Build-a-model

Lorenzo Menichetti

2022-10-27

Introduction

This brief tutorial aims at teaching you how to build a compartmental decomposition model.
The only prerequisites are:
1) Know how to use an R script (executing it)
2) Know how to use copy-paste
3) Know roughly what we are talking about with organic matter decomposition (I mean, that organic matter decomposes)

We will rely on differential equations.
This might sound scary, if you don't like maths, but computers have a lot of tricks to help you. A differential equation is extremely intuitive to read and write… it is just very difficult to solve. But we don't need to do that, the computer will do it for us.
The tutorial is written in extremely blunt and simple (maybe simplistic, sometimes) form.

Installing a functional R environment

An ODE solver?

An ODE is an Ordinary Differential Equation. These describe a process "collapsing" the time, we consider an infinitesimally small fraction of time so that time becomes somehow irrelevant. In other words, an ODE defines the derivative of a function, while its solution describes the function over time.
Solving an ODE, or even worse an ODE system, can be quite tedious, but computers have neat tricks to do it for us. They do it "numerically", which means they find an approximate solution and not an exact one, but this is totally fine for us since they can approximate with a very high precision.
We do not need to understand how an ODE solver works, here, it just does. You give it the differential equation, and the solver uses it to develop a solution over time.

Setting up the ODE

This is the important part, here you are building your model. All the rest are just technicalities.
We are going to describe the decomposition of a mass of organic carbon C_0 (carbon at time 0) over time. Every instant we have a fraction of that carbon k that decomposes, because microbes eat part of it (so the bigger the mass you have, the more C is going to leave it per unit of time). We can write it in differential form as:

$C' = C \cdot -k$

or if you prefer:

$\frac{dC}{dt} = C \cdot -k$

These two (equivalent) forms The term k is negative because we talk about decay (so it is a fraction of mass leaving the system we consider), but if you set it positive you can use the same function to describe growth (not the process we are talking about here, though).
..that's it. This is your model.
Done.

This equation becomes, when solved for time (t), becomes:

$C_t = C_0 \cdot e^{-k \cdot t}$

Which describes the evolution of the Carbon with time (C_t). It is an exponential function, rather common in nature. But for now skip the solution, because we won't need it here, and the approach you will learn is more general than that because you will be able to apply it to any ODE, and specifically will allow you to build rather complex compartmental decomposition models easily.

Writing the ODE in R and using a solver

We need to write the ODE in a form that R can digest. We will use the package `deSolve` as solver. To use it, we first need to define our ODE into an own function:

```r
exp_diff<-function(Time, State, Pars) { #this is how you define your own functions
  #in R, () are the inputs to the function and {} is the function.
  # the operator <- stores the results in the object you declare at its right
  with(as.list(c(State, Pars)), {
    C    <- C*-k # THIS LINE HERE IS YOUR MODEL, the rest does not matter
    return(list(c(C))) #this line is to make the function return the results
  })
}
```

The function is written in order to be generic, so that you can just change your ODE (or system of ODEs) and it would still work.

We then define the time over which we want to solve our equation (`Time`), the initial state of the system (`State`, in this case for only one variable C), and the parameters (`Pars`, in this case only k):

```r
Time <- seq(0, 20, by = 1) #a sequence of numbers from 0 to 20 time steps
State <- c(C=80) #we need to give all the states as a vector of states, with c()
#the vector elements are inside the parenthesis (only one in this case)
Pars <- c(k=0.15) #as above
```

We then proceed to run our ODE solver like this, with the function `ode`:

```r
library(deSolve)
```

```
## Warning: package 'deSolve' was built under R version 4.0.5
```

```r
exp_diff<-function(Time, State, Pars) {
  with(as.list(c(State, Pars)), {
    C    <- C*-k
    return(list(c(C)))
  })
}

Time <- seq(0, 20, by = 1)
State <- c(C=80)
Pars <- c(k=0.15)

output   <- ode(State, Time, exp_diff, Pars)
```

(if you are curious about the syntax of an R function, remember you can just write `?ode` in the console and press enter)

