Advanced ODE Practical: Solving ODEs with the package `DeSolve`

INSA 3BIM – Winter 2022

This document is divided in two sections. The first section is a *tutorial*, with code examples, on solving ordinary-differential equations with the`R` pacakge `deSolve`. You will find examples to solve non-stiff and stiff systems of ODEs and display their solutions, how to manage error control, how to keep positive solutions positive, examples of numerical schemes. You will also see how to deal with events in ODEs. The second section is the list of *exercises* to do in today's practical. Feel free to skip the first section and go directly to the exercises.

Solving ODEs with the package `deSolve`: a tutorial

A first example

Load the library `deSolve`.

```
library(deSolve)
```

The main `deSolve` interface for solving ODEs is the function `ode`. It takes four mandatory arguments

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

The argument `y` is the initial conditions of the system, `t` is an array of time points at which the solution must be computed, `func` is a function defining the system of equations, and `params` contains the parameter values.

All these arguments must be defined and supplied to the function `ode`. The systems equations are defined as a `function`, with the prototype

```
ode_function_name <- function(t,y,pars,...)
```

where the arguments are: the time `t`, the state vector `y`, and the list of parameters `pars` (vector or list). The return value of the function should be a `list` whose first element is a vector of the derivative of `y` with respect to time: `dy/dt`. Other elements of the returned list can be used to pass global values, but we will not use them here. For example, the equations for the Lorenz system for atmospheric chaos is a system of three nonlinear equations:

$$\begin{aligned}\frac{dX}{dt} &= aX + YZ,\\ \frac{dY}{dt} &= b(Y - Z),\\ \frac{dZ}{dt} &= -XY + cY - Z.\end{aligned}$$

These equations can be implemented in the the function `lorenz_ODE`:

```
lorenz_ODE <- function(t, state, parameters) {
  with(as.list(c(state, parameters)), {
    dXdt <-  a * X + Y * Z
    dYdt <-  b * (Y - Z)
    dZdt <- -X * Y + c * Y - Z
    list(c(dXdt, dYdt, dZdt))
  })
}
```

In the function declaration `lorenz_ODE <- function(t, state, parameters)`, there are three arguments: time `t`, state `state` and parameters `parameters`. When running numerical simulations, the function `lorenz_ODE` will be called by the numerical solver with these three arguments. The meaning of the line `with(as.list(c(state, parameters)), ...` is less obvious. We use *named arrays* to simplify the code and make it easier to read. The function `with` takes a list and create local variables out of the elements of the list, with the the names of the elements of the list as the variable names, the second argument is an expression (a code block). For instance, in

```
with(as.list(c(a = 1, b = 2, c = 3)), { d <- a + b + c; print(paste("d =", d)) })
```

```
## [1] "d = 6"
```

the named array `c(a = 1, b = 2, c = 3)` is first converted to a list, and its elements `a`, `b` and `c` assigned to local variables. The expression `{ d <- a + b + c; print(paste("d =", d)) }` is then evaluated. The value of `with` is the value of the expression (no return is needed).

In the `lorenz_ODE` function, `as.list(c(state, parameters))` is a list of all variables and parameters.

The argument `state` expects an named array with variables names `X`, `Y` and `Z`. The initial conditions (IC) both define the names and the initial values of the variables. To set the initial conditions to $X = 1, Y = 1, Z = 1$, define the named array `IC`:

```
IC <- c(X = 1.0, Y = 1.0, Z = 1.0) # named array for the state variables
```

The argument `parameters` is used to pass system parameters to the equation. The argument expects a named array. To set the parameters to $a = -8/3, b = -10, c = 28$, define the named array `params`:

```
params <- c(a = -8/3, b = -10, c = 28) # named array for the parameters
```

The time points at which we want the numerical solution is stored in an array:

```
time_points <- seq(0, 100, by = 0.01) # array [0.0, 0.01, 0.02, ..., 100.0]
```

For the Lorenz system we have just defined, the call to `ode` is

```
sol <- ode(y = IC, times = time_points, func = lorenz_ODE, parms = params)
```

The output of `ode` is a `deSolve` matrix. The first column is time, the other columns are the solution for each of the variables of the equations. You can inspect the solution sol with the command `head`:

