Buffering Stuff

Andy asked me to see if I could figure this out this buffering problem in R, and this document is instructions for how I went about this. Here’s what he initially asked for:

* A way to buffer a stream network by a certain amount (like 100m)
* Generate a grid of points (maybe 100m spacing)
* Identify which points fall within the buffer
* Compute the distance (or least cost path) between any two points while staying within the buffer

From my understanding, the purpose of this is to try to model the movements of minks in the refuge. It is known that minks do not like to move to far away from streams, so the idea is that the points outside the buffer of the stream will have a higher “cost” to the minks that they ones within the buffer. That way, we will be able to calculate the path between points with the least costs to the mink.

I’m new to R so it’s likely the process I came up with isn’t the best. That being said, please feel free to comment, and/or suggest any improvements!

The first part of this document is what I came up with initially, and the second part is stuff Andy and Tabitha put together. It might be more useful to skip ahead to that stuff - I didn’t want to delete the first part just in case. Also, the last part of the document is stuff I’ve been working on since Andy and Tabitha put together a script for this.

The following packages throughout this process:

* maptools
* rgeos
* sp
* gdistance
* shapefiles
* raster
* spatstat

**Initial Process**

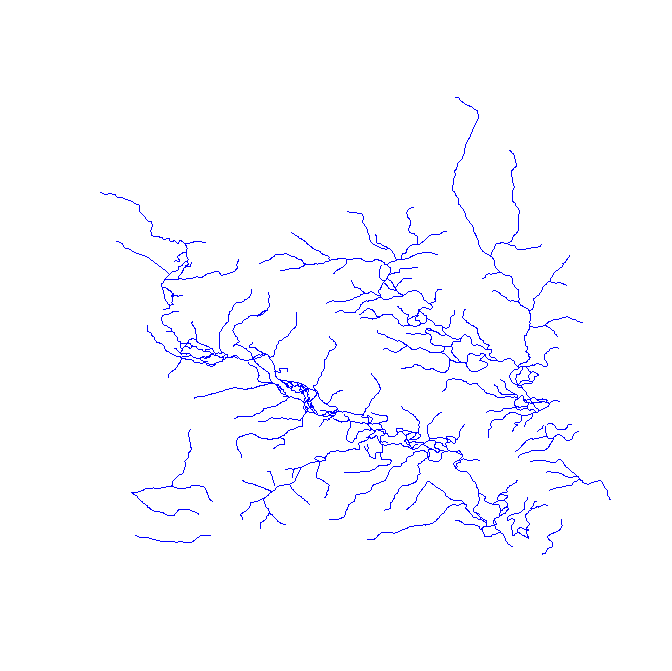
To get something to use for the stream network, I first requested a shapefile from the National Map by USGS (<http://nhd.usgs.gov/index.html>) of the hydrology of PWRC. If you want a different stream network, it took them a couple hours to email me the link to download. I can also send you the shapefile I used if you need it.

Save the shapefile in the same place that you are going to use for your working directory in R.

I then used the maptools command readShapeLines to bring that data into R.

streamnet<- readShapeLines("pwrc\_streamnet")

plot(streamnet, col="blue")

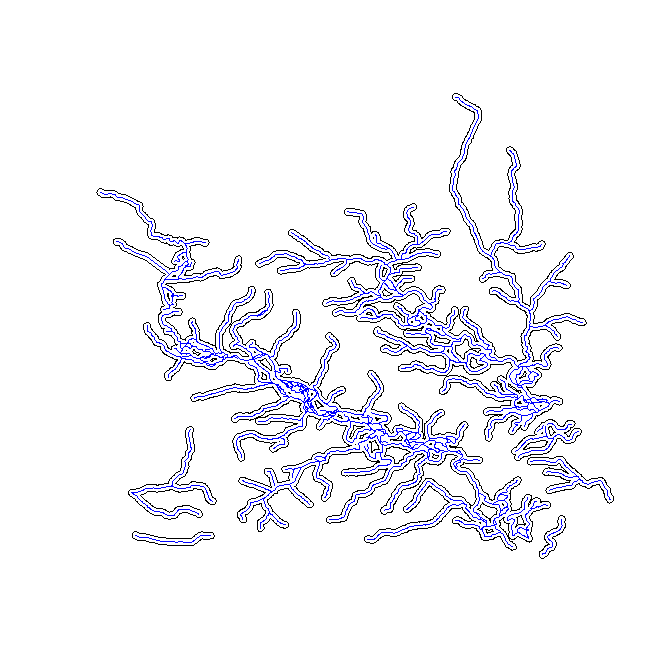


Next, I used the rgeos command gBuffer to apply a buffer of a fixed width to the streamnetwork.

