 2:3 with lines title "forward euler with v_0 /
↪ \sqrt{2}"
```

Listing 3: gnuplot code for plotting the orbit

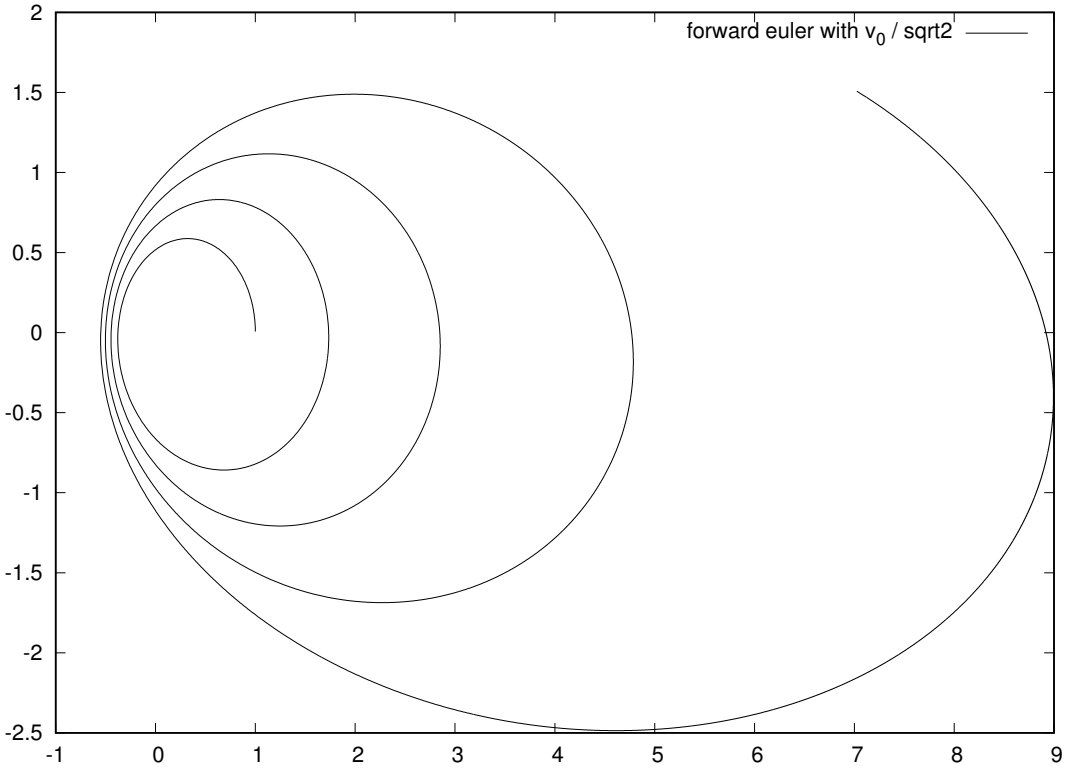

Figure 2: plot of x and y component of $\vec{s}$ with $v_0/\sqrt{2}$

