R-package ReacTran : Reactive Transport Modelling in R

Karline Soetaert
NIOO-CEME
The Netherlands

Filip Meysman
NIOO-CEME
The Netherlands

Abstract

R package **ReacTran** (Soetaert and Meysman 2009) contains functions for creating reactive- transport models in R .

Keywords: reactive-transport, diffusion, advection, reaction, porous media, rivers, estuary, water column, R .

1. General reaction transport equation in one dimension

1.1. The reaction-transport equation

The general 1-D reaction-transport equation in multi-phase environments and for shapes with variable geometry is:

$$\frac{\partial \xi C}{\partial t} = -\frac{1}{A} \cdot \frac{\partial (A \cdot J)}{\partial x} + reac$$

where

- t is time
- x is space
- C is concentration of a substance in its respective phase (units of e.g. $ML^{-3}liquid$ for sediment solutes). [1]
- ξ is the volume fraction (-), i.e. the fraction of a phase in the bulk volume (see figure). In many cases, only one phase is considered and $\xi = 1$; For sediments, ξ would be porosity (solutes), or 1-porosity (solids).
- A is the (total) surface area (L^2).
- J are fluxes, (units of $ML^{-2}t^{-1}$)

[1] here we use M for mass, L for length and t for time

The Fluxes (J), which are estimated per unit of total surface, consist of a dispersive and an advective component:

$$J = -\xi D \cdot \frac{\partial C}{\partial x} + \xi u \cdot C$$

where

- D is the diffusion (or dispersion) coefficient, units of $L^2 t^{-1}$
- u is the advection velocity, units of Lt^{-1}

1.2. Boundary conditions in 1-D models

The boundaries (at the extremes of the model domain, e.g. at x=0) can be one of the following types:

- A concentration boundary, e.g. $C|_{x=0} = C_0$
- A diffusive + advective flux boundary $J_{x=0} = J_0$
- A boundary layer convective exchange flux boundary $J_{x=0} = a_{bl} \cdot (C_{bl} - C_0)$

1.3. Numerical approximation

The reaction-transport formula consists of a partial differential equation (PDE).

To solve it, the spatial gradients are approximated using so-called numerical differences (the method-of-lines, MOL, approach). This converts the partial differential equations into ordinary differential equations (ODE).

This means that the model domain is divided into a number of grid cells, and for each grid cell i, we write:

$$\frac{d\xi_i C_i}{dt} = -\frac{1}{A_i} \cdot \frac{\Delta_i(A \cdot J)}{\Delta x_i} + reac_i$$

where Δ_i denotes that the flux gradient is to be taken over box i, and Δx_i is the thickness of box i:

$$\Delta_i(A \cdot J) = A_{i,i+1} \cdot J_{i,i+1} - A_{i-1,i} \cdot J_{i-1,i}$$

where `i,i+1` denotes the interface between box i and i+1.

The fluxes at the box interfaces are discretized as:

$$J_{i-1,i} = -\xi_{i-1,i} D_{i-1,i} \cdot \frac{C_i - C_{i-1}}{\Delta x_{i-1,i}} + \xi_{i-1,i} u_{i-1,i} \cdot (\vartheta_{i-1,i} \cdot C_{i-1} + (1 - \vartheta_{i-1,i}) \cdot C_i)$$

with $\Delta x_{i-1,i}$ the distance between the centre of grid cells i-1 and i, and ϑ the upstream weighing coefficients for the advective term.

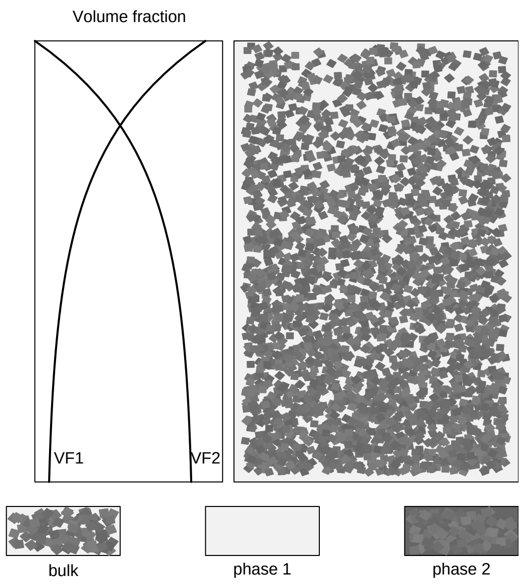

Figure 1: An example of multiple phases in **ReacTran** , adapted from figure 3.9 from Soetaert and Herman, 2009

2. one-dimensional finite difference grids and properties in ReacTran

2.1. Generating a spatial discretization grid

The 1-D spatial discretization grid can best be generated with **ReacTran** function `setup.grid.1D`. Function `setup.grid.1D` creates a grid, which can comprise several zones:

```
setup.grid.1D <- function(x.up=0,x.down=NULL, L=NULL, N=NULL,
   dx.1 =NULL, p.dx.1 = rep(1,length(L)), max.dx.1 = L,
   dx.N =NULL, p.dx.N = rep(1,length(L)), max.dx.N = L)
```

with the following arguments:

- `x.up`. The position of the upstream boundary.
- `x.down`. The positions of the downstream boundaries in each zone.
- `L, N`, the thickness and the number of grid cells in each zone.
- `dx.1, p.dx.1, max.dx.1`, the size of the first grid cell, the factor of increase near the upstream boundary, and maximal grid cell size in the upstream half of each zone.
- `dx.N, p.dx.N, max.dx.N`, the size of the last grid cell, the factor of increase near the downstream boundary, and maximal grid cell size in the downstream half of each zone.

