Using the Mosaic Package

Basic steps

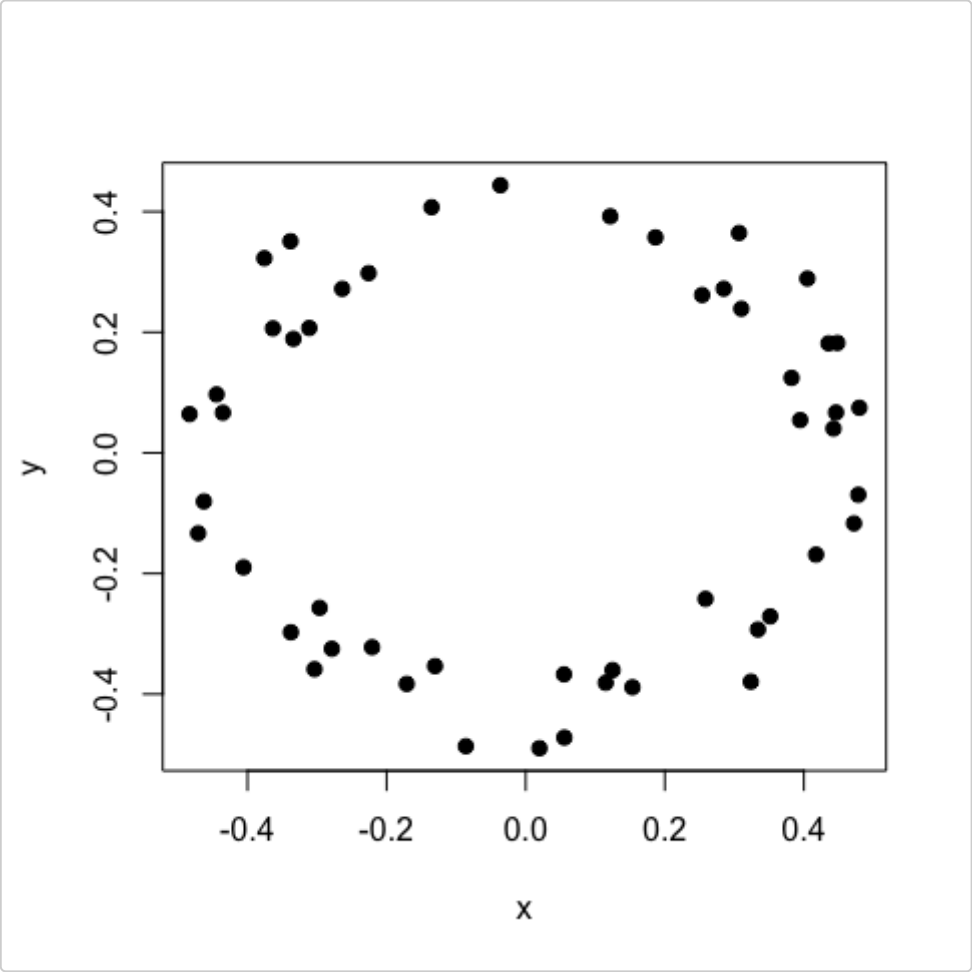
mosaic has no strict dependencies, so you can get started right away.

First, load up *mosaic*, create some coordinate data falling in a ring-shaped pattern using *mosaic*'s *ptpattern* function, and stash them in x and y arrays:

```
library(mosaic)
p <- ptpattern(50,shape='ring')

x <- p$x
y <- p$y
```

Make a quick plot to get an idea of the density of sampling, and check the ranges. The data are centered on the origin, with a default maximum range of -0.5 to 0.5. This can be changed by setting the diameter parameter, which defaults to 1.

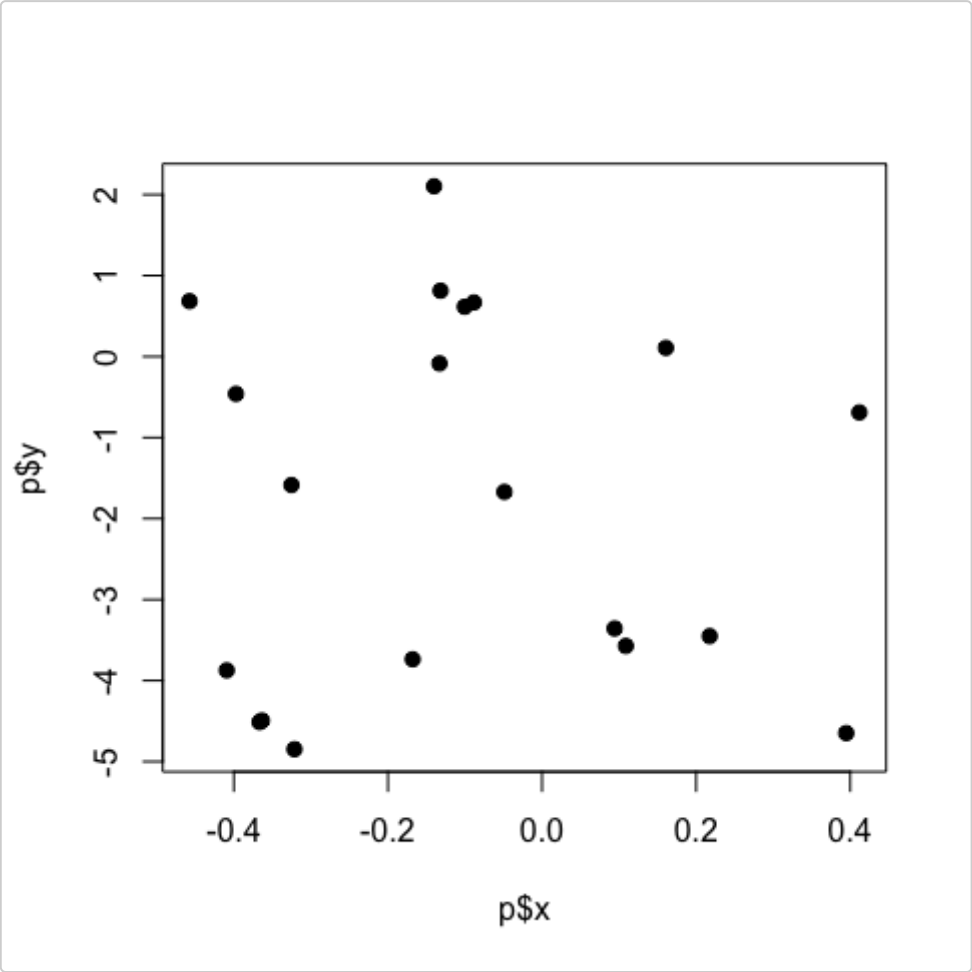
```
plot(x,y,pch=19)
```



```
range(x)
#> [1] -0.4833256  0.4797628
range(y)
#> [1] -0.4897019  0.4436670
```

We'll go back to the x and y data momentarily, but note that *ptpattern* will produce other shapes: circle, square, and rectangle. If 'rectangle' is chosen, the default aspect ratio is 2, meaning that the x-axis data range from -1 to 1. So, to create a rectangle 1 unit wide and 10 units high, use both the *diameter* and the *asp* parameters:

```
p <- ptpattern(points=20,shape='rectangle',diameter=10,asp=0.1)
plot(p$x,p$y,pch=19)
```



Note the x and y axis limits.

The main purpose of *mosaic* is to estimate areas of shapes like the ring shown above. *tgraph* bases its estimates on minimum spanning trees (MSTs) and *mgraph* bases its on MSTs combined with mosaic graphs.

