Implementing Compartment Models in `SoilR`: the `GeneralModel` Function

Markus Müller* and Carlos A. Sierra†
Max Planck Institute for
Biogeochemistry

April 1, 2012

Abstract

The objective of this vignette is to demonstrate the application range of class `Model` for the implementation of compartment models in `SoilR`. We will start with the most simple running example that focuses on the basic building blocks from a technical, rather abstract point of view.

1 Introduction

A large variety of compartment models can be implemented in `SoilR` with the function `GeneralModel`. This function is the backbone of all the different organic matter decomposition models implemented in `SoilR`; i.e., all other functions implemeting a model are wrappers to this function. In this vignette we show some examples on how to implement different compartment models with different types of input data.

First, we recall that the general model implemented by the function `GeneralModel` is given by the equation

$$\frac{d\mathbf{C}(t)}{dt} = \mathbf{I}(t) + \mathbf{A}(t)\mathbf{C}(t) \tag{1}$$

where $\mathbf{C}(t)$ is a $m \times 1$ vector of carbon stores in m pools at a given time t; $\mathbf{A}$ is a $m \times m$ square matrix containing time-dependent decomposition rates for each pool and transfer coefficients between pools; and $\mathbf{I}(t)$ is a time-dependent column vector describing the amount of inputs to each pool m.

Model structure is mainly defined by the matrix $\mathbf{A}$, which contains the decomposition rates in the main diagonal and transfer among pools in the off-diagonal.

For implementing the model, it is possible that different types of input data are available. For example, litter inputs can be defined as a constant over time, as a function that depends on other variables such as temperature, or as a time series of observed values. Similarly, the values of $\mathbf{A}$ can be either constant, generated by a function, or a `dataframe` of observed values. We will explore these different possibilities in the following examples.

*mamueller@bgc-jena.mpg.de
†csierra@bgc-jena.mpg.de

2 Abstract example

Consider the following three pool model with conection in series

$$\frac{d\mathbf{C}(t)}{dt} = \begin{pmatrix} 0.05 \\ 0 \\ 0 \end{pmatrix} + \begin{pmatrix} -0.39 & 0 & 0 \\ 0.1 & -0.35 & 0 \\ 0 & 1/3 & -0.33 \end{pmatrix} \begin{pmatrix} C_1 \\ C_2 \\ C_3 \end{pmatrix}, \qquad (2)$$

and initial conditions $\mathbf{C}(t=0) = \{0.5, 0.5, 0.5\}^T$.

To implement this model, first we load the package.

```
> library("SoilR")
```

Now we create an object of class `TimeMap` to represent the coefficient matrix $\mathbf{A}$ as a function of time. In our simple example this will be just a constant function, but we are nethertheless forced to supply information about the time-domain where this function is valid. Why this requirement is sensible will be demonstrated later in the document where the time-domain data will be used to check automatically for an inconsistent model.

```
> n=3;
> t_start=1;t_end=2
> At=TimeMap.new(
+   t_start,
+   t_end,
+   function(t0){
+         matrix(nrow=n,ncol=n,byrow=TRUE,
+           c(-0.39,    0,    0,
+              0.1, -0.35,    0,
+              0,    1/3,   -0.33)
+         )
+   }
+ )
```

Now we do the same thing for the inputrate as a function of time. We also choose the simplest possible case, which is a constant function of time producing a value of 0.05 .

```
> inputFluxes=TimeMap.new(
+     t_start,
+     t_end,
+     function(t0){matrix(nrow=3,ncol=1,c(0.05,0,0))}
+ )
```

Then we define the times where we want to compute the C-content and the C release.

```
> tn=500
> timestep=(t_end-t_start)/tn
> t=seq(t_start,t_end,timestep)
```

We also need to specify the initial values of C content in the different pools.

```
> c0=c(0.5, 0.5, 0.5)
```

