A simple solving package

Neil MacLaren

Motivation

The *deSolve* package is fantastic for simulating deterministic ordinary differential equations (ODEs): it's fast, has great documentation, easy to use (compared to other solvers I have tried), and has convenient output data. A recommended stochastic differential equation package, *sde*, didn't fit easily into my workflow. Additionally, I often need to evaluate a chosen model as either an ODE or an SDE across a range of parameter values. The *localsolver* package is a simple package that implements the Euler-Maruyama method for solving (Itô calculus) SDEs and provides a function which solves the same ODE or SDE with the same model parameters across a range of a control parameter.

Solving a simple ODE

A model of logistic growth is

$\frac{dx}{dt} = rx(1 - \frac{x}{K})$

where x is the number of individuals in a population, t is time, r is an intrinsic growth rate, and K a maximum population. When this equation is deterministic, as above, K is a true maximum and x never goes below zero (assuming a reasonable initial value).

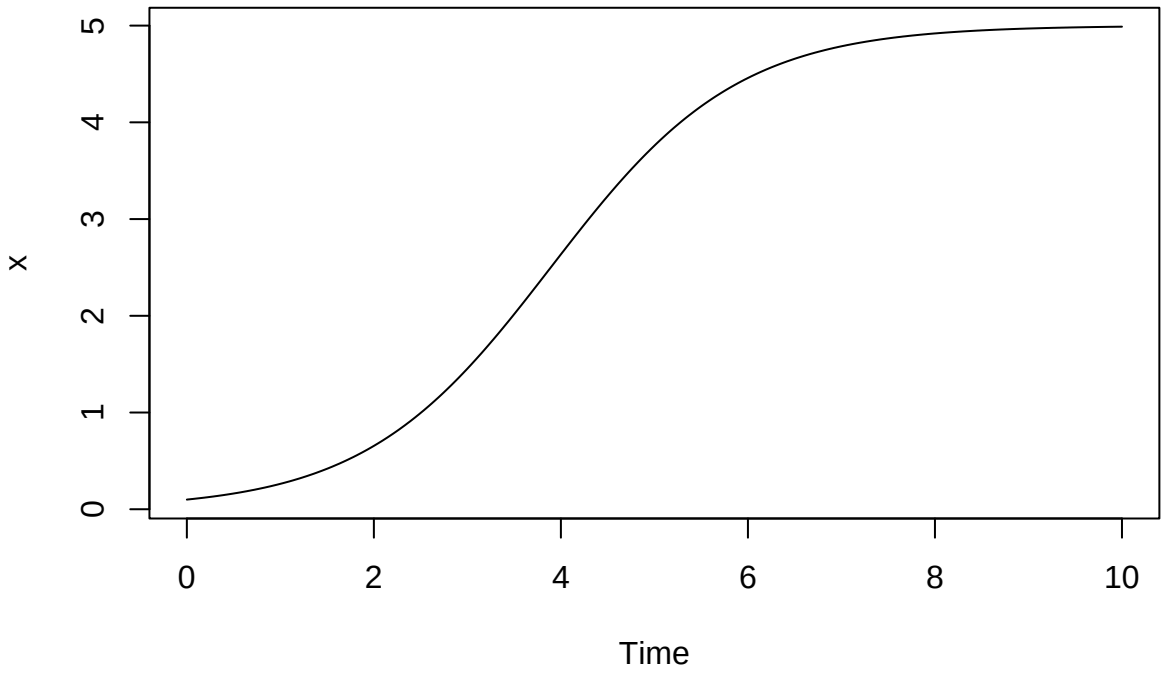
Implementing this equation with *deSolve* can be done as follows:

```
library(deSolve)

.growth <- list(xinit = 0.1, r = 1, K = 5)
growth <- function(t, x, params) {
    with(params, {
        dx <- r*x*(1 - (x/K))
        return(list(dx))
    })
}
```

I like to use a list of standard model parameters (including standard initial values), which I give a dot name which matches the model function. So, `.growth` is a named list (supporting, for example, easy access with `with()` calls) which includes an initial value ("xinit") and the two model parameters r and K. The function `growth()` is written in deSolve's preferred format and returns a derivative. (As a side note, `growth()` returns the derivative as the first element in a list because of functionality in *deSolve*, but I won't use that functionality here).

