Visualization of Dynamic Outputs from a 1D Reaction Transport Model in R

Reader Accompanying the Course Reaction Transport Modelling in the Hydrosphere

Karline Soetaert and Lubos Polerecky, Utrecht University

April 2021

Abstract

Here we show how to visualise 2D data sets in R, using functions from the deSolve and plot3D packages. Such data sets are generated, for example, in dynamic runs of 1D reaction-transport models, where the model output is a function of space and time. We illustrate the possibilities using data generated by a model describing salinity distribution in an estuary.

The salinity model

The "toy model" used in this reader describes salinity as a function of distance in an estuary. We assume that water in the estuary is mixed due to tidal dispersion, and that the water flow (advection rate, v) changes periodically over an annual cycle (e.g., due to a seasonal cycle in precipitation upstream).

First, we load the necessary packages, and define the model parameters and the model function.

```
require(deSolve)
require(ReacTran)
require(rootSolve)

Length  <- 100000                                  # m
N       <- 500                                     # - number of boxes
dx      <- Length/N                                # m grid size
x       <- seq(dx/2, by = dx, length.out = N)      # m position of cells

Salini   <- rep(0, times = N)  # initial condition of salinity
SVnames  <- c("Salinity")      # name of the state variable

# model parameters
pars <- c(
  riverSal = 0,       # river salinity
  seaSal   = 35,      # seawater salinity
  v.mean   = 400,     # mean advection velocity [m/d]
  v.amp    = 300,     # amplitude of velocity change [m/d]
  Ddisp    = 30e6     # dispersion coefficient [m2/d]
)

# model function
Estuary1D <- function(t, Salinity, pars) {
 with (as.list(pars),{
```

```
    v       <- v.mean + v.amp*cos(2*pi*t/365)  # velocity at time t

    TranSal <- tran.1D(C = Salinity, C.up = riverSal, C.down = seaSal,
                       D = Ddisp, v = v, dx = dx)

    dSalinity.dt  <- TranSal$dC    # the rates of change = only transport

    list(dSalinity.dt, v = v,
         meanSal = mean(Salinity))
    })
}
```

Now, we calculate the steady-state solution, and use it as the initial condition to calculate a dynamic solution. Note that, by default, the steady-state solution is found assuming that $t = 0$. That is, the velocity in the model function is $v = v_{mean} + v_{amp}$ when the steady-state solution is calculated using the *steady.1D* function.

```
# model run: steady
std <- steady.1D(y = Salini, func = Estuary1D, parms = pars,
    nspec = 1, dimens = N, positive = TRUE)

# model run: dynamic, output every 2 days over 2 years
out <- ode.1D(y = std$y, times = seq(0,2*365,by=2), parms = pars,
              func = Estuary1D, nspec = 1, dimens = N, names = SVnames)
```