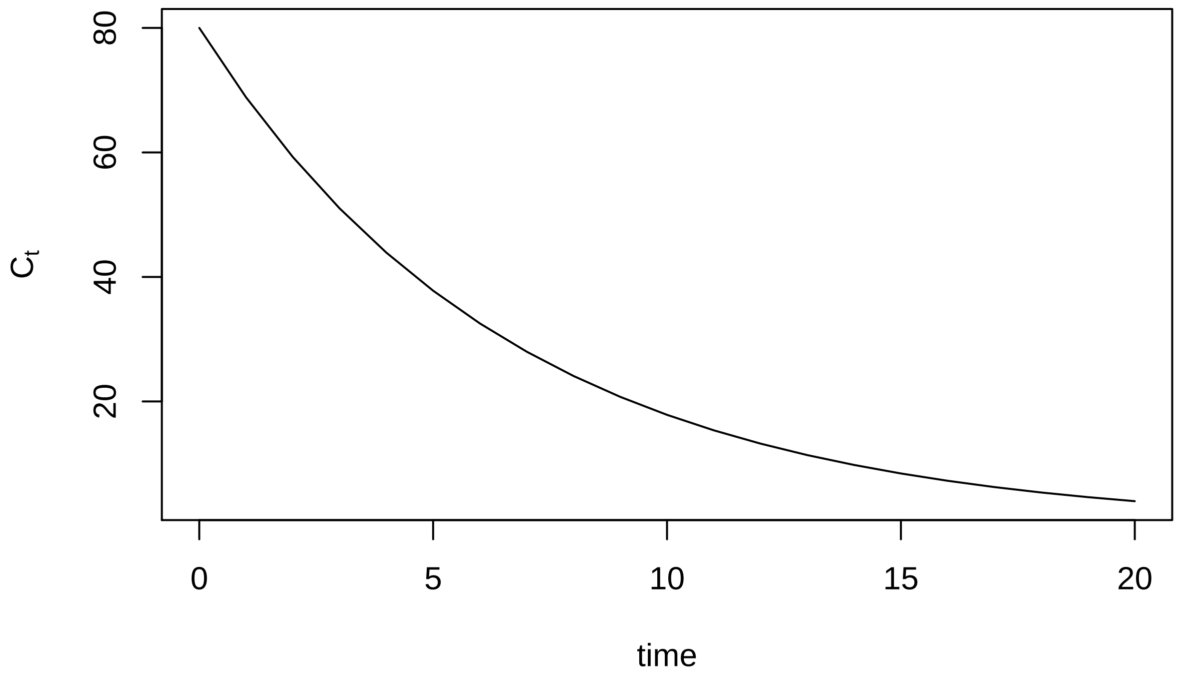
Done. The solver finds the solution and provides to calculate it for the time vector we provided (in this case a sequence from 0 to 20, let's say years).
It stores the results in the object `output`:

```
output
```

```
##    time         C
## 1     0 80.000000
## 2     1 68.856704
## 3     2 59.265533
## 4     3 51.010288
## 5     4 43.904956
## 6     5 37.789327
## 7     6 32.525593
## 8     7 27.995048
## 9     8 24.095564
## 10    9 20.739245
## 11   10 17.850435
## 12   11 15.364013
## 13   12 13.223930
## 14   13 11.381939
## 15   14  9.796525
## 16   15  8.431947
## 17   16  7.257444
## 18   17  6.246540
## 19   18  5.376446
## 20   19  4.627550
## 21   20  3.982969
```

We can plot the output with:

```
plot(output[,1], output[,2], type="l", xlab="time", ylab=expression(paste(C[t])))
```



encapsulating the ODE and the solver into a function we can calibrate

To “calibrate” means we are going to try to find the parameter combination that minimize the error of the model relatively to our target data. In order to do this, we need to write down a function that as output gives the fitness of the solved ODE system with a certain parameter (which we will then vary automatically, you will see later). To do that such function should compare the output of our model with the measured data we are using for calibration. We will then encapsulate in this function the function we wrote above and the solver.

```
#reading the data we want to calibrate on
target_data<-read.csv("Ultuna_LTBF_data.csv")



#putting the data in the right place
Time <- target_data$BF_duration
State <- c(C=target_data$Mean_C.stock_t.ha[1])

library(Metrics) #to include the function to calculate the RMSE
```

```
## Warning: package 'Metrics' was built under R version 4.0.5
```

```
exp_optim<-function(Time, State, Pars) {
  with(as.list(c(State, Pars)), {

    output   <- ode(State, Time, exp_diff, Pars) #here you are using the solver and the ODE function we
                                                 #to calculate a simulation with a certain paramter set
    fitness  <- rmse(output, target_data$Mean_C.stock_t.ha) # here you are calculating the fitness of y

    return(fitness) #this function must return one single value as object
  })
}
```

optimizing the function (A.K.A. calibrating)

We are now going forward and optimizing the function. In this case we are optimizing for one single parameter and we need to use the function `optimize`, much simpler than multi parameter optimizationa algorithms. One quirk of this particular function is the syntax, it needs to optimize a function that has as parameters only `x`, where `x` is the parameter of our objective function. We need therefore to write a small wrapper function like this:

```
exp_optim_wrapper<-function(x){exp_optim(Time, State, Pars=c(k=x))}
```

We can then proceed with the optimization, storing the results in the object `opt_results`:

```
opt_results<-optimize(exp_optim_wrapper,  lower = 0.01, upper = 1)

opt_results
```

```
## $minimum
## [1] 0.0100652
##
## $objective
## [1] 12.26911
```

The object contains two results, a `minimum`, which is the point where the optimized parameter gives the minimum cost (the RMSE in our case), and a `objective`, which is the minimum value reached by our cost (again the RMSE).

