rkf45: a Maxima function for the numerical solution of initial value problems

P. J. Papasotiriou

Department of Materials Science, University of Patras, Greece.

Abstract

In this paper, a Maxima function for solving initial value problems is introduced. The function implements the Runge-Kutta-Fehlberg method of fourth-fifth order, providing adaptive step size and error control. The syntax is discussed in detail, together with several examples and practical guidelines. Solution to stiff initial value problems is discussed as well.

1 Introduction.

Differential equations have been of fundamental importance in the application of Mathematics to the physical sciences, and their importance in biological, social, and other sciences is not be underestimated as well. However, even simple mechanical systems are described mathematically by differential equations which cannot be solved analytically, unless simplifying assumptions - sometimes very unrealistic ones - are adopted. One could safely state that "differential equations cannot be solved analytically unless if...", and that *if...* is the main subject of theoretical textbooks about differential equations. In most realistic problems, however, the use of numerical methods in order to solve the differential equations involved is more or less mandatory.

Over the years, Runge-Kutta methods for integrating differential equations became very popular, because of their great performance at relatively little computation effort. For many users, the fourth-order Runge-Kutta method is not only the first word on the subject, but the last word as well. However, a good integrator for ordinary differential equations should exert some adaptive control over its own progress, making changes in the integration step size as necessary. Although Runge-Kutta methods were originally fixed-step integrators, several improvements providing adaptive step size do exist. In particular, methods based on a technique often called *embedded pairs* were widely used for that purpose, because of their ability to estimate an "optimal" step size with reduced computational effort.

In this paper, we introduce a Maxima function, named `rkf45`, for the numerical solution of initial value problems. The method implemented is the popular *Runge-Kutta-Fehlberg fourth-fifth-order scheme*. It is able to adjust the integration step so that a predetermined accuracy in the solution is achieved with minimum computational effort (only six function evaluations are needed per step.) The reader can find more details about this algorithm in many Numerical Analysis textbooks, e.g. Nougier (2001); Brian (2006); Burden & Faires (2005); Press et al. (1992).

2 Using `rkf45`.

2.1 Syntax.

Like all numerical algorithms for solving initial value problems, `rkf45` can only solve one or more differential equations *of first order*. This is not a serious restriction, as a differential equation of higher order can be transformed to a system of first-order differential equations (Maxima simplifies this task as well.)

In its simplest form, `rkf45` is used to solve the initial value problem described by one differential equation, $\frac{dy}{dx} = f(x,y)$, and one initial condition, $y(x_0) = y_0$. Note that the differential equation should be in the form $\frac{dy}{dx} = f(x,y)$, i.e., the right-hard side should define the derivative of the dependent variable. In such cases, `rkf45` can be called as

`rkf45(ode,func,initial,interval,`*`options`*`)`,

where `ode` should define $f(x,y)$ (the right-hand side of the differential equation to be solved,) `func` is the dependent variable (the unknown function, say y,) `init` is the value, y_0, of the dependent variable at the initial value of the independent variable, x_0, and `interval` is a list of three elements. The first element identifies the independent variable, while the second and third elements are the initial and final values of the independent variable, for instance `[x,0,6]`. Initial value does not need to be less than final value, so an `interval` like `[x,6,0]` is also valid.

If a system of differential equations is to be solved, `rkf45` should be called in the form

`rkf45([ode1,ode2,...],[func1,func2,...],[init1,init2,...],interval,`*`options`*`)`,

where `[ode1,ode2,...]` is a list defining the right-hand sides of the differential equations, `[func1,func2,...]` is the list of the dependent variables, `[init1,init2,...]` is a list defining the value of the dependent variables at the initial value of the independent variable, and `interval` is a list defining the independent variable and the integration interval, as described above.

A number of optional arguments are accepted by `rkf45`:

1. `full_solution`: A Boolean, defining if `rkf45` should return full solution or not. If set to `true`, `rkf45` will return a list containing full solution at all integration points selected by the algorithm. If set to `false`, only the solution at the final point will be returned (default: `true`.)

2. `absolute_tolerance`: The desired upper bound of the error (default: 10^{-6}.) Each integration step is accepted only if the local estimated error is less than absolute tolerance. If not, the step will be rejected, and a new, refined step will be tried again, until the estimated error is small enough.

3. `max_iterations`: The maximum number of iterations (default: 10000.)

4. `h_start`: The initial integration step (default: one 100th of the integration interval.) The user does not need to care much about that optional argument. It can be useful in special cases only, and it is not absolutely necessary even then (see § 2.2.2 for an example.)

5. `report`: A Boolean, defining if a report will be printed at exit. If set to `true`, `rkf45` will print a report, giving details about the calculations done (default: `false`.)

Some general remarks should be emphasized.

1. The user has no access to the integration steps taken; they are selected automatically in such a way that the result is accurate enough. Each step taken is "optimal", in the sense that it is sufficiently small so that estimated error is less than desired `absolute_tolerance`. However, the user should remember that the algorithm only *estimates* the absolute error. Nevertheless, this error estimation is fairly good in most cases, and the actual absolute error is typically less than `absolute_tolerance` (see § 2.2.1 for an example.)

2. `absolute_tolerance` refers to *absolute* (not relative) accuracy. The default value, 10^{-6}, may be too high or too low, depending on the nature of the problem under consideration. For example, it doesn't make sense to seek for a solution with absolute error less than 10^{-6} when the functions `[func1,func2,...]` take typical values of 10^4 in the integration interval. In a similar way, the default tolerance may be too high if the functions take typical values of 10^{-7}. For that reason, It is always better to have an idea about the behavior of the functions involved in the differential equations (see § 2.2.3 for an example.)

3. If the algorithm cannot achieve an accurate solution, it exits with a warning message. In such cases, one should not blindly increase the maximum number of iterations. The first thing to do if something goes wrong is to check the differential equations passed to `rkf45`. In many cases, an error in `[ode1,ode2,...]`, even a small one, may lead to a completely different problem than the one `rkf45` is supposed to solve. Furthermore, reducing `absolute_tolerance` is worth a try as well. Trying to get a first solution, perhaps less accurate and in a more narrow interval, is a good idea if `rkf45` is unable to return a solution at a first try. Such a solution can be used as a guideline to get an idea of what went wrong in the first try, and alter optional arguments accordingly, so that a more accurate solution can be achieved.

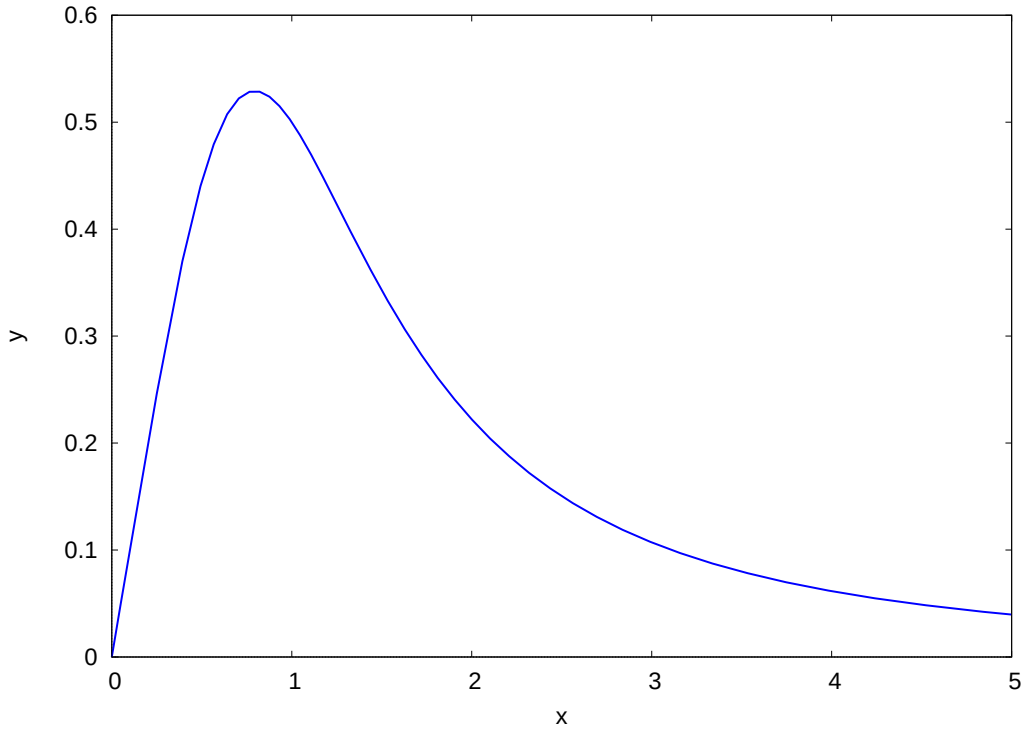
2.2 Examples.

NOTE: Numerical results presented in this paper have been obtained by running the example programs on a Debian GNU/Linux 64 bit system. Results obtained on a different system may differ slightly.

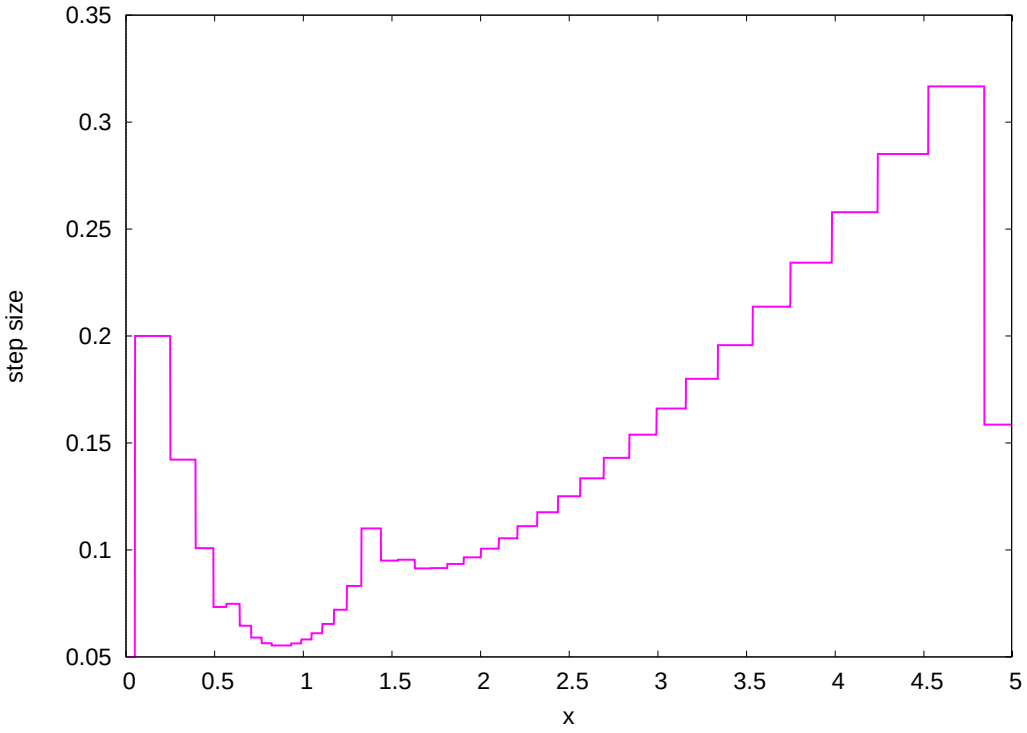
2.2.1 One first-order differential equation.

As a first simple example, consider the initial value problem

$$\frac{dy}{dx} + 3xy^2 - \frac{1}{x^3+1} = 0, \quad y(0) = 0, \tag{1}$$



(a) Numerical solution for problem (1).



(b) Integration steps.

Fig. 1: Graphical output of Listing 1 (part I.)

```
load("rkf45.mac")$

sol:rkf45(-3*x*y^2+1/(x^3+1),y,0,[x,0,5],report=true)$
plot2d([discrete,sol],[style,[lines,4]],[psfile,"Example_1a.eps"])$

x_points:map(first,sol)$
steps:part(x_points,allbut(1))-part(x_points,allbut(length(x_points)))$
x_step(x):=sum(charfun(x_points[i]<=x and x<x_points[i+1])*steps[i],i,1,
               length(steps))$
plot2d(x_step(x),[x,0,last(x_points)-lmin(steps)/10],[style,[lines,4]],
       [color,magenta],[ylabel,"step_size"],[psfile,"Example_1b.eps"])$

y_exact(x):=x/(x^3+1)$errors:map(lambda([u],abs(y_exact(u[1])-u[2])),sol)$
print("Actual_minimum_error:",lmin(part(errors,allbut(1))))$
print("Actual_maximum_error:",lmax(errors))$
plot2d([discrete,x_points,errors],[logy],[style,[lines,4]],[color,red],
       [ylabel,"error"],[psfile,"Example_1c.eps"])$
```

Listing 1: Maxima program for solving Eqs. (1).

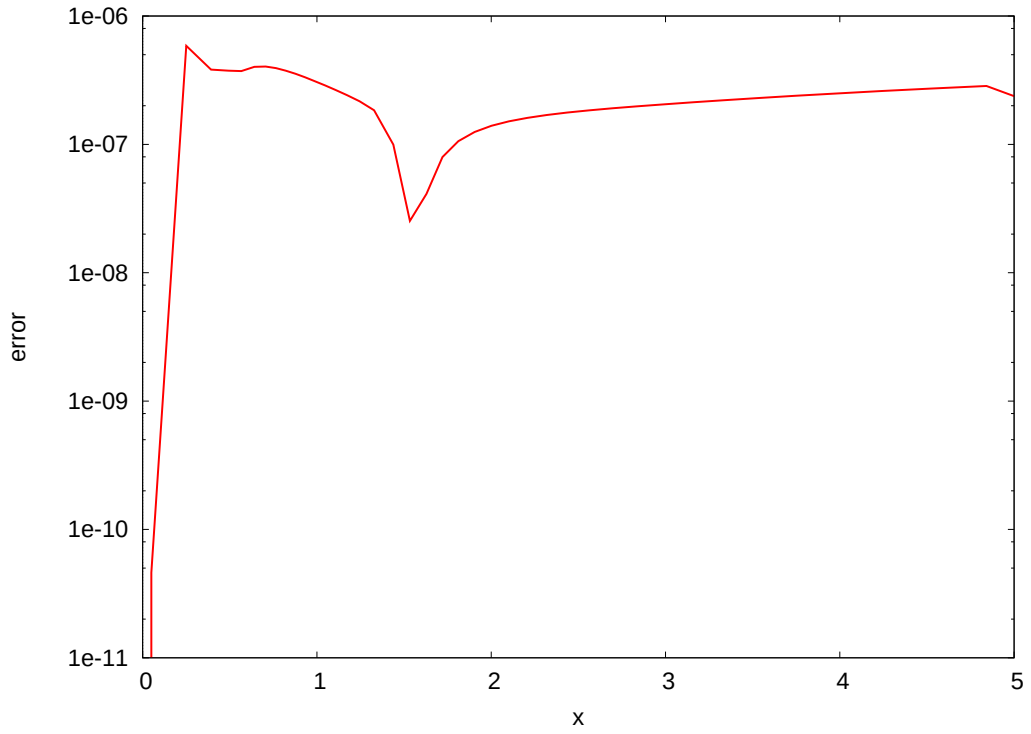


Fig. 2: Graphical output of Listing 1 (part II.) Absolute error in the numerical solution.

with $x \in [0,5]$. In fact, this is a Ricatti equation, and the exact solution is $\frac{x}{x^3+1}$, which will be used to check our results (in Maxima, the exact solution can be obtained by using the package `contrib_ode`.)

Listing 1 shows the Maxima program for solving the problem numerically. Note particularly the call of `rkf45`:

```
rkf45(-3*x*y^2+1/(x^3+1),y,0,[x,0,5])
```

(the optional argument `report=true` is also used in Listing 1, to get a report about calculations done.) Note that we needed to rewrite the differential equation in the form $\frac{dy}{dx} = -3xy^2 + \frac{1}{1+x^3}$. Results returned by `rkf45` are stored in a list, with elements in the form `[x,y]`, where `x` is an integration point (selected by the algorithm,) and `y` is the value of the function $y(x)$ at that point. Solution returned can be used to plot our results, as in Fig. 1a. The evolution of the step size is plotted in Fig. 1b; the algorithm uses small integration steps from $x \approx 0.5$ to $x \approx 2$ because $y(x)$ is changing rather rapidly in that interval. For $x \gtrsim 2$, `rkf45` starts to increase the step size, because the slope of $y(x)$ remains small, so there is no need to use small integration steps. Note that the first step taken is 0.05, which is actually the default value (recall that default initial step is one 100th of the integration interval, in this case $\frac{5}{100} = 0.05$.) Also note that the last integration step is considerably smaller than the previous ones, because the final integration point, in this case $x = 5$, has been reached.