streambuf<- gBuffer(streamnet, width=0.001)

plot(streambuf)

plot(streamnet, col="blue", add=TRUE)



I believe the units are in decimal degrees, but I’m not entirely sure.

To create a grid of points I first had to define the x and y limits. I made sure the min and max of each were a little big outside of the min and max of streambuf.

X<- seq(min, max, by=0.01)

y<- seq(min, max, by =0.01)

The command expand.grid is then used to create a set of points based on the x, and y points we just defined.

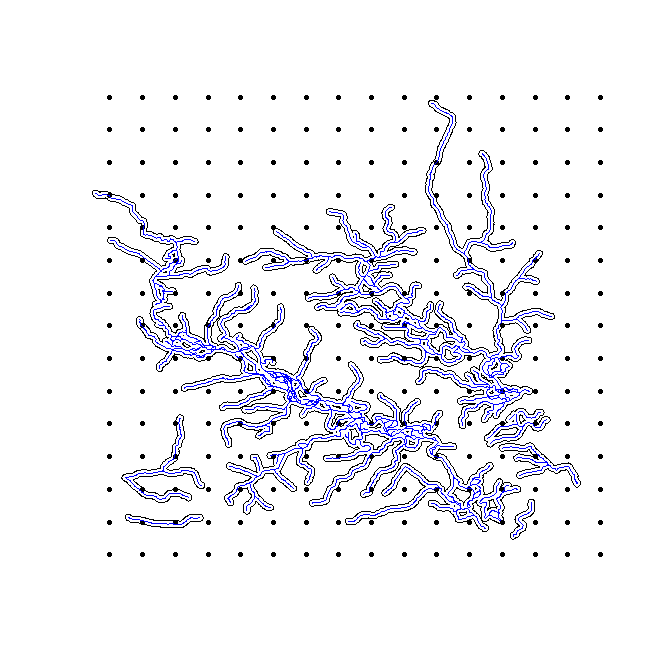
g<- expand.grid(x, y)

gg<- spatialpoints(g)

plot(gg, pch=20)

plot(streamnet, col=”blue”, add=TRUE)

plot(streambuf, add=TRUE)



I had to use the command spatialpoints because otherwise the grid points cannot be plotted with the stream network and buffer.

Next, I had to figure out which grid points fell within the buffer. After searching around awhile, I found this example online: <http://www.nceas.ucsb.edu/scicomp/usecases/point-in-polygon>

They were trying to do essentially the same thing (figure out which bear sightings fell within National park boundaries).

I essentially modified what they had done, which was using is.na with the over command to determine if each point was within the stream buffer.

inside.buf<- !is.na(over(gg, as(streambuf, "SpatialPolygons")))