We now want to check our optimization, making a simulation with he new optimal value we just found. We first redefine the parameter of our simulation as the output from the optimization:

```
Pars_opt<-c(k=opt_results$minimum)
```

And then proceed to run our model with it, storing the results in the object `simulation`:

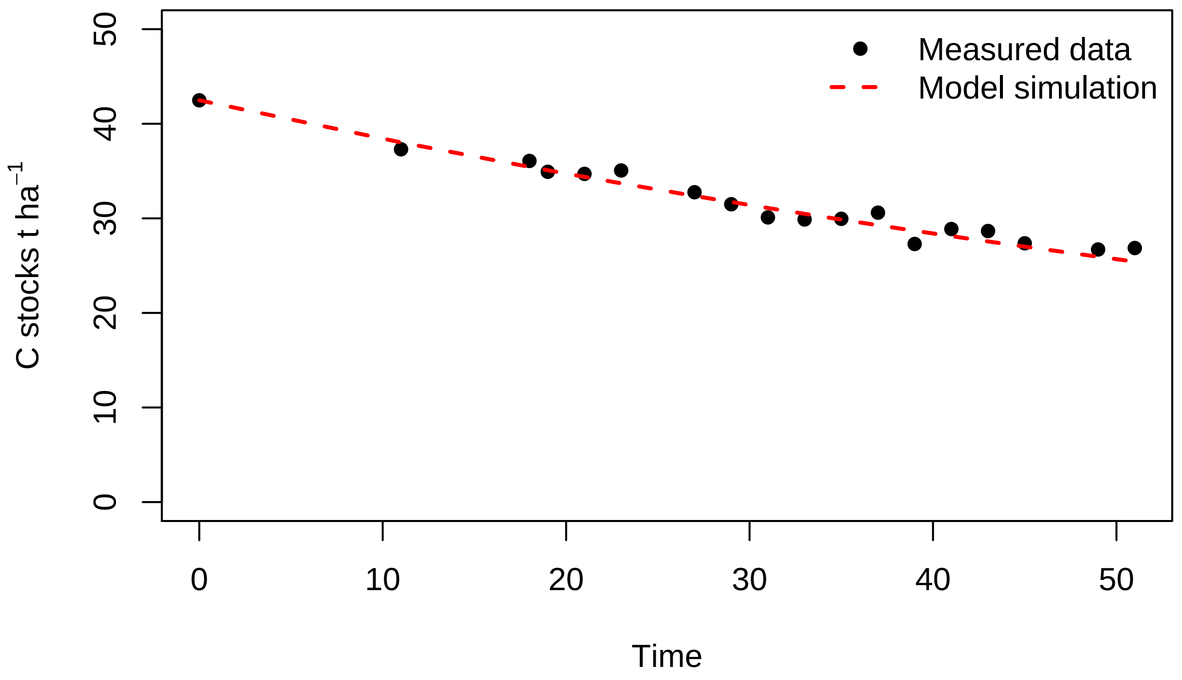
```
simulation<-as.data.frame(ode(State, Time, exp_diff, Pars_opt))

simulation
```

```
##    time        C
## 1     0 42.48000
## 2    11 38.02779
## 3    18 35.44070
## 4    19 35.08577
## 5    21 34.38654
## 6    23 33.70125
## 7    27 32.37136
## 8    29 31.72623
## 9    31 31.09395
## 10   33 30.47427
## 11   35 29.86695
## 12   37 29.27172
## 13   39 28.68836
## 14   41 28.11662
## 15   43 27.55628
## 16   45 27.00710
## 17   49 25.94137
## 18   51 25.42438
```

All seems well, and our function simulates a reasonable trend of SOC decomposition. Let's try to plot it against the measurements:

```
plot(target_data$BF_duration, target_data$Mean_C.stock_t.ha, ylim=c(0,50),
     ylab=expression(paste("C stocks t ",ha^-1)), xlab="Time", pch=16)
lines(simulation$time, simulation$C, lty=2, lwd=2, col="red")
legend("topright", c("Measured data", "Model simulation"), bty="n",
       pch=c(16, NA), lty=c(NA, 2), lwd=c(NA, 2), col=c("black", "red"))
```



Our simulation seems to work pretty well!
Congratulations, you just built and calibrated a model!