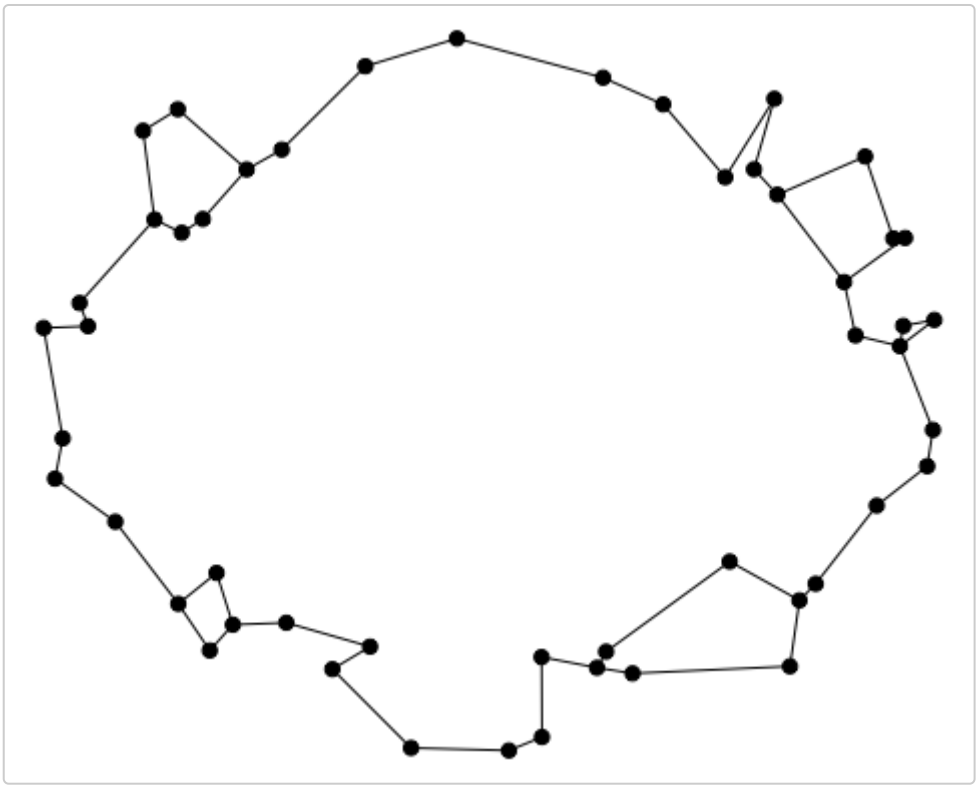
Going back to the data, the areas are easy to extract:

```
t <- tgraph(x,y)
m <- mgraph(x,y)
t$area
#> [1] 0.7838098
m$area
#> [1] 0.4450432
```

Plotting mosaics and trees

Both functions use *plot.mosaic* to present graphs. First plot a simple mosaic:

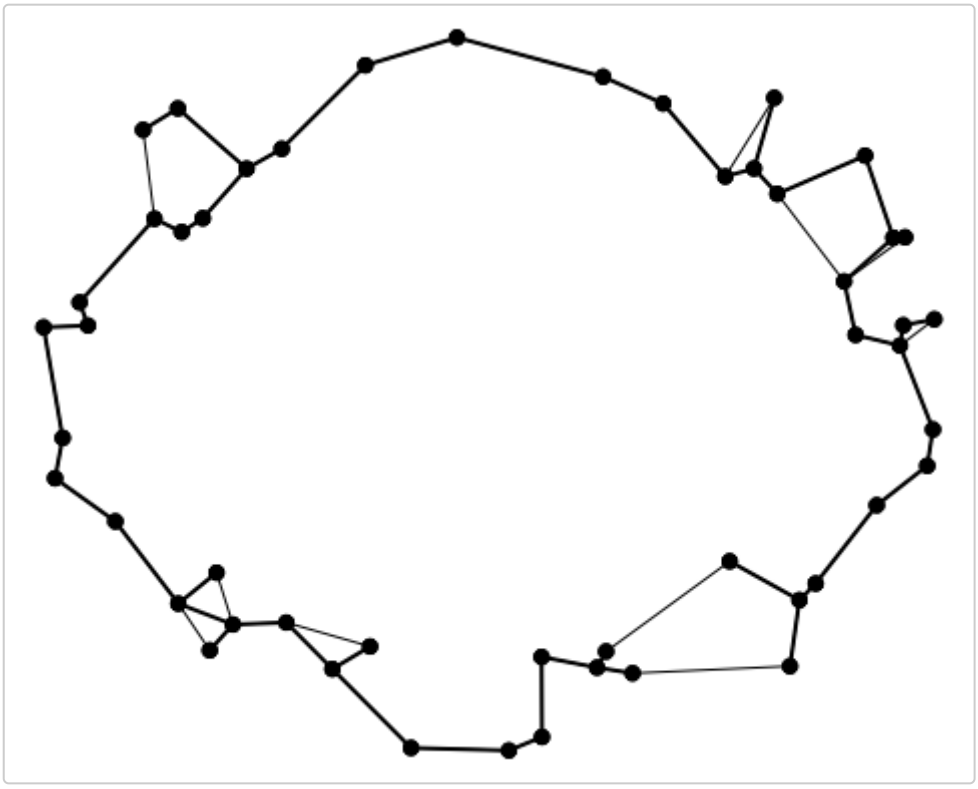
```
m <- mgraph(x,y)
plot(m)
```



See how *plot.mosaic* fills up the entire space and omits superfluous axes and axis labels.

The overall MST doesn't need to match up exactly with the mosaic, but can come close. As you might expect, *plot.mosaic* respects add=T and lwd:

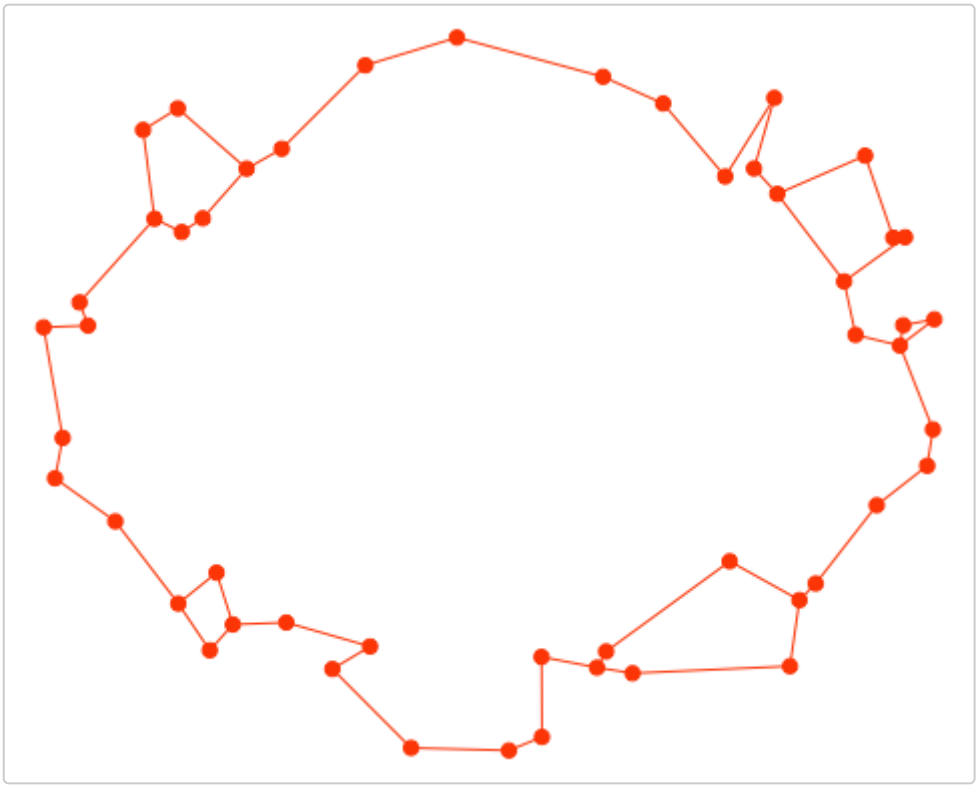
```
plot(m)
plot(t,add=T,lwd=2)
```



So if you really wanted those axes and labels, you could plot the data using some other plot method and then add the mosaic or MST using add=T. Note that axis() and mtext() wouldn't work because the figure's margins are set to zero automatically.

