Simulating spatial functions with h() functions in lookup table form

Stephen Eglen

March 17, 2010

Abstract

This document describes functions in the R package *sjedmin* for simulating univariate and bivariate spatial patterns using h() functions that have been specified as look up tables (LUTs). Specifying h() as a LUT rather than in a functional form has the advantage that it is easy to write the h() function in R and then do the simulation of patterns in C. It also allows you to use the non-parametric estimators of h() from *spatstat* package directly. (This document is created from the Rnw source file, hlookup.Rnw, which is found in the extras subdirectory of the sjedmin package.)

1 The h function

The h function can be used in a pairwise interaction point process (PIPP) model to specify interactions between pairs of cells. The h function can be given either as a function, with parameters controlling its shape, or as a lookup table (LUT). The advantage of a LUT is that gives us a lot of freedom in deciding what kinds of h() functions we want to use when simulating spatial point patterns.

By a LUT I simply mean a table where for certain distances, d, we provide the corresponding value of $h(d)$, e.g. here is a LUT for a simple step function around the point $d = 20$:

```
> h <- function(d) {
+     ifelse(d > 20, 1, 0)
+ }
> d <- seq(from = 6, to = 30, by = 2)
> h1 <- h(d)
> print(lut1 <- rbind(d, h1))
```

```
   [,1] [,2] [,3] [,4] [,5] [,6] [,7] [,8] [,9] [,10] [,11] [,12] [,13]
d     6    8   10   12   14   16   18   20   22    24    26    28    30
h1    0    0    0    0    0    0    0    0    1     1     1     1     1
```

So, in this LUT, if we want to find the value of h() when d=24, we just lookup the corresponding value in the LUT. In my code, linear interpolation between entries is used, which can be a little confusing, especially in the case of a step function:

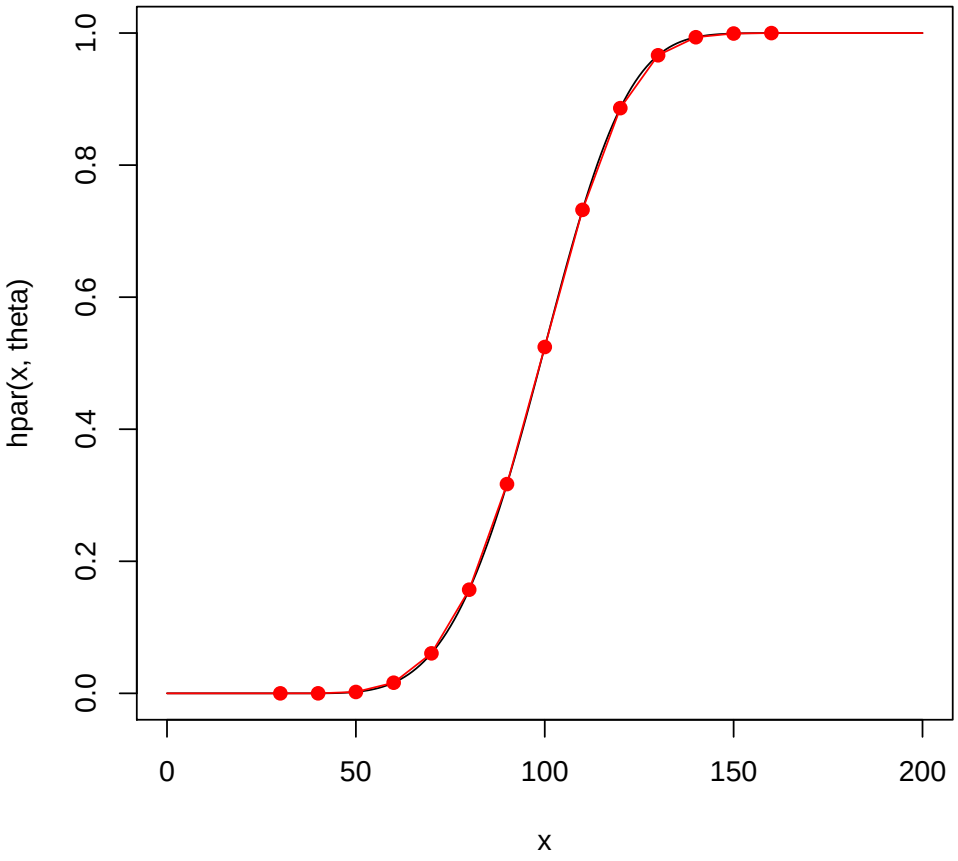
```
> hlookup(h1, d, 21)
```

```
[1] 0.5
```

Here it thinks $h(21) = 0.5$, since it has interpolated linearly between the LUT entries at 20 and 22. (In this case, to mimic a step function, it would be better to use d values at 19.99 and 20.01).