```
reset
plot "forward_euler_1" using 1:5 with lines title "eccentricity of case v_0"
```

Listing 4: gnuplot code for plotting the eccentricity

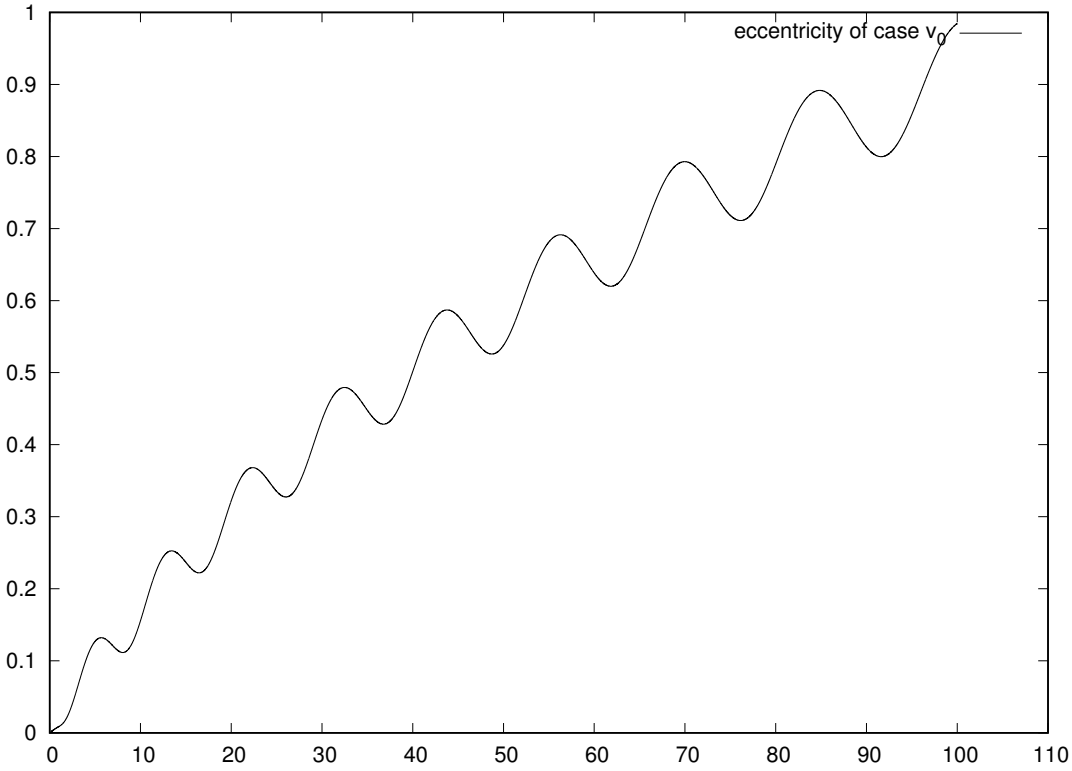

Figure 3: plot eccentricity with v_0

```
reset
plot "forward_euler_3" using 2:3 with lines title "forward euler with 2 * v_0"
```

Listing 5: gnuplot code for plotting the orbit

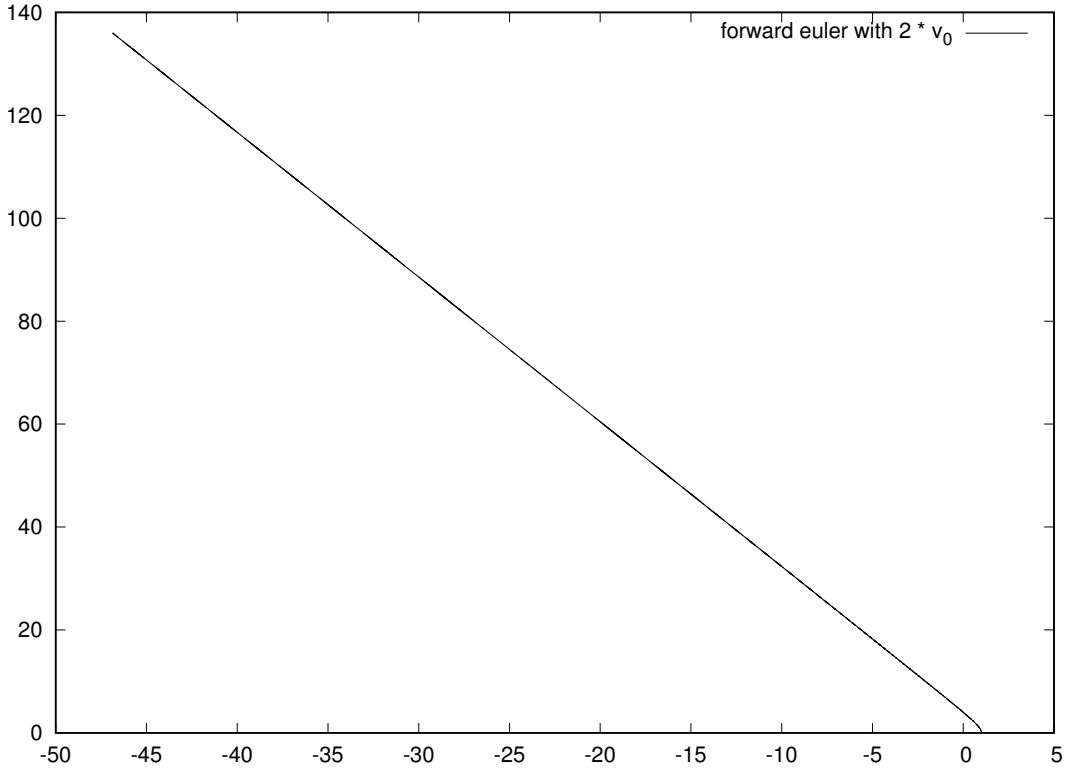

Figure 4: plot of x and y component of $\vec{s}$ with $2v_0$