It returns an element of class `grid.1D` that contains the following elements (units L), see figure 2:

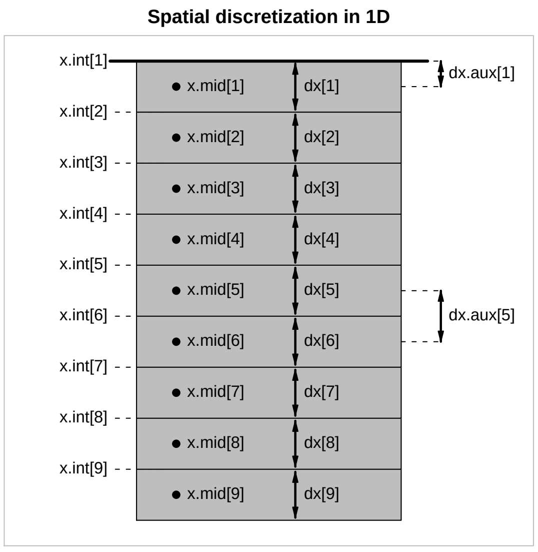

Figure 2: Nomenclature for the spatial discretization grid in `grid.1D`

- `x.up, x.down`. The position of the upstream and downstream boundary.
- `x.int`, the position of the grid cell interfaces, where the fluxes are specified, a vector of length N+1.
- `x.mid`, the position of the grid cell centres, where the concentrations are specified, a vector of length N.
- `dx`, the thickness of boxes , i.e. the distance between the grid cell interfaces, a vector of length N. This is equivalent to Δx_i
- `dx.aux`, the distance between the points where the concentrations are specified, a vector of length N+1. This is equivalent to $\Delta x_{i-1,i}$.

For example, to subdivide an estuary, 100 km long into 50 boxes, with the first box of size 1 km, we write [2]:

```
>  (grid<-setup.grid.1D(L=100,dx.1=1,N=50))

$x.up
[1] 0

$x.down
[1] 100

$x.mid
```

[2]note that by embracing the statement within brackets, we execute it AND print the result to the screen

```
 [1]  0.5001181  1.5132946  2.5526866  3.6189723  4.7128477  5.8350266
 [7]  6.9862412  8.1672431  9.3788027 10.6217109 11.8967787 13.2048383
[13] 14.5467433 15.9233695 17.3356152 18.7844020 20.2706755 21.7954056
[19] 23.3595874 24.9642416 26.6104155 28.2991833 30.0316472 31.8089377
[25] 33.6322148 35.5026683 37.4215188 39.3900187 41.4094526 43.4811383
[31] 45.6064279 47.7867083 50.0234025 52.3179700 54.6719083 57.0867537
[37] 59.5640820 62.1055100 64.7126961 67.3873420 70.1311931 72.9460399
[43] 75.8337196 78.7961165 81.8351641 84.9528455 88.1511955 91.4323012
[49] 94.7983039 98.2514003

$x.int
 [1]   0.000000   1.000236   2.026353   3.079020   4.158925   5.266771
 [7]   6.403282   7.569200   8.765286   9.992320  11.251102  12.542455
[13]  13.867221  15.226265  16.620474  18.050757  19.518047  21.023304
[19]  22.567508  24.151667  25.776816  27.444015  29.154352  30.908943
[25]  32.708933  34.555497  36.449840  38.393198  40.386839  42.432066
[31]  44.530211  46.682645  48.890772  51.156033  53.479907  55.863910
[37]  58.309598  60.818566  63.392454  66.032939  68.741745  71.520641
[43]  74.371439  77.296000  80.296233  83.374095  86.531596  89.770795
[49]  93.093807  96.502801 100.000000

$dx
 [1] 1.000236 1.026117 1.052667 1.079904 1.107846 1.136511 1.165918
 [8] 1.196086 1.227034 1.258783 1.291353 1.324766 1.359044 1.394208
[15] 1.430283 1.467291 1.505256 1.544204 1.584160 1.625149 1.667199
[22] 1.710337 1.754591 1.799990 1.846564 1.894343 1.943358 1.993642
[29] 2.045226 2.098145 2.152434 2.208127 2.265261 2.323874 2.384003
[36] 2.445688 2.508969 2.573887 2.640485 2.708807 2.778896 2.850798
[43] 2.924561 3.000233 3.077862 3.157501 3.239199 3.323012 3.408993
[50] 3.497199

$dx.aux
 [1] 0.5001181 1.0131765 1.0393920 1.0662857 1.0938754 1.1221789 1.1512147
 [8] 1.1810018 1.2115597 1.2429082 1.2750678 1.3080596 1.3419050 1.3766261
[15] 1.4122457 1.4487869 1.4862735 1.5247301 1.5641818 1.6046542 1.6461739
[22] 1.6887678 1.7324639 1.7772905 1.8232771 1.8704535 1.9188506 1.9684999
[29] 2.0194339 2.0716857 2.1252896 2.1802804 2.2366941 2.2945675 2.3539383
[36] 2.4148454 2.4773283 2.5414280 2.6071862 2.6746459 2.7438510 2.8148469
[43] 2.8876797 2.9623970 3.0390476 3.1176814 3.1983499 3.2811057 3.3660027
[50] 3.4530964 1.7485997

$N
[1] 50

attr(,"class")
[1] "grid.1D"
```