We can now assemble the `Model` object

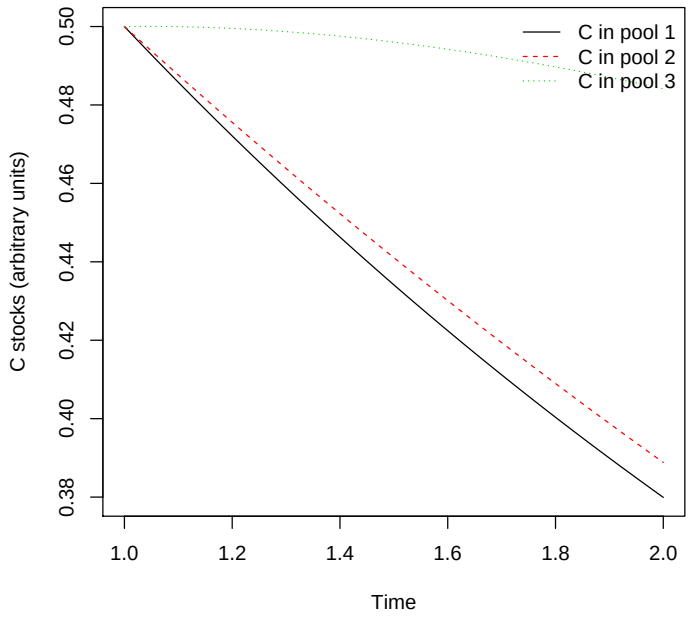
```
> mod=GeneralModel(t,At,c0,inputFluxes)
```

and ask it several questions, for instance the C content:

```
> Y_c=getC(mod)
```

which we can plot

```
> lt1=1;  lt2=2; lt3=3
> col1=1; col2=2; col3=3
> plot(t,
+          Y_c[, 1],
+          type="l",
+          lty=lt1,
+          col=col1,
+          ylab="C stocks (arbitrary units)",
+          xlab="Time",
+          ylim=c(min(Y_c),max(Y_c))
+ )
> lines(t,Y_c[,2],type="l",lty=lt2,col=col2)
> lines(t,Y_c[,3],type="l",lty=lt3,col=col3)
> legend(
+     "topright",
+     c("C in pool 1", "C in pool 2", "C in pool 3"),
+     lty=c(lt1,lt2,lt3),
+     col=c(col1,col2,col3),
+     bty="n"
+ )
```

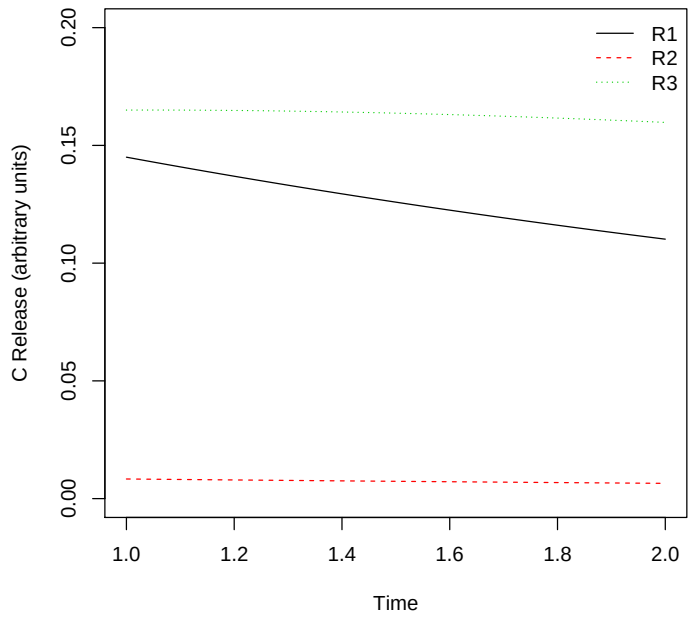


Amount of carbon in the three different pools of the model described by equation (1).