```
reset
plot "forward_euler_4_1" using 2:3 with lines title "forward euler with v_0 / 3
 ↪  and dt = 0.01"
```

Listing 6: gnuplot code for plotting the orbit

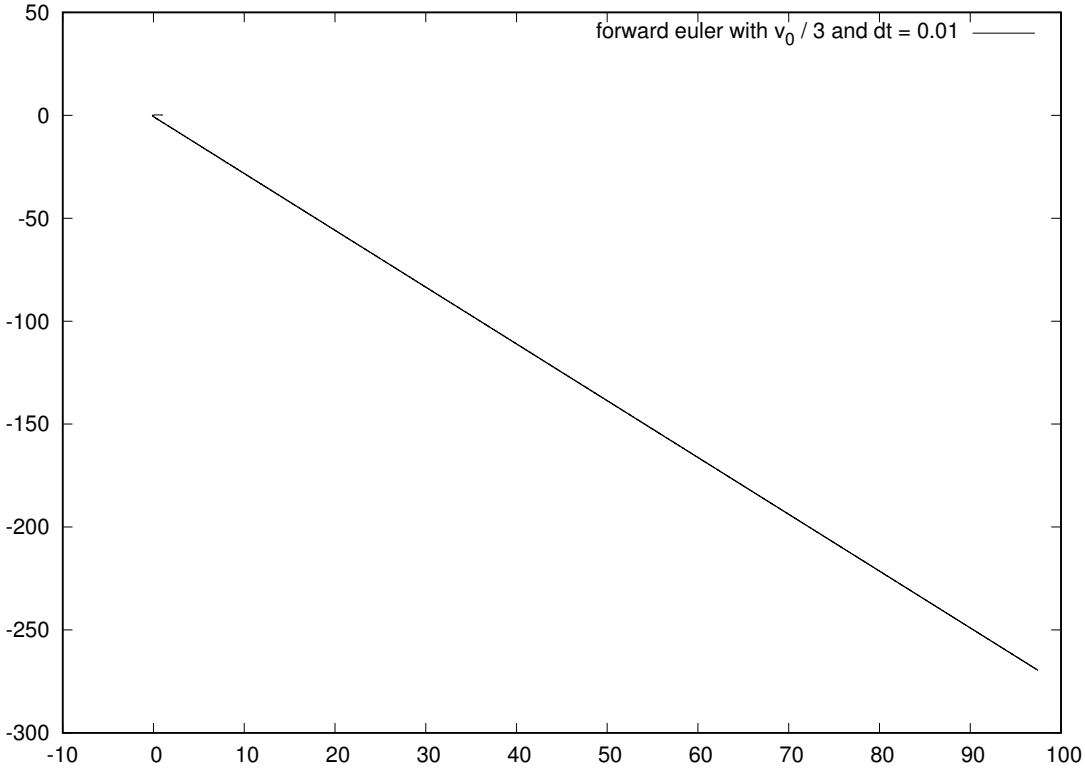

Figure 5: plot of x and y component of $\vec{s}$ with $v_0/3$ and $dt = 0.01$