Default graphical display

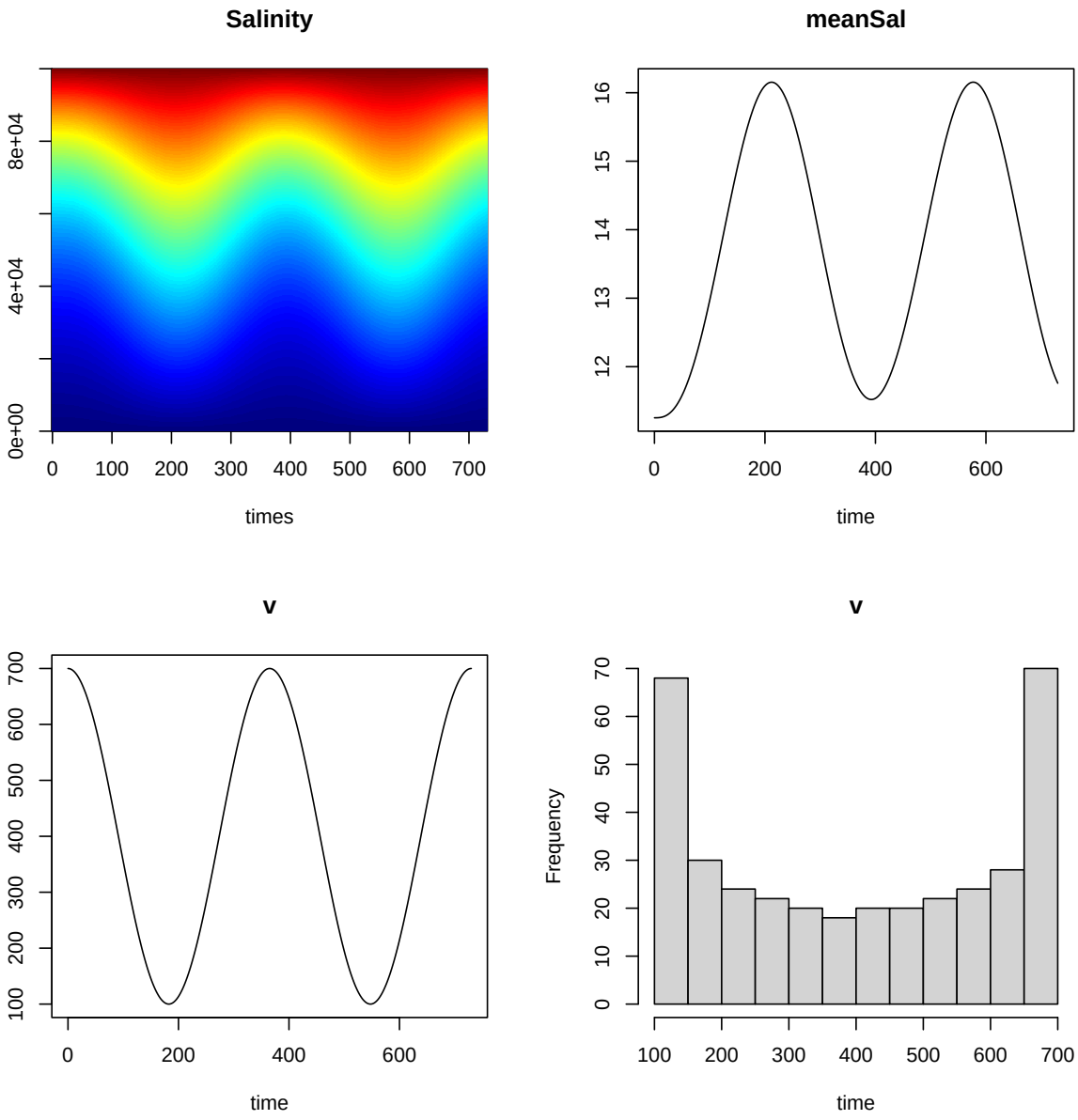
The package *deSolve* contains functions that work with dynamic 1D objects (*subset*, *image*, *matplot.1D*, *plot.1D*) and with 0D objects (*plot*, *matplot.0D*), etc. To get help on these functions, type in the console:

```
?plot.deSolve
```

By default, the plotting functions will determine a suitable arrangement of the graphs (by setting its argument *mfrow*). This can be overruled by specifying your own *mfrow* when calling the functions. Alternatively, you can set *mfrow* = *NULL*, which will not change the mfrow setting.

In the code below, it is first specified that the *image* should be positioned in an arrangement that will have two rows and two columns (*mfrow=c(2,2)*). Then, when calling the *plot* function, the argument (*mfrow=NULL*) tells R not to overrule this arrangement. In addition to a x–y plot, we also plot a histogram of velocities to illustrate how often valocities in a certain velocity interval occur throughout the modeled time domain.

```
image(out, which="Salinity", grid=x, mfrow=c(2,2))
plot(out, which=c("meanSal", "v"), mfrow=NULL)
hist(out, which = "v", mfrow=NULL)
```



Subsetting and summary

To get a summary of the output produced by *ode.1D*, we write:

```
summary(out)
```

```
##            Salinity        v    meanSal
## Min.    8.822634e-03 100.0111  11.253781
## 1st Qu. 4.819396e+00 188.3308  12.133897
## Median  1.191944e+01 401.2911  13.734756
## Mean    1.376699e+01 400.8197  13.766995
## 3rd Qu. 2.177611e+01 613.4911  15.426473
## Max.    3.495648e+01 700.0000  16.154082
```

```
## N       1.830000e+05 366.0000 366.000000
## sd      1.007374e+01 212.7108   1.671911
```

Timeseries at specific positions

We use the following steps to visualise how the salinity changes over time at specific locations in the estuary.