We also could ask for the release flux as a function of time.

```
> Y_rf=getReleaseFlux(mod)

>     plot(t,Y_rf[,1],type="l",lty=lt1,col=col1,
+     ylab="C Release (arbitrary units)",
+     xlab="Time", ylim=c(0,0.2))
>     lines(t,Y_rf[,2],lt2,type="l",lty=lt2,col=col2)
>     lines(t,Y_rf[,3],type="l",lty=lt3,col=col3)
>     legend("topright",c("R1","R2","R3"),lty=c(lt1,lt2,lt3),
+            col=c(col1,col2,col3), bty="n")
```

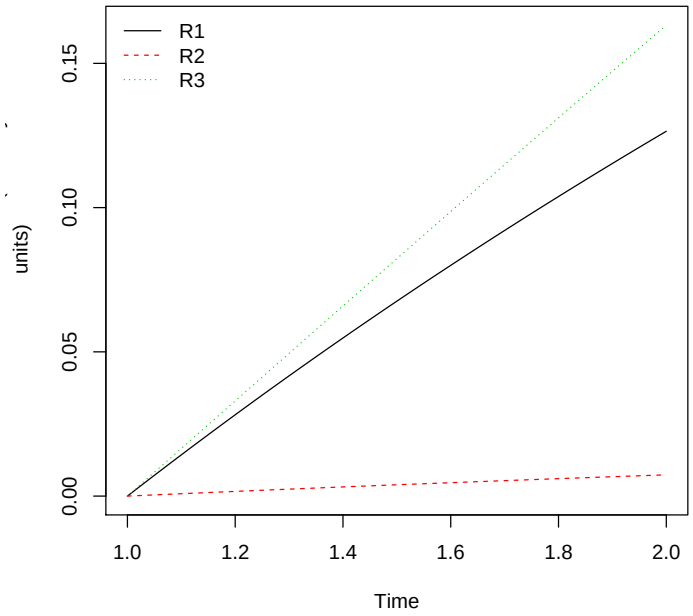


Amount of carbon released from the three different pools of the model of equation (1).

Similarly, it is possible to ask the `Model` object for the accumulated release of carbon.

```
> Y_r=getRelease(mod)

>     plot(t,Y_r[,1],type="l",lty=lt1,col=col1,
+     ylab="Accumulated Release (arbitrary
+     units)", xlab="Time",
+     ylim=c(min(Y_r),max(Y_r)))
>     lines(t,Y_r[,2],lt2,type="l",lty=lt2,col=col2)
>     lines(t,Y_r[,3],type="l",lty=lt3,col=col3)
>     legend("topleft",c("R1","R2","R3"),lty=c(lt1,lt2,lt3),
+            col=c(col1,col2,col3), bty="n")
```



Cummulative amount of carbon released from the three pools of the model described by equation (1).

2.1 Outlook

At the beginning we mentioned that the given example is very abstract and also very simple. Actually this means that there is much more abstraction than is necessary for this simple example. This up to now unused abstraction can be represented by the use of an object of class `TimeMap`. The objects of class `TimeMap` are made of a function definition and the time domain of this function. Two questions arise.

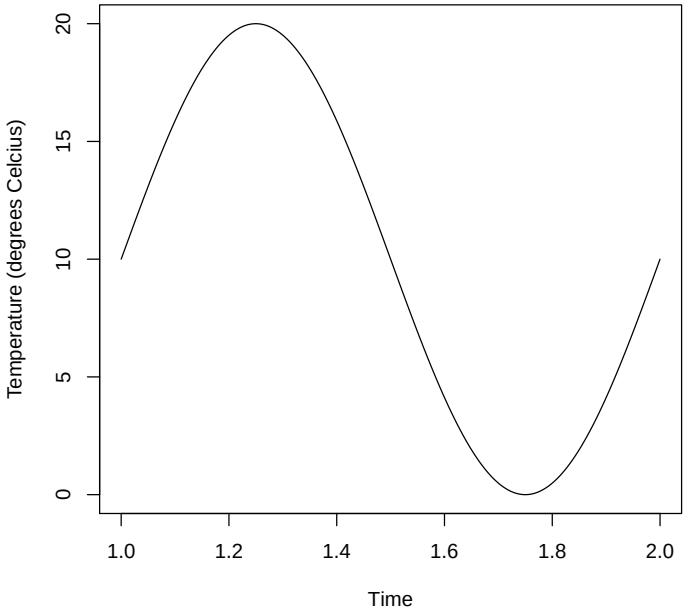
1. Why do we need a function definition?
2. Why do we need a explicit specification of the computational domain?

