B.10 Numerical integration of ordinary differential equations

In order to study continuous population dynamics, we often would like to integrate complex nonlinear functions of population dynamics. To do this, we need to use numerical techniques that turn the infinitely small steps of calculus, $\mathrm{d}x$, into very small, but finite steps, in order to approximate the change in y, given the change in x, or $\mathrm{d}y/\mathrm{d}x$. Mathematicians and computer scientists have devised very clever ways of doing this very accurately and precisely. In R, the best package for this is `deSolve`, which contains several *solvers* for differential equations that perform numerical integration. We will access these solvers (i.e. numerical integraters) using the `ode` function in the `deSolve` package. This function, `ode`, is a "wrapper" for the underlying suite of functions that do the work. That is, it provides a simple way to use any one of the small suite of functions.

When we have an ordinary differential equation (ODE) such as logistic growth,[3] we say that we "solve" the equation for a particular time interval given a set of parameters and initial conditions or initial population size. For instance, we say that we solve the logistic growth model for time at $t = 0, 1 \ldots 20$, with parameters $r = 1$, $\alpha = 0.001$, and $N_0 = 10$.

Let's do an example with `ode`, using logistic growth. We first have to define a function in a particular way. The arguments for the function must be time, a vector of populations, and a vector or list of model parameters.

```
> logGrowth <- function(t, y, p) {
+     N <- y[1]
+     with(as.list(p), {
+         dN.dt <- r * N * (1 - a * N)
+         return(list(dN.dt))
+     })
+ }
```

Note that I like to convert y into a readable or transparent state variable (N in this case). I also like to use `with` which allows me to use the names of my parameters [157]; this works only is `p` is a vector with named paramters (see below). Finally, we return the derivative as a list of one component.

The following is equivalent, but slightly less readable or transparent.

```
> logGrowth <- function(t, y, p) {
+     dN.dt <- p[1] * y[1] * (1 - p[2] * y[1])
+     return(list(dN.dt))
+ }
```

To solve the ODE, we will need to specify parameters, and initial conditions. Because we are using a vector of named parameters, we need to make sure we name them! We also need to supply the time steps we want.

```
> p <- c(r = 1, a = 0.001)
> y0 <- c(N = 10)
> t <- 1:20
```

[3] *e.g.* $dN/dt = rN(1 - \alpha N)$

Now you put it all into `ode`, with the correct arguments. The output is a matrix, with the first column being the time steps, and the remaining being your state variables. First we load the `deSolve` package.

```
> library(deSolve)
> out <- ode(y = y0, times = t, func = logGrowth, parms = p)
> out[1:5, ]
```

```
     time      N
[1,]    1  10.00
[2,]    2  26.72
[3,]    3  69.45
[4,]    4 168.66
[5,]    5 355.46
```

If you are going to model more than two species, y becomes a vector of length 2. Here we create a function for Lotka-Volterra competition, where

$$\frac{dN_1}{dt} = r_1 N_1 (1 - \alpha_{11} N_1 - \alpha_{12} N_2) \tag{B.5}$$

$$\frac{dN_2}{dt} = r_2 N_2 (1 - \alpha_{22} N_2 - \alpha_{21} N_1) \tag{B.6}$$

$$\tag{B.7}$$

```
> LVComp <- function(t, y, p) {
+     N <- y
+     with(as.list(p), {
+         dN1.dt <- r[1] * N[1] * (1 - a[1, 1] * N[1] -
+             a[1, 2] * N[2])
+         dN2.dt <- r[2] * N[2] * (1 - a[2, 1] * N[1] -
+             a[2, 2] * N[2])
+         return(list(c(dN1.dt, dN2.dt)))
+     })
+ }
```

Note that `LVComp` assumes that N and r are vectors, and the competition coefficients are in a matrix. For instance, the function extracts the the first element of `r` for the first species (`r[1]`); for the intraspecific competition coefficient for species 1, it uses the element of `a` that is in the first column and first row (`a[1,1]`). The vector of population sizes, N, contains one value for each population *at one time point*. Thus here, the vector contains only two elements (one for each of the two species); it holds only these values, but will do so repeatedly, at each time point. Only the output will contain all of the population sizxes through time.