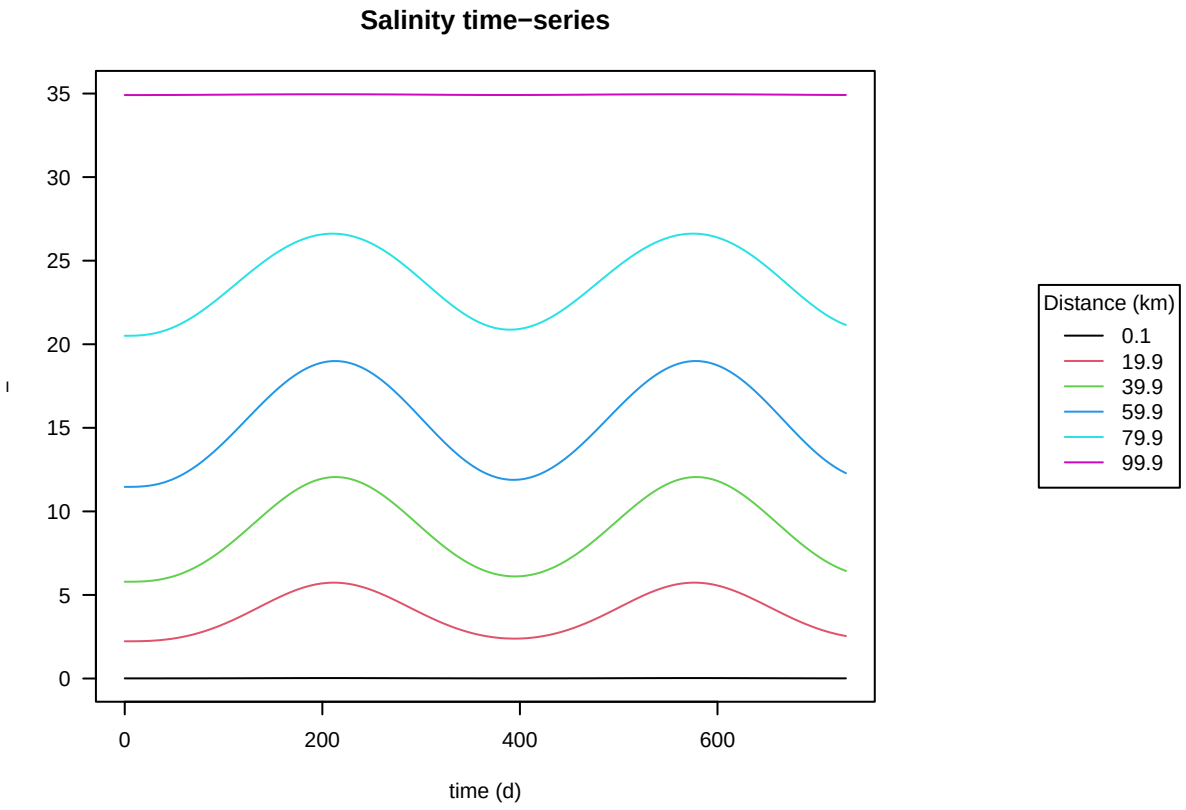
- First, we extract the 2D salinity object (time, space) using the function *subset*. This function returns a matrix with the values arranged according to time (rows) and space (columns); the times are stored as an attribute.
- Then, we use *matplot* for plotting the time variation at specific locations.

Note that, in the example code below, we plot the legend in *another* graph. Function *plot.new* just creates a new graph without doing any plotting. If you do not like the fact that the two "graphs" are equally large, you may use the function *layout*, which allows you to specify the size of the graphs. We used "layout(mat = matrix(nrow = 1, ncol = 4, data = c(1,1,1,2)))", which will make the first graph 3 times as wide as the graph with the legend.