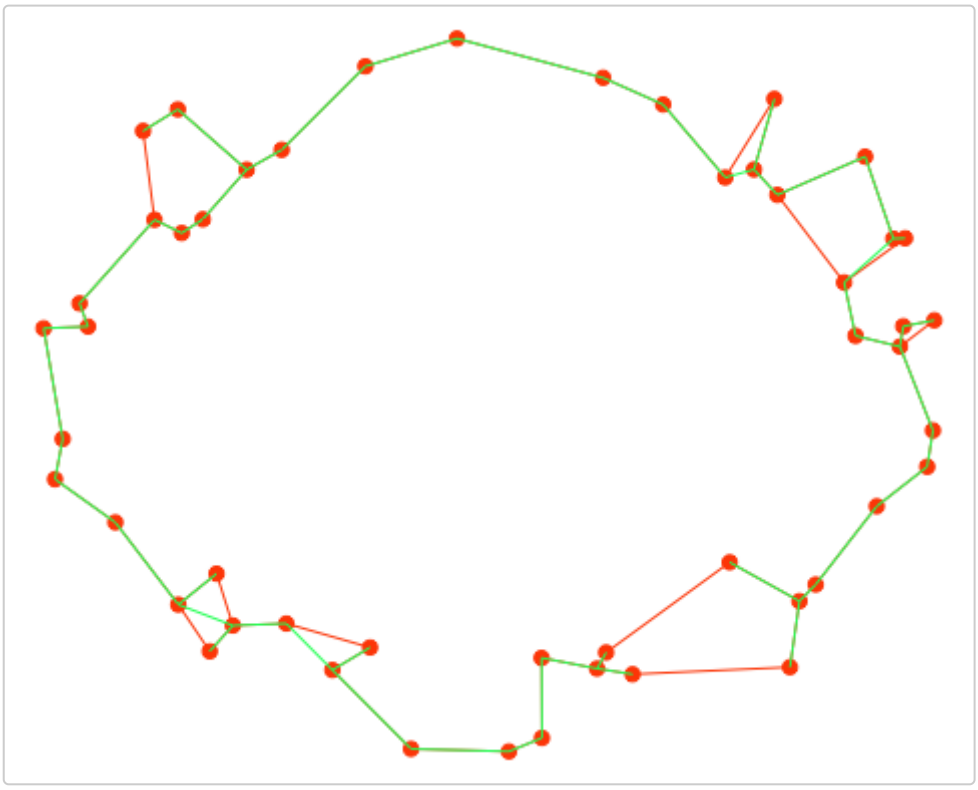
plot.mosaic has numerous options. You can specify colors by entering either a *col* value like "red" or "blue" or an *hsv* specification. In the latter case, colors are set by entering h, s, v, and alpha parameters. *plot.mosaic* defaults to a light blue color (h = 0.5), zero saturation (s = 0), medium darkness (v = 0.5), and halfway transparency (alpha = 0.5), so to get any non-black color at all you need to ramp up the saturation. Lightening things up with v=1 also helps. Also, *plot.mosaic* needs to be told to color the lines and points by toggling *pts.black* and *lines.black*. The hue of red is about 0.05. We'll omit the lwd=2 part because colors will now differentiate the graphs.

```
plot(m,h=0.05,s=1,v=1,pts.black=F,lines.black=F)
```



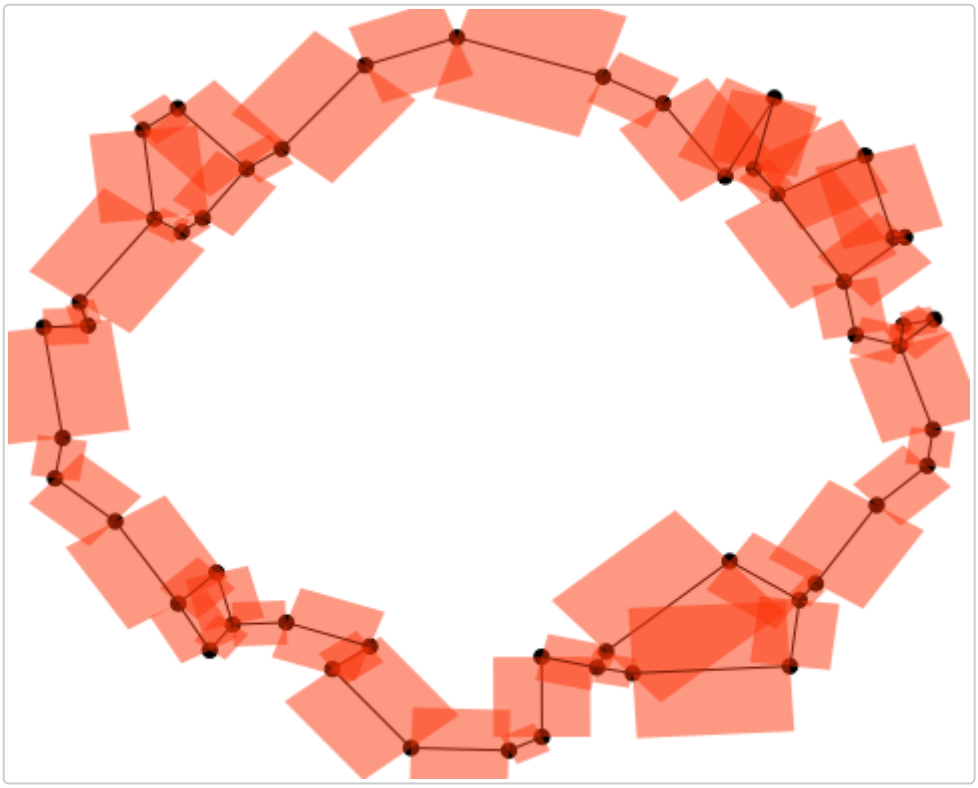
Now add the MST in green (h = 0.4), stripping out the points in the second call to *plot.mosaic* because they're shown already. This is done by setting cex to zero.

```
plot(m,h=0.05,s=1,v=1,pts.black=F,lines.black=F)
plot(t,add=T,h=0.4,s=1,v=1,lines.black=F,cex=0)
```



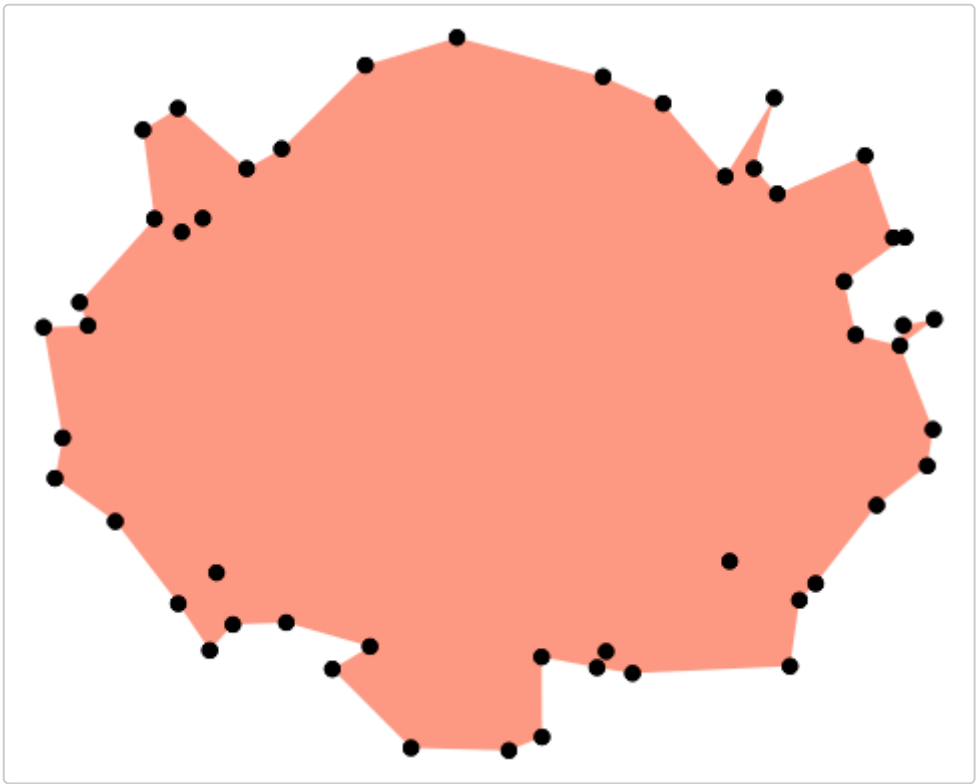
mgraph bases its estimates on the square of summed edge lengths. To visualize the "area" contribution of each edge, use the *squares* option. Leave the points and lines black so you can tell them apart from the squares.

```
plot(m,h=0.05,s=1,v=1,squares=T)
```



