Computational Physics with Maxima or R: Ch. 2, Initial Value Problems *

Edwin (Ted) Woollett

August 29, 2015

Contents

1 Introduction **3**

2 The Euler Method Using Maxima, Truncation Error, Round-off Error and Instability **4**

3 The Euler Method Using R **16**

4 Fourth Order Runge-Kutta Code for Maxima and R **21**
- 4.1 Maxima Code: rk4 . . . 22
  - 4.1.1 Five Examples of Maxima rk4 . . . 23
  - 4.1.2 Failure of Maxima rk4 for a Stiff O.D.E. . . . 28
- 4.2 R Code: myrk4 . . . 29
  - 4.2.1 Three Examples of R myrk4 . . . 30
  - 4.2.2 Failure of R myrk4 for a Stiff O.D.E. . . . 33

5 The Standard and Contributed Maxima Methods **34**
- 5.1 rk . . . 34
- 5.2 rkf45 . . . 34

6 Standard R Methods using deSolve's ode **39**
- 6.1 One First Order O.D.E.: Solving $dy/dx = -x\,y$ with $y(0) = 1$ . . . 40
- 6.2 Two First Order O.D.E.'s: Using the Parameters Argument . . . 44
- 6.3 Three First Order O.D.E.'s: The Lorenz Model . . . 46
- 6.4 Solving the Stiff Case of the van der Pol Equation . . . 50

7 Using External Forcing Data for O.D.E.'s **51**
- 7.1 Forcing Data Using Maxima . . . 51
- 7.2 Forcing Data Using R . . . 56

8 Integrating O.D.E.'s with Discontinuous Derivatives **61**
- 8.1 Example 1: Oral Drug Dose Model . . . 61
- 8.2 Example 2 . . . 63

9 Integrating O.D.E.'s with Discontinuous Dependent Variables Using R **65**
- 9.1 Events Specified by a data.frame in R . . . 65
- 9.2 Intravenous Drug Injection Model Using R . . . 66
- 9.3 Using an Event Function at Specific Times . . . 66
- 9.4 Example 1: Using an Event Function when a Root Condition is Satisfied . . . 67
- 9.5 Example 2: Event Function when a Root Condition is Satisfied . . . 68
- 9.6 Use of a Switching Parameter as a State Variable . . . 70

*The code examples use **R ver. 3.0.2** and **Maxima ver. 5.31** using **Windows 7**. This is a live document which will be updated when needed. Check `http://www.csulb.edu/~woollett/` for the latest version of these notes. Send comments and suggestions for improvements to `woollett@charter.net`

```
COPYING AND DISTRIBUTION POLICY
This document is the second chapter of a series of notes titled
Computational Physics with Maxima or R, and is made available
via the author's webpage http://www.csulb.edu/~woollett/
to encourage the use of the R and Maxima languages for computational
physics projects of modest size.

NON-PROFIT PRINTING AND DISTRIBUTION IS PERMITTED.

You may make copies of this document and distribute them
to others as long as you charge no more than the costs of printing.

Code files which accompany this chapter are
1. myode.mac
2. k2util.mac
3. myrkf45.mac
4. myode.R
```


Feedback from readers is the best way for this series of notes to become more helpful to users of **R** and **Maxima**. *All* comments and suggestions for improvements will be appreciated and carefully considered.

1 Introduction

We have discussed the *symbolic* solution of ordinary differential equations (o.d.e.'s) in Chapter 3 of **Maxima by Example** (on the author's web page). We concentrate here on the *numerical* solution of initial value problems governed by a set of o.d.e.'s.

We begin with the ancient and simple Euler method, and present some home-made code using both Maxima and **R**. We then turn to the classic fourth order fixed step Runge-Kutta method, again presenting home-made functions in both Maxima and **R**.

We then review the standard methods available in Maxima and **R**. A major attraction of the **R** platform for computational physics is the large and powerful suite of o.d.e. solvers available in the **R** package **deSolve**, and we spend most of this section introducing some aspects of the **ode** wrapper.

An important reference for users of the **deSolve** package of solvers for the **R** platform is the recent text **Solving Differential Equations in R**, by Karline Soetaert, Jeff Cash, and Francesca Mazzia, Springer-Verlag, 2012. We will often refer to this reference with the author initials **SCM**. Section 3.5, **Method Selection**, should be consulted if the default solver **ode** (called without specifying a method) is returning suspicious results.

The default method used by **ode** is called **lsoda**, based on the **FORTRAN** code **LSODA**, which is able to detect when and where an ordinary differential equation (or system of equations) becomes "stiff", and automatically implement methods which can deal with this behavior, as needed. This is a very robust method, but not necessarily the most efficient solver for one particular problem.

The default **lsoda** method used by the wrapper **ode** always starts with the non-stiff *explicit* multi-step Adams method, and when stiffness is detected, switches to an *implicit* multistep solver ("bdf": backward differentiation formula).

Maxima, at present, does not offer a stiff ode solver.

Many advanced numerical algorithms that solve differential equations are available as (open- source) computer codes, written in programming languages like **FORTRAN** or **C**, and available in repositories like **GAMS** (**http://gams.nist.gov**) or **NETLIB** (**www.netlib.org**).

An example of what can be done to make this code available for work in modern interactive numerical environments is the work of Karline Soetaert and Linda R. Petzold (and others) for the open source **R** numerical platform.

Present ode solvers in the **R** package **deSolve** use adaptive step size control, some solvers control the order of the formula adaptively, or switch between different types of methods, depending on the local properties of the ode's to be solved.

The **R** package **deSolve** includes methods to solve stiff and non-stiff problems, that deal with full, banded, or arbitrarily sparse Jacobians, etc. The implementation includes stiff and nonstiff integration routines based on the **ODEPACK FORTRAN** codes (Hindmarsh 1983). It also includes fixed and adaptive timestep explicit Runge-Kutta solvers, the Euler method, and the implicit Runge-Kutta method **RADAU** (Hairer and Wanner 2010).

In the final sections, we discuss the solution of o.d.e.'s when the first derivative of the dependent variable is discontinuous, and also when the dependent variable itself is discontinuous.

Reduction to First Order O.D.E.'s

A second order ordinary differential equation which has the form

$$z'' = f(t, z, z') \tag{1.1}$$

in which the prime **'** indicates a derivative with respect to the independent variable t, and two single primes represents the second derivative with respect to t, can be converted into a pair of first order o.d.e.'s governing the time evolution of the pair of dependent variables $[y_1(t), y_2(t)]$.

Let $y1 = z$. Then $y_1' = z' = y_2$. And then $z'' = y_2' = f(t, y_1, y_2)$. We then deal with the pair of first oders o.d.e.'s

$$\frac{d\,y_1}{d\,t} = y_2 \tag{1.2}$$

$$\frac{d\,y_2}{d\,t} = f(t, y_1, y_2) \tag{1.3}$$

A third order ordinary differential equation with the form $z''' = f(t, z, z', z'')$ can be likewise reduced to a triplet of first order o.d.e.'s for $[y_1(t), y_2(t), y_3(t)]$.

2 The Euler Method Using Maxima, Truncation Error, Round-off Error and Instability

We quote (with some light editing) an introduction to the Euler method and a discussion of truncation and round-off errors from two computational physics web pages of Richard Fitzpatrick (Physics Dept., University of Texas), at
http://farside.ph.utexas.edu/teaching/329/lectures/node32.html and at
http://farside.ph.utexas.edu/teaching/329/lectures/node33.html.

Euler's method

Consider the general first-order o.d.e., $y' = f(x, y)$, where $'$ denotes d/dx, subject to the general initial-value boundary condition $y(x_0) = y_0$.

Clearly, if we can find a method for numerically solving this problem, then we should have little difficulty generalizing it to deal with a system of n simultaneous first-order o.d.e.'s.

It is important to appreciate that the numerical solution to a differential equation is only an approximation to the actual solution. The actual solution, $y(x)$ is (presumably) a continuous function of a continuous variable, x. However, when we solve this equation numerically, the best that we can do is to evaluate approximations to the function $y(x)$ at a series of discrete grid-points, the x_n (say), where $n = 0, 1, 2, \cdots$ and $x_0 < x_1 < x_2 \cdots$. For the moment, we shall restrict our discussion to equally spaced grid-points, where $x_n = x_0 + n\,h$.

Here, the quantity h is referred to as the step-length. Let y_n be our approximation to $y(x)$ at the grid-point x_n. A numerical integration scheme is essentially a method which somehow employs the information contained in the original o.d.e. to construct a series of rules interrelating the various y_n.

The simplest possible integration scheme was invented by the celebrated 18th century Swiss mathematician Leonhard Euler, and is, therefore, called Euler's method. Incidentally, it is interesting to note that virtually all of the standard methods used in numerical analysis were invented before the advent of electronic computers. In olden days, people actually performed numerical calculations by hand - and a very long and tedious process it must have been! Suppose that we have evaluated an approximation, y_n, to the solution, $y(x)$ at the grid-point x_n. The approximate gradient of $y(x)$ at this point is, therefore, given by $y'_n = f(x_n, y_n)$.

Let us approximate the curve $y(x)$ as a straight-line between the neighbouring grid-points x_n and x_{n+1}. It follows that $y_{n+1} = y_n + y'_n\,h$, or

$$y_{n+1} = y_n + f(x_n, y_n)\,h. \tag{2.1}$$

The above formula is the essence of Euler's method. It enables us to calculate all of the y_n, given the initial value, y_0, at the first grid-point, x_0.

Numerical errors

There are two major sources of error associated with a numerical integration scheme for o.d.e.'s: namely, **truncation error** and **round-off** error.

Truncation error arises in Euler's method because the curve $y(x)$ is *not* generally a straight-line between the neighbouring grid-points x_n and x_{n+1}, as assumed above. The error associated with this approximation can easily be assessed by Taylor expanding $y(x)$ about $x = x_n$:

$$y(x_n + h) = y(x_n) + h\, y'(x_n) + \frac{h^2}{2}\, y''(x_n) + \cdots = y(x_n) + h\, f(x_n, y_n) + \frac{h^2}{2}\, y''(x_n) + \cdots \qquad (2.2)$$

In other words, every time we take a step using Euler's method (if h is sufficiently small), because we omit small terms of order $O(h^2)$ or smaller (ie., we *truncate* the expansion), we incur a (local) truncation error of $O(h^2)$, where h is the step-length.

Suppose that we use Euler's method to integrate our o.d.e. over an x-interval of order unity. This requires $O(h^{-1})$ steps. If each step incurs an error of $O(h^2)$, and the errors are simply cumulative (a fairly conservative assumption), then the net truncation error is $O(h)$. In other words, the error associated with integrating an o.d.e. over a finite interval using Euler's method is directly proportional to the step-length.

If we let $y_e(x)$ be the approximate numerical Euler solution (starting at $x = 0$), and let $y_a(x)$ be the exact analytic solution, then the "absolute error" $|y_e(1) - y_a(1)| = O(h)$.

Thus, if we want to keep the absolute error $|y_e(x) - y_a(x)|$ in the integration below about 10^{-6} then we would need to take about one million steps per unit interval in x.

Incidentally, Euler's method is termed a first-order integration method because the truncation error associated with integrating over a finite interval scales like h^1. More generally, an integration method is conventionally called nth order if its *local* truncation error per step is $O(h^{n+1})$.

Note that truncation error would be incurred even if computers performed floating-point arithmetic operations to infinite accuracy. Unfortunately, computers do not perform such operations to infinite accuracy. In fact, a computer is only capable of storing a floating-point number to a fixed number of decimal places.

For every type of computer, there is a characteristic number, η, which is defined as the smallest number which when added to a number of order unity gives rise to a new number: i.e., a number which when taken away from the original number yields a non-zero result. Every floating-point operation incurs a **round-off error** of $O(\eta)$ which arises from the finite accuracy to which floating-point numbers are stored by the computer.

Suppose that we use Euler's method to integrate our o.d.e. over an x-interval of order unity. This entails $O(h^{-1})$ integration steps, and, therefore, $O(h^{-1})$ floating-point operations. If each floating-point operation incurs an error of $O(\eta)$, and the errors are simply cumulative, then the net round-off error is $O(\eta/h)$.

The total error, ϵ, associated with integrating our o.d.e. over an x-interval of order unity is (approximately) the sum of the truncation and round-off errors. Thus, for Euler's method, $\epsilon \sim \frac{\eta}{h} + h$.

Clearly, at large step-lengths the error is dominated by truncation error, whereas round-off error dominates at small step-lengths. The net error (for the Euler method) attains its minimum value, $\epsilon_0 \sim \eta^{1/2}$, when

$h = h_0 \sim \eta^{1/2}$. There is clearly no point in making the step-length, h, any smaller than h_0, since this increases the number of floating-point operations but does not lead to an increase in the overall accuracy.

It is also clear that the ultimate accuracy of Euler's method (or any other integration method) is determined by the accuracy, η, to which floating-point numbers are stored on the computer performing the calculation.

The value of η depends on how many bytes the computer hardware uses to store floating-point numbers. For IBM-PC clones, the appropriate value for double precision floating point numbers is $\eta = 2.22 \times 10^{-16}$. It follows that the minimum practical step-length for Euler's method on such a computer is $h_0 \sim 10^{-8}$, yielding a minimum net integration error of $\epsilon_0 \sim 10^{-8}$. This level of accuracy is perfectly adequate for most scientific calculations. Note, however, that the corresponding η value for single precision floating-point numbers is only $\eta = 1.19 \times 10^{-7}$, yielding a minimum practical step-length and a minimum net error for Euler's method of $h_0 \sim 3 \times 10^{-4}$ and $\epsilon_0 \sim 3 \times 10^{-4}$, respectively. This level of accuracy is generally not adequate for scientific calculations, which explains why such calculations are invariably performed using double, rather than single, precision floating-point numbers on IBM-PC clones (and most other types of computer).

An Example of Global and Local Truncation Errors

We illustrate the concept of global and local truncation errors by using the Euler method with a large step size to find an approximate numerical solution of the first order o.d.e. $dy/dx = 8.5 - 20\,x + 12\,x^2 - 2\,x^3$ with the initial condition $y(0) = 1$. We can set the integral of the right-hand side of the o.d.e. (with respect to x from $x = 0$ to $x = x_f$) equal to $y_f - 1$ to find the analytic solution $y_{an}(x)$, which we call **ytrue(x)** in our session:

```
(%i1) ratprint:false$
(%i2) dely : integrate(8.5 - 20*x + 12*x^2 - 2*x^3,x,0,xf);
(%o2) -(xf^4-8*xf^3+20*xf^2-17*xf)/2
(%i3) dely : expand(dely);
(%o3) -xf^4/2+4*xf^3-10*xf^2+17*xf/2
(%i4) yx : 1 + dely, xf = x;
(%o4) -x^4/2+4*x^3-10*x^2+17*x/2+1
(%i5) ytrue(x) := ''yx;
(%o5) ytrue(x):=-x^4/2+4*x^3-10*x^2+17*x/2+1
(%i6) ytrue(0);
(%o6) 1
```

In line `%i5` the "double quote" operator `''` (two single quotes) was used to defeat the normal quote behavior of the delayed assignment operator `:=`. The routine use of the setting `display2d:false` inside the **Xmaxima** interface allows for easy copying of Maxima screen output for use in a later line or function definition (as we do in defining `euler_errors` here - in a separate Notebook2 text document). The following function appears in the code file `myode.mac`

```
euler_errors(n,h) :=
block([x,ye,xn,ytrue,g_err:0, gp_err:0,l_err,numer],numer:true,
   local(yt,dydx),
   /* since x is not bound yet, these func defs work */
   define(yt(x),-x^4/2 + 4*x^3 - 10*x^2 + 17*x/2 + 1),
   define(dydx(x), 8.5 - 20*x + 12*x^2 - 2*x^3),
   x:0,
   ye : yt(x),
   printf(true,"~& ~3tx ~15tytrue ~24tyeuler ~35tgl-err ~48tl-err  ~%"),
   printf(true,"~& ~5f ~10t ~9f ~9f ~%",x,ye,ye),
   for i thru n do (
       xn : x + h,
       ye : ye + dydx(x)*h,
       ytrue : yt(xn),
       g_err : (ytrue - ye)*100/ytrue,
```

```
        l_err : g_err - gp_err,
        printf(true,"~& ~5f ~10t ~9f ~9f ~34t ~6f ~47t ~6f ~%",xn,ytrue,ye,g_err,l_err),
        gp_err : g_err,
        x : xn))$
```

with the output table recording the value of **x**, **ytrue**, **yeuler**, **gl-err** (the global percent relative error) and **l-err** (the local percent relative error - which is simply the difference between the previous step global percent relative error and the current step global percent relative error).

```
(%i7) euler_errors(3,0.5)$
   x           ytrue     yeuler      gl-err        l-err
   0.0           1.0        1.0
   0.5       3.21875       5.25      -63.11        -63.11
   1.0           3.0      5.875      -95.83        -32.73
   1.5       2.21875      5.125      -131.0        -35.15
```

Because we have used such a large value of h, we are seeing truncation error here, and not round-off error. We see that the percent relative error in the first step is 63.1 percent. The percent relative error of the Euler solution after the second step is 95.8 percent, so the local relative truncation error in making the second step is 32.7 percent.

In order to make a plot showing both the exact solution and the Euler solution, we use a homemade Euler integration function we discuss in the next section, **euler1**, (with code in the file **myode.mac**) which has the syntax **euler1(dydx,y,yinit,[x,xinit,xfinal,dx])**.

```
(%i8) load(myode);
(%o8) "c:/k2/myode.mac"
(%i9) pts : euler1(8.5 - 20*x + 12*x^2 - 2*x^3,y,1,[x,0,4,0.5])$
(%i10) fll(pts);
(%o10) [[0.0,1.0],[4.0,7.0],9]
(%i11) pts;
(%o11) [[0.0,1.0],[0.5,5.25],[1.0,5.875],[1.5,5.125],[2.0,4.5],[2.5,4.75],
         [3.0,5.875],[3.5,7.125],[4.0,7.0]]
(%i12) plot2d([-x^4/2+4*x^3-10*x^2+17*x/2+1,[discrete,pts],[discrete,pts] ],
          [x,0,4], [style,[lines,3],[lines,3],
              [points,3,1,1] ],[legend,"exact","Euler",""])$
```

which produces the plot

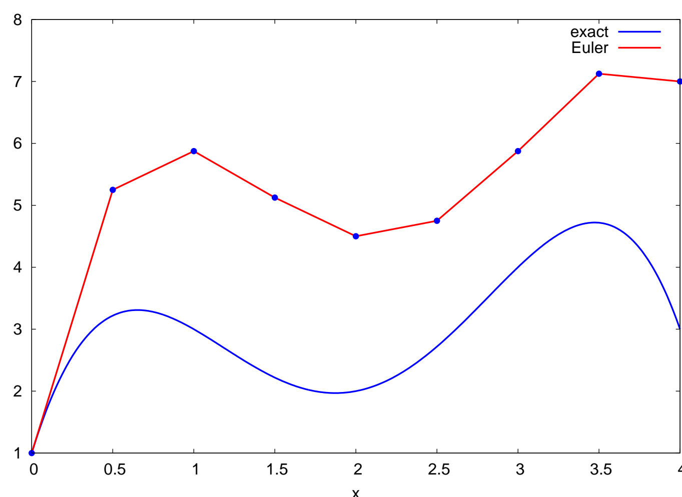

Figure 1: Euler Truncation Error with $h = 0.5$

Maxima Code: euler1: General Euler Function for One O.D.E.

The code for **euler1** is in the file **myode.mac**. This code assumes t is the independent variable, and $x(t)$ is the dependent variable. **euler1(dxdt,x,xi,[t,ti,tf,dt])** returns a list of solution pairs (t, x). (Note that **euler** is a reserved word in Maxima.)

```
/*  euler1(dxdt,x,xi,[t,ti,tf,dt]) for one dependent variable x(t),
       xi, ti, tf, dt should evaluate to numbers.
       dxdt is an expression which can contain, potentially,
       both t and x as symbols.  */

euler1(dvar,var,init,domain) :=
block([dt,t0,n,vs,dvar0,euler_soln,r,k1,numer],numer:true,
   init : float(init),
   domain : float(domain),
   local(dvdt),
   define(dvdt(domain[1],var),float(dvar)),
   dt : domain[4],
   t0 : domain[2],
   n: fix((domain[3] - t0)/dt),
   vs: init,
   dvar0 : dvdt(t0, vs),
   if (not(numberp(dvar0))) then
         error("Expecting a number when the initial state is
                 replaced in dvdt, but instead found:",dvar0),
  euler_soln : [[t0,vs]],

  for i thru n do (
      r: errcatch (k1 : dvdt(t0,vs)),
      if length(r) = 0 then return()
      else vs : vs + k1*dt,
      t0: t0 + dt,
      euler_soln : cons([t0,vs], euler_soln)),

  reverse(euler_soln))$
```

The symbol used for the independent variable does not affect the list of numbers returned - the invocation
pts : euler1(-t*y,y,1,[t,0,1,0.1])$
produces the same output as the invocation
pts : euler1(-x*y,y,1,[x,0,1,0.1])$** or **pts : euler1(-t*x,x,1,[t,0,1,0.1])$.

The function **fll**, one of a collection of small file utility functions available in the chapter two code file **k2util.mac**, returns the first and last elements of a list, and also the number of elements in the list, and has the definition

```
fll(x) := [first(x),last(x),length(x)]$
```

Example 1

Here we use **euler1** with the same example used in the last section: $dy/dx = 8.5 - 20\,x + 12\,x^2 - 2\,x^3$ with the initial condition $y(0) = 1$.

We show convergence to the exact (analytic) solution as we decrease the integration step size **dt = h**.

```
(%i1) load(myode);
(%o1) "c:/k2/myode.mac"
(%i2) dydx : 8.5 - 20*x + 12*x^2 - 2*x^3$
(%i3) yt : -x^4/2+4*x^3-10*x^2+17*x/2+1$
(%i4) case(h):= euler1(dydx,y,1,[x,0,4,h])$
(%i5) pts1 : case(0.5);
(%o5) [[0.0,1.0],[0.5,5.25],[1.0,5.875],[1.5,5.125],[2.0,4.5],[2.5,4.75],
       [3.0,5.875],[3.5,7.125],[4.0,7.0]]
```

```
(%i6) pts2 : case(0.2)$
(%i7) fll(pts2);
(%o7) [[0.0,1.0],[4.0,4.6],21]
(%i8) pts3 : case(0.1)$
(%i9) fll(pts3);
(%o9) [[0.0,1.0],[4.0,3.8],41]
(%i10) plot2d([yt,[discrete,pts1],[discrete,pts2],[discrete,pts3] ],
         [x,0,4], [style,[lines,3]],[legend,"exact","h=0.5","h=0.2","h=0.1"],
         [gnuplot_preamble,"set key bottom right;set grid"])$
```

which produces the plot

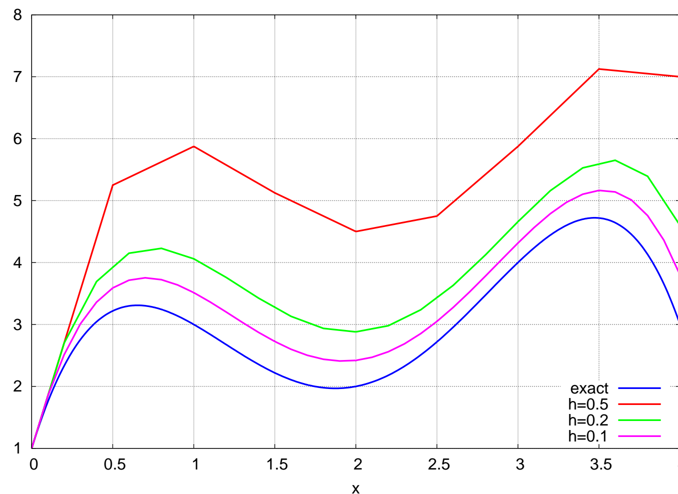

Figure 2: Convergence to Exact Solution

Next we use **euler1** to make a plot of the Euler method net error (at $x = 1$) as a function of integration step size h, for the same example o.d.e. we have been using as a test case. The function **yfdiff**, which calls **euler1**, is in the file **myode.mac**.

```
yfdiff(dydx,ytrue,xfinal,hL) :=
block([tval,yerrL:[],h,esoln,yerr,numer],numer:true,
    tval : float(subst(x = xfinal,ytrue)), /* true value */
    print(" tval = ",tval),
    for h in hL do (
         esoln : euler1(dydx,y,1,[x,0,xfinal,h]),
         yerr : second(last(esoln)) - tval,
         print(" ", h, yerr),
         yerrL : cons([h, yerr], yerrL)),
    reverse(yerrL))$
```

