Forcing functions based on Data in R

Reader Accompanying the Course Reaction Transport Modelling in the Hydrosphere

Karline Soetaert and Lubos Polerecky, Utrecht University

April 2021

Abstract

Forcing functions play an important role in dynamic models. Here we show how to implement in R a forcing function that is based on a time-series data set rather than an explicit function of time. We illustrate this using the NPZD model, where we impose interpolated time-series measurements of light intensity as the forcing function modulating photosynthetic nitrogen uptake by phytoplankton.

Forcing functions

Forcing functions are important factors that affect the output of dynamic mechanistic models. Rather than being explicitly modelled as state variables, forcing functions are *imposed* on the model as conditions that modulate the strength of interactions between state variables.

A good example of a forcing function is the intensity of the photosynthetically active solar radiation (PAR). As we have seen in the NPZD model developed in class, the PAR intensity is not included as a state variable. However, because it drives algal photosynthesis, it does affect the model output.

In the NPZD model developed in class, we imposed the PAR intensity as a sine wave with the period of one year (to mimic seasonal variation). While this may be a good strategy to test a model, it is better to use *measurements* of the PAR intensities as a forcing function to make the model more realistic. Below, we illustrate how to do this in R.

We start by reproducing the NPZD2 model developed in class. However, instead of implementing the forcing function by an explicit formula within the model function, we define the forcing function as an additional *input parameter* of the model function (`fPAR`). This allows the forcing function to be implemented *independently* (i.e., outside) of the model function. Then, within the model function, the light intensity at the current time (`t`) is calculated simply by calling the input function as `fPAR(t)`.

```
require(deSolve)  # package with solution methods

# state variables, units = molN/m3 or molN/m2 (BOT_DET)
state  <- c(DIN=0.010, PHYTO=0.0005, ZOO=0.0003, DET=0.005, BOT_DET=0.005)

# parameters
parms <- c(
  depth              = 10,        # [m] water depth
  rUptake            = 1.0,       # [/day]
  ksPAR              = 140,       # [uEinst/m2/s]
  ksDIN              = 1.e-3,     # [molN/m3]
  rGrazing           = 1.0,       # [/day]
  ksGrazing          = 1.e-3,     # [molN/m3]
  pFaeces            = 0.3,       # [-]
  rExcretion         = 0.1,       # [/day]
```