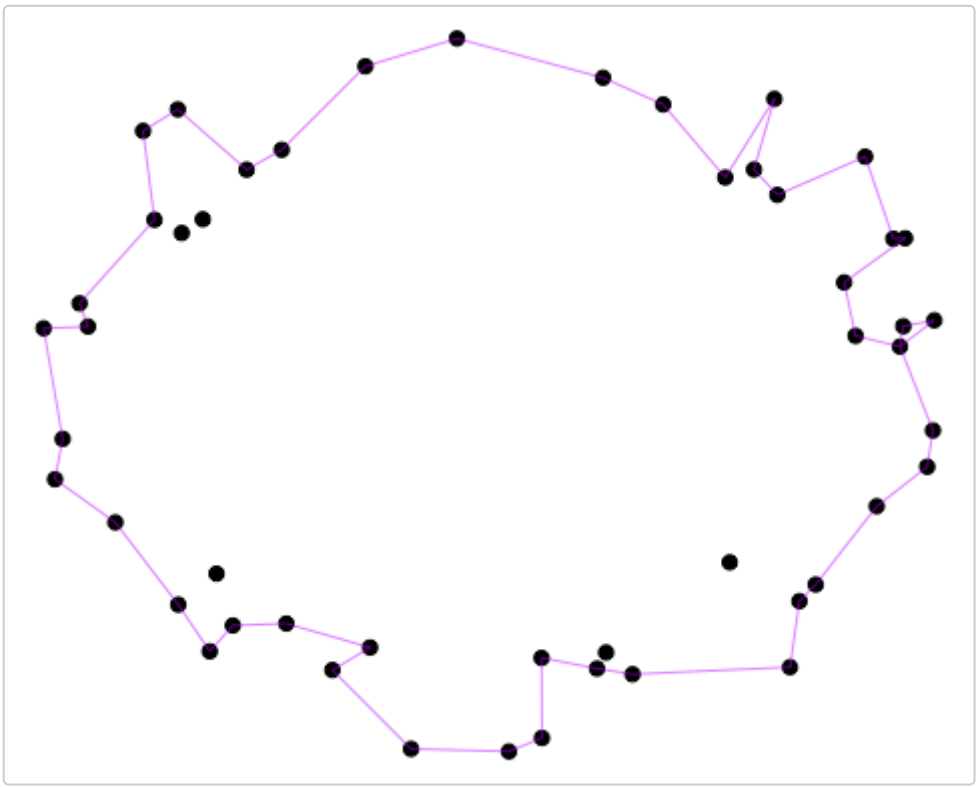
plot.mosaic also allows you to visualize the data by filling in the hull of the mosaic. This isn't a convex hull, because it strictly follows the outermost edges in the graph. *plot.mosaic* automatically uses a utility function called *mhull* to do this; *mhull* is fast, so the process is seamless.

```
plot(m,h=0.05,s=1,v=1,fill=T,lines=F)
```



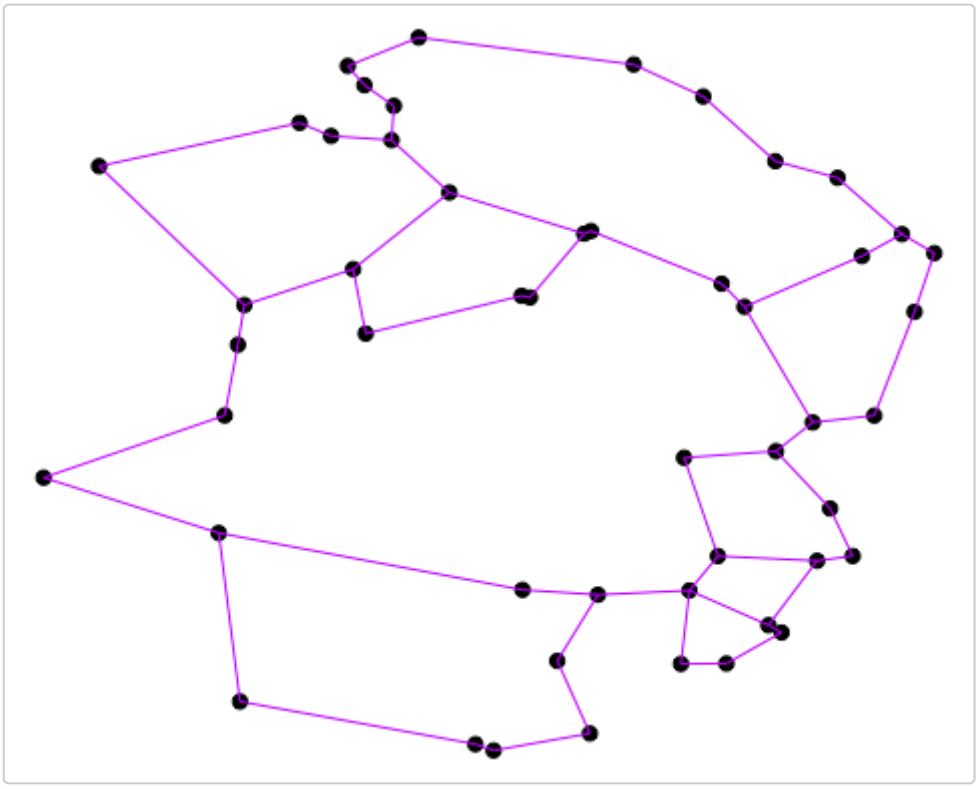
plot.mosaic can also just draw an outline to show the hull, but it will only be easily visible if you suppress the edge lines using lines=F. Let's try purple (h = 0.8) this time.

```
plot(m,h=0.8,s=1,v=1,outline=T,lines=F)
```



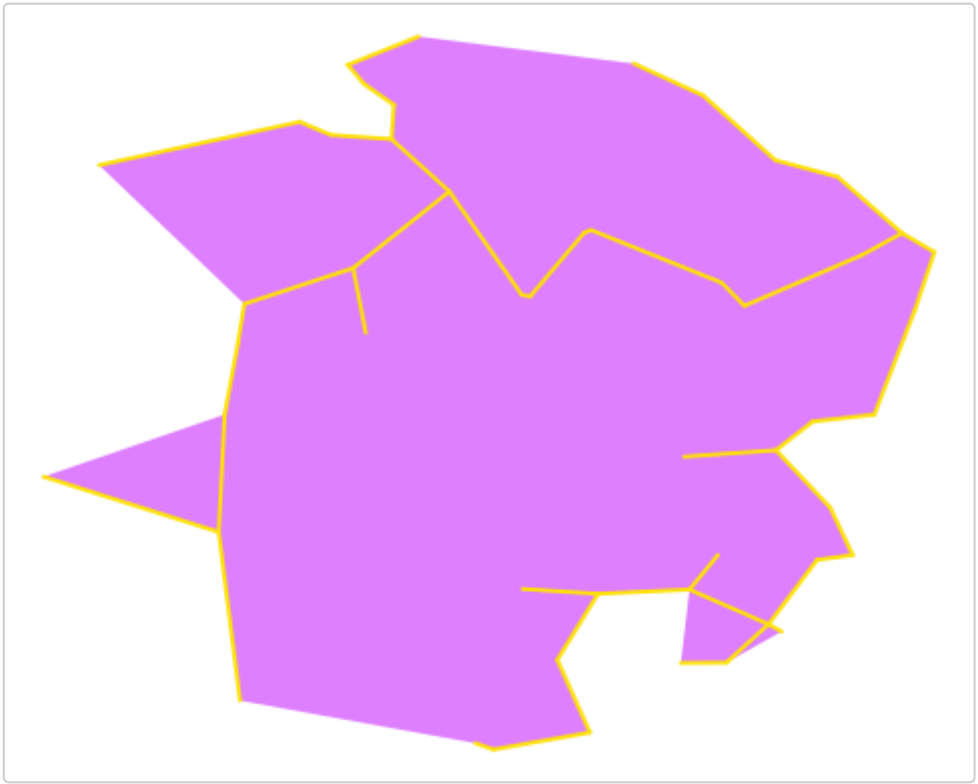
Another possibility is to add an MST. We'll use a square shape instead of a ring shape because MSTs are boring when the data fall in a ring. First, here's the basic mosaic shape. Note how *mgraph* and *tgraph* can be embedded within the *plot* call.

```
p <- ptpattern(50)
plot(mgraph(p$x,p$y),h=0.8,s=1,v=1,lines.black=F)
```



