Making maps in R

By Nick Eubank, building off excellent tutorials by Claudia Engel

This tutorial focuses on the two main tools for looking at mapping Spatial* objects – the basic `plot` command, and the more refined plot command `spplot` from the `sp` library. Note there are also ways to use tools like `ggplot` and `ggmap` (designed to make working with maps in `ggplot` easier),, but in this workshop we'll focus on `plot` and `spplot`.

Although there are other tools available for more sophisticated applications, most plotting situations can be handled through the use of two functions:

- `plot` : plot shapes associated with Spatial* or Raster objects
- `spplot` : plot shapes associated with Spatial* objects AND color them based on attributes in a associated DataFrame.

1. plot

The syntax for each should be relatively familiar to anyone used to working with `plot` in other settings – just pass the sp object to plot!

```
palo_alto <- readOGR("RGIS3_Data/palo_alto", "palo_alto")
```

```
OGR data source with driver: ESRI Shapefile
Source: "RGIS3_Data/palo_alto", layer: "palo_alto"
with 371 features and 5 fields
Feature type: wkbPolygon with 2 dimensions
```

```
plot(palo_alto)
```

It's also easy to add some basic options using standard `plot` modifiers:

```
plot(palo_alto, border = "red")
title(main = "Palo Alto", sub = "By Census Tracts")
```