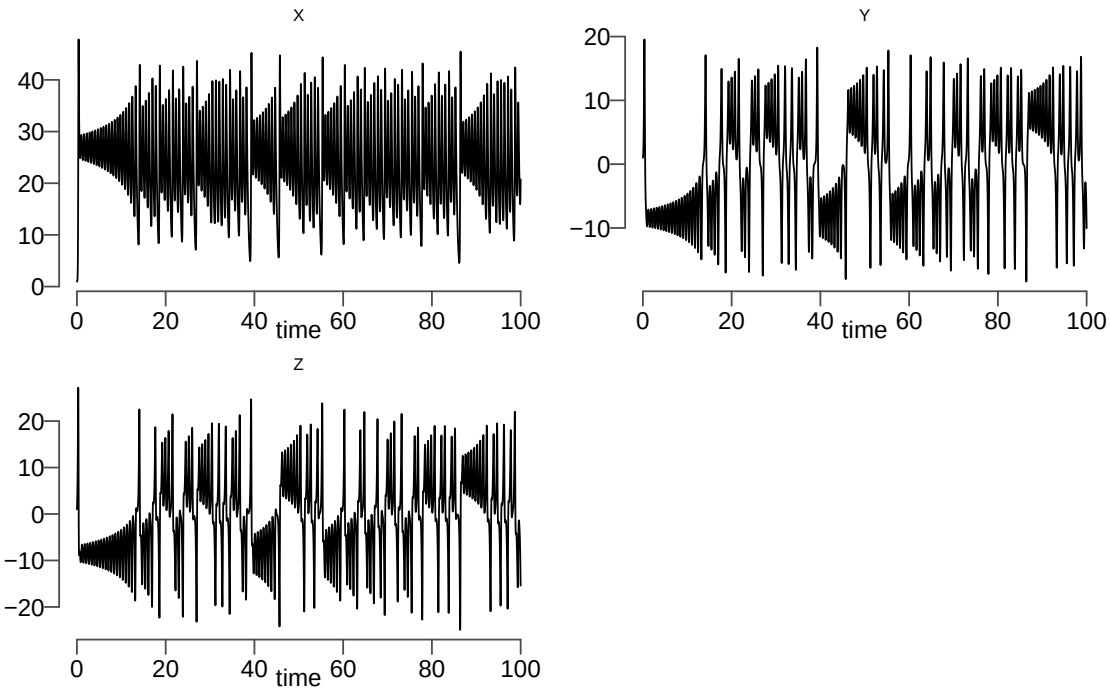
```
head(sol)
```

```
##      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
```

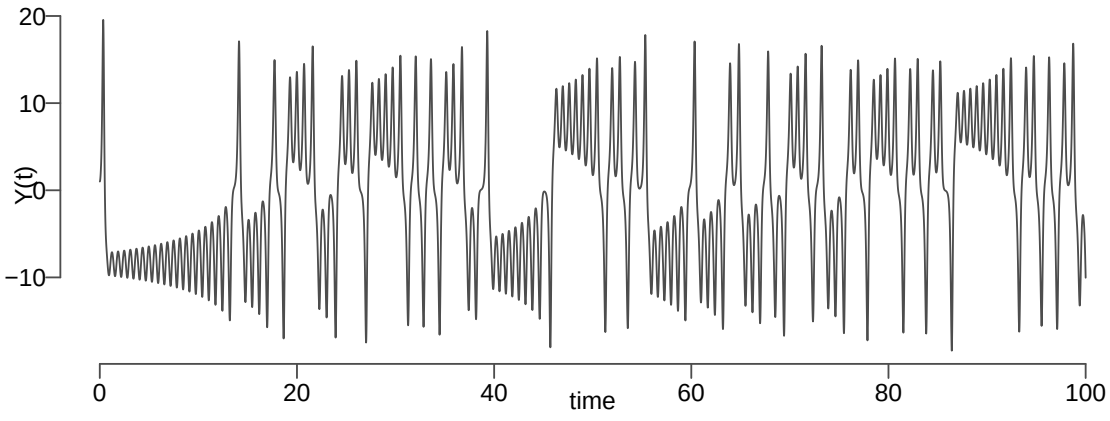
There is a plot method (`plot.deSolve`) for plotting the solution, with can be called just with `plot`. By default, `plot(sol)` will plot the time series for each state variable.

```
plot(sol)
```



Specific variables can also be plotted with the regular `plot` command

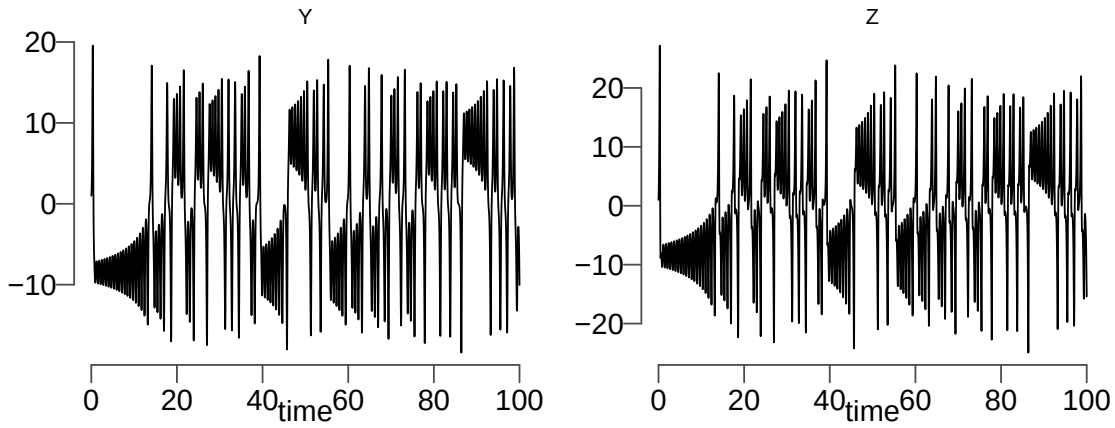
```
plot(sol[,'time'],sol[,'Y'], type = 'l', xlab = 'time', ylab = 'Y(t)')
```



Alternatively, the option `which` (or `select`) can be used to select which variables to plot.