Now, let us check the report about the computations done. In this example, `rkf45` gives the following output, which is rather typical.

```
-----------------------------------------------------
Info: rkf45:
   Integration points selected: 42
    Total number of iterations: 45
           Bad steps corrected: 4
       Minimum estimated error: 3.048850451617402E-10
       Maximum estimated error: 9.5960328100330727E-7
Minimum integration step taken: 0.05
Maximum integration step taken: 0.31667551601085
-----------------------------------------------------
```

There are some interesting things in this information. First, the algorithm found that four steps were bad. There is nothing to worry about that; it simply means that accuracy criterion was not satisfied four times during the computations, so that the corresponding integration steps were rejected, and `rkf45` tried a new, "optimal" step instead. No step is accepted if the estimated error is not less than prescribed accuracy (which is 10^{-6} in this example - the default value, as we didn't used the optional argument `absolute_tolerance`.) Second, 42 integration points were selected by the algorithm (including the initial point.) `rkf45` needed 45 iterations to solve the problem: 41 iterations for the accepted integration points (excluding the initial point) plus four iterations concerning the bad steps, which were not accepted. Third, the minimum error reported by the algorithm is $\approx 3.0 \times 10^{-10}$ (the error is actually zero at the initial point, but `rkf45` does not take that point into account, when calculating estimated errors.) The maximum error is estimated to be $\approx 9.6 \times 10^{-7}$, marginally lower than requested absolute error tolerance, 10^{-6}. In this particular

example, we know the exact solution, so the program can compute actual minimum and maximum errors, which are found to be $\approx 4.6 \times 10^{-11}$ and $\approx 5.9 \times 10^{-7}$, respectively. This verifies that our numerical results are highly accurate. In both cases, the algorithm overestimates the actual error. This is expected, as every numerical method has no means to calculate the actual error; it can only calculate the *local* truncation error, as an estimation of the actual *global* error. This means that the actual error might be lower or even higher than the error estimated by the algorithm. However, the algorithm tries to be conservative, so the difference between actual and estimated error should be small and generally not significant.

In this example, error estimation was accurate enough: actual maximum error, $\approx 5.9 \times 10^{-7}$, is indeed less than 10^{-6}, which was the (default) `absolute_tolerance` requested. Those remarks can also be verified in Fig. 2c, where the absolute error as a function of x is plotted; we see that the error is kept smaller than prescribed accuracy. It is worth mentioning here how `rkf45` increases the initial step, as it estimates that the error is several orders of magnitude smaller than `absolute_tolerance`, so there is no need to keep integrating with such a small step. In general, `rkf45` tries to select the "optimal" step, in the sense that estimated error is kept less than prescribed accuracy, but close to it.

Report also gives some information about the steps taken: the algorithm needed to take several step sizes, varying from 0.05 to ≈ 0.317, depending on the integration point. Note that the minimum step, 0.05, is actually the initial step; this is because a smaller step was not needed in this particular example.

2.2.2 An initial value problem with threshold effect.

Consider the initial value problem

$$\frac{dg}{dt} = s - 1.51g + 3.03\frac{g^2}{g^2+1}, \quad g(0) = 0, \tag{2}$$

with $t \in [0, 100]$. Here, s is a parameter. This is a mathematical model for a biochemical mechanism called *genetic switch* (Brian (2006, pp. 577-579).) It is expected that the solution, $g(t)$, reaches an equilibrium state as $t \to \infty$. What is interesting in that deceptively easy problem is that it exhibits a so called *threshold effect*: for some critical value of s, the equilibrium state of $g(t)$ undergoes an abrupt change to a higher level.

Listing 2 shows the Maxima program which solves Eqs. (2) for three different values of s (the program is simplified a lot by using Maxima's function `makelist`.) The graphical output of the program is shown in Fig. 3 (cf. Brian (2006, Fig. 7.7).) We verify that the initial value problem exhibits the threshold effect, as expected by the theory. The critical value, $s = s_c$, is somewhere between $s = 0.202$ and $s = 0.204$ (we can easily compute the exact value of s_c in Maxima, but this is out of the scope of this paper.) Now, let us examine how `rkf45` reacts on the threshold effect. In the $s = 0.202$ case, `rkf45` reports

```
-----------------------------------------------------
Info: rkf45:
   Integration points selected: 24
    Total number of iterations: 24
           Bad steps corrected: 1
      Minimum estimated error: 8.9798104760311307E-9
```

```
load("rkf45.mac")$
t_start:0$ t_end:100$

sol:makelist(rkf45(equ,g,0,[t,t_start,t_end],report=true),equ,
             makelist(s-1.51*g+3.03*g^2/(1+g^2),s,[0.206,0.204,0.202]))$
plot2d(makelist([discrete,s],s,sol),[style,[lines,4]],[xlabel,"t"],
       [ylabel,"g(t)"],[legend,"s=0.206","s=0.204","s=0.202"],
       [gnuplot_preamble,"set key left"],[psfile,"Example_2a.eps"])$

sol206:rkf45(0.206-1.51*g+3.03*g^2/(1+g^2),g,0,[t,t_start,t_end],
             absolute_tolerance=5e-8,report=true)$
plot2d([discrete,sol206],[style,[linespoints,4]],[xlabel,"t"],
       [ylabel,"g(t)"],[legend,"s=0.206, rkf45 results"],
       [gnuplot_preamble,"set key bottom right"],[psfile,"Example_2b.eps"])$

sol206_rk:rk(0.206-1.51*g+3.03*g^2/(1+g^2),g,0,
             [t,t_start,t_end,(t_end-t_start)/(length(sol206)-1)])$
plot2d([discrete,sol206_rk],[style,[linespoints,4]],[xlabel,"t"],
       [ylabel,"g(t)"],[legend,"s=0.206, rk results"],
       [gnuplot_preamble,"set key bottom right"],[psfile,"Example_2c.eps"])$
```

Listing 2: Maxima program for solving Eqs. (2) for three different values of the parameter s.

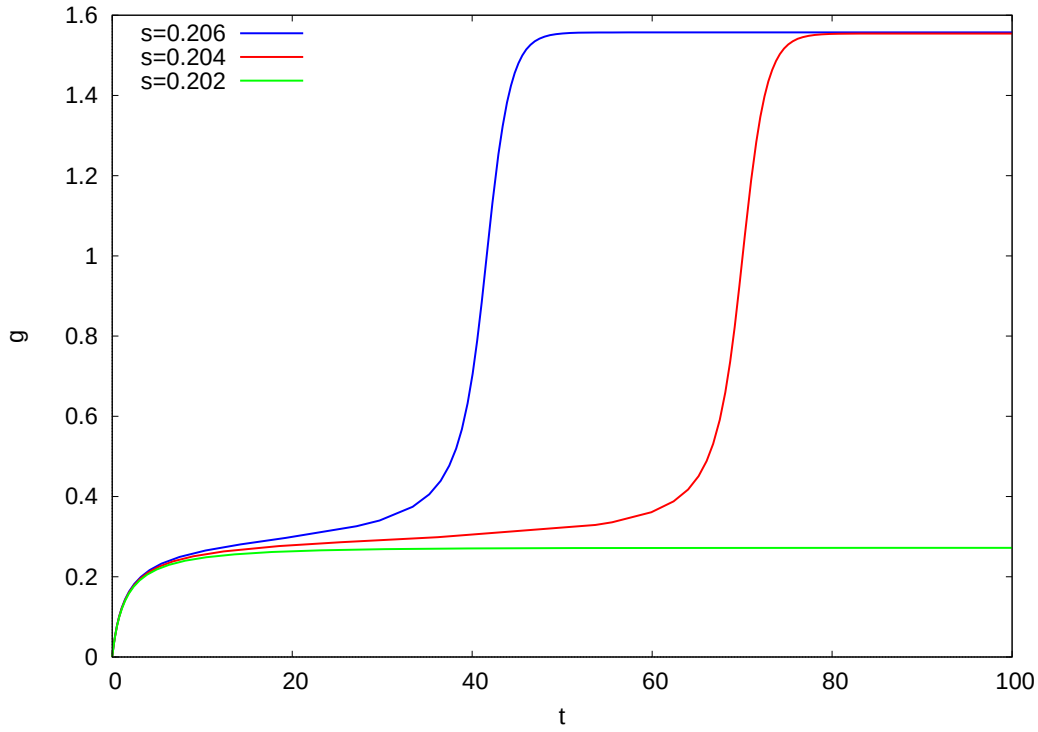


Fig. 3: Graphical output of Listing 2 (part I.)

```
       Maximum estimated error: 3.870532540022239E-7
Minimum integration step taken: 0.15874561265253
Maximum integration step taken: 19.17485694258996
-------------------------------------------------------
```

We see that 24 integration steps were selected to solve the problem. What is interesting here is that one bad step was found, rejected, and refined. It is easy to figure out why that bad step occured. By default, the initial step is set to one 100th of the integration interval, in this case 1. Apparently, that step was too big to get an accurate solution near $t = 0$, where the value of $g(t)$ is changing rapidly. We can easily verify this is the case by using the optional argument `h_start` to set a smaller value for the initial step, say `h_start=0.2`, and run the program again: we get almost the same results, but this time no bad steps are reported in the $s = 0.202$ case. It is not really needed to change the initial step manually, but it is a good example on how optional argument `h_start` can be used to investigate what happened in some special cases.

Now let us compare this report with the corresponding report for $s = 0.206$:

```
-------------------------------------------------------
Info: rkf45:
   Integration points selected: 62
    Total number of iterations: 75
           Bad steps corrected: 14
       Minimum estimated error: 1.3371075149203189E-12
```

```
       Maximum estimated error: 8.6197147915565898E-7
Minimum integration step taken: 0.15685566484769
Maximum integration step taken: 7.845262303547721
-------------------------------------------------------
```

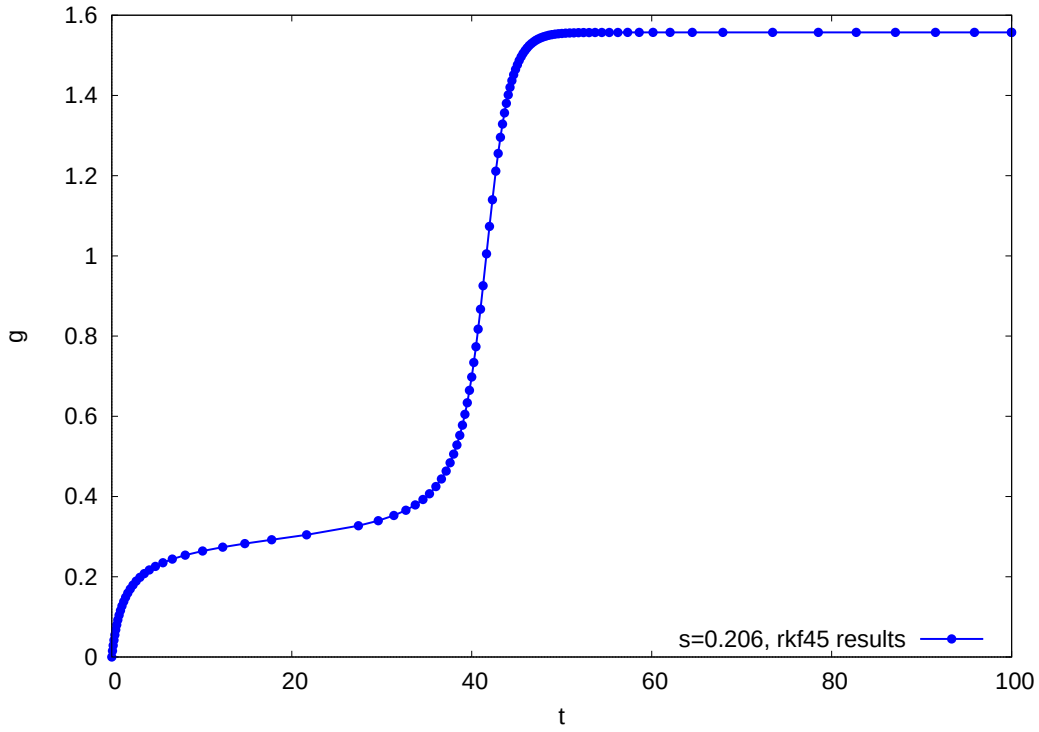
This time, 14 bad steps needed to be corrected. This is because of the high slope of the solution, which forced `rkf45` to refine the step several times, due to the rapid changes in $g(t)$. What is more interesting here is the fact that 62 integration points were selected, about 2.5 times more than in the $s = 0.202$ case. This is how the algorithm reacts to the threshold effect. Solution for $s = 0.206$ is above the critical value, s_c. The high slope of $g(t)$ between $t \simeq 30$ and $t \simeq 50$ was detected by the algorithm, and it reacted by taking more, smaller integration steps, in order to keep the estimated error as small as required; in fact, about 42% of the total integration points lie in $t \in [30, 50]$. This is a typical behavior for adaptive step size methods, and the main reason they are widely used. One can easily imagine what would happen if we used a fixed-size method to solve this problem: the algorithm would not care about the abrupt change in $g(t)$, and it is left to the user to guess how many integration steps should be taken to get an accurate solution.

In order to examine the behavior of `rkf45` compared to that of a fixed step size method, we solve the problem for $s = 0.206$ again, this time with absolute tolerance set to 5×10^{-8}, so that at least seven decimal digits should be accurate. Report in this case reveals that `rkf45` selected more integration points than before:

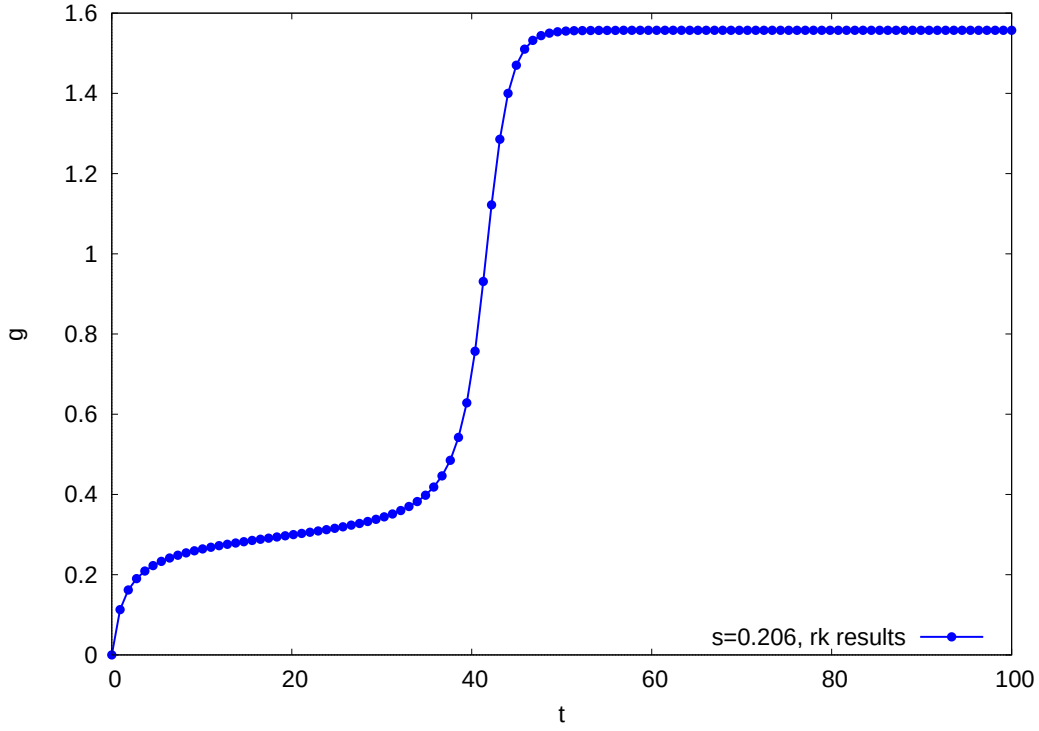
```
-------------------------------------------------------
Info: rkf45:
   Integration points selected: 110
    Total number of iterations: 122
           Bad steps corrected: 13
      Minimum estimated error: 7.5714731382439109E-17
      Maximum estimated error: 4.8700547655067209E-8
Minimum integration step taken: 0.036668228503785
Maximum integration step taken: 5.760239986218677
-------------------------------------------------------
```

We see that 110 points were selected, instead of 62 points, if accuracy is set to default, 10^{-6}. As expected, the error in computations is lower; maximum error is now estimated to be $\approx 4.9 \times 10^{-8}$, compared to $\approx 8.6 \times 10^{-7}$ in the previous case.