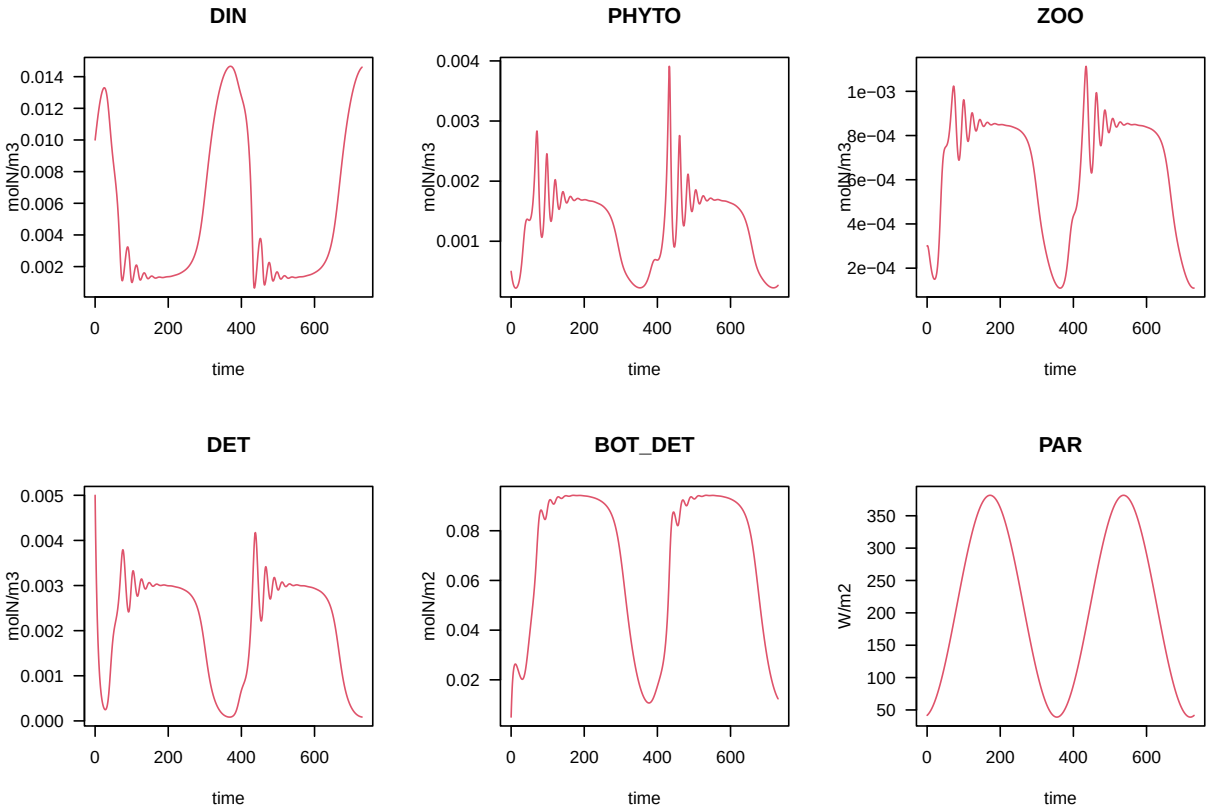
```
  rMortality         = 400,       # [/(molN/m3)/day]
  rMineralisation    = 0.05,      # [/day]
  sinkVelocity       = 1          # [m/day]
)

# Model formulation; Note the new input argument: function fPAR
NPZD2 <- function(t, state, parameters, fPAR){

  with(as.list(c(state, parameters)),{

  # PAR in the middle of the water depth; extinction coefficient 0.05/m
  # The forcing function is specified by the function fPAR passed as input!
  PAR <- fPAR(t)*exp(-0.05*depth/2)

  # Rate expressions - in units of [molN/m3/day] or [molN/m2/d]
  DINuptake <- rUptake * PAR/(PAR+ksPAR) * DIN/(DIN+ksDIN)*PHYTO # molN/m3/d
  Grazing        <- rGrazing * PHYTO/(PHYTO+ksGrazing) * ZOO     # molN/m3/d
  Faeces         <- pFaeces * Grazing                            # molN/m3/d
  ZooGrowth      <- (1-pFaeces) * Grazing                        # molN/m3/d
  Excretion      <- rExcretion * ZOO                             # molN/m3/d
  Mortality      <- rMortality * ZOO * ZOO                       # molN/m3/d
  Mineralisation <- rMineralisation * DET                        # molN/m3/d
  SinkDet        <- sinkVelocity * DET                           # molN/m2/d !
  SinkPhy        <- sinkVelocity * PHYTO                         # molN/m2/d !
  BotMin         <- rMineralisation * BOT_DET                    # molN/m2/d !

  # Mass balances [molN/m3/day]
  dDIN.dt     <- Mineralisation + Excretion - DINuptake + BotMin/depth # molN/m3/d
  dPHYTO.dt   <- DINuptake - Grazing  - SinkPhy/depth                  # molN/m3/d
  dZOO.dt     <- ZooGrowth - Excretion - Mortality                     # molN/m3/d
  dDET.dt     <- Mortality - Mineralisation + Faeces - SinkDet/depth # molN/m3/d
  dBOT_DET.dt <- SinkDet + SinkPhy - BotMin                            # molN/m2/d !

  return (list(c(dDIN.dt, dPHYTO.dt, dZOO.dt, dDET.dt, dBOT_DET.dt),
               TotalN=(DIN+PHYTO+ZOO+DET)*depth + BOT_DET,  # molN/m2
               PAR=PAR))
  })
}  # end of the model function
```

This approach means that we need to implement the function `fPAR` *before* we can actually solve the model. Initially, we implement `fPAR` as a sine function, similar to the NPZD2 model developed in class:

```
fPARsine <- function(t)
  return( 0.5*(540+440*sin(2*pi*(t-81)/365)) )
```

To solve the model using the `ode` solver, all we need to do now is to pass the `fPARsine` function as the value of the input argument `fPAR`. During the calculation, `ode` will pass the `fPARsine` function to the model function `NPZD2`.

```
outtimes <- seq(from=0, to=2*365, length.out=1000)
out <- ode(y=state, parms=parms, func=NPZD2, times=outtimes,
           fPAR=fPARsine)  # we need to pass the light function
plot(out, mfrow=c(2,3), lty=1, lwd=1, las=1, col=2,
     which=c("DIN", "PHYTO", "ZOO", "DET", "BOT_DET", "PAR"),
     ylab=c(rep("molN/m3",times=4), "molN/m2", "W/m2"))
```