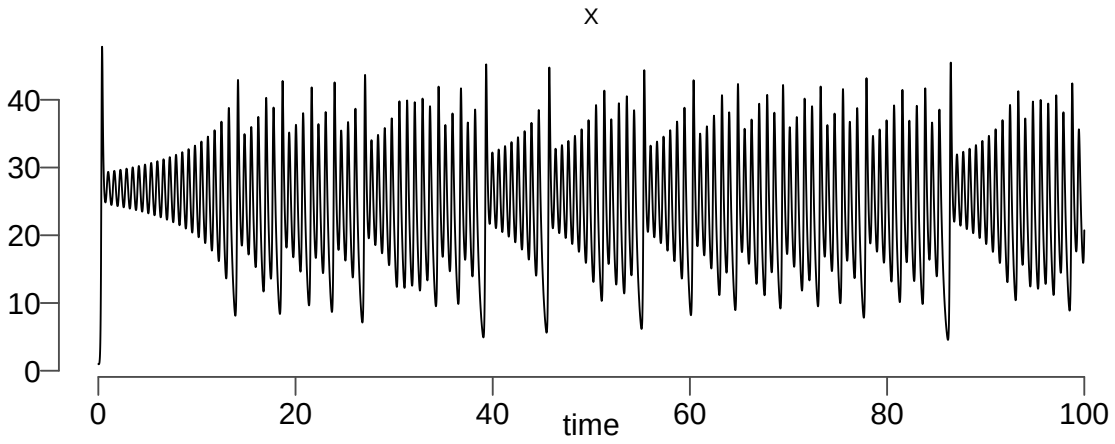
```
plot(sol, which = c('Y','Z'))
```

```
par(bg = "white", bty='n', las = 1,
      col.axis='black',fg='grey30', font.main = 1, cex.main = 0.75,
      col.lab = 'black',
     mar = c(2.,2.,1,1), mgp = c(0.8,0.5,0), cex = 0.8)
plot(sol, which = c('Y','Z'))
```



The optional argument `method` selects the numerical integrator among several. The default integrator is `lsoda`. LSODA is an integrator for solving stiff and non-stiff systems of ordinary differential equations. It was written in FORTRAN by Linda Petzold and Alan Hindmarsh. It can solve systems with dense or banded Jacobian when the problem is stiff. LSODA autamatically selects between a stiff solver (backward differencing formulae, BDF) and a non-stiff solver (Adams). This and other integrators can be used directly by their names. LSODA should be the first integrator to try when solving a new system.

```
sol.lsoda <- lsoda(y = IC, times = time_points, func = lorenz_ODE, parms = params)
plot(sol.lsoda, which = 'X')
```



Stiff equations

The stiffness of an equation has many definitions. Mathematically, a system

$$\frac{dx}{dt} = F(x)$$

is stiff if some of the eigenvalues λ_1, λ_2, of the Jacobian matrix of F have much different orders of magnitudes

$$|\lambda_1| \ll |\lambda_2|.$$

Numerically, stiff systems are best solved with *implicit* methods. Solving stiff equations with explicit methods while maintaining a good accuracy is very slow. Let's use a prototype stiff system, the van der Pol oscillator. The van der Pol oscillator is a second-order equation

$$\frac{d^2x}{dt} = \mu\Big((1-x^2)\frac{dx}{dt} - x\Big) + F(t).$$

The term $F(t)$ is a force term that we set to zero. The van der Pol oscilator is a nonlinear oscillator. Superficially, it looks like a damped harmonic oscillator, except that the damping force is *negative* when x is small, so that instead of losing energy, the system gains in energy. The system is stiff when the parameter value is large; we will set $\mu = 1e6$.

Non-stiff systems

Non-stiff systems are much easier to solve, and in most cases, explicit integrators are faster. If low accuracy is acceptable (for a non-stiff system), lower-order integrators such as `ode23`/`rk23bs`. Using explicit methods for stiff system leads either very small step sizes or to blow up of the solutions. In the worst case, the step size becomes so small it is smaller than espilon machine, so that t+dt = t, and the integrator becomes stuck. Blow up of solutions occurs when the time step is too large and solutions go out of their range of definition. For example, solution may become negative if the derivative is negative. Nonlinear functions may not be appropriately defined for negative solutions, take $1/(1+x)$, which is bounded for $x > 0$ but unbounded near $x = -1$.

The default solver `lsoda` combines two solvers (see below for details), one explicit, and one implicit. `lsoda` chooses the best solver to use at each time step, making it a powerful all-purpose numerical solver.

Error control

The integrators `lsoda` and other routines of the `deSolve` package have *error control*. The numerical methods compute approximate solution at the next time step at two different orders of accuracy. The difference between the two solutions is a good approximation of the size of the true error. This error is local, i.e. it is calculated assuming that the solution at time t is exact. Error is controled by setting *tolerance* options; if the error is larger than the tolerance, the integrator will reject the new solution and pick a smaller time step. If the error is less than the tolerance, the integrator will accept the new solution and increase the next time step to speed up the computation.