Here we use the function **yfdiff** to make a plot of the net Euler error as a function of integration step size, when integrating over the interval **[x,0,1]**. The list **hL** contains the step sizes to be used.

```
(%i1) load(myode);
(%o1) "c:/k2/myode.mac"
(%i2) dydx : 8.5 - 20*x + 12*x^2 - 2*x^3$
(%i3) yt : -x^4/2+4*x^3-10*x^2+17*x/2+1$
(%i4) hL : [0.5,0.2,0.1,0.05,0.02,0.01,0.005,0.002,0.001]$
(%i5) yerr_pts : yfdiff(dydx,yt,1,hL)$
 tval =  3.0
  0.5 2.875
  0.2 1.06
  0.1 0.515
  0.05 0.25375
  0.02 0.1006
  0.01 0.05015
```

```
  0.005 0.0250375
  0.002 0.010006
  0.001 0.0050015
(%i6) fll(yerr_pts);
(%o6) [[0.5,2.875],[0.001,0.0050015],9]
(%i7) plot2d([discrete, yerr_pts],[xlabel,"h"],[ylabel,"yfdiff(1)"],
          [style,[lines,3]],[gnuplot_preamble,"set grid"])$
```

which produces the plot

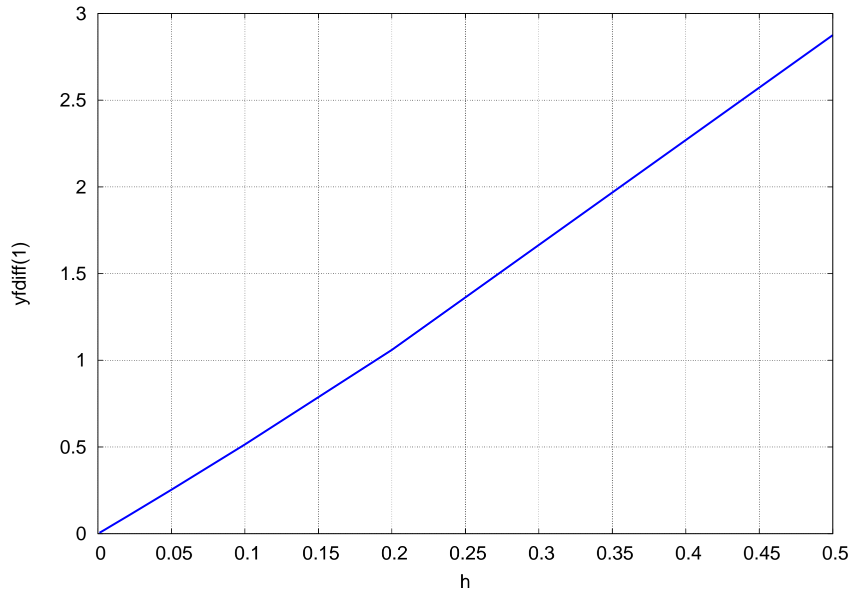

Figure 3: Net Euler Error at $x_f = 1$ vs. h

Let the global error at $x = 1$ be $E(h)$, where h is the step size. We will assume the form $E(h) = a + b\,h$ and find the best values of a and b using the standard least squares fit Maxima method. (See also our discussion of fitting data in Chapter 3 of Maxima by Example.) We will then make a plot of both the data and the straight line fit. We needed to turn the list of data into a Maxima matrix in order to use the Maxima method **lsquares_estimates**.

```
(%i8) data : abs(yerr_pts);
(%o8) [[0.5,2.875],[0.2,1.06],[0.1,0.515],[0.05,0.25375],[0.02,0.1006],
        [0.01,0.05015],[0.005,0.0250375],[0.002,0.010006],[0.001,0.0050015]]
(%i9) dataM : apply('matrix,data);
(%o9) matrix([0.5,2.875],[0.2,1.06],[0.1,0.515],[0.05,0.25375],[0.02,0.1006],
              [0.01,0.05015],[0.005,0.0250375],[0.002,0.010006],
              [0.001,0.0050015])
(%i10) load(lsquares);
(%o10) "C:/PROGRA~1/MAXIMA~3.2/share/maxima/5.31.2/share/lsquares/lsquares.mac"
(%i11) result : lsquares_estimates(dataM,[h,E],E = a+b*h,[a,b],
                initial=[0,1],iprint=[-1,0]);
(%o11) [[a = -2685980626178778230522532194715176213611334132351051506109903
            /1272392566591281706497726856625039827472468194643718179950844452,
          b = 6739084303336038655816563166211558297140319079262867181871266900
            /1176963124096935578510397342378161840412033080045439316454531181]]
(%i12) result : float(result);
(%o12) [[a = -0.0211097,b = 5.7258245]]
(%i13) myfit : a+b*h, result;
(%o13) 5.7258245*h-0.0211097
(%i14) plot2d([myfit,[discrete,data]],[h,0,0.1],
          [style,[lines,3],[points,3,1,1]],[legend,false],[xlabel,"h"],
           [ylabel,"E"],[gnuplot_preamble,"set grid"])$
```

which produces the plot

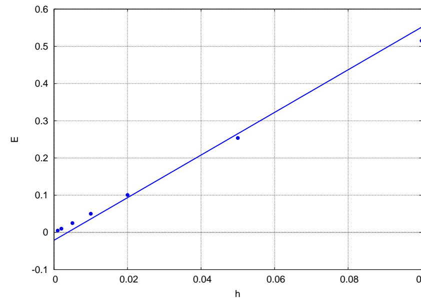

Figure 4: Global Euler Error at $x = 1$ vs. h

For small values of h the global errors are larger than we would expect if the global error $E(h) \sim O(h)$.

Example 2

A second example is the o.d.e. $dy/dx = -x\,y$ with the initial condition $y(0) = 1$, integrated over the interval **[x,0,3]**, with the analytic solution $y = e^{-\frac{1}{2}x^2}$.

```
(%i15) dydx : -x*y;
(%o15) -x*y
(%i16) yt : exp(-x^2/2);
(%o16) %e^-(x^2/2)
(%i17) hL : [0.5,0.2,0.1];
(%o17) [0.5,0.2,0.1]
(%i18) yerr_pts : yfdiff(dydx,yt,1,hL);
 tval =  0.606531
  0.5 0.143469
  0.2 0.0463308
  0.1 0.0216258
(%o18) [[0.5,0.143469],[0.2,0.0463308],[0.1,0.0216258]]
(%i19) case(h):= euler1(dydx,y,1,[x,0,3,h])$
(%i20) pts1 : case(0.5)$
(%i21) fll(pts1);
(%o21) [[0.0,1.0],[3.0,0.0],7]
(%i22) pts2 : case(0.2)$
(%i23) fll(pts2);
(%o23) [[0.0,1.0],[3.0,0.00458968],16]
(%i24) plot2d([yt,[discrete,pts1],[discrete,pts2] ],[x,0,3], [style,[lines,3]],
                 [legend,"exact","h=0.5","h=0.2"],[ylabel,"y"],
                   [gnuplot_preamble,"set grid"])$
```

which produces the plot

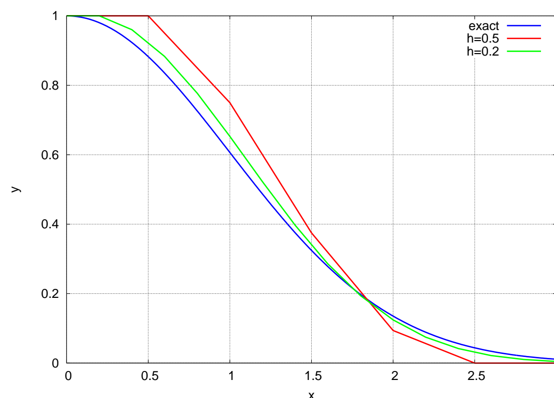

Figure 5: Euler Convergence to $dy/dx = -x\,y$ with $y(0) = 1$

Example 3

As a third example, we use **yfdiff** to determine the dependence of global error $E(h)$ on the step size h for the case of $dy/dx = -y$ with $y(0) = 1$, with the analytic solution $y(x) = e^{-x}$.

```
(%i1) load(myode);
(%o1) "c:/k2/myode.mac"
(%i2) hL : [0.1,0.05,0.02,0.01,0.005];
(%o2) [0.1,0.05,0.02,0.01,0.005]
(%i3) yfdiff(-y,exp(-x),3,hL);
 tval =  0.0497871
  0.1 -0.00739591
  0.05 -0.00371727
  0.02 -0.00149105
  0.01 -7.46174297E-4
  0.005 -3.73246258E-4
(%o3) [[0.1,-0.00739591],[0.05,-0.00371727],[0.02,-0.00149105],
       [0.01,-7.46174297E-4],[0.005,-3.73246258E-4]]
(%i4) data : %;
(%o4) [[0.1,-0.00739591],[0.05,-0.00371727],[0.02,-0.00149105],
       [0.01,-7.46174297E-4],[0.005,-3.73246258E-4]]
(%i5) data : abs(data);
(%o5) [[0.1,0.00739591],[0.05,0.00371727],[0.02,0.00149105],
        [0.01,7.46174297E-4],[0.005,3.73246258E-4]]
(%i6) dataM : apply('matrix,data);
(%o6) matrix([0.1,0.00739591],[0.05,0.00371727],[0.02,0.00149105],
              [0.01,7.46174297E-4],[0.005,3.73246258E-4])
```

We will assume the form $E(h) = a + b\,h$ and find the best values of a and b using the standard least squares fit Maxima method. (See also our discussion of fitting data in Chapter 3 of Maxima by Example.) We will then make a plot of both the data and the straight line fit. We needed to turn the list of data into a Maxima matrix in order to use the Maxima method **lsquares_estimates**.

```
(%i7) load(lsquares);
(%o7) "C:/PROGRA~1/MAXIMA~3.2/share/maxima/5.31.2/share/lsquares/lsquares.mac"

(%i8) result : lsquares_estimates(dataM,[h,E],E = a+b*h,[a,b],
                initial=[0,1],iprint=[-1,0]);
(%o8) [[a = 3009959174137407041030574176001637239330 7
           /31119164908730682331837979661477330453312703 68,
         b = 28754287790257175585747436662495393659067425
           /38898956135913352914797474576846663066640879 6]]
(%i9) result : float(result);
(%o9) [[a = 9.67236487E-6,b = 0.0739205]]
(%i10) myfit : a+b*h, result;
(%o10) 0.0739205*h+9.67236487E-6
(%i20) plot2d([myfit,[discrete,data]],[h,0,0.1],
          [style,[lines,3],[points,3,1,1]],[legend,false],[xlabel,"h"],
           [ylabel,"E"],[gnuplot_preamble,"set grid"])$
```

which produces the plot

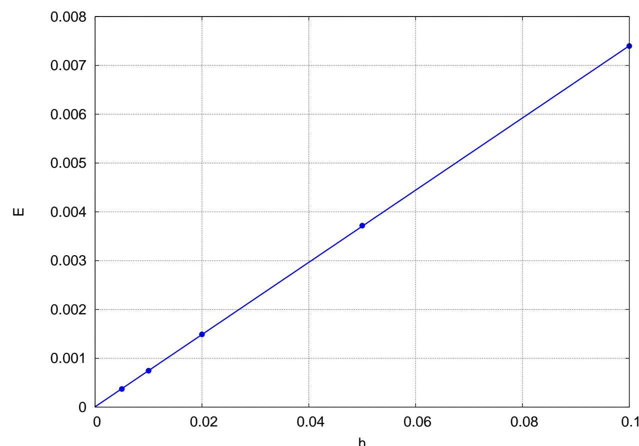

Figure 6: Global Errors vs. h for $dy/dx = -y$ with $y(0) = 1$

Maxima Code: myeuler for Arbitrary Number of First Order O.D.E.'s

In the file **myode.mac** is the code for **myeuler** which has the syntax
myeuler(dxdt,x,xinit,[t,tinit,tfinal,dt]) or
myeuler([dxdt,dydt],[x,y],[xinit,yinit],[t,tinit,tfinal,dt])
and similar for more than two first order o.d.e.'s.

```
myeuler(ode, var, init, domain) :=
block([uvw,duvw,esoln,n,k1,t0,dt,
            r,numer:true,display2d:false],
  init : float(init),
  domain : float(domain),
  if (not(listp(ode))) then (
      ode : [ode],
      var : [var],
      init : [init]),

  local(efunc),
  define(funmake(efunc,cons(domain[1],var)),float(ode)),
  translate(efunc),
  dt : domain[4],
  t0 : domain[2],
  n: fix((domain[3] - t0)/dt),
  uvw: init,

  duvw : apply(efunc,cons(t0,uvw)),
  if (not(numberp(last(duvw)))) then
    error("Expecting a number when the initial state
      is replaced in the equations, but instead
      found:",last(duvw)),

  esoln: [cons(t0, init)],
  for i thru n do (
      r: errcatch (k1: apply(efunc,cons(t0,uvw))),
      if length(r)=0 then return()
      else uvw: uvw + k1*dt,
      t0: t0 + dt,
      esoln : cons(cons(t0,uvw), esoln)),
  reverse(esoln))$
```

The Maxima code **myeuler** is adapted from the code design of the Maxima function **rk()**, which can be found in **...share/dynamics/dynamics.mac** in Maxima v. 5.28.0, copyright **2007 Jaime E. Villate <villate@fe.up.pt>**.

We test **myeuler** on the simple harmonic oscillator with unit period $dx/dt = v_x$, $dv_x/dt = -4\,\pi^2\,x$, with the initial conditions $x(0) = 1$, $v_x(0) = 0$, and integrate over the time interval **[t,0,1]** for three different values of the time step **dt = h**. The analytic solution is $x = \cos(2\,\pi\,t)$.

We remind the reader that the list utility functions **fll** and **take** are in the chapter 2 file **k2util.mac** (as well as in **myode.mac**) and have the definitions

```
fll(x) := [first(x),last(x),length(x)]$

take(%aL,%nn) := (map(lambda([x],part(x,%nn)), %aL))$
```

```
(%i1) load(myode);
(%o1) "c:/k2/myode.mac"
(%i2) case(dt) := myeuler([vx,-4*%pi^2*x],[x,vx],[1,0],[t,0,1,dt])$
(%i3) pts1 : case(0.01)$
(%i4) fll(pts1);
(%o4) [[0.0,1.0,0.0],[1.0,1.2177068,0.0631137],101]
(%i5) epts1 : [discrete,take(pts1,1),take(pts1,2)]$
(%i6) pts2 : case(0.005)$
(%i7) fll(pts2);
(%o7) [[0.0,1.0,0.0],[1.0,1.1036747,0.0143259],201]
```

```
(%i8) epts2 : [discrete,take(pts2,1),take(pts2,2)]$
(%i9) pts3 : case(0.002)$
(%i10) fll(pts3);
(%o10) [[0.0,1.0,0.0],[1.0,1.0402647,0.00216153],501]
(%i11) epts3 : [discrete,take(pts3,1),take(pts3,2)]$
(%i12) plot2d([cos(2*%pi*t),epts1,epts2,epts3],[t,0,1],[xlabel,"T"],[ylabel,"X"],
          [style,[lines,3]],[legend,"exact","h=0.01","h=0.005","h=0.002"],
          [gnuplot_preamble,"set key bottom right;set grid"])$
```

which produces the plot

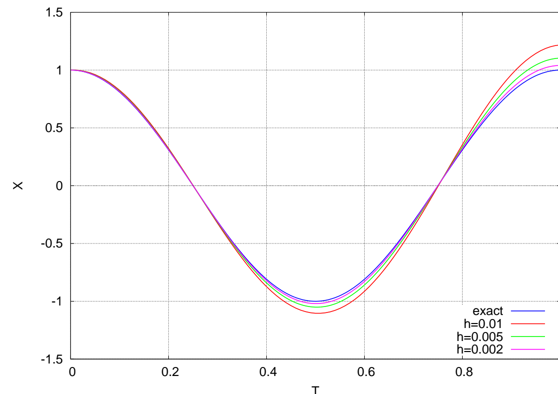

Figure 7: Exact and Euler SHO Solutions

Interactive Exploration of myeuler Code

When looking at new code with unfamiliar features, or when designing new code, it helps to step through the problem interactively first, before trying to write a general function. For example, a typical interactive approach to understanding the code of **myeuler** might be:

```
(%i1) var : [y1,y2];
(%o1) [y1,y2]
(%i2) tvL : cons(t,var);
(%o2) [t,y1,y2]
(%i3) ode : [y2,-4*%pi^2*y1];
(%o3) [y2,-4*%pi^2*y1]
(%i4) define(funmake(efunc,tvL),float(ode))$
(%i5) init : [1,0];
(%o5) [1,0]
(%i6) uvw : init;
(%o6) [1,0]
(%i7) t0 : 0;
(%o7) 0
(%i8) esoln : [cons(t0, uvw)];
(%o8) [[0,1,0]]
(%i10) tuvw : cons(t0,uvw);
(%o10) [0,1,0]
(%i11) k1 : apply(efunc,tuvw);
(%o11) [0,-39.47841760435743]
(%i12) dt : 0.01;
(%o12) 0.01
(%i13) uvw : uvw + k1*dt;
(%o13) [1,-0.39478417604357]
(%i14) t0 : t0 + dt;
(%o14) 0.01
(%i15) esoln : cons(cons(t0,uvw),esoln);
(%o15) [[0.01,1,-0.39478417604357],[0,1,0]]
```

We see that we are using the Maxima function **apply**; if **f** is not yet defined, we can see what Maxima does with **apply(f,[1,2,3])** (that is, applying **f** to a Maxima list).

```
(%i16) apply(f,[1,2,3]);
(%o16) f(1,2,3)
```

Now let's bind **f** to some function definition.

```
(%i18) f(a,b,c) := a*b/c$
(%i19) apply(f,[1,2,3]);
(%o19) 2/3
(%i20) apply('f,[1,2,3]);
(%o20) 2/3
```

Note that the single quote **'** does not defeat the use of the function definition of **f** here.

The line **define(funmake(efunc,tvL),float(ode))$** is one way to define a function:
define(funmake(myf,[x,y,z]),expr), where **expr** depends on **x,y,z**. The **expr** does not have to actually contain all of the symbols **x,y,z**. In our code for **myeuler**, and in the case of the simple harmonic oscillator done above, the actual content of the definition of **efunc** was
define(funmake(efunc,[t,y1,y2]),[y2,-39.47841760435743*y1])$, in which **expr** was a list of two expressions, neither of which contain the symbol **t**.

Numerical Instability Example

We quote a short example explaining the issue of numerical instability, presented by Richard Fitzpatrick (Physics, University of Texas) on his computational physics web page:
http://farside.ph.utexas.edu/teaching/329/lectures/node34.html.

> **Numerical instabilities**
> Consider the following example. Suppose that our o.d.e. is $y' = -\alpha\, y$, where $\alpha > 0$, subject to the boundary condition $y(0) = 1$.
>
> Of course, we can solve this problem analytically to give $y(x) = \exp(-\alpha\, x)$.
>
> Note that the solution is a monotonically decreasing function of x. We can also solve this problem numerically using Euler's method. Appropriate grid-points are $x_n = n\, h$, where $n = 0, 1, 2, \cdots$. Euler's method yields $y_{n+1} = (1-\alpha\, h)\, y_n$.
>
> Note one curious fact. If $h > 2/\alpha$ then $|y_{n+1}| > |y_n|$. In other words, if the step-length is made too large then the numerical solution becomes an oscillatory function of x of monotonically increasing amplitude: i.e., the numerical solution diverges from the actual solution. This type of catastrophic failure of a numerical integration scheme is called a numerical instability. All simple integration schemes become unstable if the step-length is made sufficiently large.

As an example, we use Euler's method with the o.d.e. $dy/dx = -20\, y$, which has the analytic solution $y(x) = e^{-20\, x}$. We expect numerical instability (using the Euler algorithm) if the step size $h > 0.1$.

```
(%i1) load(myode);
(%o1) "c:/k2/myode.mac"
(%i2) case(dx) := euler1(-20*y,y,1,[x,0,1,dx])$
(%i3) pts1 : case(0.12)$
(%i4) fll(pts1);
(%o4) [[0.0,1.0],[0.96,14.757891],9]
(%i5) pts2 : case(0.098)$
(%i6) fll(pts2);
(%o6) [[0.0,1.0],[0.98,0.664833],11]
(%i7) pts3 : case(0.05)$
(%i8) fll(pts3);
(%o8) [[0.0,1.0],[1.0,0.0],21]
(%i9) plot2d([exp(-20*x),[discrete,pts1],[discrete,pts2],
        [discrete,pts3]], [x,0,0.4], [style,[lines,3]],
          [legend,"exact","h=0.12","h=0.098","h=0.05"],
          [gnuplot_preamble,"set key bottom left;set grid"])$
```

which produces the plot

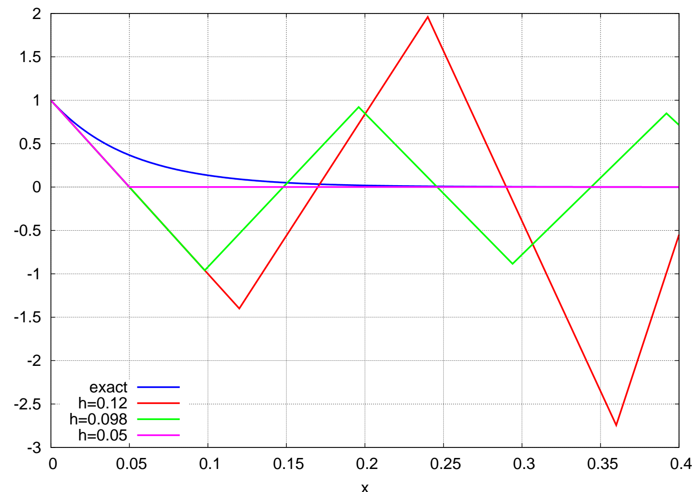

Figure 8: Euler Integration Instability Example

3 The Euler Method Using R

R Code: euler1

Rather than "translate" the Maxima code for **euler1** into **R**, we use an entirely different approach which corresponds better with the natural design of the **R** language and the natural methods of carrying out tasks in **R**.

euler1(init,grid,func) **R** code is in the file **myode.R**, and has the definition

```
## euler1(init,grid,func) calls func to advance the Euler
##  solution. func(iv,w) corresponds to independent
##  variable = iv, dependent variable = w

euler1 = function(init, grid ,func)  {
   n = length(grid)
   div = grid[2] - grid[1]
   w.num = vector(length = n)
   w.num[1] = init
   for (j in 1:(n-1)) {
      w.num [j+1] =  w.num [j] +
         div*func(grid[j], w.num[j]) }
   w.num}
```

The value of the first argument **init** is the initial (numerical) value of the dependent variable. The value of the second argument **grid** is a vector which contains the discrete numerical values of the independent variable (separation being constant) at which output values of the dependent variable are desired. The value of the third argument **func** is the name (chosen by the user) of a defined function of the form **func(indep-var, dep-var)** which implements the right hand side of the o.d.e. $dy/dt = f(t, y)$, or $dy/dx = f(x, y)$, or $du/dv = f(v, u)$, etc.

Also in **myode.R** is the vector utility function **fll** which prints to the screen the value of the first and last elements of a vector, as well as the length.

```
##  list utility: print out first, last and length of a vector
fll = function(xL) {
    xlen = length(xL)
    cat(" ",xL[1]," ",xL[xlen]," ",xlen,"\n") }
```

After loading in **myode.R**, we define the **R** function **derivs** which corresponds to the right-hand-side of the o.d.e. $dy/dt = -y$. If $y(0) = 1$, then the analytic solution is $y(t) = e^{-t}$. We then use **euler1** to integrate over the time interval **[t,0,3]**.

```
> getwd()
[1] "c:/k2"
> source("myode.R")
> derivs = function(t,y) {-y}
> tL = seq(0,3,0.1)
> fll(tL)
  0   3   31
> head(tL)
[1] 0.0 0.1 0.2 0.3 0.4 0.5
> yL = euler1(init=1,grid=tL,func=derivs)
> fll(yL)
  1   0.04239116   31
> head(yL)
[1] 1.00000 0.90000 0.81000 0.72900 0.65610 0.59049
> plot(tL, yL, pch=19, xlab = "t", ylab = "y")
> lines(tL, yL)
> grid(lty="solid", col="darkgray")
> curve(exp(-t),0,3, n=200, add=TRUE, col="blue", lwd=3, xname="t")
```

which produces the plot

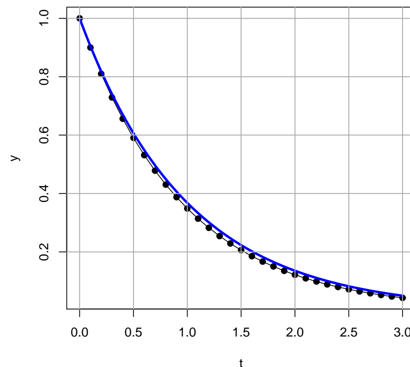

Figure 9: Euler Solution (dots) to $dy/dt = -y$ for $dt = 0.1$

Note that we could have called **euler1** with the syntax: **yL = euler1(1,tL,derivs)** since we are maintaining the default order of the arguments.

```
> yL = euler1(1,tL,derivs)
> fll(yL)
  1   0.04239116   31
> head(yL)
[1] 1.00000 0.90000 0.81000 0.72900 0.65610 0.59049
```

As a second example of using **euler1**, we integrate the o.d.e. which depends on two parameters r and K, the "logistic equation" $dy/dt = r\,y(1 - y/K)$ with $y(0) = 2$ and with $r = 1$ and two values of K.

```
> derivs = function(t,y){r*y*(1-y/K)}
> tL = seq(0,20,0.2)
> fll(tL)
  0   20   101
> r = 1; K = 10
> yL10 = euler1(2,tL,derivs)
> fll(yL10)
  2   10   101
> K = 20
> yL20 = euler1(2,tL,derivs)
> fll(yL20)
  2   20   101
> plot(tL, yL20, type="l", lwd=3, col="blue",
+      xlab = "t", ylab = "y")
```

```
> lines(tL, yL10,lwd=3,col="red")
> grid(lty="solid", col="darkgray")
> legend("bottom", col=c("blue","red"),
    legend = c("K = 20","K = 10"),lwd=3,cex=1.5)
```

which produces the plot

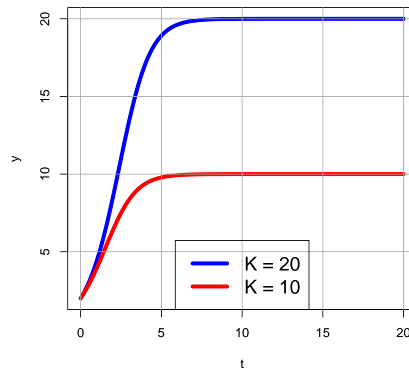

Figure 10: Euler Solutions for $dy/dt = y\,(1 - y/K)$ for $y(0) = 2, dt = 0.2$

As a third example of using **euler1** we integrate the o.d.e. $dy/dt = -t\,y$ with $y(0) = 1$. The analytic solution is $y(t) = e^{-\frac{1}{2}t^2}$.

```
> derivs = function(t,y) {-t*y}
> tL = seq(0,3,0.1)
> fll(tL)
  0   3   31
> yL = euler1(1,tL,derivs)
> fll(yL)
  1   0.007791097   31
> plot(tL, yL, type="l", lwd=3, col="blue",
+      xlab = "t", ylab = "y")
> curve(exp(-t^2/2),0,3,n=200,add=TRUE,col="red",lwd=3,xname="t")
> grid(lty="solid", col="darkgray")
> legend("topright", col=c("blue","red"),
+      legend = c("Euler","Exact"),lwd=3,cex=1.5)
```

which produces

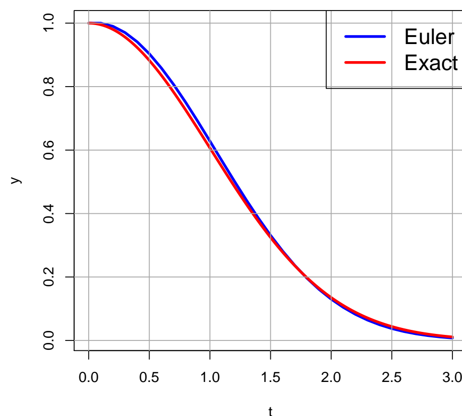

Figure 11: Euler Solution for $dy/dt = -t\,y$ for $y(0) = 1, dt = 0.1$

R Code: myeuler for Arbitrary Number of First Order O.D.E.'s

The code for **myeuler** is also in the file **myode.R** and can be used for an arbitrary number of first order o.d.e.'s, including just one o.d.e.