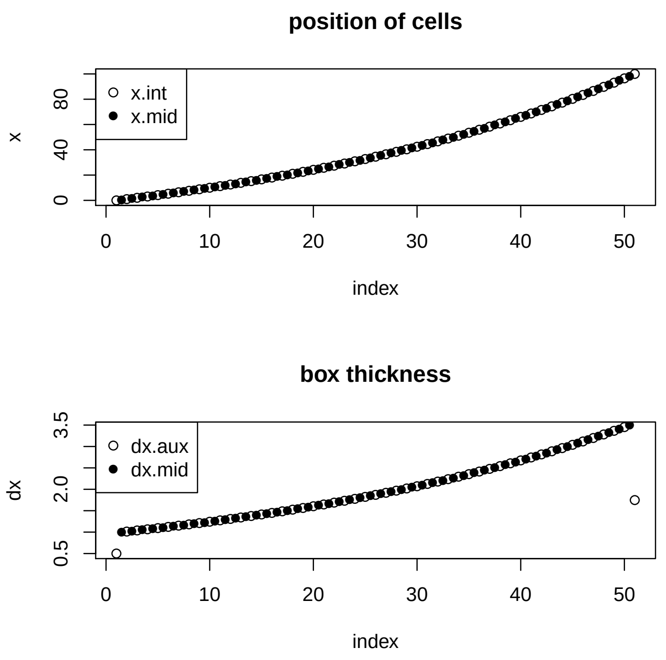

Figure 3: exponential grid size - see text for R -code

which can be plotted as follows:

```
> plot(grid)
```

2.2. Other grid properties

In the 1-D discretization formula, some properties (e.g. concentrations) are defined at the centre of boxes (C_i), while other properties (transport coefficients) are prescribed at the box interfaces (ϑ, ξ, D).

These properties can be generated conform a previously defined grid with **ReacTran** function `setup.prop.1D`. Its syntax is:

```
setup.prop.1D(func=NULL, value=NULL, xy=NULL,
  interpolate="spline", grid, ...)
```

They can be specified as a function, or as a (constant) value, or as an (x,y) data series, which is interpolated using a spline or linearly.

The function returns a list of class `prop.1D` that contains:

- `int`, the property value at the grid cell interfaces, a vector of length N+1.
- `mid`, the property value at the middle of grid cells, a vector of length N.

A number of commonly used property functions are implemented in **ReacTran** :

- `p.exp` for an exponentially decreasing transition
- `p.lin` for a linearly decreasing transition
- `p.sig` for a sigmoidally decreasing transition

Below, we demonstrate the use of `setup.prop.1D` to create the properties for use in a sediment biogeochemical model.

After defining a grid, we first use utility function `p.exp` to calculate a bioturbation profile, assuming there is a constant bioturbation in an upper layer (2 cm), declining exponentially below this layer.:

```
> grid <- setup.grid.1D(L=10,N=100)
> Db   <- setup.prop.1D(func=p.exp,grid=grid,y.0=5,y.inf=0,x.L= 2)
```

We then define a function (`exp.inc`) that specifies an exponentially increasing profile to calculate the volume fraction of solid substances in the sediment. These are represented by 1-porosity, where the porosity is an exponentially decreasing function, which can be calculated using `p.exp`.

```
> exp.inc <- function(x,y.0=1,y.inf=0.5,x.L=0,x.att=1)
+       return(1-p.exp(x,y.0,y.inf,x.L,x.att))
> VFsolid <- setup.prop.1D(func=exp.inc,grid=grid,y.0=0.9,y.inf=0.7)
```

A `plot` method is defined for class `prop.1D`; to invoke it one has to pass both the property as the grid on which it is based; `xyswap=TRUE` swaps the x- and y-axis, which is useful for plotting vertical profiles.

```
>   par(mfrow=c(1,2))
>   plot(VFsolid,grid,xyswap=TRUE,type="l",main="1-porosity")
>   plot(Db,grid,xyswap=TRUE,type="l",main="Db")
>   par(mfrow=c(1,1))
```

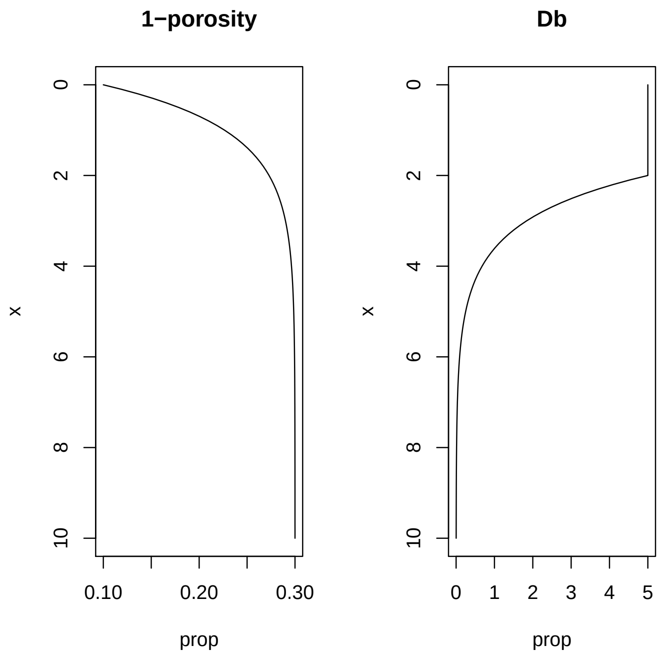

Figure 4: Two exponentially declining properties - see text for R -code

3. one-dimensional reactive-transport modelling in ReacTran

Implementing a reactive-transport model in **ReacTran** proceeds in several steps.