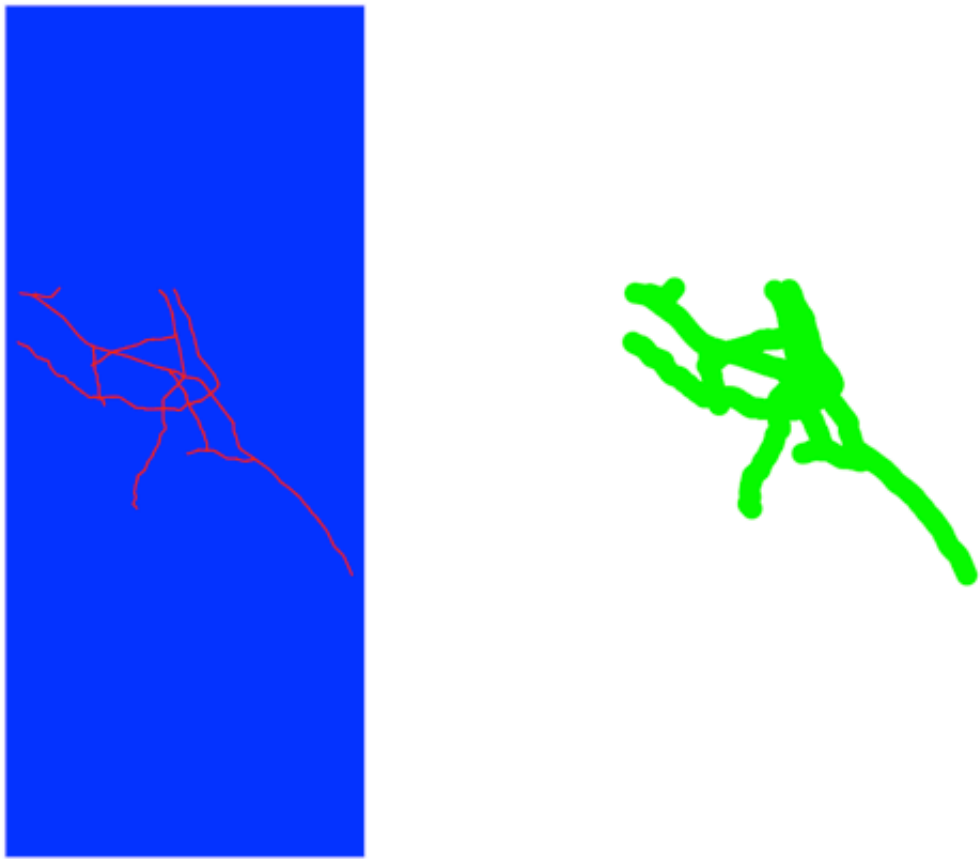
By Census Tracts

When plotting lines, you can also use lots of the basic `plot` options for lines. For example, the `lwd` option will determine the width of lines, and `col` their color.

```
freeways <- readOGR("RGIS3_Data/palo_alto", "palo_alto_freeways")
```

```
OGR data source with driver: ESRI Shapefile
Source: "RGIS3_Data/palo_alto", layer: "palo_alto_freeways"
with 955 features and 45 fields
Feature type: wkbLineString with 2 dimensions
```

```
par(mfrow = c(1, 2))
plot(freeways, col = "red", bg = "blue")
