ICBM differential form implementation in R

Lorenzo Menichetti

2023-06-08

Implementing a linear model as an ODE system

This is a very flexible approach for anyone who knows already some R basics because it allows us to build a model (any linear model, and to some extent also nonlinear) just by defining it as an ODE system.

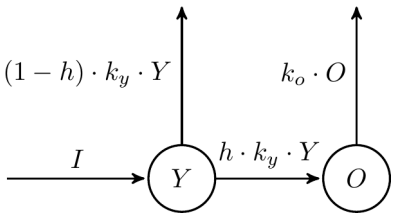
An ODE system is quite easy conceptually, particularly when compared to its analytical solution over time. This allows you to easily modify your models, adding or taking out terms depending on the hypotheses you want to represent.

ICBM as ODE

We have seen this already but as a reminder here it is the model we want to implement:

$$\frac{dY_{(t)}}{dt} = I - k_y \cdot r \cdot Y \frac{dO_{(t)}}{dt} = h \cdot k_y \cdot r \cdot Y - k_o \cdot r \cdot O$$

Which can be conceptually visualized like this:



Loading the solver

```
library(deSolve)
```

This loads a pretty standard ODE solver library, which contains various numerical methods we are not going to dig into now.
it is enough to know that it calculates the solution with numerical optimization (which is not exact but an approximation, so your results may vary ever so slightly depending on the seeds you set in R or on the different executions. It is a minimal variation anyway you can neglect).

Setting up the model ODE to solve

```
##function (differential form)
ODEfun <- function(time=seq(1:times_range), state=state,
                   parms=parms) {
  with(as.list(c(state, parms)), {
    .Y=I-ky*r*Y
    .O=h*ky*r*Y-ko*r*O
    return(list(c(.Y, .O)))
  })
}
```

The order of the results will be the one specified in the output (`return(list(..., ..., ...))`).

Definiting initial state and parameters

We need to define the initial state of the system. This is a vector containing one entry for each of the variables defined in the ODE, and the order should match.

```
init=c(Y=2, O=10)
```

We then define all our model parameters (constants)

```
parameters=c(ky=0.8, ko=0.0065, h=0.15, r=1, I=1)
```

And we specify the time vector for the solution. The time steps can be arbitraru big or small, not necessarily 1. The time unit is defined by the model (a certain parameterization will be in a specific time unit)

```
time_vector=seq(0,100)
```

Running the ODE within the solver

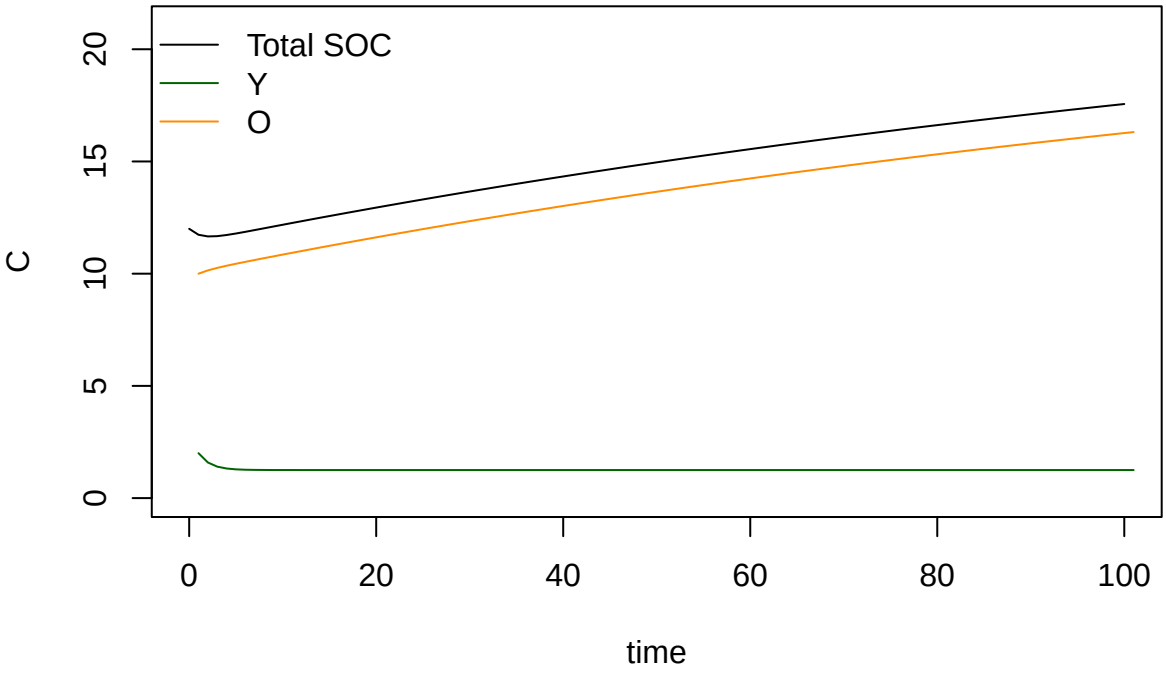
We can now run our solver to find the integrated solution over time:

```
simulation <- ode(y = init, time = time_vector, func = ODEfun,
                  parms = parameters)
```