For one dimension, the syntax is the same as for **euler1** and the design of **func(t,y)** can be the same as used with **euler1** (see below).

For two or more o.d.e.'s, **myeuler** returns a list of vectors, and each solution vector must be extracted as a list element: **out[[1]]** for example, using double brackets (see below).

Here is the code for **myeuler**.

```
myeuler = function(init, grid ,func)  {
   num.var = length(init)
   solnList = list()
   n.grid = length(grid)
   for (k in 1:num.var) {
        solnList[[k]] = vector(length = n.grid)
        solnList[[k]][1] = init[k] }
   div = grid[2] - grid[1]
   yL = vector(length = num.var)
   for (j in 1:(n.grid-1)) {
      for (k in 1:num.var) yL[k] = solnList[[k]][j]
      dyL = func(grid[j], yL)  # returns a vector of derivatives
      for (k in 1:num.var) solnList[[k]][j+1] = solnList[[k]][j] +
                                               div*dyL[k]}
   if (num.var==1) solnList[[1]] else solnList}
```

Two Examples of Using R myeuler

Example 1

Here is an example for the solution of the single o.d.e. $dy/dt = -t\,y$ for $y(0) = 1$, after loading in **myeuler.R**.

```
> tL = seq(0,3,0.1)
> fll(tL)
  0   3   31
> deriv = function(t,y) { -t*y }
> yL = myeuler(1,tL,deriv)
> fll(yL)
  1   0.007791097   31
> head(yL)
[1] 1.0000000 1.0000000 0.9900000 0.9702000 0.9410940 0.9034502
> tail(yL)
[1] 0.037617673 0.028213255 0.020877809 0.015240800 0.010973376 0.007791097
```

Example 2

Here is an example of use for the simple harmonic oscillator with unit period, which requires two first order o.d.e.'s. In the code for **func**, called **sho**, we let **y[1]** represent **x** and **y[2]** represent **vx**.

Thus the code **sho** must take the form (note **sho** returns the vector **c(dx,dvx)**; the names of the local function variables **dx** and **dvx** are of course arbitrary and can be changed):

```
sho = function(t,y) {
    with(as.list(y), {
        dx = y[2]
        dvx = -4*pi^2*y[1]
        c(dx,dvx)})}
```

It is important that the order of the vector returned be the same as the order of the numbers provided for the vector argument **init**. If **init = c(1,0)** corresponding to **x(0) = 1**, **vx(0) = 0**, then **sho** should return the derivatives **c(dxdt, dvxdt)**.

The definition of **sho** can be shortened to

```
sho = function(t,y) {
    with(as.list(y), c(y[2], -4*pi^2*y[1]) )}
```

and can be further shortened to

```
sho = function(t,y) with(as.list(y), c(y[2], -4*pi^2*y[1]) )
```

Each of these three versions of the function **sho** can be used with **myeuler**.

```
> getwd()
[1] "c:/k2"
> source("myeuler.R")
> sho = function(t,y) {
+     with(as.list(y), {
+         dx = y[2]
+         dvx = -4*pi^2*y[1]
+         c(dx,dvx)})}
> tL = seq(0,1,0.001)
> fll(tL)
  0   1   1001
> out = myeuler(c(1,0),tL,sho)
> xL = out[[1]]
> fll(xL)
  1   1.019935   1001
> vxL = out[[2]]
> fll(vxL)
  0   0.0005298591   1001
> max(vxL)
[1] 6.376897
> plot(tL,vxL,type="l",lwd=3,col="red",xlab = "t",ylab = "",ylim=c(-7,7))
> lines(tL,xL,lwd=3,col="blue")
> grid(lty="solid",col="darkgray")
> legend("topleft", col=c("blue","red"), lwd=3, legend=c("x","vx"),cex=1.2)
```

which produces

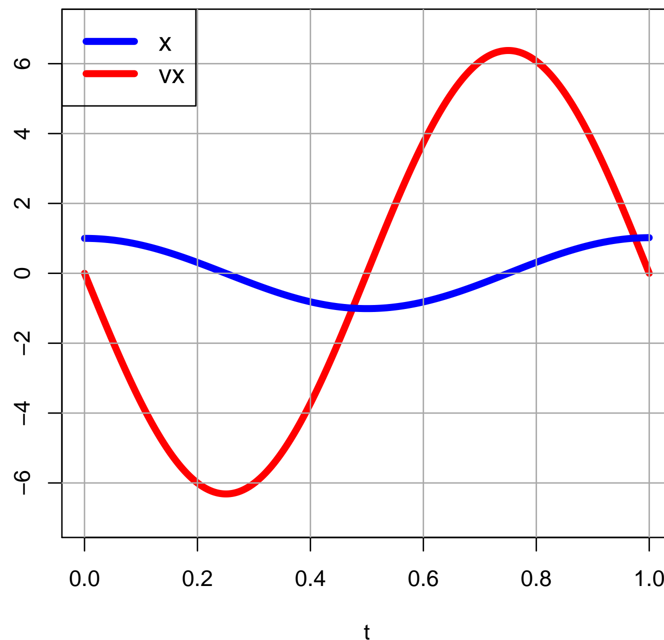

Figure 12: SHO: $dx/dt = v_x$, $dv_x/dt = -4\,\pi\,x$, $x(0) = 1, v_x(0) = 0$

4 Fourth Order Runge-Kutta Code for Maxima and R

We quote from Richard Fitzpatrick's (Physics, University of Texas) Computational Physics web page **http://farside.ph.utexas.edu/teaching/329/lectures/node35.html** with some editing and elisions. His definition of k_1, k_2, k_3, k_4 differs from ours by a factor of h.

> **Runge-Kutta methods**
>
> There are two main reasons why Euler's method is not generally used in scientific computing. Firstly, the truncation error per step associated with this method is far larger than those associated with other, more advanced, methods (for a given value of h). Secondly, Euler's method is too prone to numerical instabilities.
>
> The methods most commonly employed by scientists to integrate o.d.e.'s were first developed by the German mathematicians C.D.T. Runge and M.W. Kutta in the latter half of the nineteenth century. The basic reasoning behind so-called Runge-Kutta methods is outlined in the following.
>
> The main reason that Euler's method has such a large truncation error per step is that in evolving the solution from x_n to x_{n+1} the method only evaluates derivatives at the beginning of the interval: i.e., at x_n. The method is, therefore, very asymmetric with respect to the beginning and the end of the interval.
>
> We can construct a more symmetric integration method by making an Euler-like trial step to the midpoint of the interval, and then using the values of both x and y at the midpoint to make the real step across the interval. To be more exact, $k_1 = h\, f(x_n, y_n)$, $\quad k_2 = h\, f(x_n + h/2, y_n + k_1/2)$, $\quad y_{n+1} = y_n + k_2 + O(h^3)$.
>
> As indicated in the error term, this symmetrization cancels out the first-order error, making the method second-order. In fact, the above method is generally known as a second-order Runge-Kutta method. Euler's method can be thought of as a first-order Runge-Kutta method.
>
> Of course, there is no need to stop at a second-order method. By using two trial steps per interval, it is possible to cancel out both the first and second-order error terms, and, thereby, construct a third-order Runge-Kutta method. Likewise, three trial steps per interval yield a fourth-order method, and so on.
>
> The general expression for the total error, ϵ, associated with integrating our o.d.e. over an x-interval of order unity using an nth-order Runge-Kutta method is approximately $\epsilon \sim \frac{\eta}{h} + h^n$.
>
> Here, the first term corresponds to round-off error, whereas the second term represents truncation error. The minimum practical step-length, h_0, and the minimum error, ϵ_0, take the values $h_0 \sim \eta^{1/(n+1)}$, $\epsilon_0 \sim \eta^{n/(n+1)}$,respectively. It can be seen that h_0 increases and ϵ_0 decreases as n gets larger. However, the relative change in these quantities becomes progressively less dramatic as n increases.
>
> In the majority of cases, the limiting factor when numerically integrating an o.d.e. is not round-off error, but rather the computational effort involved in calculating the function $f(x, y)$. Note that, in general, an nth-order Runge-Kutta method requires n evaluations of this function per step. It can easily be appreciated that as n is increased a point is quickly reached beyond which any benefits associated with the increased accuracy of a higher order method are more than offset by the computational "cost" involved in the necessary additional evaluation of $f(x, y)$ per step. Although there is no hard and fast general rule, in most problems encountered in computational physics this point corresponds to $n = 4$. In other words, in most situations of interest a fourth-order Runge Kutta integration method represents an appropriate compromise between the competing requirements of a low truncation error per step and a low computational cost per step.

We now revert to **our convention** on the definition of the k_1, k_2, k_3, k_4, which differs from Fitzpatrick's by a factor of h.

To solve the single first order o.d.e. $dy/dx = f(x, y)$ using a step h, the standard fourth-order Runge-Kutta method takes the form (using **our convention**):

$$y_{n+1} = y_n + \frac{h}{6}(k_1 + 2\,k_2 + 2\,k_3 + k_4) \tag{4.1}$$

$$x_{n+1} = x_n + h \tag{4.2}$$

where

$$k_1 = f(x_n, y_n) \tag{4.3}$$

$$k_2 = f(x_n + h/2, y_n + h\, k_1/2) \tag{4.4}$$

$$k_3 = f(x_n + h/2, y_n + h\, k_2/2) \tag{4.5}$$

$$k_4 = f(x_n + h, y_n + h\, k_3) \tag{4.6}$$

4.1 Maxima Code: rk4

The standard available Maxima method for fourth order fixed step Runge-Kutta integration is **rk()**, which is currently written in Lisp, and can be viewed in
...share\dynamics\complex_dynamics.lisp.

Here we present *Maxima* (rather than Lisp) code for the standard fourth order (fixed step) Runge-Kutta integration code, adapted from Maxima (ver. 5.28) **rk** code, written by Maxima developer Jaime E. Villate, **<villate@fe.up.pt>**.

Our version is called **rk4** and can be found in the code file **myode.mac**. In this code the independent variable is called **t**, and the step length **h** is called **dt**. The current solution **uvw** is a Maxima list. This code returns a list of the form
[[t0,x1(t0),x2(t0)...],[t1,x1(t1),x2(t1),...],...].
This code can integrate an arbitrary number of first order o.d.e.'s. This code does not do syntax checks.

```
/*  the rk4  syntax is the same as Maxima's rk() syntax.

   if the dxdt expression is a function of (t,x), then:
    for one o.d.e: rk4(dxdt,x,xinit,[t,tinit,tfinal,dt])

        If the dx1dt expression and the dx2dt expression are functions of (t,x1,x2),
    then for two o.d.e.'s:

      rk4([dx1dt,dx2dt],[x1,x2],[x1init,x2init],[t,tinit,tfinal,dt])
     and so on.
     */

rk4(ode, var, init, domain) :=
block([uvw,rksoln,n,k1,k2,k3,k4,t0,t1,dt,
            r,numer:true,display2d:false],
  init : float(init),
  domain : float(domain),
  if (not(listp(ode))) then (
      ode : [ode],
      var : [var],
      init : [init]),

  local(rkfunc),
  define(funmake(rkfunc,cons(domain[1],var)),float(ode)),
  translate(rkfunc),
  dt : domain[4],
  t0 : domain[2],
  n: fix((domain[3] - t0)/dt),
  uvw: init,

  if (not(numberp(last(apply(rkfunc,cons(t0,uvw)))))) then
    error("Expecting a number when the initial state is
    replaced in the equations, but instead found:",
          last(apply(rkfunc,cons(t0,uvw)))),

  rksoln: [cons(t0, init)],
  for i thru n do (
    r: errcatch (
      t1: domain[2]+i*dt,
      k1: apply(rkfunc,cons(t0,uvw)),
      k2: apply(rkfunc,cons((t0+t1)/2, uvw+k1*dt/2)),
```

```
      k3: apply(rkfunc,cons((t0+t1)/2,uvw+k2*dt/2)),
      k4: apply(rkfunc,cons(t1,uvw+k3*dt))),
    if length(r)=0 then return()
    else uvw: uvw + dt*(k1+2*k2+2*k3+k4)/6,
    t0: t1,
    rksoln : cons(cons(t0,uvw), rksoln)),
  reverse(rksoln))$
```

4.1.1 Five Examples of Maxima rk4

Example 1

We use **rk4** here with the same large step size $h = 0.5$, and the same first order o.d.e. $dy/dx = 8.5 - 20\,x + 12\,x^2 - 2\,x^3$, with the initial condition $y(0) = 1$, as we used in Sec. (2) where we used the Euler method. In that earlier section we also derived the exact solution for comparison with the approximate numerical solution, getting $-x^4/2 + 4\,x^3 - 10\,x^2 + 17\,x/2 + 1$.

```
(%i1) load(myode);
(%o1) "c:/k2/myode.mac"
(%i2) pts : rk4(8.5 - 20*x + 12*x^2 - 2*x^3,y,1,[x,0,4,0.5])$
(%i3) fll(pts);
(%o3) [[0.0,1.0],[4.0,3.0],9]
(%i4) pts;
(%o4) [[0.0,1.0],[0.5,3.21875],[1.0,3.0],[1.5,2.21875],[2.0,2.0],
       [2.5,2.71875],[3.0,4.0],[3.5,4.71875],[4.0,3.0]]
(%o5) plot2d([-x^4/2+4*x^3-10*x^2+17*x/2+1,[discrete,pts],[discrete,pts] ],
             [x,0,4], [style,[lines,3],[lines,3],
             [points,3,1,1] ],[legend,"exact","rk4",""],
             [gnuplot_preamble, "set key bottom right;set grid"])$
```

which produces

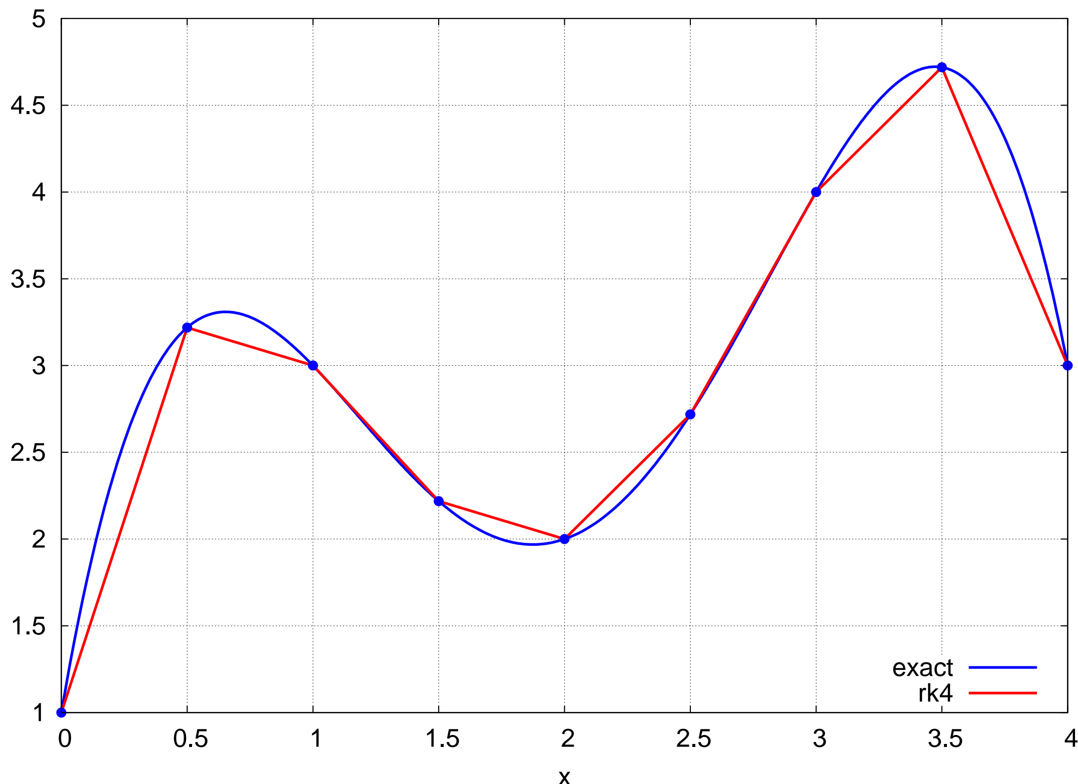

Figure 13: rk4 Truncation Error with $h = 0.5$

Despite the large step size, the fourth order Runge-Kutta method sticks quite close to the analytic solution, and the value of $y_{rk4}(4) = y_{exact}(4)$.

Example 2

A second example is the first order o.d.e. $dy/dx = -x\,y$, with the initial condition $y(0) = 1$, integrated over the interval **[x,0,3]**, with the analytic solution $y = e^{-\frac{1}{2}x^2}$.

```
(%i6) dydx : -x*y;
(%o6) -x*y
(%i7) ytrue : exp(-x^2/2);
(%o7) %e^-(x^2/2)
(%i8) hL : [0.5,0.2,0.1];
(%o8) [0.5,0.2,0.1]
(%i9) yerr_pts : yfdiff_rk4(dydx,ytrue,1,hL);
 tval =  0.606531
  0.5 -3.63124582E-5
  0.2 7.00094823E-7
  0.1 6.66862736E-8
(%o9) [[0.5,-3.63124582E-5],[0.2,7.00094823E-7],[0.1,6.66862736E-8]]
(%i10) case(h):= rk4(dydx,y,1,[x,0,3,h])$
(%i11) pts1 : case(0.5)$
(%i12) fll(pts1);
(%o12) [[0.0,1.0],[3.0,0.0130755],7]
(%i13) pts2 : case(0.2)$
(%i14) fll(pts2);
(%o14) [[0.0,1.0],[3.0,0.0111361],16]
(%i15) plot2d([ytrue,[discrete,pts1],[discrete,pts2] ],[x,0,3],
          [style,[lines,1]],[legend,"exact","h=0.5","h=0.2"],
                   [ylabel,"y"],[gnuplot_preamble,"set grid"])$
```

which produces the plot

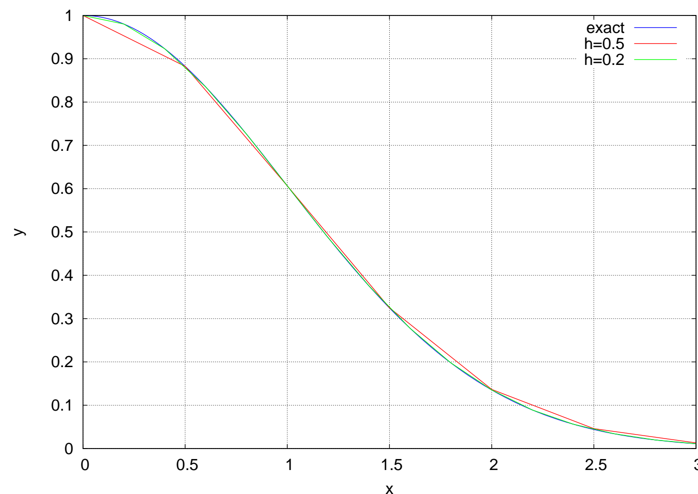

Figure 14: rk4 Convergence to $dy/dx = -x\,y$ with $y(0) = 1$

Example 3

We test **rk4** on the simple harmonic oscillator with unit period: $dx/dt = v_x$, $\quad dv_x/dt = -4\,\pi^2\,x$, with the initial conditions $x(0) = 1$, $\quad v_x(0) = 0$, and integrate over the time interval **[t,0,1]** for three different values of the time step **dt = h**. The analytic solution is $x = \cos(2\,\pi\,t)$.

