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

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

November 14, 2023

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 implementing 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.

It is possible that different types of input data are available to define the model. 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 connection in series

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

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")
```

```
## Loading required package: deSolve
##
## Attaching package: 'SoilR'
## The following object is masked from 'package:deSolve':
##
##     euler
```

Now we create an object of class `ConstLinDecompOp` to represent the coefficient matrix $\mathbf{A}$.

```
n=3;
t_start=1;t_end=2
At=ConstLinDecompOp(
        mat=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=ConstInFluxes(
    map=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"
        )
```

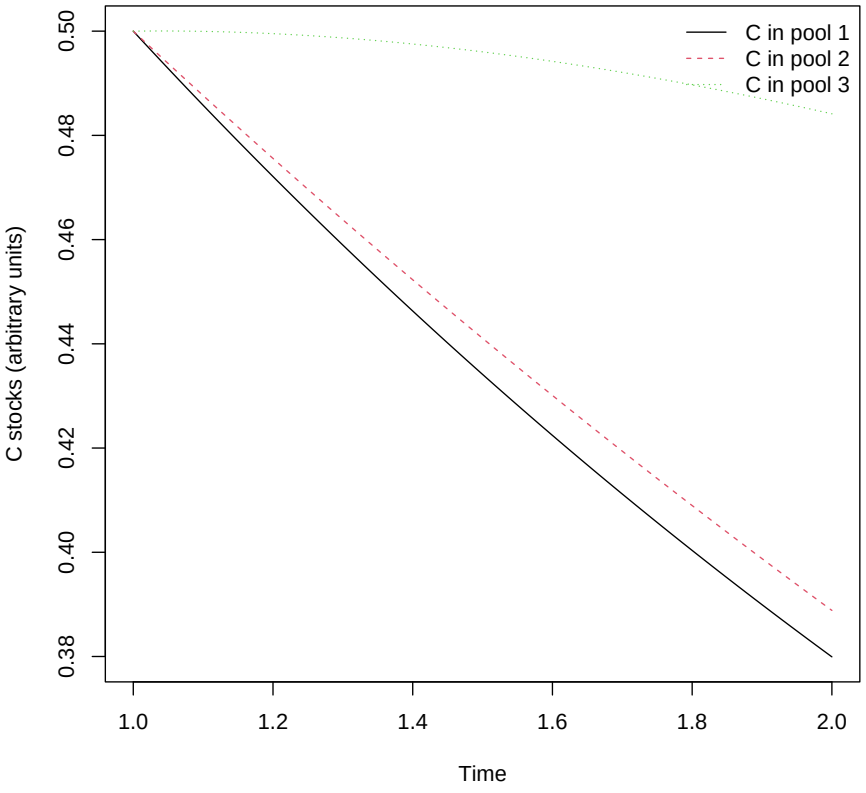


Figure 1: Amount of carbon in the three different pools of the model described by equation (1).

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")
```

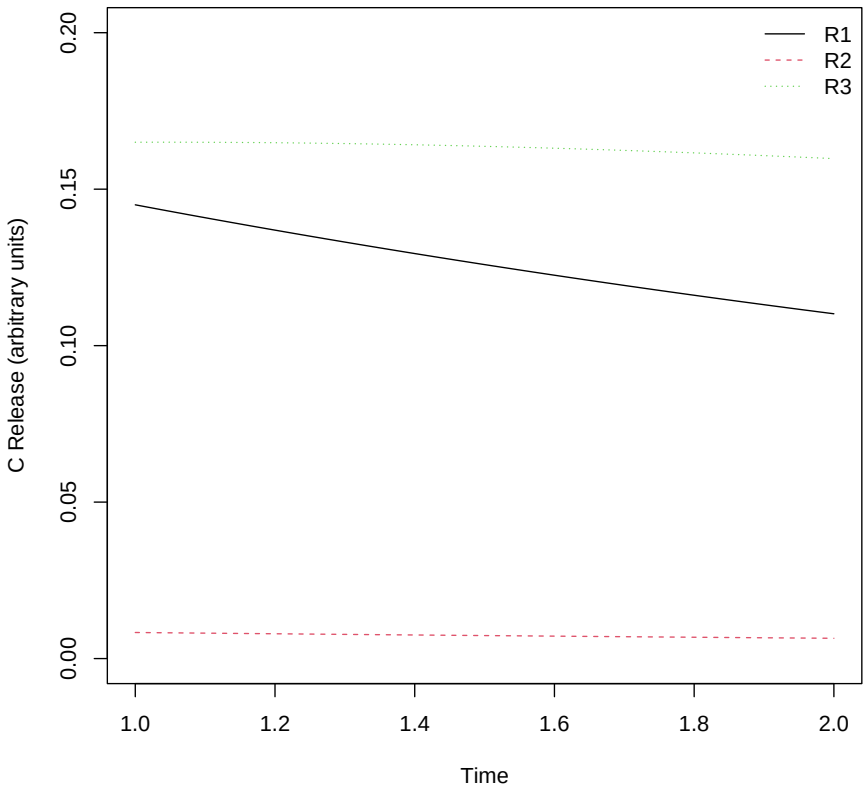


Figure 2: Amount of carbon released from the three different pools of the model of equation (1).

