Numerical Methods used for Reaction Transport Modelling in R

Reader Accompanying the Course Reaction Transport Modelling in the Hydrosphere

Karline Soetaert and Lubos Polerecky, Utrecht University

May 2021

Abstract

In class you have been able to solve complex models in R, seemingly without effort. For instance, to solve zero-dimensional differential equation models, you used the `ode` solver from the package deSolve, while the `ode.1D` and `steady.1D` functions solved one-dimensional problems. The fact that these methods provide a solution so easily may give the impression that they are trivial functions, but the contrary is true. Here we explain the basics of these functions. We make a short excursion to the field of numerical analysis, first dealing with how ReacTran implements the spatial derivatives. We then show how a steady-state solution is calculated, and end with the integration methods used in class. You may skip reading if you are content with the "abracadabra" that surrounds these solvers.

Numerical methods

In numerical analysis, a numerical "approximation" is used to solve mathematical problems. Thus, rather than being exact symbolic answers, approximate solutions are created within specified error bounds.

Important for numerical methods is to keep these errors as small as possible, while also ensuring that they are numerically stable, i.e. the error they introduce does not grow to be much larger during the calculation.

To understane what a numerical solution is, you may recall the definition of a derivative of a function:

$$\frac{dC}{dx} = \lim_{\Delta x \to 0} \frac{C_{x+\Delta x} - C_x}{\Delta x}$$

In a finite difference approximation, this derivative is calculated using a finite value for Δx :

$$eq1: \quad \frac{dC}{dx} \approx \frac{C_{x+\Delta x} - C_x}{\Delta x}$$

Obviously the smaller is Δx, the more accurate this numerical approximation will be.

Approximating the spatial gradient in the transport equation

Assume that we model how a conservative tracer S (this is a tracer that does not react) is transported in an estuary with length L. The 1D reaction transport equation is:

$$\frac{\partial S}{\partial t} = -\frac{\partial}{\partial x}(v \cdot S - D \cdot \frac{\partial S}{\partial x})$$

where v is velocity, D is the dispersion coefficient, and x is distance. Boundary conditions are $S_0 = 0$, $S_L = 35$

There are two spatial derivates in this formula: the flux gradient (outer derivative) and the salinity gradient in the diffusive flux (last term on right hand side).

In the *tran.1D* function from the *ReacTran* package, both these derivatives are approached by numerical (finite) differences (as in eq. 1). In order to take into account the boundary conditions, the salinity vector is augmented by pasting the boundary values at the start ($S_0 = 0$) and end ($S_L = 35$). Thus we obtain a vector of length N+2: $S^* = [0, S_1, S_2, \ldots S_N, 35]$.

From this vector, the advective+diffusive flux at the boundary between cell $i-1$ and i is estimated as:

$$Flux_{i-1,i} = v_{i-1} \cdot S^*_{i-1} - D \cdot \frac{S^*_i - S^*_{i-1}}{\Delta x}$$

There are N+1 such fluxes, from the upstream to the downstream boundary.

The temporal derivative is then estimated based on these fluxes as:

$$\frac{\partial S}{\partial t}_i = -\frac{Flux_{i,i+1} - Flux_{i-1,i}}{\Delta x}$$

Implementation in R

In R numerical differences can be easily calculated with function "diff".

The above second-order partial differential equation can be approximated by applying the diff function twice, first to estimate the diffusive fluxes, then to take the flux gradient. Note that we add the boundary conditions to the long vector of concentrations (here salinity) before we estimate the differences. We need to impose the concentration at the upper and lower boundary for the diffusive fluxes (*c(riverSal, Salinity, seaSal)*), while we only impose the upper boundary for the advective flux (*c(riverSal, Salinity)*).