plot(freeways, lwd = 10, col = "green")
```



Points work similarly, with `pch` determining shape and `cex` determining size. You can find a table of symbol-to-number mappings here (http://www.statmethods.net/advgraphs/parameters.html)

1.1 Multiple layers with `plot`

It's also easy to plot multiple layers using `plot` with the `add` option:

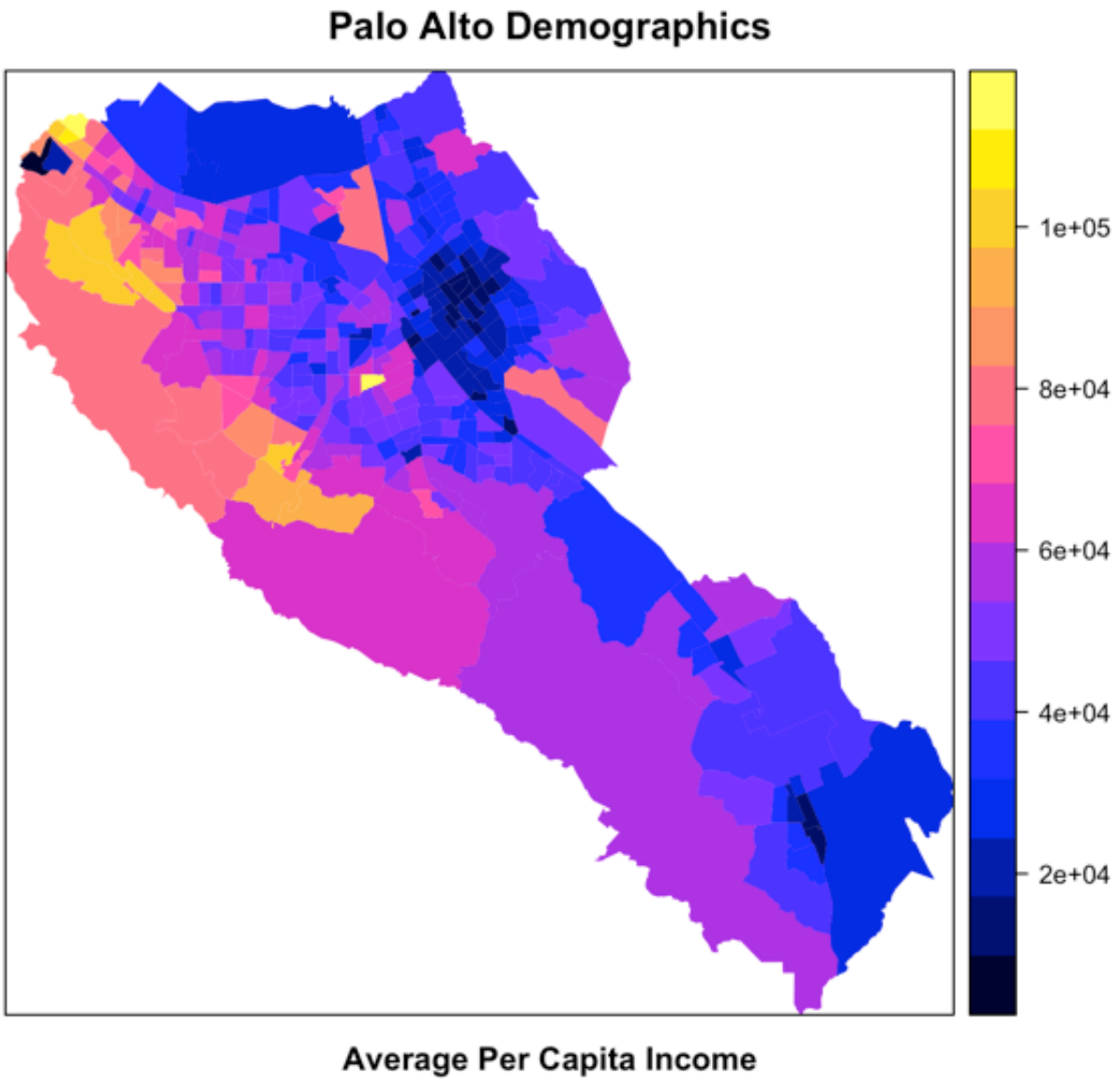
```
stopifnot(proj4string(palo_alto) == proj4string(freeways))  # Check in same projec
tion before combining!

plot(palo_alto)
plot(freeways, col = "blue", add = T)
```

