Epidemics in Space

David L. Smith

May 17, 2005

Here, I'm going to introduce spatial models and show you some tricks for writing deterministic spatial models in R.

Two Patches, Two Ways

The basic ideas for spatial models can be illustrated for an epidemic in populations that occupy two patches. Obviously, understanding spatial patterns requires models that are closer to more familiar spaces, but this is particularly useful.

Movement

The first method assumes that hosts move back and forth between the two patches. The characteristic residence time in the i^{th} patch is $1/m_i$:

$$\begin{array}{rrrr} \dot{S}_1 = & -\beta_1 S_1 I_1 & & -m_1 S_1 + m_2 S_2 \\ \dot{I}_1 = & \beta_1 S_1 I_1 & -\nu I_1 & -m_1 I_1 + m_2 I_2 \\ \dot{R}_1 = & & \nu I_1 & -m_1 R_1 + m_2 R_2 \\ \dot{S}_2 = & -\beta_2 S_2 I_2 & & -m_2 S_2 + m_1 S_1 \\ \dot{I}_2 = & \beta_2 S_2 I_2 & -\nu I_2 & -m_2 I_2 + m_1 I_1 \\ \dot{R}_2 = & & \nu I_2 & -m_2 R_2 + m_1 R_1 \end{array} \tag{1}$$

We are implicitly assuming that the system is closed and that local population sizes remains constant, so $m_1 N_1 = m_2 N_2$.

Analysis of these models is similar to the SIR model in a single patch, but with a slight twist. We define a quantity called the single-visit reproductive number, the expected number of new infections generated by an infected host during a single visit to the i^{th} patch, denoted $S_{i,0}$:

$$S_{i,0} = \frac{\beta_i N_i}{\nu + m_i}.$$

If $S_{i,0} > 1$, then the parasite can invade the i^{th} patch, so $R_0 > 1$. It is possible that $S_{i,0} < 1$ in both patches, but the parasite can still invade. For example, if the residence times in each patch $1/m_i$ are much shorter than the infectious period, $1/\nu$, then a typical infected host can visit each patch more than once before clearing the infection. Conceptually, R_0 is the sum of all those visits for a typical host.

The tricky part is to define a typical host. One way to proceed is to construct the next-generation matrix, a matrix that describes the number of new infectious hosts in each patch produced by infectious hosts from each patch in each generation of the parasite [1]. Now, the generation is redefined to be the average time spent infectious in each patch, $1/(nu + m_i)$, so the faster the swtiching, the shorter the generation. The number of each hosts produced by infected hosts in their own patch is $S_{i,0}$. The number produced in the other patch

[1] Diekmann O, Heesterbeek JAP, Metz JAJ. 1990. On the definition and the computation of the basic reproduction ratio R_0 in models for infectious diseases in heterogeneous populations. J Math Biol. **28**365-382

is $m_i N_i/(nu + m_i)$. Thus, the next-generation matrix is:

$$\begin{bmatrix} \frac{\beta_1 N_1}{\nu + m_1} & \frac{m_1 N_1}{nu + m_2} \\ \frac{m_2 N_2}{nu + m_2} & \frac{\beta_2 N_2}{\nu + m_2} \end{bmatrix} \tag{2}$$

R_0 is the lead eigenvalue of this matrix.

To simulate the two-patch epidemic, consider the simple SIR epidemic on the closed population distributed in two patches:

```
> twopatch = function(t, x, pars) {
+     S = x[c(1, 2)]
+     I = x[c(3, 4)]
+     R = x[c(5, 6)]
+     with(as.list(pars), {
+         dS = -beta * I * S - m * S + m * Psi %*% S
+         dI = beta * I * S - gamma * I - m * I + m * Psi %*% I
+         dR = gamma * I - m * R + m * Psi %*% R
+         res = c(dS, dI, dR)
+         list(res)
+     })
+ }
```

This code introduces two tricks that will come in handy later. First, we pass the variables as a vector $< S_1, S_2, I_1, I_2, R_1, R_2 >$, then read them back so that S is now the vector $< S_1, S_2 >$. The ode solver computes the derivatives, but it operates on vectors, so no looping is required and the program runs faster. Second, to deal with emigration, we use matrix multiplication, with the symbol `%*%`. The matrix Ψ is defined to be:

```
> Psi = matrix(c(0, 1, 1, 0), 2, 2)
```

Now, `Psi %*% S` returns the vector $< S_2, S_1 >$, so mS_2 is added to the derivative for S_1 and mS_1 is added to the derivative for S_2. This is a trick we'll extend when we simulate epidemics on more complicated spaces.

