Solving partial differential equations, using R package ReacTran

Karline Soetaert and Filip Meysman
Royal Netherlands Institute of Sea Research (NIOZ)
Yerseke
The Netherlands

Abstract

R -package **ReacTran** (Soetaert and Meysman 2012) contains functions to solve reactive-transport equations, as used e.g. in the environmental sciences.

Essentially, it

1. Provides functions that subdivide the spatial extent into a number of discrete grid cells.
2. Approximates the advective-diffusive transport term by finite differences or finite volumes.

The main package vignette (Soetaert and Meysman 2010) explains how **ReacTran** can be used to model reaction-transport phenomena.

However, the functions from **ReacTran** can be use to solve more general types of partial differential equations ($\leq$ order 2).

In this vignette, show how the package can be used to solve partial differential equations of the parabolic, hyperbolic and elliptic type, providing one example each.

Keywords: Partial Differential Equations, hyperbolic, parabolic, elliptic, R .

1. Partial differential equations

In **partial differential equations** (PDE), the function has several independent variables (e.g. time and depth) and contains their partial derivatives.

A first step to solve partial differential equations (PDE), is to discretise one or more of the independent variables.

Usually, the independent variable "time" is not discretised, while other variables (e.g. spatial axes) are discetised, so that a set of ODE is obtained, which can be solved with appropriate initial values solvers from package **deSolve** (Soetaert, Petzoldt, and Setzer 2010). This technique, the method-of-lines, applies to hyperbolic and parabolic PDEs.

For time-invariant problems, usually all independent variables are discretised, and the resulting algebraic equations solved with root-solving functions from package **rootSolve** (Soetaert 2009).

Functions `tran.1D`, `tran.2D`, and `tran.3D` from R package **ReacTran** implement finite difference approximations of the general diffusive-advective transport equation, which for 1-D is:

$$-\frac{1}{A_x\xi_x} \cdot (\frac{\partial}{\partial x} A_x \cdot (-D \cdot \frac{\partial \xi_x C}{\partial x}) - \frac{\partial}{\partial x}(A_x \cdot v \cdot \xi_x C))$$

Here D is the "diffusion coefficient", v is the "advection rate", and A_x and ξ are the surface area and volume fraction respectively.

Assuming that A, ξ, D and v are constant along x, we can rewrite this in a more general form:

$$D\frac{\partial^2 C}{\partial x^2} - u\frac{\partial C}{\partial x}$$

In which the first term is a second-order, the second term a first-order derivative.

R -function `tran.1D` is defined as (simplified):

```
tran.1D <- function (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, ...)
```

where `C.up` and `C.down` are the upstream and downstream boundary values, `flux.up` and `flux.down` are the upstream and downstream fluxes, `v` and `D` are the advection and diffusion coefficient respectively, `A` is the surface area, `x` contains the grid, and `VF` is the volume fraction (ξ). For the other arguments, see the help file of `tran.1D`.

A suitable *grid* can be generated with functions `setup.grid.1D` and `setup.grid.2D` (there is no corresponding 3D function), while a *property* can be added to this grid using functions `setup.prop.1D`, and `setup.prop.2D`. These latter two functions are useful to have the variable surface areas, volume fractions, advection and diffusion rates being defined at the correct places of the grid.

These functions are defined as (simplified):

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

setup.grid.2D <- function(x.grid = NULL, y.grid = NULL)

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

setup.prop.2D <- function (func = NULL, value = NULL, grid, y.func = func,
      y.value = value, ...)
```

2. A parabolic PDE

As an example of the parabolic type, consider the 1-D diffusion-reaction model, in spherical, cylindrical and cartesian coordinates, defined for r in $[0, 10]$:

$$\begin{aligned}\frac{\partial C}{\partial t} &= \frac{1}{r^2} \cdot \frac{\partial}{\partial r}\left(r^2 \cdot D \cdot \frac{\partial C}{\partial r}\right) - Q \\ \frac{\partial C}{\partial t} &= \frac{1}{r} \cdot \frac{\partial}{\partial r}\left(r \cdot D \cdot \frac{\partial C}{\partial r}\right) - Q \\ \frac{\partial C}{\partial t} &= \frac{\partial}{\partial r}\left(D \cdot \frac{\partial C}{\partial r}\right) - Q\end{aligned}$$

with t the time, r the (radial) distance from the origin, Q, the consumption rate, and with boundary conditions (values at the model edges):

$$\begin{aligned}\frac{\partial C}{\partial r}_{r=0} &= 0 \\ C_{r=10} &= Cext\end{aligned}$$

To solve this model in R , first the 1-D model `Grid` is defined; it divides 10 cm (`L`) into 1000 equally-sized boxes (`N`).