```
reset
plot "forward_euler_4_2" using 2:3 with lines title "forward euler with v_0 / 3
 ↪  and dt = 0.0001"
```

Listing 7: gnuplot code for plotting the orbit

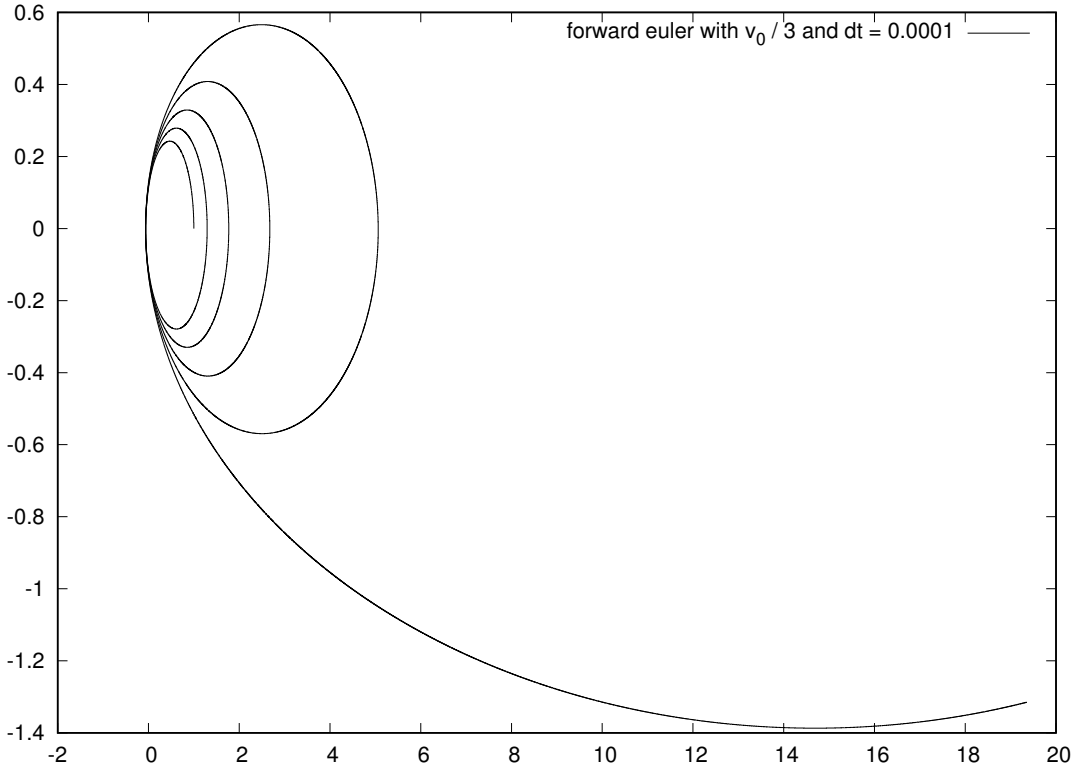

Figure 6: plot of x and y component of $\vec{s}$ with $v_0/3$ and $dt = 0.0001$

```
reset
plot "forward_euler_4_3" using 2:3 with lines title "forward euler with v_0 / 3
 ↪  and dt = 0.000001"
```

Listing 8: gnuplot code for plotting the orbit

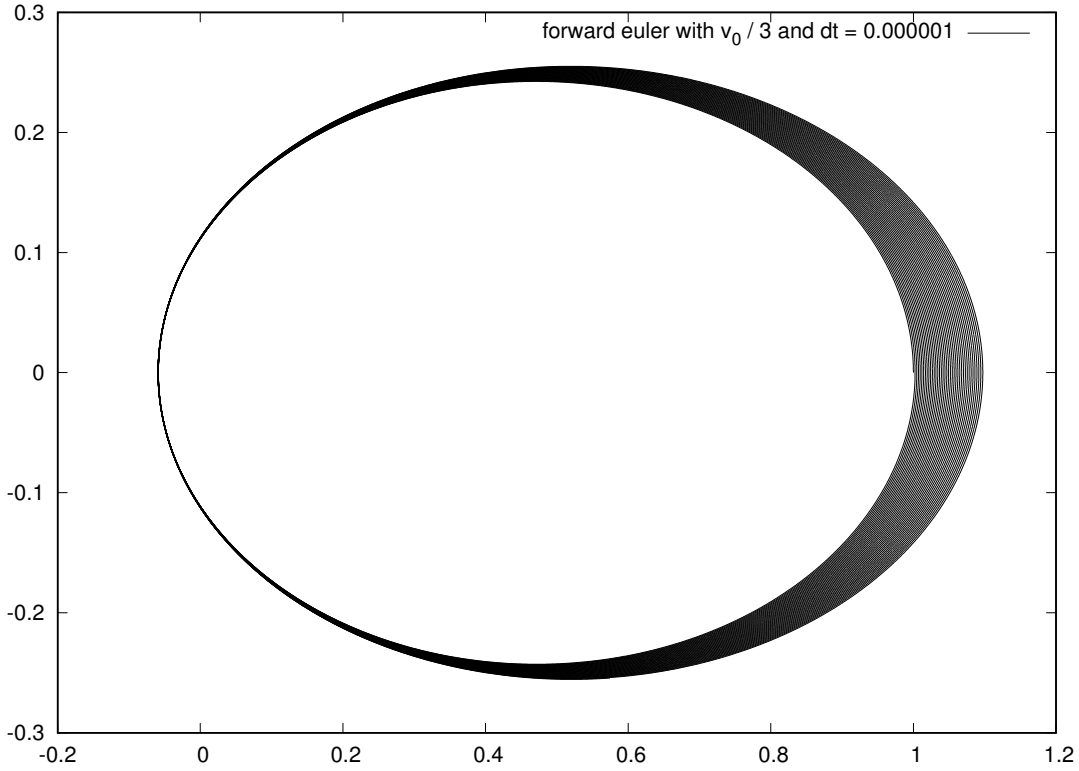

Figure 7: plot of x and y component of $\vec{s}$ with $v_0/3$ and $dt = 0.000001$

Choosing a initial velocity greater than $\sqrt{2}v_0$ means the initial velocity is greater than the escape velocity and the orbit is a hyperbola. Choosing $v_0/3$ as initial velocity means the orbit is very elliptical thus the masses get really close and the acceleration is gets very big. This increases the error through the discrete timesteps. But even choosing much smaller timesteps does not result in a stable system, because of the limited precision of floatingpoint representation and the error inherent from the euler scheme.

2 Error analysis of euler scheme

As initial velocities $v = v_0$, $v = 1.5 \cdot v_0$ and $v_0/2$ are choosen. For each of the initial velocities stepsizes dt_n are used:

$$dt_n = 10^{-n/4-1} \quad \text{where} \quad n \in \{0, \ldots, 20\}$$

The relative error of the energy ε is calculated after one orbit $\tau = 2\pi$.

$$\varepsilon = \frac{|E_{t=\tau} - E_{t=0}|}{|E_{t=0}|}$$

$$E_t = \frac{\omega_t^2}{2} - \frac{1}{s}$$

This is done for the forward euler integration scheme, the velocity verlet scheme and the leapfrog scheme. The relative error is then plotted against the different stepsizes.