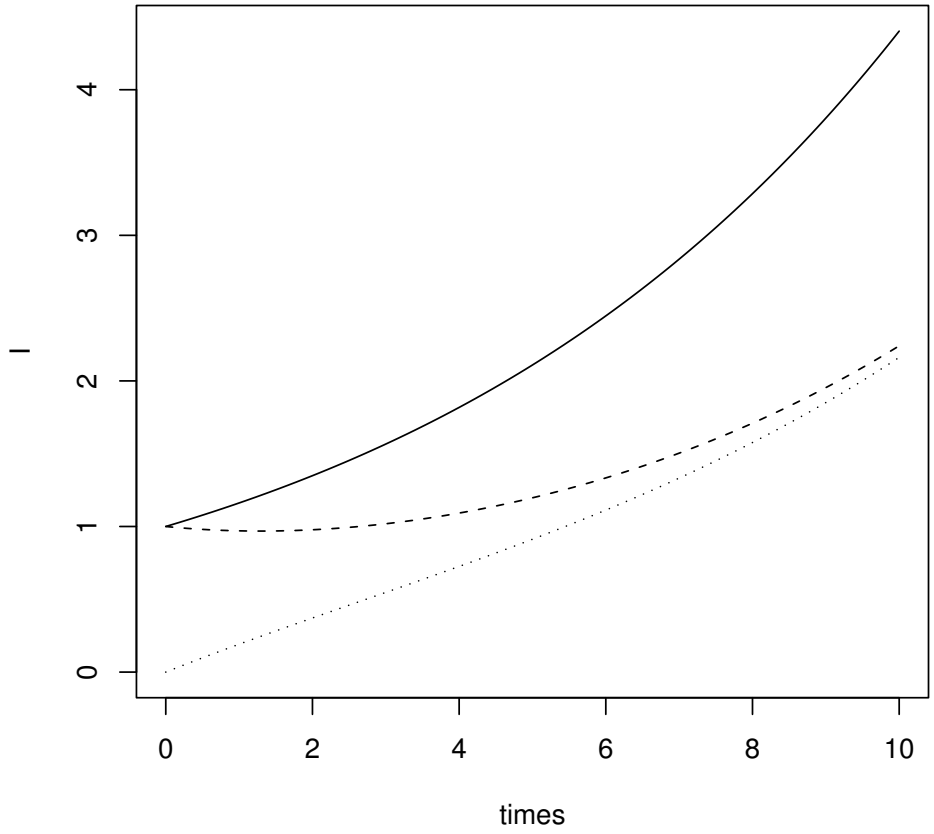
Now, we need to set up the parameter values and run the equation:

```
> require(odesolve)

Loading required package: odesolve
[1] TRUE

> times = seq(0, 10, by = 1/40)
> inits = c(S = c(100, 100), I = c(1, 0), R = c(0, 0))
> params = c(beta = 2/1000, gamma = 1/20, m = 1/5)
> odeup = lsoda(inits, times, twopatch, params)
> I.1 = odeup[, 4]
> I.2 = odeup[, 5]
```

Note that the two I variables are the 4^{th} and 5^{th} columns of `odeup` – ode puts time in the first column.