In what follows we implement the differential equation model that describes transport of salinity in the estuary:

```
require(rootSolve)      # package to estimate the steady-state condition
```

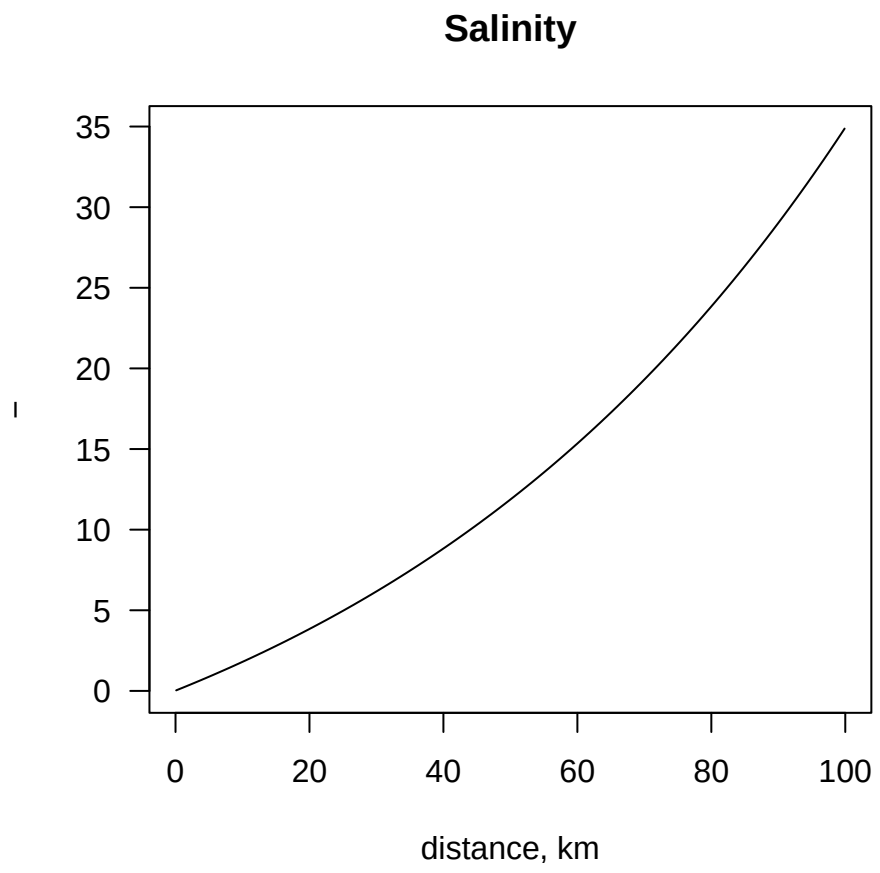
```
## Loading required package: rootSolve
```

```
Length  <- 100000                           # [m] length of the estuary
N       <- 500                              # [-] number of boxes
dx      <- Length/N                         # [m] grid size
x       <- seq(dx/2, by=dx, length.out=N)   # [m] middle of cells

Sal.ini  <- 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        = 400,     # [m/d] mean advection velocity
  Ddisp    = 30e6     # [m2/d] dispersion coefficient
)

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

   # Advective and diffusive fluxes (vectors of length N+1)
```

```
   # Boundary concentrations are padded at the end

    Flux.adv    <- v *c(riverSal, Salinity)
    Flux.diff   <- -Ddisp * diff(c(riverSal, Salinity, seaSal))/dx

   # Derivative = - flux gradient

    dSalinity.dt <- -diff(Flux.adv + Flux.diff)/dx

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

To show that this works, we calculate the steady-state condition of salinity.