inside.buf

[1] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE

[13] FALSE FALSE FALSE FALSE FALSE TRUE TRUE FALSE FALSE FALSE FALSE FALSE

[25] FALSE FALSE FALSE FALSE TRUE FALSE FALSE FALSE FALSE TRUE FALSE FALSE

[37] FALSE FALSE FALSE FALSE TRUE FALSE TRUE TRUE TRUE FALSE FALSE FALSE

[49] FALSE FALSE TRUE FALSE FALSE TRUE FALSE TRUE TRUE FALSE TRUE FALSE

[61] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE TRUE FALSE FALSE TRUE

[73] TRUE TRUE TRUE FALSE TRUE TRUE FALSE FALSE FALSE FALSE FALSE FALSE

[85] FALSE TRUE TRUE FALSE FALSE FALSE FALSE FALSE TRUE FALSE FALSE FALSE

[97] FALSE FALSE FALSE FALSE TRUE TRUE TRUE FALSE FALSE TRUE FALSE FALSE

[109] TRUE FALSE FALSE FALSE FALSE TRUE FALSE TRUE FALSE TRUE FALSE FALSE

[121] FALSE TRUE TRUE FALSE TRUE FALSE FALSE FALSE FALSE FALSE TRUE FALSE

[133] FALSE FALSE FALSE TRUE TRUE TRUE FALSE FALSE TRUE FALSE FALSE FALSE

[145] FALSE TRUE TRUE FALSE FALSE FALSE TRUE FALSE TRUE FALSE FALSE TRUE

[157] FALSE TRUE FALSE FALSE FALSE TRUE FALSE FALSE FALSE FALSE FALSE FALSE

[169] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE TRUE FALSE FALSE FALSE

[181] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE

[193] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE TRUE FALSE

[205] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE

[217] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE

[229] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE

summary(inside.buf)

Mode FALSE TRUE NA's

logical 192 48 0

As you can see, this assigned each point either FALSE or TRUE (False meaning it did not fall within the buffer, True meaning it did). This shows that there were 48 instances where the point was within the buffer.

Next, I extracted the 48 points that were inside the buffer, to create a new object which I called p.inside.

p.inside<- gg[inside.buf, ]

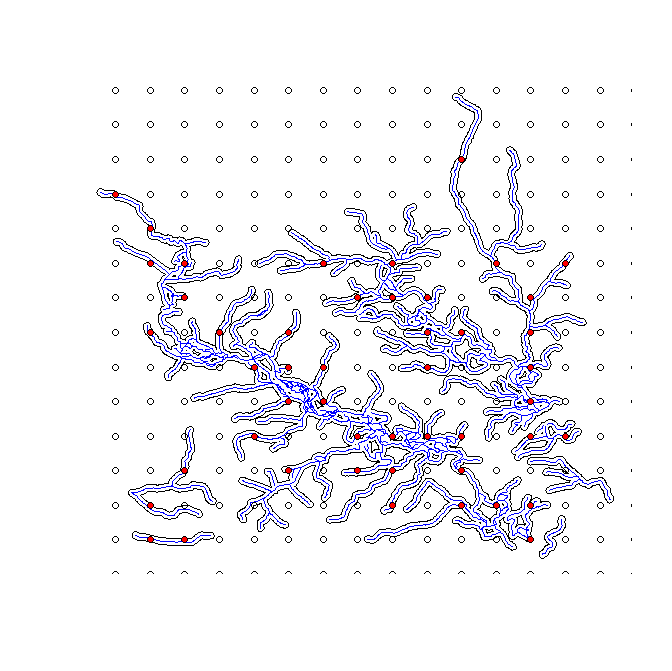
I re-plotted everything to make sure the points were correct.

plot(streamnet, col="blue")

plot(streambuf, add=TRUE)

plot(gg, add=TRUE, pch=1)

plot(p.inside, add=TRUE, pch=20, col="red")



So finally, we need to find the distance between any two points that fall within the buffer.

There’s another R package called gdistance that has a command to calculate the least cost path between two points. (NOTE: gdistance is different from the command gDistance that is part of the package rgeos) Andy had sent me some code that uses the gdistance package but I wasn’t able to modify it to get it to work for this. Here’s what Andy sent me:

LCP<- function(beta1,beta2,outD) {  
 costcell<- 1 + beta1\*var1 + beta2\*var2 #  
 r<-setValues(r,costcell)  
 r<-1/r  
 tr1 <- transition(r, transitionFunction = mean, directions=8)  
 tr1CorrC <- geoCorrection(tr1, type = "c", multpl = FALSE) #have to do this geocorrection step even if in UTMs with these settings...   
 costs1<-costDistance(tr1CorrC, sP)  
 outD<-as.matrix(costs1)  
 outD<-outD[lower.tri(outD,diag=FALSE)]  
 }

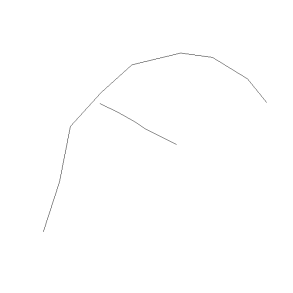
The command called costDistance is what we would need to use to calculate the least cost path. However, the help stuff online wasn’t enough for me to figure out what attributes/arguments it needed to work. I tried a bunch of different things, but I’m not sure if I have enough experience with R or least cost path calculations to get it to work.