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

```
Y_r=getAccumulatedRelease(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")
```

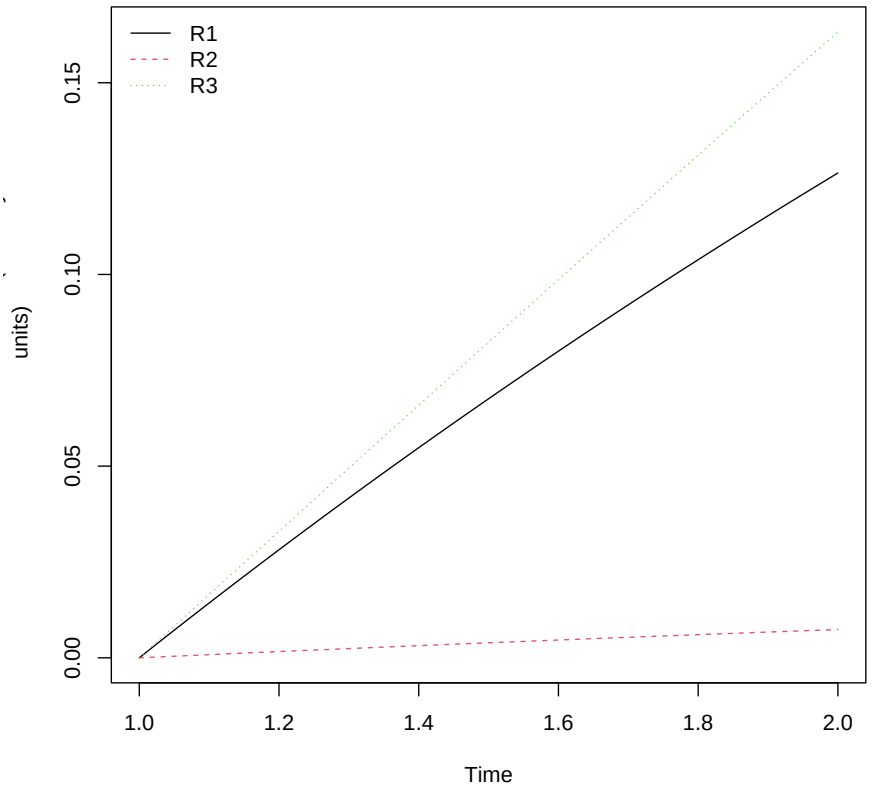


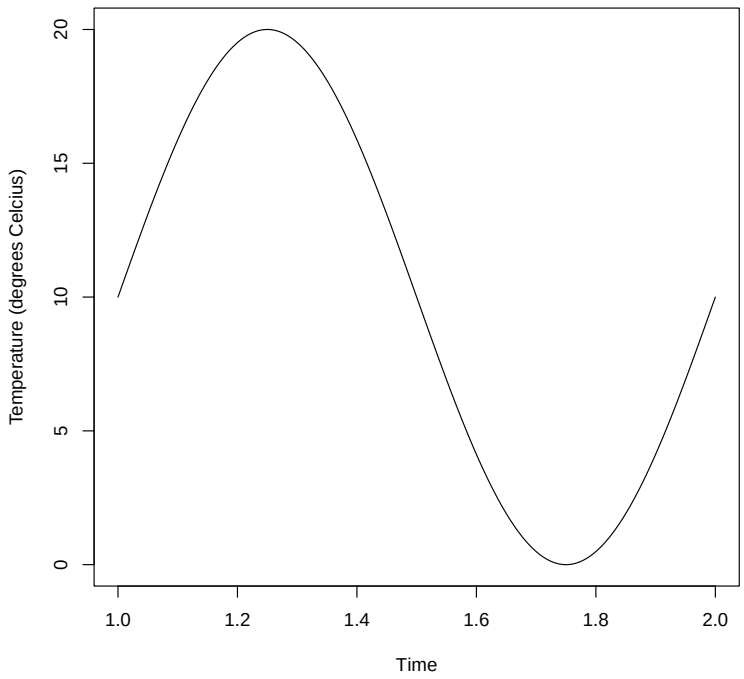
Figure 3: Cummulative amount of carbon released from the three pools of the model described by equation (1).