2. spplot

`spplot` is an extension of `plot` specifically for making maps of Spatial* objects. In particular, it's very useful for filling in polygon colors based on attributes in an associated DataFrame (what are called "chloropleth" maps). Just pass an Spatial*DataFrame object and the name of columns you want to plot (if you don't pass specific column names, a separate figure will be created for each column.)

Note the `col="transparent"` option just supresses the plotting of polygon borders for a slightly better ascetic.

```
spplot(palo_alto, "PrCpInc", main = "Palo Alto Demographics", sub = "Average Per C
apita Income",
    col = "transparent")
```



A big list of example graphs with associated code can be found here (http://rspatial.r-forge.r-project.org/gallery/)

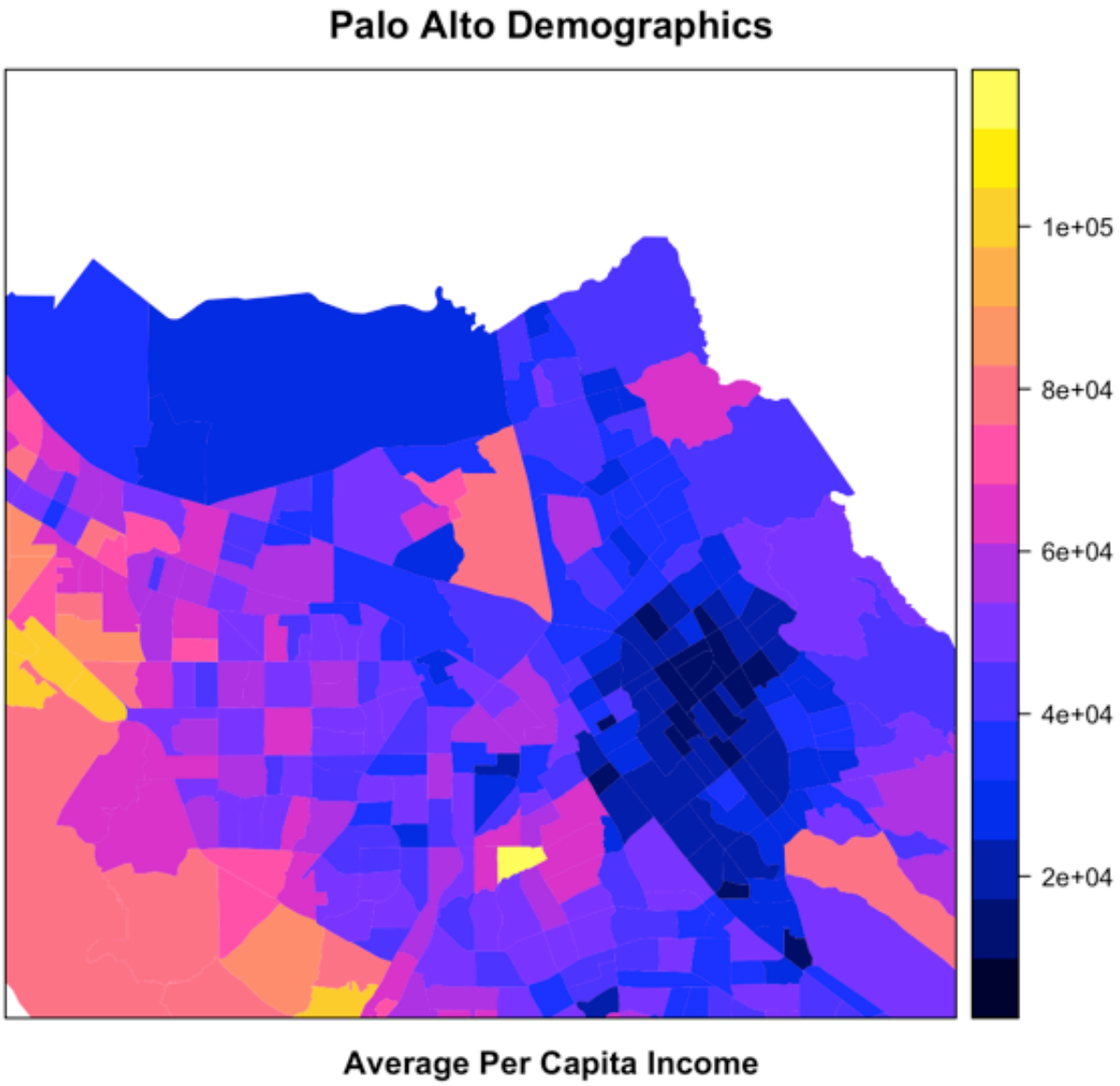
Another guide is here (https://sites.google.com/site/spatialr/plottingmaps)