**Andy and Tabitha’s Script**

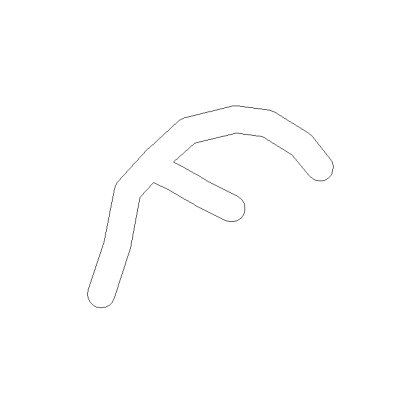
The SCR-river script goes through the entire process, including calculating the least cost path. Some of the steps are the same as before, but some are not. It also doesn’t use the PWRC stream network, but a simple, made-up stream. In the next few days I’m going to try to create another script to incorporate a PWRC shapefile I used before.

I’m not going to go into as much detail in this section since you can see all the commands in the script file, and because I’m not entirely sure what some functions do.

The beginning of the script is mostly creating the fake stream network, which looks like this:

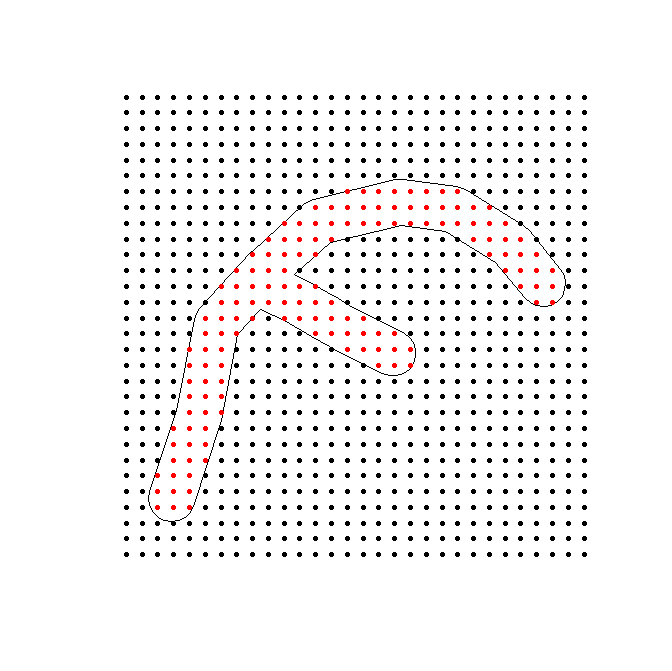


It then uses gBuffer, just like before, to create a buffer around the stream.



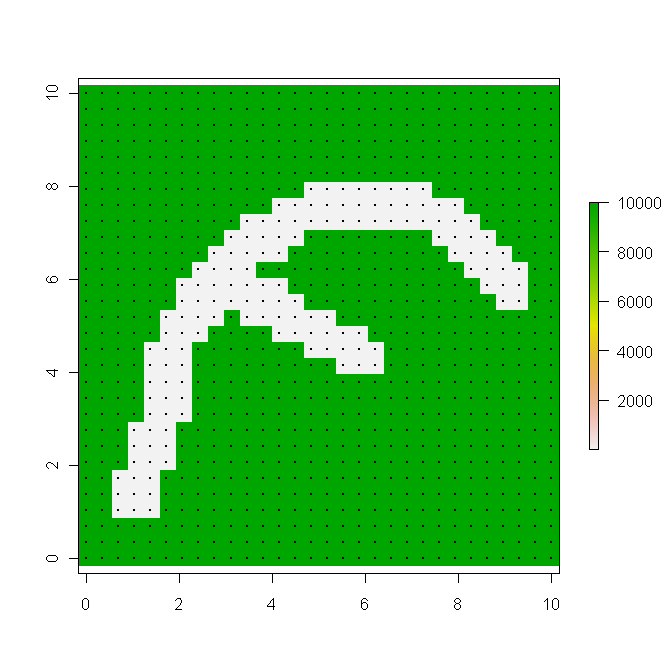
Then they created a grid of points by creating two sequences of numbers (one for x, and one for y) then combining them and converting them into points.