Here is a more natural example, using the h() parameterisation suggested by Peter recently.

```
> hpar <- function(d, theta) {
+     delta <- theta[1]
+     sigma <- theta[2]
+     kappa <- theta[3]
+     res <- (0 * (d < delta)) + (d >= delta) * (1 - exp(-((d -
+         delta)/sigma)^kappa))
+     if (any(is.nan(res)))
+         res[which(is.nan(res))] <- 0
+     res
+ }
> theta <- c(35, 70, 4)
> x <- seq(from = 0, to = 200, by = 1)
> plot(x, hpar(x, theta), type = "l")
> x.lut <- seq(from = 30, to = 160, by = 10)
> y.lut <- hpar(x.lut, theta)
> points(x.lut, y.lut, pch = 19, col = "red")
> lines(x, sapply(x, function(d) {
+     hlookup(y.lut, x.lut, d)
+ }), col = "red")
```



Here, the black curve is the true h() function from the parametric h function; the red circles are the entries from the LUT (x.lut, y.lut). The hlookup() routine linearly interpolates between these LUT entries to produce a close approximation to the real function. (So close that hopefully the underlying black curve is mostly covered by the red line.)

The hlookup() function makes several assumptions about the LUT:

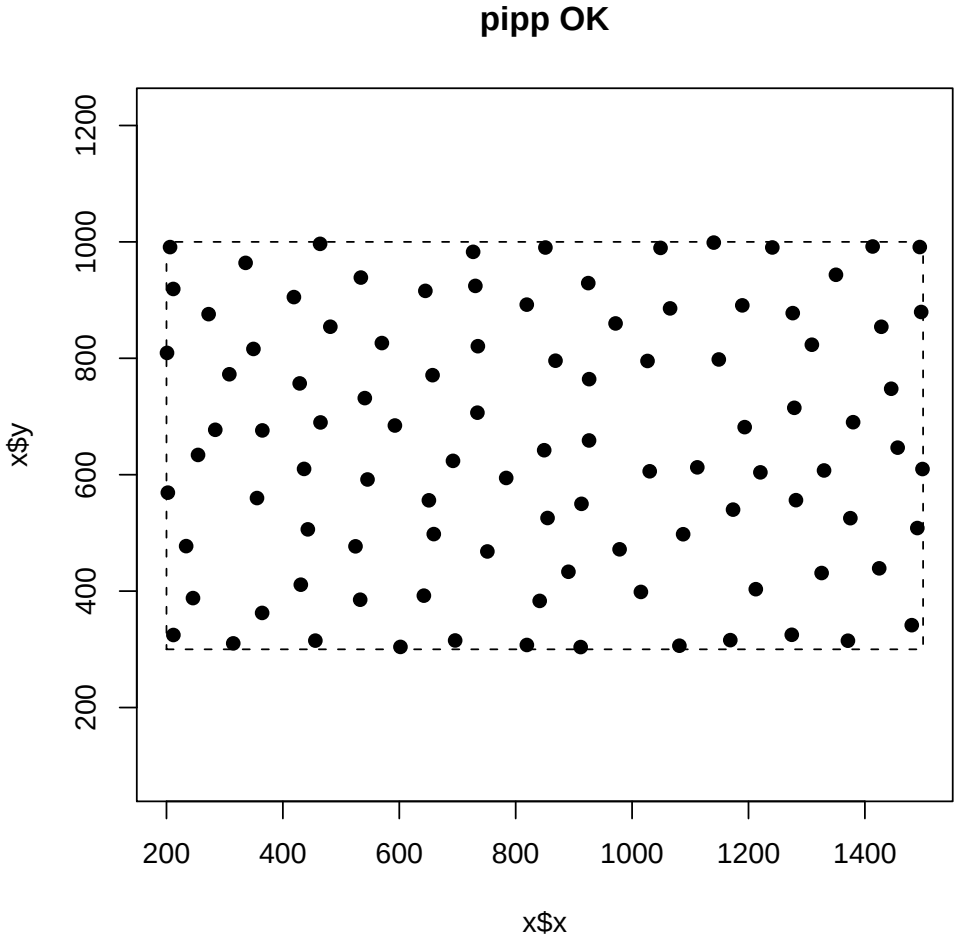
1. The d entries (distance) are ordered, smallest first.
2. Below the first entry in the LUT, it is assumed that h(d)=0.
3. After the last entry in the LUT, it is assumed that h(d)=1.

These last two points can be seen in the example above, since x.lut is only in the range [30,160]. Note also that the d values do not need to be equally-spaced along the x axis, they simply need to be monotically increasing.

