

Exercise sheet 3

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

February 1, 2019

1 Three-Body Problem

The Runge-Kutta-4 integrator for a ordinary differential equation of first order $y'(x) = f(y, x)$ is given by

$$\begin{aligned}k_1 &= hf(y_i, x_i) \\k_2 &= hf\left(y_i + \frac{k_1}{2}, x_i + \frac{h}{2}\right) \\k_3 &= hf\left(y_i + \frac{k_2}{2}, x_i + \frac{h}{2}\right) \\k_4 &= hf(y_i + k_3, x_i + h) \\x_{i+1} &= x_i + h \\y_{i+1} &= y_i + \frac{1}{6}(k_1 + 2k_2 + 2k_3 + k_4)\end{aligned}$$

Consider three bodies at positions \vec{x}_i with masses m_i ($i \in \{1, 2, 3\}$). Their acceleration is then given by

$$\ddot{\vec{x}}_i = G \sum_{\substack{j=1 \\ j \neq i}}^3 \frac{m_j}{(\vec{x}_i - \vec{x}_j)^3} (\vec{x}_i - \vec{x}_j)$$

when restricting the motion to two dimensions this gives us two second order ordinary differential equations of each body. These can be converted to a set of four first order ordinary differential equations by introducing the velocity \vec{v}_i . This results in a total of twelve first order differential equations.

$$\dot{\vec{x}}_i = \vec{v}_i \quad \dot{\vec{v}}_i = G \sum_{\substack{j=1 \\ j \neq i}}^3 \frac{m_j}{(\vec{x}_i - \vec{x}_j)^3} (\vec{x}_i - \vec{x}_j)$$

This can be written as

$$\begin{aligned}\vec{y}'(t) &= \vec{f}(\vec{y}, t) \\ \vec{y} &= ((\vec{x}_1 \quad \vec{v}_1 \quad \dots \quad \vec{x}_3 \quad \vec{v}_3))^T \\ \vec{f}(\vec{y}, x) &= \left(\left(\vec{v}_1 \quad G \sum_{\substack{j=1 \\ j \neq i}}^3 \frac{m_j}{(\vec{x}_1 - \vec{x}_j)^3} (\vec{x}_1 - \vec{x}_j) \quad \dots \right) \right)^T\end{aligned}$$

Using this notation the Runge-Kutta-4 integration for the three-body problem can be written as

$$\begin{aligned}
 \vec{k}_1 &= h \vec{f}(\vec{y}_i, t_i) \\
 \vec{k}_2 &= hf\left(\vec{y}_i + \frac{\vec{k}_1}{2}, t_i + \frac{h}{2}\right) \\
 \vec{k}_3 &= hf\left(\vec{y}_i + \frac{\vec{k}_2}{2}, t_i + \frac{h}{2}\right) \\
 \vec{k}_4 &= hf\left(\vec{y}_i + \vec{k}_3, t_i + h\right) \\
 \vec{t}_{i+1} &= t_i + h \\
 \vec{y}_{i+1} &= \vec{y}_i + \frac{1}{6}(\vec{k}_1 + 2\vec{k}_2 + 2\vec{k}_3 + \vec{k}_4)
 \end{aligned}$$

The appended implementation of the Runge-Kutta scheme in *rust* is generic and can be used with arbitrary (implicit) Runge-Kutta schemes.

Integrating the first set of initial conditions results in a figure eight. The integration seems to be stable for quite a long time, as there is no drift in the orbits visible. This is probably caused by the periodic nature of the orbits.

```

1 reset
2 plot "a" u 1:2 w l title "body 1", "a" u 3:4 w l title "body 2", "a" u 5:6 w l
  ↪ title "body 3",

```

Listing 1: Gnuplot code for plotting the orbits of the first set of initial conditions