To figure out which points fall within the buffer, the function point.in.polygon was used. Here’s the result:



Next, they applied “costs” to the points, and the higher the costs the less likely it is for the mink to travel to that point. The points outside the buffer got a cost of 10000, while the points within the buffer got a cost of 1.

Then, it looks like the grid of points is converted to a raster and re-plotted to give a better visual representation of the costs of moving in and out of the buffer.



Now, the least cost path can be calculated. The way this is gone about is similar to the code I couldn’t get to work. More specifically, it uses the functions transition, geoCorrection, and costDistance (in that order). All three of the functions are part of the gdistance package. Correct me if I’m wrong, but transition associates all the adjacent cells, geocorrection corrects for the projections, and costDistance calculated the least cost path.

To apply the least cost path function two points were then picked out. This is what was done:

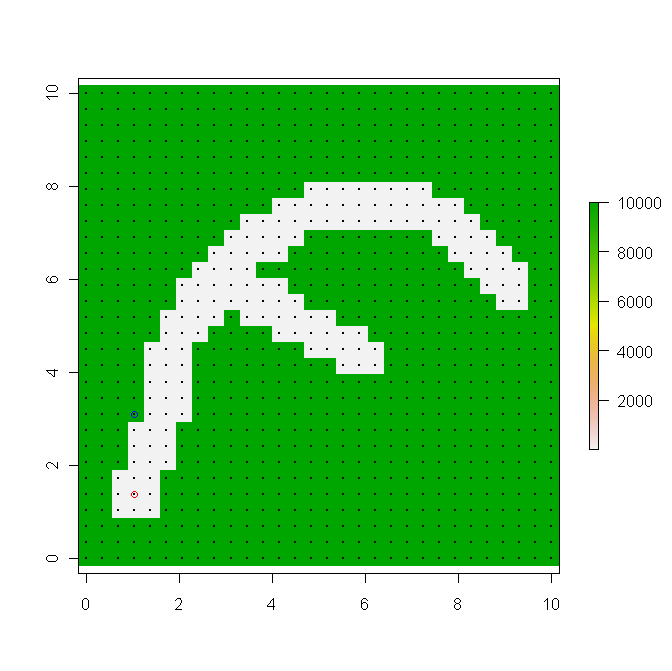
outD<-as.matrix(costs1)

points(matrix(pts[116,],ncol=2),col="red")

points(matrix(pts[111,],ncol=2),col="blue")

outD[116,100:120]

116 and 111 are the individual points in the grid, as each one has a consecutive number to identify it. This code also identifies the points on the map with a red and blue marker.



I’m not entirely sure what the outD part of the code means, but it will output the distances between the points. Here’s what you get:

100 101 102 103 104 105

9758.5958971 8330.2732683 6901.9506394 4881.1649188 3452.8422899 3452.0098025

106 107 108 109 110 111

3.7339404 3.3891128 3.0442852 2.6994576 2.3546300 1.7241379

112 113 114 115 116 117

1.3793103 1.0344828 0.6896552 0.3448276 0.0000000 0.3448276

118 119 120

0.6896552 3448.9655172 6897.2413793

The 100 numbers are the points in the grid, and the numbers underneath is the distance from that point to point 116 (I think?)

The script does quite a bit more but I’m not entirely sure what at this point.

**New stuff**

For the more recent stuff I’ve clipped the stream network to just include the main water body in Patuxent to keep things simpler.

To be honest, I forget why but Andy wanted me to see if there was a way to make the stream network a set of points instead of lines. After searching around for a little I found the function pointsOnLines will place points evenly along lines and is a part of the spatstat package. However, for it to work the input needs to be something called a psp object, which is a line segment pattern.