```
std <- steady.1D(y=Sal.ini, func=Estuary1D, parms=pars,
                 nspec=1, dimens=N, positive=TRUE, names=SVnames)
plot(std, type="l", grid=x/1000, xlab="distance, km", las=1, ylab="-")
```



Basics of finding a steady-state solution: Newton's method

When solving a differential equation to steady-state, we need to find the "root" of the differential equation, this is, the value of y for which

$$\frac{dy}{dt} = f(y) = 0$$

Package *rootSolve* has implemented a modified form of Newton's method.

This method is based on the fact that, if the root is in the vicinity of y_0, then an even better estimate of the root is given by y_1, calculated as follows:

$$y_1 = y_0 - \frac{f(y_0)}{f'(y_0)}$$

and where $f'(y_0)$ is the derivative of f, evaluated at y_0.

This formula is applied for a sequence of y-values until the difference between successive values becomes sufficiently small. Thus, Newton's method is an iterative method, where the iterations ($\nu = 0, 1...$) are defined as:

$$y^{\nu+1} = y^{\nu} - \frac{f(y^{\nu})}{f'(y^{\nu})}$$

and where $f'(y^{\nu})$ is the Jacobian matrix, containing $\frac{\partial f(y_i^{\nu})}{\partial y_j^{\nu}}$.

The Jacobian matrix itself can be created by numerical approximation: $\frac{\partial f(y_i)}{\partial y_j} = \frac{f(y^*)_i - f(y)_i}{\Delta y_j}$ and where y^* is the y-vector that contains the perturbed value of y_j, i.e. $y^* = y + [0, \ldots, \Delta y_j, \ldots, 0]$.

Implementation in R

Below is a simple implementation of the Newton-Raphson method, that finds the root for a model with one state variable.

```
Model <- function(t,y,p)  # the derivative function
  list(1 - 0.1*y^2)

Newton <- function(func, t=0, y=0, p=NULL){

  DY <- 1e6      # Dy = differences between iterations
  dy <- 1e-4     # Value to perturb y

  while (abs(DY) > 1e-6){  # stop when differences are <= 1e-6

    Fy  <- func(t, y, p)[[1]]   # function value for current y

    # numerical approximation of F'(y)
    yn  <- y+dy                 # perturbed y-value
    Fn  <- func(t, yn, p)[[1]]  # function value for perturbed y
    dFy <- (Fn-Fy)/dy           # estimate of F'(y)

    # update y
    DY <- Fy/dFy
```

```
    y  <- y - DY
  }
  return(y)
}

Newton(func = Model)
```

```
## [1] 3.162278
```

A lot of time can be save when the Jacobian matrix is efficiently created and inverted (or decomposed). We will deal with how this is used in rootSolve later.

Basics of integration: Euler

The rules for finding temporal solutions of differential equations in R is that you define a "derivative function" that takes as input the time, value of the state variables at that time, and the parameter values, and where the derivatives at that time point are calculated and returned to R.

The solution is then generated by simply calling *ode*, while passing as arguments the derivative function, the initial conditions of the state variables, the parameter values, and the times for which you want output.

To understand what happens within the *ode* function from the deSolve package, we make a caricatural version of it that implements the euler integration method[1].

Euler integration essentially finds the solution for the next time step ($y_{t+\Delta t}$) by combining the values at the previous time step (y_t) with the derivative at the previous time step ($\frac{dy}{dt}|_t$) as follows:

$$y_{t+\Delta t} = y_t + \Delta t \cdot \frac{dy}{dt}|_t$$

This is called a numerical approximation of the integration. The formula is derived from the numerical approximation of the derivative:

$$\frac{dy}{dt}|_t = \frac{y_{t+\Delta t} - y_t}{\Delta t \cdot}$$

Implementation in R

Here is an implementation of the Euler method:

```
ODE.euler <- function(times, y, func, parms){
  out <- NULL                     # will contain the results (a matrix)
  dt  <- diff(times)[1]           # calculate the time step.

  for (t in times){               # step through time
    derivs <- func(t, y, parms)   # call derivative function, result in derivs
    dy.dt  <- derivs[[1]]         # first element of derivs: derivative
    vars   <- derivs[[-1]]        # other elements of derivs: ordinary variables
    out    <- rbind (out,         # add output to the output matrix
                     c(t, y, vars))
    y      <- y + dt*dy.dt        # update state variables to next time step
  }