To integrate these populations, we need to specify new initial conditions, and new parameters for the two-species model.

```
> a <- matrix(c(0.02, 0.01, 0.01, 0.03), nrow = 2)
> r <- c(1, 1)
> p2 <- list(r, a)
> N0 <- c(10, 10)
```

```
> t2 <- c(1, 5, 10, 20)
> out <- ode(y = N0, times = t2, func = LVComp, parms = p2)
> out[1:4, ]

     time     1     2
[1,]    1 10.00 10.00
[2,]    5 35.54 21.80
[3,]   10 39.61 20.36
[4,]   20 39.99 20.01
```

The `ode` function uses a superb ODE solver, `lsoda`, which is a very powerful, well tested tool, superior to many other such solvers. In addition, it has several bells and whistles that we will not need to take advantage of here, although I will mention one, `hmax`. This tells `lsoda` the largest step it can take. Once in a great while, with a very *stiff* ODE (a very wiggly complex dynamic), ODE assumes it can take a bigger step than it should. Setting `hmax` to a smallish number will limit the size of the step to ensure that the integration proceeds as it should.

One of the other solvers in the `deSolve`, `lsodar`, will also return roots (or equilibria), for a system of ODEs, if they exist. Here we find the roots (i.e. the solutions, or equilibria) for a two species enemy-victim model.

```
> EV <- function(t, y, p) {
+     with(as.list(p), {
+         dv.dt <- b * y[1] * (1 - 0.005 * y[1]) -
+             a * y[1] * y[2]
+         de.dt <- a * e * y[1] * y[2] - s * y[2]
+         return(list(c(dv.dt, de.dt)))
+     })
+ }
```

To use `lsodar` to find equilibria, we need to specify a root finding function whose inputs are are the sme of the ODE function, and which returns a scalar (a single number) that determines whether the rate of change ($\mathrm{d}y/\mathrm{d}x$) is sufficiently close to zero that we can say that the system has stopped changed, that is, has reached a steady state or equilibrium. Here we sum the absolute rates of change of each species, and then subtract 10^{-10}; if that difference is zero, we decide that, for all pratcial purposes, the system has stopped changing.

```
> rootfun <- function(t, y, p) {
+     dstate <- unlist(EV(t, y, p))
+     return(sum(abs(dstate)) - 1e-10)
+ }
```

Note that `unlist` changes the `list` returned by `EV` into a simple vector, which can then be summed.

Next we specify parameters, and time. Here all we want is the root, so we specify that we want the value of y after a really long time ($t = 10^{10}$). The `lsodar` function will stop sooner than that, and return the equilibrium it finds, and the time step at which it occurred.

```
> p <- c(b = 0.5, a = 0.02, e = 0.1, s = 0.2)
> t <- c(0, 1e+10)
```

Now we run the function.

```
> out <- ode(y = c(45, 200), t, EV, parms = p, rootfun = rootfun,
+     method = "lsodar")
> out[, ]

      time   1     2
[1,]   0.0  45 200.0
[2,] 500.8 100  12.5
```

Here we see that the steady state population sizes are $V = 100$ and $E = 12.5$, and that given our starting point, this steady state was achieved at $t = 500.8$. Other information is available; see `?lsodar` after loading the `deSolve package`.

B.11 Numerical Optimization

We frequently have a function or a model that we think can describe a pattern or process, but we need to "play around with" the numerical values of the constants in order to make the right shape with our function/model. That is, we need to find the value of the constant (or constants) that create the "best" representation of our data. This problem is known as *optimization*.

Optimization is an entire scientific discipline (or two). It boils down to quickly and efficiently finding parameters (i.e. constants) that meet our criteria. This is what we are doing when we "do" statistics. We fit models to data by telling the computer the structure of the model, and asking it to find values of the constants that minimize the residual error.

Once you have a model of the reality you want to describe, the basic steps toward optimization we consider are (i) create an *objective function*, (ii) use a routine to *minimize* (or *maximize*) the objective function through optimal choice of parameter values, and (iii) see if the "optimal" parameters values make sense, and perhaps refine and interpret them.

An *objective function* compares the data to the predicted values from the model, and returns a quantitative measure of their difference. One widely used objective function the *least-squares criterion*, that is, the objective function is the average or the sum of the squared deviations between the model values and the data — just like a simple ANOVA might. An optimization routine then tries to find model parameters that minimize this criterion.

Another widely used objective function is the likelihood function, or *maximum likelihood*. The likelihood function uses a probability distribution of our choice (often the normal distribution). The objective function then calculates the collective probability of observing those data, given the parameters and fit of the model. In other words, we pretend that the model and the predicted values are true, measure how far off each datum is from the predicted value, and then use a probability distribution to calculate the probability of seeing each datum. It then multiplies all those probabilities to get the *likelihood* of