We will answer the first question in the next section.

3 Application including moisture and temperature dependence

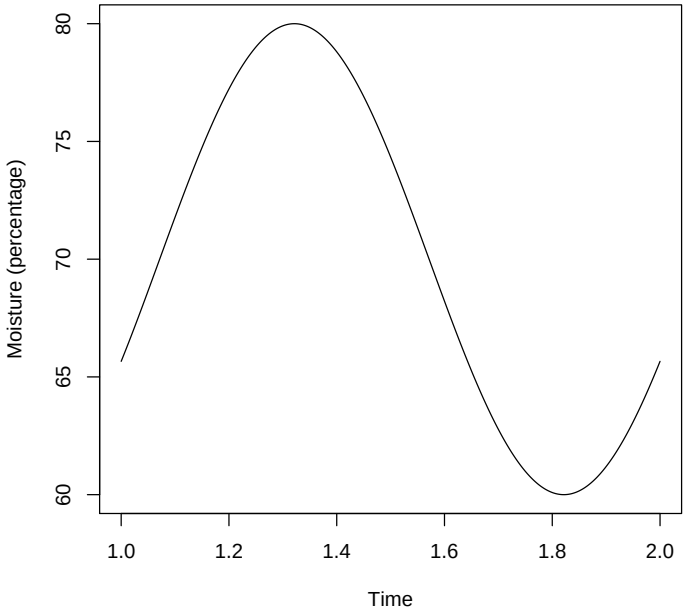
We now create a time dependent coeffictient matrix where we give moisture and temperature as functions of time and create the coefficients as functions of moisture and temperature. We will give the inputrate as a periodic function. Let's start with a somewhat arbitrary definition of a daily temperature curve.

```
> Temp=function(t0){ #Temperature in Celsius
+     T0=10   #anual average temperature in Celsius degree
+     A=10    #Amplitude in K
+     P=1     #Period in years
+     T0+A*sin(2*pi*P*t0)
+ }
> plot(t,Temp(t),xlab="Time",ylab="Temperature (degrees Celcius)",type="l")
```



and something similar arbirtrary for moisture.

```
> Moist=function(t0){#Moisture in percent
+     W0=70       #average moisture in percent
+     A=10        #Amplitude of change
+     P=1         #Period in years
+     ps=pi/7     #phase shift
+     W0+A*sin(2*pi*P*t0-ps)
+ }
> plot(t,Moist(t),xlab="Time",ylab="Moisture (percentage)",type="l")
```



Now we choose a function for determining the temperature effects on decomposition rates. Actually we have $A(t) = \xi(t)A_0$ with a constant A_0 and $\xi(t)$ is given by the product of functions `fT.Daycent1` and `fW.Daycent2` : where we have to take into account that `fW.Daycent2` returns a dataframe from which we will have to extract the decay coefficient influencing part first.

```
> xi=function(t0){
+     fT.Daycent1(Temp(t0))*as.numeric(fW.Daycent2(Moist(t0))["fRWC"])
+ }
```