We remind the reader that the list utility functions **fll** and **take** are in the chapter 2 file **k2util.mac** (as well as in **myode.mac**) and have the definitions

```
fll(x) := [first(x),last(x),length(x)]$

take(%aL,%nn) := (map(lambda([x],part(x,%nn)), %aL))$
```

```
(%i1) load(myode);
(%o1) "c:/k2/myode.mac"
```

```
(%i2) case(dt) := rk4([vx,-4*%pi^2*x],[x,vx],[1,0],[t,0,1,dt])$
(%i3) pts1 : case(0.1)$
(%i4) rpts1 : [discrete,take(pts1,1),take(pts1,2)]$
(%i5) fll(pts1);
(%o5) [[0.0,1.0,0.0],[1.0,0.99592,0.0440659],11]
(%i6) pts2 : case(0.05)$
(%i7) rpts2 : [discrete,take(pts2,1),take(pts2,2)]$
(%i8) fll(pts2);
(%o8) [[0.0,1.0,0.0],[1.0,0.999868,0.00309201],21]
(%i9) plot2d([cos(2*%pi*t),rpts1,rpts2],[t,0,1],[xlabel,"T"],[ylabel,"X"],
          [style,[lines,3]],[legend,"exact","h=0.1","h=0.05"],
          [gnuplot_preamble,"set key bottom right;set grid"])$
```

which produces the plot

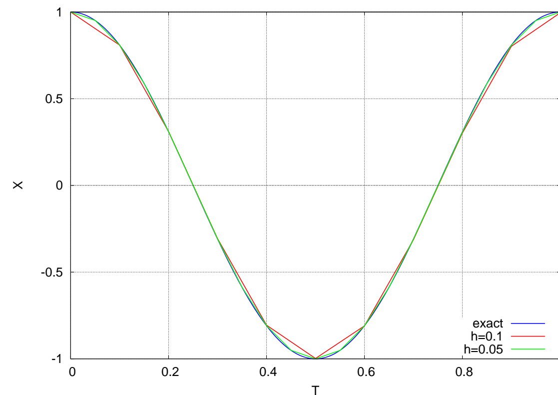

Figure 15: Exact and rk4 SHO Solutions

Example 4: Rigid Body Motion

The three Euler equations of motion for the angular velocity vector components $\omega_1, \omega_2, \omega_3$ of an unforced rigid body with principal moments of inertia I_1, I_2, I_3 are

$$\frac{d\,\omega_1}{d\,t} = \frac{I_2 - I_3}{I_1}\,\omega_2\,\omega_3 \tag{4.7}$$

$$\frac{d\,\omega_2}{d\,t} = \frac{I_3 - I_1}{I_2}\,\omega_3\,\omega_1 \tag{4.8}$$

$$\frac{d\,\omega_3}{d\,t} = \frac{I_1 - I_2}{I_3}\,\omega_1\,\omega_2 \tag{4.9}$$

We seek a solution for which $I_1 = 0.5, \quad I_2 = 2, \quad I_3 = 3$, and the initial conditions are $\omega_1(0) = 1, \quad \omega_2(0) = 0, \quad \omega_3(0) = 0.9$.

```
(%i1) load(myode);
(%o1) "c:/k2/myode.mac"
(%i2) (i1 : 0.5, i2 : 2, i3 : 3)$
(%i3) rsoln : rk4([(i2-i3)*w2*w3/i1, (i3-i1)*w3*w1/i2, (i1-i2)*w1*w2/i3],
              [w1,w2,w3],[1,0,0.9],[t,0,20,0.01])$
(%i4) fll(rsoln);
(%o4) [[0.0,1.0,0.0,0.9],[20.0,0.606204,0.628747,0.807385],2001]
(%i5) tL : take(rsoln,1)$
(%i6) fll(tL);
(%o6) [0.0,20.0,2001]
(%i7) (w1L : take(rsoln,2), w2L : take(rsoln,3), w3L : take(rsoln,4))$
(%i8) pts1 : [discrete,tL,w1L]$
(%i9) pts2 : [discrete,tL,w2L]$
(%i10) pts3 : [discrete,tL,w3L]$
(%i11) plot2d([ pts1,pts2,pts3], [y,-1.1,1.5],[style,[lines,3]],
         [xlabel,"t"],[ylabel,"Angular Velocity Components"],
         [legend, "w1","w2","w3"])$
```

which produces the plot

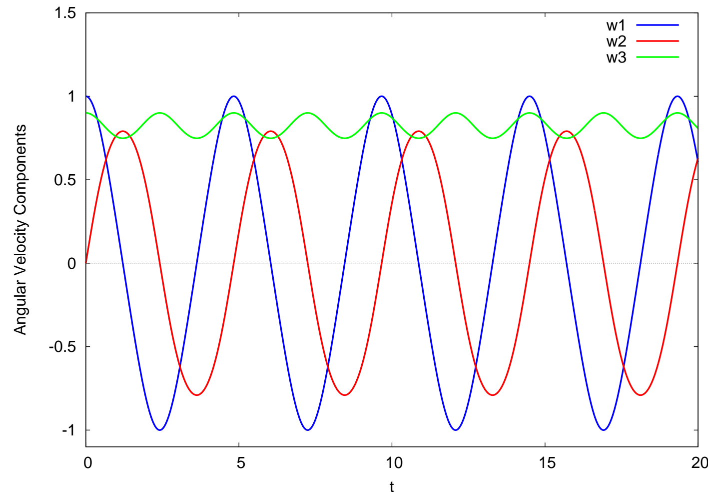

Figure 16: Unforced Rigid Body Angular Velocity Vector Components

Example 5: The Lorenz Equations

An example of a model with three o.d.e.'s is the Lorenz model

$$\frac{dx}{dt} = a\,x + y\,z \tag{4.10}$$

$$\frac{dy}{dt} = b\,(y - z) \tag{4.11}$$

$$\frac{dz}{dt} = -x\,y + c\,y - z \tag{4.12}$$

which we solve with our Maxima code **rk4** assuming the initial conditions are $x(0) = 1, \quad y(0) = 1, \quad z(0) = 1$, and the parameters have the values $a = -8/3, \quad b = -10$, and $c = 28$. We use a homemade function **range(list)**, which is included in **myode.mac** and in **k2util.mac** and which has the definition

```
range(aaL) := print(" min = ",lmin(aaL),"  max = ", lmax(aaL))$
```

```
(%i1) load(myode);
(%o1) "c:/k2/myode.mac"
(%i2) (a : -8/3, b : -10, c : 28)$
(%i3) ode : [a*x + y*z, b*(y-z), -x*y + c*y -z]$
(%i4) var : [x,y,z]$
(%i5) init : [1,1,1]$
(%i6) domain : [t,0,1,0.01]$
(%i7) rksoln : rk4(ode,var,init,domain)$
(%i8) tL : take(rksoln,1)$
(%i9) fll(tL);
(%o9) [0.0,1.0,101]
(%i10) xL : take(rksoln,2)$
(%i11) range(xL);
 min =   0.961737    max =   47.833954
(%o11) 47.833954
(%i12) yL : take(rksoln,3)$
(%i13) range(yL)$
 min =   -9.7615215    max =   19.555041
(%i14) zL : take(rksoln,4)$
(%i15) range(zL)$
 min =   -10.394135    max =   27.183473
(%i16) xpts : [discrete, tL, xL]$
(%i17) ypts : [discrete, tL, yL]$
(%i18) zpts : [discrete, tL, zL]$
```

```
(%i19) plot2d([xpts,ypts,zpts],[x,0,1],
       [style,[lines,3]], [xlabel,"t"],[ylabel,""],
       [legend,"x","y","z"],
       [gnuplot_preamble,"set key top left;set grid"])$
```

which produces the plot

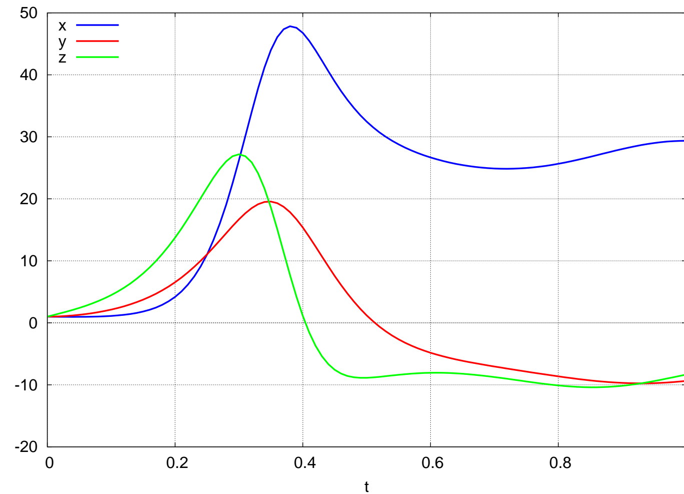

Figure 17: Maxima myrk4 with Lorenz Equations

We also plot $y(x)$:

```
(%i20) plot2d([discrete,xL,yL],[style,[lines,3]],[xlabel,"x"],
              [ylabel,"y"],[gnuplot_preamble,"set grid"])$
```

which produces the plot

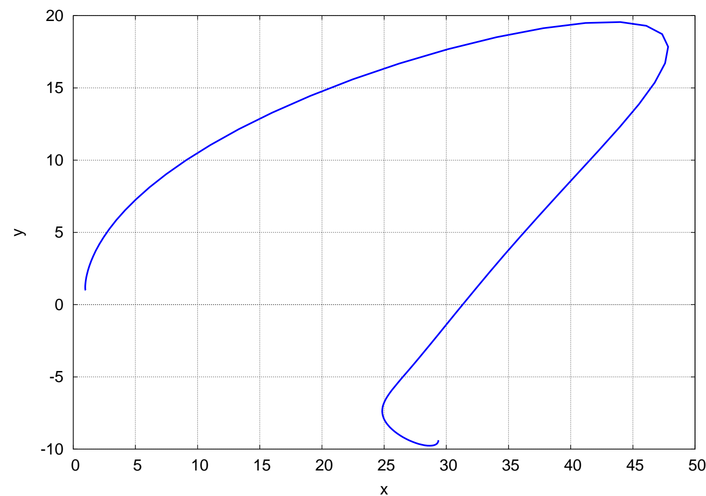

Figure 18: $y(x)$ for Lorenz Equations

As practice, we write the nested list **rksoln** as a four column data file using **write_data**.

```
(%i21) write_data(rksoln,"c:/k2/mydata.txt");
(%o21) done
```

If you look at the file **mydata.txt** with a text editor, the top of the file is

```
0.0 1.0 1.0 1.0
0.01 0.984891 1.0125672 1.2599178
0.02 0.973114 1.0488237 1.5239971
0.03 0.965159 1.1072089 1.7983099
0.04 0.961737 1.186868 2.0885401
0.05 0.963806 1.2875571 2.4001545
```

which displays the four columns of space separated data which represents the output of **rk4** for this problem.

To read such a space separated text data file into your Maxima session, you use **read_nested_list**, which returns a nested list identical to the output of **rk4**.

```
(%i22) mysoln : read_nested_list("c:/k2/mydata.txt")$
(%i23) fll(mysoln);
(%o23) [[0.0,1.0,1.0,1.0],[1.0,29.362404,-9.3786158,-8.35706],101]
(%i24) fll(rksoln);
(%o24) [[0.0,1.0,1.0,1.0],[1.0,29.362404,-9.3786158,-8.35706],101]
```

4.1.2 Failure of Maxima rk4 for a Stiff O.D.E.

The van der Pol equation

$$z'' - \mu\,(1 - z^2)\,z' + z = 0 \tag{4.13}$$

describes a non-conservative oscillator with non-linear damping and was originally designed as a model for electric circuits using vacuum tubes. The solution $z(t)$, for large μ, changes slowly with t over a region, and then changes very rapidly in the next region, with the solution approaching a distorted square wave for large μ.

We let $y_1 = z$ and $y_2 = z'$. Then $y_1' = z' = y_2$, and $y_2' = z'' = \mu\,(1 - y_1^2)\,y_2 - y_1$. We solve this pair of o.d.e.'s first for the "non-stiff" case, using $\mu = 1$, with no problems.

```
(%i1) load(myode);
(%o1) "c:/k2/myode.mac"
(%i2) nonstiff : rk4([y2, y2*(1-y1^2) - y1],[y1,y2],[2,0],[t,0,30,0.01])$
(%i3) fll(nonstiff);
(%o3) [[0.0,2.0,0.0],[30.0,-2.0079102,0.0519626],3001]
(%i4) tL :  take(nonstiff,1)$
(%i5) fll(tL);
(%o5) [0.0,30.0,3001]
(%i6) zL : take(nonstiff,2)$
(%i7) fll(zL);
(%o7) [2.0,-2.0079102,3001]
(%i8) plot2d([discrete,tL,zL],[style,[lines,3]],[ylabel,"z"],[xlabel,"t"])$
```

which produces the plot

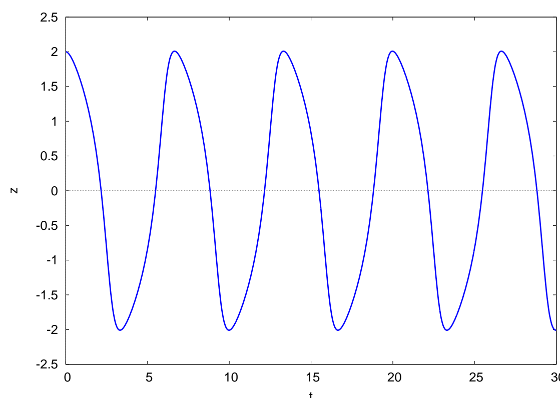

Figure 19: Non-stiff van der Pol Case

However, trying to use **rk4** with $\mu = 1000$, leads to a floating point overflow error return.

```
(%i9) stiff : rk4([y2, 1000*y2*(1-y1^2) - y1],[y1,y2],[2,0],[t,0,30,0.01])$
EXPT: floating point overflow.
```

For this type of problem, the **deSolve R** package of solvers is needed.

By the way, if you try the same stiff case with the current Maxima method **rk**, you will see **many** screens of an identical error message:

```
(%i10) stiff : rk([y2, 1000*y2*(1-y1^2) - y1],[y1,y2],[2,0],[t,0,30,0.01])$
EXPT: floating point overflow.
EXPT: floating point overflow.
EXPT: floating point overflow.
EXPT: floating point overflow.
EXPT: floating point overflow.
etc., etc.
```

The adjustable step contributed Maxima method **rkf45** can be used to solve moderately stiff initial value problems, although it is not designed for that purpose. We will discuss more about the syntax of **rkf45** in a later section. The main difference (compared with **rk**) is that instead of specifying the step size, you omit the step size entirely. We try **rkf45** here to see that it gives at least a graceful exit.

```
(%i11) load(rkf45);
(%o11) "C:/PROGRA~1/MAXIMA~3.2/share/maxima/5.31.2/share/contrib/rkf45/rkf45.mac"
(%i12) stiff : rkf45([y2, 1000*y2*(1-y1^2) - y1],[y1,y2],[2,0],[t,0,30])$
Warning: rkf45: Integration stopped at x = 8.7296685 (stiff problem?)
                Iterations limit has been reached. Check if differential
                equations/initial conditions are given correctly, reduce
                accuracy, and/or increase maximum number of steps.
```

4.2 R Code: myrk4

If you have already loaded in the **R** package **deSolve**, that package contains a method called **rk4**. In order to prevent confusion with that method (which requires a different **func** template: see our later discussion), we call **our** code **myrk4**.

The **R** code for **myrk4** is in **myode.R**.

```
##  myrk4: each element of solnList is a vector
##  which contains the grid values
##  of one of the dependent variables.

myrk4 = function(init, grid ,func)  {
   num.var = length(init)
   solnList = list()
   n.grid = length(grid)
   for (k in 1:num.var) {
        solnList[[k]] = vector(length = n.grid)
        solnList[[k]][1] = init[k] }
   h = grid[2] - grid[1]   # step size
   yL = vector(length = num.var)  # solution at beginning of each step
   for (j in 1:(n.grid-1)) {
      for (k in 1:num.var) yL[k] = solnList[[k]][j]
      k1 = func(grid[j], yL)  #  vector of derivatives
      k2 = func(grid[j] + h/2, yL + h*k1/2)  #  vector of derivatives
      k3 = func(grid[j] + h/2, yL + h*k2/2)  #  vector of derivatives
      k4 = func(grid[j] + h, yL + h*k3)  #  vector of derivatives
      for (k in 1:num.var) solnList[[k]][j+1] = solnList[[k]][j] +
                              h*(k1[k] + 2*k2[k] + 2*k3[k] + k4[k])/6 }
   if (num.var==1) solnList[[1]] else solnList}
```

The syntax of the required external function **func** to compute the needed derivatives is the same as in the case of the **R** function **myeuler** above.

4.2.1 Three Examples of R myrk4

Example 1

Here we use **myrk4** and **myeuler** to compare the solution of the single o.d.e. $dy/dt = -t\,y$ for $y(0) = 1$, after loading in **myode.R**.

```
> source("myode.R")
> tL = seq(0,3,0.1)
> fll(tL)
  0   3   31
> deriv = function(t,y) { -t*y }
> yL = myrk4(1,tL,deriv)
> fll(yL)
  1   0.01111038   31
> yL.euler = myeuler(1,tL,deriv)
> fll(yL.euler)
  1   0.007791097   31
> plot(tL,yL,type="l",lwd=3,col="red",xlab="t",ylab="y")
> abline(h=0,v=0)
> grid()
> lines(tL,yL.euler,lwd=3,col="green")
> legend("topright", col=c("red","green"),lwd=3,
+           legend=c("myrk4","euler"))
```

which produces the plot

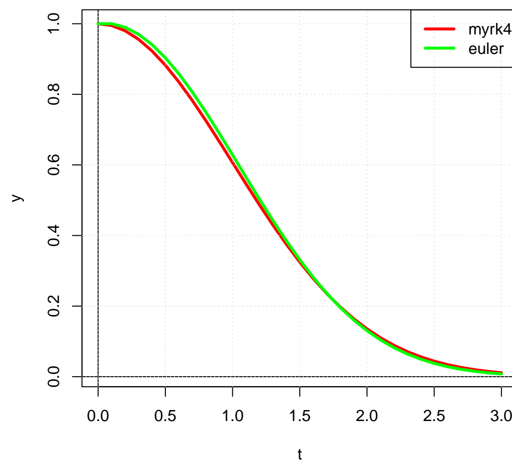

Figure 20: myrk4 and myeuler for $h = 0.1$

Example 2

We next test **myrk4** on the simple harmonic oscillator with unit period: $dx/dt = v_x, \quad dv_x/dt = -4\,\pi^2\,x$, with the initial conditions $x(0) = 1, \quad v_x(0) = 0$, and compare with the performance of **myeuler**. The analytic solution is $x = \cos(2\,\pi\,t)$.

```
> tL = seq(0,1,0.001)
> fll(tL)
  0   1   1001
> sho = function(t,y) with(as.list(y), c(y[2], -4*pi^2*y[1]) )
> out = myrk4(c(1,0),tL,sho)
> xL = out[[1]]
> fll(xL)
  1   1   1001
> vxL = out[[2]]
> fll(vxL)
  0   5.127315e-10   1001
```

```
> max(vxL)
[1] 6.283185
> out.euler = myeuler(c(1,0),tL,sho)
> xL.euler = out.euler[[1]]
> fll(xL.euler)
  1   1.019935   1001
> vxL.euler = out.euler[[2]]
> fll(vxL.euler)
  0   0.0005298591   1001
> plot(tL,xL,type="l",lwd=2,col="red",xlab="t",ylab="x")
> lines(tL,xL.euler,lwd=2,col="green")
> legend("top", col=c("red","green"),lwd=3,
+           legend=c("myrk4","myeuler"))
```

which produces the plot

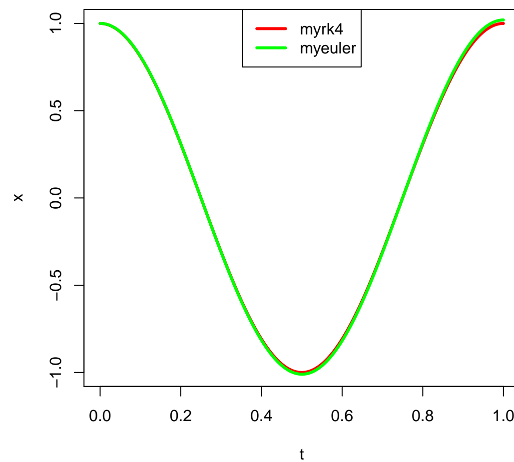

Figure 21: SHO: myrk4 and myeuler for $h = 0.001$

Example 3: The Lorenz Model with R

An example of a model with three o.d.e.'s is the Lorenz model

$$\frac{d\,x}{d\,t} = a\,x + y\,z \tag{4.14}$$

$$\frac{d\,y}{d\,t} = b\,(y - z) \tag{4.15}$$

$$\frac{d\,z}{d\,t} = -x\,y + c\,y - z \tag{4.16}$$

which we solve with our **R** code **myrk4** assuming the initial conditions are $x(0) = 1, \quad y(0) = 1, \quad z(0) = 1$, and the parameters have the values $a = -8/3, \quad b = -10$, and $c = 28$.

```
> source("myode.R")
> lorenz = function(t,y) {
+      with( as.list(y), {
+         dx = a*y[1] + y[2]*y[3]
+         dy = b*(y[2] - y[3])
+         dz = -y[1]*y[2] + c*y[2] - y[3]
+         c(dx,dy,dz)})}
> tL = seq(0,1,0.01)
> a = -8/3 ; b = -10; c = 28
> yini = c(1,1,1)
```

```
> out = myrk4(yini,tL,lorenz)
> xL = out[[1]]
> range(xL)
[1]  0.9617372 47.8339541
> yL = out[[2]]
> range(yL)
[1] -9.761521 19.555041
> zL = out[[3]]
> range(zL)
[1] -10.39413  27.18347
> plot(tL,xL,type="l",lwd=3,col="blue",ylim=c(-12,50),xlab="t",ylab="")
> lines(tL,yL,lwd=3,col="red")
> lines(tL,zL,lwd=3,col="green")
> grid(lty="solid", col="darkgray")
> legend("topright",col= c("blue","red","green"),lwd=3,
+           legend = c("x","y","z"))
```

which produces the plot

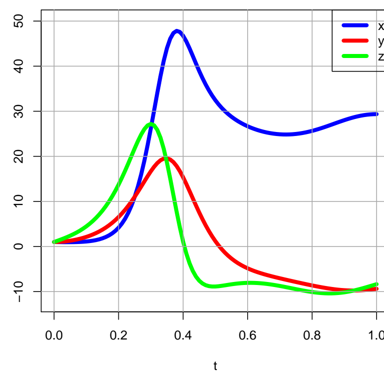

Figure 22: Lorenz Model with our R code myrk4

We also make of plot of $y(x)$:

```
> plot(xL,yL,type = "l",lwd=3,col="blue",xlab="x",ylab="y")
> grid(lty="solid",col="darkgray")
```

which produces

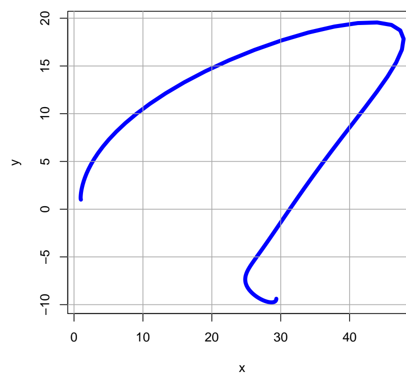

Figure 23: Lorenz Model: $y(x)$

4.2.2 Failure of R myrk4 for a Stiff O.D.E.

We again integrate the van der Pol equation (see Sec. (4.1.2)). First a non-stiff case with $\mu = 1$.

```
> vanderPol = function(t,y){
+     with( as.list(y), c(y[2], mu*y[2]*(1-y[1]^2) - y[1]))}
> tL = seq(0,30,0.01)
> fll(tL)
  0   30   3001
> yini = c(2,0)
> mu = 1
> nonstiff = myrk4(yini,tL,vanderPol)
> zL = nonstiff[[1]]
> fll(zL)
  2   -2.00791   3001
> plot(tL,zL,type="l",lwd=3,col="blue",xlab="t",ylab="z")
```

which produces

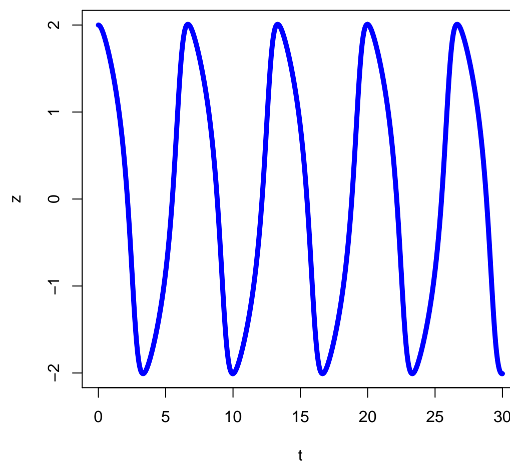

Figure 24: Non-Stiff van der Pol case $\mu = 1$

Next we try a stiff case, with $\mu = 1000$, and the same sequence of times and the same initial conditions.

```
> mu = 1000
> stiff = myrk4(yini,tL,vanderPol)
> zL = stiff[[1]]
> any(is.nan(zL))
[1] TRUE
> all(is.nan(zL))
[1] FALSE
> fll(zL)
  2   NaN   3001
> head(zL)
[1] 2.000000e+00 1.993400e+00 6.460842e+01 8.833844e+42          NaN
[6]          NaN
> tail(zL)
[1] NaN NaN NaN NaN NaN NaN
```

All but the first four values of `zL` are `NaN` (not a number), which indicates a **failure** of `myrk4` in dealing with this stiff case.

We have used `any(is.nan(vec))`, which returns `TRUE` if at least one element of `vec` is `NaN`.

```
> xL = c(1,2,3,4,NaN,5,NaN)
> yL = c(1,2,3,4,5)
> any(is.nan(xL))
[1] TRUE
> any(is.nan(yL))
[1] FALSE
```

5 The Standard and Contributed Maxima Methods

5.1 rk

The standard fixed step fourth order Runge-Kutta method is **rk**, which has the same syntax as our Maxima code **rk4**:

```
    if the dxdt expression is a function of (t,x), then:
       for one o.d.e: rk(dxdt,x,xinit,[t,tinit,tfinal,dt])

        If the dx1dt expression and the dx2dt expression are functions of (t,x1,x2),
    then for two o.d.e.'s:

    rk([dx1dt,dx2dt],[x1,x2],[x1init,x2init],[t,tinit,tfinal,dt])
     and so on.
```

Nothing has to be loaded into your work session to use **rk**. Since **rk** is a fixed step method, with the step size chosen by the user, the value of the step size should be reduced at least once from its initial value, in order to assess the effects on the global solution. A rough rule of thumb is to start with a step size that is of the order of one hundredth of the integration interval.
In order to easily extract the list of times **tL**, and the list of the first dependent variable **y1L**, etc., from the list returned by **rk**, we recommend the use of our home-made list utility function **take**, which can be made available by loading in **k2util.mac** from our Chapter 2 files.

5.2 rkf45

This is a contributed method, located in **...share\contrib\rkf45**, where you can find **rkf45.mac** (which contains the needed code), **rkf45.pdf**, **rkf45.dem**, as well as a test file.

The version of **rkf45** in Maxima ver. 5.31 has a number of bugs which will be fixed in later versions. The chapter 2 file **myrk4.mac** contains the bug fixes.

The author of this method has a web page:
https://sites.google.com/site/pjpapasot/maxima/libraries/rkf45.

Fehlberg discovered a 5-th order method that only requires 6 evaluations while the same combination of evaluations but with different factors yields a 4-th order scheme. Therefore this approach allows an error estimation (by comparing the two cases at each step) at a much reduced computational cost (with 6 evaluations only). By making such an error estimate for each step, the step size can be adjusted to keep the estimated errors small.

Depending upon the computational effort required to compute the function evaluations, this method can produce a significant decrease in computational effort.