To convert the stream network object to psp you need to use the function as.psp, which is also a function from spatstat.

pspstreamnet<- as.psp(streamnet)

Warning message:

In as.psp.SpatialLinesDataFrame(streamnet) :

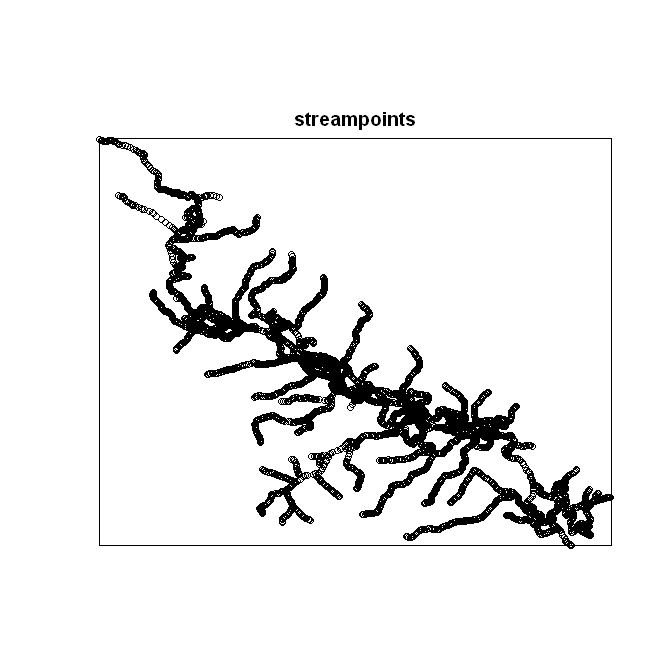
13 columns of data frame discarded

Even though there was a warning it still seemed to work

I then used pointsOnLines to convert the line to points.

streampoints<- pointsOnLines(pspstreamnet)

Here’s what it looks like plotted.

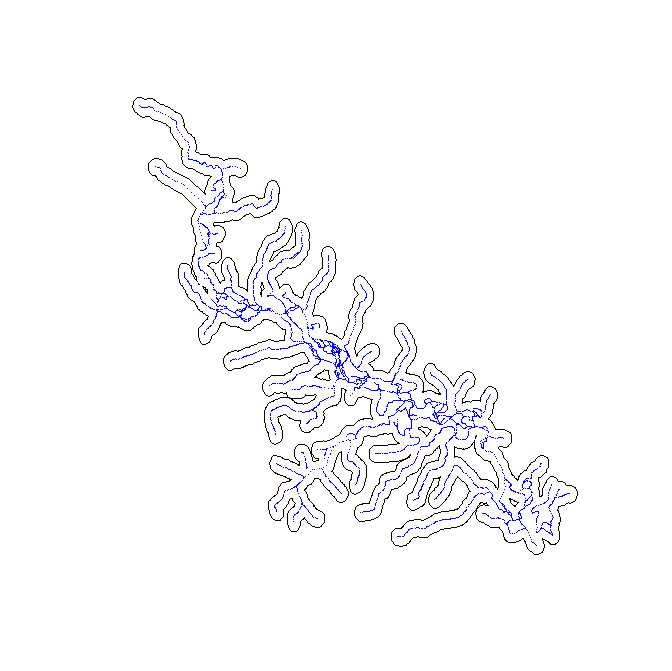


Since the stream points are now a psp object you can’t use it with the gBuffer function to create the buffer. However using the original shapefile (which I called streamnet) with gBuffer should get you the same results (this is the same method as I used in the first section of this document). The psp object might also affect some calculations later depending what you want to do. I know this is annoying, but the only other R function I saw to convert to point just created points at the vertices of the lines, which might not be that useful for us. I also couldn’t figure out which package that function was in. I’ll keep looking into this – hopefully there’s an alternative function or a way to convert the psp object back into points.

Because of the symbology, it’s hard to see both the buffer and stream points at the same time so I had to do this:

> plot(streambuf)

> plot(streampoints, pch=".", col="blue", add=TRUE)



Since the points are very close together it still looks like lines, but they are definitely points.

I have also taken the script written by Andy and Tabitha and changed it around to work with the shapefile’s I’ve been using. For now, I’ve stopped after calculating the cost distance between points 111 and 116 because I’m not sure what the rest of the code does.