Also note that the specific locations are defined using *indices* of the vector x, where $x[1]$ and $x[500]$ corresponds to the beginning and the end of the estuary, respectively.

```
Salinity <- subset(out, which = "Salinity")
time     <- attributes(Salinity)$time
select   <- c(1,100,200,300,400,500)  # indices, not real distances!

layout(mat = matrix(nrow = 1, ncol = 4, data = c(1,1,1,2)))
matplot(time, Salinity[,select], col=1:6, type ="l", lty=1,
      las=1, ylab="-", xlab="time (d)", main="Salinity time-series")
plot.new()
legend("center", col=1:6, lty=1, legend=x[select]/1000,
  title="Distance (km)")
```



Spatial gradients at specific time points

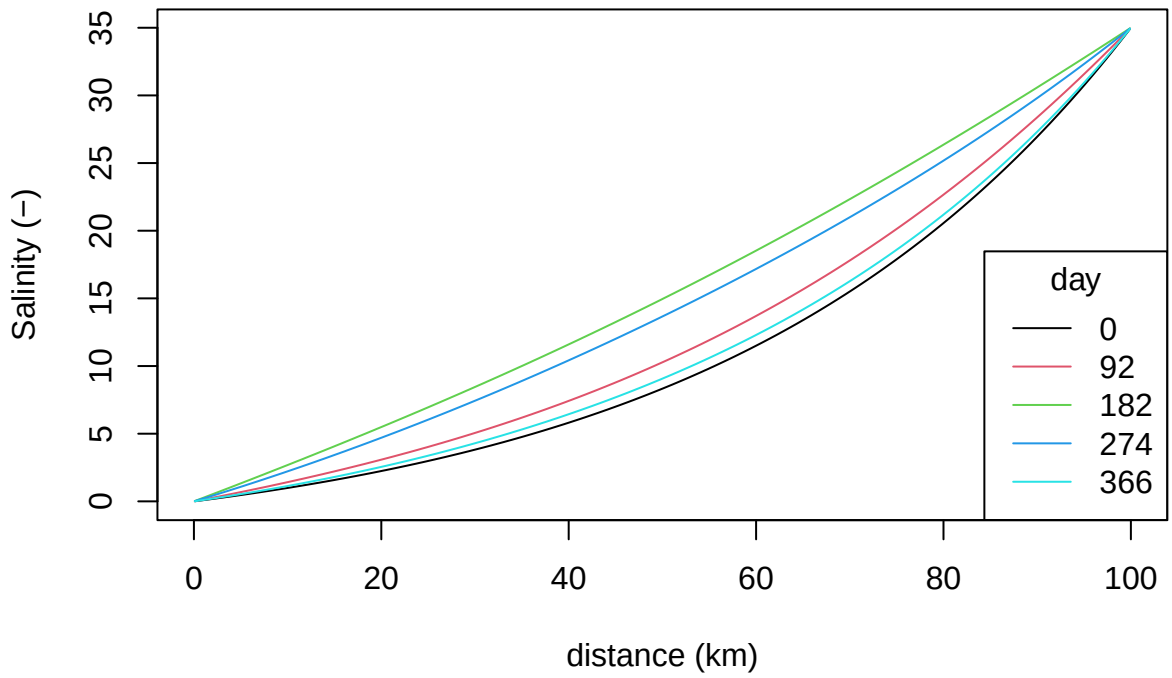
We use the following steps to visualise the varation of salinity in space at specific time points.

- First, we extract the 2D salinity object (time, space) using the function *subset*. This function returns a matrix with the values arranged according to time (rows) and space (columns); the times are stored as an attribute.
- Then, we use *matplot*.

Note that now we specify the time points as *days* rather than via indices. To find the indices in the vector *time* that correspond to these specific days, we use the function *which*.

```
Salinity <- subset (out, which = "Salinity")
time     <- attributes(Salinity)$time
tselect  <- which(time %in% c(0,92,182,274,366)) # day 0, 92, 182, ...

matplot(x=x/1000, y=t(Salinity[tselect,]), col=1:5, type ="l", lty=1,
      xlab="distance (km)", ylab="Salinity (-)")
legend("bottomright", col=1:5, lty=1, legend=time[tselect],
  title="day")
```



Richer graphical display

The function *image2D* from the R-package *plot3D* is more flexible with respect to plotting of images. For instance, it allows you to label the color key (argument *clab*); add *contour*s, increase the plotting resolution (*resfac*), etc.