We include part of the top section of **rkf45.mac** here as a guide to getting started with this method (with some very light editing). We have replaced the symbol **func** with the symbol **var** in the following.

```
Author: Panagiotis J. Papasotiriou
--------------------------------------------------------------------------------
Brief description:

rkf45 is a Maxima function for solving initial value problems with automatic
step size and error control.
This is an implementation of the Runge-Kutta-Fehlberg 4th-5th-order scheme.
--------------------------------------------------------------------------------
Syntax:

rkf45(ode,var,init,interval,options)
rkf45([ode1,ode2,...],[var1,var2,...],[init1,init2,...],interval,options)

The first form solves a first-order differential equation, (o.d.e.), with
respect to the initial condition init, where var is the dependent variable
and init is the value of the dependent variable at the initial point.
```

```
Similarly, the second form solves a system of first-order differential
equations, ode1,ode2,..., with respect to the initial conditions
init1,init2,..., where var1, var2,... are the dependent variables and
init1,init2,... are the values of the dependent variables at the initial point.

Differential equation(s) should be given as expressions depending only on the
independent and dependent variables, and should define the derivative of the
dependent variable with respect to the independent variable. For instance, the
differential equation y'(x)+(x+1)*y=0 should be given as -(x+1)*y.

The argument "interval" should be a list of three elements. The first element
identifies the independent variable, while the second and third elements are
the initial and final values for the independent variable, as in [x,0,6].

The initial value does not need to be less than final value, so an interval
such as [x,6,0] is also valid.

rkf45 accepts the following optional arguments:

*      full_solution: A Boolean. If set to true, a full list of the solution at
                      all intermediate points will be returned. If set to false,
                      only the solution at the last integration point is
                      returned. Default: true.

* absolute_tolerance: The desired absolute tolerance of the result. Default:
                      1e-6.

*     max_iterations: Maximum number of iterations. Default: 10000.

*            h_start: Initial integration step. Default: one 100th of the
                      integration interval, (interval[3]-interval[2])/100.

*             report: A Boolean. If set to true, rkf45 prints a report at
                      exit, giving details about the calculations done.
                      Default: false.

The integration step size is selected automatically in such a way that the
estimated local error is less than user-specified absolute tolerance.

The result is returned as a list with n+1 columns, where n is the number of
first-order differential equations. The first column contains the values of
the independent variable selected by the algorithm. The second column
contains the values of the first dependent variable at the corresponding
value of the independent variable. Similarly, the third column contains the
values of the second dependent variable at the corresponding value of the
independent variable, and so on.

rkf45 can be used to solve moderately stiff initial value problems, although
it is not designed for that purpose.
--------------------------------------------------------------------------------
Examples:

(1) A first-order differential equation, y'=x*(y-1)+3, with y(0)=-2:
    rkf45(x*(y-1)+3,y,-2,[x,0,4]) returns the solution at selected points
        from x=0 to x=4.

(2) A second-order differential equation, y''=x+y*y', with y(-1)=2, y'(-1)=0:
    rkf45([y2,x+y1*y2],[y1,y2],[2,0],[x,-1,4]) returns the solution at selected
    points from x=-1 to x=4.
```

Example 1

Here is an example from the demo file **rkf45.dem** which finds the solution of the single o.d.e. $dy/dx = -3\,x\,y^2 + 1/(1+x^3)$ over the range **[x,0,5]** and with the initial condition $y(0) = 0$. The optional argument **report = true** is included. See comments by the author of **rkf45** (**Panagiotis J. Papasotiriou**) in his file **rkf45.pdf**. (To use the bugfree code version, you would use **load(myrkf45)** or **load("myrkf45.mac")**).

```
(%i1) fpprintprec:7$
(%i2) load(rkf45);
(%o2) "C:/PROGRA~1/MAXIMA~3.2/share/maxima/5.31.2/share/contrib/rkf45/rkf45.mac"
(%i3) rksoln : rkf45(-3*x*y^2+1/(x^3+1),y,0,[x,0,5],report=true)$
------------------------------------------------------
Info: rkf45:
   Integration points selected: 42
    Total number of iterations: 45
           Bad steps corrected: 4
       Minimum estimated error: 3.0488505E-10
       Maximum estimated error: 9.5960328E-7
Minimum integration step taken: 0.05
Maximum integration step taken: 0.31668
------------------------------------------------------
(%i4) plot2d([discrete,rksoln],[xlabel,"x"],[ylabel,"y"],
        [style,[lines,3]],[gnuplot_preamble,"set grid"])$
```

which produces the plot

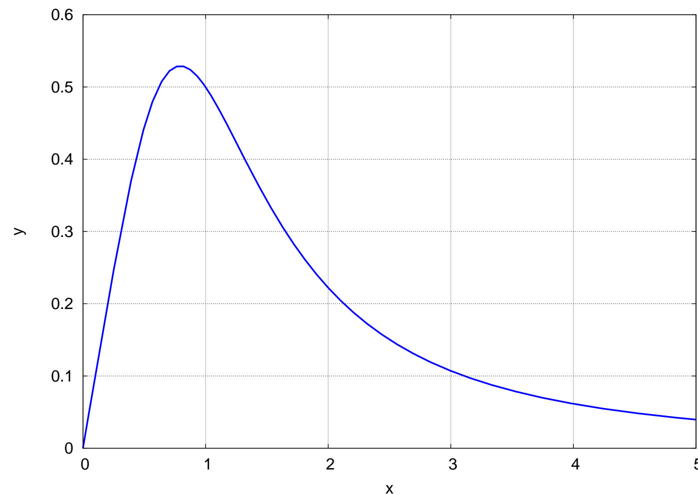

Figure 25: rkf45 Solution

We now repeat with a request for more accuracy, by using the optional **absolute_tolerance** flag.

```
(%i5) rksoln2 : rkf45(-3*x*y^2+1/(x^3+1),y,0,[x,0,5],
             absolute_tolerance = 1e-12,  report=true)$
------------------------------------------------------
Info: rkf45:
   Integration points selected: 1168
    Total number of iterations: 1173
           Bad steps corrected: 6
       Minimum estimated error: 4.8670316E-15
       Maximum estimated error: 9.4750206E-13
Minimum integration step taken: 0.0017906
Maximum integration step taken: 0.016874
------------------------------------------------------
(%i6) plot2d([[discrete,rksoln],[discrete,rksoln2]],
          [xlabel,"x"],[ylabel,"y"],
           [style,[lines,1]],[gnuplot_preamble,"set grid"],
           [legend,"1e-6","1e-12"])$
```

which produces the plot

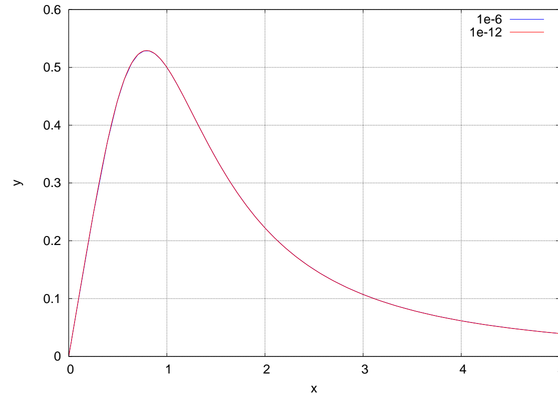

Figure 26: rkf45 Solutions

Example 2

Here is another example from the demo file, which solves a single o.d.e. for three values of a parameter called **s**. See comments by the author of **rkf45** (**Panagiotis J. Papasotiriou**) in his file **rkf45.pdf**.

The very useful Maxima function **makelist** is used here with the particular syntax **makelist(expr, p, pList)**, in which the parameter **p** takes on values in the list **pList**, and for each such **p**, **expr** is evaluated.

We first use **makelist** to form a list (**eqnL**) of symbolic expressions for dy/dt for three values of a parameter **s**. We then make a list (**solnL**) of numerical solutions for these three cases of **s** using **rkf45** with $y(0) = 0$, integrating over the time interval **[t,0,100]**, and accepting all the defaults. We then use the same syntax of **makelist** to form the main first argument of **plot2d**:
makelist([discrete,sv],sv,solnL), in which **sv** is a dummy argument which takes on successively the values in the list **solnL**.

```
(%i8) eqnL : makelist(s-1.51*y+3.03*y^2/(1+y^2),s,[0.206,0.204,0.202]);
(%o8) [3.03*y^2/(y^2+1)-1.51*y+0.206,3.03*y^2/(y^2+1)-1.51*y+0.204,
        3.03*y^2/(y^2+1)-1.51*y+0.202]
(%i9) solnL : makelist(rkf45(ode,y,0,[t,0,100]),ode,eqnL)$
(%i10) plot2d(makelist([discrete,sv],sv,solnL),[style,[lines,2]],[xlabel,"t"],
         [ylabel,"y"],[legend,"s=0.206","s=0.204","s=0.202"],
         [gnuplot_preamble,"set key left"])$
```

which produces the plot

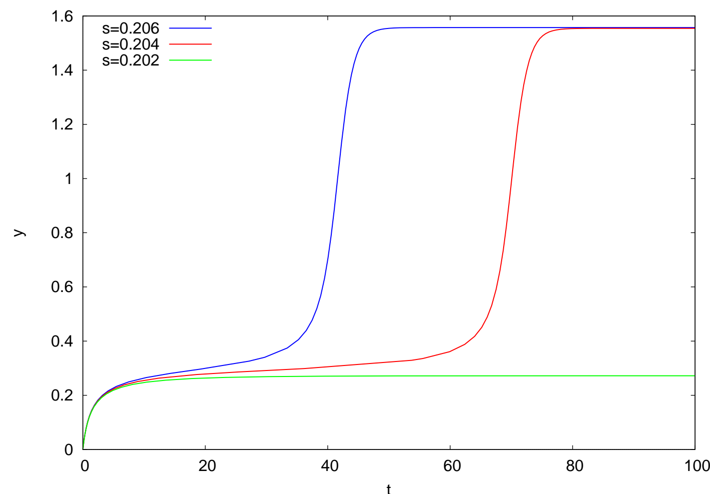

Figure 27: rkf45 Solutions for Different Values of s

Example 3

Here is an example of integrating the Lorenz model (see above in Sec. (4.2.1)). We need to load the contributed file **rk45f.mac**, and we also load our list utility file **k2util.mac**, which provides definitions of **fll**, **take**, and **range**.

```
(%i1) fpprintprec:7$
(%i2) load(rkf45);
(%o2) "C:/PROGRA~1/MAXIMA~3.2/share/maxima/5.31.2/share/contrib/rkf45/rkf45.mac"
(%i3) load(k2util);
(%o3) "c:/k2/k2util.mac"
(%i4) (a : -8/3, b : -10, c : 28)$
(%i5) ode : [a*x + y*z, b*(y-z), -x*y + c*y -z]$
(%i6) var : [x,y,z]$
(%i7) init : [1,1,1]$
(%i8) interval : [t,0,1]$
(%i9) rksoln : rkf45(ode,var,init,interval)$
(%i10) tL : take(rksoln,1)$
(%i13) fll(tL);
(%o13) [0,1.0,282]
(%i14) xL : take(rksoln,2)$
(%i15) range(xL)$
 min =  0.96172   max =  47.84057
(%i16) yL : take(rksoln,3)$
(%i17) range(yL)$
 min =  -9.761498   max =  19.56929
(%i18) zL : take(rksoln,4)$
(%i19) range(zL)$
 min =  -10.39593   max =  27.18298
(%i20) xpts : [discrete, tL, xL]$
(%i21) ypts : [discrete, tL, yL]$
(%i22) zpts : [discrete, tL, zL]$
(%i23) plot2d([xpts,ypts,zpts],
      [style,[lines,3]], [xlabel,"t"],[ylabel,""],
      [legend,"x","y","z"],
      [gnuplot_preamble,"set key top left;set grid"])$
```

which produces the plot

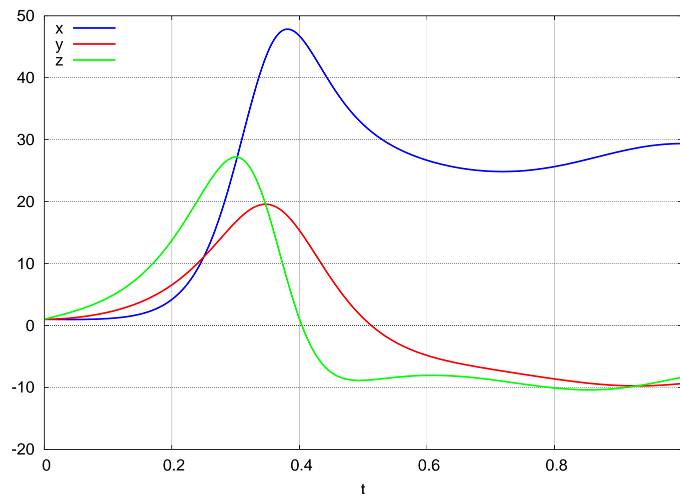

Figure 28: The Lorenz Model Using rkf45

This solution agrees with that we found earlier (Sec. (4.1.1)) using our Maxima code **rk4**, as well as the solution using either **R:myrk4** (Sec. (4.2.1)) or **R:ode** (see next section).

6 Standard R Methods using deSolve's ode

The `R` package **deSolve** contains a large number of numerical methods for the integration of systems of ordinary differential equations in the context of initial value problems.

The text **Solving Differential Equations in R**, by Karline Soetaert, Jeff Cash, and Francesca Mazzia, Springer-Verlag, 2012, presents many examples of the use of the solvers available in **deSolve**. In addition to conventional initial value problems, the authors present methods for the solution of differential algebraic equations, delay differential equations, partial differential equations, and boundary value problems.

New users of this package can just use the `ode` function, which has the syntax

```
ode (y, times,  func, parms, ...)
```

in which `y` is the initial value (or vector of initial values `c(y10, y20,...)` of the dependent variables, `times` is a vector of times at which output is requested (typically produced using the function `seq`), `func` is a `R` function which returns the rate of change (first derivatives) of the dependent variables, and must have a particular form, and `parms` contains the value(s) of parameters to be used in the integration.

If there are no parameters which are needed to completely define the derivatives returned by `func`, or if you wish to just use gobal assignments to set the values of the parameters, then you can use `parms=NULL` as the input to `ode`.

Once you have loaded **deSolve** using `library(deSolve)`, you can get the manual page for `ode` using `? ode`. Here is an edited version of the top of that page (note that this description of the syntax and methods assumes that the independent variable is time):

```
ode {deSolve}        R Documentation
General Solver for Ordinary Differential Equations
Description

Solves a system of ordinary differential equations;
    a wrapper around the implemented ODE solvers

Usage

ode(y, times, func, parms,
        method = c("lsoda", "lsode", "lsodes", "lsodar",
         "vode", "daspk", "euler", "rk4", "ode23", "ode45",
         "radau", "bdf", "bdf_d", "adams", "impAdams",
         "impAdams_d", "iteration"), ...)

Arguments

y: the initial (state) values for the ODE system, a vector. If y
   has a name attribute, the names will be used to label the output matrix.

times: time sequence for which output is wanted; the first value
    of times must be the initial time.

func: either an R-function that computes the values of the derivatives
    in the ODE system (the model definition) at time t, or a character
     string giving the name of a compiled function in a dynamically
     loaded shared library.

   If func is an R-function, it must be defined as:
         func = function(t, y, parms,...).
     t is the current time point in the integration.
     y is the current estimate of the variables in the ODE system.
       If the initial values y has a names attribute, the names
       will be available inside func.
     parms is a vector or list of parameters;
     ... (optional) are any other arguments passed to the function.
```

```
    The return value of func should be a list, whose first element
     is a vector containing the derivatives of y with respect to
     time, and whose next elements are global values that are required
     at each point in times. The derivatives must be specified in
     the same order as the state variables y.

    If func is a string, then dllname must give the name of the shared
     library (without extension) which must be loaded before ode is
     called. See package vignette "compiledCode" for more details.

parms:         parameters passed to func.

method: (optional: the default method is lsoda)
        the integrator to use, either a function that performs integration,
         or a list of class rkMethod, or a string: "lsoda", "lsode",
         "lsodes","lsodar","vode", "daspk", "euler", "rk4", "ode23",
         "ode45", "radau", "bdf", "bdf_d", "adams", "impAdams" or
         "impAdams_d" ,"iteration".
         Options "bdf", "bdf_d", "adams", "impAdams" or "impAdams_d" are
         the backward differentiation formula, the BDF with diagonal
         representation of the Jacobian, the (explicit) Adams and the
         implicit Adams method, and the implicit Adams method with
         diagonal representation of the Jacobian respectively
         (see details). The default integrator used is lsoda.

         (The method "iteration" is special in that here the function
         func should return the new value of the state variables rather
         than the rate of change. This can be used for individual
         based models, for difference equations, or in those cases
         where the integration is performed within func). See last example.

...         additional arguments passed to the integrator or to the methods.

Details

This is simply a wrapper around the various ode solvers.

The default integrator used is lsoda.

Value:  A matrix of class deSolve with up to as many rows as elements
      in times and as many columns as elements in y plus the number
      of "global" values returned in the second element of the
      return from func, plus an additional column (the first)
      for the time value. There will be one row for each element in
      times unless the integrator returns with an unrecoverable error.
      If y has a names attribute, it will be used to label the columns
      of the output value.

Author: Karline Soetaert <karline.soetaert@nioz.nl>
```

The default `lsoda` method used by the wrapper `ode` always starts with the non-stiff *explicit* multi-step Adams method, and when stiffness is detected, switches to an *implicit* multistep solver ("bdf": backward differentiation formula).

6.1 One First Order O.D.E.: Solving $dy/dx = -x\,y$ with $y(0) = 1$

We will use the simple o.d.e. $dy/dx = -x\,y$ with $y(0) = 1$ to explore the behavior of `ode`. We know that, by default, `ode` calls the independent variable a time t, so we have to adapt our approach to `ode`'s behavior. We will see that we have much freedom in defining the names of the derivative function, and the names of its three formal arguments. We can override column labeling on the output matrix which `ode` returns by using `colnames`. We can also force our intended labels and title on the final plot produced.

We start with defining the needed derivatives `func` containing the three required arguments, accepting `t` as the independent variable for now. Note that, crucially, the derivative function returns a list.

```
> library(deSolve)
> deriv = function(t, y, parms) list( -t*y)
> tL = seq(0,3,0.01)
> yini = 1
> out = ode(y=yini, times=tL, func=deriv, parms=NULL)
> head(out)
     time         1
[1,] 0.00 1.0000000
[2,] 0.01 0.9999500
[3,] 0.02 0.9998000
[4,] 0.03 0.9995501
[5,] 0.04 0.9992003
[6,] 0.05 0.9987508
> plot(out)
```

which produces the basic default plot of the output (matrix)

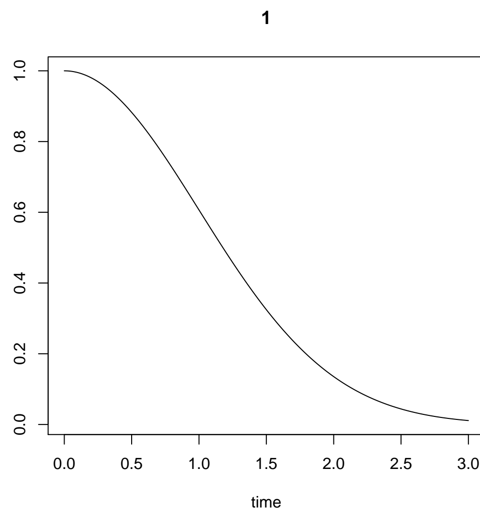

Figure 29: basic ode solution: $dy/dt = -t\,y$ for $y(0) = 1$

In the output of the **head** command, we see the first column has the label "times" (which we can change using **colnames**), and the second column has the label "1". In the simple default plot produced by **plot(out)** we get the "1" again as the plot title.

We can extract the vector **yL** of values of **y(t)** from the matrix **out** returned by **ode** using **yL = out[,2]**, which extracts the second column of the matrix **out**.

Our next approach is to change the definition of **deriv** to **deriv = function(x,y,parms) ...**, with **x** playing the role of the independent variable, and defining a sequence of **x** values using **xL = seq(...)**. We also give **yini** the "name attribute" by supplying the name **y** in the form **yini = c(y = 1)**.

```
> deriv = function(x, y, parms) list( -x*y)
> yini = c(y = 1)
> xL = seq(0,3,0.01)
> out = ode(yini, xL, func=deriv, parms=NULL)
> head(out)
     time         y
[1,] 0.00 1.0000000
[2,] 0.01 0.99995s00
[3,] 0.02 0.9998000
[4,] 0.03 0.9995501
[5,] 0.04 0.9992003
[6,] 0.05 0.9987508
> plot(out)
```

which produces the plot

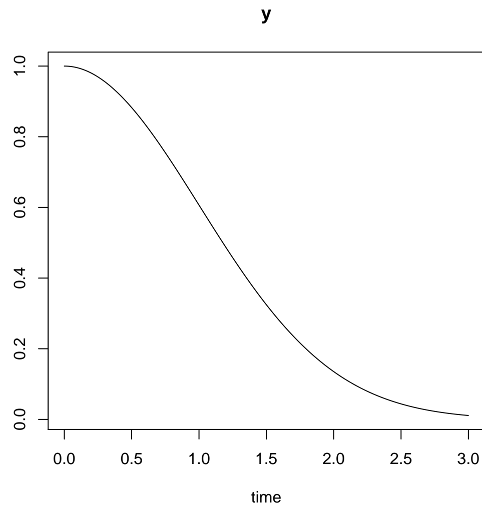

Figure 30: trying ode solution: $dy/dx = -x\,y$ for $y(0) = 1$

We can add some optional arguments to get rid of the "time" label on the x-axis, and also dress up the plot with color and extra line thickness and a grid:

```
> plot(out,lwd=3,col="blue",xlab="x",ylab="y")
> grid(lty="solid",col="darkgray")
```

which produces

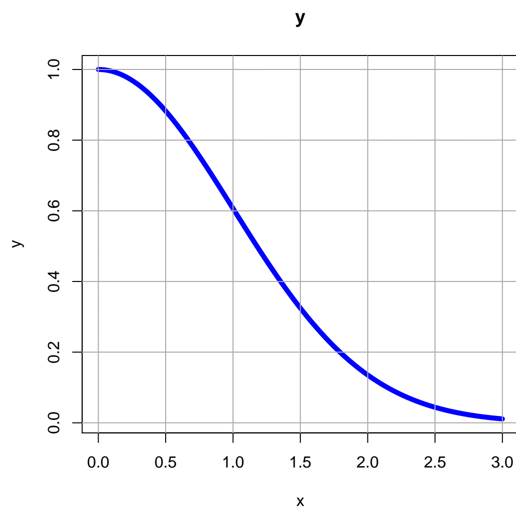

Figure 31: better ode solution: $dy/dx = -x\,y$ for $y(0) = 1$

Perhaps an easier way to override the "times" label is to use **`colnames`**.

```
> colnames(out)
[1] "time" "y"
> colnames(out) = c("x","y")
> colnames(out)
[1] "x" "y"
```

```
> head(out)
        x         y
[1,] 0.00 1.0000000
[2,] 0.01 0.9999500
[3,] 0.02 0.9998000
[4,] 0.03 0.9995501
[5,] 0.04 0.9992003
[6,] 0.05 0.9987508
> plot(out)
```

which gets a basic plot with the correct labels:

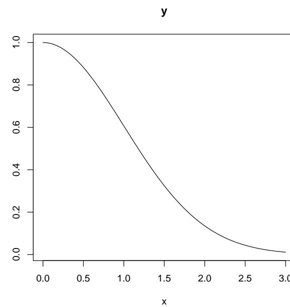

Figure 32: colnames cure for ode solution: $dy/dx = -x\,y$ for $y(0) = 1$

We can now modify the notation of **deriv** to use **ode** to solve the o.d.e. $dz/dx = -x\,z$ with $z(0) = 1$.

```
> deriv = function(x, z, parms) list( -x*z)
> zini = c(z = 1)
> xL = seq(0,3,0.01)
> out = ode(zini, xL, func=deriv, parms=NULL)
> colnames(out)
[1] "time" "z"
> colnames(out) = c("x","z")
> head(out)
        x         z
[1,] 0.00 1.0000000
[2,] 0.01 0.9999500
[3,] 0.02 0.9998000
[4,] 0.03 0.9995501
[5,] 0.04 0.9992003
[6,] 0.05 0.9987508
> plot(out)
```

which produces the default plot with the correct title and axis label:

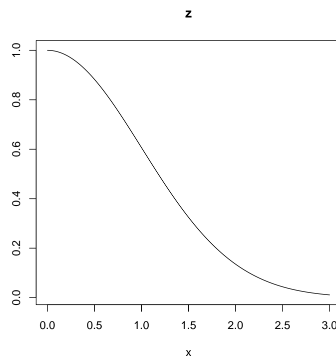

Figure 33: basic ode solution: $dz/dx = -x\,z$ for $z(0) = 1$

We can use a different name for the third argument of **deriv** here, say, **p**, and still get the proper solution.

```
> deriv = function(x, z, p) list( -x*z)
> out = ode(zini, xL, func=deriv, parms=NULL)
> colnames(out)
[1] "time" "z"
> colnames(out) = c("x","z")
> head(out)
        x         z
[1,] 0.00 1.0000000
[2,] 0.01 0.9999500
[3,] 0.02 0.9998000
[4,] 0.03 0.9995501
[5,] 0.04 0.9992003
[6,] 0.05 0.9987508
```

As long as we leave the args to **ode** in the standard order, we don't need to use **func=deriv**, but just **deriv**, for example.

```
> out = ode(zini, xL, deriv, NULL)
> colnames(out) = c("x","z")
> head(out)
        x         z
[1,] 0.00 1.0000000
[2,] 0.01 0.9999500
[3,] 0.02 0.9998000
[4,] 0.03 0.9995501
[5,] 0.04 0.9992003
[6,] 0.05 0.9987508
```

We can use any other name for the **func** derivatives function.

```
> dzdx = function(x, z, p) list( -x*z)
> out = ode(zini, xL, dzdx, NULL)
> colnames(out) = c("x","z")
> head(out)
        x         z
[1,] 0.00 1.0000000
[2,] 0.01 0.9999500
[3,] 0.02 0.9998000
[4,] 0.03 0.9995501
[5,] 0.04 0.9992003
[6,] 0.05 0.9987508
```

6.2 Two First Order O.D.E.'s: Using the Parameters Argument

We discuss an instructive example adapted from page 73 (and following) in the pdf document "Using R for Scientific Computing", by Karline Soetaert and Filip Meysman, Centre for Estuarine and Marine Ecology, Netherlands Institute of Ecology, The Netherlands, February 2011.