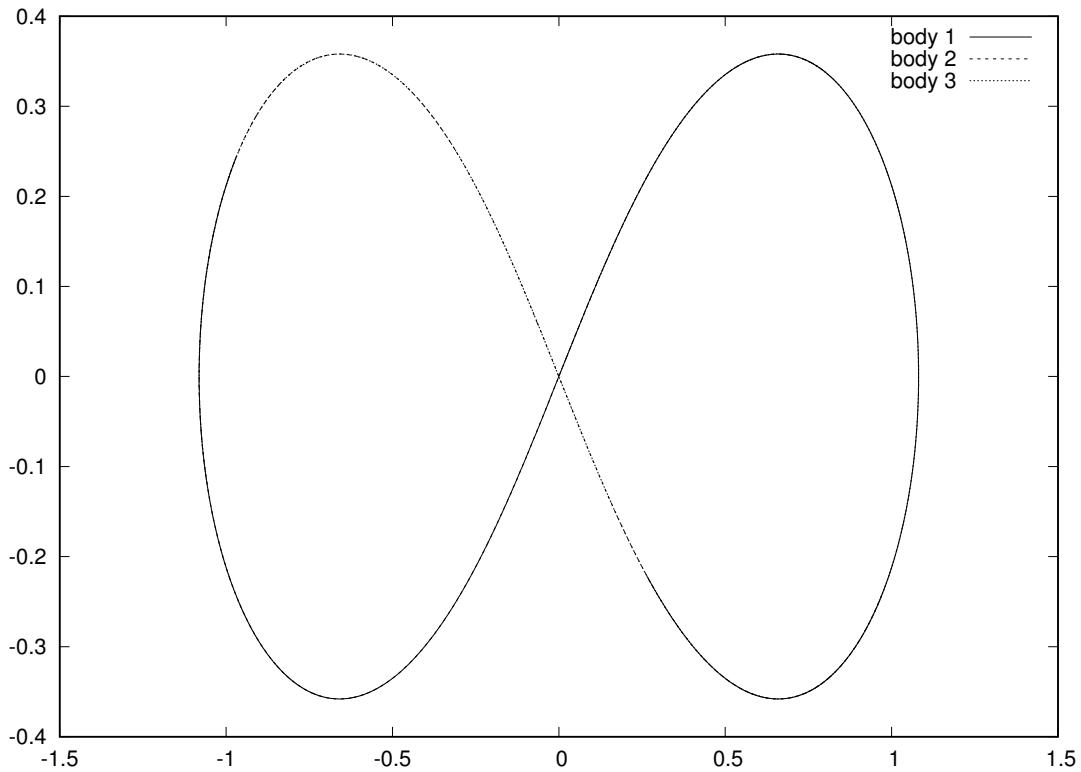


Figure 1: Plot of the orbit of the three bodies for the first set of initial conditions

The second set of initial conditions leads to a highly chaotic system. The minimum separations are found by comparing the last three distances and searching for local minima. This is done for each pair of two bodies. For h a value of 10^{-e} for $e \in \{1, \dots, 5\}$ is chosen. In Table 1 the times of a minimum separation in dependence of the chosen e are shown. Comparing the times for different values of e shows that a value of e between 3 and 4 are necessary to get *ok* estimates for the time of the first closest encounters. But even with $e = 4$ there is quite a big difference between the times for the fourth and fifth time of a local minimum in the separation.

For the following graphs of the orbits, energy and distances for different h the integration was stopped when the system started to dissolve.

```

1 reset
2 set xrange [-3:4]
3 set yrange [-4:4]
4 plot "b_3" u 1:2 w l title "body 1", "b_3" u 3:4 w l title "body 2", "b_3" u 5:6
   ↪ w l title "body 3",

```

Listing 2: Gnuplot code for plotting the orbits of the second set of initial conditions with $e = 3$