Two errors are computed, the *relative error* and the *absolute error*. In `lsoda` the solution is accepted if

$$err = \max |y_{high} - y_{low}| < \text{rtol} \times |y| + \text{atol}$$

The option `rtol` and `atol` are the *relative tolerance* and the *absolute tolerance* respectively. The smaller the tolerance the more stringent the condition for accpeting the solution. The relative tolerance control the precision of the solution when far away from zero. A relative tolerance of 1e-6 specifies 6 digits of accuracy. The absolute tolerance specifies the accuracy when solutions are close to zero. An absolute tolerance of 1e-6 states that any solution smaller than 1e-6 should be regarded as effectively zero. for system of equations, the error is set to the max of the errors on each coefficients.

Remark The error control expression involves the sum of the tolerances. This is equivalent to an OR: if the error is smaller than either tolerances, the solution will be accepted. Therefore, it is absolutely possible to set rtol or atol to zero (but not both).

In summary, rtol controls the solution away from zero and atol controls the solution close to zero.

Non-negative solutions

It can often inconvenient to have negative a solution to an initial value problem to ODEs that theoretically should have only positive solution. Problems in biology and chemistry involving concentrations or population densities should admit only positive solutions, the RHS of the ODEs are often not defined outside the range of positive states. The initial value problem (IVP) for a system of ODE is

$$\frac{dy}{dt} = f(y,t),\ y(0) = y_0.$$

For example, the IVP

$$\frac{dy}{dt} = -|y|,\ y(0) = 1.$$

The solution is $y(t) = \exp(-t)$ is positive, and decays to zero. Suppose that the numerical solution reaches a value $y^* < 0$ at a time t^*. Then the solution, starting at t^*, is $u(t) = y^* \exp(t - t^*)$. This solution is decreasing (towards $-\infty$). Thus the IVP is *unstable*, in the sense that small perturbations of the solution lead to large deviation from the true solution. This IVP will be the first test problem, solved on $[0, 40]$

Example of a Runge-Kutta pair of order 2 and 3: Bogacki-Shampine RK3(2)

The numerical scheme is a pair of Runge-Kutta formulas, one of order 3 and one of order 2. For the differential equation

$$\frac{dy(t)}{dt} = f(t, y(t)),\ a \le t \le b,$$
$$y(a) = y_a.$$

The two formulas are used to advance the solution by a time step h, given a solution y_n at time step t_n. The difference between the third order $\hat{y}_{n+1}$ and the second order step y_{n+1} provides an

estimate of the error E to the true solution. Each formula requires several evaluations of the function f. Evaluation is done in S stages:

$$y_{n+1} = \hat{y}_n + h \sum_{i=1}^{S} b_i k_i.$$

Stages k_i are calculated iteratively: k_i depends only on the previous k_j, $j = 1, ..., i-1$. The first stage is always

$$k_1 = f(t_n, y_n).$$

The other stages are

$$k_i = f\big(x_n + c_i h, y_n + h \sum_{j=1}^{i-1} a_{i,j} k_j\big),$$

and

$$c_i = \sum_{j=1}^{i-1} a_{i,j}$$

To reduce the total number of function evaluations, a strategy consists in sharing the stages k_i between the two formulas, this is called embedding. Third order Runge-Kutta formulas need at least three stages, and second order formulas at least two stages. Each stage require one evalutation of the function f. The first step k_1 is always shared. If the second stage of the third order formula is also used in the second order formula, the total number of function evaluations is 3: we get the second order approximation almost for free.

However, Bogacki and Shampine reason in a different way. They want to match as closely as possible the regions of stability of the two formulas. All two-stage, second order formulas have the same region of stability and have all three-stages, third order formulas. The only way to match the regions of stability is to increase the number of stages. Increasing the number of stages leaves quite a lot of room for choosing the coefficient, and Bogacki and Shampine took a four stages for the second order formula. This seems rather superfluous, but they also reuse the first three stages in the second-order formula. The fourth stage of the second order formula is

$$k_4 = f(t_n + h, \hat{y}_n + h \sum_{j=1}^{3} \hat{b}_j k_j).$$

Notice that k_4 is excatly the value of the function f at $t_{n+1} = t_n + h$ and $\hat{y}_{n+1} = \hat{y}_n + h \sum_{j=1}^{3} \hat{b}_j k_j$. This is the first stage k_1 of the next time step! Once we realize that most steps are successful, we see that the four-stage, second order formula come at little extra computational cost: k_4 is carried over to the next time step. This approach is call FSAL: First Same As Last.

The Butcher tableau for the pair of formula is

```
0   |
1/2 | 1/2
3/4 |   0   3/4
  1 | 2/9   1/3 4/9
--- - ----  --- --- ---
    | 2/9   1/3 4/9   0
    | 7/24  1/4 1/3 1/8
```

Event location

Event location is an option to track the solution of a system of ODE while it is being solved for. This useful, but not limited to cases when the dynamic variable are suddenly changed, i.e. when the dynamical variables are discontinuous. Normal integration routines cannot deal very well with discontinuities. The `events` option specifies when the events or discontinuities should occur, so that the integrator can deal with it. Events are imposed either by a `data.frame` that specifies the times at which the variables have jumps, or by an `event function` that monitor the state of the variable. A `root function` will trigger an event when the function takes the value zero.