Everything so far seemed to work well with the script after some minor modifications.

Again, let me know if you want me to send you the shapefiles.

**Fixing the transition function**

Andy realized that the results of costDistance weren’t making sense, and he and Tabitha were able to figure out what was going wrong.

The main problem with the way we were calculating costDistance was the transitionFunction we were using called max. We should really be using transitionFunction=function(x) 1/mean(x) with directions = 4 instead of 8.

We were also inversing the raster, r, which they also found unnecessary. Instead, tr1CorrC (the result of the function geoCorrection) needed to be inversed. However, this caused an error because this object doesn’t have a summary object to tell it what to do. Later, andy found that this also was unnecessary.

To test this, andy created a simple 4x4 raster to understand costDistance better. This is how it was done:

library("rgeos")

library("shapefiles")

library("gdistance")

library("raster")

###png("raster.png",width=5,height=5, units="in", res=400)

r<-raster(nrows=4,ncols=4)

projection(r)<- "+proj=utm +zone=12 +datum=WGS84"

extent(r)<-c(.5,4.5,.5,4.5)

values(r)<-matrix(1:16,4,4,byrow=FALSE)

par(mfrow=c(1,1))

plot(r)

###dev.off()

#points(pts,pch=20,cex=.4)

#r<-1/r

## use max = doesn't count moving through boundary pixel

tr1<-transition(r,transitionFunction=function(x) 1/mean(x),directions=4)

tr1CorrC<-geoCorrection(tr1,type="c",multpl=FALSE,scl=FALSE)

tr1CorrC<-1/tr1CorrC

xg<-seq(1,4,1)

yg<-4:1

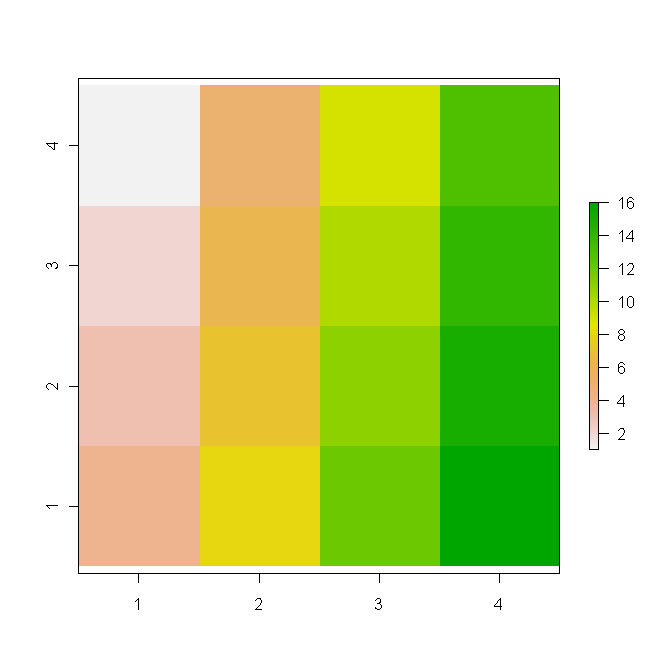
pts<-cbind( sort(rep(xg,4)),rep(yg,4))

costs1<-costDistance(tr1CorrC,pts)

outD<-as.matrix(costs1)

outD[3,5]

Here is what it looked like



There are 16 pixels, with 1 started at the top left, and ending at 16 at the bottom right. In addition, the “cost” to move through the pixel gets higher as the color becomes more green.

As you can see by the code, the cost distance between pixel 3 and pixel 5 is being calculated. The result of that is 7. To get from 3 to 5, you would have to travel from 3 to 2, then 2 to 1, and finally 1 to 5. The cost to move each time needs to be calculated then added together:

(1) move from pixel 3 to pixel 2, distance =1, cost = (3+2)/2 = 2.5

(2) move from pixel 2 to pixel 1, distance =1, cost = (2+1)/2 = 1.5

(3) move from pixel 1 to 5, distance =1, cost = (1+5)/2 = 3

2.5+1.5+3 = 7