- Setting up a finite difference grid and defining properties attached to this grid (see previous section)
- Specifying a model function that describes the rate of change of substances due to transport and reaction. The transport of properties is done with **ReacTran** function `tran.1D`
- Solving the model. Depending on whether a transient or a steady-state solution is desired, solving the model makes use of functions `ode.1D` or `steady.1D` from packages **deSolve** and **rootSolve**.

Below we first explain how to use **ReacTran** function `tran.1D`, after which it is used in a model, which is consequently solved.

3.1. R-function tran.1D

The default input for the `tran.1D()` function in R is:

```
tran.1D(C, C.up = C[1], C.down = C[length(C)],
  flux.up = NULL, flux.down = NULL,
  a.bl.up = NULL, a.bl.down = NULL,
  D = 0, v = 0, AFDW = 1, VF = 1, A = 1, dx,
  full.check = FALSE, full.output = FALSE)
```

with the following arguments:

- `C, C.up, C.down`. The concentrations, per unit of phase volume, in the centre of each grid cell, a vector of length N (`C`) and at the upstream or downstream boundary, one value (`C.up, C.down`).
- `flux.up, flux.down`. The fluxes, per unit of total surface, at the upstream and downstream boundaries, $(ML^{-2}t^{-1})$.
- `a.bl.up, a.bl.down`, the convective transfer coefficients.
- `D`, the diffusion (dispersion) coefficients, either one value, or a vector (`D`) or packed as a `grid`, (L^2t^{-1}).
- `v`, the advective velocity (Lt^{-1}).
- `AFDW` the weights used in the finite difference approximation for advection (-).
- `VF`, the volume fractions (-).
- `A` the surface areas (L^2)

- `dx`, the distances between cell interfaces (L), the discretization grid. Must be specified.
- `full.check`, when `TRUE`, the consistency of the input is checked.
- `full.output`, when `TRUE` full output is returned.

Note that several properties (`dx, D, v, AFDW, VF, A`) can be passed in different ways:

- a single number, in which case they are assumed constant
- a vector of length N+1, i.e. defined on the grid interfaces
- a list of type `grid.1D` or of type `prop.1D`, as created by `setup.grid` (spatial discretization) or by `setup.prop.1D` (see previous section).

Function `reac.1D` returns the rate of change of `C` due to transport, and the fluxes up-and downstream; if `full.output` is `TRUE` then also the advective and dispersive fluxes at all layer interfaces are returned.

For example:

```
> tran.1D(C=1:20, D=0, flux.up = 1, v=1, dx=1)

$dC
 [1]  0 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1

$flux.up
[1] 1

$flux.down
[1] 20

> tran.1D(C=1:20, D=0, flux.up = 1, v=1, dx=1, full.output=TRUE)

$dC
 [1]  0 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1

$C.up
[1] 1

$C.down
[1] 20

$dif.flux
 [1] 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0

$adv.flux
 [1]  1  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19 20
```

```
$flux
 [1]  1  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19 20

$flux.up
[1] 1

$flux.down
[1] 20
```

3.2. A 1-D reaction transport model

Function `tran.1D` estimates the *rate of change* of substances as a function of *transport* processes.

If, in addition to transport, reaction terms are added, a 1-D *reaction-transport model* is obtained.

In line with the way ordinary differential equations are solved in R , a reaction-transport model is specified in an R -function that computes the derivatives in the ODE at a certain time `t`.

This function will be called by the solution methods as: `func(t,y,parms,...)` where `t` is the current time point, `y` are the current values of the state variables in the ODE system and `parms` are model parameters.

It should return a list whose first element is a vector containing the derivatives of y. (for more details, see packages **deSolve** , **rootSolve**)

For instance, the function representing the following model

$$\frac{\partial C}{\partial t} = -v \cdot \frac{\partial C}{\partial x} - kC$$

with boundary condition:

$$v \cdot C|_{x=0} = F_0$$

can be implemented in R as:

```
>     parms <- c(F0 = 1, v = 1, k = 0.1, dx = 1)

>     advModel <- function(t, C, parms) {
+
+       with (as.list(parms), {
+
+         Tran <- tran.1D(C=C, D=0, flux.up = F0, v=v, dx=dx)
+         Consumption <-  k*C
+         dC   <- Tran$dC - Consumption
+
+         return (list(dC = dC, Consumption= Consumption,
+                      flux.up = Tran$flux.up, flux.down = Tran$flux.down))
+       })
+
+     }
```

Note the use of `with (as.list(parms),...` which allows to access by name the previously defined model parameters (`parms`) within the function.

3.3. Solving a 1-D reaction transport model

In R , 1-D models consisting of ordinary differential equations can be solved in two ways.

- by estimating the *steady-state condition*, using function `steady.1D` from R -package rootSolve (Soetaert 2009).
- by running the model *dynamically*, using functions `ode.1D` from R -package deSolve (Soetaert, Petzoldt, and Setzer 2010).

To solve the above model to steady-state, we invoke `steady.1D`:

```
>   out <- steady.1D(func=advModel, y=runif(25), parms=parms,
+                      nspec=1, positive=TRUE)
```

Function `steady.1D` estimates the steady-state iteratively and requires an initial guess of the state variables. For most cases, the exact values are not important; the initial guess of the state variables in the above example consists simply of 25 uniformly distributed random numbers ([0,1]);

we specify that the model comprises only one species (`nspec` and that we are interested only in a solution consisting of positive numbers (`positive=TRUE`), as negative concentrations do not exist (in the real environment; they may exist mathematically).