This pdf document can be found in zip format at **http://cran.r-project.org/other-docs.html**.

Assume the pair of first order o.d.e.s $dA/dt = r\,(x - A) - k\,A\,B$ and $dB/dt = r\,(y - B) + k\,A\,B$, determine the time evolution of concentrations $A(t)$ and $B(t)$, subject to the values of the fixed parameters r, x, k, y, and given initial values of A and B.

A possible form for the needed derivatives function is

```
derivs = function(t, y, p) {
   with (as.list(c(y, p)), {
      dA = r*(x-A) - k*A*B
      dB = r*(y-B) + k*A*B
      list(c(dA, dB)) }) }
```

Note that the symbol **y** appears both as the formal second argument to **derivs**, and also as a parameter in calculating the derivatives. This looks dangerous, but will work fine as long as we pass the value of the parameter **y** via the third argument **p**. A more cautious approach would be to use **derivs = function(t,state,parameters) {...}**.

The R-statement `with( as.list (c(y,p)), ...`, together with supplying `yini` with the "names attribute", will allow the state variables and parameters to be addressed by their names. Again, note carefully that a list must be returned by the derivatives function, with the elements of the enclosed vector in the same order as given in `yini`.

```
> library(deSolve)
> derivs = function(t, y, p) {
+    with (as.list(c(y, p)), {
+       dA = r*(x-A) - k*A*B
+       dB = r*(y-B) + k*A*B
+       list(c(dA, dB)) }) }
> params = c(x = 1, y = 0.1, k = 0.05, r = 0.05)
> yini =  c(A = 1, B = 1)
> yini
A B
1 1
> tL =  seq(0,300,1)
> out = ode(yini, tL, func = derivs, parms = params)
> head(out)
     time         A         B
[1,]    0 1.0000000 1.0000000
[2,]    1 0.9523189 1.0037869
[3,]    2 0.9090687 1.0052854
[4,]    3 0.8699226 1.0047151
[5,]    4 0.8345728 1.0022854
[6,]    5 0.8027203 0.9982009
```

Times and concentration vectors can be defined using matrix notation, since `ode` returns a matrix. Of course, we already have the vector of times `tL` which we started with, but this is also given by `out[,"time"]`, which returns the first column of the matrix `out`, which has the label "time". The same vector is of course returned using `out[,1]`.

A vector containing the concentrations of species A can be defined by `AL = out[,"A"]`, and likewise `BL = out[,"B"]`. However, we could also use `AL = out[,2]` and `BL = out[,3]`.

We use the `R` function `range`, which returns a vector, to control the vertical extent of the plot via the `plot` argument `ylim`.

```
> range(out[,"A"])
[1] 0.5739162 1.0000000
> range(out[,"B"])
[1] 0.3702302 1.0052854
> yrange  = range (c(out[,"A"], out[,"B"])); yrange
[1] 0.3702302 1.0052854
> plot(out, which = "A", xlab = "time", ylab = "concentration",
+    lwd = 3, type = "l", ylim = yrange, main = "concentation model" )
> lines(out[,"time"], out[,"B"], lwd = 3, lty = 2)
> legend("topright", legend = c("A", "B"),lwd = 3, lty = c(1, 2))
```

which produces the plot

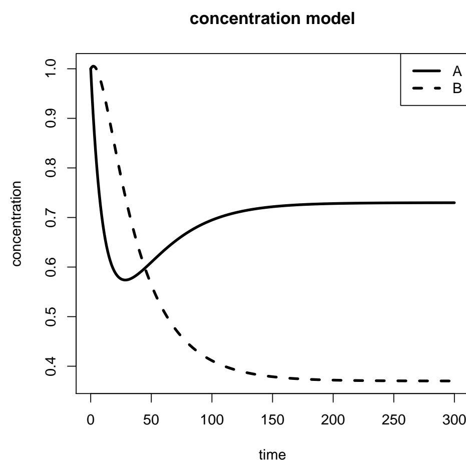

Figure 34: Concentrations of A and B vs. Time

which can be contrasted with the default and simplest plot:

```
> plot(out,lwd=2)
```

which produces

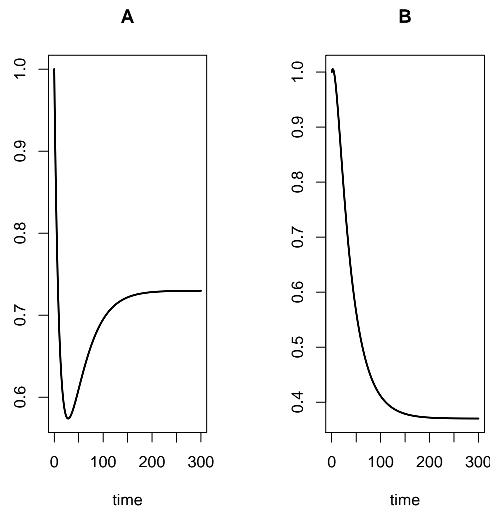

Figure 35: basic plot(out): Concentrations of A and B vs. Time

and in which the two side-by-side plots have different vertical scales.

Of course, as we have seen before, we can define the parameters used in the above problem as global parameters, and then call **ode** with the fourth arg as **parms = NULL**, and get the same solutions and the same plots.

However, you will find that **R** returns an error, claiming that the derivatives function is returning three derivatives instead of the needed two derivatives (corresponding to the number of initial values). This occurs because the symbol **y** is now being used in two ways: as a globally set parameter with the value **y = 0.1**, and also as the formal second argument of **derivs**.

As we have seen, it is no problem to have a parameter called **y** inside the **derivs** function as well as being the second formal argument of **derivs** *as long as you pass the value of the parameter* ***y*** *via the third argument of* ***derivs*** , as we did above.

So with globally defined parameter **y**, we must be more careful and redefine **derivs**:

```
> derivs = function(t, state, parameters) {
+    with (as.list(state), {
+       dA = r*(x-A) - k*A*B
+       dB = r*(y-B) + k*A*B
+       list(c(dA, dB)) }) }
> x = 1; y = 0.1; k = 0.05; r = 0.05
> out = ode(yini, tL, func = derivs, parms = NULL)
> head(out)
     time         A         B
[1,]    0 1.0000000 1.0000000
[2,]    1 0.9523189 1.0037869
[3,]    2 0.9090687 1.0052854
[4,]    3 0.8699226 1.0047151
[5,]    4 0.8345728 1.0022854
[6,]    5 0.8027203 0.9982009
```

6.3 Three First Order O.D.E.'s: The Lorenz Model

An example of a model with three o.d.e.s is the Lorenz model, (see Sec. (4.2.1) for a treatment which used our **R** code **myrk4**).

$$\frac{d\,x}{d\,t} = a\,x + y\,z \qquad (6.1)$$

$$\frac{dy}{dt} = b\,(y - z) \tag{6.2}$$

$$\frac{dz}{dt} = -x\,y + c\,y - z \tag{6.3}$$

which we solve here with **R:ode** assuming the initial conditions are $x(0) = 1, \quad y(0) = 1, \quad z(0) = 1$, and the parameters have the values $a = -8/3, \quad b = -10$, and $c = 28$. We look only at the initial stage of the evolution.

```
> library(deSolve)
> yini = c(x=1, y=1, z=1)
> a = -8/3; b = -10; c = 28
> lorenz = function(t,state,parameters) {
+     with( as.list(state), {
+         dx = a*x + y*z
+         dy = b*(y - z)
+         dz = -x*y + c*y -z
+         list( c(dx, dy, dz) )})}
> tL = seq(0,1,0.01)
> out = ode(y = yini, times = tL, func = lorenz, parms = NULL)
> head(out)
     time         x        y        z
[1,] 0.00 1.0000000 1.000000 1.000000
[2,] 0.01 0.9848912 1.012567 1.259918
[3,] 0.02 0.9731148 1.048823 1.523999
[4,] 0.03 0.9651593 1.107207 1.798314
[5,] 0.04 0.9617377 1.186866 2.088545
[6,] 0.05 0.9638068 1.287555 2.400161
```

Before making a plot, we review the steps needed to save the solution generated by **ode** to a text data file, and the methods needed to read that data file into **R** again. Since the output of **ode** is already a matrix, we don't need to use **as.matrix** to convert a **data.frame** into a matrix. We can then use **write** with the transpose of the matrix **t(out)**, providing a file name in the form of a string, and providing the number of columns the data should occupy. We can use **file.show** to display the resulting text file contents in a separate window. We can then use **read.table** to turn the contents of the text data file into a **data.frame** which we call **mydata** here.

```
> is.matrix(out)
[1] TRUE
> write(t(out),file="mydata.txt",ncolumns=4)
> file.show("mydata.txt")
```

The use of **file.show** displays the contents of the data file in a separate window, and the top part of that file is

```
0 1 1 1
0.01 0.9848912 1.012567 1.259918
0.02 0.9731148 1.048823 1.523999
0.03 0.9651593 1.107207 1.798314
0.04 0.9617377 1.186866 2.088545
0.05 0.9638068 1.287555 2.400161
0.06 0.9726091 1.409569 2.738552
```

We can then read the data file contents into a **data.frame** using **read.table**:

```
> mydata = read.table(file="mydata.txt",header=FALSE)
> head(mydata)
    V1        V2       V3       V4
1 0.00 1.0000000 1.000000 1.000000
2 0.01 0.9848912 1.012567 1.259918
3 0.02 0.9731148 1.048823 1.523999
4 0.03 0.9651593 1.107207 1.798314
5 0.04 0.9617377 1.186866 2.088545
6 0.05 0.9638068 1.287555 2.400161
> colnames(mydata) = c("t","x","y","z")
> head(mydata)
     t         x        y        z
1 0.00 1.0000000 1.000000 1.000000
2 0.01 0.9848912 1.012567 1.259918
3 0.02 0.9731148 1.048823 1.523999
4 0.03 0.9651593 1.107207 1.798314
5 0.04 0.9617377 1.186866 2.088545
6 0.05 0.9638068 1.287555 2.400161
```

```
> str(mydata)
'data.frame':   101 obs. of  4 variables:
 $ t: num  0 0.01 0.02 0.03 0.04 0.05 0.06 0.07 0.08 0.09 ...
 $ x: num  1 0.985 0.973 0.965 0.962 ...
 $ y: num  1 1.01 1.05 1.11 1.19 ...
 $ z: num  1 1.26 1.52 1.8 2.09 ...
 > is.data.frame(mydata)
[1] TRUE
> head(mydata$"t")
[1] 0.00 0.01 0.02 0.03 0.04 0.05
> head(mydata[[1]])
[1] 0.00 0.01 0.02 0.03 0.04 0.05
```

Note that the **data.frame** produced by **read.table** is not a matrix, and we must use more direct methods to extract the columns of **mydata** in order to make plots.

We now return to the matrix **out** (of class "deSolve"), produced by **ode** to make the default plot.

```
> plot(out,lwd=3,col="blue")
```

which produces the plot

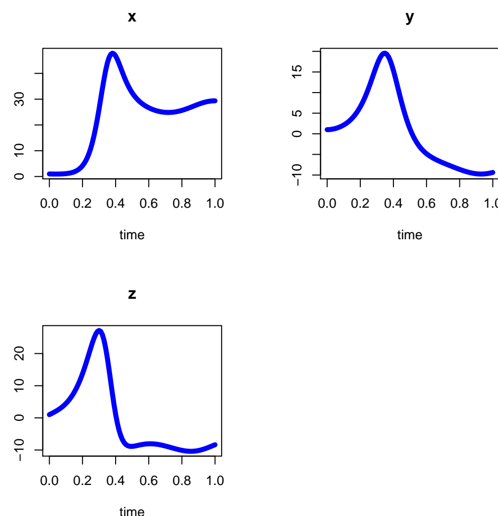

Figure 36: The Lorenz Model Using R:ode

This example illustrates the default behavior of **plot**, when the first slots are matrices of class **deSolve**, (except that we have thickened the lines, and chosen a blue color) once you have loaded in the **R** package **deSolve**. If you have passed the names of the variables to **ode** indirectly via the special form of **yini** above (" x = 1", for example), the dependent variables are plotted separately in rows of (a maximum of) two columns, with titles above, using black color, and using the style **type = "l"** automatically.

The horizontal and vertical axes are not drawn, nor is a grid added. The first column of **out** is taken to represent "times", and that automatically is the label of the horizontal axis.

The main benefit of passing the names of the dependent variables in **yini** argument of **ode** is that you can freely refer to these names in your code for the function **lorenz**, instead of referring to the dependent variables via **y[1], y[2], y[3]**.

We now use the data-frame **mydata**, generated by reading into **R** the solution data from a saved text data file, to make a plot of the **x** column versus the **t** column.

```
> plot(mydata$"t",mydata$"x",lwd=3,col="blue",ylab="x",xlab="t",type="l")
> grid(lty = "solid", col = "darkgray")
```

which produces the plot

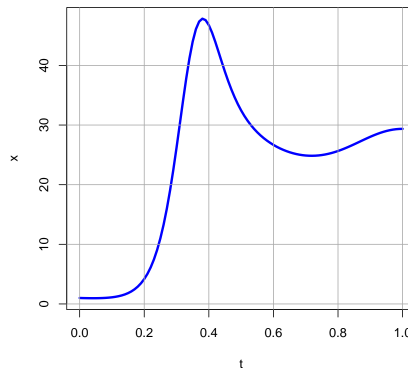

Figure 37: $x(t)$ from data-frame mydata

Instead of using the dollar sign extraction method, we can use the double bracket extraction method for obtaining a vector from a data-frame (as we must when extracting elements of a **R** list). Thus

```
> plot(mydata[[1]],mydata[[2]],lwd=3,col="blue",ylab="x",xlab="t",type="l")
> grid(lty = "solid", col = "darkgray")
```

produces the same plot.

We now combine the three curves implied by `out` into one plot.

```
> yrange = range(c(out[,"x"],out[,"y"],out[,"z"])); yrange
[1] -10.39412  47.83396
> plot(out,which = "x", type = "l", lwd = 3, col="blue",
+       xlab = "time", ylab = "", ylim = yrange,
+       main = "Lorenz Equations")
> lines(tL, out[,"y"], lwd = 3, col = "red")
> lines(tL, out[,"z"], lwd = 3, col = "green")
> grid(lty = "solid", col = "darkgray")
> legend("topright",lwd=3,col=c("blue","red","green"),
+             legend = c("x","y","z"), cex=1.2 )
```

which produces the plot

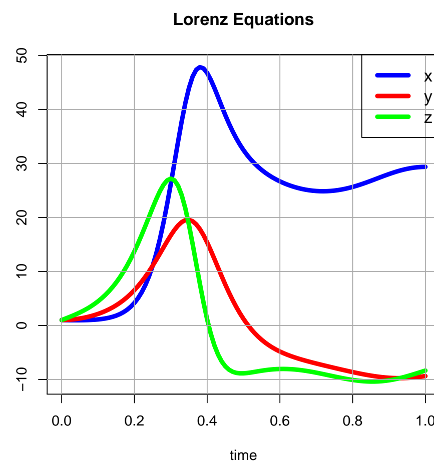

Figure 38: The Lorenz Model Using R:ode

We also plot $y(x)$.

```
> plot(out[,"x"],out[,"y"],type="l",lwd=3,col="blue",xlab="x",
+            ylab="y")
> grid(lty = "solid", col = "darkgray")
```

which produces

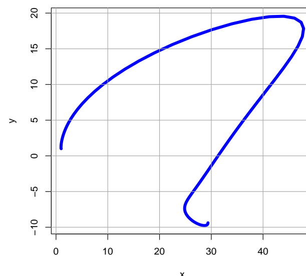

Figure 39: The Lorenz Model: $y(x)$

6.4 Solving the Stiff Case of the van der Pol Equation

We return to the van der Pol equation (see Sec. (4.1.2)). We start with the non-stiff case `mu = 1`.

```
> derivs = function(t,y,p){
+      with( as.list(y), {
+         dy1 = y[2]
+         dy2 =  mu*y[2]*(1-y[1]^2) - y[1]
+         list( c(dy1, dy2) )})}
> library(deSolve)
> tL = seq(0,30,0.01)
> yini = c(2,0)
> mu = 1
> out = ode(y=yini, times=tL, func=derivs, parms=NULL)
> colnames(out) = c("t","z","vz")
> plot(out, which="z", type="l", lwd=2, col="blue")
```

which produces the plot of `z` versus time:

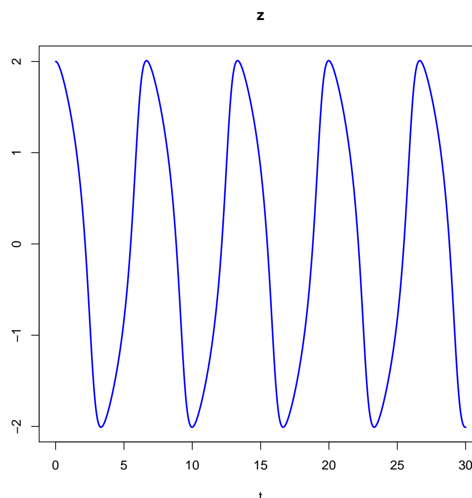

Figure 40: Non-stiff $\mu = 1$ Case van der Pol Equation

and then a stiff case $\mu = 1000$:

```
> mu = 1000
> tL = 0:3000
> fll(tL)
  0   3000   3001
> out = ode(y=yini,times=tL,func=derivs,parms=NULL)
> colnames(out) = c("t","z","vz")
> plot(out,which="z",type="l",lwd=2,col="blue")
```

which produces the plot of **z** versus time:

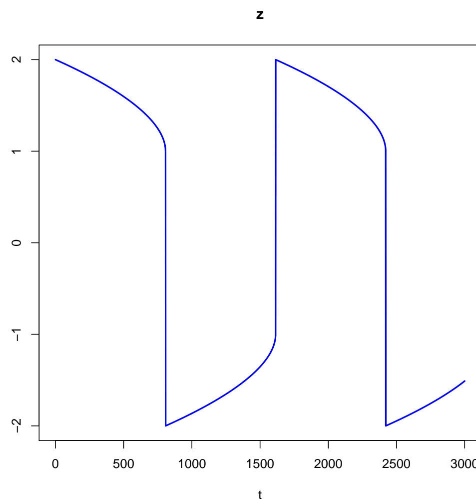

Figure 41: Stiff $\mu = 1000$ Case van der Pol Equation

7 Using External Forcing Data for O.D.E.'s

7.1 Forcing Data Using Maxima

Consider integrating $dy/dt = -t\,y + f(t)$, where $f(t)$ is an interpolating function based on a set of data for a time dependent driving force.

To be able to compare the integration with a (possibly) noisy data set with the integration with a noiseless driving term, we will start with an analytic "signal" $\cos(2\,\pi\,t)$, and find the solution using that analytic signal first. We will then add some noise to that signal and assume we need to integrate the given differential equation subject to a discrete set of noisy signal points.

This can be done by creating an interpolating function, based on that discrete noisy data, and we will then compare the solutions.

Finally, we will write the noisy discrete data to a text data file, and practice reading that data file into Maxima (as a nested list), which can then be used to create an interpolating function and solve the given driven o.d.e.

We will use the phrase "analytic solution" to refer to the solution of the o.d.e. generated by **rk** using the analytic signal expression. We assume you have loaded **k2util.mac** into your Maxima session and thus have access to the functions **fll**, **jitter**, **xyData**, and others we will use.

```
(%i1) signal(t) := cos(2*%pi*t)$
(%i2) dydt : -t*y + signal(t)$
(%i3) soln_a : rk(dydt,y,1,[t,0,1,0.01])$
(%i4) fll(soln_a);
(%o4) [[0.0,1.0],[1.0,0.62964204202687],101]
(%i5) fpprintprec:7$
```

```
(%i6) taL : take(soln_a,1)$
(%i7) fll(taL);
(%o7) [0.0,1.0,101]
(%i8) yaL : take(soln_a,2)$
(%i9) fll(yaL);
(%o9) [1.0,0.62964,101]
```

We then make a plot of the "analytic solution" of our o.d.e.

```
(%i10) plot2d([discrete,taL,yaL],[t,0,1],[xlabel,"t"],
        [ylabel,"ya"],[style,[lines,3]],
        [gnuplot_preamble,"set grid"])$
```

which produces the plot

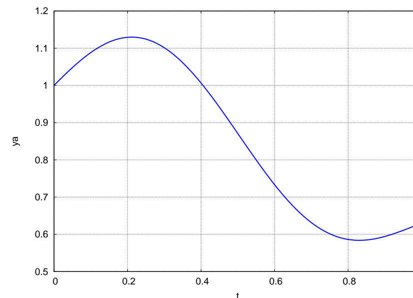

Figure 42: Solution of the ODE Using the Analytic Signal

We next generate a discrete list of signal values **sL** corresponding to the times **tL**, using the given analytic signal. We then add some semi-random noise to create the list of noisy signal values **snL** at those same discrete times.

```
(%i11) tL : makelist(t,t,0,1,0.02)$
(%i12) fll(tL);
(%o12) [0,1.0,51]
(%i13) sL : float(map('signal, tL))$
(%i14) fll(sL);
(%o14) [1.0,1.0,51]
(%i15) s1 : make_random_state(2014)$
(%i16) set_random_state(s1)$
(%i17) snL : jitter(sL, 0.2)$
(%i18) fll(snL);
(%o18) [0.90291,1.053531,51]
(%i19) plot2d([[discrete,tL,sL],[discrete,tL,snL]],
        [style,[lines],[points,1,5,1]],[legend,"signal","noisy signal"],
        [x,0,1],[xlabel,"t"],[ylabel,"signal"],
        [gnuplot_preamble,"set key bottom left"])$
```

which shows the noisy data points together with the original "analytic signal".