We now solve the problem for $s = 0.206$ using Maxima's function `rk`, which is a fixed-step, fourth-order Runge-Kutta method. We set the fixed step size so that `rk` uses 110 integration points (same as in the `rkf45` case above.) Fig. 4 shows our results graphically, and it is easy to see the difference between adaptive and fixed step size methods. An adaptive step size method selects many integration points in the steep parts of the curve, and only a few points in the rest. On the contrary, a fixed step size method just uses the same step size everywhere, regardless of the slope of the curve. In this example, `rkf45` selected $\approx 49\%$ of the total integration points in the most steep part of the curve, $t \in [30, 50]$, while `rk` used only 20% of the total points in the same interval. On the other hand, for $t > 50$, `rkf45` selected $\approx 23\%$ of the total integration points, while `rk` used 50% of the total points; most of them are actually wasted computations, since function $g(t)$ practically reaches an equilibrium state soon after $t = 50$. Note that `rkf45` could select even less integration points in



(a) Numerical solution obtained by `rkf45`. Accuracy is set to 5×10^{-8}.



(b) Numerical solution obtained by `rk`, using same number of integration points as in (a).

Fig. 4: Graphical output of Listing 2 (part II.)

```
load("rkf45.mac")$
k1:0.4*8.8/62/0.03$ k2:0.5*8.8/139/0.2/0.003$ k3:0.4*8.8/139/0.2/0.003$
t_start:0$ t_end:2$

sol:rkf45([k1*(C-L),k2*(32-C)+k3*(L-C)],[L,C],[150,150],[t,t_start,t_end],
          report=true)$
plot2d([[discrete,map(lambda([u],part(u,[1,2])),sol)],
        [discrete,map(lambda([u],part(u,[1,3])),sol)]],
       [style,[lines,4]],[xlabel,"time (hours)"],[ylabel,"temperature (F)"],
       [legend,"liquid (L(t)","container (C(t)"],[psfile,"Example_3a.eps"])$
plot2d([[discrete,map(lambda([u],part(u,[1,2])),sol)],
        [discrete,map(lambda([u],part(u,[1,3])),sol)]],[x,0,0.3],
       [style,[linespoints,4]],[point_type,bullet],[xlabel,"time (hours)"],
       [ylabel,"temperature (F)"],[legend,"liquid (L(t)","container (C(t)"],
       [psfile,"Example_3b.eps"])$
```

Listing 3: Maxima program for solving Eqs. (3).

that region, but algorithm implementation is rather conservative, and does not allow too big step size changes.

2.2.3 A system of two first-order differential equations.

In this section we shall see how to use `rkf45` in order to solve a system of two first-order differential equations with respect to two initial conditions,

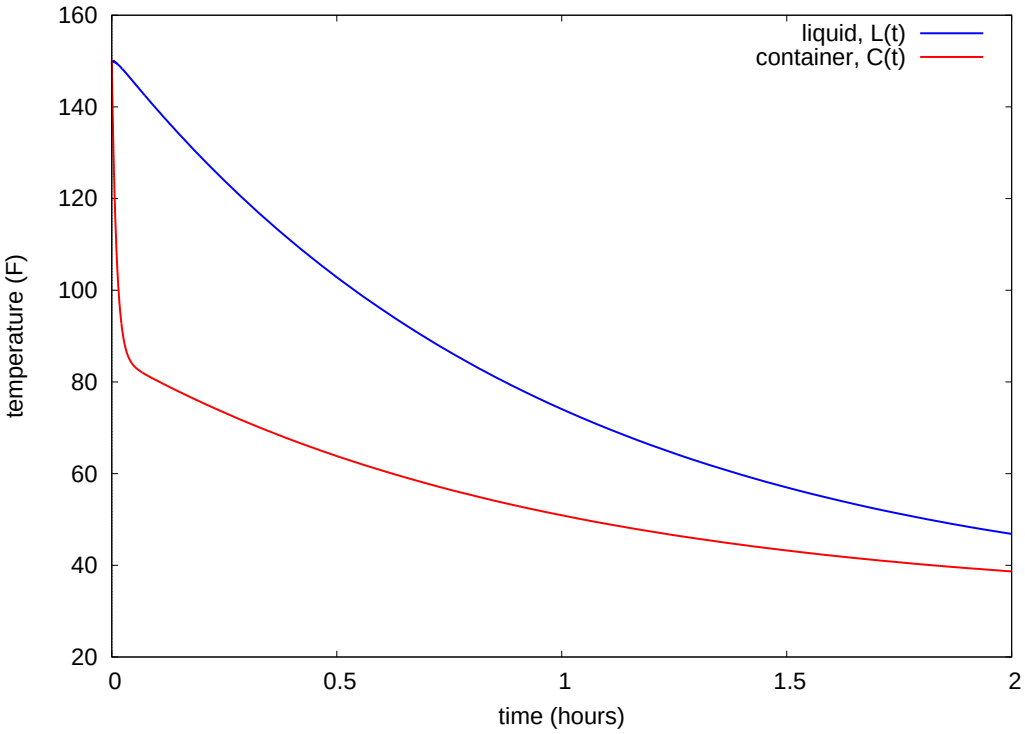
$$\left\{\begin{array}{rcl}\frac{dL}{dt} &=& k_1\left(C-L\right)\\ \frac{dC}{dt} &=& k_2\left(32-C\right)+k_3\left(L-C\right)\end{array}\right.,\quad \left\{\begin{array}{rcl}L\left(0\right) &=& 150\\ C\left(0\right) &=& 150\end{array}\right., \tag{3}$$

with $t\in[0,2]$. Eqs. (3) describe the cooling of a container and its liquid contains. Here, t is the time, L, C are the temperatures of the liquid and its container, respectively, and k_1, k_2, k_3 are constants (see Brian (2006, pp. 626-627) for details.) Listing 3 shows the Maxima program used to solve the problem. Notice how `rkf45` is called in this case,

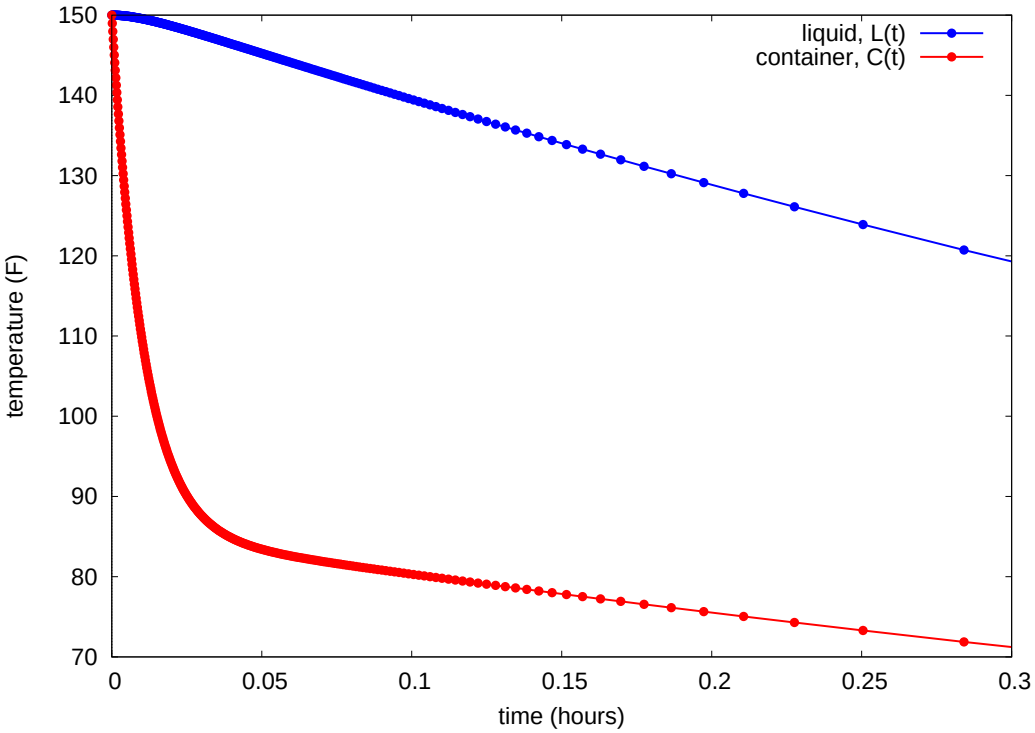
```
rkf45([k1*(C-L),k2*(32-C)+k3*(L-C)],[L,C],[150,150],[t,t_start,t_end])
```

(we also used the optional argument `report=true` in Listing 3, to get an idea of the calculations done.) Result is stored in a list, with elements in the form `[t,L,C]`, where `t` is the integration point, and `L`, `C` are the values of the functions L and C at `t`. The graphical output of the program is shown in Fig. 5 (cf. Brian (2006, Fig. 7.16).) Note particularly that the container is cooled much faster that its liquid contents, and function $C\left(t\right)$ is decreasing rapidly in the beginning. In this case, report printed by `rkf45` is as follows

```
-------------------------------------------------------
Info: rkf45:
```



(a) Functions $L(t)$ and $C(t)$.



(b) Magnification near $t = 0$.

Fig. 5: Graphical output of Listing 3.

```
load("rkf45.mac")$
t_start:0$ t_end:20$ mu:4$
equ:'diff(x,t,2)+mu*(x^2-1)*'diff(x,t)+x=0$
equ2:['diff(x1,t)=x2,ev(solve(equ,'diff(x,t,2))[1],'diff(x,t,2)='diff(x2,t),
                                                    'diff(x,t)=x2,x=x1)];
equ2:map(rhs,equ2);

sol:rkf45(equ2,[x1,x2],[0.75,0],[t,t_start,t_end],report=true)$
plot2d([[discrete,map(lambda([u],part(u,[1,2])),sol)],
        [discrete,map(lambda([u],part(u,[1,3])),sol)]],
       [style,[lines,4]],[xlabel,"t"],[ylabel,"x(t), x'(t)"],
       [legend,"x(t)","x'(t)"],[psfile,"Example_4a.eps"])$
plot2d([discrete,map(lambda([u],part(u,[2,3])),sol)],[style,lines[4]],
       [color,magenta],[xlabel,"x"],[ylabel,"x'"],[psfile,"Example_4b.eps"])$
```

Listing 4: Maxima program for solving Eqs. (5).

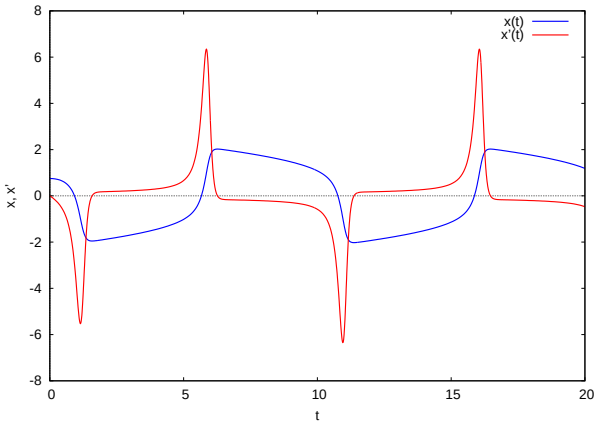
```
   Integration points selected: 316
    Total number of iterations: 332
           Bad steps corrected: 17
      Minimum estimated error: 5.4650657325752074E-8
      Maximum estimated error: 9.7738230565866937E-7
Minimum integration step taken: 1.6519396700522609E-4
Maximum integration step taken: 0.044380874549758
-------------------------------------------------
```

We see that the algorithm selected 316 integration points in this case, much more than in previous examples. This is because a very small step was needed for small t, due to the rapidly decreasing function $C(t)$, and because of the high accuracy required. In fact, 234 out of 316 points (about 74%) were needed for a small time interval, from $t = 0$ to $t = 0.1$ (5% of the total time interval.) This fact is more pronounced in 5b. Furthermore, the fact we solved the problem using default accuracy, 10^{-6}, is an exaggeration, considering that $C(t) \gtrsim 40\,^{\circ}\mathrm{F}$. We can therefore safely reduce the accuracy needed to a more plausible value, say 5×10^{-3}, so that results should still be accurate to *at least* two decimal digits. Indeed, adding `accuracy=5e-3` in the call of `rkf45` would reduce the integration points to 93 (≈ 3.4 times less than above,) without any noticeable loss of accuracy.

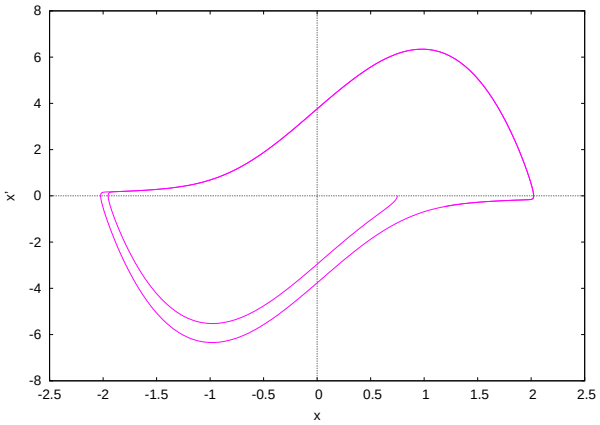
2.2.4 A second-order differential equation: the van der Pol equation.

As an example of a second-order differential equation, we shall solve the initial value problem described by the well-known *van der Pol equation* and two initial conditions (see, e.g., Hairer et al. (1993, § I.16), Hairer & Wanner (2002, Eq. 1.5), Brian (2006, Example 7.28))

$$\frac{dx^2}{dt^2} + \mu\left(x^2 - 1\right)\frac{dx}{dt} + x = 0, \quad \left\{ \begin{array}{rcl} x(0) & = & 0.75 \\ \frac{dx}{dt}\big|_{x=0} & = & 0 \end{array} \right. , \tag{4}$$



(a) Numerical solution.



(b) Phase portrait.

Fig. 6: Graphical output of Listing 4.

where $t \in [0, 20]$, and μ is a positive parameter. This is an oscillator with non-linear dumping; what makes this problem interesting is exactly the damping term, $\mu\left(x^2-1\right)\frac{dx}{dt}$. For $|x| > 1$, the damping coefficient, $\mu\left(x^2-1\right)$, is positive, and thus the damping term acts as friction or resistance, draining energy from the system. However, if $|x| < 1$ the damping coefficient is negative, and the term acts as "negative resistance", supplying energy to the system. From a computational point of view, the dumping parameter, μ, is crucial; a small value of μ makes the damping term negligible, and the problem is very easy to solve. On the other hand, a higher value of μ makes the problem stiff, and thus much harder to solve numerically. In this example, we take $\mu = 4$, a rather small value.

The initial value problem (4) can be solved by transforming the original second-order equation to a system of two first-order differential equations. This can done by putting $x = x_1$, $\frac{dx}{dt} = x_2$ (and, consequently, $\frac{dx^2}{dt^2} = \frac{dx_2}{dt}$). so that Eqs. (4) are now written as

$$\left\{\begin{array}{lcl} \frac{dx_1}{dt} & = & x_2 \\ \frac{dx_2}{dt} & = & -\mu\left(x_1^2-1\right)x_2 - x_1 \end{array}\right. , \quad \left\{\begin{array}{lcl} x_1(0) & = & 0.75 \\ x_2(0) & = & 0 \end{array}\right. . \tag{5}$$

Transformation can be easily done in Maxima, as shown in Listing 4. Eqs. (5) can now be solved by `rkf45`. just as any other system of first-order differential equations. Solution is returned as a list, with elements in the form `[t,x1,x2]`, where `t` is the integration point, and `x1`, `x2` are the values of the functions x_1, x_2 (or, equivalently, x, $\frac{dx}{dt}$) at that point. Graphical output of the Maxima program 4 is shown in Fig. 6, where the non-linear oscillatory nature of the solution is apparent (cf., e.g., Brian (2006, Fig. 7.18).)

In this example, the absolute tolerance is set to 10^{-6} (the default value,) and the report returned by `rkf45` is as follows.

```
-------------------------------------------------------
Info: rkf45:
   Integration points selected: 632
    Total number of iterations: 652
           Bad steps corrected: 21
```

```
        Minimum estimated error: 2.839347393139819E-8
        Maximum estimated error: 9.914713641306354E-7
Minimum integration step taken: 0.0068707778178574
Maximum integration step taken: 0.10198351520456
-------------------------------------------------------
```

Here 632 integration points were selected, and the maximum estimated error is $\approx 9.9 \times 10^{-7}$. The high number of integration points is caused by the slopes of the curves, especially $\frac{dx}{dt}$ (see Fig. 6a.) Although computation time is not an issue in this example, computations needed can be reduced considerably by reducing `absolute_tolerance`. For example, if four accurate decimal digits are enough, we can set `absolute_tolerance=5e-5`, and `rkf45` would now need 252 points; that is, computations needed would be reduced by about 60%, but the result would still be very close to what we get with more than double the computational effort.