2.2 Controlling Extent

By default, `spplot` will zoom to the extent of the Spatial* object being plotted. However, this can be overridden by setting the `xlim` and `ylim` parameters, which determine the edges of the plot.

However, these can be a little hard to work with. With that in mind, the following code allows the user to modify the zoom and shift the center of the plot with three more intuitive parameters:

```
# Change these parameters
scale.parameter = 0.5  # scaling paramter. less than 1 is zooming in, more than 1
zooming out.
xshift = -0.1  # Shift to right in map units.
yshift = 0.2  # Shift to left in map units.
original.bbox = palo_alto@bbox  # Pass bbox of your Spatial* Object.

# Just copy-paste the following
edges = original.bbox

edges[1, ] <- (edges[1, ] - mean(edges[1, ])) * scale.parameter + mean(edges[1,
    ]) + xshift
edges[2, ] <- (edges[2, ] - mean(edges[2, ])) * scale.parameter + mean(edges[2,
    ]) + yshift


# In `spplot`, set xlim to edges[1,] and ylim to edges[2,]
spplot(palo_alto, "PrCpInc", main = "Palo Alto Demographics", sub = "Average Per C
apita Income",
    col = "transparent", xlim = edges[1, ], ylim = edges[2, ])
```

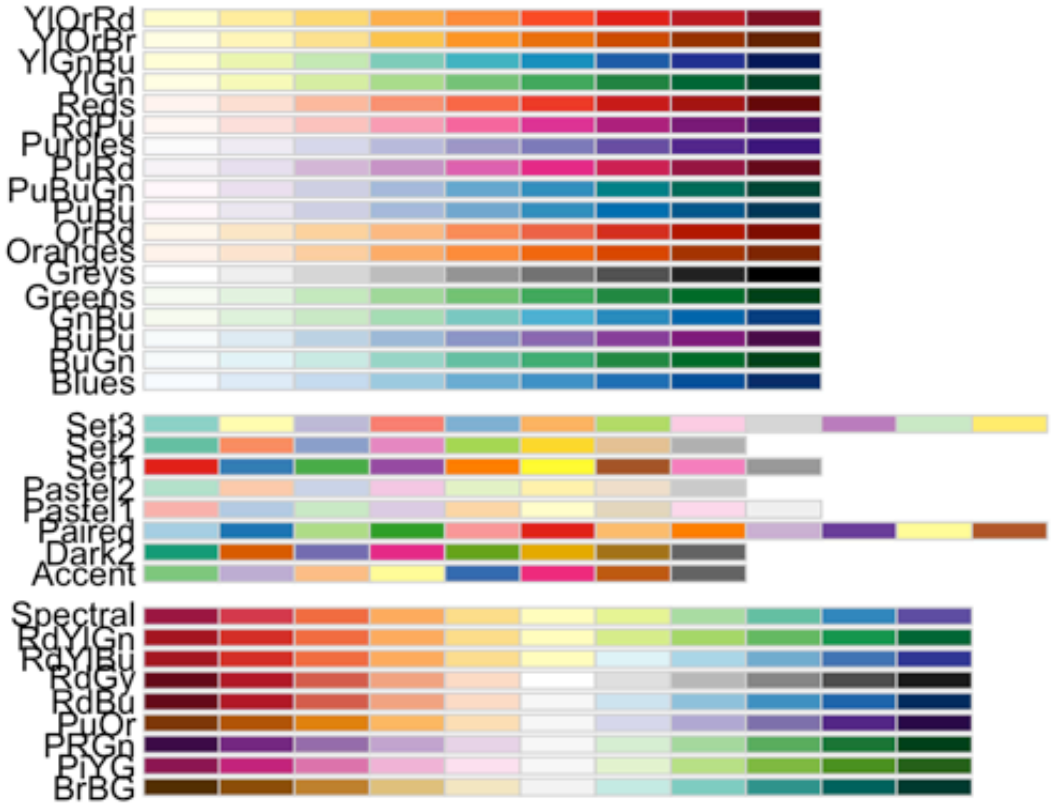


2.2 Controlling Colors

Custom Palettes for `spplot`

If you don't like the default color scheme, you can use your own with ColorBrewer, which you can install with `install.packages("RColorBrewer")`. Once loaded, you can see a list of all the color pallets that come with `RColorBrewer` with the command:

```
library(RColorBrewer)
display.brewer.all()
```