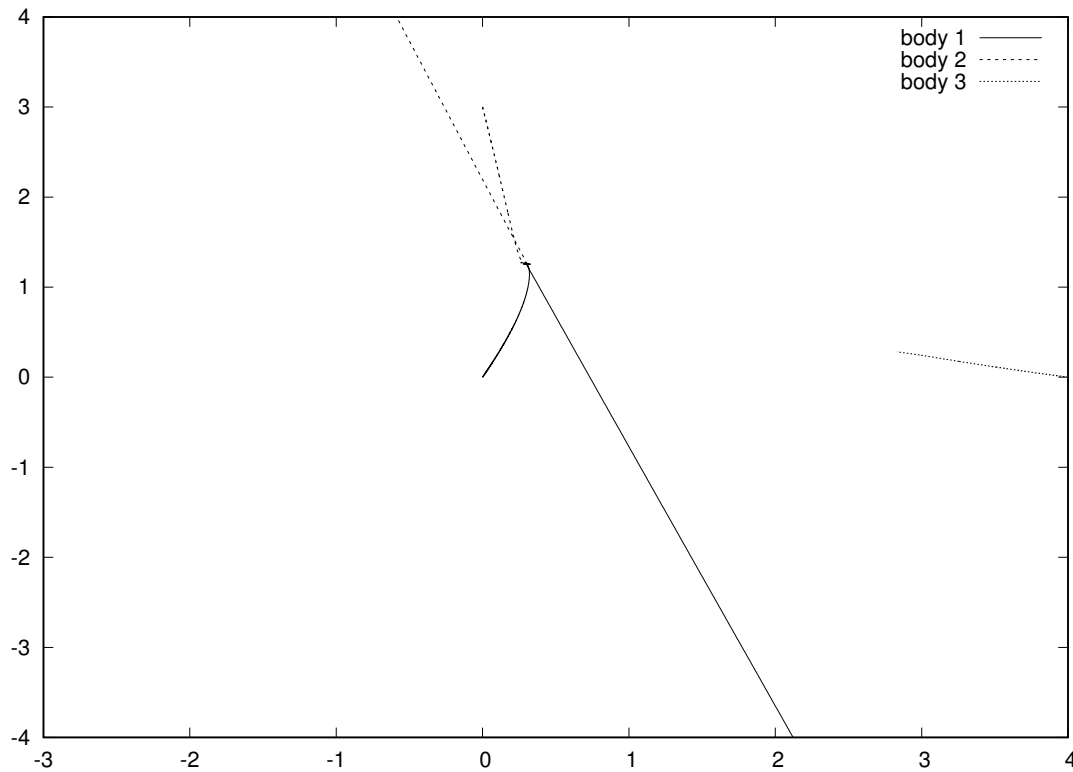
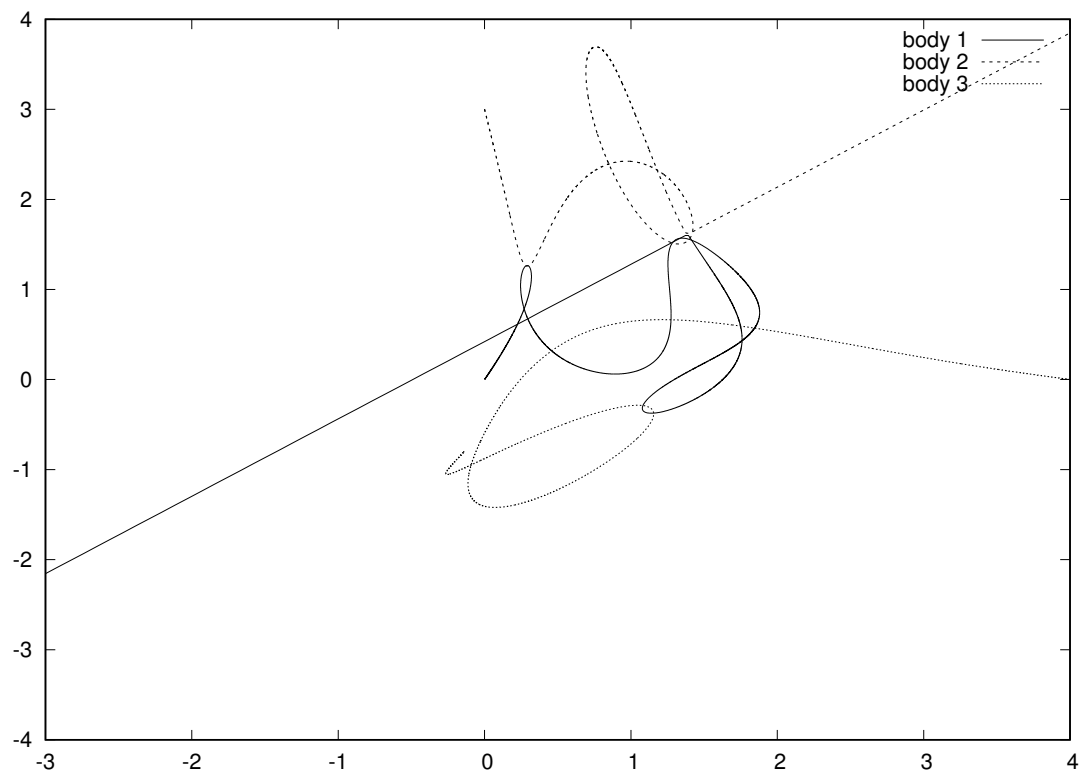