2.2.5 A system of two second order differential equations: the double pendulum.

The double pendulum is a simple physical system consisting of one pendulum attached to another. Despite its simplicity, it exhibits rich dynamic behavior, varying from a simple linear system to a chaotic system. One can easily derive the Lagrangian and the equations of motion using Maxima, but we shall concentrate on the result of that algebra, which is two coupled second-order differential equations; together with four initial conditions, they form the initial value problem,

$$\left\{\begin{array}{lcl} \frac{d^2\theta_1}{dt^2} & = & \frac{-g(2m_1+m_2)\sin\theta_1 - m_2 g\sin(\theta_1-2\theta_2) - 2\sin(\theta_1-\theta_2)m_2\left(\left(\frac{d\theta_2}{dt}\right)^2\ell_2+\left(\frac{d\theta_1}{dt}\right)^2\ell_1\cos(\theta_1-\theta_2)\right)}{\ell_1(2m_1+m_2-m_2\cos(2\theta_1-2\theta_2))} \\ \frac{d^2\theta_2}{dt^2} & = & \frac{2\sin(\theta_1-\theta_2)\left(\left(\frac{d\theta_1}{dt}\right)^2\ell_1(m_1+m_2)+g(m_1+m_2)\cos\theta_1+\left(\frac{d\theta_2}{dt}\right)^2\ell_2 m_2\cos(\theta_1-\theta_2)\right)}{\ell_2(2m_1+m_2-m_2\cos(2\theta_1-2\theta_2))} \end{array}\right. , \tag{6}$$

$$\theta_1(0) = \frac{\pi}{8}, \quad \theta_2(0) = \frac{\pi}{4}, \quad \left.\frac{d\theta_1}{dt}\right|_{t=0} = 0, \quad \left.\frac{d\theta_2}{dt}\right|_{t=0} = 0, \tag{7}$$

where g is is the local acceleration of gravity, m_1, m_2 are the two masses, ℓ_1, ℓ_2 are the lengths of the two rods, and θ_1, θ_2 are the angles that each pendulum swings away from vertical downwards (as usual, counter-clockwise angles are positive.) In order to solve this problem numerically, we need to transform Eqs. (6) to a system of four first-order differential equations, and transform the initial conditions (7) accordingly. The procedure is similar to that of § 2.2.4: We put $\frac{d\theta_1}{dt} = \omega_1$, $\frac{d\theta_2}{dt} = \omega_2$ (angular velocities of the two rods,) and, consequently, $\frac{d^2\theta_1}{dt^2} = \frac{d\omega_1}{dt}$, $\frac{d^2\theta_2}{dt^2} = \frac{d\omega_2}{dt}$, so that the problem is now written as

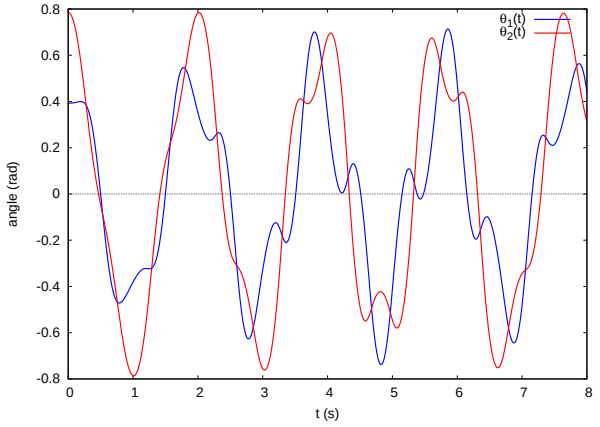
$$\left\{\begin{array}{lcl} \frac{d\theta_1}{dt} & = & \omega_1 \\ \frac{d\theta_2}{dt} & = & \omega_2 \\ \frac{d\omega_1}{dt} & = & \frac{-g(2m_1+m_2)\sin\theta_1 - m_2 g\sin(\theta_1-2\theta_2) - 2\sin(\theta_1-\theta_2)m_2\left(\omega_2^2\ell_2+\omega_1^2\ell_1\cos(\theta_1-\theta_2)\right)}{\ell_1(2m_1+m_2-m_2\cos(2\theta_1-2\theta_2))} \\ \frac{d\omega_2}{dt} & = & \frac{2\sin(\theta_1-\theta_2)\left(\omega_1^2\ell_1(m_1+m_2)+g(m_1+m_2)\cos\theta_1+\omega_2^2\ell_2 m_2\cos(\theta_1-\theta_2)\right)}{\ell_2(2m_1+m_2-m_2\cos(2\theta_1-2\theta_2))} \end{array}\right. , \tag{8}$$

$$\theta_1(0) = \frac{\pi}{8}, \quad \theta_2(0) = \frac{\pi}{4}, \quad \omega_1(0) = 0, \quad \omega_2(0) = 0. \tag{9}$$

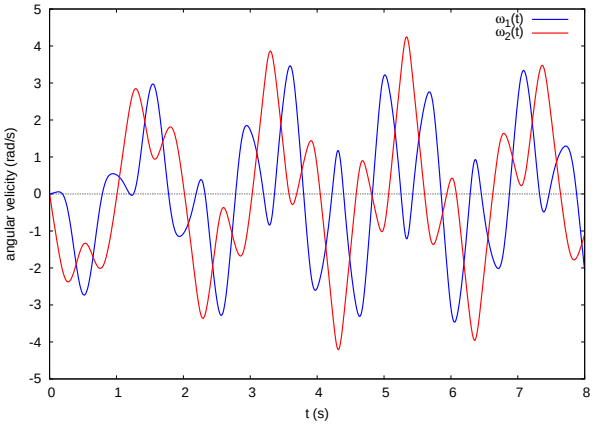
The Maxima program that solves this problem is shown in Listing 5. The program is quite similar than the previous ones (it is more lengthy mainly because several quantities are plotted.)

```
load("rkf45.mac")$
m1:1$ m2:1.5$ l1:0.4$ l2:0.6$ g:9.81$ t_start:0$ t_end:8$
d2th1dt2:(-g*(2*m1+m2)*sin(th1)-m2*g*sin(th1-2*th2)
          -2*sin(th1-th2)*m2*('diff(th2,t)^2*l2+'diff(th1,t)^2*l1*cos(th1-th2)))
         /(l1*(2*m1+m2-m2*cos(2*th1-2*th2)))$
d2th2dt2:(2*sin(th1-th2)*('diff(th1,t)^2*l1*(m1+m2)+g*(m1+m2)*cos(th1)
          +'diff(th2,t)^2*l2*m2*cos(th1-th2)))
         /(l2*(2*m1+m2-m2*cos(2*th1-2*th2)))$
equs:ev([omega1,omega2,d2th1dt2,d2th2dt2],'diff(th1,t)=omega1,
        'diff(th2,t)=omega2);
th1_start:float(%pi/8)$ th2_start:float(%pi/4)$ omega1_start:0$ omega2_start:0$

sol:rkf45(equs,[th1,th2,omega1,omega2],
          [th1_start,th2_start,omega1_start,omega2_start],[t,t_start,t_end],
          report=true)$
plot2d([[discrete,map(lambda([u],part(u,[1,2])),sol)],
        [discrete,map(lambda([u],part(u,[1,3])),sol)]],
       [style,[lines,4]],[xlabel,"t (s)"],[ylabel,"angle (rad)"],
       [legend,"{/Symbol q}_1(t)","{/Symbol q}_2(t)"],
       [psfile,"Example_5a.eps"])$
plot2d([[discrete,map(lambda([u],part(u,[1,4])),sol)],
        [discrete,map(lambda([u],part(u,[1,5])),sol)]],
       [style,[lines,4]],[xlabel,"t (s)"],[ylabel,"angular velicity (rad/s)"],
       [legend,"{/Symbol w}_1(t)","{/Symbol w}_2(t)"],
       [psfile,"Example_5b.eps"])$
plot2d([discrete,map(lambda([u],part(u,[2,3])),sol)],[style,[lines,4]],
       [xlabel,"{/Symbol q}_1 (rad)"],[ylabel,"{/Symbol q}_2 (rad)"],
       [color,magenta],[legend,false],[psfile,"Example_5c.eps"])$
plot2d([discrete,map(lambda([u],part(u,[4,5])),sol)],[style,[lines,4]],
       [xlabel,"{/Symbol w}_1 (rad/s)"],[ylabel,"{/Symbol w}_2 (rad/s)"],
       [color,magenta],[legend,false],[psfile,"Example_5d.eps"])$
plot2d([discrete,map(lambda([u],part(u,[2,4])),sol)],[style,[lines,4]],
       [xlabel,"{/Symbol q}_1 (rad)"],[ylabel,"{/Symbol w}_1 (rad/s)"],
       [color,green],[legend,false],[psfile,"Example_5e.eps"])$
plot2d([discrete,map(lambda([u],part(u,[3,5])),sol)],[style,[lines,4]],
       [xlabel,"{/Symbol q}_2 (rad)"],[ylabel,"{/Symbol w}_2 (rad/s)"],
       [color,green],[legend,false],[psfile,"Example_5f.eps"])$
```

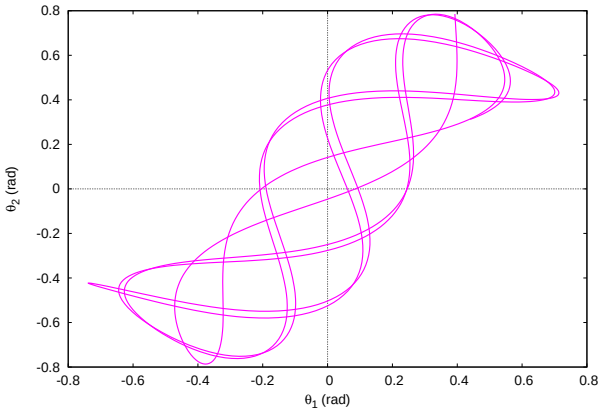
Listing 5: Maxima program for solving Eqs. (8-9).



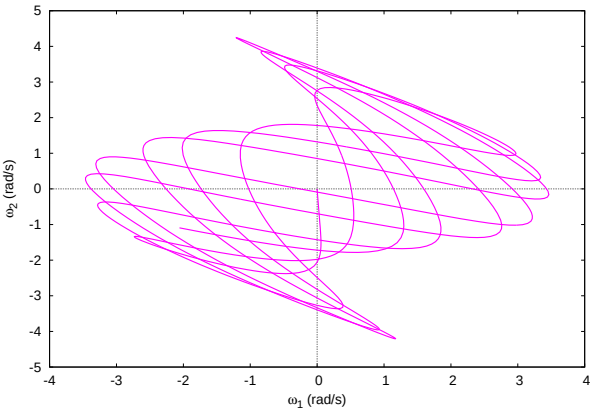
(a) Angles $\theta_1(t)$ and $\theta_2(t)$.



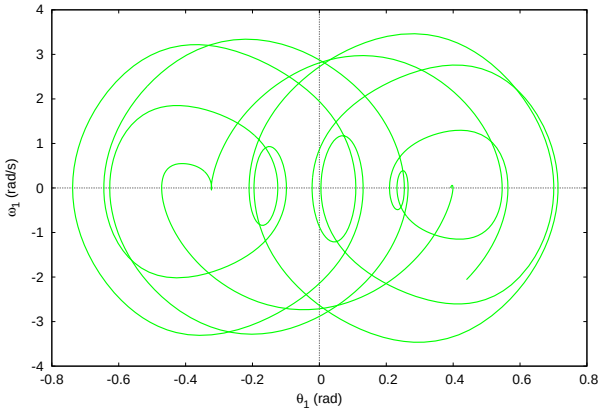
(b) Angular velocities $\omega_1(t)$ and $\omega_2(t)$.



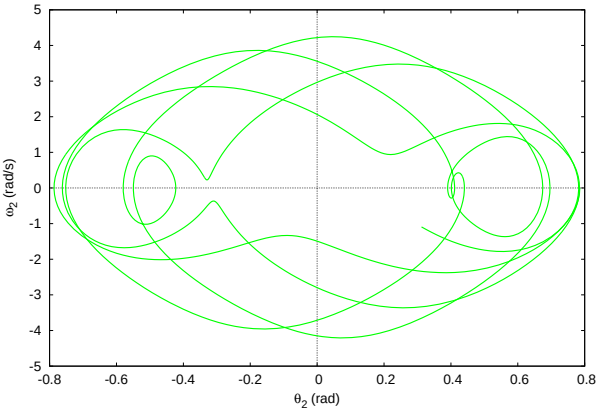
(c) θ_2 vs. θ_1.



(d) ω_2 vs. ω_1.



(e) ω_1 vs. θ_1.



(f) ω_2 vs. θ_2.

Fig. 7: Graphical output of Listing 5.

In this particular example, we take $m_1 = 1\,\mathrm{kg}$, $m_2 = 1.5\,\mathrm{kg}$, $\ell_1 = 0.4\,\mathrm{m}$, $\ell_2 = 0.6\,\mathrm{m}$, and $g = 9.81\,\mathrm{m/s^2}$. Transformation from Eqs. (6-7) to Eqs. (8-9) and numerical solution via `rkf45` is similar to that in Listing. 4. The transformed system is solved by `rkf45` as a system of four first-order differential equations (for $t \in [0, 8]$.) Solution is returned as a list, with elements in the form `[t,th1,th2,omega1,omega2]`, where `t` is the integration point, and `th1`, `th2`, `omega1`, `omega2` are the values of the functions θ_1, θ_2, ω_1, ω_2 at that point. Using that information, the program creates several plots, shown in Fig. 7. Report returned by `rkf45` shows that 845 integration points were used:

```
-----------------------------------------------------
Info: rkf45:
   Integration points selected: 845
    Total number of iterations: 877
           Bad steps corrected: 33
      Minimum estimated error: 2.350379310893348E-8
      Maximum estimated error: 9.7924074620423794E-7
Minimum integration step taken: 0.0056717734638438
Maximum integration step taken: 0.01858201039005
-----------------------------------------------------
```

The maximum error is estimated to be $\approx 9.8 \times 10^{-7}$. As in the previous example, setting accuracy to 5×10^{-5}, reduces the number of integration points by about 60%.

2.2.6 The Pleiades problem.

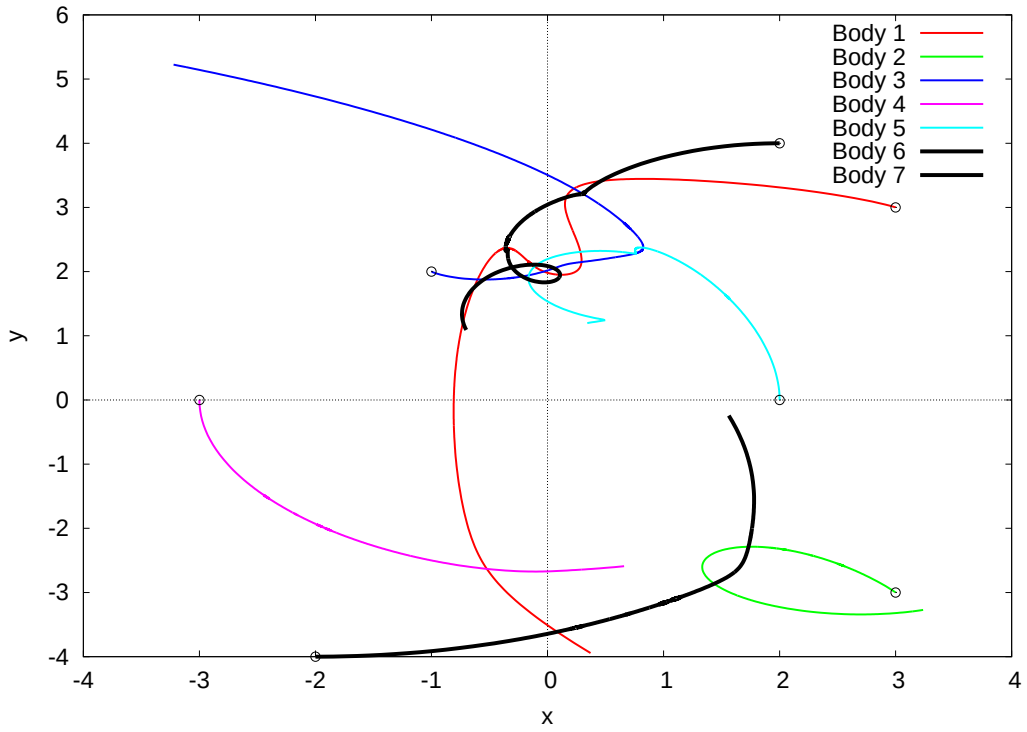
The *Pleiades problem* is an artificial celestial mechanics problem of seven stars moving in the same plane. Mathematically, it is described by a system of 14 second-order differential equations, together with 28 initial conditions, defining initial position and velocity for each celestial body. The problem is not stiff, but it is complex enough. Because of its complexity, it is included in several collections of test problems for ordinary differential equation solvers (see, e.g., Mazzia & Magherini (2008); Nowak et al. (2010).) Traditionally, the problem is solved for seven celestial objects, apparently as a simple simulation of the well-known *Pleiades star cluster*. However, the problem can be easily expanded for any number of objects, not necessarily in the same plane. In this paper we shall use formulation and data taken from Hairer et al. (1993), as all collections of test problems do.