In this example, the single-visit reproductive number is 0.8, but the disease invades because of the cases generated in the other patches.

```
> I = I.1 + I.2
> plot(times, I, type = "l", ylim = c(0, max(I)))
> lines(times, I.1, lty = "dashed")
> lines(times, I.2, lty = "dotted")
```



Transmission Kernel

The second method assumes that hosts stay in a patch but infectious hosts can infect susceptible hosts in other patches. In a two-patch system we let $\beta_{i,j}$ denote the contact parameter for a susceptible host in patch i and an infectious host in patch j:

$$\begin{aligned}
\dot{S}_1 &= -(\beta_{1,1}I_1 + \beta_{1,2}I_2)\,S_1 \\
\dot{I}_1 &= (\beta_{1,1}I_1 + \beta_{1,2}I_2)\,S_1 - \nu I_1 \\
\dot{R}_1 &= \nu I_1 \\
\dot{S}_2 &= -(\beta_{2,1}I_1 + \beta_{2,2}I_2)\,S_2 \\
\dot{I}_2 &= (\beta_{2,1}I_1 + \beta_{2,2}I_2)\,S_2 - \nu I_2 \\
\dot{R}_2 &= \nu I_2
\end{aligned} \tag{3}$$

Here a transmission kernel describes contact between infectious and susceptible hosts separated by some distance. In a two-patch model, the concept of distance is fairly simple, but a kernel can be extended to any complex space defined by any distance metric.

These equations can be simulated in R; the transmission is written a matrix, rather than the migration rates.

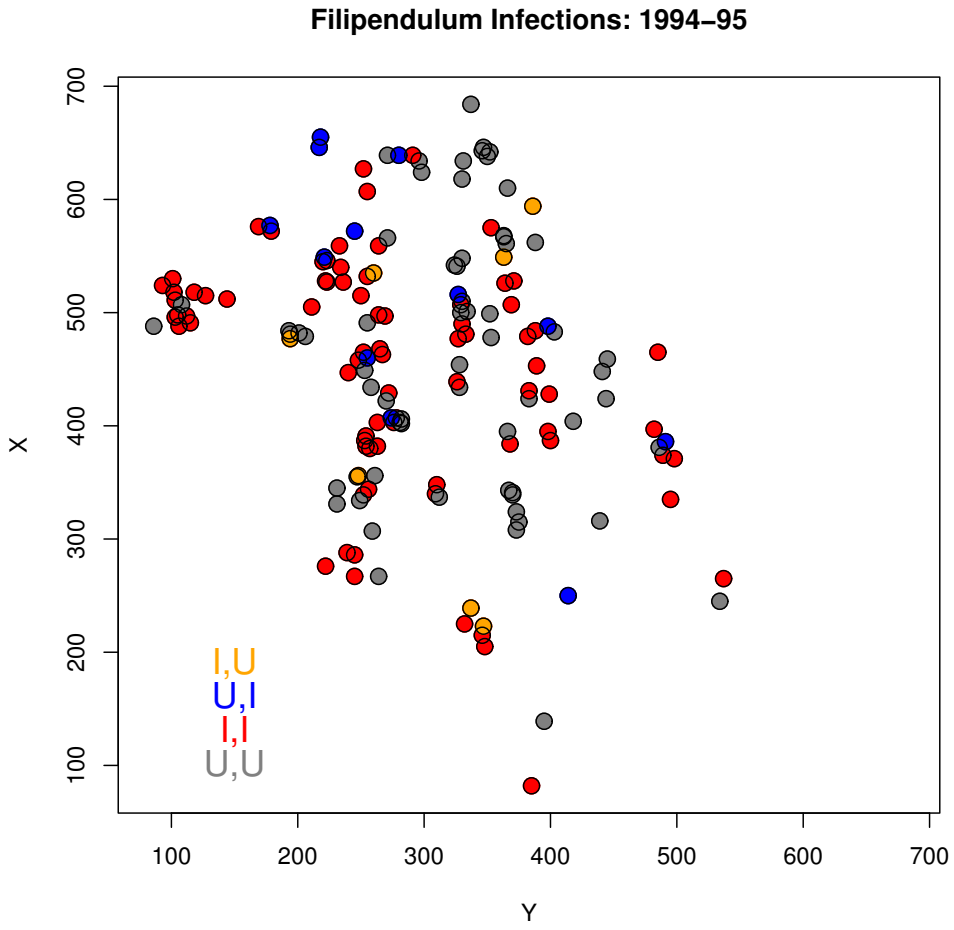


Figure 1: Populations that were infected in 1994 and 1995 are colored red (I,I). Populations that were not infected in either year are grey (U,U). Populations that became infected are colored blue (U,I).Populations that lost infections are colored orange (I,U).