The event function has the same syntax as the ode function: `function(t,y,params, ...)`. The `data.frame` should have the following columns, in that order: `var` the state variable name or number affected by the event, `time` the time at which the event takes place, `value` the magnitude of the event, `method` how to set the new values, one of "replace", "add", "multiply".

The row:

```
"v1" 10 2 "add"
```

would **add** 2 to the variable `v1` at time 10.

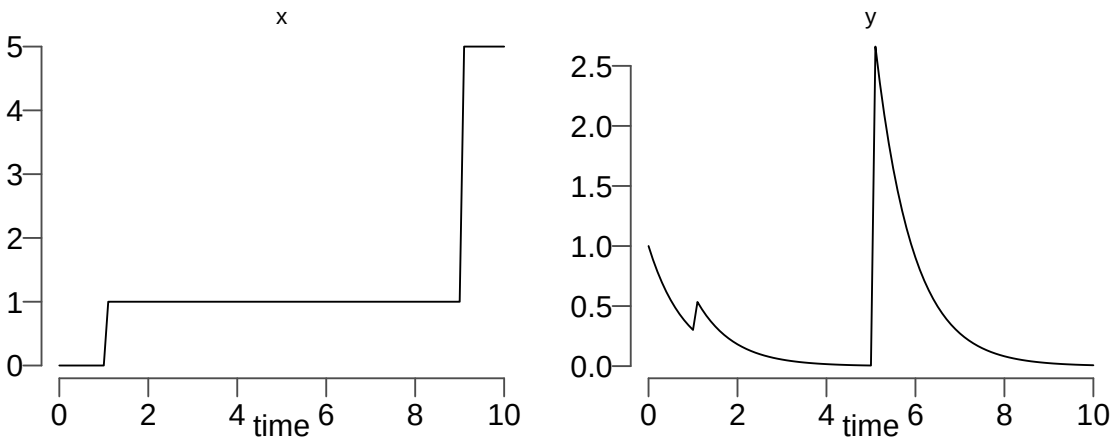
The routines `lsoda` (and some others) have root-finding abilities. If a root function is specified, the solver will stop at the first root. To monitor the location of many events, an event function must be set that will leave the state variables unaltered.

Example 1 Event in a data.frame

```r
# ODE function
myOdefunc <- function(t, var, pars)
{
  with(as.list(c(pars, var)), {
    dx <- 0
    dy <- - a * y
    list(c(dx,dy))
  })
}

y0 <- c(x = 0, y = 1) # initial condition
                      # (x,y must be the same x,y as inside myOdeFunc)
pars <- c(a = 1.2)    # parameters (a must be the same a as inside myOdeFunc)
times <- seq(0,10,by=0.1)

# define a data.frame called eventDataFrame
eventDataFrame <- data.frame(var = c("x", "y", "y", "x"),
                             time = c(1,1,5,9),
```

```
                              value = c(1,2,3,4),
                              method = c("add", "mult", "rep", "add"))

sol <- ode(y = y0, func = myOdefunc, times = times, parms = pars,
           event = list(data = eventDataFrame))
```

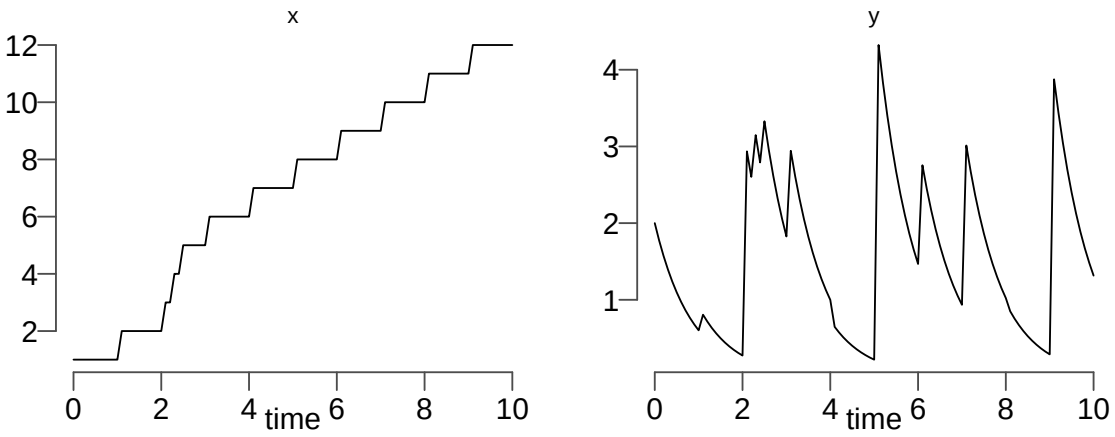
```
plot(sol)
```



Example 2 Event in a function

```
myOdefunc <- function(t, var, pars)
{
  with(as.list(c(pars, var)), {
    dx <- 0
    dy <- - a * y
    list(c(dx,dy))
  })
}

# events: add 1 to x, multiply y with a random number
eventFun <- function(t, var, parms){
  with (as.list(var),{
    x <- x + 1
    y <- 5 * runif(1)
    c(x, y)
  })
}

y0 <- c(x = 1, y = 2)
times <- seq(0, 10, by = 0.1)
pars <- c(a = 1.2)

sol <- ode(func = myOdefunc, y = y0, times = times, parms = pars,
           events = list(func = eventFun, time = c(1:9, 2.2, 2.4)) )
```

```
plot(sol, type = "l")
```