For every object in the plane, position is defined by the vector function $\overrightarrow{r_i} = x_i\overrightarrow{u_x} + y_i\overrightarrow{u_y}$, where $i = 1, \ldots, 7$, and $\overrightarrow{u_x}$, $\overrightarrow{u_y}$ are the unit vectors along the $x-$ and $y-$axis, respectively. Similarly, velocity is defined by the vector functions and $\frac{d\overrightarrow{r_i}}{dt} = \frac{dx_i}{dt}\overrightarrow{u_x} + \frac{dy_i}{dt}\overrightarrow{u_y}$. The unknown functions of the problem are x_i, y_i, $\frac{dx_i}{dt}$, $\frac{dy_i}{dt}$ for $i = 1, \ldots, 7$, and they are all functions of time, t. It is easy to derive the 14 equations of motion for such a system; together with the initial conditions generally adopted in all similar tests, they form the initial value problem

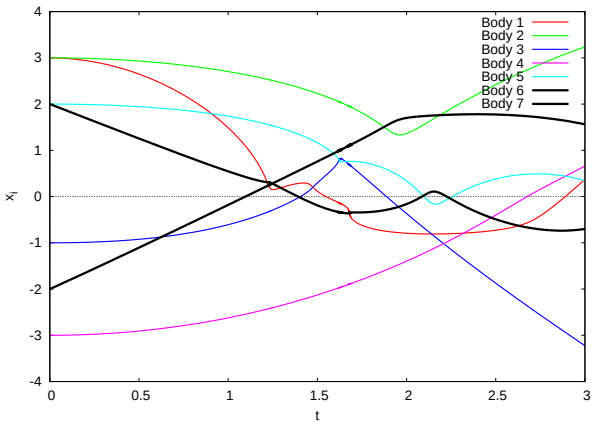
$$\left\{\begin{array}{rcl} \frac{d^2x_i}{dt^2} & = & \sum_{j\neq i} m_j\left(x_i - x_j\right)/r_{ij}^{\frac{3}{2}} \\ \frac{d^2y_i}{dt^2} & = & \sum_{j\neq i} m_j\left(y_i - y_j\right)/r_{ij}^{\frac{3}{2}} \end{array}\right. , \quad \left\{\begin{array}{rcl} x_i\left(0\right) & = & \left[3, 3, -1, -3, 2, -2, 2\right]^T \\ y_i\left(0\right) & = & \left[3, -3, 2, 0, 0, -4, 4\right]^T \\ \left.\frac{dx_i}{dt}\right|_{t=0} & = & \left[0, 0, 0, 0, 0, 1.75, -1.5\right]^T \\ \left.\frac{dy_i}{dt}\right|_{t=0} & = & \left[0, 0, 0, -1.25, 1, 0, 0\right]^T \end{array}\right. , \qquad (10)$$

```
load("rkf45.mac")$

fx(i):=sum(if j#i then j*(concat(x,j)-concat(x,i))/
                       ((concat(x,i)-concat(x,j))^2+
                        (concat(y,i)-concat(y,j))^2)^1.5
           else 0,j,1,7)$
fy(i):=sum(if j#i then j*(concat(y,j)-concat(y,i))/
                       ((concat(x,i)-concat(x,j))^2+
                        (concat(y,i)-concat(y,j))^2)^1.5
           else 0,j,1,7)$
equs:flatten([makelist(concat(dx,i),i,1,7),makelist(concat(dy,i),i,1,7),
              makelist(fx(i),i,1,7),makelist(fy(i),i,1,7)])$
funcs:flatten([makelist(concat(x,i),i,1,7),makelist(concat(y,i),i,1,7),
               makelist(concat(dx,i),i,1,7),makelist(concat(dy,i),i,1,7)])$
init:[3,3,-1,-3,2,-2,2,3,-3,2,0,0,-4,4,0,0,0,0,0,1.75,-1.5,0,0,0,-1.25,1,0,0]$
t_start:0$ t_end:3$

sol:rkf45(equs,funcs,init,[t,t_start,t_end],report=true)$

initial_points:[discrete,makelist([init[i],init[i+7]],i,1,7)]$
trajectories:makelist([discrete,map(lambda([u],part(u,[1+i,8+i])),sol)],i,1,7)$
styles:append([style],makelist([lines,4],i,1,7),[points])$
colors:[color,red,green,blue,magenta,cyan,yellow,black]$
legends:append([legend],makelist(sconcat("Body ",i),i,1,7),[""])$
plot2d(endcons(initial_points,trajectories),styles,[point_type,circle],
       endcons(black,colors),[xlabel,"x"],[ylabel,"y"],legends,
       [psfile,"Example_6a.eps"])$
plot2d(makelist([discrete,map(lambda([u],part(u,[1,1+i])),sol)],i,1,7),styles,
       colors,[xlabel,"t"],[ylabel,"x_i (t)"],legends,
       [psfile,"Example_6b.eps"])$
plot2d(makelist([discrete,map(lambda([u],part(u,[1,8+i])),sol)],i,1,7),styles,
       colors,[xlabel,"t"],[ylabel,"y_i (t)"],legends,
       [psfile,"Example_6c.eps"])$
```

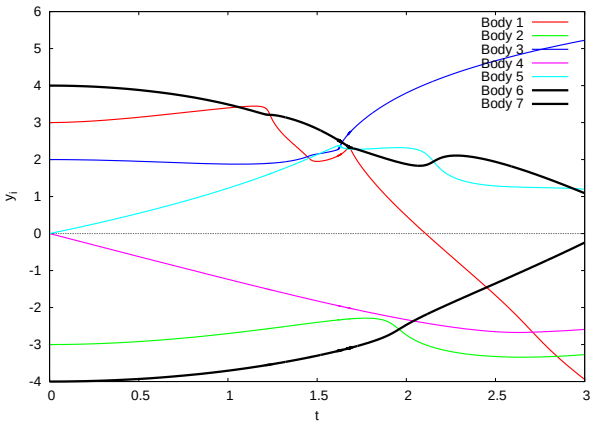
Listing 6: Maxima program for solving Eqs. (11).



(a) Trajectories of the seven bodies (circles mark the initial position for each body.)



(b) Abscissae $x_i(t)$.



(c) Ordinates $y_i(t)$.

Fig. 8: Graphical output of Listing 6.

where m_j are the masses of the objects, and $r_{ij} = (x_i - x_j)^2 + (y_i - y_j)^2$. In this example, we take $m_j = j$, and the problem is solved for $t \in [0, 3]$.

In order to solve Eqs. (10), we need to transform the second-order differential equations to a system of first-order differential equations. This is easily done by putting $\frac{dx_i}{dt} = \chi_i$, $\frac{dy_i}{dt} = \psi_i$, so that Eqs. (10) are now written as

$$\left\{ \begin{array}{lcl} \frac{dx_i}{dt} & = & \chi_i \\ \frac{dy_i}{dt} & = & \psi_i \\ \frac{d\chi_i}{dt} & = & \sum_{j \neq i} m_j \left(x_i - x_j\right) / r_{ij}^{\frac{3}{2}} \\ \frac{d\psi_i}{dt} & = & \sum_{j \neq i} m_j \left(y_i - y_j\right) / r_{ij}^{\frac{3}{2}} \end{array} \right. , \quad \left\{ \begin{array}{lcl} x_i\left(0\right) & = & \left[3, 3, -1, -3, 2, -2, 2\right]^T \\ y_i\left(0\right) & = & \left[3, -3, 2, 0, 0, -4, 4\right]^T \\ \chi_i\left(0\right) & = & \left[0, 0, 0, 0, 0, 1.75, -1.5\right]^T \\ \psi_i\left(0\right) & = & \left[0, 0, 0, -1.25, 1, 0, 0\right]^T \end{array} \right. . \tag{11}$$

Listing 6 shows the Maxima program for solving the initial value problem (11). It is worth mentioning here that Maxima offers the ability to define the set of differential equations in a very elegant way, using built-in functions `concat` and `makelist`. `rkf45` returns the solution as a list, with elements in the form `[t,x1,...,x7,y1,...,y7,dx1,...,dx7,dy1,...,dy7]`, where `t` is the integration point, and `x1,...,x7`, `y1,...,y7`, `dx1,...,dx7`, `dy1,...,dy7` are the values of the functions x_i, y_i, $\chi_i = \frac{dx_i}{dt}$, $\psi_i = \frac{dy_i}{dt}$ at that point. Graphical output of the program is shown in Fig. 8. Report of `rkf45` is as follows.

```
-----------------------------------------------------
Info: rkf45:
   Integration points selected: 1143
    Total number of iterations: 1144
           Bad steps corrected: 2
       Minimum estimated error: 3.8961905793680196E-8
       Maximum estimated error: 9.8286131077889909E-7
Minimum integration step taken: 2.5152068765271884E-5
Maximum integration step taken: 0.065073956846732
-----------------------------------------------------
```

The algorithm selected 1143 integration points. In a modern personal computer, `rkf45` solves the above problem in somewhat less than eight seconds. Compared to previous examples, this computation time is at least seven times larger. This is expected, considering the fact we are solving a set of 28 non-linear differential equations; the term $r_{ij}^{-\frac{3}{2}} = \left(\left(x_i - x_j\right)^2 + \left(y_i - y_j\right)^2\right)^{-\frac{3}{2}}$, involved in 14 differential equations, is also a reason of the larger computation time. As in previous examples, reducing accuracy, even slightly, would reduce computation time considerably. For example, setting absolute tolerance to 5×10^{-5} reduces integration points and computation time by about 62%.

3 Stiff initial value problems.

3.1 General discussion.

Although a single, precise definition of "stiffness" does not exist, an initial value problem is generally considered as *stiff* if explicit numerical methods either work very slowly, or they don't work at all, because the integration step needs to be so small that it is practically impossible to solve the

problem. Such a small step size is necessary not to achieve accuracy requirements, but because numerical integration becomes unstable otherwise, giving completely erroneous results. In other words, the step size is dictated by stability requirements rather than by accuracy requirements. The most common reason for this behavior is a set of differential equations where the derivatives of the unknown functions depend on the functions themselves in a very strong way. Typically, this means that solution may vary rather slowly in the majority of the integration interval, but there is at least one region, often very narrow, where functions vary rapidly, or even extremely rapidly, in very stiff problems. Another reason of stiffness might be unusual initial conditions; the reader can find a detailed analysis of the term "stiffness" in Hairer & Wanner 2002.

It is worth emphasizing that not all difficult problems are stiff, but all stiff problems are difficult for solvers not specifically designed for them. Special methods, designed for solving stiff differential equations, do exist in the literature; most notably, methods based on the backward differentiation formula, also called *Gear methods*, are known to work very well on stiff problems. More sophisticated algorithms, able to switch from non-stiff to stiff mode and vice versa, do exist as well. For example, the package *ODEPACK* (Hindmarsh (1983)) is an excellent collection of Fortran solvers for both non-stiff and stiff initial value problems. Among them, the LSODAR solver (and its double-precision version, DLSODAR) switches automatically between non-stiff and stiff methods (Petzold (1983).) In addition, DLSODAR is able to find the root of at least one of a set of constraint functions of the independent and dependent variables, and stop integration at that point; this feature is particularly useful for solving initial value problems where the domain of integration is not known in advance (see, e.g., Papasotiriou et al. (2007).)

`rkf45` is an explicit Runge-Kutta method, and, as such, it is not specifically designed for solving stiff differential equations. Its ability to adapt the step size makes it capable to deal with such difficult problems, at least in several cases. However, as all explicit methods, and despite its adaptive step size, `rkf45` still needs many integration steps to solve a stiff problem, while a special method, designed to solve stiff systems, would need much less integration points to achieve the same accuracy (but at higher computational cost per step.) Consequently, `rkf45`, as every other explicit method, is not the method of choice for solving stiff problems. That being said, and given the lack of a better way to solve stiff initial value problems in Maxima, we shall give some examples of such problems, and show how they can be solved by `rkf45`.

3.2 Examples of stiff problems.

3.2.1 One first-order differential equation.

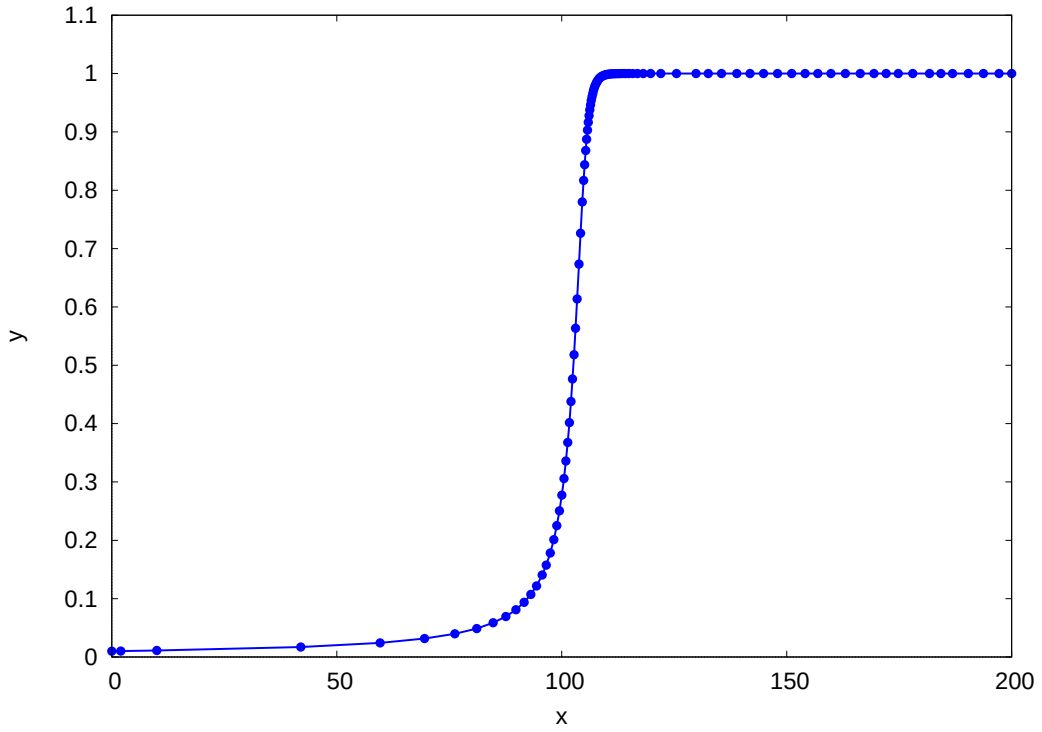
A well-known example, which can be used to demonstrate the meaning of stiffness, is the initial value problem

$$\frac{dy}{dx} = y^2 - y^3, \quad y\left(0\right) = \delta, \quad x \in \left[0, \frac{2}{\delta}\right], \tag{12}$$