  return(out) # matrix - first column:time, then state variable, and ordinary variable
}
```

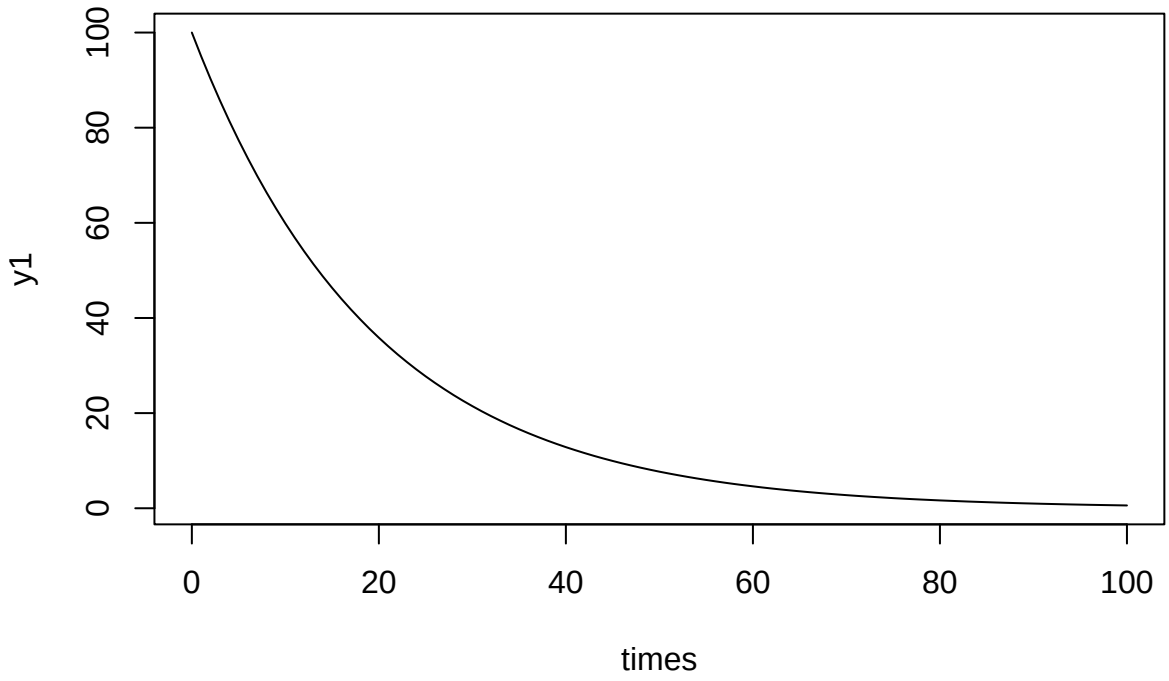
[1] the actual ode function is much more complex than that!

We apply the Euler integration to a simple derivative function that models first order decay of two substances.

```
decay.fun <- function(t, y, p){
  return(list(-p*y,             # derivatives
              yTot = sum(y))) # output variable
}

out <- ODE.euler(times=0:100, y=c(y1=100, y2=20), func=decay.fun,
                 parms=c(decay=0.05))
head(out)
```

```
##               y1       y2
## [1,] 0 100.00000 20.00000 120.00000
## [2,] 1  95.00000 19.00000 114.00000
## [3,] 2  90.25000 18.05000 108.30000
## [4,] 3  85.73750 17.14750 102.88500
## [5,] 4  81.45063 16.29012  97.74075
## [6,] 5  77.37809 15.47562  92.85371
```