```
1 // cargo-deps: rulinalg
2 #[macro_use]
3 extern crate rulinalg;
4
```

```rust
use std::fs::File;
use std::path::Path;
use std::io::Write;
use rulinalg::vector::Vector;
use rulinalg::norm::Euclidean;

const G: f64 = 1.0;


fn forward_euler(s: &Vector<f64>, w: &Vector<f64>, dt: f64, _: bool, _: bool) ->
 ↪  (Vector<f64>, Vector<f64>) {
    let sn = s + w * dt;
    let wn = w + acceleration(s) * dt;

    (sn, wn)
}

fn velocity_verlet(s: &Vector<f64>, w: &Vector<f64>, dt: f64, _: bool, _: bool)
 ↪  -> (Vector<f64>, Vector<f64>) {
    let a = acceleration(s);
    let sn = s + w * dt + a.clone() * dt * dt * 0.5;
    let an = acceleration(&sn);
    let wn = w + (a + an) * dt * 0.5;

    (sn, wn)
}

fn leapfrog(s: &Vector<f64>, w: &Vector<f64>, dt: f64, first: bool, last: bool)
 ↪  -> (Vector<f64>, Vector<f64>) {
    if first {
        let (_, wn) = forward_euler(s, w, dt / 2.0, first, last);
        return (s.clone(), wn);
    } else if last {
        let (sn, _n) = forward_euler(s, w, dt / 2.0, first, last);
        return (sn, w.clone());
    }

    let sn = s + w * dt;
    let wn = w + acceleration(&sn) * dt;

    (sn, wn)
}


fn normalize(M: f64, r: Vector<f64>, v: Vector<f64>, h: f64, R0: f64) ->
 ↪  (Vector<f64>, Vector<f64>, f64) {
      let s = r / R0;
      let V0 =  ((G * M) / R0).sqrt();
      let w = v / V0;
      let dt = h * V0 / R0;
```

```rust

        (s, w, dt)
}

fn cross(a: &Vector<f64>, b: &Vector<f64>) -> Vector<f64> {
        let mut c = vector![0.0, 0.0, 0.0];
        c[0] = a[1] * b[2] - a[2] * b[1];
        c[1] = a[2] * b[0] - a[0] * b[2];
        c[2] = a[0] * b[1] - a[1] * b[0];
        c
}

type Integrator =    &'static Fn(&Vector<f64>, &Vector<f64>, f64, bool, bool) ->
↪  (Vector<f64>, Vector<f64>);

fn error_analysis(integrator: Integrator, file: &str) {
    let v0 = (2.0_f64).sqrt();
    let v0s = &[v0, 1.5 * v0, v0 / 2.0];
    let dts = (0..21).map(|e| 10.0_f64.powf(-(e as f64) / 4.0 - 1.0));

    let path = Path::new(file);
    let mut file = File::create(&path).unwrap();

    for dt in dts {
        let mut line = format!("{}", dt).to_string();
        for v in v0s {
            let error = single_run(integrator, dt, *v);
            line = format!("{}, {}", line, error);
        }