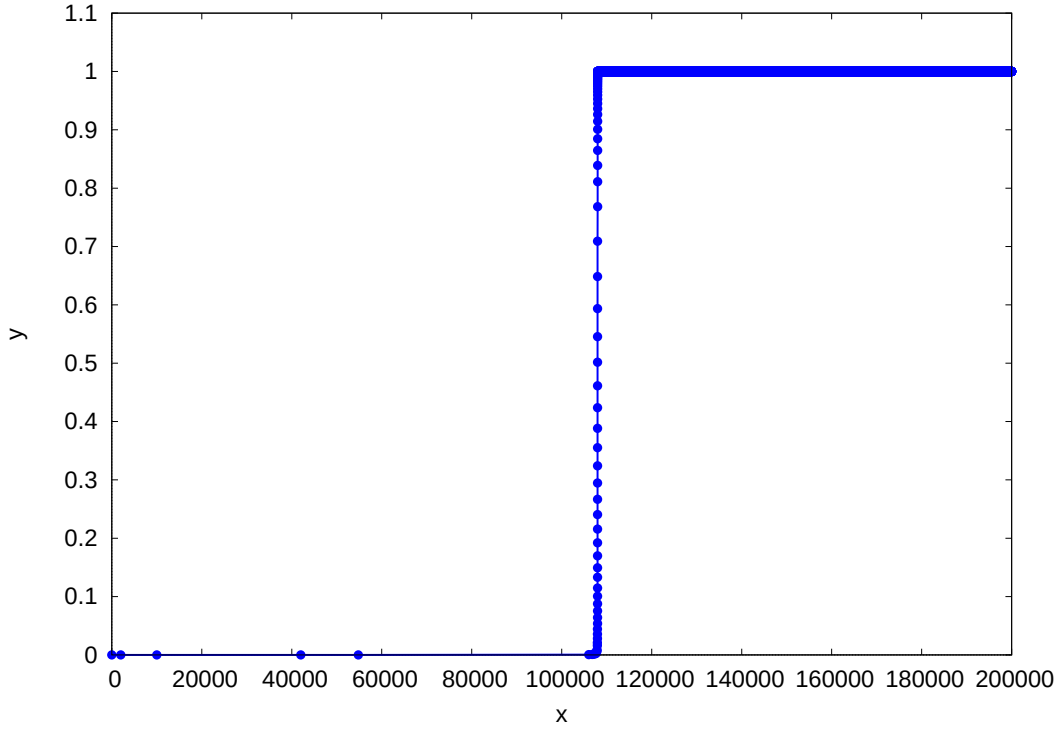
sometimes called *Shampine's ball of flame*. This problem might seem simple, but it is not. The parameter δ plays a crucial role on the stiffness of the problem. High values of δ make the problem non-stiff, and easy to solve numerically. However, as δ is decreasing the problem becomes stiffer, and eventually much more difficult to solve. Listing 7 shows a Maxima program that solves the problem for two representative values of δ: (a) $\delta = 0.01$ (mildly stiff) and (b) $\delta = 0.00001$ (very stiff.) In both cases, the absolute tolerance is set to 5×10^{-8}.

```
load("rkf45.mac")$
x_start:0$

d:0.01$ x_end:2/d$
sol_nonstiff:rkf45(y^2-y^3,y,d,[x,x_start,x_end],absolute_tolerance=5e-8,
                   report=true)$
plot2d([discrete,sol_nonstiff],[style,[linespoints,4]],[psfile,"Stiff_1a.eps"])$
yconst:part(sol_nonstiff,
            sublist_indices(sol_nonstiff,lambda([u],abs(1-u[2])<5e-6)))$
print("Solution is essentially constant for x>",yconst[1][1])$
print(length(yconst),"integration points were selected in that region.")$


d:0.00001$ x_end:2/d$
sol_stiff:rkf45(y^2-y^3,y,d,[x,x_start,x_end],absolute_tolerance=5e-8,
                max_iterations=40000,report=true)$
plot2d([discrete,sol_stiff],[style,[linespoints,4]],[psfile,"Stiff_1b.eps"])$
yconst:part(sol_stiff,sublist_indices(sol_stiff,lambda([u],abs(1-u[2])<5e-6)))$
print("Solution is essentially constant for x>",yconst[1][1])$
print(length(yconst),"integration points were selected in that region.")$
```

Listing 7: Maxima program for solving Eqs. (12).



(a) Solution for $\delta = 0.01$ (mildly stiff case.)



(b) Solution for $\delta = 0.00001$ (stiff case.)

Fig. 9: Graphical output of Listing 7.

In Fig. 9a, the solution for $\delta = 0.01$ is plotted. There is nothing really surprising in this case; the plot is qualitatively the same as in Fig. 4a. `rkf45` takes very small steps where it is needed, i.e., at the part of the curve where the slope is high; on the other hand, the step size is much larger where solution $y(x)$ varies slowly. The fact that the problem is not very stiff is also apparent in the report returned by `rkf45`:

```
-----------------------------------------------------
Info: rkf45:
   Integration points selected: 100
    Total number of iterations: 118
           Bad steps corrected: 19
      Minimum estimated error: 1.8451422293479214E-13
      Maximum estimated error: 4.8629641270472728E-8
Minimum integration step taken: 0.16629844052309
Maximum integration step taken: 32.0
-----------------------------------------------------
```

Here, minimum integration step taken is ≈ 0.17, and apparently corresponds to the interval where $y(x)$ varies quickly. On the other hand, maximum integration step taken is 32, corresponding to higher values of x, where the step size does not need to be small. To make things more quantitative, a few commands were added in Listing 7, in order to compute how many integration points were taken for the part of the curve where $|1 - y(x)| \leq 5 \times 10^{-6}$. We see that, for $\delta = 0.01$, solution is essentially constant for $x \gtrsim 117$, and `rkf45` selected 29 integration points in that region (29% of the total integration points,) which is rather more than what was expected; however, as it has been pointed out already, the algorithm is generally conservative, and avoids too big step size changes. To conclude, all the facts mentioned above are more or less expected; there is nothing really unusual in the solution for $\delta = 0.01$.

Fig. 9b shows the solution for $\delta = 0.00001$, and it is obvious that something is wrong in this case. The algorithm selects only a few integration points in the beginning, and much more points where $y(x)$ varies quickly; this is normal and expected. However, `rkf45` keeps using a very small integration step, even for higher values of x, where $y(x)$ is essentially constant, and equal to 1. There is no apparent reason for this behavior; a few steps should be more than enough in this interval. This is not a bug in `rkf45`, however; it is a direct consequence of stiffness.

Let us examine what happened more thoroughly. This is what `rkf45` reports about the computations:

```
-----------------------------------------------------
Info: rkf45:
   Integration points selected: 30247
    Total number of iterations: 35205
           Bad steps corrected: 4959
      Minimum estimated error: 1.9654044991423955E-19
      Maximum estimated error: 4.998424021413516E-8
Minimum integration step taken: 0.16632940865998
Maximum integration step taken: 51200.0
-----------------------------------------------------
```

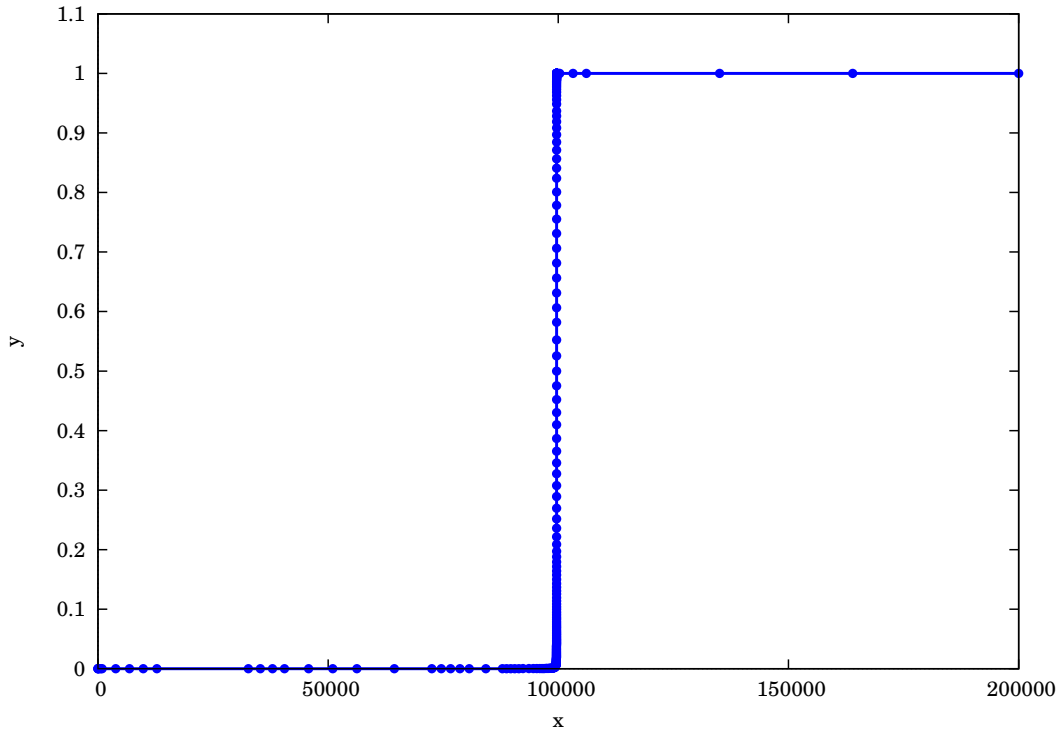


Fig. 10: Same as Fig. 9b, but using the Fortran solver DLSODAR instead of `rkf45`.

We see that 30247 integration points were selected, much more than in the $\delta = 0.01$ case. However, it is obvious from Fig. 9 that the majority of those points lie in an interval where $y(x)$ is more or less constant. Specifically, solution is essentially constant for $x \gtrsim 107985$. However, `rkf45` selected 30165 integration points in that region ($\approx 99.7\%$ of the total integration points.) This seems to be a total waste of computational effort and time, but in fact it is absolutely necessary for an explicit method like `rkf45`. Unlike the $\delta = 0.01$ case, the problem is very stiff for $\delta = 0.00001$. Stiffness can cause numerical instability if a large integration step is used, even if the function is essentially constant. In fact, this is the usual definition for the term "stiffness"; a very small step size is quite expected if an *explicit* method is used to solve a stiff problem, even though accuracy requirements could be satisfied with much larger integration steps. This is why an explicit method is definitely not the method of choice for such kind of problems. Nevertheless, `rkf45` was able to solve the problem with great accuracy, albeit excessive computational effort was needed.

For comparison, the Fortran subroutine DLSODAR takes only 224 integration steps to solve the problem for $\delta = 0.00001$, with the same settings as in `rkf45` (solution obtained by the Fortran solver is plotted in Fig. 10.) It is obvious that DLSODAR, which is a solver specifically designed for non-stiff and stiff problems as well, is much more efficient in this case. Note, however, that the problem solved here is extremely stiff for the purpose of demonstrating the meaning of stiffness, in an extreme case. This explains the huge number of integration steps taken by `rkf45`; in practice, many stiff problems do not need such a huge amount of computations to be solved, even if an explicit method like `rkf45` is used.

```
load("rkf45.mac")$
x_start:0$ x_end:4$ funcs:[y1,y2]$ equs:[998*y1+1998*y2,-999*y1-1999*y2]$
eigenvalues(jacobian(equs,funcs));

sol:rkf45(equs,funcs,[1,0],[x,x_start,x_end],report=true)$
plot2d([[discrete,map(lambda([u],part(u,[1,2])),sol)],
        [discrete,map(lambda([u],part(u,[1,3])),sol)]],
       [style,[lines,4]],[xlabel,"x"],[ylabel,"y_1, y_2"],
       [legend,"y_1(t)","y_2(t)"],[psfile,"Stiff_2a.eps"])$
plot2d([[discrete,map(lambda([u],part(u,[1,2])),sol)],
        [discrete,map(lambda([u],part(u,[1,3])),sol)]],
       [x,x_start,0.02],[style,[linespoints,4]],[point_type,bullet],
       [xlabel,"x"],[ylabel,"y_1, y_2"],[legend,"y_1(t)","y_2(t)"],
       [gnuplot_preamble,"set key center right"],[psfile,"Stiff_2b.eps"])$
y1_exact(x):=2*exp(-x)-exp(-1000*x)$
y2_exact(x):=-exp(-x)+exp(-1000*x)$
errors:part(map(lambda([u],[abs(y1_exact(u[1])-u[2]),abs(y2_exact(u[1])-u[3])]),
                sol),allbut(1))$
print("Maximum actual error:",lmax(flatten(errors)))$
```

Listing 8: Maxima program for solving Eqs. (13).

3.2.2 A linear stiff problem.

A very common example of a stiff initial value problem is

$$\left\{\begin{array}{lcl} \frac{dy_1}{dx} &=& 998y_1+1998y_2 \\ \frac{dy_2}{dx} &=& -999y_1-1999y_2 \end{array}\right. , \quad \left\{\begin{array}{lcl} y_1(0) &=& 1 \\ y_2(0) &=& 0 \end{array}\right. \tag{13}$$

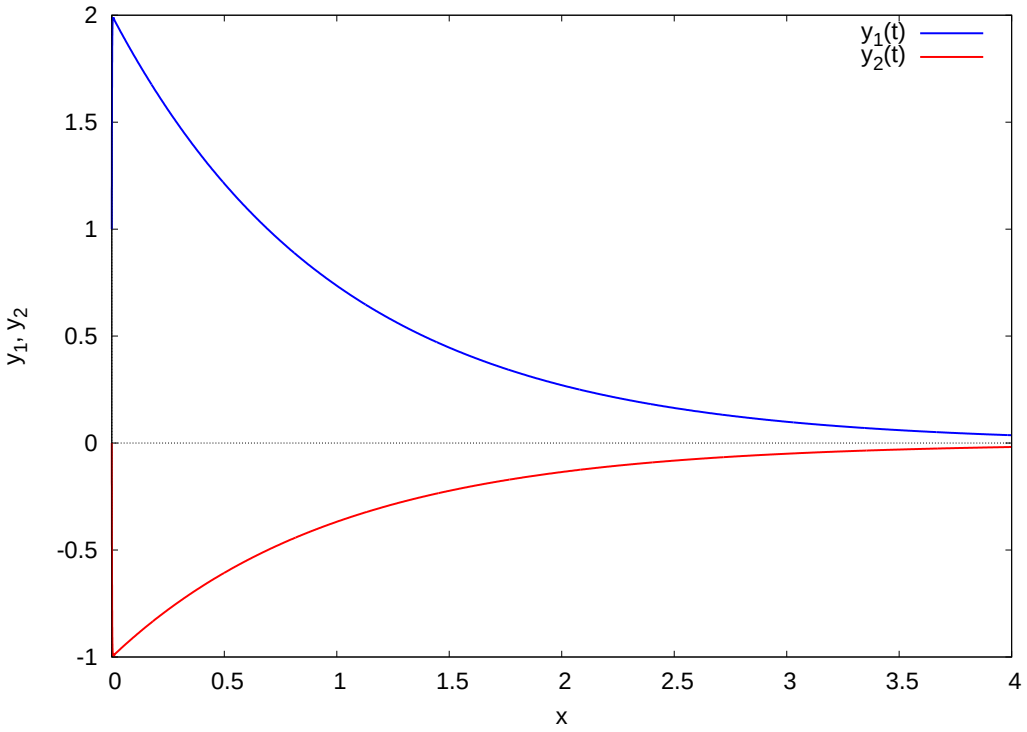
(see, e.g. Press et al. (1992, Eqs. (16.6.1-16.6.2)).) The coefficients in the right-hand side of these equations may vary in the literature, but the general idea is the same: the derivatives $\frac{dy_1}{dx}, \frac{dy_2}{dx}$ depend strongly on the functions y_1 and y_2. The usual test for stiffness is to derive the Jacobian of the right-hand side of the differential equations, and compute its eigenvalues. In this case, the eigenvalues of the Jacobian,

$$J = \frac{\partial\left(998y_1+1998y_2, -999y_1-1999y_2\right)}{\partial\left(y_1,y_2\right)} = \left[\begin{array}{cc} 998 & 1998 \\ -999 & -1999 \end{array}\right], \tag{14}$$

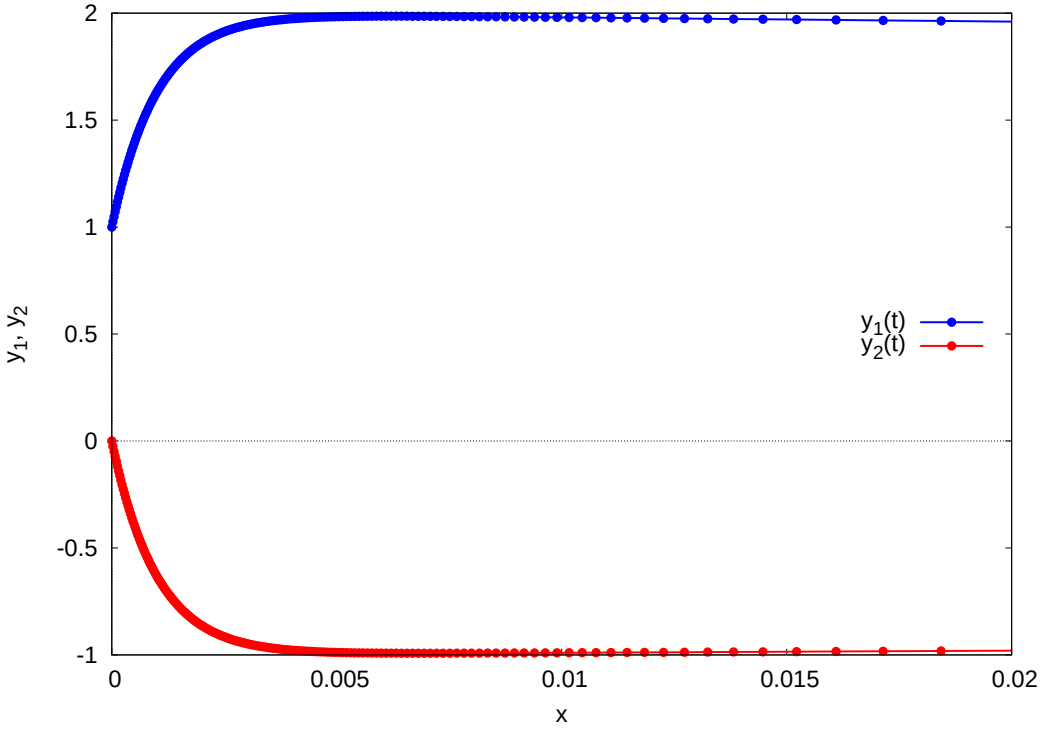
are $\lambda_1 = -1$ and $\lambda_2 = -1000$, as one can easily verify using Maxima. The fact that $|\lambda_2| \gg |\lambda_1|$ is a clear indication of stiffness. Furthermore, the stiff nature of a problem can be verified if an analytic solution is available. In our example, the partial solution that satisfies the initial conditions is

$$\left\{\begin{array}{lcl} y_1(x) &=& 2e^{-x}-e^{-1000x} \\ y_2(x) &=& -e^{-x}+e^{-1000x} \end{array}\right. , \tag{15}$$

which can be easily obtained in Maxima. Now, the reason of stiffness (and, consequently, numerical instability) is the term e^{-1000x}, which is important for very small values of x, but it is completely negligible otherwise (it becomes essentially zero very quickly.)