The outcome of this model is a list called `out`, which contains the steady-state condition of the state-variables (item `y`), the consumption rate and the fluxes at the upper and lower boundary as defined in function `advModel`. In addition, attribute `precis` gives a measure of how far the system is from steady-tate at each iteration of the steady-state solver; only two iterations were needed.

```
> out


$y
 [1] 0.9090909 0.8264463 0.7513148 0.6830135 0.6209213 0.5644739 0.5131581
 [8] 0.4665074 0.4240976 0.3855433 0.3504939 0.3186308 0.2896644 0.2633313
[15] 0.2393921 0.2176291 0.1978447 0.1798588 0.1635080 0.1486436 0.1351306
[22] 0.1228460 0.1116782 0.1015256 0.0922960

$Consumption
 [1] 0.09090909 0.08264463 0.07513148 0.06830135 0.06209213 0.05644739
 [7] 0.05131581 0.04665074 0.04240976 0.03855433 0.03504939 0.03186308
[13] 0.02896644 0.02633313 0.02393921 0.02176291 0.01978447 0.01798588
[19] 0.01635080 0.01486436 0.01351306 0.01228460 0.01116782 0.01015256
```

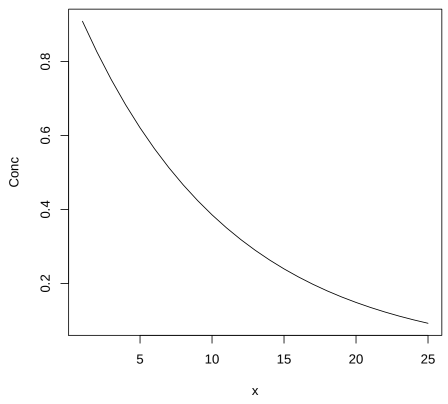

Figure 5: Solution of the uni-component reactive-transport model in 1-D - see text for R -code

```
[25] 0.00922960

$flux.up
[1] 1

$flux.down
[1] 0.092296

attr(,"precis")
[1] 3.008114e-01 1.516853e-09
attr(,"steady")
[1] TRUE
attr(,"class")
[1] "steady1D" "list"
attr(,"nspec")
[1] 1
```

It can be plotted as:

```
> plot (out$y ,type="l",xlab="x", ylab="Conc")
```

It is good modelling practice to test mass conservation of a model, i.e. as a check that mass is not created or destroyed by numerical means (programming or modelling error).

For this simple example, it is obvious that mass is conserved, but we will test it to exemplify the procedures.

If mass is conserved and the system is at steady-state, then the net input by transport in the system should equal the total net consumption. For our model, the net input to the system is given by `flux.up - flux.down`, while the total consumption is simply the sum of the consumption in each box.

```
> with (out, print(sum(Consumption)-(flux.up-flux.down)))

[1] 4.026226e-09
```

The fact that the quantity is not completely zero is due to the small deviation from steady-state (as is clear from attribute `precis`).

3.4. example: 1-D transport in a porous spherical body

This, somewhat more complex example models oxygen consumption in a spherical aggregate. The example is more complex because it assumes that both the surface area (`A`) and the volume fraction (`VF`) (here the "porosity") vary along the spatial axis.

We start by formulating the model function (`Aggregate.Model`).

```
> Aggregate.Model <- function(time,O2,pars) {
+
+   tran <- tran.1D(C=O2, C.down=C.ow.O2,
+     D=D.grid, A=A.grid,
+     VF=por.grid, dx=grid )
+
+   reac <- - R.O2*(O2/(Ks+O2))
+   return(list(dCdt = tran$dC + reac, reac = reac,
+               flux.up = tran$flux.up, flux.down = tran$flux.down))
+
+ }
```

next the parameters are defined:

```
> C.ow.O2 <- 0.25      # concentration O2 water [micromol cm-3]
> por     <- 0.8       # porosity
> D       <- 400       # diffusion coefficient O2 [cm2 yr-1]
> v       <- 0         # advective velocity [cm yr-1]
> R.O2    <- 1000000   # O2 consumption rate [micromol cm-3 yr-1]
> Ks      <- 0.005     # O2 saturation constant [micromol cm-3]
```

and the spatial extent of the model discretized (grid definition).

```
> R <- 0.025            # radius of the agggregate [cm]
> N <- 100              # number of grid layers
> grid <- setup.grid.1D(x.up=0,L=R,N=N)
```

We assume that porosity (the volume fraction) and the diffusion coefficient are constant. Both properties ae defined as a grid list.

```
> por.grid <- setup.prop.1D(value=por,grid=grid)
> D.grid <- setup.prop.1D(value=D,grid=grid)
```

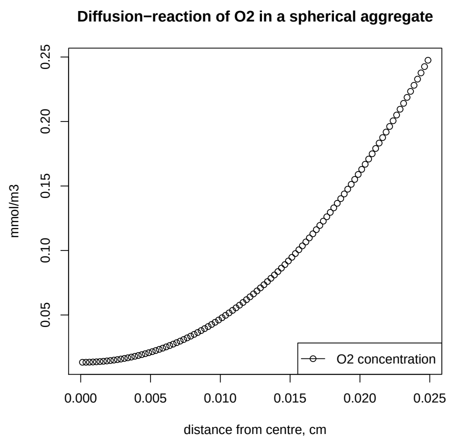

Figure 6: Solution of the aggregate model - see text for R -code

In this model, each "grid cell" is equivalent to a thin spherical region; the further away from the origin, the larger the surface area of this layer.