```
Grid <- setup.grid.1D(N = 1000, L = 10)
```

Next the properties r and r^2 are defined on this grid:

```
r  <- setup.prop.1D(grid = Grid, func = function(r) r)
r2 <- setup.prop.1D(grid = Grid, func = function(r) r^2)
```

The model equation includes a transport term, approximated by **ReacTran** function `tran.1D` and a consumption term (`Q`); only the downstream boundary condition, prescribed as a concentration (`C.down`) needs to be specified, as the zero-gradient at the upstream boundary is the default:

```
library(ReacTran)
pde1D <- function(t, C, parms, A = 1) {
+     tran <- tran.1D(C = C, A = A, D = D, C.down = Cext, dx = Grid)$dC
+     list(tran - Q)
+ }
```

The model parameters are defined:

```
D <- 1
Q <- 1
Cext <- 20
```

2.1. Steady-state solution

In a first application, the model is solved to *steady-state*, which retrieves the condition where the concentrations are invariant, e.g. for the cylindrical coordinate case:

$$0 = \frac{1}{r^2} \cdot \frac{\partial}{\partial r} \left(r^2 \cdot D \cdot \frac{\partial C}{\partial r} \right) - Q$$

In R , steady-state conditions can be estimated using functions from package **rootSolve** which implement amongst others a Newton-Raphson algorithm (Press, Teukolsky, Vetterling, and Flannery 1992). For 1-dimensional models, `steady.1D` should be used. The initial "guess" of the steady-state solution (`y`) is unimportant; here we take simply `N` random numbers. Argument `nspec = 1` informs the solver that only one component is described.

Although a system of 1000 equations needs to be solved, this takes only a fraction of a second:

```
 library(rootSolve)
 Cartesian   <- steady.1D(y = runif(Grid$N),
+   func = pde1D, parms = NULL, nspec = 1, A = 1)
 Cylindrical <- steady.1D(y = runif(Grid$N),
+   func = pde1D, parms = NULL, nspec = 1, A = r)

 print(system.time(
+   Spherical   <- steady.1D(y = runif(Grid$N),
+     func = pde1D, parms = NULL, nspec = 1, A = r2)
+ ))

   user  system elapsed 
      0       0       0 
```

The values of the state-variables (`y`) are plotted against the radial distance, in the middle of the grid cells (`Grid$x.mid`). We use `rootSolve`'s plot method to do so. This function accepts several steady-state outputs at once:

```
 plot(Cartesian, Cylindrical, Spherical, grid = Grid$x.mid,
+    main = "steady-state PDE", xlab = "x", ylab = "C",
+    col = c("darkgreen", "blue", "red"), lwd = 3, lty = 1:3)
 legend("bottomright", c("cartesian", "cylindrical", "spherical"),
+   col = c("darkgreen", "blue", "red"), lwd = 3, lty = 1:3)
```

The analytical solutions compare well with the numerical approximation for all three cases:

```
 max(abs(Q/6/D*(r2$mid - 10^2) + Cext - Spherical$y))