Kernel Estimation Methods

To illustrate, I'll fit a kernel to one pair of years from a long-term study of a plant and its specialist fungal, foliar pathogen.

For background, the populations are found on the shoreline of an island archipelago in Sweden. Every year, the harsh winters kill the above-ground plant tissue. The fungus over-winters in the dead plant material, then re-infects the new plant tissue as it grows up through the debris. Infected plant debris can also be blown around, or wash up onto populations as flotsam. Finally, in living tissue, the fungus produces spores that are wind-dispersed to other plants.

Thus, every spring, every plant must be re-infected from one of last year's infected plants. A plant is considered to be either infected or not, so infection can be modelled as a Bernouli random variable.

Here, we fit a very simple kernel, $\phi e^{-\lambda d_{i,j}}$. First, we read in the data and create a distance matrix:

```
> y94 = read.table("y94.txt", header = T)
> dst = as.matrix(dist(y94[, c(3, 4)]))
> N = dim(y94)[1]
```

Here's the function that computes the log-likelihood, for particular values of ϕ and λ:

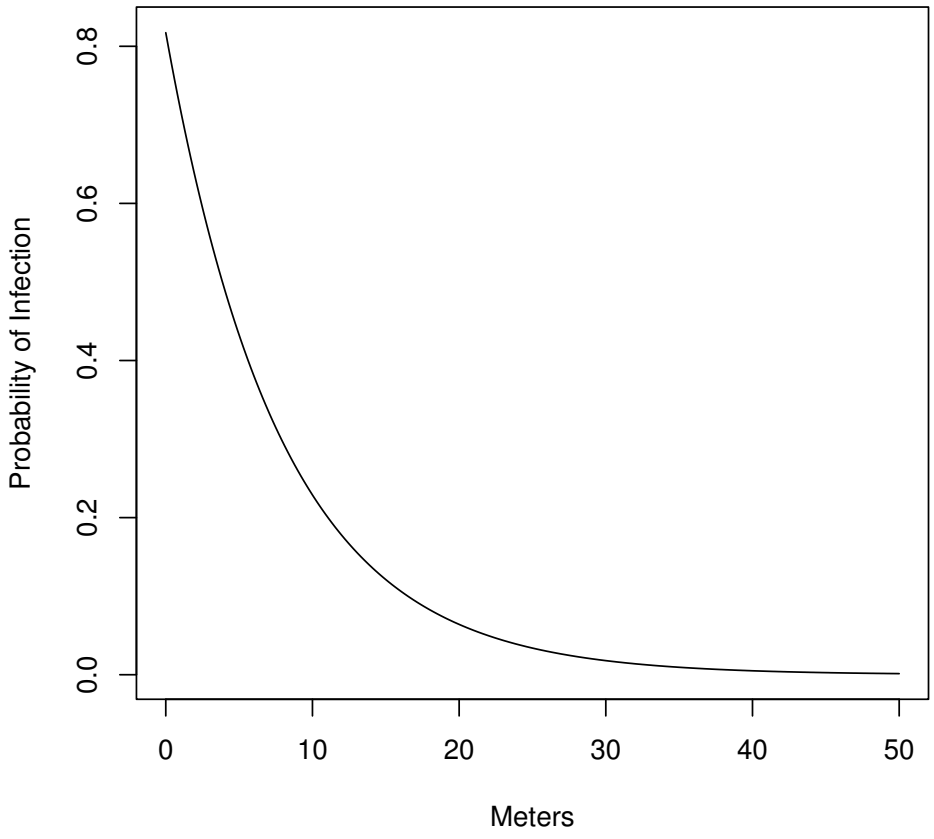
```
> kernel.loglik = function(x) {
+     lambda = x[1]
+     phi = x[2]
+     P = rep(0, N)
+     idx = which(y94$now == 1)
+     for (i in 1:N) {
+         P[i] = 1 - prod(1 - phi * exp(-lambda * dst[i, idx]))
+     }
+     loglik = log(P * y94$nxt + (1 - P) * (1 - y94$nxt))
+     -sum(loglik)
+ }
```