Figure 2: Plot of the orbit of the three bodies for the second set of initial conditions with $e = 3$

```

1 reset
2 set xrange [-3:4]
3 set yrange [-4:4]
4 plot "b_4" u 1:2 w l title "body 1", "b_4" u 3:4 w l title "body 2", "b_4" u 5:6
   ↪ w l title "body 3",

```

Listing 3: Gnuplot code for plotting the orbits of the second set of initial conditions with $e = 4$ Figure 3: Plot of the orbit of the three bodies for the second set of initial conditions with $e = 4$

```

1  reset
2  set xrange [-3:4]
3  set yrange [-4:4]
4  plot "b_5" u 1:2 w l title "body 1", "b_5" u 3:4 w l title "body 2", "b_5" u 5:6
   ↪ w l title "body 3",

```

Listing 4: Gnuplot code for plotting the orbits of the second set of initial conditions with $e = 5$

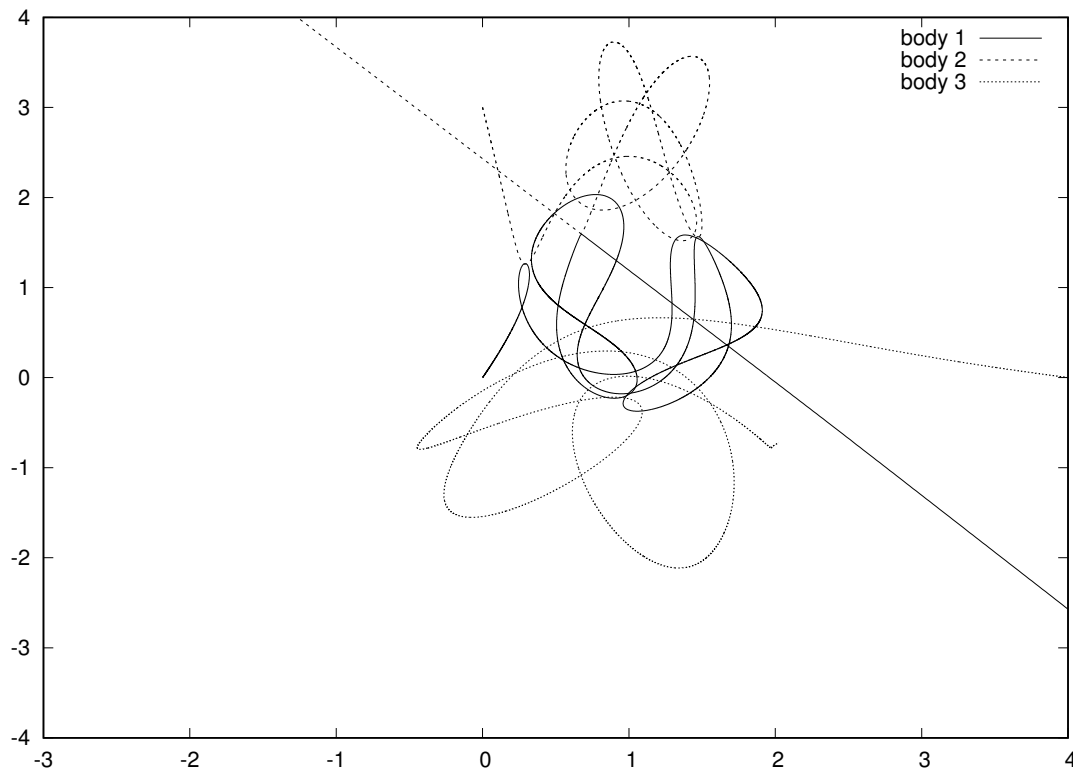


Figure 4: Plot of the orbit of the three bodies for the second set of initial conditions with $e = 5$