2 Simulating univariate spatial patterns

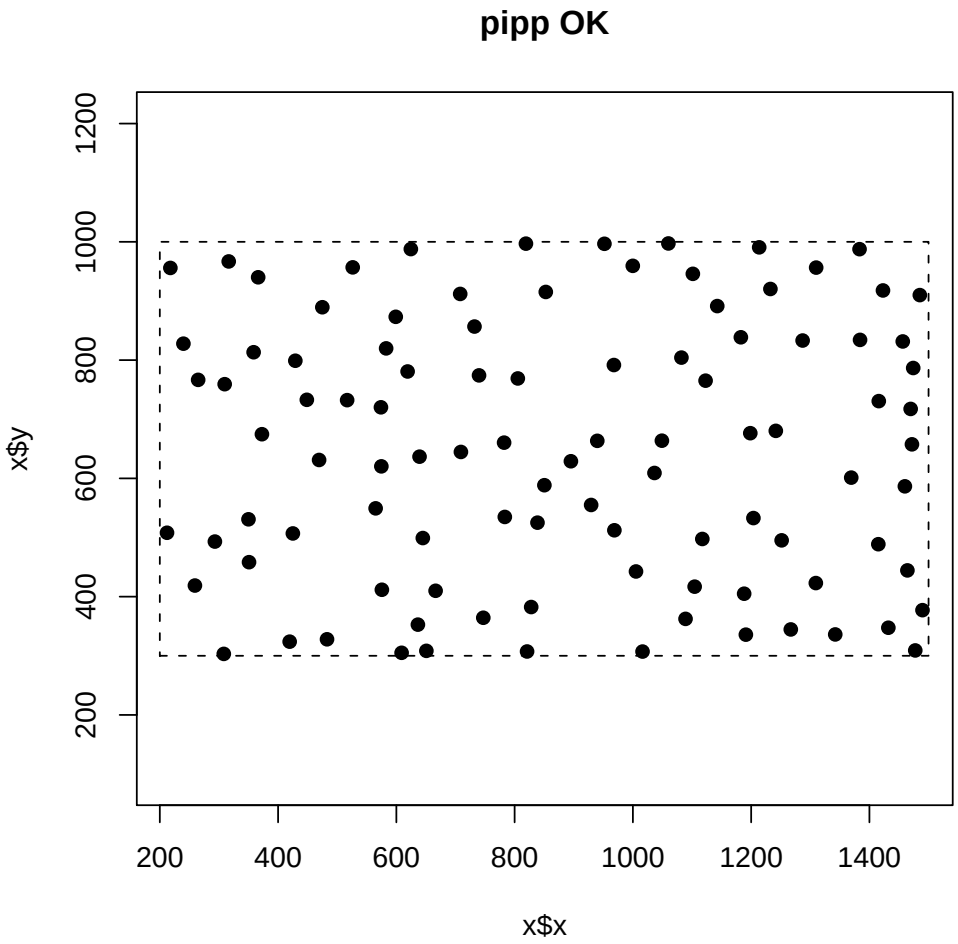
The routine pipp.lookup() will take an h() function in the form of a LUT and simulate a pattern according to that h function. Here is an example, using the previous LUT:

```
> w <- c(200, 1500, 300, 1000)
> npts <- 100
> p <- pipp.lookup(w = w, n1 = npts, pts = NULL, h = y.lut, d = x.lut,
+     nsweeps = 10, verbose = F)

> plot(p)
```



Compare this with a plot where the interactions are over smaller distances:

```
> p <- pipp.lookup(w = w, n1 = npts, pts = NULL, h = hpar(x.lut,
+     theta = c(35, 50, 1)), d = x.lut, nsweeps = 10, verbose = F)
> plot(p)
```



The arguments to pipp.lookup are:

w Sampling window, given as (xlo, xhi, ylo, yhi).

n1 Number of points to simulate.

pts Initial set of points to start moving, if not NULL. If given, the points must be given as an array of size (n1,2).

h The h values (y-axis values) for the LUT.

d The d values (x-axis values) for the LUT.

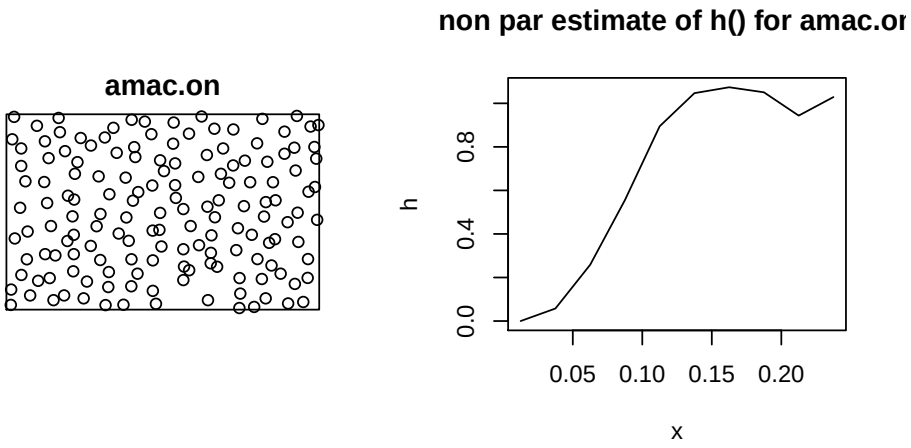
nsweeps The number of sweeps (where one sweep consists of moving all cells, in turn, once).

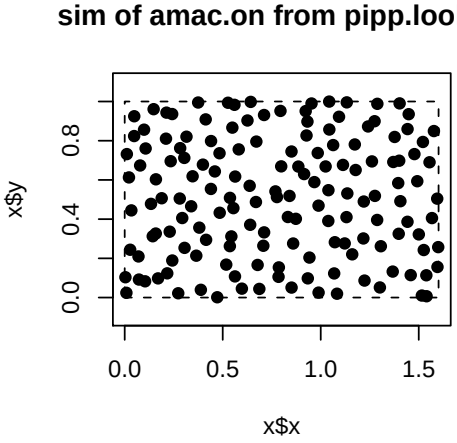
verbose A debugging argument.