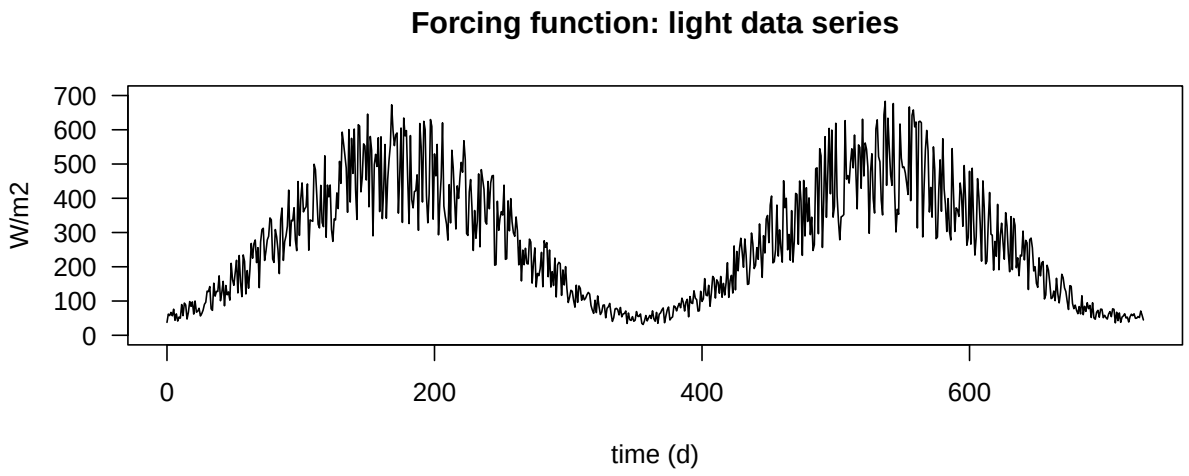
Forcing function based on data series

Assume that we have the following dataset describing the daily average of the light intensity over 2 years:

```
head(PARdata, n=3)
```

```
##      time    Light
## [1,]    0 37.66929
## [2,]    1 61.85253
## [3,]    2 57.44711
```

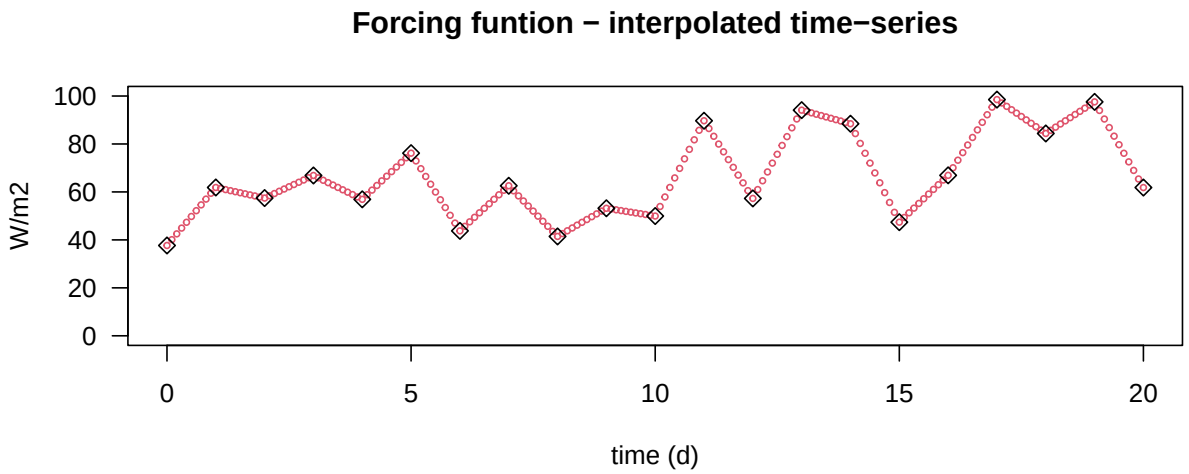
```
plot(PARdata, type="l", main="Forcing function: light data series",
     ylab="W/m2", las=1, xlab="time (d)", ylim=c(0, 700))
```



These data are presented at intervals of one day. However, during the model run, the values will also be required at other times.[1] We estimate these values at any time encompassed in the time series by *interpolating* the available data by a linear function. To do this, we use the R function `approxfun`, which takes the time-series data as input and returns the y-value for *any* x-value.

In the code below, we define a new forcing function (`fPARdata`) and illustrate its output at 0.1 day intervals for the first 20 days.

```
fPARdata <- approxfun(x=PARdata)
tt <- seq(from=0, to=20, by=0.1)
plot(tt, fPARdata(tt), main="Forcing funtion - interpolated time-series",
     xlab="time (d)", ylab="W/m2", las=1, ylim=c(0, 100), col=2, cex=0.5)
points(PARdata, col=1, pch=5)
```