We can simulate this model at specific time steps, and plot the result:

```
simtimes <- seq(0, 10, by = 0.01)
x.ode <- ode(.growth$xinit, simtimes, growth, .growth)

plot(x.ode[, 1], x.ode[, 2], type = "l", xlab = "Time", ylab = "x")
```



The output from `ode()` is a matrix, the first column of which is the time variable values and the remaining columns are the values of the variables (e.g., x) at those times.

An SDE version

It should be easy for the user to say, "Ok, now show me that with noise!" Unfortunately, it's not that easy. Enter the *localsolver* package. *localsolver* provides a function, `sde()`, which attemps to make the transition from ODE to SDE as seamless as possible. We are now interested in the following model:

$dx = (rx(1 - \frac{x}{K}))dt + d\sigma W$

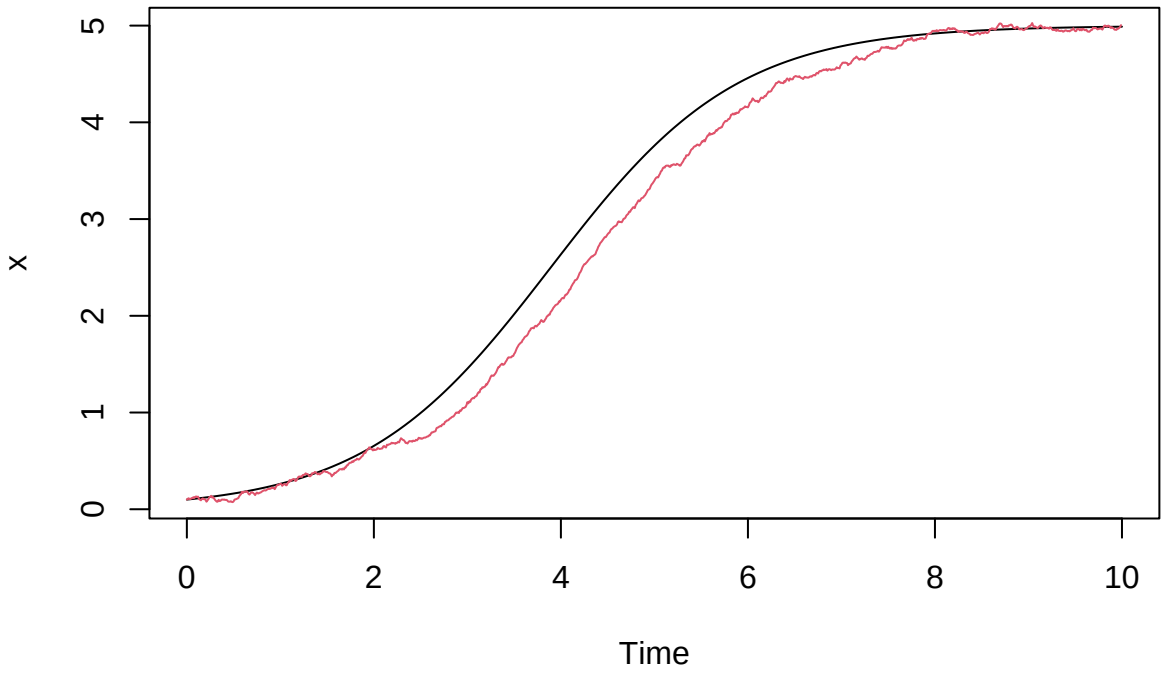
I hope I have written that correctly. In any case, our observable x is a sum of a deterministic process (i.e., $rx(1 - \frac{x}{K})$ and a noise process W with strength (standard deviation) σ. We can no longer rely on *deSolve*'s machinery to compute our answer for us, so `sde()` uses the brute force method to simulate each time step based on the initial conditions and each time step before the current one. Thus, `sde()` requires that you also specify the first and last time step, the Δt, and the noise strength σ. Our simulation code now looks like this:

```
library(localsolver)

.growth$sigma <- 0.1
control <- list(times = 0:10, deltaT = 0.01)

x.sde <- sde(.growth$xinit, control$times, growth, .growth, control)
```

Note that we are still using `.growth` and `growth()` from before. Additionally, we can pass a sequence of times—this is because I found it easier to think about time that way.