2.1 Non-parametric estimates of h()

Note that we can also use the spatstat library to compute a non parametric estimate of h() and then directly simulate from that estimate. Here we use the spatstat library to compute an estimate of h() for the on amacrine cells, and then simulate them.

```
> library(spatstat)

This is mgcv 1.6-1. For overview type `help("mgcv-package")'.
deldir 0.0-12

     Please note: The process for determining duplicated points
     has changed from that used in version 0.0-9 (and previously).

spatstat 1.18-0
Type âĂŸhelp(spatstat)âĂŹ for an overview of spatstat
     âĂŸlatest.news()âĂŹ for news on latest version
     âĂŸlicence.polygons()âĂŹ for licence information on polygon calculations

> data(amacrine)
> amac.on <- unmark(amacrine[amacrine$marks == "on"])
> rs <- seq(from = 0.025, by = 0.025, length = 10)
> qs <- quadscheme(amac.on, method = "grid", ntile = c(40, 30))
> x <- ppm(qs, ~1, PairPiece(r = rs), correction = "isotropic")
> h <- summary(x)$interaction$printable
> if (any(is.na(h))) h[which(is.na(h))] <- 0
> x <- apply(rbind(rs, c(0, rs[1:(length(rs) - 1)])), 2, mean)
> p <- pipp.lookup(n1 = amac.on$n, pts = NULL, w = c(amac.on$window[2]$xrange,
+     amac.on$window[3]$yrange), h = h, d = x, nsweeps = 10, verbose = F)

> par(mfrow = c(2, 2))
> plot(amac.on)
> plot(x, h, main = "non par estimate of h() for amac.on", type = "l")
> plot(p, main = "sim of amac.on from pipp.lookup")
```



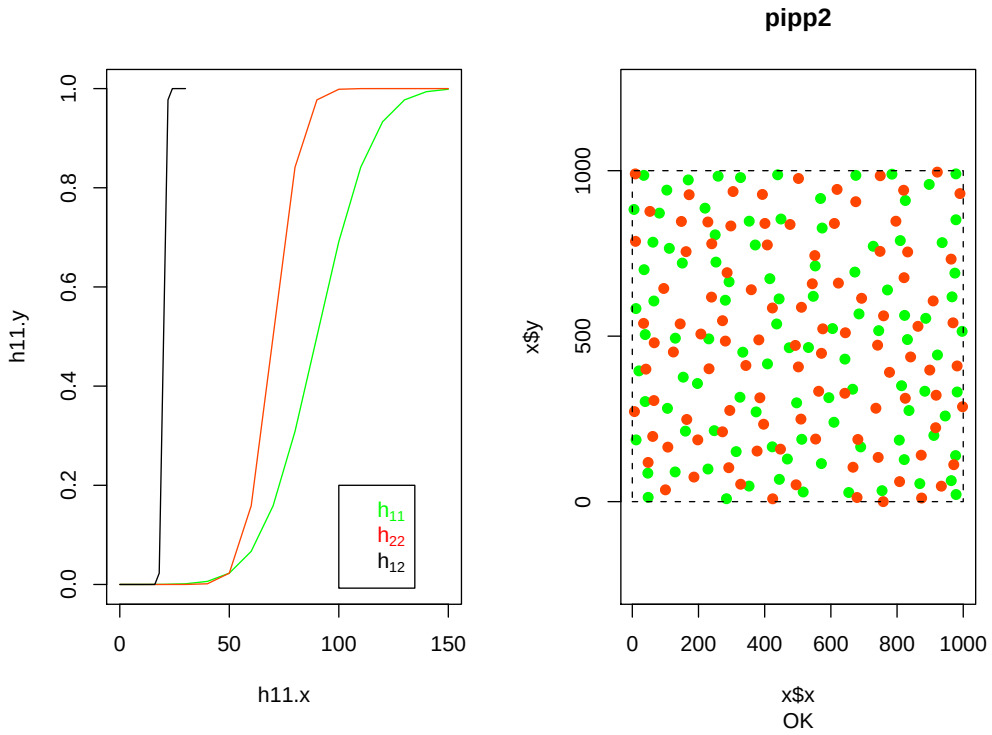


3 Simulating bivariate spatial patterns

A second routine, pipp2.lookup, allows us to provide three LUTs for simulating bivariate patterns: h_{11} for interactions between type 1 cells, h_{22} for type 2 cells and h_{12} for interactions between the two types of cells. The arguments to pipp2.lookup are similar to those for pipp.lookupm, as the following example demonstrates:

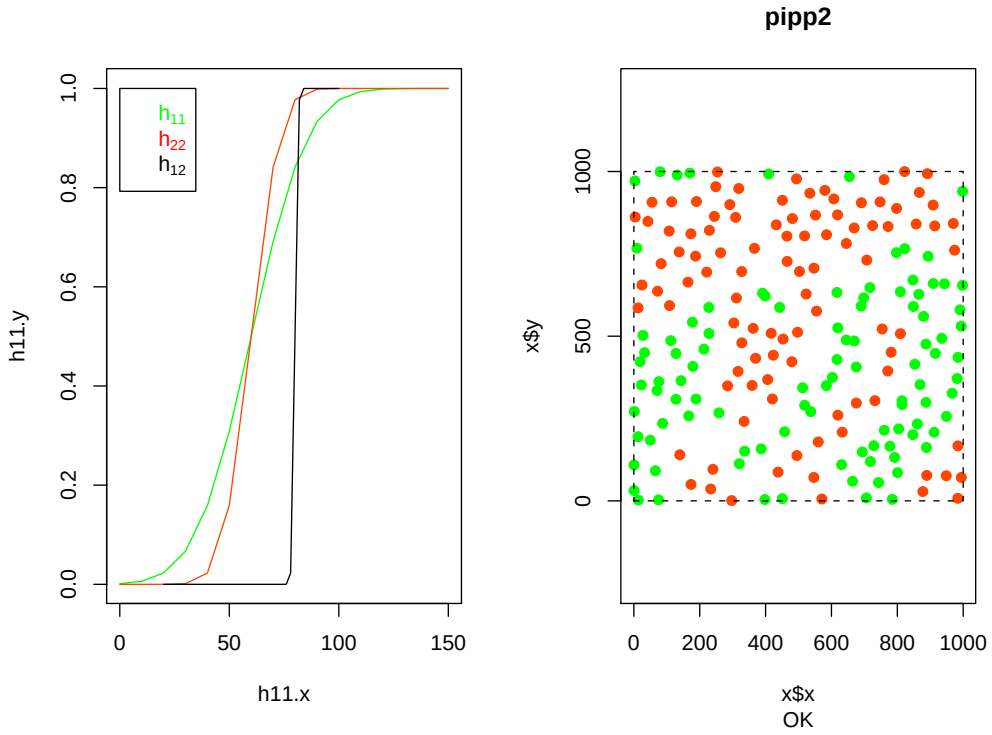
```
> w <- c(0, 1000, 0, 1000)
> n1 <- 100
> n2 <- 100
> h11.x <- seq(from = 0, to = 150, by = 10)
> h11.y <- pnorm(h11.x, mean = 90, sd = 20)
> h22.x <- seq(from = 0, to = 150, by = 10)
> h22.y <- pnorm(h11.x, mean = 70, sd = 10)
> h12.x <- seq(from = 0, to = 30, by = 2)
> h12.y <- pnorm(h12.x, mean = 20, sd = 1)
> par(mfrow = c(1, 2))
> plot(h11.x, h11.y, type = "l", col = "green")
> lines(h22.x, h22.y, col = "orangered")
> lines(h12.x, h12.y, col = "black")
> legend(100, 0.2, c(expression(h[11]), expression(h[22]), expression(h[12])),
+     text.col = c("green", "red", "black"))
```

```
> p <- pipp2.lookup(w = w, pts1 = NULL, pts2 = NULL, n1 = n1, n2 = n2,
+     h1 = h11.y, d1 = h11.x, h2 = h22.y, d2 = h22.x, h12 = h12.y,
+     d12 = h12.x, nsweeps = 10, verbose = FALSE)
> plot(p)
```



As a curious example, if we make the inhibition distance for h_{12} much larger than the homotypic h() functions, we get clustering of cells:

```
> par(mfrow = c(1, 2))
> h11.x <- seq(from = 0, to = 150, by = 10)
> h11.y <- pnorm(h11.x, mean = 60, sd = 20)
> h22.x <- seq(from = 0, to = 150, by = 10)
> h22.y <- pnorm(h11.x, mean = 60, sd = 10)
> h12.x <- seq(from = 20, to = 100, by = 2)
> h12.y <- pnorm(h12.x, mean = 80, sd = 1)
> plot(h11.x, h11.y, type = "l", col = "green")
> lines(h22.x, h22.y, col = "orangered")
> lines(h12.x, h12.y, col = "black")
> legend(0, 1, c(expression(h[11]), expression(h[22]), expression(h[12])),
+     text.col = c("green", "red", "black"))
> p <- pipp2.lookup(w = w, pts1 = NULL, pts2 = NULL, n1 = n1, n2 = n2,
+     h1 = h11.y, d1 = h11.x, h2 = h22.y, d2 = h22.x, h12 = h12.y,
+     d12 = h12.x, nsweeps = 10, verbose = FALSE)
> plot(p)
```