We define A_0 and combine it with ξ to the complete the `TimeMap` object.

```
> A_0=matrix(nrow=n,ncol=n,byrow=TRUE,
+   c(-0.2,    0,    0,
+      0.1, -0.7,    0,
+      0,    1/2,   -0.5)
+ )
> A_t=TimeMap.new(
+   t_start,
+   t_end,
+   function(t0){xi(t0)*A_0}
+   )
```

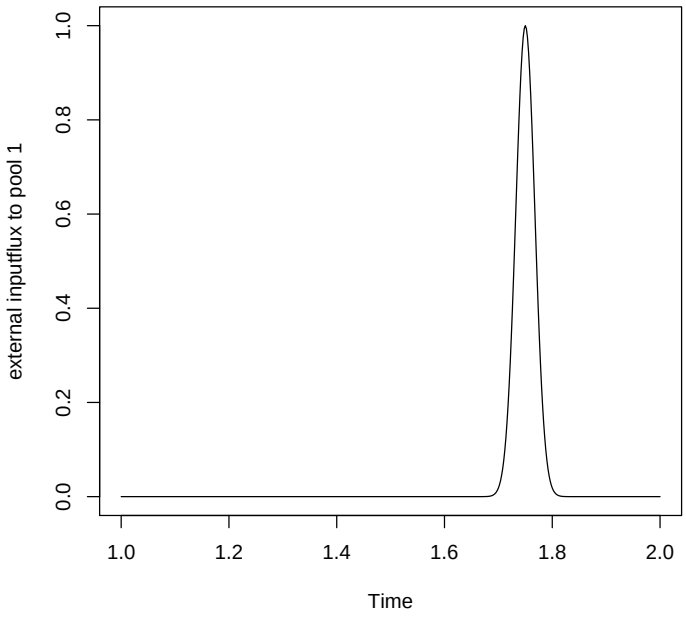
We define the input fluxes explicitly as:

```
> inputFluxes=function(t0){
+     t_peak1=0.75
+     t_peak2=1.75
+     matrix(nrow=3,
+            ncol=1,
+            c(
```

```
+              exp(-((t0-t_peak1)*40.0)^2)+exp(-((t0-t_peak2)*40.0)^2),
+              0,
+              0
+            )
+     )
+ }
> inputFluxes_tm=TimeMap.new(
+     t_start,
+     t_end,
+     inputFluxes
+ )
```

Since `inputFluxes` is a matrix valued function we have to evaluate it more explicitly to be able to plot it. We chose the input to the first pool here.

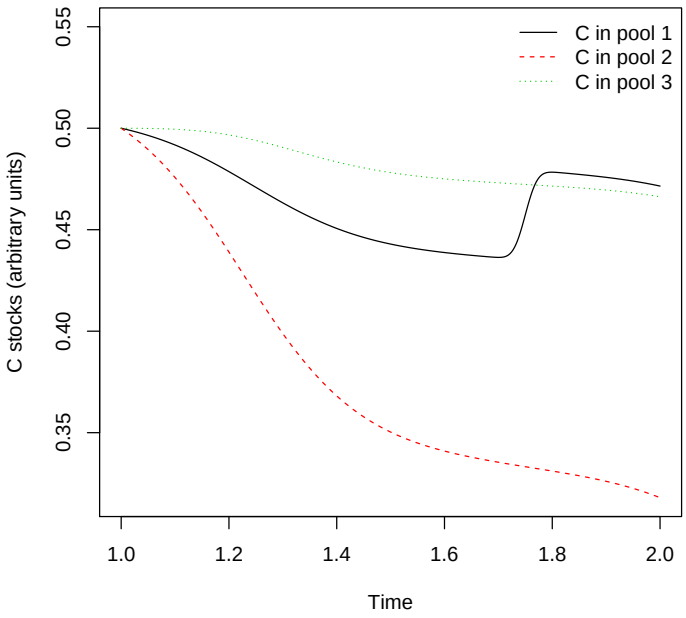
```
> ifl_1=matrix(nrow=1,ncol=length(t))
> for (i in 1:length(t)){ifl_1[i]=inputFluxes(t[i])[1]}
> plot(t,ifl_1,xlab="Time",ylab="external inputflux to pool 1",type="l")
```