Let's plot the two results together:

```
plot(x.ode[, 1], x.ode[, 2], type = "l", xlab = "Time", ylab = "x")
lines(x.sde[, 1], x.sde[, 2], type = "l", col = 2)
```

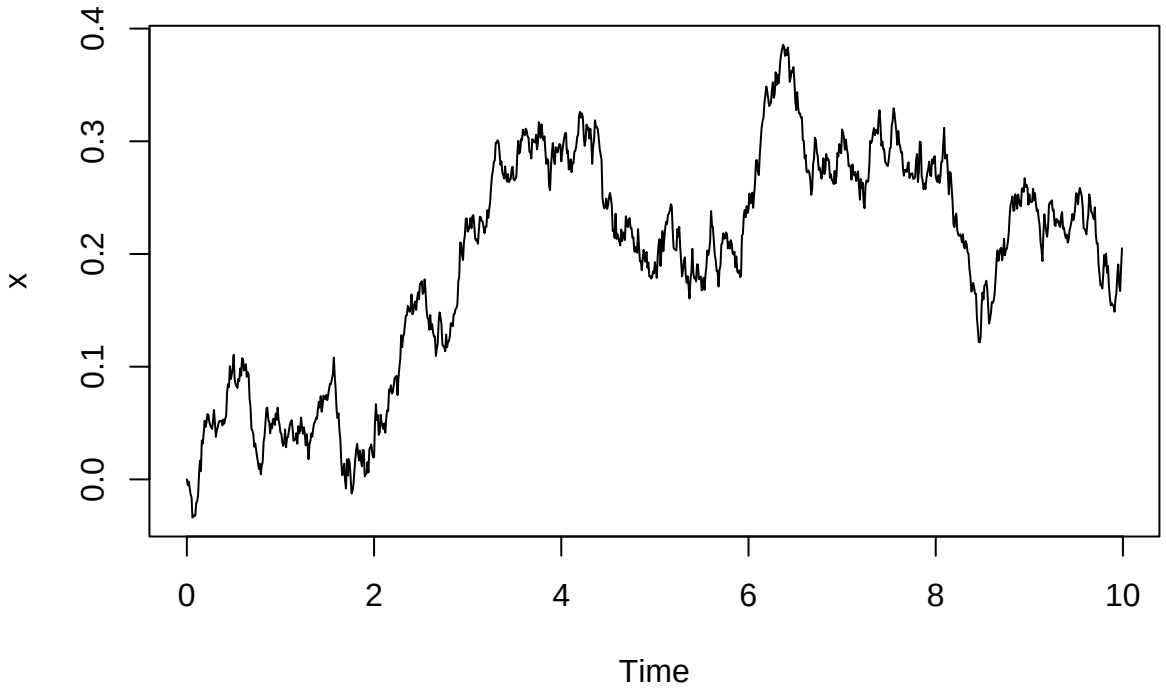


Brownian motion

Sometimes we need more than one variable. When the system can be modeled as a set of coupled single-variable systems, we can use an adjacency matrix method—more on that below. If each node in our system must be described by more than one variable, however, we need a different workflow, demonstrated in this section.

Let's look first at 1-dimensional Brownian motion with no drift:

```
.Brownian1D <- list(xinit = 0, mu = 0, sigma = 1e-1)
Brownian1D <- function(t, x, params) {
    with(params, {
        dx <- mu*x
        return(list(dx))
    })
}

x <- sde(.Brownian1D$xinit, control$times, Brownian1D, .Brownian1D, control)
plot(x[, 1], x[, 2], type = "l", xlab = "Time", ylab = "x")
```