Second, the filled mosaic with the MST on top (in yellow: h = 0.15). Note that the MST doesn't have to follow the mosaic, so it can even jump outside of the filled hull. Drop the points, and don't forget lines.black=F and add=T:

```
plot(mgraph(p$x,p$y),cex=0,h=0.8,s=1,v=1,fill=T,lines=F)
plot(tgraph(p$x,p$y),cex=0,h=0.15,s=1,v=1,lwd=2,lines.black=F,add=T)
```

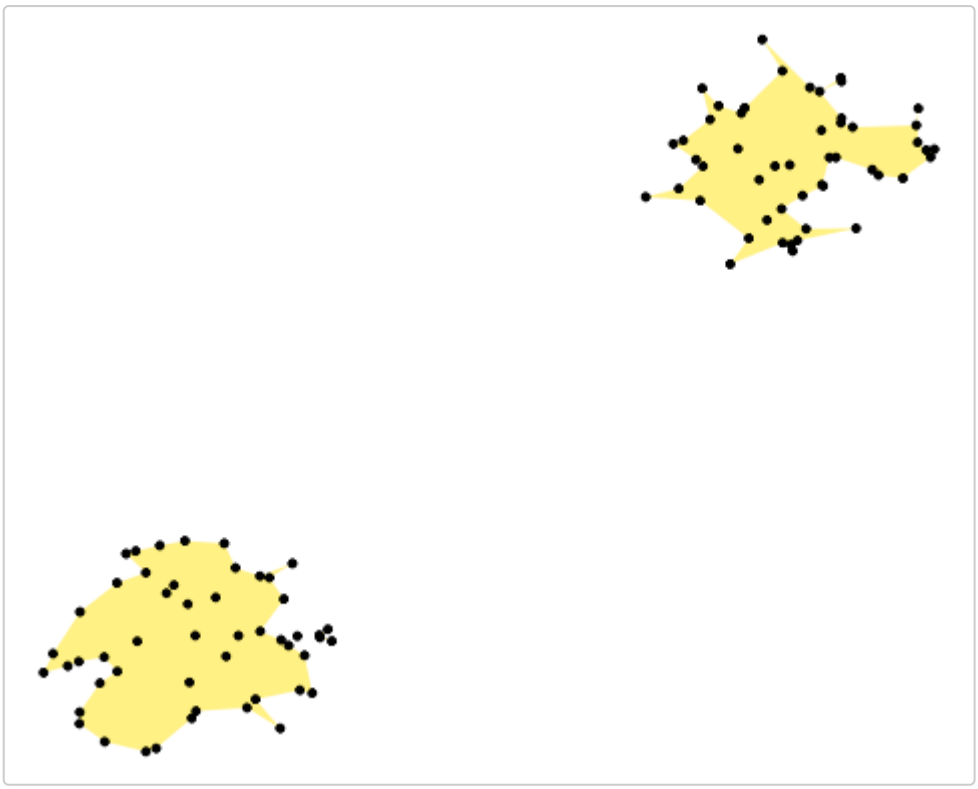


mgraph will break up the data into separate networks depending on how the *neighbors* and *mutual* options are set. By default, if all of the points in one region have nearest neighbors in that region alone, and likewise

the points in other regions, then no edges will be drawn between the regions. To put this more simply, if the data form tight clusters then the clusters will be kept separate.

This is nothing to worry about because the routines for estimating area and for plotting the data handle separate clusters automatically. Here's an example showing two disconnected subgraphs.

```
p1 <- ptpattern(50)
p2 <- ptpattern(50)
x <- c(p1$x,p2$x + 2)
y <- c(p1$y,p2$y + 2)
plot(mgraph(x,y),cex=0.5,h=0.15,s=1,v=1,fill=T,lines=F)
```



To summarize, *mgraph* provides x and y coordinates in addition to area estimates. It also can tell you the length of the mosaic, the distances between points, and the structure of the graph in terms of a binary, triangular matrix, where 1 = connected and 0 = disconnected:

```
m <- mgraph(runif(10),runif(10))
m$length
#> [1] 3.379446
m$distances
#>           1         2         3         4         5         6         7
#> 2  0.9179151
#> 3  0.5270440 0.4495898
#> 4  0.7805398 0.4474246 0.5756086
#> 5  0.6311777 0.5912622 0.2375642 0.8075274
#> 6  0.4209958 0.5038314 0.1377395 0.5101811 0.3537447
#> 7  0.7741790 0.3370293 0.2486464 0.6567099 0.2702296 0.3766480
#> 8  0.3870182 0.8498942 0.6319758 0.5351396 0.8331078 0.4944868 0.8563963
#> 9  0.2215408 0.8234515 0.3844753 0.8018674 0.4271127 0.3287066 0.6133956
#> 10 0.8630405 0.1759575 0.3476868 0.5663142 0.4356140 0.4432510 0.1685972
#>           8         9
```

```
#> 2
#> 3
#> 4
#> 5
#> 6
#> 7
#> 8
#> 9  0.5473781
#> 10 0.8722164 0.7319818
m$graph
#>    1 2 3 4 5 6 7 8 9
#> 2  0
#> 3  0 0
#> 4  0 1 0
#> 5  0 0 1 0
#> 6  0 0 1 1 0
#> 7  0 0 0 0 1 0
#> 8  1 0 0 0 0 1 0
#> 9  1 0 0 0 0 1 0 0
#> 10 0 1 0 0 0 0 1 0 0
```

Some users may wish to explot the filled mosaics for use in packages like *sf*, *sp*, and *raster*. The function *mhull* provides a polygon in simple features format, which *sf* understands natively:

```
p <- mhull(m)$polygon
p
#> [[1]]
#>               hx         hy
#>  [1,] 0.45654016 0.85474233
#>  [2,] 0.71492333 0.77560837
#>  [3,] 0.85899392 0.68803744
#>  [4,] 0.95918347 0.54338927
#>  [5,] 0.80158031 0.12464107
#>  [6,] 0.45713858 0.50099815
#>  [7,] 0.27283842 0.04214021
#>  [8,] 0.05884571 0.36461538
#>  [9,] 0.13623611 0.57219928
#> [10,] 0.45713858 0.50099815
#> [11,] 0.51706609 0.62501779
#> [12,] 0.45654016 0.85474233
#>
#> attr(,"class")
#> [1] "XY"      "POLYGON" "sfg"
```

This means you can use handy *sf* functions like *st_area* without any hassle. Actually, I don't recommend using *st_area* because the whole point of this package is to provide accurate area estimates, and it only creates polygons for illustrative purposes. But that's still an option.

Note that if *mgraph* breaks up the data into more than one subgraph, meaning disconnected networks, *mhull* will export a batch of polygons bound together as a list. Again, *sf* will understand immediately what this means, so you don't have to do anything in particular.

It's also easy to get the data into a format that *sp* will understand using the *sf* functions *st_sfc* and *as_Spatial*, as in:

p <- as_Spatial(st_sfc(p))

And you can export the data to a shapefile using the *raster* function *shapefile*:

shapefile(as_Spatial(st_sfc(p)),'mosaic.shp')

Plotting GBIF data

This section shows how to draw data for an individual species from the Global Biodiversity Information Facility, draw a map, and plot a mosaic on the map. Fortunately, the *rgbif* and *maps* libraries make this easy. To get rid of any obviously bogus coordinates, we'll use the *CoordinateCleaner* package.

```
library(CoordinateCleaner)
library(maps)
library(rgbif)
```

The geographic range of the humble yellow-spotted rock hyrax (*Heterohyrax brucei*) is a representative example of GBIF data. The *rgbif* function *occ_search* has lots of options, but most aren't important for a simple example. However, it's useful to throw out the records lacking coordinates, and museum collection data are more reliable (basisOfRecord='PRESERVED_SPECIMEN'). *occ_search* returns a big complicated object, of which only the third list item is needed. We'll convert that to a data frame. The actual coordinates can be backed out of it easily.

```
species <- 'Heterohyrax brucei'
o <- occ_search(scientificName=species,hasCoordinate=T,basisOfRecord='PRESERVED_SPECIMEN')
d <- data.frame(o[3])
lat <- d$data.decimalLatitude
long <- d$data.decimalLongitude
```