To take into account this expanding surface, we define a grid with the surfaces of these spherical layers. Function `sphere.surf` calculates the surface of a sphere.

```
> sphere.surf <- function (x) 4*pi*x^2
> A.grid  <- setup.prop.1D(func=sphere.surf,grid=grid)
```

The model is then solved to steady-state

```
> O2.agg <- steady.1D (y = runif(N), func=Aggregate.Model, nspec=1,
+                       positive=TRUE, atol=1e-10)
```

and the output plotted

```
> par(mfrow=c(1,1))
> plot(grid$x.mid,O2.agg$y,xlab="distance from centre, cm",
+ ylab="mmol/m3",
+ main="Diffusion-reaction of O2 in a spherical aggregate")
> legend ("bottomright",pch=c(1,18),lty=1,col=c("black"),
+         c("O2 concentration"))
```

Note that in this model, we have imposed a zero-flux boundary at the centre of the sphere (or the "upstream" boundary) (zero-gradient is the default boundary condition). This makes sense as we assume that the sphere is symmetrical. However, oxygen is dffusing into the aggregate, at the "downstream" boundary.

The influx, per unit of surface area is in `O2.agg$flux.down`.

```
>   O2.agg$flux.up
```

```
[1] 0
```

```
>   O2.agg$flux.down
```

```
[1] -6384.782
```

Calculating the mass conservation for this model is slightly more complex than in previous example, as we need to take volumetric averages of the rates, AND correct for the porosity effect.

The volume in each spherical layer is simply equal to the surface at the centre of the layer (`A.grid$mid`) times the grid size (`grid$dx`).

```
>  Volume <- A.grid$mid * grid$dx
> (Consump <- - sum(O2.agg$reac * Volume * por.grid$mid))
```

```
[1] 50.14596
```

The total flux into the aggregate equals the flux per unit surface times the surface area at the downstream boundary:

```
>  (Fluxin <- - O2.agg$flux.down*A.grid$int[N+1])
```

```
[1] 50.14596
```

The flux equals the total consumption, up to a certain numerical precision, hence mass is conserved.

```
>   Consump - Fluxin
```

```
[1] 2.486900e-13
```

4. volumetric advective-diffusive transport in an aquatic system

The volumetric reaction-transport equation in 1-D is best derived in two steps: first we rewrite the discretisation over a numerical grid:

$$\frac{dC_i}{dt} = -\frac{1}{A} \cdot \frac{\Delta_i(A \cdot J)}{\Delta x_i} + reac_i$$

as

$$\frac{dC_i}{dt} = -\frac{\Delta_i(A \cdot J)}{\Delta V_i} + reac_i$$

where we defined cell-volume as the product of mid-surface and cell thickness

$$V_i = A_i \Delta x_i$$

and then redefine the mass fluxes:

$$A \cdot J = -D \cdot A \frac{\Delta C}{\Delta x} + A \cdot u \cdot C$$

as :

$$A \cdot J = -E \Delta C + Q \cdot C$$

where $Q = A \cdot u$ and $E = \frac{D \cdot A}{\Delta x}$ are the flow rate and the bulk dispersion coefficient respectively, both in units of $L^3 t^{-1}$.

Volumetric transport implies the use of total flows (mass per unit of time) rather than fluxes (mass per unit of area per unit of time) as is done in `tran.1D`.

`tran.volume.1D` implements this in **ReacTran** .

The `tran.volume.1D` routine is particularly suited for modelling channels (like rivers, estuaries) where the cross-sectional area changes, but where this area change is not explicitly modelled (as a function of time).

4.1. R-function tran.volume.1D

The default input for the `tran.volume1D()` function in R is:

```
tran.volume.1D(C, C.up = C[1], C.down = C[length(C)],
  C.lat=0, F.up=NULL, F.down=NULL, F.lat=NULL,
  Disp=NULL, flow = 0, flow.lat=NULL, AFDW = 1, V=NULL,
  full.check = FALSE, full.output = FALSE)
```

with the following arguments:

- `C, C.up, C.down`. The concentrations in the centre of each grid cell, a vector of length N (`C`) and at the upstream or downstream boundary, one value (`C.up, C.down`).
- `F.up, F.down`. Total input at the upstream and downstream boundaries, (Mt^{-1}).
- `F.lat`. Total input laterally, defined at the grid cells, (Mt^{-1}).

- `Disp`, the bulk dispersion coefficients, (L^3t^{-1}).
- `Q`, the water flow rate, (L^3t^{-1}).
- `AFDW` the weights used in the finite difference approximation for flow (-).
- `V`, the volume of each grid cell (L^3).
- `full.check`, when `TRUE`, the consistency of the input is checked.
- `full.output`, when `TRUE` full output is returned.

4.2. An estuarine model

Consider the following example that models organic carbon decay in an estuary.

Two scenarios are simulated: the baseline includes only input of organic matter at the upstream boundary. The second scenario simulates the input of an important side river halfway the estuary.