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

fn energy(s: &Vector<f64>, w: &Vector<f64>) -> f64 {
    let w_norm = w.norm(Euclidean);
    let s_norm = s.norm(Euclidean);

    w_norm * w_norm / 2.0 - 1.0 / s_norm
}

fn single_run(integrator: Integrator, dt_0: f64, v0: f64) -> f64 {
        let M1: f64 = 1.0;
        let M2: f64 = 1.0;
        let R0: f64 = 1.0;
        let r = vector![1.0, 0.0, 0.0];
        let v = vector![0.0, v0, 0.0];
        let te: f64 = 2.0 * std::f64::consts::PI;
        let h: f64 = 0.01;
        let mut t = 0.0;
```

```
100
101         let (mut s, mut w, mut dt) = normalize(M1 + M2, r, v, h, R0);
102         dt = dt_0;
103
104         let e0 = energy(&s, &w);
105
106         let (sn, wn) = integrator(&s, &w, dt, true, false);
107         s = sn; w = wn;
108         t += dt;
109
110         while t < te - dt {
111             t += dt;
112
113             let (sn, wn) = integrator(&s, &w, dt, false, false);
114
115             s = sn; w = wn;
116         }
117
118         let (sn, wn) = integrator(&s, &w, dt, false, true);
119         s = sn; w = wn;
120
121
122         let ee = energy(&s, &w);
123         (ee - e0).abs() / e0.abs()
124 }
125
126 fn acceleration(s: &Vector<f64>) -> Vector<f64> {
127     let sa = s.norm(Euclidean);
128     -s / (sa * sa * sa)
129 }
130
131 fn main() {
132   error_analysis(&forward_euler, "euler");
133   error_analysis(&velocity_verlet, "velocity_verlet");
134   error_analysis(&leapfrog, "leapfrog");
135 }
```

Listing 9: forward euler integration of the two body problem

```
1 reset
2 set log x
3 set log y
4 set xrange [.11:0.000009] reverse
5 plot "euler" using 1:2 with lines title "v_0", "euler" using 1:3 with lines title
  ↪  "1.5 * v_0", "euler" using 1:4 with lines title "v_0 / 2"
```

Listing 10: Gnuplot code for plotting the relativ energy error of forward euler integration

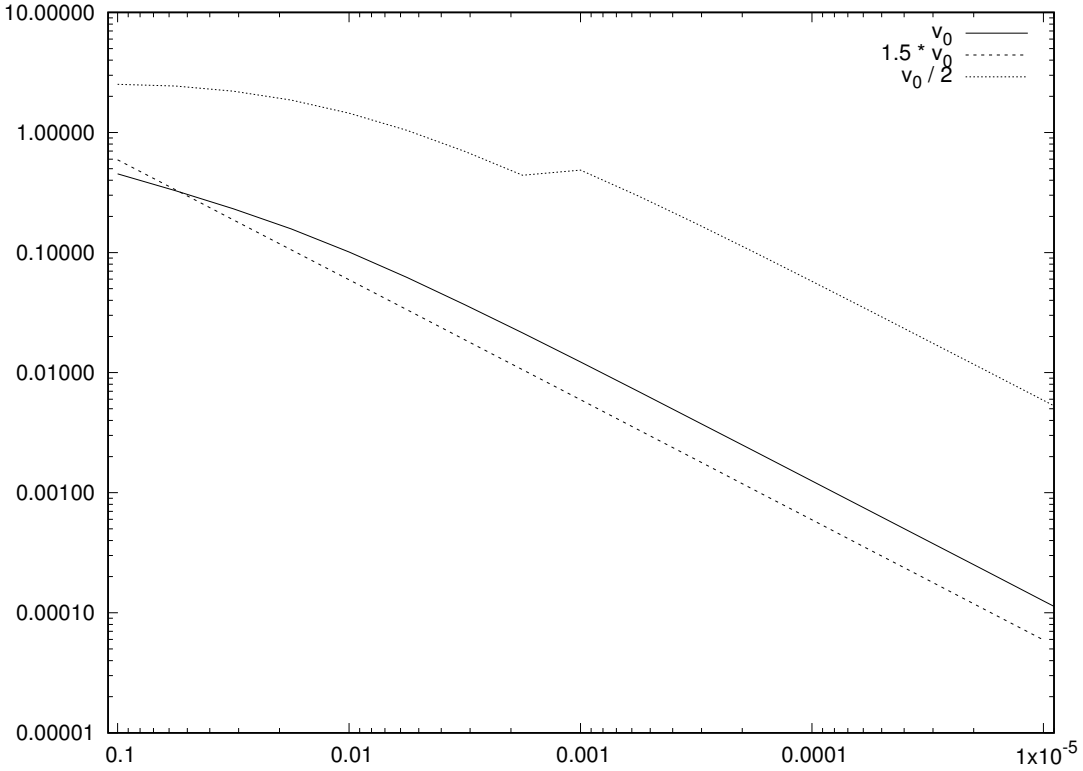

Figure 8: Plot of the relative error of euler integration against different stepsizes for three different initial velocities

For the euler integration the error says quite big, even with very small timesteps. This could be observed in the previous task very well, even a circular orbit deteriorated after a short time. Using a even small initial velocity means both particles get closer to each other and this means the accelerations increases very fast as the particles get close. This means that the error introduced by discrete timesteps is quite big. This can also be seen in the relative error of the energy, it is a lot bigger for $v_0/2$ than for v_0.