CoordinateCleaner has an option to remove zero values that's too aggressive, but the other ones are useful. It returns a matrix of true and false values (= passed or failed the test), so we'll mark any row with an F as a bad row. Finally, the lat and long values will be trimmed down by only taking the good ones.

```
x <- data.frame(long,lat,species)
cc <- clean_coordinates(x,lon='long',lat='lat',species='species',
 tests=c("capitals","centroids","gbif","institutions"))
#> Testing coordinate validity
#> Flagged 0 records.
#> Testing country capitals
#> Flagged 5 records.
#> Testing country centroids
#> Flagged 0 records.
#> Testing GBIF headquarters, flagging records around Copenhagen
#> Flagged 0 records.
#> Testing biodiversity institutions
#> Flagged 0 records.
#> Flagged 5 of 124 records, EQ = 0.04.

bad <- array(dim=length(lat),data=0)
for (j in 1:nrow(cc))
    if (sum(cc[j,] == F) > 0)   {
        bad[j] <- 1
```

```
    }
lat <- lat[bad == 0]
long <- long[bad == 0]
```

At this point, plotting the map becomes trivial. The Albers projection is a pretty good one. Most of the *map* options are self-explanatory. It helps to set zero margins so the map takes up more room. lforce='s' is used to restrict the part of the map that is plotted; it's not always needed.

```
par(mar=c(0,0,0,0))
map(project='albers',parameters=c(0,0),xlim=c(-17,57),ylim=c(-30,30),fill=T,col='gray90',
 border=NA,lforce='s')
points(long,lat,cex=0.4,pch=21,col='black',bg='white')
```

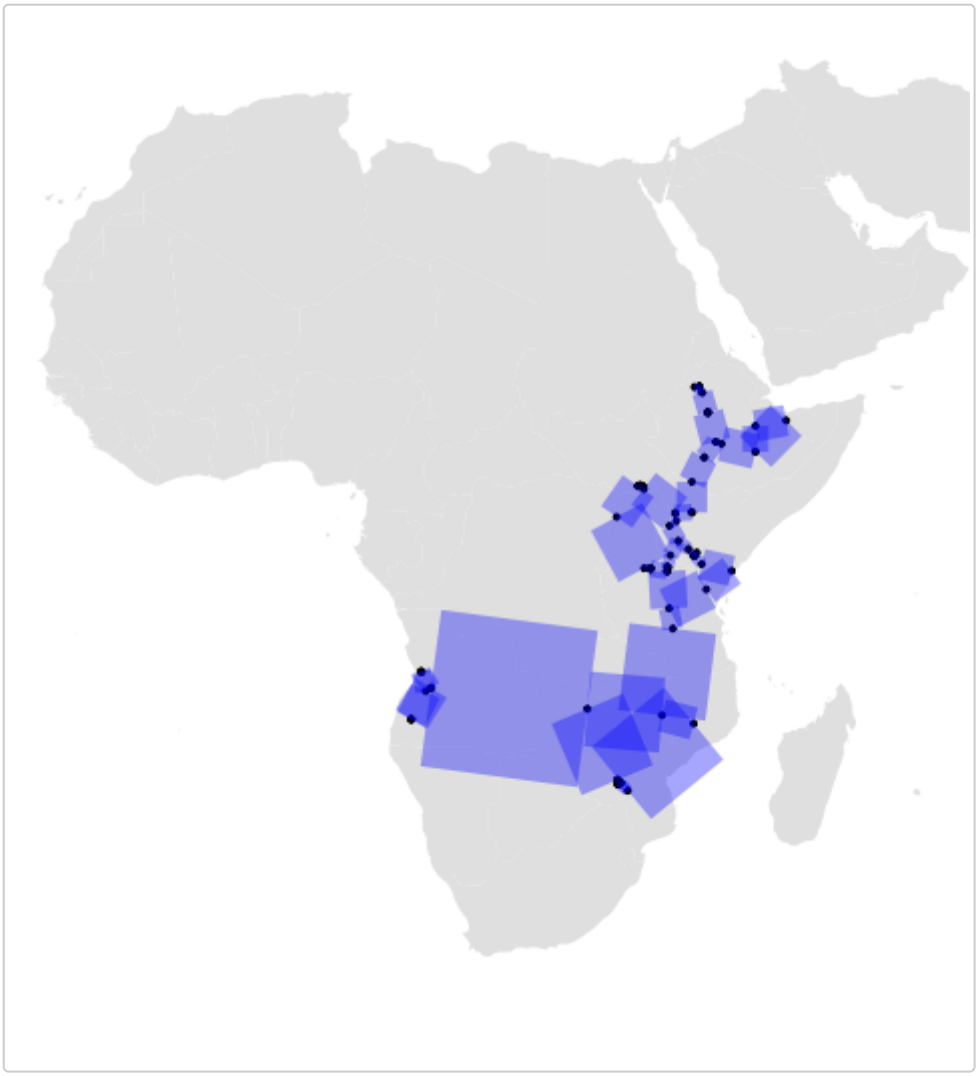
Now it's easy to add a filled mosaic. For ease of reading, first get the graph object and then plot it. longlat = T tells *mgraph* that distances between points should be great circle distances, not Euclidean distances. h = 0.65 sets the color to a purplish blue; alpha = 0.35 makes the filled mosaic mostly transparent.

```
par(mar=c(0,0,0,0))
map(project='albers',parameters=c(0,0),xlim=c(-17,57),ylim=c(-30,30),fill=T,col='gray90',
 border=NA,lforce='s')
m <- mgraph(long,lat,longlat=T)
plot(m,cex=0.4,h=0.65,s=1,v=1,alpha=0.35,fill=T,outline=T,lines=F,add=T)
```

Note that the mosaic includes several thin lines. These are bridges between mosaic pieces, and they are a totally normal thing to see when a data set is sparse.

Finally, another good way to visualize the data is to delete the fill and add squares that follow the edges.

```
par(mar=c(0,0,0,0))
map(project='albers',parameters=c(0,0),xlim=c(-17,57),ylim=c(-30,30),fill=T,col='gray90',
 border=NA,lforce='s')
plot(m,cex=0.4,h=0.65,s=1,v=1,alpha=0.35,lines=F,squares=T,add=T)
```



What about the area estimates, which are the whole point of the exercise? The mosaic and MST values are similar, but certainly not identical. The values are in square degrees.

```
mgraph(long,lat,longlat=T)$area
#> [1] 286.9118
tgraph(long,lat,longlat=T)$area
#> [1] 200.0254
```

Simulation analysis

Mosaic areas have three main advantages: they are accurate, insensitive to variation in sample size, and able to handle complex shapes. In this section, we'll contrast mosaic and MST areas with areas based on convex hulls (= minimum convex polygons) and on 95% kernel density estimates (KDEs).

We'll skip hypervolumes because the computations are very slow. But for the record, you can compute hypervolumes easily by loading the library *hypervolume* and calling *get_volume(hypervolume_gaussian(pts))*.

Note that the areas will be of circles (the default shape used by *ptpattern*).

To start with, we'll need the *sf* library to calculate the hull areas and *MASS* to get KDEs:

```
library(MASS)
library(sf)
```

```
#> Linking to GEOS 3.7.2, GDAL 2.4.2, PROJ 5.2.0
```

Differences among methods are pretty visible at sample sizes of around 10, and only about 1000 trials are needed to see fine-scale differences. Start by setting these parameters and arrays:

```
points <- 10
trials <- 1000
hull_area <- array(dim=trials,data=NA)
kde_area <- array(dim=trials,data=NA)
mosaic_area <- array(dim=trials,data=NA)
mst_area <- array(dim=trials,data=NA)
```

The D array stores random shape diameters that fall over a broad, log-normally distributed range, with a mean of one linear unit (zero log units) and a standard deviation of 0.5. The true area values follow directly from the diameters, although they're also computed by the *ptpattern* function. Note if *r* is the radius and *D* is the diameter, pi *r*^2 is the same as pi *D*^2/4.

```
D <- exp(rnorm(trials,0,0.5))
true_area <- array(dim=trials,data=pi * D^2 / 4)
```