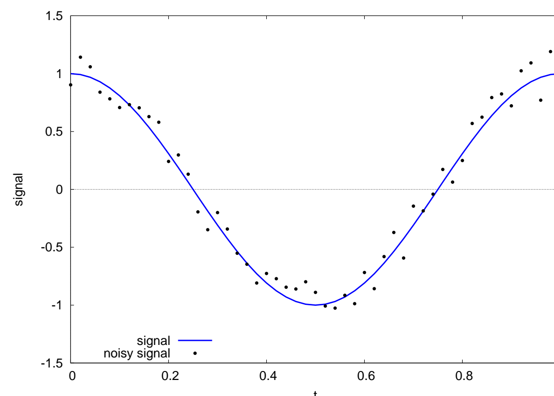

Figure 43: Noisy Data Points with Analytic Signal

Now we can use this discrete set of noisy data points to create a linear interpolating function **slin(t)**.

```
(%i20) load("interpol.mac");
(%o20) "C:/PROGRA~1/MAXIMA~3.2/share/maxima/5.31.2/share/numeric/interpol.mac"
(%i21) tsnL : xyData(tL,snL)$
(%i22) fll(tsnL);
(%o22) [[0,0.90291],[1.0,1.053531],51]
(%i23) define(slin(x), linearinterpol(tsnL))$
(%i24) slin(0.5);
(%o24) -0.88955
```

We plot the noisy data points together with the newly created linear interpolating function **slin(t)**.

```
(%i25) plot2d([[discrete,tL,snL], slin(t)],[t,0,1],
         [style,[points,2,5,1],[lines,2]],[legend,"noisy signal","slin"],
         [xlabel,"t"],[ylabel,""],
         [gnuplot_preamble,"set key bottom left"])$
```

which produces the plot

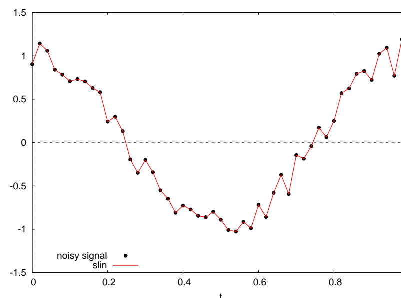

Figure 44: Noisy Data Points with Linear Interpolation

We now generate a solution of the given o.d.e. using **slin(t)** as the external driving term.

```
(%i26) dydt : -t*y + slin(t)$
(%i27) soln_lin : rk(dydt,y,1,[t,0,1,0.01])$
(%i28) fll(soln_lin);
(%o28) [[0.0,1.0],[1.0,0.65303],101]
(%i29) t1L : take(soln_lin,1)$
(%i30) fll(t1L);
(%o30) [0.0,1.0,101]
(%i31) y1L : take(soln_lin,2)$
(%i32) fll(y1L);
(%o32) [1.0,0.65303,101]
```

We then plot the analytic solution **y(t)** and the solution generated using the linear interpolating function **slin(t)**.

```
(%i33) plot2d([[discrete,taL,yaL],[discrete,t1L,y1L]],[x,0,1],
         [style,[lines]],[xlabel,"t"],[ylabel,"y"],
         [legend, "y analytic", "y slin"],
         [gnuplot_preamble,"set key bottom left"])$
```

which produces

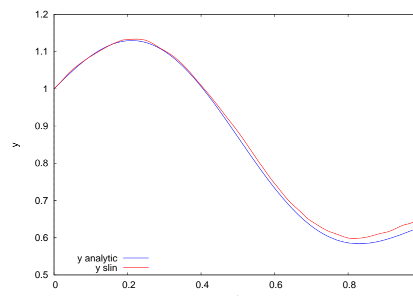

Figure 45: Solution Generated Using Linear Interpolating Function Compared

We next create a cubic spline interpolating function **csp(t)** from the noisy signal points.

```
(%i34) define(csp(x), cspline(tsnL))$
(%i35) csp(0.5);
(%o35) -0.88955
(%i36) plot2d([[discrete,tL,snL], csp(t)],[t,0,1],
         [style,[points,1,5,1],[lines]],[legend,"noisy signal","csp"],
         [xlabel,"t"],[ylabel,"signal"],
         [gnuplot_preamble,"set key bottom left"])$
```

which shows the plot

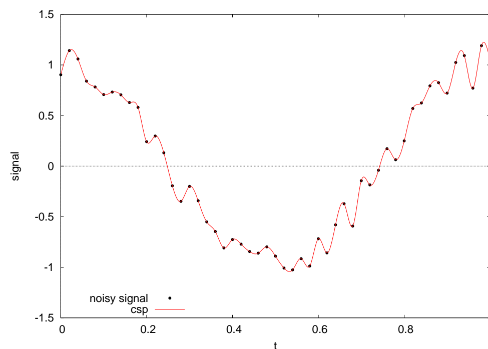

Figure 46: Noisy Data Points with Cubic Spline Interpolation

We now generate a solution of o.d.e. using the cubic spline interpolation function **csp(t)** as driving term.

```
(%i37) dydt : -t*y + csp(t)$
(%i38) soln_csp : rk(dydt,y,1,[t,0,1,0.01])$
(%i39) fll(soln_csp);
(%o39) [[0.0,1.0],[1.0,0.65395],101]
(%i40) t2L : take(soln_csp,1)$
(%i41) fll(t2L);
(%o41) [0.0,1.0,101]
(%i42) y2L : take(soln_csp,2)$
(%i43) fll(y2L);
(%o43) [1.0,0.65395,101]
```

We now make a plot of the analytic solution and the solution generated with with the cubic spline interpolating function **csp(t)**.

```
(%i44) plot2d([[discrete,taL,yaL],[discrete,t2L,y2L]],[x,0,1],
         [style,[lines]],[xlabel,"t"],[ylabel,"y"],
         [legend, "y analytic", "y csp"],
         [gnuplot_preamble,"set key bottom left"])$
```

which produces the plot

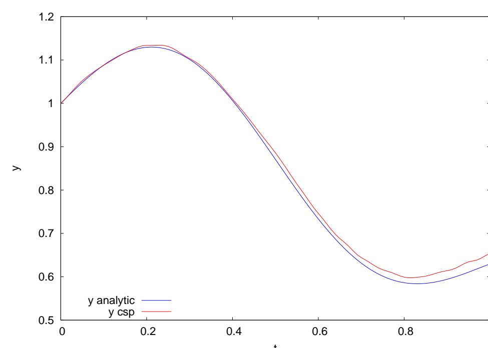

Figure 47: Solution Generated Using Cubic Spline Interpolating Function Compared

We next create a "moving average" interpolating function **sma(t)**, using **MA_smooth** from **k2util.mac**.

```
(%i45) snLs : MA_smooth(snL,3)$
(%i46) fll(snLs);
(%o46) [1.034703,1.004872,49]
(%i47) fll(tL);
(%o47) [0,1.0,51]
```

The list of "smoothed" points returned by **MA_smooth** is shorter (by two elements) than the input list **snL** and **tL**, so we chop off the first and last elements of **tL** to create **tLr**. We then use **xyData** from **k2util.mac** to create the kind of input list which we need to create an interpolating function **sma(t)**.

```
(%i48) tLr : rest(rest(tL),-1)$
(%i49) fll(tLr);
(%o49) [0.02,0.98,49]
(%i50) tsnLs : xyData(tLr, snLs)$
(%i51) fll(tsnLs);
(%o51) [[0.02,1.034703],[0.98,1.004872],49]
(%i52) define(sma(x), linearinterpol(tsnLs))$
(%i53) sma(0.5);
(%o53) -0.89857
(%i54) plot2d([[discrete,tL,snL], sma(t)],[t,0,1],
         [style,[points,1,5,1],[lines]],[legend,"noisy signal","sma"],
         [xlabel,"t"],[ylabel,"signal"],
         [gnuplot_preamble,"set key bottom left"])$
```

which produces the plot of the noisy data points with the linear interpolation **sma(t)** of the smoothed data points.

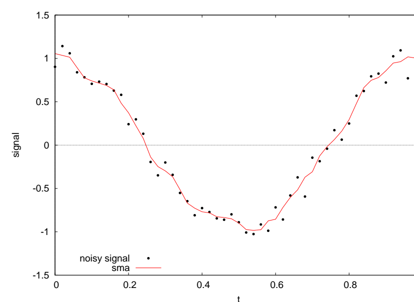

Figure 48: Noisy Data Points with the Linear Interpolation of the Smoothed Points

We can now generate a solution of the o.d.e. using **sma(t)** as the driving term.

```
(%i55) dydt : -t*y + sma(t)$
(%i56) soln_sma : rk(dydt,y,1,[t,0,1,0.01])$
(%i57) fll(soln_sma);
(%o57) [[0.0,1.0],[1.0,0.65134],101]
(%i58) t3L : take(soln_sma,1)$
(%i59) y3L : take(soln_sma,2)$
(%i60) plot2d([[discrete,taL,yaL],[discrete,t3L,y3L]],[x,0,1],
         [style,[lines]],[xlabel,"t"],[ylabel,"y"],
         [legend, "y analytic", "y sma"],
         [gnuplot_preamble,"set key bottom left"])$
```

which produces the comparison plot

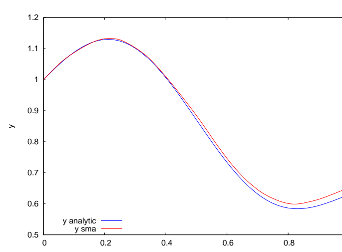

Figure 49: Solution Generated Using Linear Interpolation of Moving Average

We now write the noisy data **tsnL** to a text data file with space separation (the default) using **write_data**.

```
(%i61) fll(tsnL);
(%o61) [[0,0.90291],[1.0,1.053531],51]
(%i62) write_data(tsnL,"c:/k2/mydata.txt");
(%o62) done
```

If the file happens to be open, the write is done after the file is closed, and overwrites the previous contents.

You can load the data in the data file just created back into Maxima in the form of a nested list (which is what we need to create an interpolating function) by using **read_nested_list**. The latter function again assumes that the data on each line of the file is space separated. If the data is comma separated, you would use **read_nested_list(file-path, comma)**.

```
(%i63) tydata : read_nested_list("c:/k2/mydata.txt")$
(%i64) fll(tydata);
(%o64) [[0,0.90291],[1.0,1.053531],51]
```

7.2 Forcing Data Using R

Consider integrating $dy/dt = -t\,y + f(t)$, where $f(t)$ is an interpolating function based on a set of data for a time dependent driving force.

To be able to compare the integration with a (possibly) noisy data set with the integration with a noiseless driving term, we will start with an "analytic signal" $\cos(2\,\pi\,t)$, and find the solution using that analytic signal first. We will then add some noise to that signal and assume we need to integrate the given differential equation subject to a discrete set of noisy signal points.

This can be done by creating an interpolating function, based on that discrete noisy data, and we will then compare the solutions.

Finally, we will write the noisy discrete data to a text data file, and practice reading that data file into R, which can then be used to create an interpolating function and solve the given driven o.d.e.

We will use the phrase "analytic solution" to refer to the solution of the o.d.e. generated by **ode** using the analytic signal expression. We use the utility function **fll** included in **myode.R**, defined by:

```
##  vector utility: print out first, last and length of a vector

fll = function(xL) {
    xlen = length(xL)
    cat(" ",xL[1]," ",xL[xlen]," ",xlen,"\n") }
```

Here we define the "analytic" signal, use **ode** to solve the given o.d.e. and then make a plot of the "analytic solution" of our o.d.e.

```
> library(deSolve)
> signal = function(t) cos(2*pi*t)
> deriv = function(t,y,p) list(-t*y + signal(t))
> taL = seq(0,1,0.01)
> yini = 1
> out.a = ode(y=yini,times=taL,func=deriv,parms=NULL)
> colnames(out.a) = c("t","y")
> is.matrix(out.a)
[1] TRUE
> head(out.a)
        t        y
[1,] 0.00 1.000000
[2,] 0.01 1.009942
[3,] 0.02 1.019743
[4,] 0.03 1.029363
[5,] 0.04 1.038758
[6,] 0.05 1.047890
> fll(out.a[,"t"])
  0   1   101
> fll(out.a[,"y"])
  1   0.6296414   101
> plot(out.a,lwd=3,col="blue",main = "ode soln with analytic signal")
> grid(lty="solid",col="darkgray")
```

which produces the plot

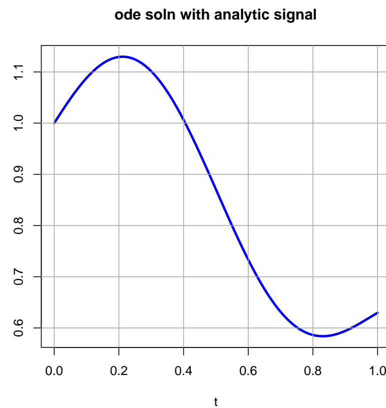

Figure 50: Solution of ODE Using Analytic Signal

We next generate a vector of signal values **sL** corresponding to a set of discrete times **tL**, using the given analytic signal. We then add some semi-random noise (using **R**'s built-in function **jitter**) to create a vector of noisy signal values **snL** at those same discrete times.

```
> tL = seq(0,1,0.02)
> fll(tL)
  0   1   51
> sL = sapply(tL, signal)
> fll(sL)
  1   1   51
> set.seed(2014)
> snL = jitter(sL, amount=0.2)
> fll(snL)
  0.9143223   1.197421   51
> curve(signal,0,1,n=200,lwd=3,col="blue",xlab="t",ylab="signal")
> points(tL,snL,pch=19)
```

which produces a plot of the signal plus noisy points:

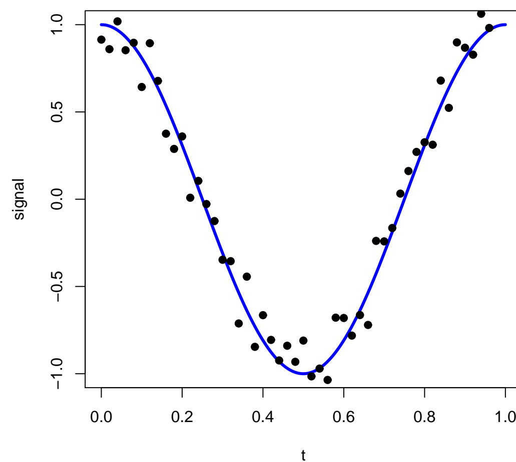

Figure 51: Signal Plus Noisy Signal Points

Now we use the **R** function **approxfun** with this discrete set of noisy data points to create a linear interpolating function **slin(t)**. We then show both the noisy data points plus the linear interpolation.

```
> slin = approxfun(tL,snL)
> slin(0.5)
[1] -0.8101318
> curve(slin,0,1,n=200,lwd=2,col="red",xlab="t",ylab="signal")
> points(tL,snL,pch=19)
```

which produces the plot:

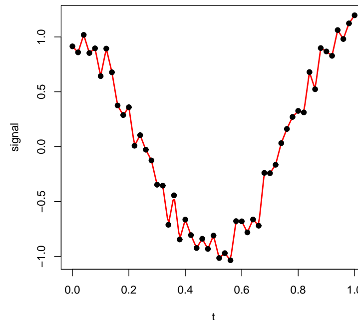

Figure 52: Noisy Signal Points Plus Linear Interpolation

We can now generate a solution of the given o.d.e. using **slin(t)** as the external driving term.

```
> deriv = function(t,y,p) list(-t*y + slin(t))
> out.lin = ode(y=yini, times=taL, func=deriv, parms=NULL)
> colnames(out.lin) = c("t","y")
> head(out.lin)
        t        y
[1,] 0.00 1.000000
[2,] 0.01 1.008956
[3,] 0.02 1.017538
[4,] 0.03 1.026278
[5,] 0.04 1.035705
[6,] 0.05 1.045012
> plot(out.a,lwd=2,col="blue")
> lines(out.lin,lwd=2,col="red")
> grid(lty="solid",col="darkgray")
> legend("topright",col=c("blue","red"),
+         legend=c("analytic","noisy - linear interp "),lwd=2)
```

which produces the comparison plot

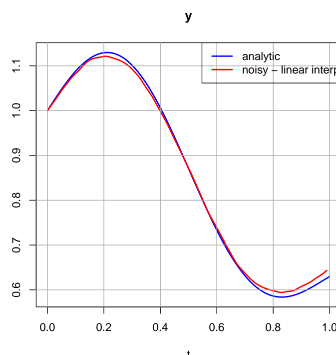

Figure 53: Solution with Linear Interpolation of Noisy Data Compared

We next create a smoothed cubic spline interpolating function **csp(t)** from the noisy signal points, using the **R** function **smooth.spline**. The function **smooth.spline** expects a **data.frame** type of argument, so we combine the times and the noisy signal values into a **data.frame**. The output of **smooth.spline** is a complicated object (use **str** on the output to see how to access the information returned). The function **smooth.spline** does not itself produce an interpolation function, but we will need to use **approxfun** on its output to get that interpolating function **csp(t)**. If you need to extract the output times and smoothed values, you use **out$x** and **out$y** respectively. Also, **out$yin** extracts the vector of input noisy values.

```
> tsn.pts = data.frame(t=tL, sn = snL)
> head(tsn.pts)
     t        sn
1 0.00 0.9143223
2 0.02 0.8596782
3 0.04 1.0189480
4 0.06 0.8536510
5 0.08 0.8962448
6 0.10 0.6429496
> tsn.pts.sp = smooth.spline(tsn.pts)
> head(tsn.pts.sp$x)
[1] 0.00 0.02 0.04 0.06 0.08 0.10
> head(tsn.pts.sp$y)
[1] 1.0076263 0.9637875 0.9178666 0.8667904 0.8083882 0.7409158
> head(tsn.pts.sp$yin)
[1] 0.9143223 0.8596782 1.0189480 0.8536510 0.8962448 0.6429496
> csp = approxfun(tsn.pts.sp)
> csp(0.5)
[1] -0.929025
> curve(csp,0,1,n=200,lwd=2,col="red",xlab="t",ylab="signal")
> points(tL,snL,pch=19)
```

which shows the plot

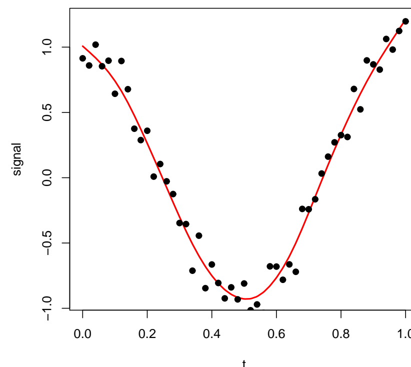

Figure 54: Noisy Signal Points Plus smooth.spline Interpolation

We can now generate a solution of the given o.d.e. using **csp(t)** as the external driving term.

```
> deriv = function(t,y,p) list(-t*y + csp(t))
> out.csp = ode(y=yini, times=taL, func=deriv, parms=NULL)
> colnames(out.csp) = c("t","y")
> head(out.csp)
        t        y
[1,] 0.00 1.000000
[2,] 0.01 1.009916
[3,] 0.02 1.019511
[4,] 0.03 1.028778
[5,] 0.04 1.037710
[6,] 0.05 1.046291
> plot(out.a,lwd=2,col="blue")
> lines(out.csp,lwd=2,col="red")
> grid(lty="solid",col="darkgray")
> legend("topright",col=c("blue","red"),
+          legend=c("analytic","noisy - smooth.spline "),lwd=2)
```

which produces the plot

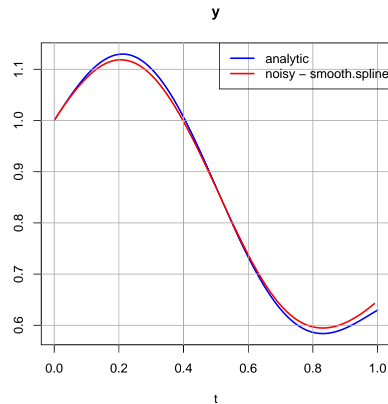

Figure 55: Solution with smooth.spline Interpolation of Noisy Data Compared

We could have skipped some of the above steps, and simply used **csp = approxfun(smooth.spline(data.frame(tL,snL)))** to generate the interpolation function based on **smooth.spline**.

```
> csp = approxfun(smooth.spline(data.frame(tL,snL)))
> csp(0.5)
[1] -0.929025
```

To use the **R** function **write** to create (or overwrite) a text data file containing the noisy signal points, we need to create a **data.frame**, convert to matrix form, and then write the transpose of that matrix, providing a file name, and the number of columns.

```
> tsn.df = data.frame(tL,snL)
> head(tsn.df)
    tL       snL
1 0.00 0.9143223
2 0.02 0.8596782
3 0.04 1.0189480
4 0.06 0.8536510
5 0.08 0.8962448
6 0.10 0.6429496
> tsn.M = as.matrix(tsn.df)
> write(t(tsn.M), file="mydata.txt", ncolumns = 2)
> file.show("mydata.txt")
```

causes the noisy data to be written to **myfile.txt**. The **R** function **file.show** opens a separate "R Information" window, and you will see that there are no text column headings written to the file, and that the default write is to use "space separation" as the data separators.

We can then read back in this noisy signal data using the **R** function **read.table**, which also assumes the default space separated data, and returns a **data.frame**.

```
> tsn.dat = read.table(file="mydata.txt",header=FALSE)
> str(tsn.dat)
'data.frame':   51 obs. of  2 variables:
 $ V1: num  0 0.02 0.04 0.06 0.08 0.1 0.12 0.14 0.16 0.18 ...
 $ V2: num  0.914 0.86 1.019 0.854 0.896 ...
> head(tsn.dat$V1)
[1] 0.00 0.02 0.04 0.06 0.08 0.10
> head(tsn.dat$V2)
[1] 0.9143223 0.8596782 1.0189480 0.8536510 0.8962448 0.6429496
```

This **data.frame** version of the noisy signal data can then be immediately turned into a (linear) interpolating function **ns(t)** and used as above, or turned into a smoothed-spline interpolating function **ns.csp(t)**, and used as above.

```
> ns = approxfun(tsn.dat)
> ns(0.5)
[1] -0.8101318
> ns.csp = approxfun(smooth.spline(tsn.dat))
> ns.csp(0.5)
[1] -0.929025
```

8 Integrating O.D.E.'s with Discontinuous Derivatives

8.1 Example 1: Oral Drug Dose Model

Example 1 Using Maxima

Let t be the time in days, $y_1(t)$ be the drug concentration in the intestine, and $y_2(t)$ be the drug concentration in the bloodstream. We quote the description of a two-compartment model describing oral drug intake, from Sec. 3.4.1.1 of Solving Differential Equations in R, by Karline Soetaert, Jeff Cash, and Francesca Mazzia, Springer-Verlag, 2012.

> Consider a person taking a pill every day at the same time [which is taken here to be at midnight]. As the pill passes the gastro-intestinal tract, the drug enters the blood by absorption through the gut wall. The delivery of the drug to the gastro-intestinal tract proceeds for 1 hour after which it ceases until the next ingestion and so on. Once in the blood, the drug distributes in the tissues, where it is chemically inactivated and subsequently excreted from the body.

An (overly) simple two-compartment model is $dy_1/dt = -a\,y_1 + u(t)$, and $dy_2/dt = a\,y_1 - b\,y_2$.

> Here a is the [blood] absorption rate, b is the removal rate from the blood, and the term $u(t)$ represents the daily delivery of the drug to the intestinal tract, which we assume to occur over a period of 1 hour. The discontinuity in this model lies in the dosing of the drug to the intestine, which takes a constant value for one hour and is then zero for the rest of the day.

We assume $a = 6$, $b = 0.6$, and $u(t) = 2$ during the drug ingestion hour at the start of each day. Since this is a periodic process, we can use the Maxima modulo function **mod**. We try two different time steps with **rk**. We need to make the time steps small enough for the solver to catch the drug ingestion process.

```
(%i1) u(t) := (if mod(24*t,24) <= 1 then 2 else 0)$
(%i2) u(1/26);
(%o2) 2
(%i3) u(1/20);
(%o3) 0
(%i4) soln_a : rk([u(t) - 6*y1, 6*y1 -0.6*y2],[y1,y2],[0,0],
              [t,0,10,0.05])$
(%i5) taL : take(soln_a,1)$
(%i6) fll(taL);
(%o6) [0.0,10.0,201]
(%i7) y1aL : take(soln_a,2)$
(%i8) fll(y1aL);
(%o8) [0.0,0.016942082631278,201]
(%i9) fpprintprec:7$
(%i10) y2aL : take(soln_a,3)$
(%i11) fll(y2aL);
(%o11) [0.0,0.13582,201]
(%i12) soln_b : rk([u(t) - 6*y1, 6*y1 -0.6*y2],[y1,y2],[0,0],
              [t,0,10,0.01])$
(%i17) tbL : take(soln_b,1)$
(%i18) fll(tbL);
(%o18) [0.0,10.0,1001]
(%i13) y1bL : take(soln_b,2)$
(%i14) fll(y1bL);
(%o14) [0.0,0.0035768,1001]
(%i15) y2bL : take(soln_b,3)$
(%i16) fll(y2bL);
(%o16) [0.0,0.11798,1001]
(%i20) plot2d([[discrete,taL,y1aL],[discrete,tbL,y1bL]],
        [xlabel,"days"],[ylabel,"y1"],
        [legend,"dt=0.05","dt=0.01"])$
```

which produces a plot of the intestine concentration $y_1(t)$ for two different time steps, with the smaller time step producing drug doses which accurately reflect the given conditions that the drug dose is provided over the course of one hour at the start of each day.

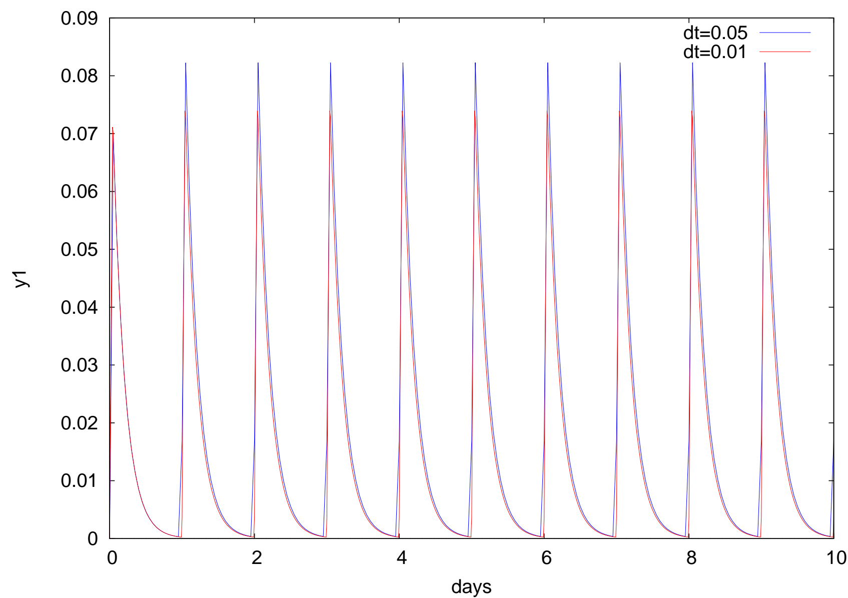

Figure 56: Intestine Drug Concentration y_1 Over Ten Days

We then make a plot of the concentration of the drug in the bloodstream $y_2(t)$ for the same two time steps.

```
(%i25) plot2d([[discrete,taL,y2aL],[discrete,tbL,y2bL]],
        [xlabel,"days"],[ylabel,"y2"],
        [legend,"dt=0.05","dt=0.01"],
        [gnuplot_preamble,"set key bottom right;set grid"])$
```

which produces the plot

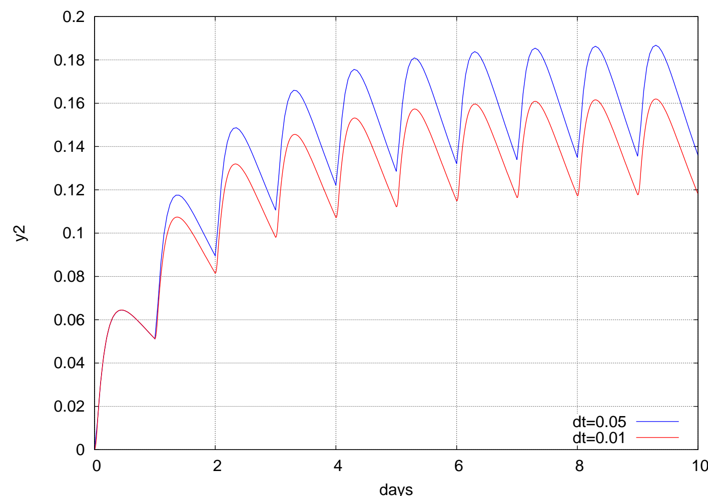

Figure 57: Blood Drug Concentration y_2 Over Ten Days

At the start of the solution, and for each first hour of the day, the drug is ingested, which causes a steep rise in the intestinal concentration. As the drug enters the blood, its concentration in the intestine decreases exponentially, while initially increasing in the blood, where it is degraded. Since the inflow to the blood drops exponentially, at a certain point in time, loss will exceed input, and the concentration in the blood will start to decrease until the next drug dose.