(a) Numerical solution for $y_1(x)$, $y_2(x)$ (solution obtained with default accuracy.)



(b) Magnification near $x = 0$.

Fig. 11: Graphical output of Listing 8.

The Maxima program that solves the stiff problem (13) is shown in Listing 8. The essential part is the call of `rkf45`, which does not differ from that of any other program solving a system of differential equations (the problem is solved using default absolute tolerance, 10^{-6}.) Let us have a look at the report now:

```
-------------------------------------------------------
Info: rkf45:
   Integration points selected: 1473
    Total number of iterations: 1695
           Bad steps corrected: 223
      Minimum estimated error: 6.4847649710556282E-8
      Maximum estimated error: 9.9734487940981384E-7
Minimum integration step taken: 2.4897150300248846E-5
Maximum integration step taken: 0.0043726569115022
-------------------------------------------------------
```

What we see here is that 1473 integration points were selected. Bad (but refined and corrected) steps occured 223 times; such a large number of bad steps is rather common in stiff problems, but it is not an indication of stiffness by itself.

There is one interesting fact that needs to be discussed here. The algorithm estimated that the maximum error is $\approx 9.98 \times 10^{-7}$, marginally lower than requested accuracy, as usual. In this particular example, the exact analytic solution is available, so we can check the accuracy of the numerical solution obtained. The actual maximum error is $\approx 5.47 \times 10^{-9}$. In other words, actual error is more than two orders of magnitude lower than estimated error. This is unusual, because, as already pointed out, `rkf45` returns a solution more accurate than requested, but the actual error is typically close to the error estimated by the algorithm. However, it is easy to understand what happened. The problem is stiff, so integration step needed to be kept small, even though a larger step would satisfy accuracy requirements. This means that, in stiff problems, the results are expected to be considerably more accurate than requested, exactly because a very small integration step will be used, due to numerical stability requirements. This is actually the case here, and it is a clear indication of stiffness.

The graphical output of the program is shown in Fig. 11. The "vertical" segments of the curves near $x = 0$ are not artifacts of the plotting utility. It is the exact behavior of the solution; the values of $y_1(x)$ and $y_2(x)$ change extremely rapidly at very small values of x, due to the factor e^{-1000x} involved in Eqs. (15). However, the algorithm was able to detect this behavior and act accordingly, by reducing the step as required. In fact, `rkf45` has selected 167 integration points (more than 11% of the total points) in a tiny interval near the origin, $x \in [0, 0.02]$ (0.5% of the total integration interval.) Solution for $x \in [0, 0.02]$ is plotted in Fig. 11b. One can see how fast both $y_1(x)$ and $y_2(x)$ are changing in that tiny interval, forcing `rkf45` to reduce the step, and start to increase it again after $x \approx 0.01$. The situation is similar to that in § 2.2.2 and § 2.2.3, but in the present case the need for a small step size is much more apparent. It is worth emphasizing, however, that, although the integration step is increasing after $x \approx 0.01$, it is still kept relatively small; 1306 integration points were needed for $x \in [0.02, 4]$, which is too much, considering the smooth behavior of the solution in that interval. Again, this is due to the stiffness of the problem; integration step size is kept small because of stability (not accuracy) requirements. A different problem, with a solution similar to that in Fig. 11a (but without the steep part of the curve near $x = 0$) would need much less integration points to be solved, with the same accuracy requirements.

3.2.3 A stiff van der Pol oscillator.

In § 2.2.4 an initial value problem involving the var der Pol differential equation was solved. In this section we shall solve the problem for a higher value of the dumping parameter, μ, which makes the problem stiff. We shall use a different form of the differential equation, however (see, e.g., Hairer & Wanner (2002), Eq.. (1.5'),)

$$\frac{dx^2}{dt^2} = \frac{1}{\epsilon}\left(\left(1-x^2\right)\frac{dx}{dt} - x\right) = 0, \quad \left\{ \begin{array}{rcl} x\left(0\right) & = & 2 \\ \frac{dx}{dt}\big|_{x=0} & = & 0 \end{array} \right. , \tag{16}$$

where ϵ is a positive parameter, corresponding to $\frac{1}{\mu}$ in Eqs. (4). Eqs. (16) are more convenient for stiff problems, because the integration interval does not need to be changed, depending on the parameter, as it was the case in Eqs. (4); instead, integration interval is fixed to $t \in [0, 2]$. Low values of ϵ correspond to stiff problems; in this example, we shall use the value $\epsilon = 0.01$.

In order to solve the problem, the second-order differential equation must be transformed to a system of two first-order differential equations, as in § 2.2.4. By putting $x = x_1$, $\frac{dx}{dt} = x_2$, Eqs. (16) are written as

$$\left\{ \begin{array}{rcl} \frac{dx_1}{dt} & = & x_2 \\ \frac{dx_2}{dt} & = & \frac{1}{\epsilon}\left(\left(1-x_1^2\right)x_2 - x_1\right) \end{array} \right. , \quad \left\{ \begin{array}{rcl} x_1\left(0\right) & = & 2 \\ x_2\left(0\right) & = & 0 \end{array} \right. . \tag{17}$$

Listing 9 shows the Maxima program that solves this problem. Graphical output of the program is shown in Fig. 12; Note the peaks of $\frac{dx}{dt}$ in Fig. 12a, which are actually the reason of stiffness in this problem. Solution is initially computed with default accuracy tolerance, 10^{-6}. Report given by `rkf45` in that case is

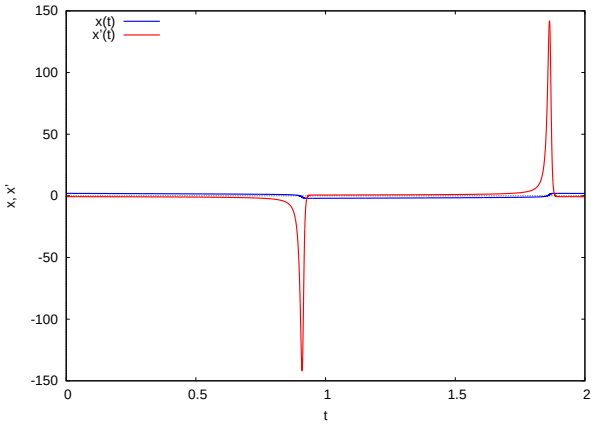
```
-------------------------------------------------------
Info: rkf45:
   Integration points selected: 2148
    Total number of iterations: 2165
           Bad steps corrected: 18
       Minimum estimated error: 3.7395389218163665E-8
       Maximum estimated error: 8.8569168942399216E-7
Minimum integration step taken: 6.08069431864205E-5
Maximum integration step taken: 0.0027205802470264
-------------------------------------------------------
```

In this case, 2148 integration points were needed. Minimum integration step is $\approx 6.1 \times 10^{-5}$ – very small, compared to the integration interval, $[0, 2]$. Such a small step is actually needed in the steep parts of the curve $\frac{dx}{dt}$. In particular, 664 integration points ($\approx 31\%$ of the total points) were used in the interval $t \in [0.8, 1]$, where the first peak of $\frac{dx}{dt}$ lies, and a similar amount of integration points is used for the second peak.

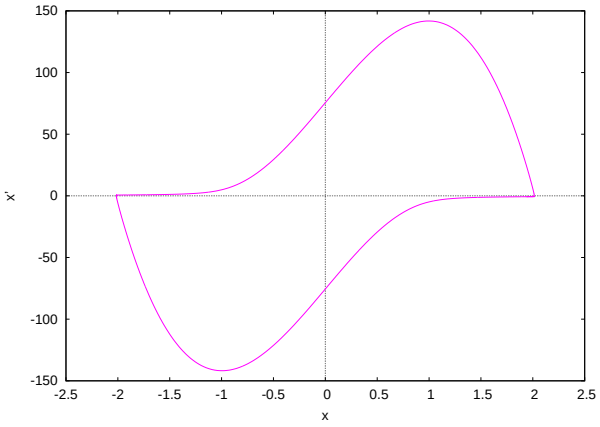
Apparently, a fixed-step method would fail to compute the solution accurately, unless a very small integration step would be used *globally*, which would result in higher computational times. In this particular example, a fixed-step Runge-Kutta method of fourth order would need at least 32891 integration points to achieve an accuracy equal to that obtained by `rkf45` using 2148 integration points. In terms of function evaluations (which is often used as a measure of the computation time,) a Runge-Kutta method of fourth order would thus need 131560 function evaluations, while `rkf45`

```
load("rkf45.mac")$
t_start:0$ t_end:2$ epsilon:1e-2$
funcs:[x1,x2]$ equs:[x2,((1-x1^2)*x2-x1)/epsilon]$

sol:rkf45(equs,funcs,[2,0],[t,t_start,t_end],report=true)$
plot2d([[discrete,map(lambda([u],part(u,[1,2])),sol)],
        [discrete,map(lambda([u],part(u,[1,3])),sol)]],
       [style,[lines,4]],[xlabel,"t"],[ylabel,"x(t), x'(t)"],
       [legend,"x(t)","x'(t)"],[gnuplot_preamble,"set key top left"],
       [psfile,"Stiff_3a.eps"])$
plot2d([discrete,map(lambda([u],part(u,allbut(1))),sol)],[style,lines[4]],
       [color,magenta],[xlabel,"x"],[ylabel,"x'"],[psfile,"Stiff_3b.eps"])$

sol:rkf45(equs,funcs,[2,0],[t,t_start,t_end],absolute_tolerance=5e-3,
          report=true)$
plot2d([discrete,map(lambda([u],part(u,[1,2])),sol)],[x,0.8,1],
       [style,[linespoints,4]],[xlabel,"t"],[ylabel,"x(t)"],
       [legend,"adaptive step size (rkf45 results)"],[psfile,"Stiff_3c.eps"])$
plot2d([discrete,map(lambda([u],part(u,[1,3])),sol)],[x,0.8,1],
       [style,[linespoints,4]],[xlabel,"t"],[ylabel,"x'(t)"],[color,red],
       [legend,"adaptive step size (rkf45 results)"],[psfile,"Stiff_3d.eps"])$

sol_rk:rk(equs,funcs,[2,0],[t,t_start,t_end,(t_end-t_start)/(length(sol)-1)])$
plot2d([discrete,map(lambda([u],part(u,[1,2])),sol_rk)],[x,0.8,1],
       [style,[linespoints,4]],[xlabel,"t"],[ylabel,"x(t)"],
       [legend,"fixed step (rk results)"],[psfile,"Stiff_3e.eps"])$
plot2d([discrete,map(lambda([u],part(u,[1,3])),sol_rk)],[x,0.8,1],
       [style,[linespoints,4]],[xlabel,"t"],[ylabel,"x'(t)"],
       [legend,"fixed step (rk results)"],[color,red],[psfile,"Stiff_3f.eps"])$
```

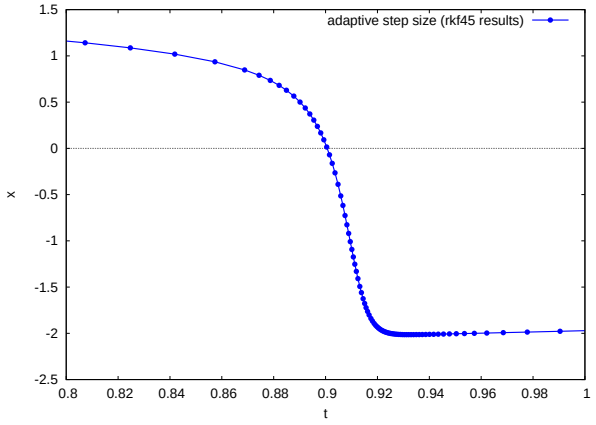
Listing 9: Maxima program for solving Eqs. (17).



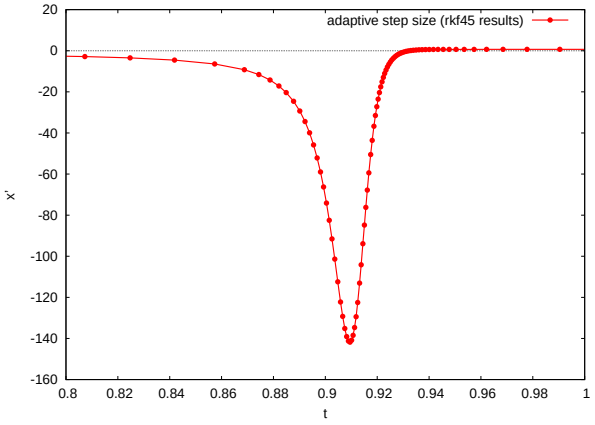
(a) Solution obtained by `rkf45` (default accuracy.)



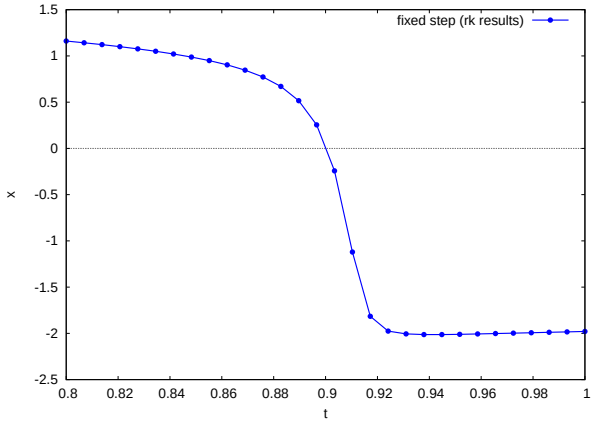
(b) Phase portrait for solution shown in (a).



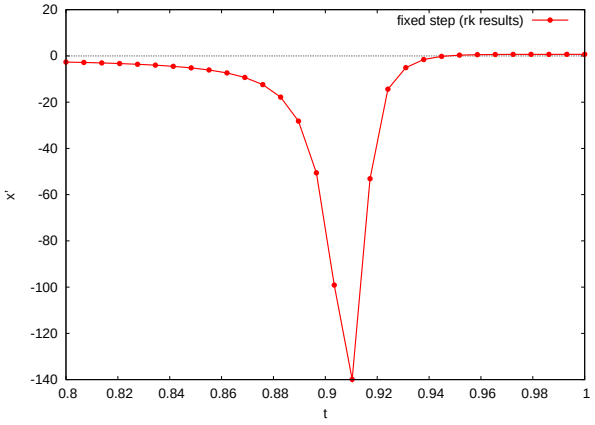
(c) Function $x\,(t)$, computed by `rkf45`.