3 Application including moisture and temperature dependence

We now create a time dependent coefficient 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 arbitrary 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)
}
```

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 `BoundLinDecompOp` .

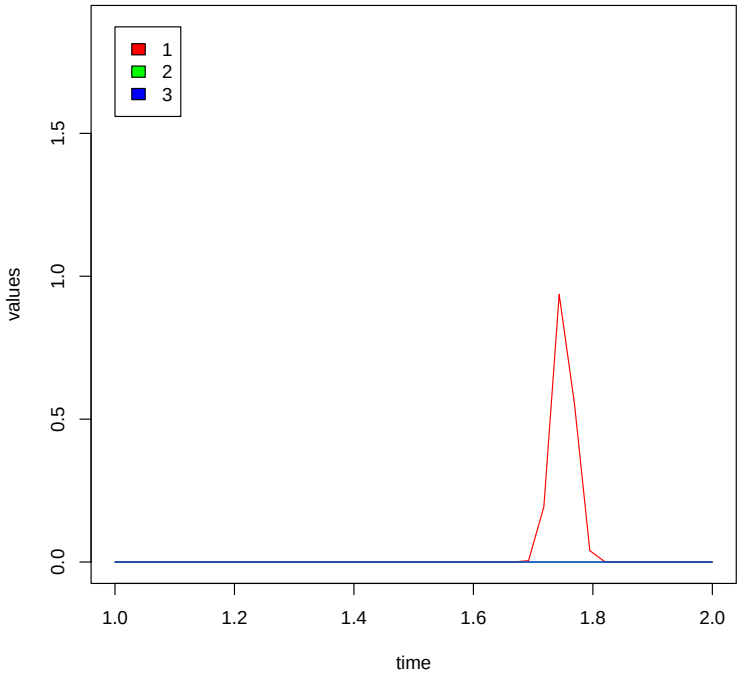
```
A_0=matrix(nrow=n,ncol=n,byrow=TRUE,
  c(-0.2,    0,    0,
     0.1, -0.7,    0,
     0,    1/2,   -0.5)
)
A_t=BoundLinDecompOp(
  function(t0){xi(t0)*A_0},
  t_start,
  t_end
  )
```

We define the input fluxes explicitly:

```
inputFluxes<-function(t0){
    t_peak1=0.75
    t_peak2=1.75
    c(
        exp(-((t0-t_peak1)*40.0)^2)
        +exp(-((t0-t_peak2)*40.0)^2)
        ,0
        ,0
    )
}
inputFluxes_tm<-BoundInFluxes(
    inputFluxes,
    t_start,
    t_end
)
```

We can get a quick overview plot by typing:

```
plot(inputFluxes_tm)
```

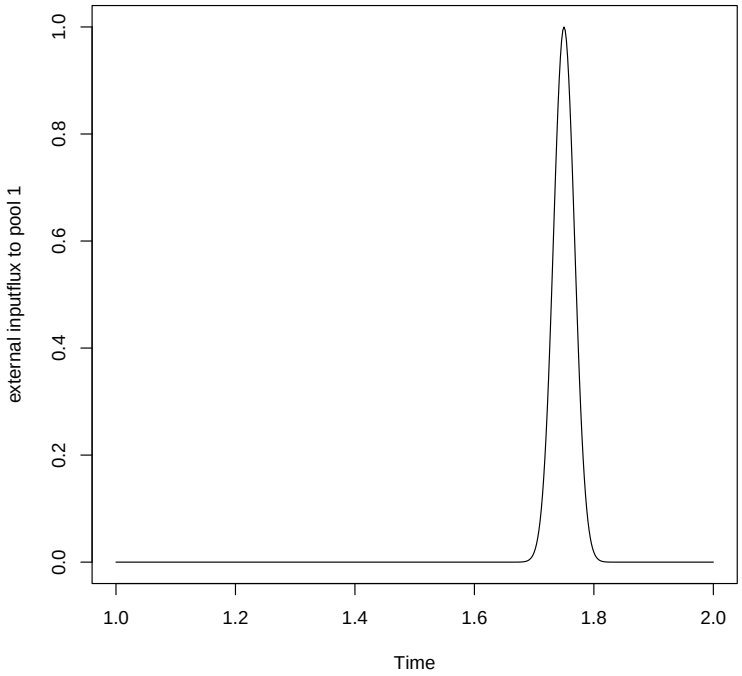



```
## $rect
## $rect$w
## [1] 0.1101875
##
## $rect$h
## [1] 0.3135746
##
## $rect$left
## [1] 1
##
## $rect$top
## [1] 1.872737
##
##
## $text
## $text$x
## [1] 1.07875 1.07875 1.07875
##
## $text$y
## [1] 1.794343 1.715950 1.637556
```

but since **inputFluxes** is a vector valued function we can also evaluate it more explicitly for one pool. We chose the input to the first pool here.