The orbit is highly chaotic and dissolves quite quickly due to numerical errors. Decreasing h only helps a little, which is unexpected for a fourth order integrator.

```

1  reset
2  set log y
3  set xlabel "t"
4  plot "b_5" u 11:10 w l title "e = 5", "b_4" u 11:10 w l title "e = 4", "b_3" u
   ↪ 11:10 w l title "e = 3"

```

Listing 5: Gnuplot code for plotting the error of the energy of the second set of initial conditions for different e

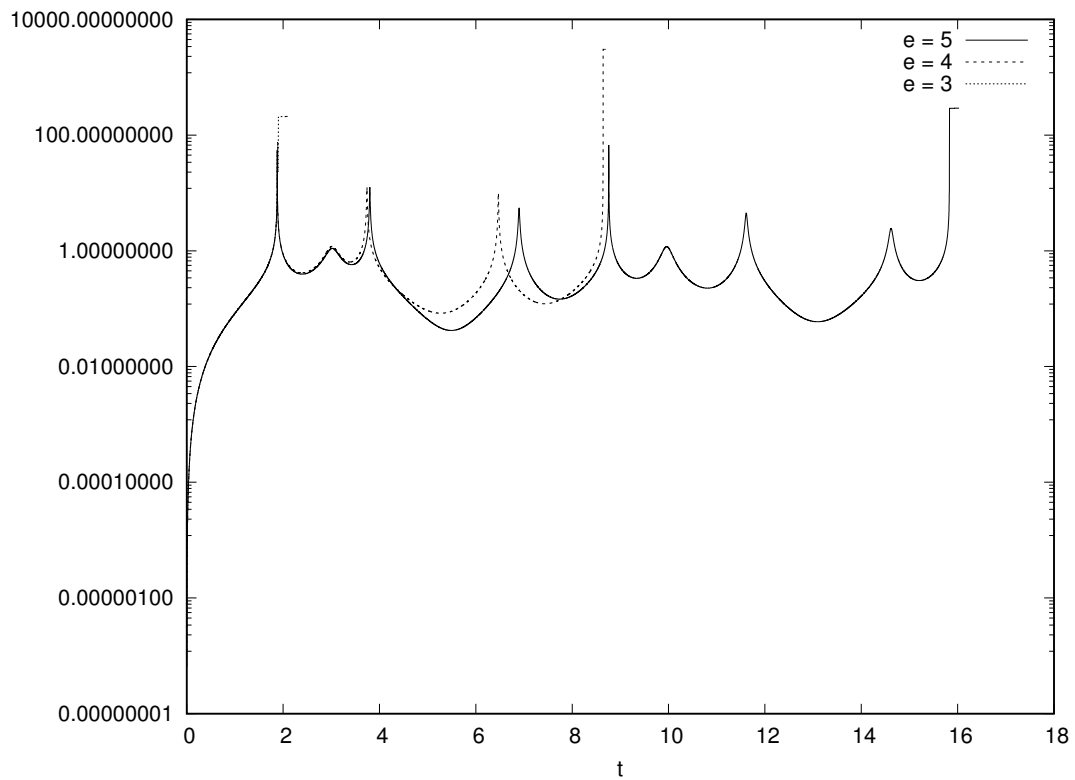


Figure 5: Plot of the relative error of the total energy for the second set of initial conditions and different e

The error of the energy is quite big and increases rapidly for the close encounters, although it rapidly decreases after the close encounters. Increasing the timesteps also does not seem to help much with the error of the total energy.

```

1 reset
2 set log y
3 set xlabel "t"
4 plot "b_5" u 11:7 w l title "e = 5", "b_4" u 11:7 w l title "e = 4", "b_3" u 11:7
   ↪ w l title "e = 3"

```

Listing 6: Gnuplot code for plotting the distance of the first two bodies for the second set of initial conditions for different e