[1] 5.820809e-05

 max(abs(Q/4/D*(r2$mid - 10^2) + Cext - Cylindrical$y))
```

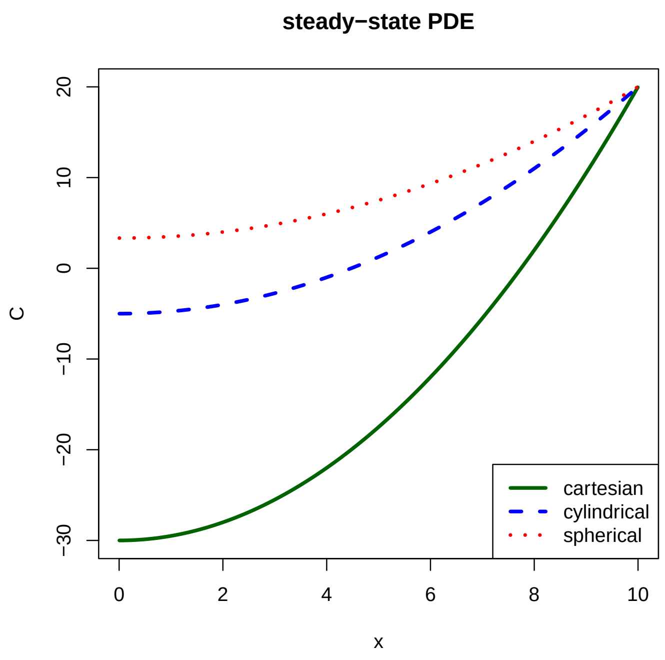

Figure 1: Steady-state solution of the 1-D diffusion-reaction model

```
[1] 6.250002e-06
```

```
 max(abs(Q/2/D*(r2$mid - 10^2) + Cext - Cartesian$y))
```

```
[1] 1.25e-05
```

Note that there is no automatic error checking/control here, so to reduce this error, the number of boxes can be increased.

2.2. The method of lines

Next the model (for spherical coordinates) is run dynamically for 100 time units using **deSolve** function `ode.1D`, and starting with an initially uniform distribution (`y = rep(1, Grid$N)`):

```
 require(deSolve)
 times <- seq(0, 100, by = 1)
 system.time(
+   out <- ode.1D(y = rep(1, Grid$N), times = times, func = pde1D,
+     parms = NULL, nspec = 1, A = r2)
+ )
```

```
   user  system elapsed
   0.13    0.06    0.19
```

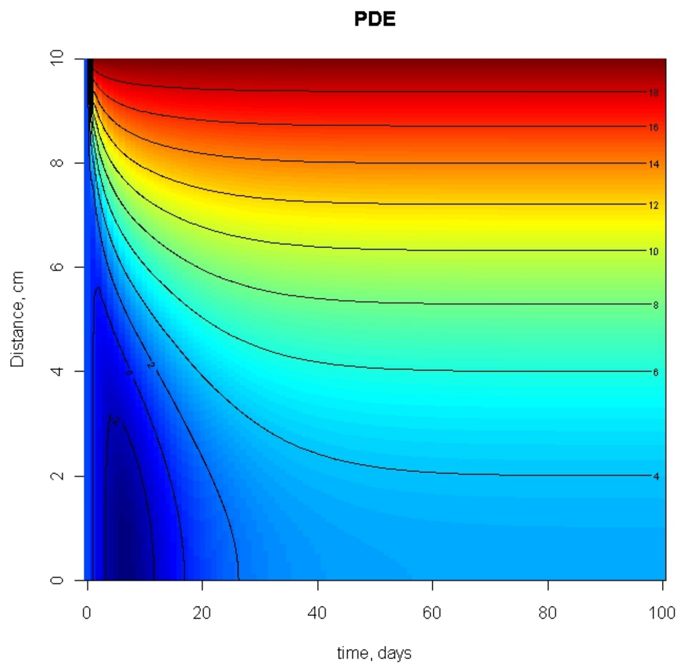

Figure 2: Dynamic solution of the 1-D diffusion-reaction model

Here, `out` is a matrix, whose 1st column contains the output times, and the next columns the values of the state variables in the different boxes:

```
tail(out[, 1:4], n = 3)
```

```
        time        1        2        3
 [99,]    98 3.332278 3.332303 3.332366
[100,]    99 3.332383 3.332408 3.332471
[101,]   100 3.332478 3.332503 3.332566
```

We plot the result using a blue-yellow-red color scheme, and using deSolve's S3 method `image`:

```
 image(out, grid = Grid$x.mid, xlab = "time, days",
+      ylab = "Distance, cm", main = "PDE", add.contour = TRUE)
```

3. A hyperbolic PDE

The equation for a wave travelling in one direction (x) is given by:

$$\frac{\partial^2 u}{\partial t^2} = c^2 \frac{\partial^2 u}{\partial x^2} \tag{1}$$

where c is the propagation speed of the wave, and u is the variable that changes as the wave passes. This equation is second-order in both t and x. The wave equation is the prototype of a "hyperbolic" partial differential equation.

For it to be solved in R , the equation is rewritten as two coupled equations, first-order in time:

$$\frac{\partial u_1}{\partial t} = u_2 \tag{2}$$

$$\frac{\partial u_2}{\partial t} = c^2 \frac{\partial^2 u_1}{\partial x^2} \tag{3}$$

We solve the equation with the following initial and boundary conditions:

$$\begin{aligned} u_1(0,x) &= \exp^{-0.05x^2} \\ u_2(0,x) &= 0 \\ u_{t,-\infty} &= 0 \\ u_{t,\infty} &= 0 \end{aligned}$$

where the first condition represents a Gaussian pulse.

The implementation in R starts with defining the box size `dx` and the grid, `xgrid`. To comply with the boundary conditions (which are defined at ∞), the grid needs to be taken large enough such that u remains effectively 0 at the boundaries, for all run times.

Here, the grid extends from -100 to 100:

```
dx    <- 0.2
xgrid <- setup.grid.1D(-100, 100, dx.1 = dx)
x     <- xgrid$x.mid
N     <- xgrid$N
```

The initial condition, `yini` and output `times` are defined next:

```
uini  <- exp(-0.05 * x^2)
vini  <- rep(0, N)
yini  <- c(uini, vini)
times <- seq (from = 0, to = 50, by = 1)
```

The wave equation derivative function first extracts, from state variable vector `y` the two properties `u1, u2`, both of length `N`, after which **ReacTran** function `tran.1D` performs transport of `u1`. The squared velocity (c^2) is taken as 1 (`D=1`):

```
 wave <- function (t, y, parms) {
+   u1 <- y[1:N]
+   u2 <- y[-(1:N)]
+
+   du1 <- u2
+   du2 <- tran.1D(C = u1, C.up = 0, C.down = 0, D = 1, dx = xgrid)$dC
+   return(list(c(du1, du2)))
+ }
```

The wave equation can be solved efficiently with a non-stiff solver such as the Runge-Kutta method `ode45`.

```
 out <- ode.1D(func = wave, y = yini, times = times, parms = NULL,
+          nspec = 2, method = "ode45", dimens = N, names = c("u", "v"))
```

We now plot the results (Fig. 3) using `deSolve`s function `matplot.1D`; intial condition in black, the values for selected time points in `darkgrey`; a `legend` with times is written.

```
 matplot.1D(out, which = "u", subset = time %in% seq(0, 50, by = 10),
+    type = "l", col = c("black", rep("darkgrey", 5)), lwd = 2,
+    grid = x, xlim = c(-50,50))
 legend("topright", lty = 1:6, lwd = 2, col = c("black", rep("darkgrey", 5)),
+        legend = paste("t = ",seq(0, 50, by = 10)))
```

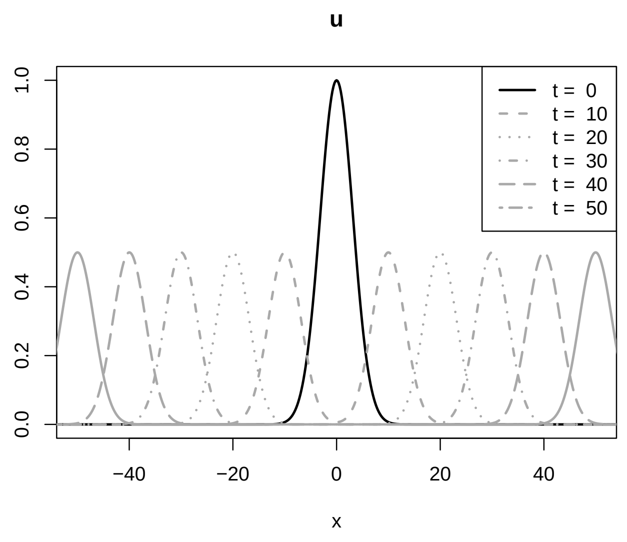

Figure 3: The 1-D wave equation; black = initial condition; grey: several time lines

S3-method `image` can also be used to generate `persp`-like plots:

```
 par(mar=c(0,0,0,0))
 image(out, which = "u", method = "persp", main = "",
+          border = NA, col = "lightblue", box = FALSE,
+          shade = 0.5, theta = 0, phi = 60)
```

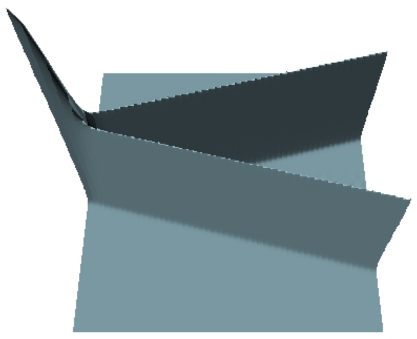
Figure 4: The 1-D wave equation as a persp plot

You may also want to try the following "movie":

```
plot.1D(out, grid = x, which = "u", type = "l",
        lwd = 2, ylim = c(0, 1), ask = TRUE)
```

4. An elliptic PDE

The final example describes again a diffusion-reaction system with production p, consumption rC, and diffusive transport (diffusion coefficients Dx, Dy) of a substance C in 2 dimensions (x, y); the boundaries are prescribed as zero-gradient (the default).

$$\frac{\partial C}{\partial t} = \frac{\partial}{\partial x}[D_x \cdot \frac{\partial C}{\partial x}] + \frac{\partial}{\partial y}[D_y \cdot \frac{\partial C}{\partial y}] - rC + p_{xy} \tag{4}$$

The parameter p_{xy} is the production rate, which is zero everywhere except for 50 randomly positioned spots where it is 1.

Transport is performed by `ReacTran` function `tran.2D`; the state variable vector (`y`) is recast in matrix form (`CONC`) before it is transported. The first-order consumption rate `-r * CONC` is added to the rate of change due to transport (`Tran$dC`). Production, `p` is added to 50 cells indexed by `ii`, and which are randomly selected from the grid. The function returns a list, containing the derivatives, as a vector:

```
 require(ReacTran)
 pde2D <- function (t, y, parms) {
+   CONC <- matrix(nr = n, nc = n, y)
+   Tran <- tran.2D(CONC, D.x = Dx, D.y = Dy, dx = dx,  dy = dy)
+   dCONC <- Tran$dC - r * CONC
+   dCONC[ii]<- dCONC[ii] + p
+   return(list(as.vector(dCONC)))
+ }
```

Before running the model, the grid sizes (`dx, dx`), diffusion coefficients (`Dx, Dy`), 1^{st} order consumption rate (`r`) are defined. There are 100 boxes in `x`- and `y` direction (`n`). Furthermore, we assume that the substance is produced in 50 randomly chosen cells (`ii`) at a constant rate (`p`):

```
 n <- 100
 dy <- dx <- 1
 Dy <- Dx <- 2
 r <- 0.001
 p <- runif(50)
 ii <- trunc(cbind(runif(50) * n, runif(50) * n) + 1)
```

The steady-state is found using **rootSolve** 's function `steady.2D`. It takes as arguments amongst other the dimensionality of the problem (`dimens`) and the length of the work array needed by the solver (`lrw = 600000`). It takes less than 0.5 seconds to solve this 10000 state variable model.

```
 require(rootSolve)
 Conc0 <- matrix(nr = n, nc = n, 10)
 print(system.time(ST <- steady.2D(y = Conc0, func = pde2D, parms = NULL,
+     dimens = c(n, n), lrw = 6e+05)))
```