Getting the data in shape for the *sf* area calculation is a little longwinded. The base function *chull* is used to find the points that form the convex hull and put them in order; the first point is added to the end of the matrix so the hull will wrap around completely; *st_polygon* and *st_sfc* are used to prep the data; and *st_area* is only used at the very end. Here's an example:

```
p <- ptpattern(points=points,diameter=D[1])
pts <- cbind(p$x,p$y)

ch <- pts[chull(pts),]
ch <- rbind(ch,ch[1,])
p <- st_sfc(st_polygon(list(ch)))
st_area(p)
#> [1] 0.2591643
```

Meanwhile, the KDE calculations are a little complicated. First we need to get the mean and range of the data across each axis. The reason is that the *kde2d* function defaults to only calculating densities over the narrowest possible range. Note that *kde2d* spits out these values for a grid of cells.

```
mx <- mean(pts[,1])
my <- mean(pts[,2])
rx <- 10 * diff(range(pts[,1]))
ry <- 10 * diff(range(pts[,2]))
```

In the following lines, the densities are obtained over a wide range using *kde2d*; the area of each grid cell used by *kde2d* is calculated; the densities are sorted, standardized so they add up to 1, and cumulated; the cells making up the top 95% of the distribution are counted; and the count is scaled up with the area value ("cell").

```
k <- kde2d(pts[,1],pts[,2],lims=c(mx - rx,mx + rx,my - ry,my + ry))
cell <- diff(k$x)[1] * diff(k$y)[1]
z <- sort(k$z,decreasing=T)
```

```
z <- z / sum(z)
z <- cumsum(z)
(sum(z < 0.95) + 1) * cell
#> [1] 1.019301
```

By contrast, *mgraph* and *tgraph* do all the work for you:

```
mgraph(pts)$area
#> [1] 0.6112228
tgraph(pts)$area
#> [1] 0.8694804
```

Finally, we're ready to loop through.

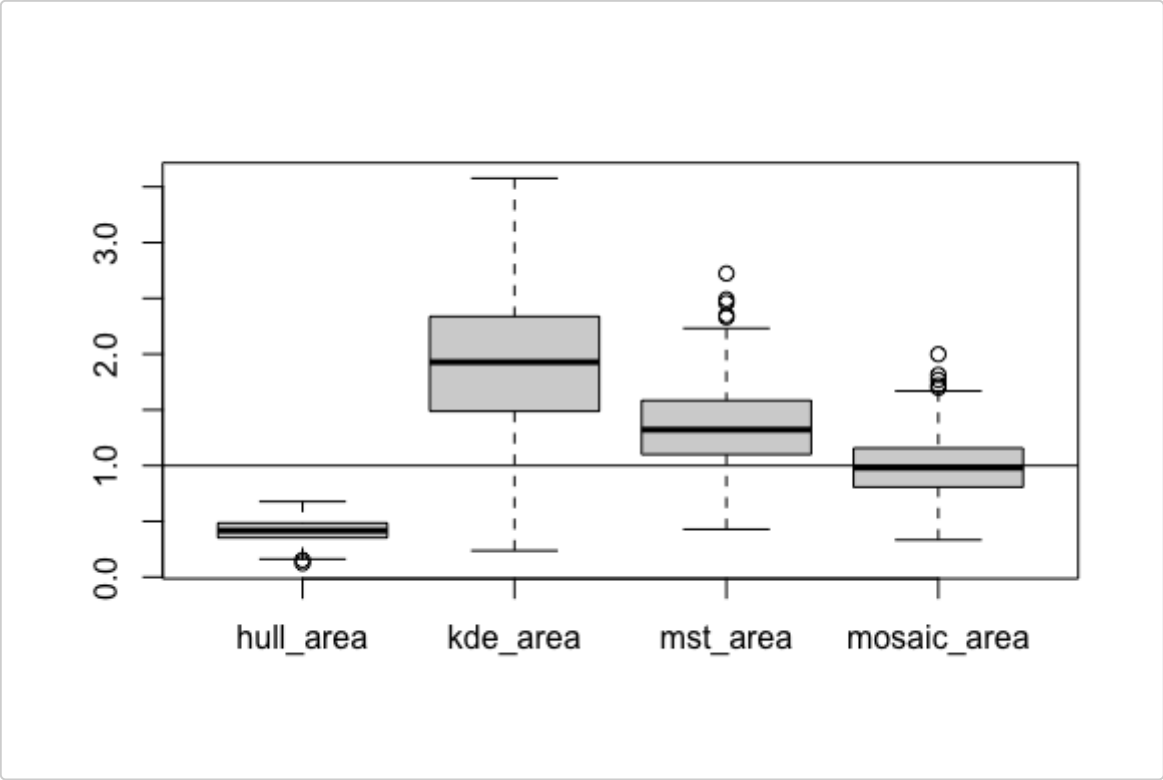
```
for (i in 1:trials) {
    p <- ptpattern(points=points,diameter=D[i])
    pts <- cbind(p$x,p$y)

    ch <- pts[chull(pts),]
    ch <- rbind(ch,ch[1,])
    p <- st_sfc(st_polygon(list(ch)))
    hull_area[i] <- st_area(p)

    mx <- mean(pts[,1])
    my <- mean(pts[,2])
    rx <- 10 * diff(range(pts[,1]))
    ry <- 10 * diff(range(pts[,2]))

    k <- kde2d(pts[,1],pts[,2],lims=c(mx - rx,mx + rx,my - ry,my + ry))
    cell <- diff(k$x)[1] * diff(k$y)[1]
    z <- sort(k$z,decreasing=T)
    z <- z / sum(z)
    z <- cumsum(z)
    kde_area[i] <- (sum(z < 0.95) + 1) * cell

    mosaic_area[i] <- mgraph(pts)$area
    mst_area[i] <- tgraph(pts)$area
}
```

The following two lines spit out the geometric mean of the ratio of estimated to true area. These simple statistics make the point pretty clearly.

```
d <- data.frame(hull_area,kde_area,mst_area,mosaic_area) / true_area
exp(colMeans(log(d)))
#>   hull_area    kde_area    mst_area mosaic_area
#>   0.4046323   1.8092980   1.3065140   0.9544581
```

There are lots of ways to illustrate the differences in accuracy and precision between the methods, such as a simple box plot:

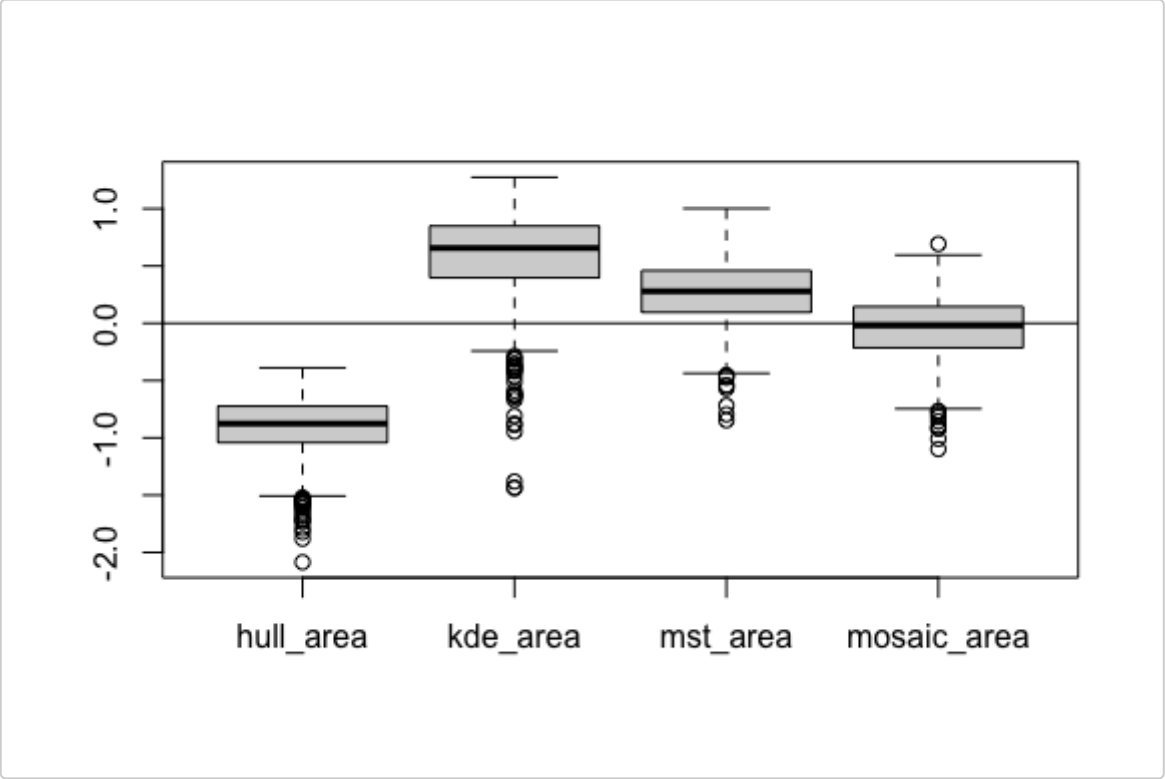
```
boxplot(d)
abline(1,0)
```



Note that an accurate method would center the data on a value of 1 (meaning the ratio of the estimated to true area is 1). Mosaic and MST areas do this; the other two methods don't.