The minimization is a simple call to the `nlm` package, for non-linear minimization. Since we are miminizing, the function returns the negative log-likelihood.

```
> A = nlm(kernel.loglik, p = c(0.1, 0.5))
```

Finally, `A$est` will give us the estimated parameters:

```
> lambda = A$est[1]
> phi = A$est[2]
> distance = seq(0, 50, by = 0.1)
> plot(distance, phi * exp(-lambda * distance), type = "l", xlab = "Meters",
+     ylab = "Probability of Infection")
```



More Complicated Spaces

We want to develop some tools for dealing with more complicated spaces. Sometimes, the space is naturally defined. At other times, we make continuous space more computationally and operationally managable by chopping it up into a set of patches. Complicated spaces can be dealt with using the same tricks as the two-patch–we simply need an easy way to index patches and write the adjacency matrix Ψ. Conceptually, Ψ is a matrix where the i, j^{th} entry describes the weight of the connection between the i^{th} and j^{th} patches.

Here, I'll develop some methods for defining the nearest-neighbor matrix on four common spaces: arrays, rectangular grids, hexagonal grids, and arbitrary sets of points.

Arrays

The adjacency matrix for an array is relatively simple. The patches are indexed in the natural way, so that the i^{th} patches neighbors are indexed $i \pm 1$. The adjacency matrix has positive entries on the off-diagonals. For example, the adjacency matrix for an array of 5 patches has the form:

$$\begin{bmatrix} 0 & 1 & 0 & 0 & 0 \\ 1 & 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 & 1 \\ 0 & 0 & 0 & 1 & 0 \end{bmatrix} \tag{4}$$

One small detail was swept under the rug. We assumed that the patches at the edge of the array did not have any neighbors. There are several ways to deal with edges. If the linear space is the shoreline of an island, the patches with the highest and lowest index are each other's neighbors, so the adjacency matrix is the following:

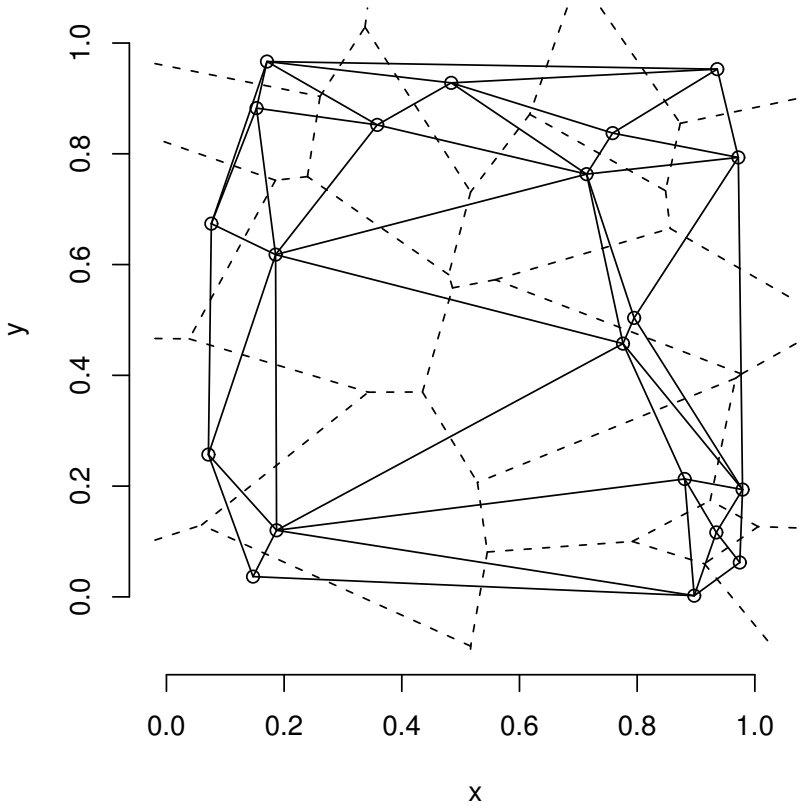
$$\begin{bmatrix} 0 & 1 & 0 & 0 & 1 \\ 1 & 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 & 1 \\ 1 & 0 & 0 & 1 & 0 \end{bmatrix} \tag{5}$$

What happens at the edges? Do the emigrants stay put, as if there were a barrier there? Is the rate of emigration at the edge lower? These are questions to be considered carefully in real problems.