```
   user  system elapsed 
   0.05    0.01    0.08 
```

```
image(ST, main = "steady-state 2-D PDE")
```

References

Press WH, Teukolsky SA, Vetterling WT, Flannery BP (1992). *Numerical Recipes in FORTRAN. The Art of Scientific Computing.* 2nd edition. Cambridge University Press.

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 (2010). *ReacTran: R-package ReacTran: Reactive Transport Modelling in R.* R package vignette.

Soetaert K, Meysman F (2012). "Reactive transport in aquatic ecosystems: Rapid model prototyping in the open source software R." *Environmental Modelling and Software*, **32**, 49–60.

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
Royal Netherlands Institute of Sea Research (NIOZ)
4401 NT Yerseke, Netherlands
E-mail: `karline.soetaert@nioz.nl`
URL: `http://www.nioz.nl`

Filip Meysman
Royal Netherlands Institute of Sea Research (NIOZ)
4401 NT Yerseke, Netherlands
E-mail: `filip.meysman@nioz.nl`
URL: `http://www.nioz.nl`

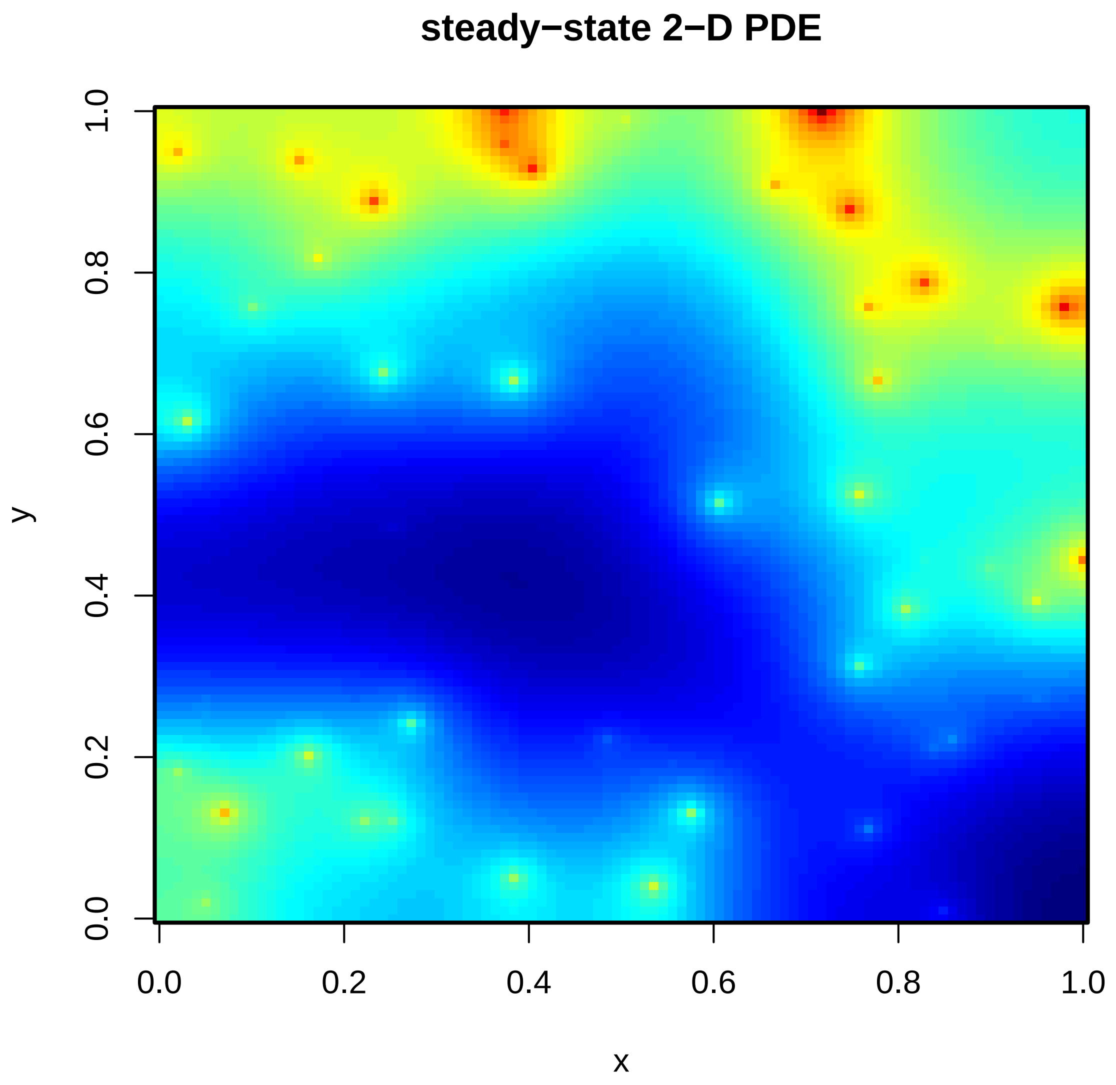

Figure 5: Steady-state solution of the 2-D diffusion-reaction model