We can now combine the time dependent functions for the coefficients and the inputrates to a Model.

```
> mod=GeneralModel(t,A_t,c0,inputFluxes_tm)
> Y_c=getC(mod)
>       plot(t,Y_c[,1],type="l",lty=lt1,col=col1,
+            ylab="C stocks (arbitrary units)",
+            xlab="Time",
+            ylim=c(min(Y_c),1.1*max(Y_c))
+       )
>       lines(t,Y_c[,2],type="l",lty=lt2,col=col2)
>       lines(t,Y_c[,3],type="l",lty=lt3,col=col3)
```

```
>       legend(
+          "topright",
+          c("C in pool 1",
+            "C in pool 2",
+            "C in pool 3"
+          ),
+          lty=c(lt1,lt2,lt3),
+          col=c(col1,col2,col3), bty="n"
+       )
```



4 Real data combined with synthetic functions

Until now all functions were given explicitly, but sometimes some of them will be given by observational data that have to be interpolated. Of course you could just produce the interpolating function and proceed as in the previous examples. There is a shortcut however. Objects of class `TimeMap` can be produced directly from `dataframes` eliminating the need to specify the time range explicitly. There are some small example plain text files in the `inst/extdata` directory of the package which contain the data for time dependend input fluxes. (Although every `dataframe` could be used we read the data from plain text files here to emphasize the fact that in the application of `SoilR` this data will usually be provided by the user generally in some text based format.) We will read the data from these files in a `dataframe` and then create an object of class `TimeMap` automatically using a function for this purpose. It will produce an interpolation of the data, and will also determine t_{start} and t_{end} previously used in the explicit

creation of TimeMap objects.

First we define some filenames and paths:

```
>     fn="inputFluxForVignetteGeneralModel"
>     fn2="inputFluxForVignetteGeneralModelShort"
>     subdir=file.path(system.file(package="SoilR"),"extdata")
>     p=file.path(subdir,fn)
>     p2=file.path(subdir,fn2)
```

You can have a look at the example files by uncommenting:

```
> #file.show(p)
```

We will read the first file and create an object of class TimeMap automatically.

```
> dfr=read.csv(p)
> iTm=TimeMap.from.Dataframe(dfr)
```

We have the usual ingredients for a Model object and can create it.

```
> mod=GeneralModel(t,A_t,c0,iTm)
```

4.1 Safety net

We will now show what happens if we try to extrapolate a given dataset by accident. To show this, we have created a different file containing a time series of an input flux with a smaller time range compared to the explicit time argument to GeneralModel.

```
> dfr2=read.csv(p2)
> iTm=TimeMap.from.Dataframe(dfr2)
```

If we try to create a Model from this (by removing the comment in the next example) we will get an error message.

```
> #mod=GeneralModel(t,A_t,c0,iTm)
```

This is because the dataset we used does not contain data valid for the times we required in the first argument and also in the other objects of class TimeMap that would be used to create the Model. To see this we can inquire the TimeMap objects for the time range:

```
> getTimeRange(iTm)

t_min t_max
 1.25  1.75

> min(t)

[1] 1

> max(t)

[1] 2
```

Note that the time range of iTm is smaller than the range we required. If we only have data for this small range we only have to decrease also the timevector. Actually the Model allows different time ranges for all the components as long as the time argument specifies only times in the range that is covered by all TimeMap objects contributing. In other words the requested times must be part of the intersection of the ranges of all TimeMap objects present. This means that we could repair the situation very easily by just adjusting the t argument, and the model will no longer refuse to be built.

```
> ts2
[1] 1.25
> te2
[1] 1.75
> t=seq(ts2,te2,timestep)
> mod=GeneralModel(t,A_t,c0,iTm)
```