(d) Function $\frac{dx}{dt}$, computed by `rkf45`.



(e) Function $x\,(t)$, computed by `rk`.



(f) Function $\frac{dx}{dt}$, computed by `rk`.

Fig. 12: Graphical output of Listing 9. Figs. (a) and (b) correspond to solution obtained using `rkf45` with default absolute error tolerance, 10^{-6}; Figs. (c) and (d) correspond to solution obtained with absolute error tolerance set to 5×10^{-3}. For comparison, Figs. (e) and (f) correspond to solution obtained by `rk`, using the same number of integration points, as in (c) and (d).

needs 12990 function evaluations (taking into account that 18 steps were rejected.) In other words, `rkf45` is about ten times faster in this example.

To visualize the difference between `rkf45` and a fixed-step method, we solve the problem again, this time with absolute tolerance set to 5×10^{-3}. This would reduce the integration points needed, so that it will be easier to see the difference graphically. Indeed, report given by `rkf45` is now

```
-----------------------------------------------------
Info: rkf45:
   Integration points selected: 291
    Total number of iterations: 335
           Bad steps corrected: 45
      Minimum estimated error: 2.953992239004648E-5
      Maximum estimated error: 0.0049405216100875
Minimum integration step taken: 5.0200898447719971E-4
Maximum integration step taken: 0.019136888549425
-----------------------------------------------------
```

That is, only 291 integration points were needed, while solution returned is still accurate to at least two decimal digits. Results obtained that way are shown in Figs. 12c-12d, where solution is plotted around the first peak of $\frac{dx}{dt}$. We also solve the problem using Maxima's function `rk` (a fixed-step method); we set the fixed integration step in `rk` so that a total of 291 integration points are used, as in `rkf45` above. Plots of the results obtained that way are shown in Figs. 12e-12f. It is obvious that a fixed-step method fails to give an accurate solution.

3.2.4 The Brusselator.

The *Brusselator* (1D diffusion) is an initial value problem, consisting of two first-order differential equations, together with two initial conditions. It is included in several collections of stiff initial value problems (see, e.g., Hairer & Wanner (2002); Nowak et al. (2010).) Strictly speaking, the problem is not very stiff, but it is certainly very interesting as a test for initial value problem solvers. The typical form of the Brusselator equations found in the literature is

$$\left\{ \begin{array}{lcl} \frac{dy_1}{dx} & = & A + y_1^2 y_2 - (B+1)\, y_1 \\ \frac{dy_2}{dx} & = & B y_1 - y_1^2 y_2 \end{array} \right. , \quad \left\{ \begin{array}{lcl} y_1(0) & = & 1 \\ y_2(0) & = & 4.2665 \end{array} \right. , \tag{18}$$

where $A = 2$, $B = 8.533$, and the problem is solved for $x \in [0, 20]$.

A Maxima program for solving Eqs. (18) is given in Listing 10. Solution obtained with default absolute tolerance is shown in Fig. 13, where it is apparent that both $y_1(x)$ and $y_2(x)$ exhibit abrupt changes and very high slopes periodically. As in all previous cases, `rkf45` detected this behavior and reduced integration step size accordingly. Report given by `rkf45` is

```
-----------------------------------------------------
Info: rkf45:
   Integration points selected: 1115
    Total number of iterations: 1150
           Bad steps corrected: 36
      Minimum estimated error: 5.3027036774583149E-8
```

```
load("rkf45.mac")$
x_start:0$ x_end:20$ y_start:[1,4.2665]$ A:2$ B:8.533$
equs:[A+y1^2*y2-(B+1)*y1,B*y1-y1^2*y2]$

sol:rkf45(equs,[y1,y2],y_start,[x,x_start,x_end],report=true)$
plot2d([[discrete,map(lambda([u],part(u,[1,2])),sol)],
        [discrete,map(lambda([u],part(u,[1,3])),sol)]],[style,[lines,4]],
        [ylabel,"y_1, y_2"],[legend,"y_1(x)","y_2(x)"],[psfile,"Stiff_4a.eps"])$

plot2d([[discrete,map(lambda([u],part(u,[1,2])),sol)],
        [discrete,map(lambda([u],part(u,[1,3])),sol)]],[x,5.15,5.4],
        [style,[linespoints,4]],[point_type,bullet],
        [ylabel,"y_1, y_2 (rkf45 results)"],
        [legend,"y_1(x)","y_2(x)"],[psfile,"Stiff_4b.eps"])$

sol_rk:rk(equs,[y1,y2],y_start,
          [x,x_start,x_end,(x_end-x_start)/(length(sol)-1)])$
plot2d([[discrete,map(lambda([u],part(u,[1,2])),sol_rk)],
        [discrete,map(lambda([u],part(u,[1,3])),sol_rk)]],[x,5.15,5.4],
        [style,[linespoints,4]],[point_type,bullet],
        [ylabel,"y_1, y_2 (rk results)"],
        [legend,"y_1(x)","y_2(x)"],[psfile,"Stiff_4c.eps"])$
```

Listing 10: Maxima program for solving Eqs. (18).

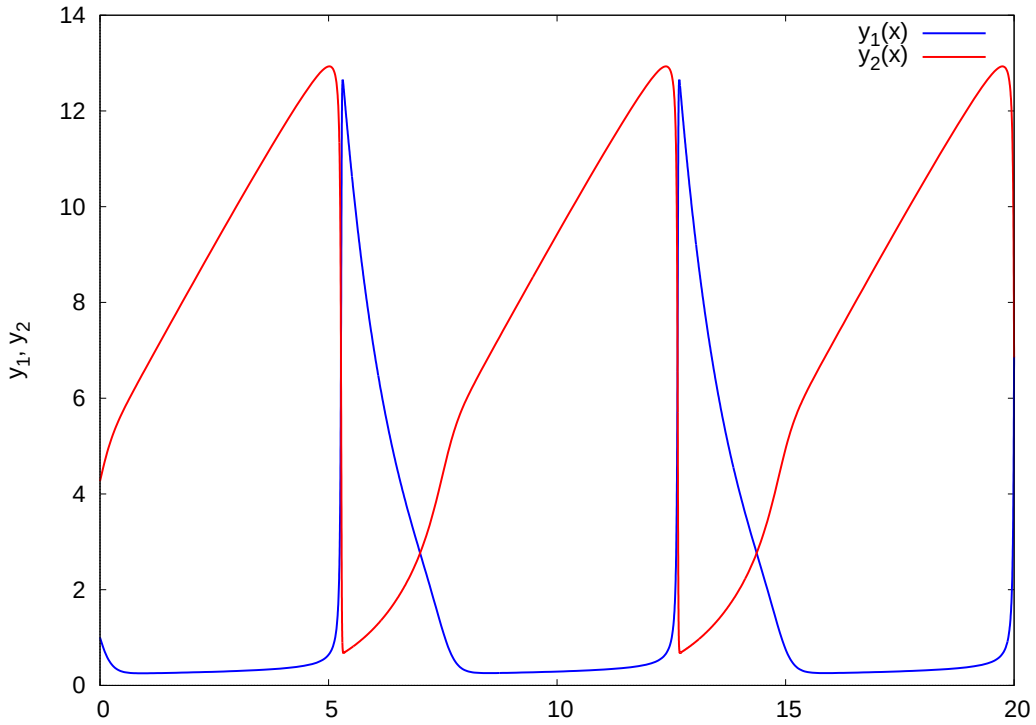


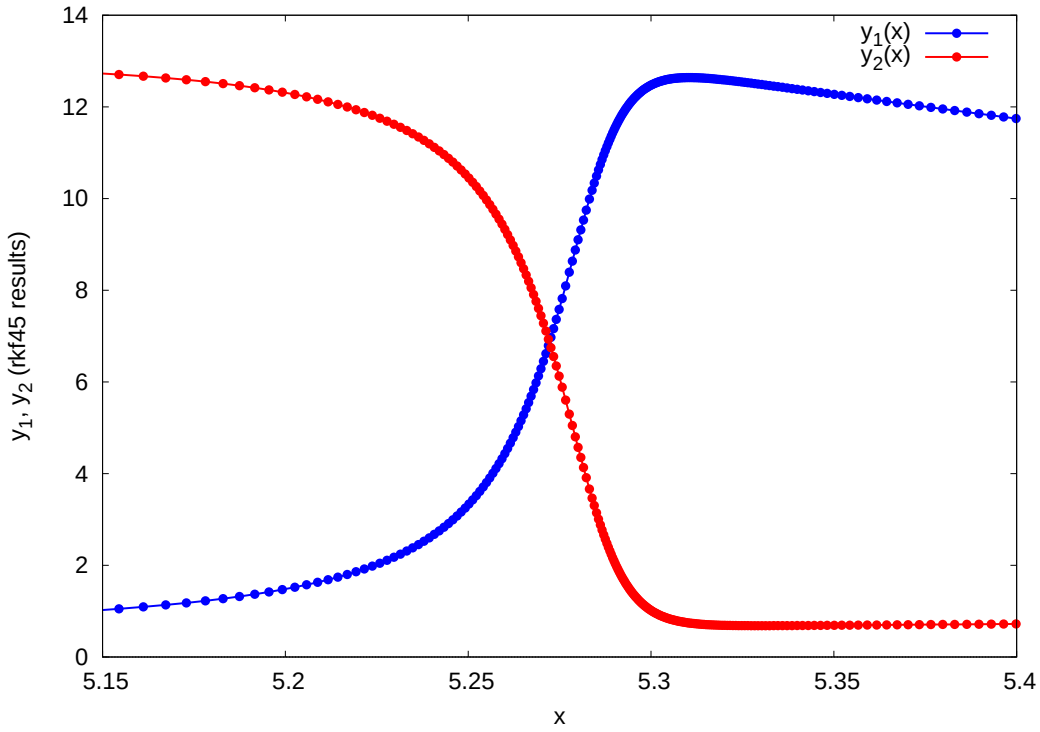
Fig. 13: Graphical output of Listing 10 (part I.)

```
      Maximum estimated error: 9.8233659815093057E-7
Minimum integration step taken: 3.3850814947041446E-4
Maximum integration step taken: 0.21818753157269
-------------------------------------------------------
```

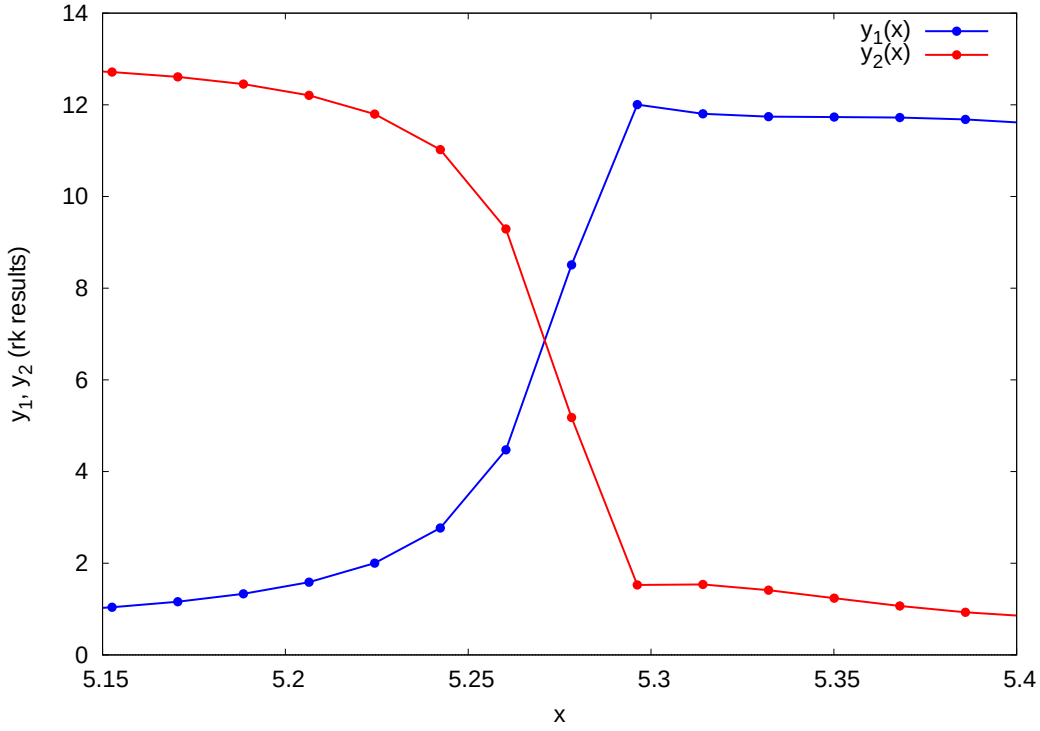
We see that 1115 integration points were selected by `rkf45`. This large number of integration points can be explained by the very high slopes of $y_1(x)$ and $y_2(x)$. Because of this, minimum step size is also very small, $\approx 3.4 \times 10^{-4}$.

It is easy to do estimations about the computational time, as in § 3.2.3. In this example, a fixed-step Runge-Kutta method of fourth order would need at least 59084 integration points to achieve the same accuracy as in `rkf45` above. This means that 236332 function evaluations would be needed, while `rkf45` needs 6900 function evaluations (we take into account that 36 steps were rejected.) Consequently, `rkf45` is about 34 times faster in this example.

For comparison, we solve the problem again, this time using a fixed-step method (`rk`); we set the fixed step size so that `rk` uses 1115 integration points, as in `rkf45` above. Results obtained by `rkf45` and `rk` are plotted in Fig. 14; we focus on the interval $x \in [5.15, 5.4]$, where both $y_1(x)$ and $y_2(x)$ are very steep. Although both methods use exactly the same number of integration points, `rk` fails to compute an accurate solution due to its fixed step size; integration points are just distributed uniformly along the integration interval, and thus only a 14 integration points lie in the small interval $[5.15, 5.4]$ (this is only 1.25% of the total integration interval.) On the other hand, `rkf45` reduced the step size in that interval, so that 207 integration points ($\approx 18.6\%$ of the total integration points) were selected for $x \in [5.15, 5.4]$. That way, `rkf45` managed to get an accurate solution.



(a) Solution obtained by `rkf45` for $x \in [5.15, 5.4]$.



(b) Solution obtained by `rk`, using the same number of integration points as in `rkf45`.

Fig. 14: Graphical output of Listing 10 (part II.)

References

Jean-Pierre Nougier, *Methodes de calcul numerique* Vol. 2, Hermes Science Publications, Paris (2001).

Brian Bradie, *A Friendly Introduction to Numerical Analysis*, Pearson Prentice Hall, New Jersey (2006).

Richard L. Burden and J. Douglas Faires, *Numerical Analysis*, Thomson Higher Education, Belmont (2005).

William H. Press, Saul A. Teukolsky, William T. Vellerling and Brain P. Flannery, *Numerical Recipes - The Art of Scientific Computing* (Second Edition), Cambridge University Press, Cambridge (1992).

E. Hairer, S. P. Nørset and G. Wanner, *Solving Ordinary Differential Equations I: Nonstiff Problems*, 2nd edition, Springer Verlag, Berlin (1993).

E. Hairer and G. Wanner, *Solving Ordinary Differential Equations II: Stiff and Differential-Algebraic Problems*, 2nd edition, Springer-Verlag, Berlin (2002). Test problems for stiff ordinary differential equations described in that book are available at `http://www.unige.ch/~hairer/testset/testset.html`.

F. Mazzia and C. Magherini. *Test Set for Initial Value Problem Solvers*, Department of Mathematics, University of Bari and INdAM, Research Unit of Bari (2008). Available at `http://www.dm.uniba.it/~testset`.

U. Nowak, S. Gebauer, U. Pöhle and L. Weimann, *ODELab - Towards an Interactive WWW Laboratory for Numerical ODE Software*, Konrad-Zuse-Zentrum für Informationstechnik Berlin (2010). Available at `http://num-lab.zib.de/public/odelab`.

A. C. Hindmarsh, *ODEPACK, A Systematized Collection of ODE Solvers — Scientific Computing*, eds. R. S. Stepleman et al., North-Holland, Amsterdam, p. 55 (1983). Available at `http://www.netlib.org/odepack`.

L. R. Petzold, *Siam J. Sci. Stat. Comput.* **4**, 136 (1983).

P. J. Papasotiriou, V. S. Geroyannis and S. A. Sanidas, *Int. J. Mod. Phys. C* **18**(11), 1735 (2007).