```
reset
set log x
set log y
set xrange [.11:0.000009] reverse
plot "velocity_verlet" using 1:2 with lines title "v_0", "velocity_verlet" using
 ↪  1:3 with lines title "1.5 * v_0", "velocity_verlet" using 1:4 with lines
 ↪  title "v_0 / 2"
```

Listing 11: Gnuplot code for plotting the relativ energy error of velocity verlet integration

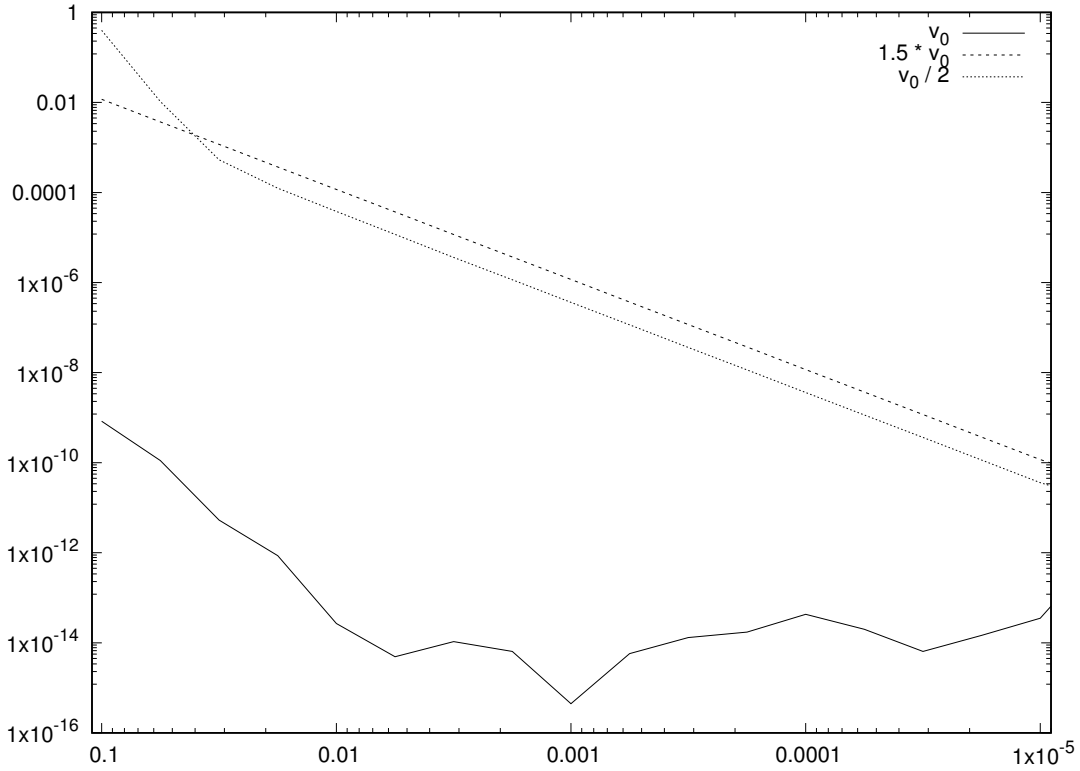

Figure 9: Plot of the relative error of velocity verlet integration against different stepsizes for three different initial velocities

The relative error of the energy is a lot smaller using the velocity verlet scheme than it was using the euler scheme. For a circular orbit it seems to reach the error introduced by machine precision. This could explain, why the error does not continuously shrink with decreasing timesteps. The difference in error between other initial velocities is also smaller than when using euler integration. The improved errors are of course expected, because the velocity verlet scheme is of second order, inlike the euler scheme which is only of first order.