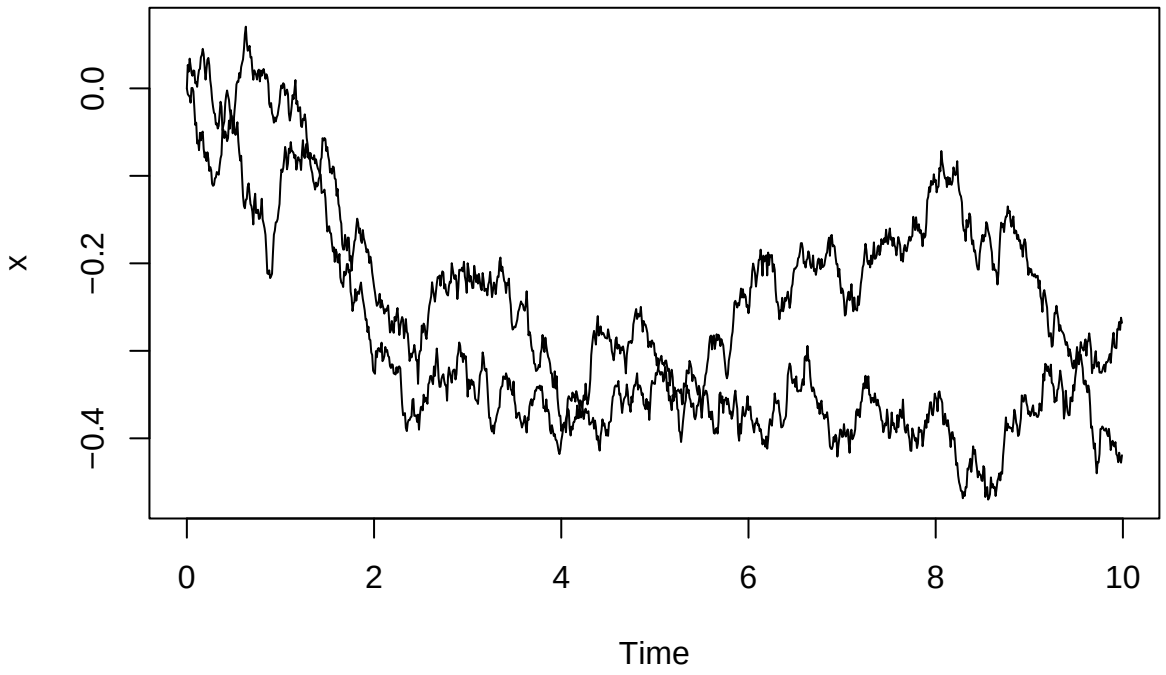
Note that our workflow is very similar—all we've had to do is write a different deterministic part of the model.

To make a 2-dimensional Brownian motion model we need to use a trick recommended in the *deSolve* documentation, as follows:

```
.Brownian2D <- list(xinit = c(y = 0, z = 0), mu = 0, sigma = 1e-1, N = 1, nstatevars = 2)
Brownian2D <- function(t, x, params) {
    with(params, {
        y <- x[1:N]
        z <- x[(N+1):(2*N)]

        dy <- mu*y
        dz <- mu*z
        return(list(dy, dz))
    })
}

res <- sde(.Brownian2D$xinit, control$times, Brownian2D, .Brownian2D, control)
```

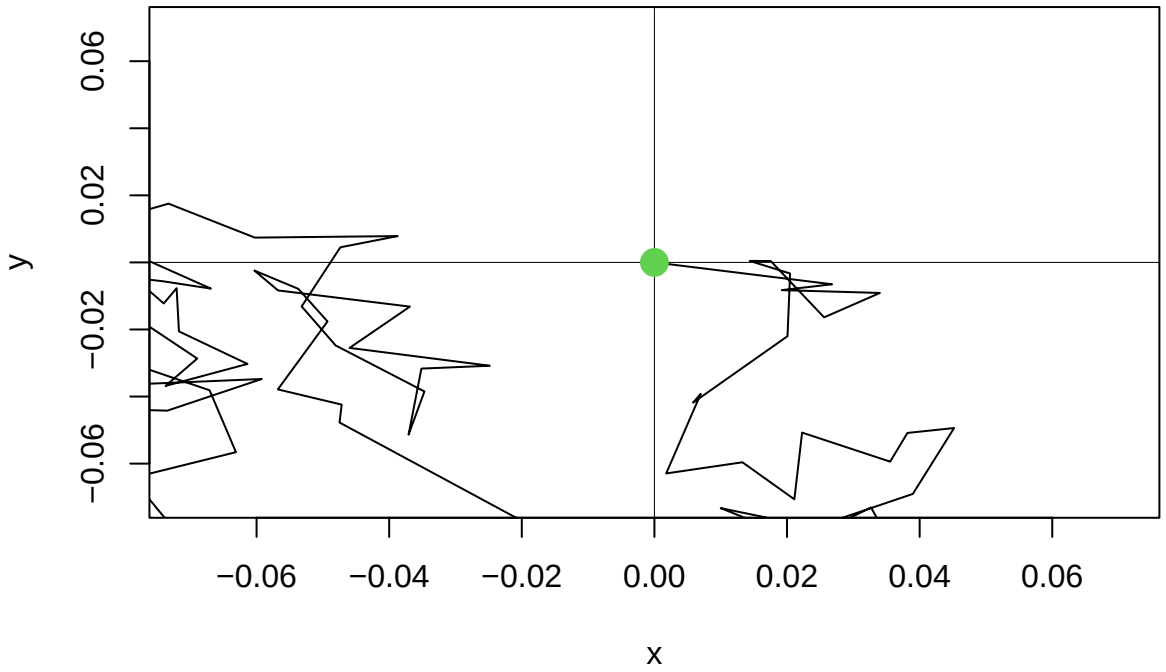
Our "derivative" now is the concatenation of the derivatives for our two variables. We can plot our result over time

```
matplot(res[, 1], res[, 2:3], type = "l", lty = 1, col = 1, lwd = 1, xlab = "Time", ylab = "x")
```