See

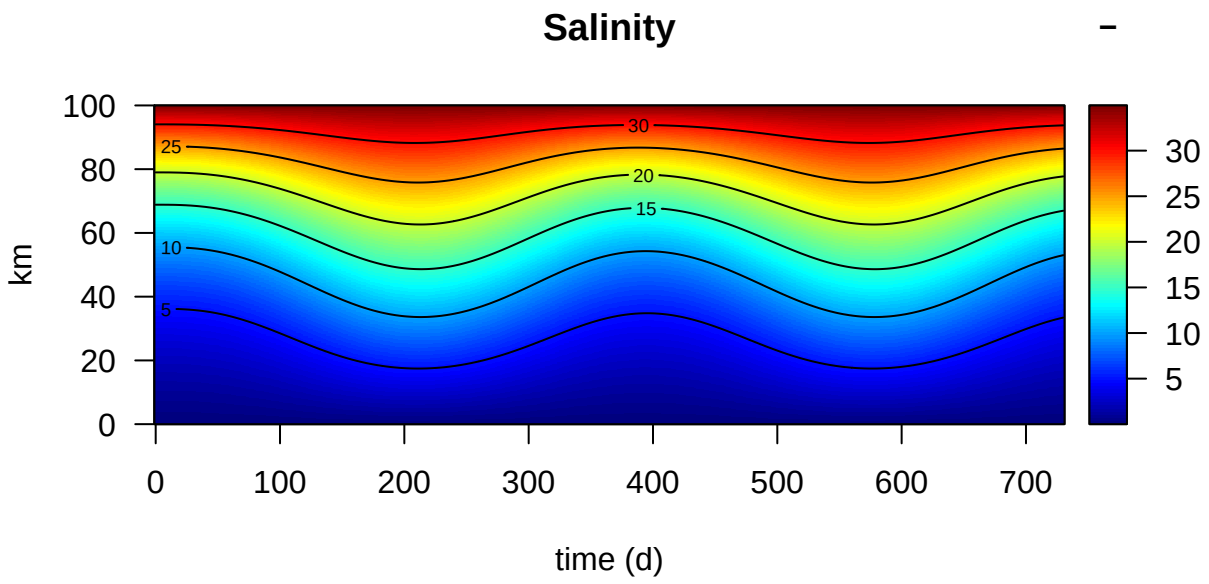
```
?image2D
?contour2D
```

for more details.

The plot3D function image2D can be simply passed as an alternative method to the deSolve image function:

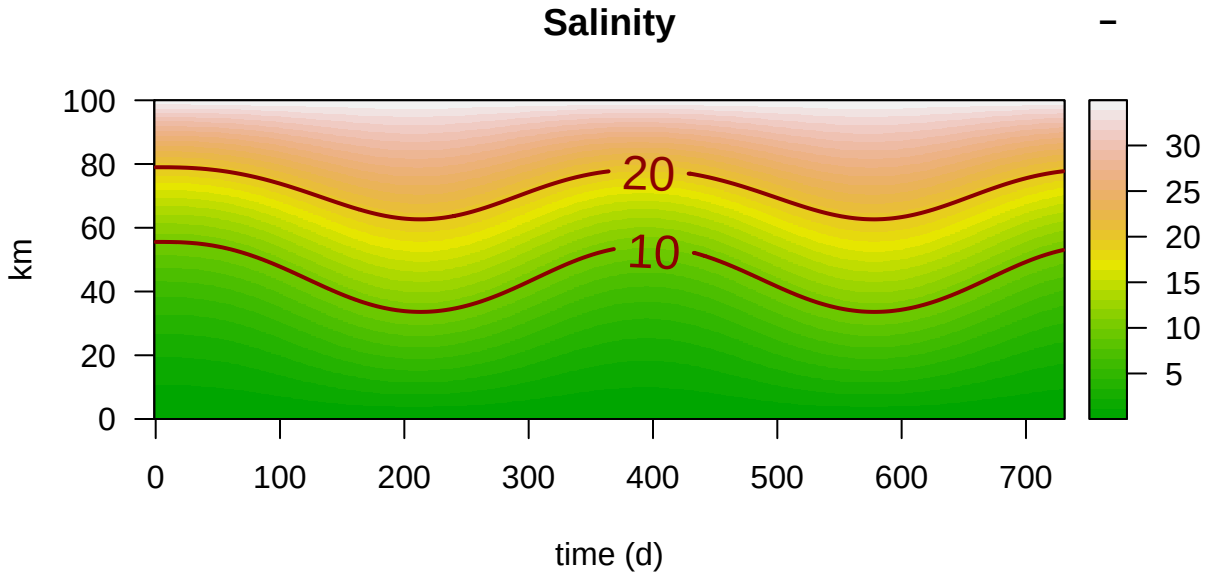
```
require(plot3D)

image(out, which="Salinity", grid=x/1000, ylab="km", xlab="time (d)",
      method="image2D", clab="-", las=1, contour=TRUE)
```