The initial concentration in the blood is very small, but as time proceeds, the daily-averaged concentration increases to reach some kind of dynamic equilibrium...

Example 1 Using R

See the description of this problem in the previous section. **mod(a,b)** in Maxima is **a %% b** in **R** (" a modulus b").

```
> a = 6; b = 0.6
> yini = c(intestine=0, blood=0)
> derivs = function(t,y,p) {
+     if ( (24*t) %% 24 <= 1)
+         uptake = 2
+     else
+         uptake = 0
+     dy1 = -a*y[1] + uptake
+     dy2 = a*y[1] - b*y[2]
+     list(c(dy1, dy2))}
> tL = seq(0, 10, 1/24)
> library(deSolve)
> out = ode(y = yini, times = tL, func = derivs, parms=NULL)
> plot(out, lwd = 2, xlab = "day")
```

which produces the plot

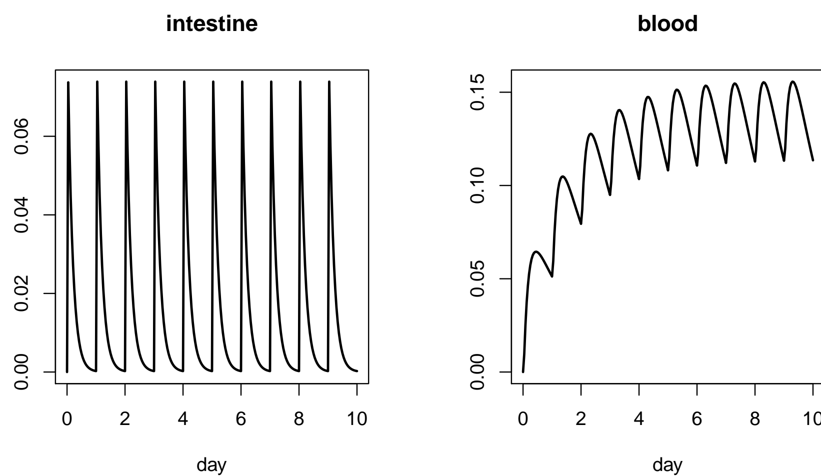

Figure 58: Intestine and Blood Drug Concentration Over Ten Days

8.2 Example 2

Example 2 Using Maxima

We solve $dy/dt = a(t)\,t\,y$, where $a(t) = 1$ for $t < 3$ and $a(t) = 0.2$ for $t > 3$. Assume $y(0) = 0.1$, and find the solution in the range $0 \le t \le 5$.

As a check on the Runge-Kutta solution, we can find the analytic solution in the separate time intervals and match the analytic solutions at $t = 3$, finding $y_1(t) = 0.1\,e^{t^2/2}$ for $0 \le t < 3$ and $y_2(t) = 3.6598\,e^{0.1\,t^2}$ for $t > 3$.

```
(%i1) a(t) := (if t < 3 then 1 else 0.2)$
(%i2) a(1);
(%o2) 1
(%i3) a(4);
(%o3) 0.2
(%i4) soln : rk(a(t)*t*y,y,0.1,[t,0,5,0.01])$
(%i5) fll(soln);
(%o5) [[0.0,0.1],[5.0,44.40743307705374],501]
(%i6) fpprintprec:7$
(%i7) plot2d([discrete,soln])$
(%i8) t1L : makelist(t,t,0,3,0.1)$
(%i9) y1L : map(lambda([t],0.1*exp(0.5*t^2)),t1L)$
(%i10) fll(y1L);
(%o10) [0.1,9.001713,31]
```

```
(%i11) t2L : makelist(t,t,3,5,0.1)$
(%i12) fll(t2L);
(%o12) [3,5.0,21]
(%i13) y2L : map(lambda([t],3.6598*exp(0.1*t^2)),t2L)$
(%i14) fll(y2L);
(%o14) [9.001655,44.58549,21]
(%i16) plot2d([[discrete,t1L,y1L],[discrete,t2L,y2L],
          [discrete, soln]],[xlabel,"t"],[ylabel,"y"],
          [style,[lines,1,1],[lines,1,1],[lines,1,2]],
          [legend,"analytic","analytic","rk"],
          [gnuplot_preamble,"set key top left;set grid"])$
```

which produces the plot

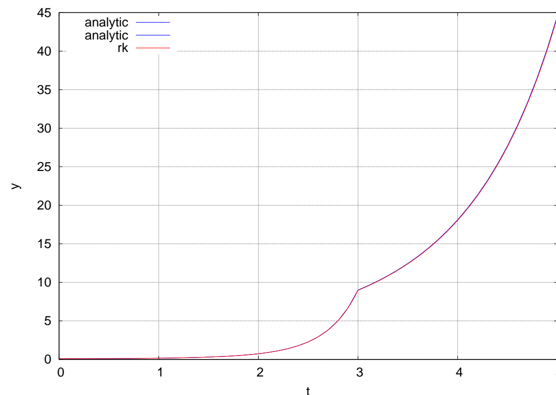

Figure 59: Analytic and rk Solution Compared

We see that the Runge-Kutta solution agrees with the analytic solution.

Example 2 Solution Using R

Here we only show the numerical integration result using **ode**.

```
> yini = 0.1
> deriv = function(t,y,p){
+      if (t<3) a=1 else a=0.2
+     list( a*t*y )}
> tL = seq(0,5,0.01)
> out = ode(y = yini, times = tL, func = deriv, parms=NULL)
> plot(out,lwd=2,col="blue",main="Example 2",xlab="t",ylab="y")
```

which produces the plot

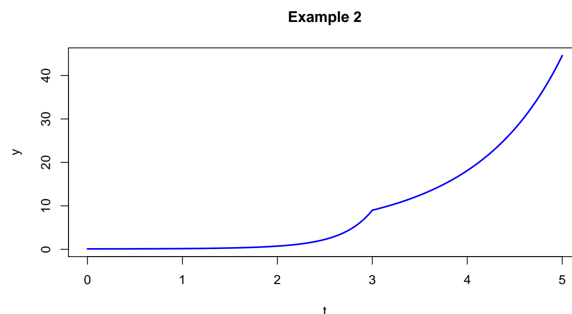

Figure 60: Example 2: ode Solution Using R

9 Integrating O.D.E.'s with Discontinuous Dependent Variables Using R

The optional `events` argument is used with `ode` or `lsoda`, etc., to allow suddent changes to the dependent variables in a set of first order ordinary differential equations.

Once **deSolve** has been loaded, using `source(deSolve)`, you can find information on the `events` argument which can be used with `ode` or some of the specific solvers, by typing `? events`. (The function `ode` uses `lsoda` by default.)

9.1 Events Specified by a data.frame in R

The first example discussed (in that `events` manual page) is a model of two variables, each of which has zero derivative, and each of which only changes in value because of a set of timed events which are described by a data frame, called `eventdat` in this example.

The four columns of this supplied data frame must be in the order (var, time, value, method), and must define 1.) the name of the variable to be affected, 2.) the time the variable is to be affected, 3.) the "value" to be used, and 4.) one of three methods.

The three available methods are `"add"`, `"multiply"` (which can be abbreviated `"mult"`), and `"replace"` (which can be abbreviated `"rep"`).

The list of output times `tL` which are supplied for the argument `times` to `ode` should include the event times in the second column of the data frame, otherwise the event will be missed.

```
> derivs = function(t,v,p){list(c(0,0))}
> yini =  c(v1 = 1, v2 = 2)
> tL = seq(0, 10, 0.1)
> eventdat = data.frame( var = c("v1", "v2", "v2", "v1"),
+                        time = c(1, 1, 5, 9) ,
+                        value = c(1, 2, 3, 4),
+                        method = c("add", "mult", "rep", "add"))
> eventdat
  var time value method
1  v1    1     1    add
2  v2    1     2   mult
3  v2    5     3    rep
4  v1    9     4    add
> out = ode(func = derivs, y = yini, times = tL, parms = NULL,
+            events = list(data = eventdat))
> plot(out)
```

which produces the plot

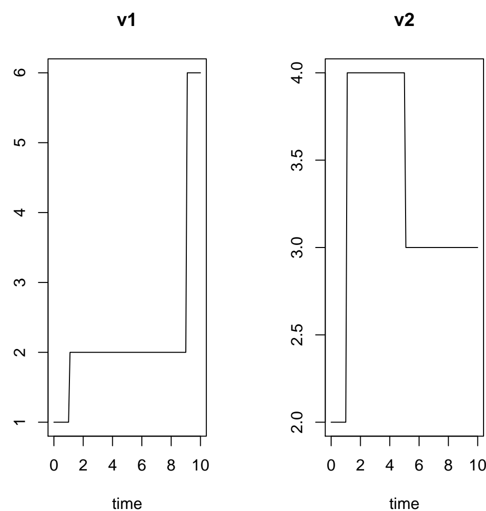

Figure 61: Events Example Using R

9.2 Intravenous Drug Injection Model Using R

An example of using a data frame to supply a sequence of events is provided by SCM in their Section 3.4.1.2. They show how to integrate a first order differential equation in which the value of the dependent variable (here the drug concentration in the blood) is suddenly increased by 40 units every day above its value just prior to the injection.

The drug is injected directly into the blood stream and then degrades as in the above example of oral ingestion of a pill, with the behavior $dy/dt = -b\,y$.

```
> b = 0.6
> yini = c(blood = 0)
> deriv = function(t,blood,p){ list(-b*blood ) }
> injectevents = data.frame(var = "blood", time = 0:20,
+                           value = 40, method = "add" )
> head(injectevents)
    var time value method
1 blood    0    40    add
2 blood    1    40    add
3 blood    2    40    add
4 blood    3    40    add
5 blood    4    40    add
6 blood    5    40    add
> tL = seq(0,10,1/24)
> library(deSolve)
> out = ode(y = yini, times = tL, func = deriv, parms=NULL,
+           events = list( data = injectevents) )
> plot(out,lwd=2,xlab="days")
```

which produces the plot

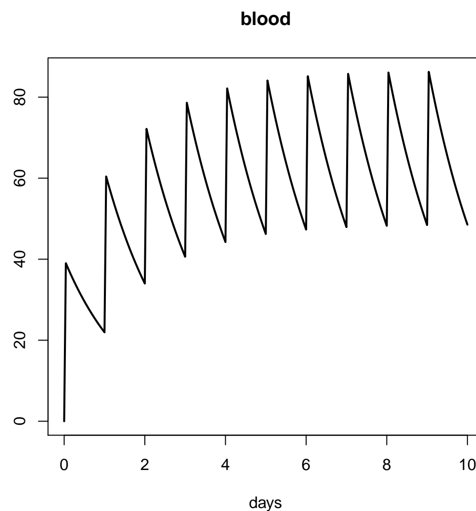

Figure 62: Daily Intravenous Drug Injection: Blood Concentration Using R

9.3 Using an Event Function at Specific Times

The second example in the **events** manual page (after some modifications here) considers two dependent variables v_1 and v_2 which obey the equations $dv_1/dt = 0$ and $dv_2/dt = -v_2/2$ except for special event moments in which v_1 increases by one unit and v_2 is replaced by the product of the new value of v_1 with a random number in the range 0 to 5.

This example calls **runif** to obtain a random number between 0 and 1. (The syntax **runif(n, min = 0, max = 1)** returns **n** semi-random numbers in the range **min** to **max**, which have the default values shown.)

We include printouts of the values of the time, the random number generated, the new value of v_1 and the new value of v_2. The **time** argument in the **events** list is chosen to ask for these event actions to take place at $t = 3$ and $t = 7$.

Once we seed the random number generator, a unique sequence of semi-random numbers in generated, which can help us understand the operation of this mode of using **ode**.

```
> derivs = function(t, v, p) {list(c(0, -0.5*v[2]))}
> yini = c(v1 = 1, v2 = 2)
> tL = seq(0, 10, 0.1)
> set.seed(2014)
> 5*runif(5)
[1] 1.4290282 0.8445435 3.1295609 1.5484316 2.7492267
> eventfun = function(t, y, p){
+     with (as.list(y),{
+         v1 = v1 + 1
+         ar = 5*runif(1)
+         v2 = ar*v1
+         cat("t = ",t," ar = ",ar," v1 = ",v1," v2 = ",v2,"\n")
+         c(v1, v2)})}
> set.seed(2014)
> out = ode( y = yini, times = tL, func = derivs, parms = NULL,
+            events = list(func = eventfun, time = c(3,7)) )
t =  0  ar =  1.429028  v1 =  2  v2 =  2.858056
t =  3  ar =  0.8445435  v1 =  2  v2 =  1.689087
t =  7  ar =  3.129561  v1 =  3  v2 =  9.388683
> plot(out,lwd=2, col="blue")
```

which produces the plot

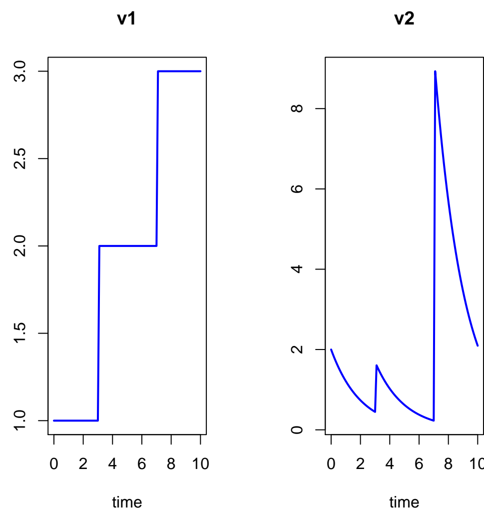

Figure 63: Event Function Example

We see from the value printouts that the function is called once at $t = 0$, which pulls the first random number **1.4290282**, but we see from the plot that this initial call to **eventfun** does not affect the values of v_1 or v_2. The second call to **eventfun** occurs at $t = 3$, and the second random number **0.8445435** is used to determine the new value of v_2. The third call to **eventfun** occurs at $t = 7$, and the third random number **3.129561** is used to determine the new value of v_2.

9.4 Example 1: Using an Event Function when a Root Condition is Satisfied

In this example (also from the **events** manual page) we consider a single dependent variable $y(t)$ for which $y(0) = 2$, and except for isolated times, $dy/dt = -y/10$, causing $y(t)$ to be a decreasing function. However, whenever the condition $y = 1/2$ is satisfied, the value of y is immediately reset to the value $y = 1$.

An "event function" called (here) **eventfun** defines this reset of y. A "root function" called (here) **rootf** defines the quantity which must evaluate to zero in order for the action specified by **eventfun** to take place. The list supplied for the **ode** arg **events** should include both the value of **func**, and the element **root = TRUE**, which replaces the list of event times, as in **time = c(3,7)** in the previous example. Finally, a new argument, **rootfun**, recognised by **ode**, must be given the chosen name of the "root function", as in **rootfun = rootf**.

```
> deriv = function(t, y, p) list(-0.1 * y)
> rootf = function (t, y, p) y - 0.5
> eventfun = function(t, y, p) y = 1
> yini = 2
> tL = seq(0, 100, 0.1)
```

```
> out = ode(y = yini, times = tL, func = deriv, parms = NULL,
+            events = list(func = eventfun, root = TRUE),
+            rootfun = rootf)
> plot(out,lwd=2,col="blue",main="",xlab="t",ylab="y")
```

which produces the plot

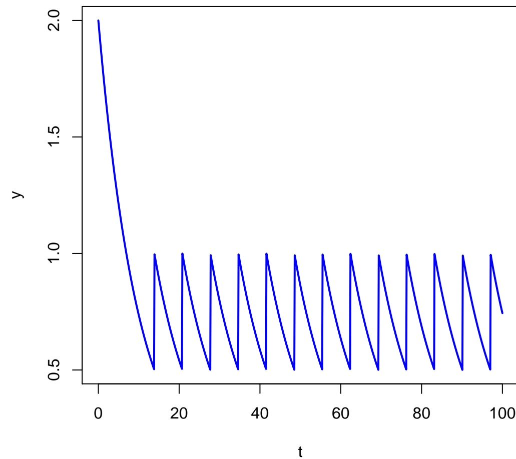

Figure 64: Event and Root Function Example

9.5 Example 2: Event Function when a Root Condition is Satisfied

Consider a simple harmonic oscillator with unit angular frequency, so $\frac{d^2x}{dt^2} = -x$. Let **y[1]** represent x and let **y[2]** represent
$v_x = dx/dt$. Then **dy[1]/dt = y[2]** and **dy[2]/dt = -y[1]**. Let **sho** be the derivatives function. Define **rootf** as the "root function", which is satisfied when **y[2] = vx = 0**. Define **eventfun** to simply return the dependent variables unaltered when an event is triggered. We assume the initial conditions **x0 = 5, vx0 = 5**, so the analytic solutions are **x = 7.0711*cos(t - pi/4)** and **vx = -7.0711*sin(t - pi/4)**, where **7.0711 = 5/sin(pi/4)**.

```
> sho = function(t, y, p )  list(c( y[2], -y[1] ))
> rootf = function(t, y, p) y[2]
> eventfun = function(t, y, p) y
> tL = seq(0, 15, 0.1)
> yini = c(5, 5)
```

We first call **ode** to define a solution without supplying an events list. In this case there is no "event", and the integration stops when the root function condition is satisfied, that is when **y[2] = 0**.

```
> out1 = ode(y = yini, times = tL,  func = sho,  parms = NULL,
+             rootfun = rootf)
> colnames(out1)=c("t","x","vx")
> plot(out1,lwd=2,col="blue")
```

which produces the plot

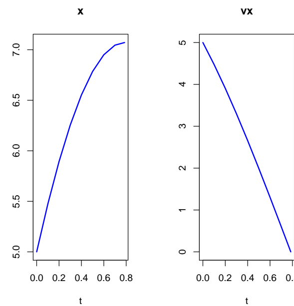

Figure 65: SHO and Root Function Example

which shows the integration halting when the velocity reaches zero, (to within double precision floating point numbers), which occurs when $t = \pi/4$.

```
> tail(out1)
              t        x            vx
[4,] 0.3000000 6.254294  3.299082e+00
[5,] 0.4000000 6.552406  2.658210e+00
[6,] 0.5000000 6.785048  1.990779e+00
[7,] 0.6000000 6.949897  1.303457e+00
[8,] 0.7000000 7.045306  6.031120e-01
[9,] 0.7853966 7.071073 -3.505903e-16
```

We next call **ode** with the **events** list, and in this case the integration continues until the end of the times provided in **tL**, and nothing interesting happens since the event function returns the dependent variables unaltered.

```
> out = ode(y = yini, times = tL,  func = sho,
+             parms = NULL, rootfun = rootf,
+             events = list(func = eventfun, root = TRUE))
> colnames(out) = c("t","x","vx")
> plot(out,lwd=2,col="blue")
```

which produces the plot

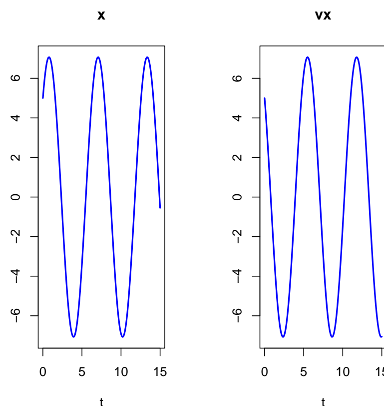

Figure 66: SHO with Event and Root Function Example 1

The object returned by **ode**, when invoked with the **rootfun** argument, includes the times when the root condition was satisfied, which is the vector extracted via: **attributes(out)$troot**.

```
> troot = attributes(out)$troot  # time of roots
> troot
[1]  0.7853978  3.9269903  7.0685830 10.2101758 13.3517685
```

which allows us to place small dots on our plot at the time values in the vector **troot**.

```
> points(troot, rep(0, length (troot)), pch=19)
```

which produces a plot in which only the second plot is altered:

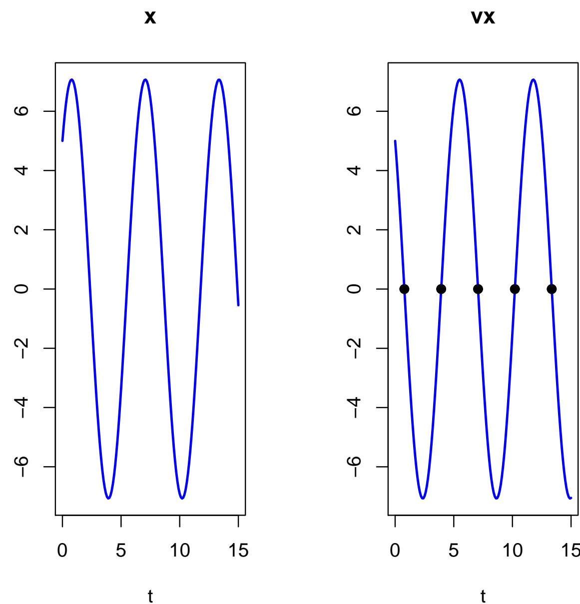

Figure 67: SHO with Event and Root Function Example 2

The last example at the end of the **events** manual page shows how to deal with multiple roots, for example triggering an event when either **x=0** or **vx=0**. SCM present the example of integration of a bouncing ball in their Sec. 3.4.2.

9.6 Use of a Switching Parameter as a State Variable

This section is related to SCM, Sec. 3.4.3, "Temperature in a Climate Controlled Room." The model dynamics will be different depending on a "switching function". If $g(t)$ is the switch, and we consider a process in which there are only two different dynamical regimes, then, for example, if **g = TRUE** then $dy/dt = f_1(t)$ and if **g = FALSE** then $dy/dt = f_2(t)$. If there are more than two regimes, we can assign numerical values to the switch to indicate the different regimes.

A specific example is a temperature controlled room in which the heating is switched on when the temperature drops below 18 deg C, and the heating is switched off when the temperature is higher than 20 deg C. We also assume a constant heating or cooling rate in the two phases.

A "parameter" is always assumed to remain constant during an integration, so the switching function cannot be represented as a parameter. Instead, we increase the number of dependent variables by one by promoting the switching parameter to be a state variable which has zero derivative, but can change value when an "event" takes place.

Let the temperature be called **temp** and be represented by **y[1]**. Let the switching parameter be called **heating_on** and be represented by **y[2]**. We assume the initial temperature ($t = 0$) is 18 deg.C (**y[1]=18**), and the heating is initially switched on (**y[2] = TRUE**).

We assume the temperature **y[1]** either increases at a rate of 1 deg.C per unit time (heat on mode: **y[2] = TRUE**) or decreases at a rate of 0.5 deg.C per unit time (heat off mode: **y[2] = FALSE**).

An alternative approach used in SCM is to let **y[2] = 1** when the heating is on and let **y[2] = 0** when the heating is off. This approach takes advantage of the fact that the numerical value **0** is interpreted as having the logical value **FALSE**, and the numerical value **1** is interpreted as

having the logical value **TRUE**. (See **? Logic**.) Then one can use the logical negation operator **!** to change the value from effectively **FALSE** to effectively **TRUE**, as shown here.

```
> as.logical(0)
[1] FALSE
> as.logical(1)
[1] TRUE
> !0
[1] TRUE
> !1
[1] FALSE
> as.numeric(!0)
[1] 1
> as.numeric(!1)
[1] 0
```

However, we choose in the following to let **y[2]** be consistently treated as a logical variable, with value either **TRUE** or **FALSE**. In the definition of **eventfunc** below, the last line must cause the function to return both of the dependent variables. In the definition of **out** below, note that **lsode** is used rather than **ode**; the latter (unless called with the option **"method" = lsode**) uses **lsoda** as the default integrator, and the regime remains in a heating on mode throughout the times specified (the events are not properly caught).

```
> yini  = c(temp = 18, heating_on = TRUE)
> derivs = function(t, y, p) {
+   dy1 = ifelse(y[2] , 1.0, -0.5)
+   dy2 = 0
+   list(c(dy1, dy2))}
> rootf = function(t, y, p) c(y[1]-18, y[1]-20)
> eventfunc = function(t, y, p) {
+     y[1] = y[1]
+     y[2] = ! y[2]
+     y}
> tL = seq(0, 20,  0.1)
> out = lsode( y = yini, times = tL, func = derivs,
+              parms = NULL, rootfun = rootf,
+              events = list(func = eventfunc, root = TRUE))
> plot(out, lwd=2, col="blue")
```

which produces the plot

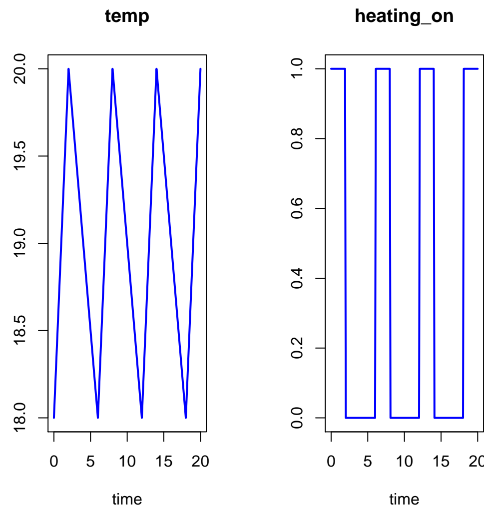

Figure 68: Temperature Controlled Room

The solver has stored the times the heating was turned on or turned off, given by the **$troot** attribute of **out**.

```
>  attributes(out)$troot
[1]  2  6  6  6  8 12 14 18
```

The fact that **t=6** is listed three times (instead of once) is a "numerical artifact" of the solver used, and SCM assert that the solver **radau** does a better job than **lsode** coping with these root events.