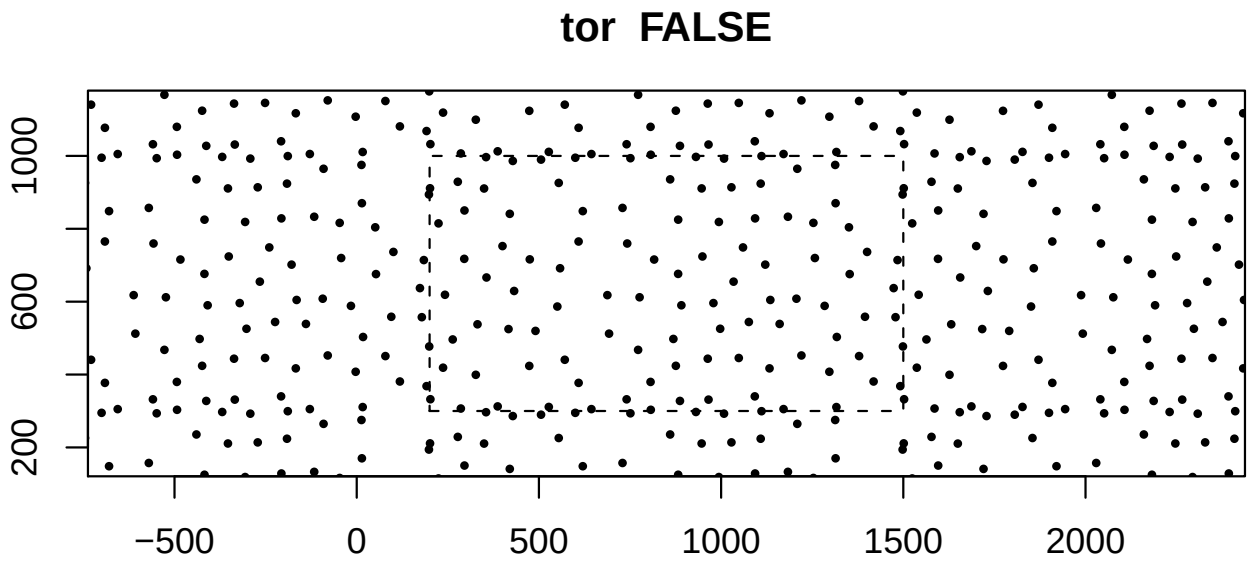
4 Toroidal conditions for hlookup

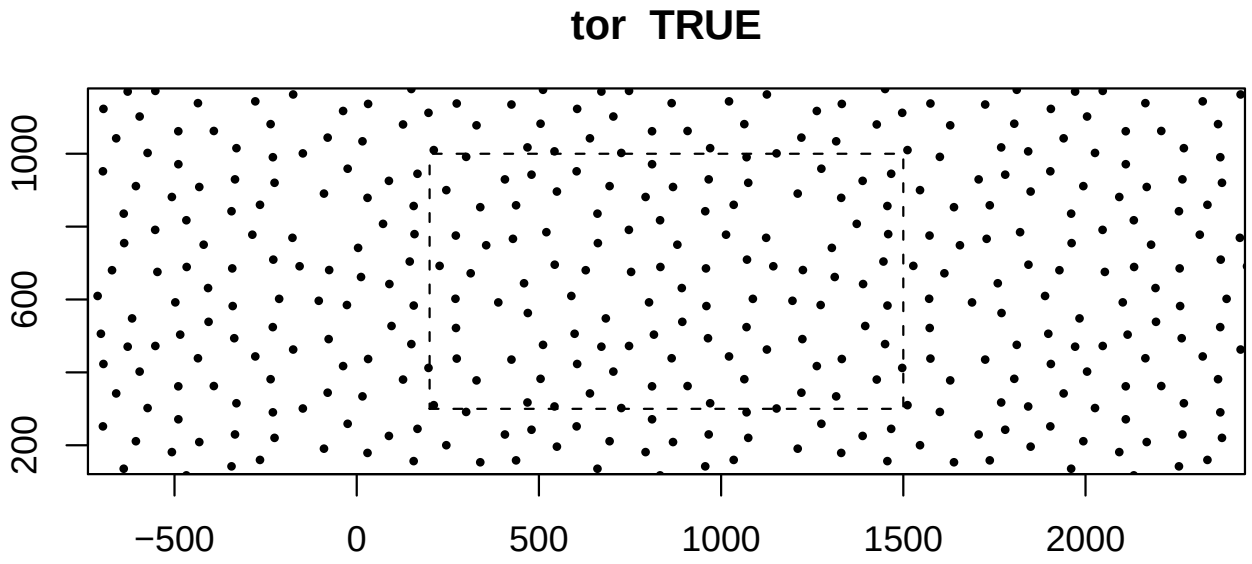
The univariate function hlookup includes a flag for toroidal boundary conditions. Here is an example, both without and with toroidal conditions. To check that the toroidal conditions are working, in the plot below the original data set is inside the dashed rectangle; copies of the dataset are wrapped around outside, just to check.

In this example, I use a simple inhibition for h(), such that all points closer than 70 units are rejected.

```
> w <- c(200, 1500, 300, 1000)
> npts <- 100
> x.lut <- seq(from = 30, to = 160, by = 10)
> y.lut <- ifelse(x.lut > 70, 1, 0)
> par(mfrow = c(2, 1), mar = c(3, 2, 3, 1))
> for (tor in c(FALSE, TRUE)) {
+     p <- pipp.lookup(w = w, n1 = npts, pts = NULL, h = y.lut,
+         d = x.lut, nsweeps = 10, verbose = F, tor = tor)
+     x0 <- p$x
+     y0 <- p$y
+     wid <- w[2] - w[1]
+     ht <- w[4] - w[3]
+     all <- rbind(cbind(x0 - wid, y0 + ht), cbind(x0, y0 + ht),
+         cbind(x0 + wid, y0 + ht), cbind(x0 - wid, y0), cbind(x0,
+             y0), cbind(x0 + wid, y0), cbind(x0 - wid, y0 - ht),
+         cbind(x0, y0 - ht), cbind(x0 + wid, y0 - ht))
+     dw <- wid * 0.2
+     dh <- ht * 0.2
+     plot(all, xlim = c(w[1] - dw, w[2] + dw), ylim = c(w[3] -
+         dh, w[4] + dh), pch = 19, cex = 0.4, asp = 1)
```

```
+     title(main = paste("tor ", tor))
+     rect(w[1], w[3], w[2], w[4], lty = 2)
+ }
> par(mfrow = c(1, 1))
```