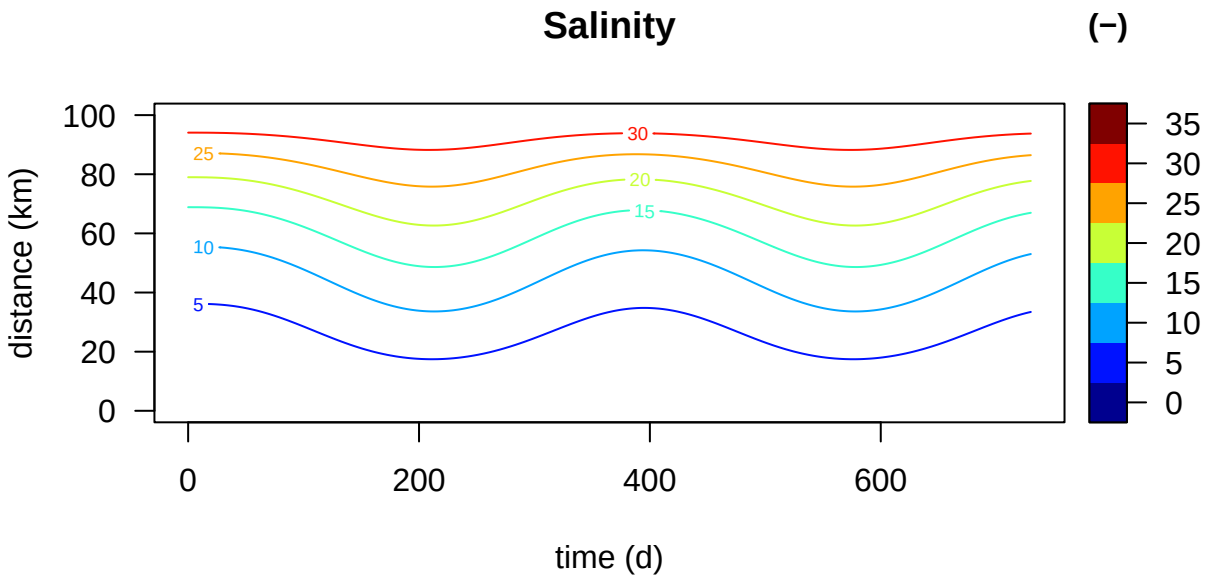
It is more flexible to first extract the salinity and then use the function image2D directly. Note that here we additionally change the colormap (argument *col*).

```
S     <- subset(out, which = "Salinity")
times <- attributes(S)$times
image2D(x=times, y=x/1000, z=S, ylab="km", xlab="time (d)", main="Salinity",
        clab="-", las=1, col = terrain.colors(32),
        contour=list(levels = c(10,20), col="darkred", labcex=1.5, lwd=2))
```



Also, to use the *contour2D* function from *plot3D*, the data have to be extracted first, and the x and y variables have to be explicitly passed as input arguments:

```
S <- subset(out, which = "Salinity")
times <- attributes(S)$times
contour2D(z=S, x=times, y=x/1000, las=1, main="Salinity",
          clab="(-)", xlab="time (d)", ylab="distance (km)")
```



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/.

Soetaert Karline (2009). rootSolve: Nonlinear root finding, equilibrium and steady-state analysis of ordinary differential equations. R-package version 1.6

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

Soetaert, Karline and Meysman, Filip, (2012). Reactive transport in aquatic ecosystems: Rapid model prototyping in the open source software R Environmental Modelling & Software, 32, 49-60.

Karline Soetaert (2019). plot3D: Plotting Multi-Dimensional Data. R package version 1.3. https://CRAN.R-project.org/package=plot3D