Spatial Analysis Techniques in R

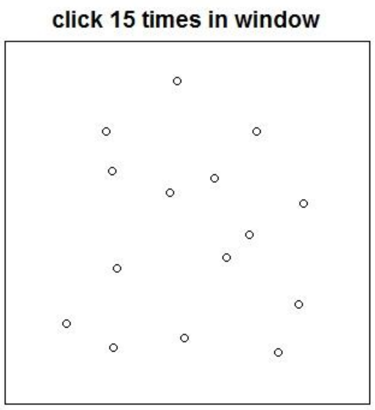
Mid-week bit of fun (2)

This week we look at some facilities to create and record point patterns. Both the methods allow us to interact with a point pattern in some way.

a) Creating your own point pattern in spatstat

How 'random' are you? Use clickppp in spatstat to create a point pattern and then check on it. In the example we have just 15 events in a unit square:

```
> my_dat<-clickppp(n=15)
```



```
> clarkevans(my_dat)
   naive Donnelly      cdf
1.141838 1.012138 1.095965
#my-dat is a planar point pattern …
> summary(my_dat)
Planar point pattern: 15 points
Average intensity 15 points per square unit
Window: rectangle = [0, 1]x[0, 1]units
Window area =  1 square unit
```

I usually score somewhere around 1.1 on the Clark Evans statistic but can you think of more serious uses for this method?

b) Digitizing data from old plots using 'digitize'

digitize is a handy package if you occasionally find yourself needing to retrieve locational data from printed maps or other graphics. For reasons I do not understand it isn't currently on the CRAN mirrors and has to be extracted from an archive and recompiled for use using devtools. So we fire up R, click on the ‘Packages’ tab, go our our favourite CRAN mirror and install devtools:

```
> utils:::menuInstallPkgs()
--- Please select a CRAN mirror for use in this session ---
trying URL 'http://www.stats.bris.ac.uk/R/bin/windows/contrib/3.1/devtools_1.6.1.zip'
Content type 'application/zip' length 284735 bytes (278 Kb)
opened URL
downloaded 278 Kb
package ‘devtools’ successfully unpacked and MD5 sums checked
The downloaded binary packages are in
        C:\Users\David\AppData\Local\Temp\Rtmp86MJmz\downloaded_packages
Next:

> require("devtools")
Loading required package: devtools
```

So we can now access the digitize package:

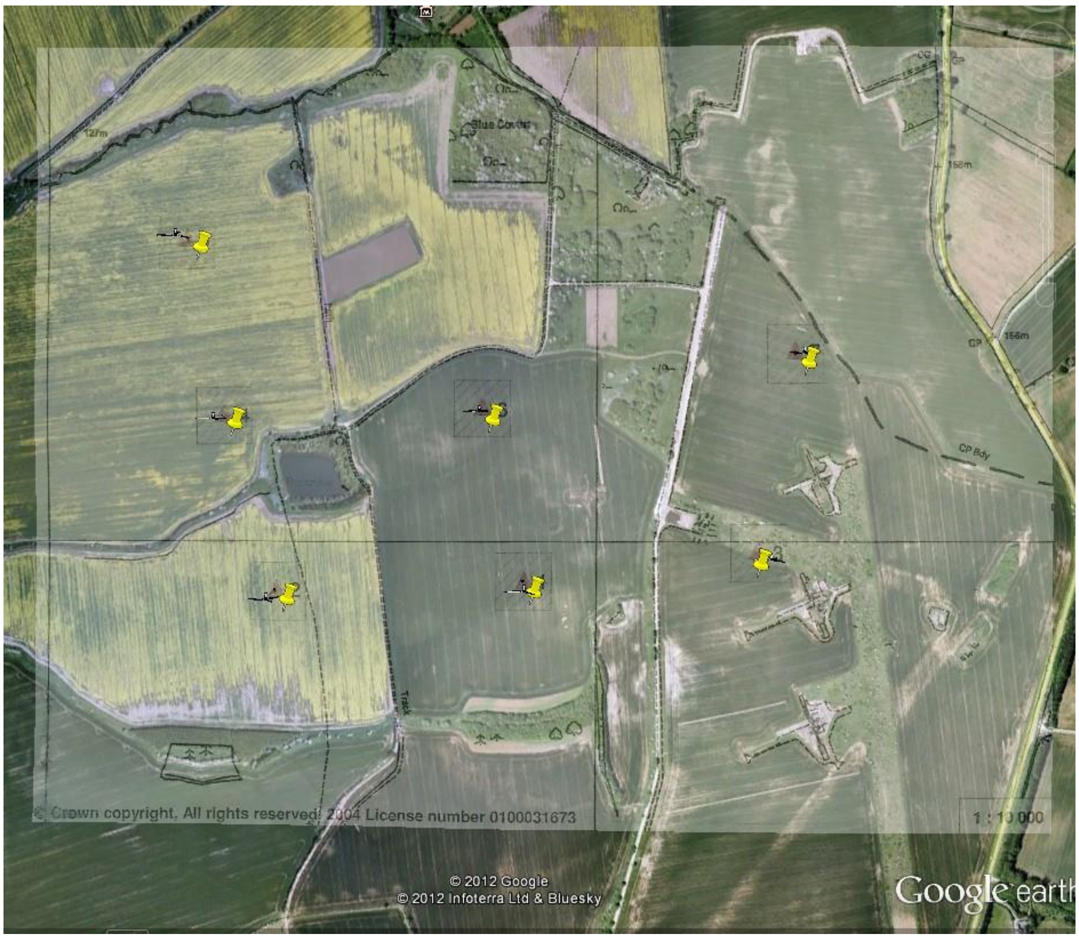
```
> install_github("digitize", username="tpoisot", subdir="digitize")
```