```
plot(out[,1:2], type="l", xlab="times")
```



Why Euler is never used

The above implemented Euler integration method is an *explicit* integration method, which means that the state of a system at a *next* time step $(t + \Delta t)$ is calculated from the derivative that is estimated from the state at the *current* time step t. Although this may seem a logical way to integrate, the Euler method is

-because of that- both imprecise and unstable. For most -if not all- of the examples that we have seen in class the Euler method would completely fail to find a solution.

Implicit integration methods

Much more robust is to use so-called *implicit* methods, which estimate the state of a system at a next time step from the state of the system at the current time step and the derivative from the *next* time step. The implicit solution methods have much better stability properties than explicit methods. But as they introduce non-trivial computational overhead, they have to be carefully implemented.

The simplest of these methods is the backward Euler that implements[2]:

$$y_{t+\Delta t} = y_t + \Delta t \cdot \frac{dy}{dt}|_{t+\Delta t}$$

You may note that the derivative $\frac{dy}{dt}|_{t+\Delta t}$ is estimated at $t + \Delta t$, i.e. using the unknown values of $y_{t+\Delta t}$.

It would lead too far to explain how implicit methods work in detail, but essentially these methods do not march forward in time like the explicit euler method, but instead they find a solution by solving an *equation* involving the Jacobian. The equation is solved using Newton's method that we explained above.

First the equation is rewritten as:

$$g(y_{t+\Delta t}) = y_{t+\Delta t} - y_t - \Delta t \cdot \frac{dy}{dt}|_{t+\Delta t}$$

and so the solution of this step in the differential equation is to find the value of $y_{t+\Delta t}$ so that $g(y_{t+\Delta t}) \approx 0$.

The Newton iterations ($\nu = 0, 1, \ldots$) are (see previous chapter):

$$y^{\nu+1}_{t+\Delta t} = y^{\nu}_{t+\Delta t} - \frac{g(y^{\nu}_{t+\Delta t})}{(\frac{\partial g}{\partial y})}$$

or

$$y^{\nu+1}_{t+\Delta t} = y^{\nu}_{t+\Delta t} - (\frac{\partial g}{\partial y})^{-1}\ g(y^{\nu}_{t+\Delta t})$$

which can be written as:

$$y^{\nu+1}_{t+\Delta t} = y^{\nu}_{t+\Delta t} - (I - \Delta t \frac{\partial f}{\partial y})^{-1}\ g(y^{\nu}_{t+\Delta t})$$

Here $\frac{\partial f}{\partial y}$ is the Jacobian matrix evaluated at the point $y^{\nu}_{t+\Delta t}$[3].

Obviously, compared to the explicit Euler, more work needs to be done to estimate the values at the next time step, as the Jacobian $\frac{\partial f}{\partial y}$ has to be created and decomposed.

Nevertheless in many real-world problems, the time steps (Δt) that can be taken this way are much larger than the time steps that could be taken when using an explicit method. Because of that, it is not uncommon that the use of implicit methods saves multiple orders of computing time.

To grasp the size of the Jacobian matrix: for a 1D reaction-transport model that describes 3 species in 1000 boxes, this Jacobian matrix is of size (3000, 3000). Fortunately, for reaction-transport models, the Jacobian

[2] this method is also much simpler than the integration routines in deSolve

[3] The Jacobian matrix contains, on row i and column j the element $\frac{\partial \frac{\partial y}{\partial t}|_i}{\partial y_j}$

matrix contains a lot of zero's - it is a "sparse" matrix. There exist very efficient ways to decompose such matrices. Also, it is possible to design clever strategies to create this matrix by perturbing combinations of state variables at once.

Integrators and steady-state methods in deSolve and rootSolve

ode

The *deSolve* package uses as the main integrator (*ode*) a method that (Petzold, 1983)

(1) chooses whether to use an implicit or explicit method - and that may even switch between both integration types and
(2) that adapts the integration time step to the prevailing numerical properties of the solution.

ode.1D and steady.1D

In addition, the solvers called *ode.1D* and *steady.1D* have been designed to create and to decompose the Jacobian matrix in a very efficient way, by taking into account the (known) sparsity structure.

Indeed, for 1D reaction transport model, the Jacobian can be efficiently generated and decomposed if the dimensionality of the problem, and the number of species is known. Based upon that information we can calculate which elements of the Jacobian will be nonzero.

This is why you need to pass the arguments "dimens" and "nspec" when calling the solver *steady.1D* or *ode.1D*.

References

Petzold, Linda R. (1983) Automatic Selection of Methods for Solving Stiff and Nonstiff Systems of Ordinary Differential Equations. Siam J. Sci. Stat. Comput. 4, 136–148.

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, 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 (2009). rootSolve: Nonlinear root finding, equilibrium and steady-state analysis of ordinary differential equations. R-package version 1.6. https://CRAN.R-project.org/package=rootSolve

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.