or showing the bivariate random walk:

```
xlim <- ylim <- c(-max(res[, 2:3]), max(res[, 2:3]))
plot(NULL, xlim = xlim, ylim = ylim, xlab = "x", ylab = "y")
abline(h = 0, lwd = 0.5)
abline(v = 0, lwd = 0.5)
lines(res[, 2], res[, 3])
points(res[1, 2], res[1, 3], col = 3, pch = 16, cex = 2) # starting point
points(res[nrow(res), 2], res[nrow(res), 3], col = 2, pch = 16, cex = 2) # stopping point
```



The guts

My `sde()` function is a brute force simulator, using the derivative function (the deterministic part) to simulate each next state from the previous state. A trick, suggested by the *sde* package, is to generate all the random numbers ahead of time. This improves the computation speed. Thus, *localsolver* includes some helper functions (`determine_ntimesteps()`, `preallocate_noise()`) which are called by `sde()` and enable the user to remain in the model space and user time units, not worrying so much about the implementation. Solving SDEs with *localsolver* will be slower than solving ODEs with *deSolve*, but hopefully it won't be more confusing.

Dynamics on networks

If we want to simulate dynamics on networks, we often have a model that looks something like this:

$\frac{dx}{dt} = -(x - r_1)(x - r_2)(x - r_3) + D \sum_{j=1}^{N} a_{i,j} x_j$

The first part of the model is some nonlinear dynamics (here, a cubic function of x) and the second is a coupling function. Here, D can be thought of a coupling strength, so the coupling function simply sums the states of all of node i's neighbors, scaled by D. To implement this in *localsolver* as an SDE, we might do something like this:

```
.doublewell <- list(
    xinit.low = 1, xinit.high = 5,
    r = c(1, 3, 5),
    D = 0.05, Ds = seq(0, 1, length.out = lout),
    u = 0, us.up = seq(0, 5, length.out = lout), us.down = seq(0, -5, length.out = lout),
    sigma = 1e-2,
    lower.basin.limit = 2, upper.basin.limit = 4
)

doublewell <- function(t, x, params) {
    with(params, {
        coupling <- D*rowSums(A*outer(rep(1, length(x)), x))
        dx <- -(x - r[1])*(x - r[2])*(x - r[3]) + coupling + u
        return(list(c(dx)))
    })
}
```

Both `.doublewell` and `doublewell()` are included in *localsolver*; `.doublewell` has more information than we need at present. We can simulate the states of the variables in a network for a given level of D as follows:

```
library(igraph)
g <- largest_component(sample_gnm(10, 20, directed = FALSE, loops = FALSE))
A <- as_adj(g, "both", sparse = FALSE)
N <- vcount(g)

params <- c(.doublewell, list(A = A))
params$sigma <- 0.1 # to make the noise more visible

x.sde <- sde(rep(.doublewell$xinit.low, N), control$times, doublewell, params, control)
```

Simulating across a control parameter range

When we need to consider the equilibrium state of a system across a range of parameter values, we need to simulate the system many times in almost the same way every time, changing only the one parameter.

localsolver provides `solve_in_range()` for this purpose.

```
library(parallel) # required for solve_in_range, may not work on Windows
ncores <- detectCores()-1

X.ode <- solve_in_range(params$Ds, "D", doublewell, rep(params$xinit.low, N), params, control, "ode")
X.sde <- solve_in_range(params$Ds, "D", doublewell, rep(params$xinit.low, N), params, control, "sde")

xlim <- range(params$Ds)
ylim <- range(c(X.ode, X.sde))

matplot(
    params$Ds, X.ode, xlim = xlim, ylim = ylim, xlab = "D", ylab = "x", lty = 1, col = 1, lwd = 1, type
)
matlines(params$Ds, X.sde, lty = 1, col = 2, lwd = 1)
```