You should now be ready to go, but you also need a .jpg image on which to test the package. Mine (available on the learning management system) is a simple scanned Google Earth Image™ on which are the locations of seven planned wind turbines close to my home in England.

Using *digitize* involves four steps:

- Accessing the .jpg image
- Producing some calibration data
- Digitising selected points
- Applying the calibration to create a data.frame containing the points.

The image called Harrington_with_turbines.jpg looks like:


Blue Covert
127m
156m
CP Bdy
Track
© Crown copyright. All rights reserved. 2004 License number 0100031673
1 : 10 000
© 2012 Google
© 2012 Infoterra Ltd & Bluesky
Google earth

Each turbine has a yellow Google pin (the three funny cross shaped things are the ruins of three launch pads for Thor/Delta IRBM's from the Cold War days of 1959-1962; the obvious old runways were those of a WW2 American air base used to fly spies into occupied Europe)

```
>library(digitize) ## load the package
Loading required package: jpeg
> cal <-
ReadAndCal("C:\\Users\\David\\Desktop\\Harrington_with_turbines.jpg")
```

This opens the jpg image in a plotting window and lets you define points on the **X** and **Y** axes. I would be the first to admit that these steps aren't very user-friendly but once you are used to them there isn't a problem. First, note that the cursor has become a large black cross. You *must* start by clicking on the *left-most* **X**-axis point, then the *right-most* **X** axis point, followed by the *lower* **Y**-axis point (which might be the same location as your first point but does not have to be) and finally the *upper* **Y**-axis point. These *tick points*, as they are called in the GIS world, will have blue crosses on them. Make sure you know the real coordinate values at each of these tick marks, you will need them later. It is worth looking at the cal object we have created:

```
> cal
$x
```

```
[1] 0.002965642 0.998987342 0.002965642 0.002965642
$y
[1] 0.003043478 0.001086957 0.001086957 0.996956522
```

Note that digitize initially assumes a unit square (The co-ordinates are close to 0,1). Next we capture the event locations:

```
> data.points <- DigitData(col = 'red')
```

The cross hair cursor becomes blue and we work our way round the required event locations, *left* clicking on every one. The operation is completed using a *right* click (which gives an option to stop or continue if you miss a point). We now have our seven events in the same co-ordinates as the tick points:

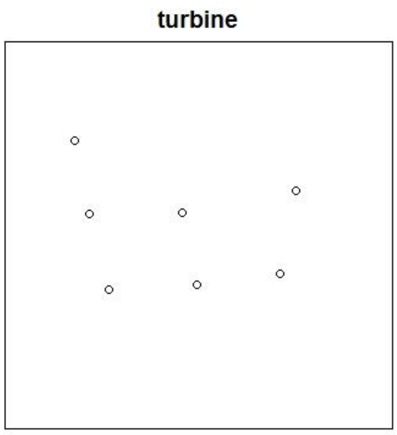
```
> data.points
$x
[1] 0.1826401 0.2197468 0.2724774 0.4580108 0.4970705 0.7529114 0.7099458
$y
[1] 0.7406522 0.5528261 0.3591304 0.5547826 0.3708696 0.6115217 0.3982609
```

Calibration is straight forward. In this case I simply ensure that the tick points really do define a unit square, but any other real world values could be used:

```
> df  <- Calibrate(data.points, cal, 0,1,0,1) ## the arguments are obvious …
> df
          x         y
1 0.1803922 0.7426326
2 0.2176471 0.5540275
3 0.2705882 0.3595285
4 0.4568627 0.5559921
5 0.4960784 0.3713163
6 0.7529412 0.6129666
7 0.7098039 0.3988212
```

Just to show it can be done we import these data into spatstat as a *ppp* class of object using a sequence we have used before:

```
> library (spatstat)
> xlim<-c(0,1)
> ylim<-c(0,1)
> turbine<-ppp(df$x,df$y,xlim,ylim)
> plot(turbine)
```



If you intend at some point in the future to use this approach with some real data please note that the original image (or the captured data) will probably need to be rotated so that the **X** and **Y** axes are perfectly horizontal and vertical and that digitize only seems to recognise jpeg images.

I can think of lots of uses for this package, but for now let's …

```
>quit()
```

Revised with new example 22-10-14, 02-12-15