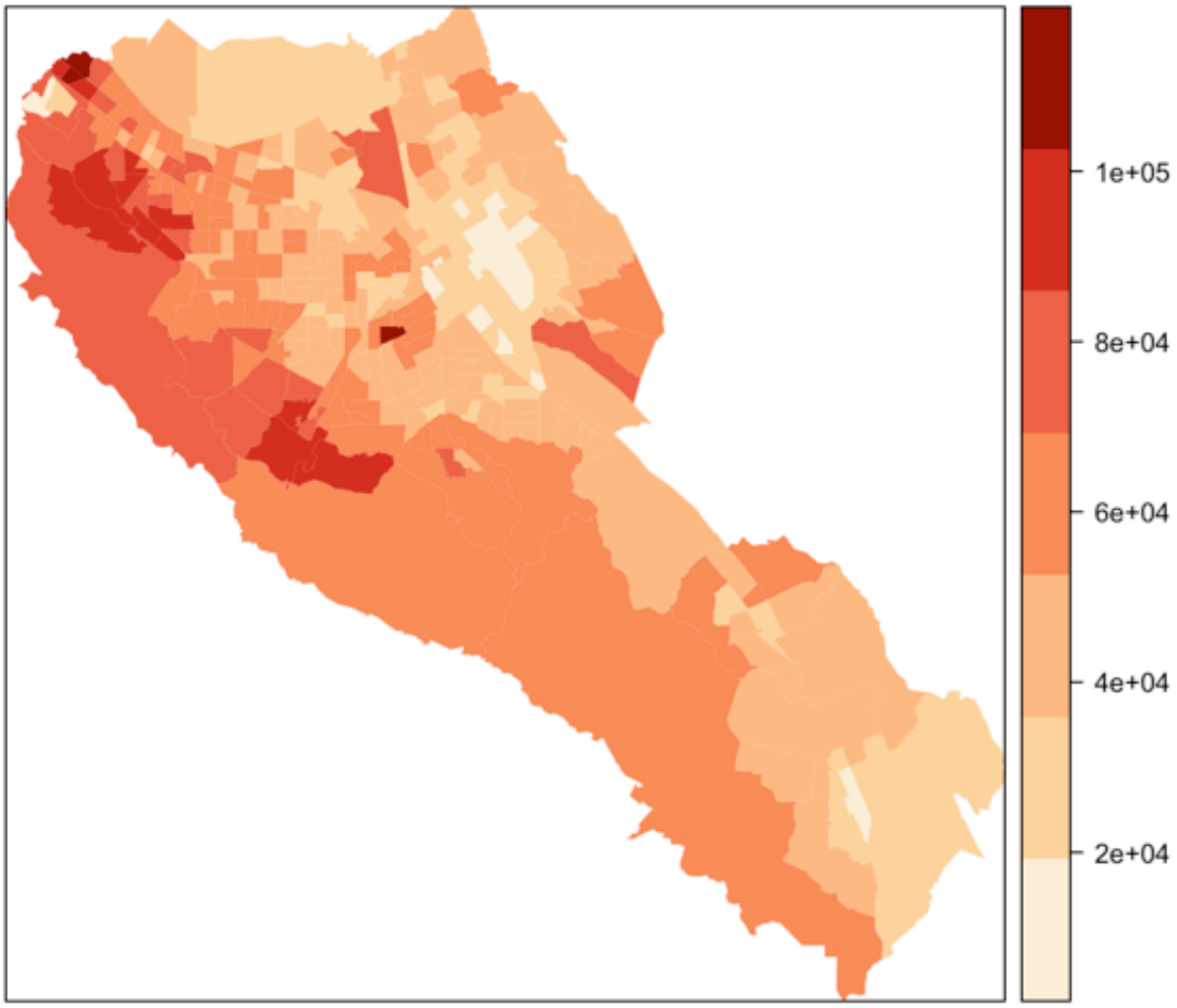


Once you've picked a palette you like, create a palette object as follows, where `n` is the number of cuts you want to use, and `name` is the name of the color ramp. Note that different palettes have different limits on the maximium and minimium number of cuts allowed.

```
my.palette <- brewer.pal(n = 7, name = "OrRd")
```

Then you can just pass this pallet to `spplot`, making sure to set the number of cuts to **1 minus the number of colors**.