```
1 reset
2 set log x
3 set log y
4 set xrange [.11:0.000009] reverse
5 plot "leapfrog" using 1:2 with lines title "v_0", "leapfrog" using 1:3 with lines
  ↪  title "1.5 * v_0", "leapfrog" using 1:4 with lines title "v_0 / 2"
```

Listing 12: Gnuplot code for plotting the relativ energy error of forward leapfrog integration

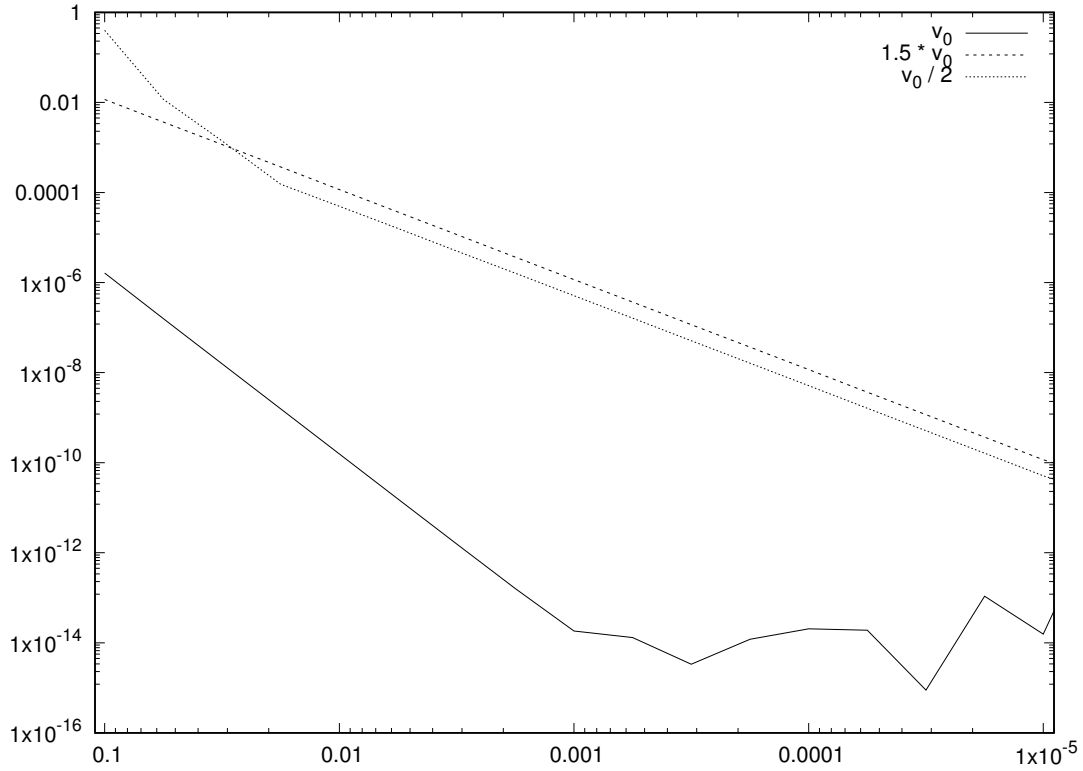

Figure 10: Plot of the relative error of leapfrog integration against different stepsizes for three different initial velocities

The leapfrog scheme and the velocity verlet scheme produce quite similar errors. This is also expected, because of the similarity of both schemes. The velocity calculation of the velocity verlet scheme can also be written as a half stepsize step. The only major difference in relative error between the two is seen when the orbit is circular. This could be caused by the euler steps needed at the start and end of the leapfrog scheme, as these are not needed for the velocity verlet scheme.

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.