0.1 Tesselations of the plane

A tesselation is simply a subdivision of the plane into small regions, called tiles. To deal with the other three examples, we'll use an R package called `deldir` that does the Delaunay triangulation and the Dirichlet tesselation. We'll illustrate the algorithm for a random set of points:

```
> L = 20
> x = runif(L, 0, 1)
> y = runif(L, 0, 1)
> require(deldir)

Loading required package: deldir
deldir
version: 0.0-2
[1] TRUE

> dd = deldir(x, y)
> plot.deldir(dd)
```



For 20 random points in the unit square. The algorithm subdivides space into a set of tiles such that every point inside the tile is closest to one of the finite set of specified points. The dotted lines show the boundaries of the tiles. The solid lines show the nearest neighbors.

The last two columns of `dd$delsgs` are the indices of the adjacent pairs, so we can use these to construct the adjacency matrix:

```
> Psi = matrix(0, L, L)
> i = dd$delsgs[, 5]
> j = dd$delsgs[, 6]
> Psi[i, j] = 1
> Psi[j, i] = 1
```

Note that each pair is ordered and listed once, so we have to worry about $\Psi[i, j]$ and $\Psi[j, i]$. To create the adjacency matrix for a grid, we need only specify the distribution of points and `deldir` will do the work for us. The specify the squares on a chessboard, we use the very useful function:

```
> xy = expand.grid(1:8, 1:8)
```

A hexagonal grid is only slightly more difficult, but we only need supply the centers. The following set of points does the trick:

```
> x1 = c(1:L)
> y1 = c(1:L) * sqrt(3)
> x2 = x1 + 1/2
> y2 = y1 + sqrt(3)/2
```

```
> xy1 = expand.grid(x1, y1)
> xy2 = expand.grid(x2, y2)
> xy = rbind(xy1, xy2)
```

Travelling Waves

Having an algorithm in hand, we can now simulate an epidemic on an array, and visualize a travelling wave on an array:

```
> spatial.sir = function(t, x, pars) {
+     idx = c(1:L)
+     S = x[idx]
+     I = x[L + idx]
+     R = x[2 * L + idx]
+     with(as.list(pars), {
+         dS = -beta * I * S - m * mig * S + m * Psi %*% S
+         dI = beta * I * S - gamma * I - m * mig * I + m * Psi %*%
+             I
+         dR = gamma * I - m * mig * R + m * Psi %*% R
+         res = c(dS, dI, dR)
+         list(res)
+     })
+ }
> L = 100
> Psi = matrix(0, L, L)
> i = c(2:L)
> j = i - 1
> idx = cbind(c(i, j), c(j, i))
> Psi[idx] = 1
> ones = rep(1, L)
> mig = Psi %*% ones
> Pop = 100
> params = c(gamma = 0.2, beta = 3/Pop * 0.2, m = 0.003, mig = mig,
+     Psi = Psi)
> S.init = rep(Pop, L)
> I.init = rep(0, L)
> R.init = rep(0, L)
> S.init[1] = Pop - 1
> I.init[1] = 1
> inits = c(S.init, I.init, R.init)
> times = seq(0, 1000, by = 1)
> lng = length(times)
> odeup = lsoda(inits, times, spatial.sir, params)
> plotit = function(i, odeup) {
+     idx = c(1:L)
+     S = odeup[i, idx + 1]
+     I = odeup[i, idx + L + 1]
+     R = odeup[i, idx + 2 * L + 1]
+     plot(idx, S, type = "l", col = "blue", ylim = c(0, Pop))
+     lines(idx, I, col = "red")
+     lines(idx, R, col = grey(0.5))
+ }
> movie = function() {
+     for (i in 1:lng) {
```

```
+         plotit(i, odeup)
+         Sys.sleep(0.01)
+     }
+ }
> plotit(round(lng/2), odeup)
```