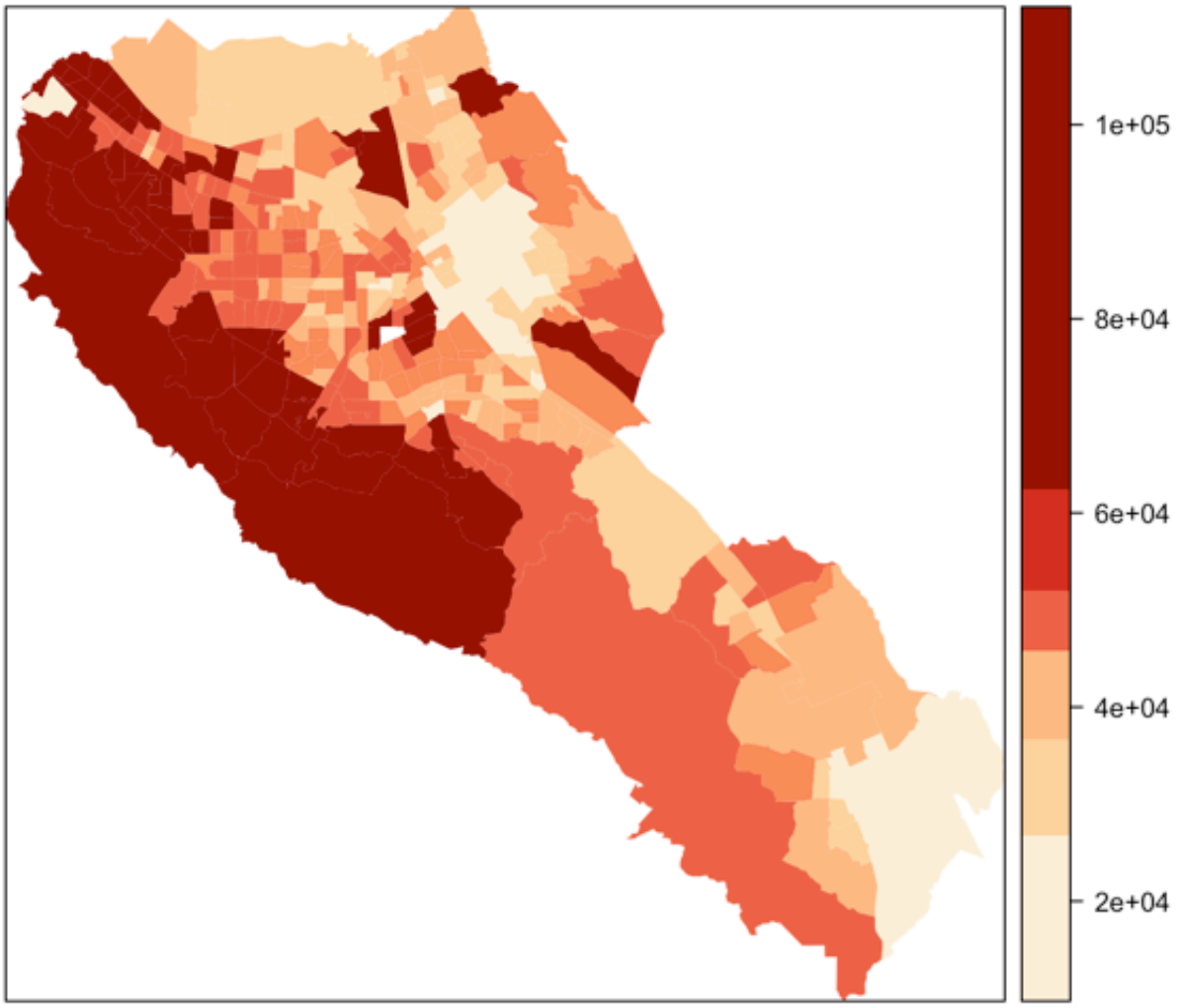
```
spplot(palo_alto, "PrCpInc", col.regions = my.palette, cuts = 6, col = "transparen
t")
```



Controlling Color Breaks

If you don't want evenly spaced cutoffs between colors, you can use the `classInt` library to make custom cuts.

```
library(classInt)

breaks.qt <- classIntervals(palo_alto$PrCpInc, n = 6, style = "quantile", interval
Closure = "right")

spplot(palo_alto, "PrCpInc", col = "transparent", col.regions = my.palette,
    at = breaks.qt$brks)
```



Exercise 1

1. Pick a city to plot. Three cities are included in RGIS3_Data, or find your own.
2. Read-in the city polygons.
3. Use `spplot` to plot average incomes for your city with default colors.
4. Now use RColorBrewer to map income using a custom color scheme.
5. Use the `classInt` library to cut income by quantiles rather than even intervals.