We have now a data frame with three columns, the first is time and the others are the solutions of our ODE in the order specified in the model definition. We can sum up the two pools to find the total SOC

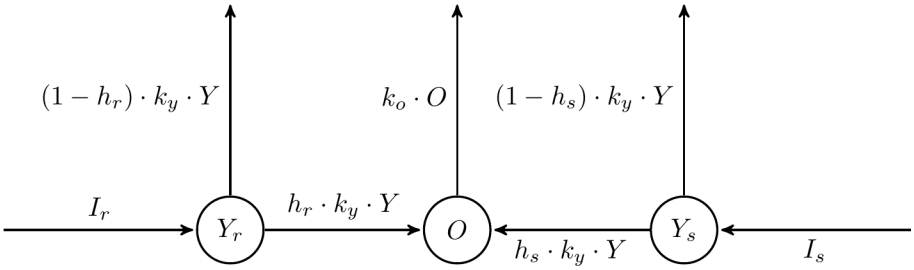
```
total_SOC=simulation[,2]+simulation[,3]
```

Plotting the results



What if?

For example, what if you wanted to modify the model adding one specific Y pool for roots and one for shoots?



What if?

You can then just define the model as an ODE system:

$$\frac{dY_{r,}}{dt} = I_r - k_y \cdot r \cdot Y \frac{dY_s}{dt} = I_s - k_y \cdot r \cdot Y$$

We then have to define these fluxes into the O pool as a sum:

$$\frac{dO}{dt} = h_r \cdot k_y \cdot r \cdot Y_r + h_s \cdot k_y \cdot r \cdot Y_s - k_o \cdot r \cdot O$$

And now we are ready to write this system into the solver, and then we have our model.

What if?

This is just an example. We assumed, among many other things, that the two `Y` pools decay at the same constant rate `k_y`, but this is not necessarily the case, it all depends on the hypothesis you want to test. You can think to virtually any hypothesis (provided it is fitting a compartmental linear model definition) and represent it with an ODE. And you can also go further (like nonlinear hypotheses) since `deSolve` also solves PDE systems, I would just not go there without knowing what you are doing in terms of the properties of the model object you are creating...

Calibrating a model

We want to calibrate a model on a time series of measured SOC. We will use the time series of the measurements in the farmyard manure (here below)

```
measured_SOC= data.frame(
  c(1956, 1967,1974, 1975,1977, 1979,1983,1985,1987,1989,1991,1993,1995,1997,1999,2001,
2005,2007,2009,2011,2013,2015,2017,2019),
c(43315.2,46099.1354432654,54402.0514037605,54041.8051533193,55034.1640770398,
  54615.8930041639,52681.7594782169,53624.496505942,54005.0133116738,60017.1966448382,
  61107.5003781939,51574.6724815113,54275.1183294204,57965.4787115247,57956.4965618206,
  56938.4828493794,55910.6109731281,56683.1588144316,58995.1969362421,55438.5423266927,
  56744.1691233292,56825.0174199966,57116.9017798636,54074.4482017914)
)
colnames(measured_SOC)=c("year", "SOC")
measured_SOC$year=(measured_SOC$year-measured_SOC$year[1])+1
```

You can read the time series you want of course, here I am inserting them manually so that we don't need to use external files in the tutorial.

Optimizing for one single parameter (univariate)

The first step is to write the function we want to minimize (or maximize, depending on the target we chose), often called the "cost function". We will use RMSE as fitness measurement, from the hydroGOF package:

```
ICBM_cost_univariate<-function(x, pars, inputs){
  #defining the lenght of the simulation
  sim_length=100
  #assuming an initialization where Y is 5% and O is 95% of the total SOC
  initialization=c(Y=measured_SOC$SOC[1]*0.05, O=measured_SOC$SOC[1]*0.95)
  #defining the parameter list for the ODE functino
  parms=c(ky=x, ko=pars[1], h=pars[2], r=pars[3], I=pars[4])
  #running the ODE solver
  simulation <- ode(y = initialization, time = seq(1:sim_length), func = ODEfun, parms)
  #summing up the pools and calculating the RMSE
  total_SOC=simulation[,2]+simulation[,3]
  RMSE<-rmse(total_SOC[measured_SOC$year], measured_SOC$SOC)
  return(RMSE)
}
```

We need to declare again the arguments to the ode solver (for solving ODEfun) because this function is now running within the environment of the cost function (there are other ways to implement this, but let's skip for not)

Optimize

It is then just a matter of running the optimization

```
library(hydroGOF)
```

```
## Loading required package: zoo
```

```
##
## Attaching package: 'zoo'
```

```
## The following objects are masked from 'package:base':
##
##     as.Date, as.Date.numeric
```

```
#define some parameters fo pass to the optimize function to run the model
parameters=c( 0.0065, 0.1500, 1.0000, 840+1770)

xmin <- optimize(ICBM_cost_univariate, c(0, 1), tol = 0.0001, pars=parameters)
```

The object we just created:

```
xmin
```

```
## $minimum
## [1] 0.2308074
##
## $objective
## [1] 2508.176
```

contains the optimization value we were looking for (for ky) and the final value of the RMSE where the optimization converged