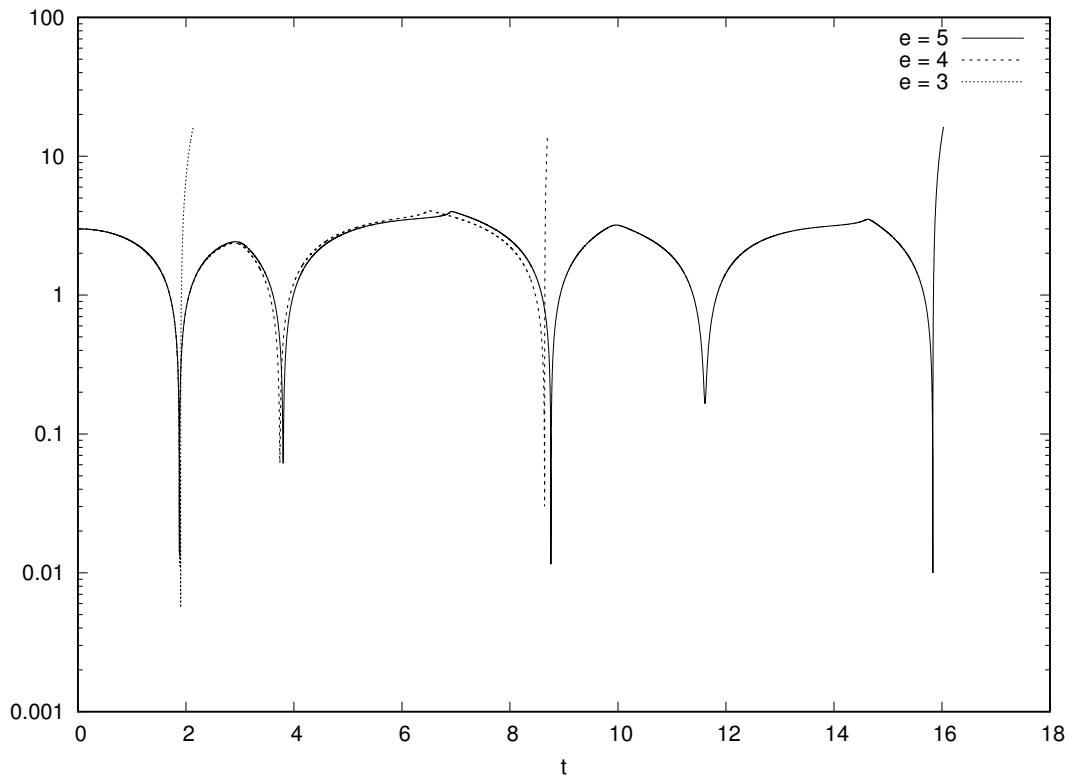


Figure 6: Plot of the distance of the first two bodies for the second set of initial conditions with different e

The logarithmic plot of the distance shows that there is a high dynamic range of the relevant distances. This mean using a fixed timestep is suboptimal for this system.

2 Rust implementation

```

1 // cargo-deps: rulinalg
2 #[macro_use]
3 extern crate rulinalg;
4
5 use rulinalg::vector::Vector;
6 use rulinalg::norm::Euclidean;
7 use std::ops::Mul;
8 use std::fs::File;
9 use std::path::Path;
10 use std::io::Write;
11
12 pub trait MulEx<RHS = Self> {
13     type Output;
14     fn mul(self, rhs: RHS) -> Self::Output;
15 }
16
17 pub trait AddEx<RHS = Self> {
18     type Output;
19     fn add(self, rhs: RHS) -> Self::Output;
20 }

```

```

21
22 impl MulEx<f64> for f64 {
23     type Output = f64;
24     fn mul(mut self, f: f64) -> f64 {
25         self * f
26     }
27 }
28
29 impl AddEx<f64> for f64 {
30     type Output = f64;
31     fn add(mut self, f: f64) -> f64 {
32         self + f
33     }
34 }
35
36 impl<T: Clone, TV: Clone + MulEx<T, Output = TV>> MulEx<T> for Vector<TV> {
37     type Output = Vector<TV>;
38     fn mul(mut self, f: T) -> Vector<TV> {
39
40         for val in self.mut_data().iter_mut() {
41             *val = val.clone().mul(f.clone());
42         }
43
44         self
45     }
46 }
47
48 impl<TV: Clone + AddEx<TV, Output = TV>> AddEx<Vector<TV>> for Vector<TV>
49 {
50     type Output = Vector<TV>;
51     fn add(mut self, f: Vector<TV>) -> Vector<TV> {
52         let mut i = 0;
53         for val in self.mut_data().iter_mut() {
54             *val = val.clone().add(f[i].clone());
55             i += 1;
56         }
57
58         self
59     }
60 }
61
62 fn runge_kutta<T: Clone>(t: f64, x0: &Vector<T>, h: f64, f: &Fn(f64, &Vector<T>))
63     ↪ -> Vector<T>, rk_tab: &Vec<Vec<f64>>) -> Vector<T>
64 where Vector<T> : MulEx<f64, Output=Vector<T>>
65     , Vector<T> : AddEx<Vector<T>, Output=Vector<T>>
66 {
67     let order = rk_tab.len() - 1;
68     let mut k = vec![x0.clone(); order];
69
70     for i in 0..order {