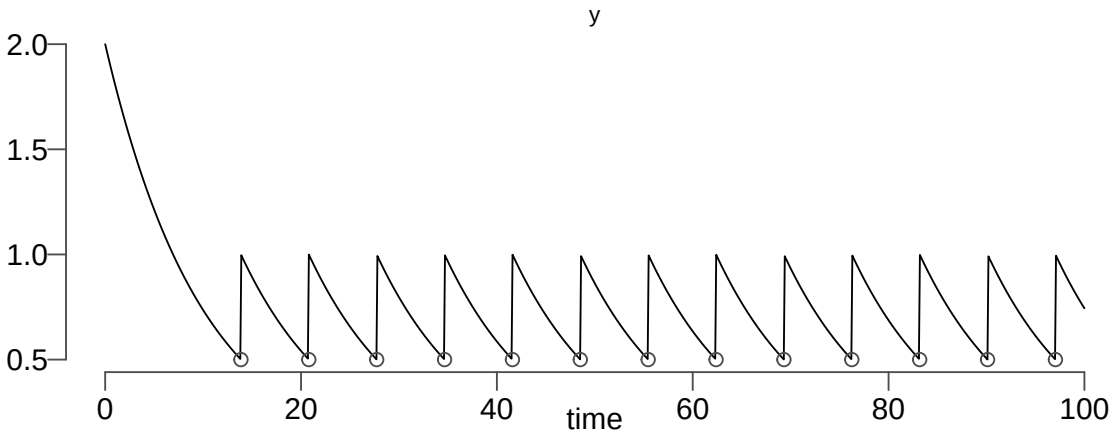


Example 3 Event triggered by a root function

```
## ODE: simple first-order decay
firstOrderDecay <- function(t, var, pars) {
  with(as.list(c(pars, var)), {
    dy <- - a * y
    list(c(dy))
  })
}

## event triggered if state variable y = 0.5
rootFun <- function (t, var, pars) {
  with(as.list(c(pars, var)), {
    y - 0.5
  })
}

## sets state variable = 1
eventFun <- function(t, y, pars) {
    with (as.list(y),{
    y <- 1
    return(y)
  })
}

y0 <- c(y = 2)
times <- seq(0, 100, 0.1)
pars <- c(a = 0.1)

## uses ode to solve; root = TRUE specifies that the event is
## triggered by a root.
sol <- ode(times = times, y = y0, func = firstOrderDecay, parms = pars,
           events = list(func = eventFun, root = TRUE),
```

```
            rootfun = rootFun)
```

```
plot(sol, type = "l")

## time of the root:
times_root <- attributes(sol)$troot
points(times_root, rep(0.5, length(times_root)))
```



Exercises

Exercise 1 on stiff equations, the van der Pol oscillator

The van der Pol equation is a second-order ODE.

$$\frac{d^2x}{dt^2} - \mu\Big((1-x^2)\frac{dx}{dt} - x\Big) = 0.$$

It looks a bit like the harmonic oscillator with friction, except that the friction term $1-x^2$ is positive when $|x| < 1$, meaning that the oscillator gains energy. We consider the van der Pol equation for times $t \in [0, 6.3]$, with initial conditions $dx(0)/dt = 0.0$ and $x(0) = 2.0$.

1. Express the equation as a system of two first-order equations by setting a second variable y, such that
$$\frac{dx}{dt} = y.$$
2. Write an `ode` function called `van_der_pol` for the van der Pol oscillator, using the two first order-equations found in 1 (use `lorenz_ODE` as a template):

```
van_der_pol <- function(t, state, parameters) {
  with(as.list(c(state, parameters)), {
    ...
    list(c(dxdt, dydt))
    })
}
```

3. Define the initial conditions `y0` and parameters `pars` as named arrays, with initial values $y = 0.0, x = 2.0$ and parameter $\mu = 500$. Set time points to `seq(0, 6.3, by = 0.01)`.

4. Solve the van der Pol oscillator with the explicit solver `rk4`. This is an explicit, method of the fourth order, and a very general purpose solver. Use a step size of 1e-3 (option `hini`). Compute the time required.

```
start_time <- Sys.time()
sol.vanderpol.rk4 <- ode(y = y0, times = time_points, func = van_der_pol,
                         parms = pars, method = 'rk4', hini = 1e-3)
end_time <- Sys.time()
end_time - start_time

plot(sol.vanderpol.rk4)
```

5. How good is the solution? Compare your solution with a solution computed at higher accuracy, by using a step size of 1e-4.

```
start_time <- Sys.time()
sol.vanderpol.rk4.high <- ode(y = y0, times = time_points, func = van_der_pol,
                              parms = pars, method = 'rk4', hini = 1e-4)
end_time <- Sys.time()
end_time - start_time

plot(time_points, abs(sol.vanderpol.rk4[,'x'] - sol.vanderpol.rk4.high[,'x']),
     type = 'l', log = 'y')
```