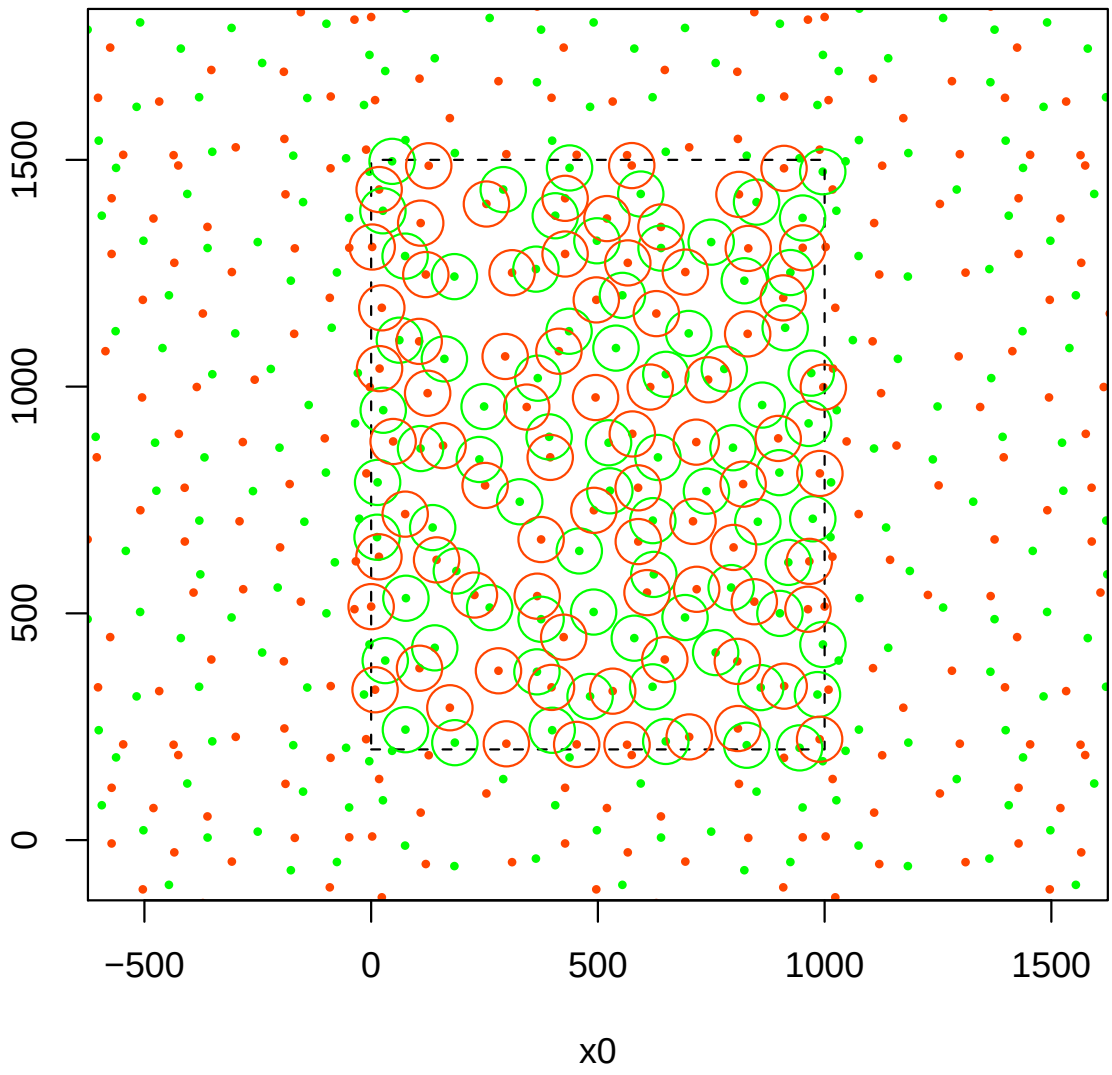
In the following, here are bivariate simulations, in each case the homotypic constraint is a hard exclusion (at 100 um), and there is a heterotypic constrain set around 60 ± 2 um. When tor is FALSE, you should be able to see border cells that have cells outside the rectangle breaking the constraint. This does not happen when tor is TRUE.

```
> r1 <- r2 <- 100
> example.tor2 <- function(tor) {
+     w <- c(0, 1000, 200, 1500)
+     n1 <- n2 <- 75
+     h11.x <- seq(from = 0, to = 150, by = 10)
+     h11.y <- ifelse(h11.x > r1, 1, 0)
+     h22.x <- seq(from = 0, to = 150, by = 10)
```