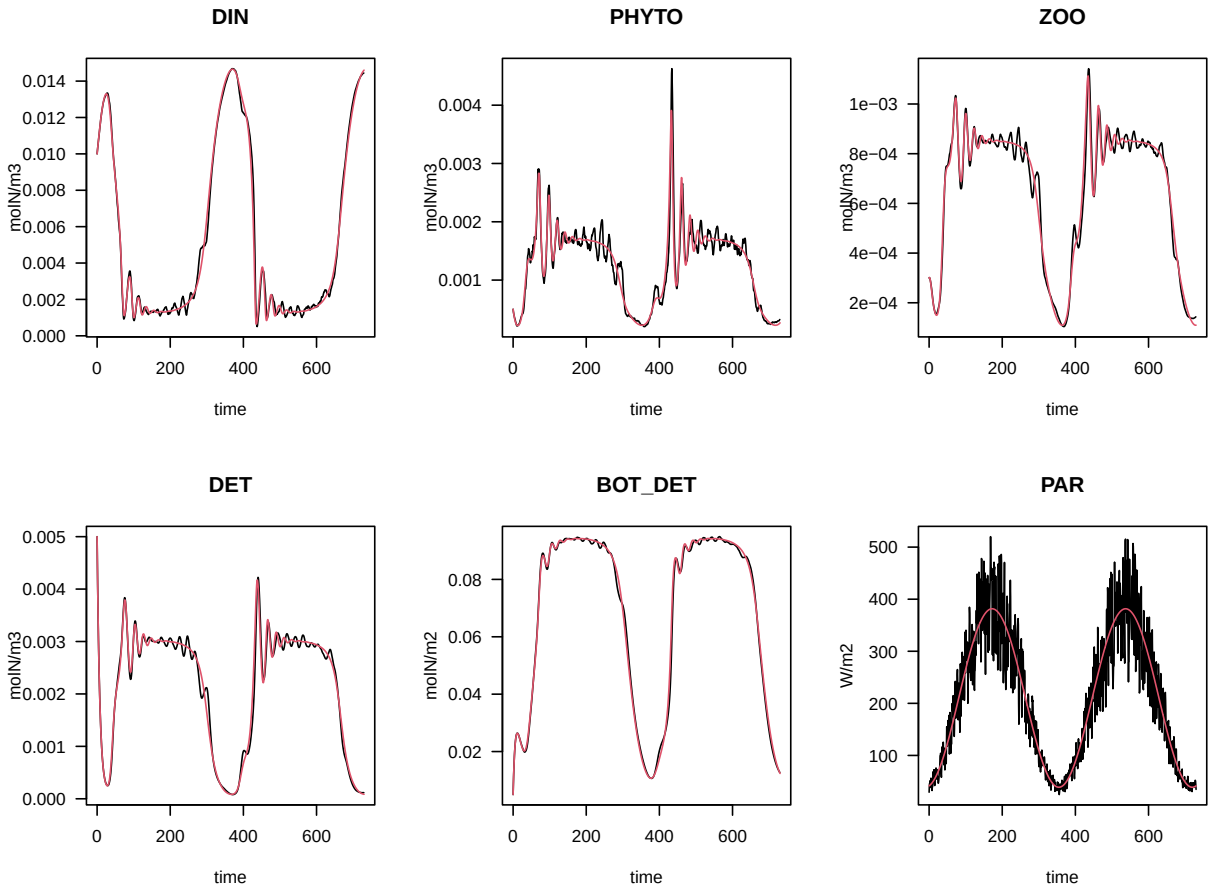


We can now solve the NPZD2 model again, but now passing the forcing function based on the data (`fPARdata`) rather than using the sine wave. The results below show that in addition to the rapid fluctuations induced be feedback loops due to interactions among state variables, there are also fluctuations imposed by the rapidly

[1] The solvers from the deSolve package dynamically adapt the time step, so it is impossible to know in advance at which times the value of the forcing function will be required.

fluctuating light intensities.

```
outDat <- ode(y=state, parms=parms, func=NPZD2, times=outtimes,
              fPAR=fPARdata) # now we pass the data-based function
plot(outDat, out, mfrow=c(2,3), lty=1, lwd=1, las=1,
     which=c("DIN", "PHYTO", "ZOO", "DET", "BOT_DET", "PAR"),
     ylab=c(rep("molN/m3",times=4), "molN/m2", "W/m2"))
```



You can learn more about how to impose forcing functions by reading the help file that is part of the deSolve package:

```
?forcings
```

References

R Core Team (2020). R: A language and environment for statistical computing. R Foundation for Statistical Computing, Vienna, Austria. URL https://www.R-project.org/.

Karline Soetaert, Thomas Petzoldt, R. Woodrow Setzer (2010). Solving Differential Equations in R: Package deSolve. Journal of Statistical Software, 33(9), 1–25. URL http://www.jstatsoft.org/v33/i09/ DOI 10.18637/jss.v033.i09