6. Now, use instead an implicit method, the backward-differentiation formula `BDF`. This is a variable step size method. Step size is determined to satisfy a tolerance on the accuracy of the solution. Here, use a tolerance of 1e-3. Time the simulation

```
start_time <- Sys.time()
sol.vanderpol.bdf <- ode(y = y0, times = time_points, func = van_der_pol,
                         parms = pars, method = 'bdf', rtol = 1e-3, atol=1e-3)
end_time <- Sys.time()
end_time - start_time

plot(sol.vanderpol.bdf)
```

Compare with the high accuracy solution `rk4` solution

```
plot(time_points, abs(sol.vanderpol.rk4.high[,'x'] - sol.vanderpol.bdf[,'x']),
     type = 'l', log = 'y', xlab = 'time', ylab = 'error (log)')
```

Plot both solutions on the same axes

```
plot(sol.vanderpol.rk4.high, which = 'x', ylim = c(-2,2), col = 'red', lwd = 2)
lines(sol.vanderpol.bdf[,'time'],sol.vanderpol.bdf[,'x'], col = 'blue')
```

Exercise 2 Substrate Producer Consumer model

Here we look at a Substrate Producer Consumer model (3D) with a nonlinear function passed as argument to the model. Model parameters are constant values, but what if we want to pass a time-dependent input to the system? The option `input` is there.

```r
# define the ODE equations
SPCmodel <- function(t, x, parms, input)  {
  with(as.list(c(parms, x)), {
    s0 <- input(t)                      # substrate production is time-dependent
    dS <- s0 - b*S*P + g*C              # substrate
    dP <- c*S*P  - d*C*P                # producer
    dC <- e*P*C  - f*C                  # consumer
    res <- c(dS, dP, dC)
    list(res)
  })
}

# set the parameter names and values
pars <- c(b = 0.001, c = 0.1, d = 0.1, e = 0.1, f = 0.1, g = 0.0)

# set time points
times <- seq(0, 200, by = 1.0)

# define the substrate production function
subs <- data.frame(times = times,
                   prod = rep(0, length(times)))

# = 0.2 between t = 10 and t = 11
subs$prod[subs$times >= 10 & subs$times <= 11] <- 0.2

# function interpolating the vector subs$prod
subsprod <- approxfun(subs$times, subs$prod, rule = 2)

# set initial conditions
init_conds <- c(S = 1, P = 1, C = 1)

# solve the ODE system
sol <- ode(y = init_conds, times = times, func = SPCmodel,
           parms = pars, input = subsprod)

# plot solutions
plot(sol)
```

1. Plot the input function `subsprod`
2. Modify the input function to make it periodic with period $T = 50$, with a rate of 0.04 instead of 0.2.
3. Run the simulation again with the new input function.

```r
# solve the ODE system
sol <- ode(y = init_conds, times = times, func = SPCmodel,
           parms = pars, input = subsprod)

# plot solutions
plot(sol)
```

Exercise 3 on error control

Consider Bazikin's model. We have seen that when the carrying capacity parameter ϵ^{-1} becomes large, population densities periodically come closer and closer to 0. The parameter `atol` controls what quantities should be distinguished from 0: a value `atol = 1e-2` tells the solver to ignore errors of less than 0.01. This means that when the solutions are close to zero, the solver accept any solution such that $|y| < 0.01$. For solutions to Bazikin's model, this can lead to serious numerical errors.

1. Run the code chunk below with increasing values `atol = 1e-6`, `atol 1e-4`, ... What do you see?

```
bazikin <- function(t,vars,pars) {
  with(as.list(c(vars,pars)), {
    dy1 <- x - x*y/(1+0.5*x) - epsilon*x*x
    dy2 <- - y*(1 - x/(1+0.5*x))
    list(c(dy1, dy2))
  })
}

y0 <- c(x = 5.0, y = 7.5)
tf <- 100
times.accurate <- seq(0,tf,by=0.01)
times.low <- seq(0,tf,by=2.0)
pars <- c(epsilon = 0.035)

atol <- 1e-4
rtol <- 1e-3
# first an accurate solution
sol.accurate <- ode(y = y0, times = times.accurate, func = bazikin,
           parms = pars, method = 'lsoda', rtol = 1e-9, atol = 1e-9)
# compute a low accuracy solution
sol.low <- ode(y = y0, times = times.low, func = bazikin,
           parms = pars, method = 'lsoda', rtol = rtol, atol = atol)

plot(times.accurate, sol.accurate[,'x'] , type = 'l', ylab = 'x', log = 'y')
lines(times.low, sol.low[,'x'], type = 'l', col = 'red')
points(times.low, sol.low[,'x'], pch = 19, cex = 0.5, ylab = 'x', col = 'red')
abline( h = atol, lty = 2)
```

2. Can you fix the problem by reducing `rtol`? Set `atol = 1e-4` and decrease the value `rtol`. Can you get a better solution?