```



```

70     let mut x = x0.clone();
71     let rk_row = &rk_tab[i];
72
73     for j in 0..order {
74         x = x.add(k[j].clone().mul(h * rk_row[j + 1]));
75     }
76
77
78     k[i] = f(t + h * rk_row[0], &x);
79     }
80
81     let mut xn = x0.clone();
82
83     for i in 0..order {
84         xn = xn.add(k[i].clone().mul(h * rk_tab[order][i + 1]));
85     }
86
87     xn
88 }
89
90 fn gravitation(_t: f64, x: &Vector<Vector<f64>>, m: &Vector<f64>) ->
91 ↪ Vector<Vector<f64>> {
92     let n = x.size() / 2;
93     let mut xn = x.clone();
94
95     for i in 0..n {
96         xn[2 * i + 0] = x[2 * i + 1].clone();
97         xn[2 * i + 1] = &xn[2 * i + 1] - &xn[2 * i + 1];
98
99         for j in 0..n {
100             if j == i { continue; }
101
102             let r = &x[2 * i + 0] - &x[2 * j + 0];
103             let rn = r.norm(Euclidean);
104
105             xn[2 * i + 1] -= (&r / (rn * rn * rn)) * m[j];
106         }
107     }
108     xn
109 }
110
111 fn energy(y: &Vector<Vector<f64>>, m: &Vector<f64>) -> f64 {
112     let n = y.size() / 2;
113     let mut e = 0.0;
114
115     for i in 0..n {
116         e += 0.5 * y[2 * i + 1].dot(&y[2 * i + 1]) * m[i];
117         for j in 0..n {
118             if j == i { continue; }

```

```

119         let r = y[2 * i + 0].clone() - y[2 * j + 0].clone();
120         e -= m[i] * m[j] / r.norm(Euclidean);
121     }
122 }
123
124 e
125 }
126
127 fn t<T>(v: Vector<T>, f: &Fn(f64, &Vector<T>) -> Vector<T>) -> Vector<T> {
128     f(0.0, &v)
129 }
130
131 fn main() {
132     let rk4 = vec![vec![0.0, 0.0, 0.0, 0.0, 0.0]
133         ,vec![0.5, 0.5, 0.0, 0.0, 0.0]
134         ,vec![0.5, 0.0, 0.5, 0.0, 0.0]
135         ,vec![1.0, 0.0, 0.0, 1.0, 0.0]
136         ,vec![0.0, 1.0/6.0, 1.0/3.0, 1.0/3.0, 1.0/6.0]];
137
138     let x0 = vector![-0.97000436, 0.24308753];
139     let v0 = vector![-0.46620368, -0.43236573];
140
141     let x1 = vector![ 0.97000436, -0.24308753];
142     let v1 = vector![-0.46620368, -0.43236573];
143
144     let x2 = vector![ 0.0, 0.0];
145     let v2 = vector![ 0.93240737, 0.86473146];
146
147     let y0: Vector<Vector<f64>> = vector![x0, v0, x1, v1, x2, v2];
148
149     let mut yn = y0;
150
151     let m = vector![1.0, 1.0, 1.0];
152
153     let mut t = 0.0;
154     let dt = 0.00005;
155
156     let path = Path::new("a");
157     let mut file = File::create(&path).unwrap();
158
159     for i in 0..100000 {
160         if i % 10 == 0 {
161             file.write_all(
162                 format!("{}", {}, {}, {}, {}, {}, {} \n",
163                     yn[0][0], yn[0][1], yn[2][0], yn[2][1], yn[4][0], yn[4][1])
164                     .as_bytes()).unwrap();
165         }
166
167         yn = runge_kutta(t, &yn, dt, &|t, x| {
168             gravitation(t, x, &m)