This plot is deceptive because the low hull area values are scrunched together. Log transforming the data makes more sense. Now 0 = perfectly accurate:

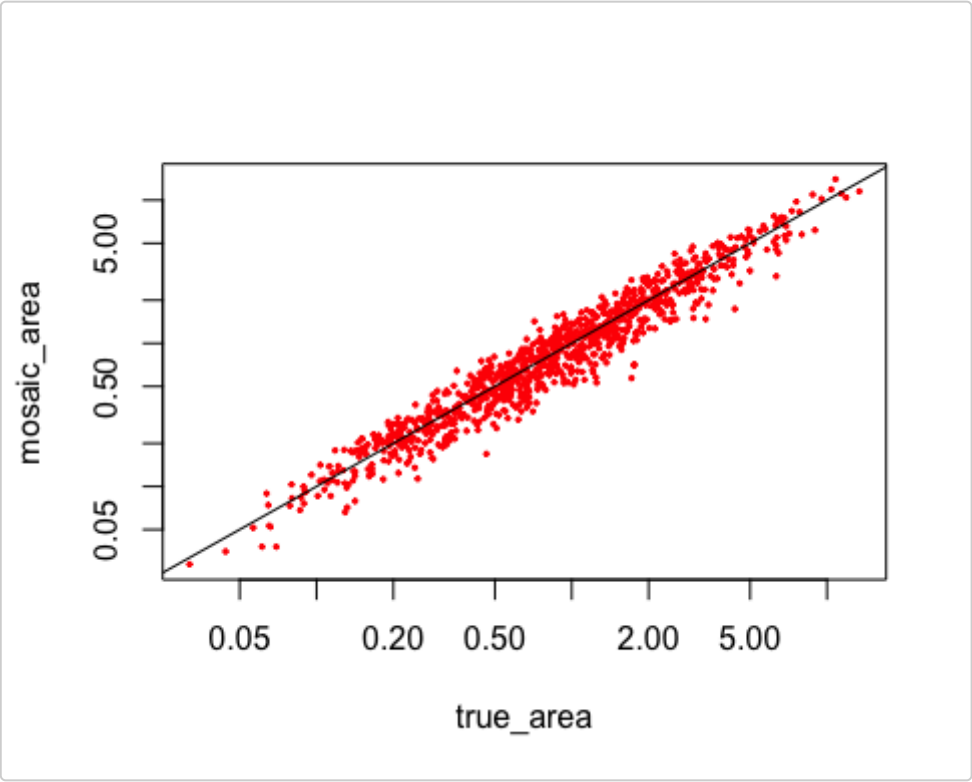
```
boxplot(log(d))
abline(0,0)
```



You can see that differences in precision (the size of the boxes) are small, but convex hulls are a bit noisier. Accuracy is the real issue.

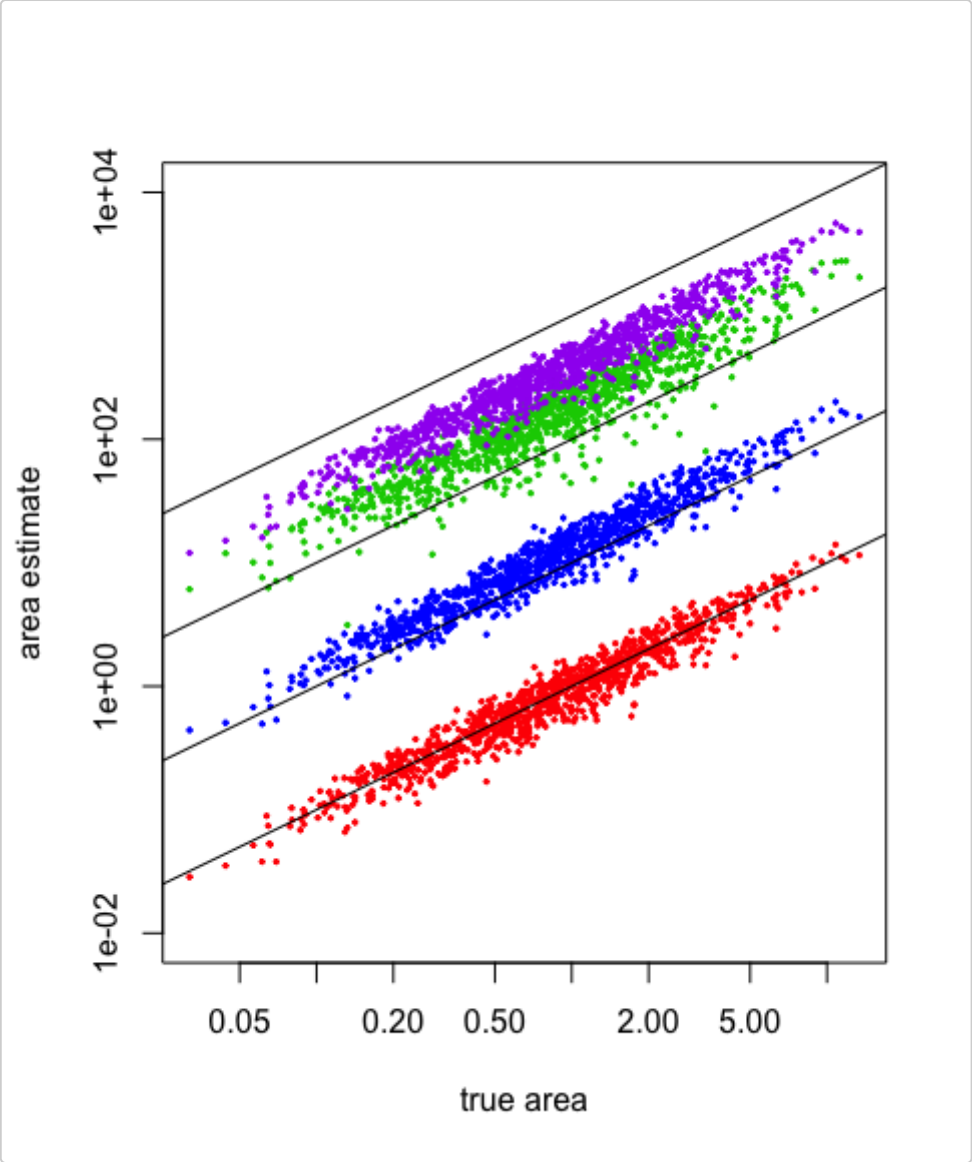
A scatter plot makes the high accuracy of mosaic areas easier to grasp:

```
plot(true_area,mosaic_area,pch=19,cex=0.3,col='red',log='xy')
lines(c(0.01,100),c(0.01,100))
```



Staggering the data for each method by a factor of, say, 10 helps to compare them:

```
plot(true_area,mosaic_area,pch=19,cex=0.3,col='red',log='xy',ylim=c(0.01,10000),xlab='true
        area',ylab='area estimate')
points(true_area,10 * mst_area,pch=19,cex=0.3,col='blue')
points(true_area,100 * kde_area,pch=19,cex=0.3,col='green3')
points(true_area,1000 * hull_area,pch=19,cex=0.3,col='purple')
lines(c(0.01,100),c(0.01,100))
lines(c(0.01,100),10 * c(0.01,100))
lines(c(0.01,100),100 * c(0.01,100))
lines(c(0.01,100),1000 * c(0.01,100))
```



The lines are lines of unity, not regression lines, so not falling on them is a real problem. Specifically, the KDE values (green) are so high and the convex hull values (purple) are so low that they overlap heavily, despite the offsetting!