3. Now we set $\epsilon = 0.06$ so that solutions do not oscillate too close to zero. Run the code chunk below with increasing values of `rtol = 1e-4`, `rtol = 1e-3`, ... What do you see?

```
pars <- c(epsilon = 0.06)

atol <- 1e-3
rtol <- 1e-4
# first an accurate solution
sol.accurate <- ode(y = y0, times = times.accurate, func = bazikin,
```

```
           parms = pars, method = 'lsoda', rtol = 1e-9, atol = 1e-9)
# compute a low accuracy solution
sol.low <- ode(y = y0, times = times.low, func = bazikin,
           parms = pars, method = 'lsoda', rtol = rtol, atol = atol)

plot(times.accurate, sol.accurate[,'x'] , type = 'l', ylab = 'x')
lines(times.low, sol.low[,'x'], type = 'l', col = 'red')
points(times.low, sol.low[,'x'], pch = 19, cex = 0.5, ylab = 'x', col = 'red')
abline( h = atol, lty = 2)
```

Exercise 4 on non-negative solutions

1. Numerically solve the equation above, using the code below. How would you modify the code to make sure solutions with positive initial conditions stay positive?

```
# define the ODE equations
nonStiffTest1 <- function(t, x, parms)  {
  with(as.list(c(parms, x)), {
    dy <- - abs(y)
    list(dy)
  })
}

# set time points
times <- seq(0, 40)

# set initial conditions
y0 <- c(y = 1)

# solve the ODE system
sol <- ode(y = y0, times = times, parms = NULL, func = nonStiffTest1)

# plot solutions
plot(sol)
```

2. Another test is the Robertson problem. It is a semi-stable chemical reaction system

$$\begin{aligned}
\frac{dy_1}{dt} &= -0.04y_1 + 10^4 y_2 y_3, \\
\frac{dy_2}{dt} &= 0.04y_1 - 10^4 y_2 y_3 - 3 \times 10^7 y_2^2, \\
\frac{dy_3}{dt} &= 3 \times 10^7 y_2^2,
\end{aligned}$$

with initial conditions $(1, 0, 0)^t$ with time interval $[0, 4 \times 10^{11}$. Solutions should stay non-negative. This is a *stiff* problem that need to be solved over a very large time interval. Run the code below, and check whether solutions stay positive. Modify the code to make sure solutions stay positive.

```
robertson <- function(t, state, parameters) {
  with(as.list(c(state, parameters)), {
    dy1 <-  -a*y1 + b*y2*y3
    dy2 <-   a*y1 - b*y2*y3 - c*y2*y2
    dy3 <-   c*y2*y2
    list(c(dy1, dy2, dy3))
  })
}

# set the parameter names and values
pars <- c(a = 0.04, b = 1e4, c = 3e7)

# set time points
times <- seq(0, 4e11, length.out = 50)

# set initial conditions
y0 <- c(y1 = 1, y2 = 0, y3 = 0)

# solve the ODE system
sol <- ode(y = y0, times = times, parms = pars, func = robertson,
           rtol = 1e-6, atol = 1e-6)
```

```
# plot solutions
plot(sol, which = 'y1')
```

3. The Lotka-Volterra model for predator-prey dynamics admits oscillatory solutions that periodically come arbitrarily close to zero. The problem is

$$\frac{dy_1}{dt} = 0.5y_1\left(1 - \frac{y_1}{20}\right) - 0.1y_1y_2, \frac{dy_2}{dt} = 0.01y_1y_2 - 0.001y_2,$$

with initial conditions $(25, 5)^t$ over the time interval $[0, 1000]$.

Run the code below, and modify the code to make sure solutions stay positive.

```
lotka_volterra <- function(t, state, parameters) {
  with(as.list(c(state, parameters)), {
    dy1 <-  a * y1 * ( 1 - y1/b ) - c * y1 * y2
    dy2 <-  d * y1*y2 - e * y2
    list(c(dy1, dy2))
  })
}

# times
times <- seq(0,1000, length.out = 101)

# set the parameter names and values
pars <- c(a = 0.5, b = 20, c = 0.1, d = 0.01, e = 0.001)

# set initial conditions
y0 <- c(y1 = 25, y2 = 5)
```

```r
# solve the ODE system
sol.lotka <- ode(y = y0, times = times, parms = pars, func = lotka_volterra)

# plot solutions
plot(sol.lotka)
```

4 . The IVP

$$\frac{dy}{dt} = \sqrt{1-y^2},\ y(0) = 0.$$

The solution $y(t) = \sin(t)$ increases to 1 at $t = \pi/2$, and the right-hand-side is not defined for $y > 1$ (it is not Lipschitz). The solution it not unique after $t^* = \pi/2$.

Run the following chunk of code, and check that the solution cannot be prolonged after $t = \pi/2$.

```r
# define the ODE equations
ball <- function(t, x, parms)  {
  with(as.list(c(parms, x)), {
    dy <- sqrt(1-y*y)
    list(dy)
  })
}

# set time points
times <- seq(0, 3, length.out = 100)

# set initial conditions
y0 <- c(y = 0)

# solve the ODE system
sol <- ode(y = y0, times = times, parms = NULL, func = ball, method = 'bdf')

# plot solutions
plot(sol)
```

Exercise 5 on events, conditional daily drug administration

1. Using the ode function `firstOrderDecay`, define an root function that returns zero if y becomes smaller than 0.5, or if `(t %% 24) -8` crosses zero (`%%` is the remainder of the division of `t` by 24). Define an event function that sets $y \to y+1$ if $y \leq 0.5$ and $y \to y+0.5$ otherwise.