```
f=getFunctionDefinition(inputFluxes_tm)
ifl_1=matrix(nrow=1,ncol=length(t))
```

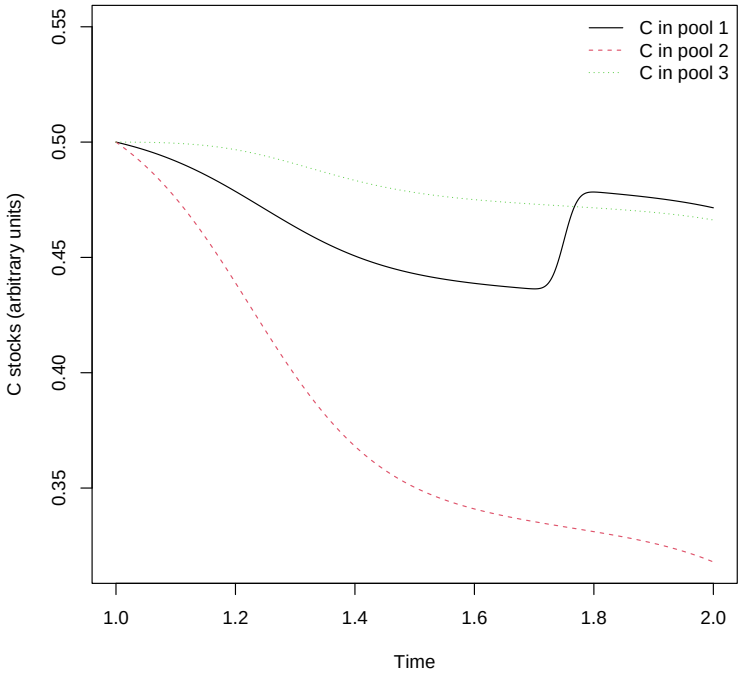
```
for (i in 1:length(t)){ifl_1[i]=f(t[i])[1]}
plot(
    t
    ,ifl_1
    ,xlab="Time"
    ,ylab="external inputflux to pool 1"
    ,type="l"
)
```



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 classes `BoundInFluxes` or `BoundLinDecompOp` 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 dependent 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 the appropriate class automatically using specialized constructor functions for this purpose. They will produce an interpolation of the data, and will also determine t_{start} and t_{end} previously used in the explicit creation of the 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)
```

```
# 1.)
# To make it simpler to keep consistency between the
# data and the vignette
# we provide the code to create the datafiles here
# To this end we  now create a dataframe using the same time
# range as all the examples before
t_peak1=0.75
t_peak2=1.75
df=data.frame(
    "time"=t
    ,"inputFlux"=exp(-((t-t_peak1)*40.0)^2)+
            exp(-((t-t_peak2)*40.0)^2)
)
# Additionally we create a second dataset spanning a smaller
# time range which we will use later to demonstrate
# the safety net provided by the use of the classes.
d=t_end-t_start
ts2=t_start+d/4
te2=t_end-d/4
t2=seq(ts2,te2,timestep)
i2=exp(-((t2-t_peak1)*40.0)^2)+
        exp(-((t2-t_peak2)*40.0)^2)
df2=data.frame(t2,i2)
# temporary uncomment the next lines to write the file to
# the appropriate source dir and comment it out again
# before you check in the vignette because it will not pass
# the package check otherwise
#mmdir="~/SoilR/RPackages/SoilR/pkg/inst/extdata"
#write.csv(df ,row.names=FALSE,file.path(mmdir,fn))
#write.csv(df ,row.names=FALSE,p)
#write.csv(df2,row.names=FALSE,file.path(mmdir,fn2))
#write.csv(df2,row.names=FALSE,p2)
```

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

```
dfr=read.csv(p)
iTm=BoundInFluxes(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=BoundInFluxes(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 that would be used to create the `Model`. To see this we can interrogate the 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 objects contributing. In other words the requested times must be part of the intersection of the ranges of all 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)
```