The model is formulated in function `river.model`

```
> river.model <- function (t=0,OC,pars=NULL)
+ {
+   tran <- tran.volume.1D(C=OC,F.up=F.OC,F.lat=F.lat,Disp=Disp,
+       flow=flow,V=Volume)
+   reac <- - k*OC
+
+   return(list(dCdt = tran$dC + reac,
+               F.up = tran$F.up, F.down = tran$F.down,
+               F.lat = sum(tran$F.lat)))
+ }
```

The estuary is 100 km long (`lengthEstuary`; it is subdivided in 500 grid cells (`nbox`).

```
> nbox          <- 500                 # number of grid cells
> lengthEstuary <- 100000              # length of estuary [m]
> BoxLength     <- lengthEstuary/nbox # [m]
```

The estuarine cross-sectional area widens sigmoidally towards the estuarine mouth (`CrossArea`); based on this area and the lenght of a box, the volume of each box is easily estimated.

```
> Distance      <- seq(BoxLength/2, by=BoxLength, len=nbox) # [m]
> CrossArea <- 4000 + 72000 * Distance^5 /(Distance^5+50000^5)
> Volume  <- CrossArea*BoxLength
```

The dispersion coefficient (`Disp`) and the upstream flow rate (`flow`) are parameters.

```
> Disp    <- 1000   # m3/s, bulk dispersion coefficient
> flow    <- 180    # m3/s, mean river flow
```

The organic carbon input on upstream boundary (`F.OC`), the lateral input of carbon (`F.lat.0`) and the decay rate of organic carbon (`k`) are declared next:

```
> F.OC    <- 180                # input organic carbon [mol s-1]
> F.lat.0 <- F.OC               # lateral input organic carbon [mol s-1]
> k       <- 10/(365*24*3600)  # decay constant organic carbon [s-1]
```

In the first scenario, the lateral flux of material is zero.

```
> F.lat <- rep(0,length.out=nbox)
```

The model is solved using **rootSolve** function `steady.1D` which finds the steady-state solution, given an initial guess, `y` (here simply 500 random numbers).

```
> sol  <- steady.1D(y=runif(nbox), fun=river.model, nspec=1, atol=1e-15,
+                     rtol=1e-15, positive = TRUE)
> Conc1 <- sol$y
```

In the second scenario, there is lateral input of organic carbon:

```
> F.lat <- F.lat.0*dnorm(x=Distance/lengthEstuary,
+                         mean = Distance[nbox/2]/lengthEstuary,
+                         sd = 1/20, log = FALSE) /nbox
> sol2<- steady.1D(runif(nbox), fun=river.model, nspec=1, atol=1e-15,
+                     rtol=1e-15, positive = TRUE)
> Conc2 <- sol2$y
```

We set the tolerances for the steady-state calculation (`atol` and `rtol`) to a very small value so that the mass budget is very tight (see below).
Finally the output is plotted.

```
> matplot(Distance/1000,cbind(Conc1,Conc2),lwd=2,
+ main="Organic carbon decay in an estuary",xlab="distance [km]",
+ ylab="OC Concentration [mM]",
+ type="l")
> legend ("topright",lty=1,col=c("black","red"),
+         c("baseline","with lateral input"))
```

and a budget estimated (total input = consumption)

```
> sol$F.up + sol$F.lat - sol$F.down - sum(sol$y*k*Volume)
```

```
[1] 2.842171e-13
```

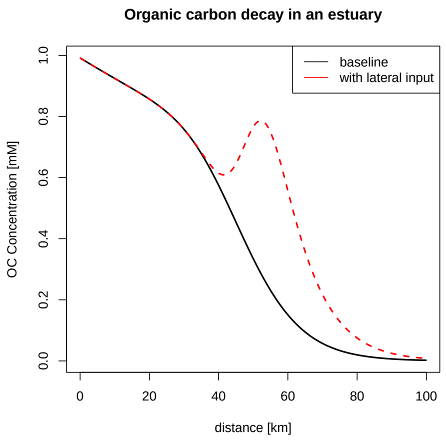

Figure 7: Solution of the estuarine model - see text for R -code

5. Transport in two dimensions

The function that performs transport in two dimensions is similar to its 1-D equivalent. Here is its default input

```
tran.2D ( C, C.x.up=C[1,], C.x.down=C[nrow(C),],
  C.y.up=C[,1], C.y.down=C[,ncol(C)],
  flux.x.up=NULL, flux.x.down=NULL, flux.y.up=NULL, flux.y.down=NULL,
  a.bl.x.up=NULL, a.bl.x.down=NULL, a.bl.y.up=NULL, a.bl.y.down=NULL,
  D.grid=NULL, D.x=NULL, D.y=D.x,
  v.grid=NULL, v.x=0, v.y=0,
  AFDW.grid=NULL, AFDW.x=1, AFDW.y=AFDW.x,
  VF.grid=NULL,VF.x=1, VF.y=VF.x,
  A.grid=NULL, A.x=1, A.y=1,
  grid=NULL, dx=NULL, dy=NULL,
  full.check = FALSE, full.output = FALSE)

}
```

where

- `C, C.x.up, C.x.down, C.y.up, C.y.down` are the concentration in the 2-D grid (`C`) and the boundary concentrations
- `flux.x.up, flux.x.down, ...` are the fluxes prescribed at the boundaries,
- `a.bl.up, ...` are the exchange coefficients for transfer across the various boundary layers
- `D.x, ...` are the diffusion coefficients
- `v.x, ...` are the advective velocities
- `AFDW.x, ...` are the weights used in the numerical approximation of the advective component
- `VF.x, ...` are the volume fractions of the various phases
- `A.x, ...` are the surface areas
- `dx, dy, grid` define the discretisation grid
- `full.check, full.output` whether full output needs writing or the input needs checking

By making clever use of the surface areas, it is possible to describe reactive transport processes in a 2-D channel with variable surface area, e.g. for widening estuaries that are stratified.

It is also possible to mimic certain 3-D applications, e.g. by assuming cylindrical symmetries.

5.1. example of transport in 2 dimensions

We model the dynamics of oxygen, on a 2-D grid, and subjected to diffusion in the two directions, to advection in y-direction, from left to right, first-order consumption and a source in a central spot (e.g. an animal ventilating its burrow).

At the upper boundary, the concentration is 300 mM, the other boundaries are zero-flux boundaries.