```

```

169     }, &rk4);
170     t += dt;
171 }
172
173 println!("| $e$ | $t$");
174 println!("|-");
175
176 for e in 1..6 {
177     let x0 = vector![0.0, 0.0];
178     let v0 = vector![0.0, 0.0];
179
180     let x1 = vector![0.0, 3.0];
181     let v1 = vector![0.0, 0.0];
182
183     let x2 = vector![4.0, 0.0];
184     let v2 = vector![0.0, 0.0];
185
186     let y0: Vector<Vector<f64>> = vector![x0, v0, x1, v1, x2, v2];
187
188     let mut yn = y0;
189
190     let m = vector![5.0, 4.0, 3.0];
191
192     let mut t = 0.0;
193     let dt = 10.0_f64.powf(-e as f64);
194
195     let filename = &format!("b_{}", e);
196     let path = Path::new(filename);
197     let mut file = File::create(&path).unwrap();
198
199     let mut dist12 = vec![100.0, 100.0, 100.0];
200     let mut dist13 = vec![100.0, 100.0, 100.0];
201     let mut dist23 = vec![100.0, 100.0, 100.0];
202
203     let e0 = energy(&yn, &m);
204
205     for i in 0..((100.0 / dt) as u64) {
206         if yn[0].norm(Euclidean) > 10.0 || yn[2].norm(Euclidean) > 10.0 ||
↪ yn[4].norm(Euclidean) > 10.0 {
207             break;
208         }
209
210         dist12.remove(0);
211         dist13.remove(0);
212         dist23.remove(0);
213
214         dist12.push((&yn[0] - &yn[2]).norm(Euclidean));
215         dist13.push((&yn[0] - &yn[4]).norm(Euclidean));
216         dist23.push((&yn[2] - &yn[4]).norm(Euclidean));
217

```

```

218     if (dist12[0] > dist12[1] && dist12[1] < dist12[2])
219     || (dist13[0] > dist13[1] && dist13[1] < dist13[2])
220     || (dist23[0] > dist23[1] && dist23[1] < dist23[2]) {
221     println!("{}", e, t - dt);
222     }
223
224     let en = (energy(&yn, &m) - e0).abs() / e0.abs();
225
226     if i % std::cmp::max((0.001 / dt) as u64, 1) == 0 {
227     file.write_all(
228         format!("{}", {}, {}, {}, {}, {}, {}, {}, {}, {}, {} \n",
229             yn[0][0], yn[0][1], yn[2][0], yn[2][1], yn[4][0], yn[4][1],
230             dist12[2], dist13[2], dist23[2], en, t)
231         .as_bytes()).unwrap();
232     }
233
234     yn = runge_kutta(t, &yn, dt, &|t, x| {
235     gravitation(t, x, &m)
236     }, &rk4);
237     t += dt;
238     }
239     }
240 }

```

Listing 7: rust imilementation of the runge-kutta-4 integrator and application to the three-body problem

Table 1: times of local minima in the separation of the bodies for different e

e	t
1	1.8000000000000005
1	1.9000000000000004
1	2.1000000000000005
1	2.4000000000000001
2	1.8500000000000014
2	1.8800000000000014
2	2.699999999999864
2	3.099999999999978
3	1.8549999999999065
3	1.8789999999999039
3	1.8819999999999035
3	1.885999999999903
3	1.8879999999999029
3	1.8889999999999028
3	1.8969999999999019
3	1.9019999999999013
3	1.9639999999998945
4	1.8546999999998122
4	1.8792999999998095
4	2.944400000001789
4	3.019100000001947
4	3.7401000000034683
4	3.7467000000034822
4	6.467599999998265
4	6.482099999998232
4	8.641799999993198
4	8.665199999993144
5	1.854690000003683
5	1.8793400000038445
5	2.944550000010823
5	3.0242100000113448
5	3.8005000000164304
5	3.8072600000164747
5	6.897689999908037
5	6.930049999906812
5	8.759219999837564
5	8.759749999837544
5	9.918959999793659
5	9.962809999791999
5	11.61184999972957
5	11.624909999729075
5	14.617479999615783
5	14.661899999614102
5	15.829899999569884
5	15.829909999569884
5	15.902859999567122

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.