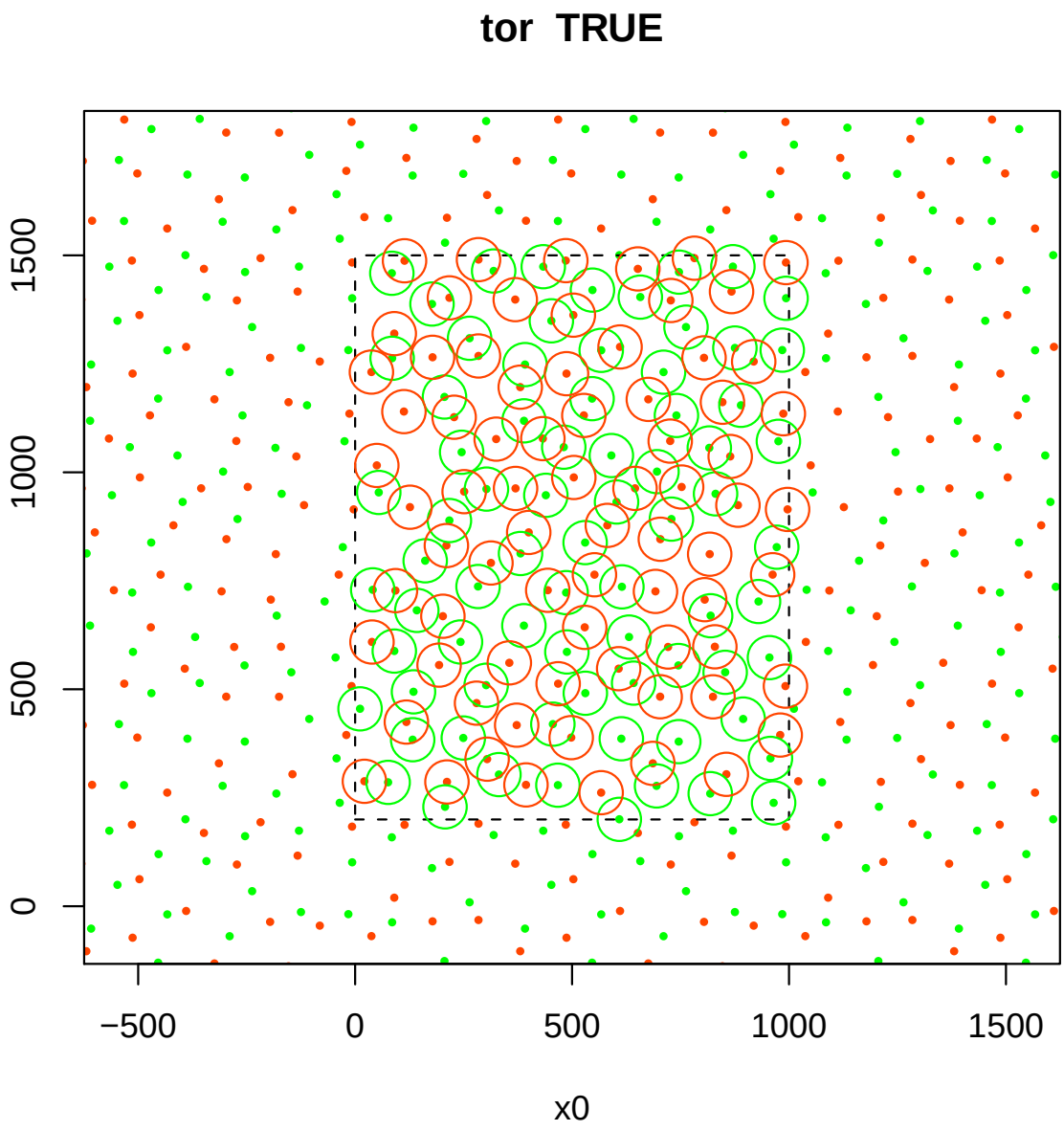
```
+     h22.y <- ifelse(h22.x > r2, 1, 0)
+     h12.x <- seq(from = 0, to = 60, by = 2)
+     h12.y <- pnorm(h12.x, mean = 40, sd = 1)
+     p <- pipp2.lookup(w = w, pts1 = NULL, pts2 = NULL, n1 = n1,
+         n2 = n2, h1 = h11.y, d1 = h11.x, h2 = h22.y, d2 = h22.x,
+         h12 = h12.y, d12 = h12.x, nsweeps = 10, verbose = FALSE,
+         tor = tor)
+     x0 <- p$x
+     y0 <- p$y
+     wid <- w[2] - w[1]
+     ht <- w[4] - w[3]
+     all <- rbind(cbind(x0 - wid, y0 + ht), cbind(x0, y0 + ht),
+         cbind(x0 + wid, y0 + ht), cbind(x0 - wid, y0), cbind(x0,
+             y0), cbind(x0 + wid, y0), cbind(x0 - wid, y0 - ht),
+         cbind(x0, y0 - ht), cbind(x0 + wid, y0 - ht))
+     dw <- wid * 0.2
+     dh <- ht * 0.2
+     cols <- c(rep("green", p$n1), rep("orangered", p$n2))
+     plot(all, xlim = c(w[1] - dw, w[2] + dw), ylim = c(w[3] -
+         dh, w[4] + dh), asp = 1, col = cols, pch = 19, cex = 0.4)
+     title(main = paste("tor ", tor))
+     rect(w[1], w[3], w[2], w[4], lty = 2)
+     symbols(x0, y0, circles = rep(r1/2, n1 + n2), inch = FALSE,
+         add = TRUE, fg = cols)
+ }
```

The next two plots show bivariate PIPP without and with toroidal conditions, using the same h functions as before.

```
> example.tor2(tor = FALSE)
```




```
> example.tor2(tor = TRUE)
```


tor TRUE
1500
1000
500
0
−500
0
500
1000
1500
x0