We start by defining the parameters and the grid

```
> n     <- 100           # number of grid cells
> dy    <- dx <- 100/n   # grid size
> Dy    <- Dx <- 5   # diffusion coeff, X- and Y-direction
> r     <- -0.02     # production/consumption rate
> Bc    <- 300       # boundary concentration
> irr   <- 20        # irrigation rate
> vx    <- 1         # advection
```

As initial concentrations we assume that oxygen is 0 everywhere.

```
> y  <- matrix(nr=n,nc=n,0)
```

In the model function, the state variables are passed as one vector (`y`). They are recast as a matrix first (`CONC`).

```
> Diff2D <- function (t, y, parms, N)
+
+ {
+   CONC <- matrix(nr=N, nc=N, y)
+ # Transport
+
+    Tran    <-tran.2D(CONC, D.x=Dx, D.y=Dy, C.y.down=Bc,
+                      dx=dx, dy=dy, v.x = vx)
+
+ # transport + reaction
+    dCONC   <- Tran$dC + r*CONC
+
+ # Bioirrigation in a central spot
+     mid <- N/2
+     dCONC[mid,mid] <- dCONC[mid,mid]  + irr*(Bc-CONC[mid,mid])
+
+   return (list(as.vector(dCONC)))
+ }
```

The model is solved in two ways.

- `steady.2D` solves the steady-state condition
- `ode.2D` runs the model dynamically

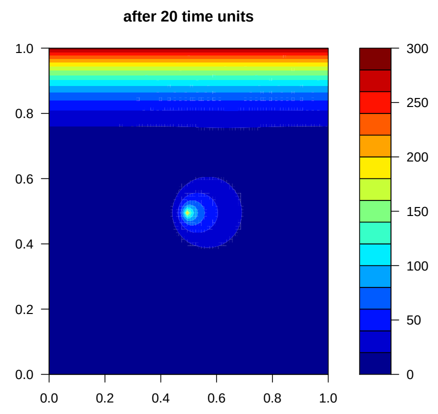

Figure 8: Solution of the 2-dimensional diffusion and irrigation model, after 20 time units - see text for R -code

The model consists of 100*100 = 10000 equations. We print the time it takes to obtain these solutions (in seconds).

Note that for both these methods, we need to pass the dimension of the problem (e.g. the number of boxes in x and y-direction), and the work- space required (`lrw`).

```
> print(system.time(
+ std  <-  steady.2D(func=Diff2D, y=as.vector(y), time=0, N=n,
+           parms=NULL, lrw=1000000, dimens=c(n,n),
+           nout=0, positive=TRUE)
+ ))
```

```
   user  system elapsed 
   1.08    0.00    1.08 
```

We run the model for 200 time units, producing output every 5 units.

```
> times <- seq(0,100,5)
> print(system.time(
+   out2  <-  ode.2D(func=Diff2D, y=as.vector(y), times=times, N=n,
+              parms=NULL, lrw=10000000, dimens=c(n,n))
+ ))
```

```
   user  system elapsed 
   7.47    0.01    7.56 
```

```
> mat  <-  matrix(nr=n,nc=n,out2[5,-1])
> filled.contour(mat,zlim=c(0,Bc), color=femmecol,main="after 20 time units")
```

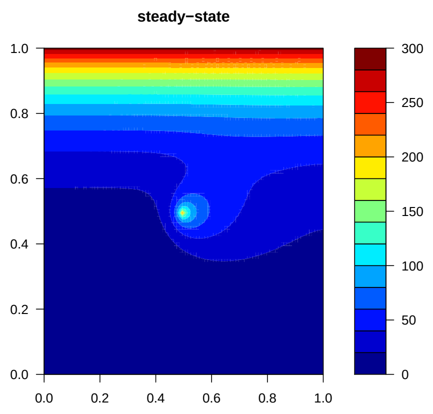

Figure 9: Steady-state solution of the 2-dimensional diffusion and irrigation model - see text for R -code

```
> mat  <-  matrix(nr=n,nc=n,std$y)
> filled.contour(mat,zlim=c(0,Bc), color=femmecol,main="steady-state")
```

6. finally

This vignette was made with Sweave (Leisch 2002).

References

Leisch F (2002). "Sweave: Dynamic Generation of Statistical Reports Using Literate Data Analysis." In W Härdle, B Rönz (eds.), "Compstat 2002 - Proceedings in Computational Statistics," pp. 575–580. Physica Verlag, Heidelberg. ISBN 3-7908-1517-9, URL `http://www.stat.uni-muenchen.de/~leisch/Sweave`.

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

Soetaert K, Meysman F (2009). *ReacTran: Reactive transport modelling in 1D, 2D and 3D*. R package version 1.1.

Soetaert K, Petzoldt T, Setzer RW (2010). "Solving Differential Equations in R: Package deSolve." *Journal of Statistical Software*, **33**(9), 1–25. ISSN 1548-7660. URL `http://www.jstatsoft.org/v33/i09`.

Affiliation:

Karline Soetaert
Centre for Estuarine and Marine Ecology (CEME)
Netherlands Institute of Ecology (NIOO)
4401 NT Yerseke, Netherlands
E-mail: `k.soetaert@nioo.knaw.nl`
URL: `http://www.nioo.knaw.nl/ppages/ksoetaert`

Filip Meysman
Centre for Estuarine and Marine Ecology (CEME)
Netherlands Institute of Ecology (NIOO)
4401 NT Yerseke, Netherlands
E-mail: `f.meysman@nioo.knaw.nl`
URL: `http://www.nioo.knaw.nl/ppages/fmeysman`