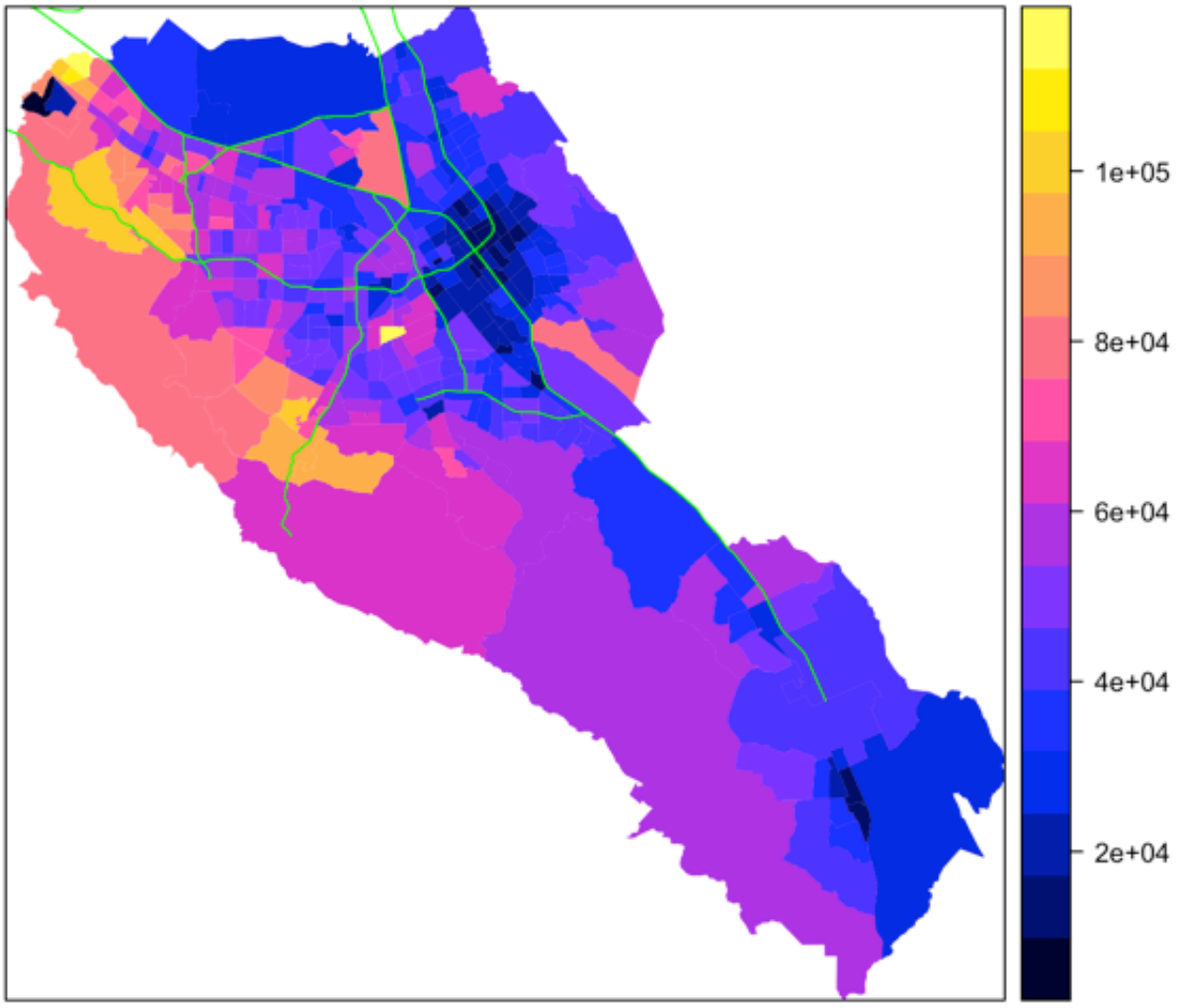
2.3 Multiple layers with `spplot`

`spplot` allows users to add multiple layers or symbols using the `sp.layout` argument. To use `sp.layout`, you first create a new list where:

- the first item is the type of layer to add,
- the second argument is the actual object to plot,
- and any following items are plotting options.

You then pass this list to the `sp.layout` argument. For example, if I wanted to overlay the buffered grant locations on electoral districts, I would use the following codes:

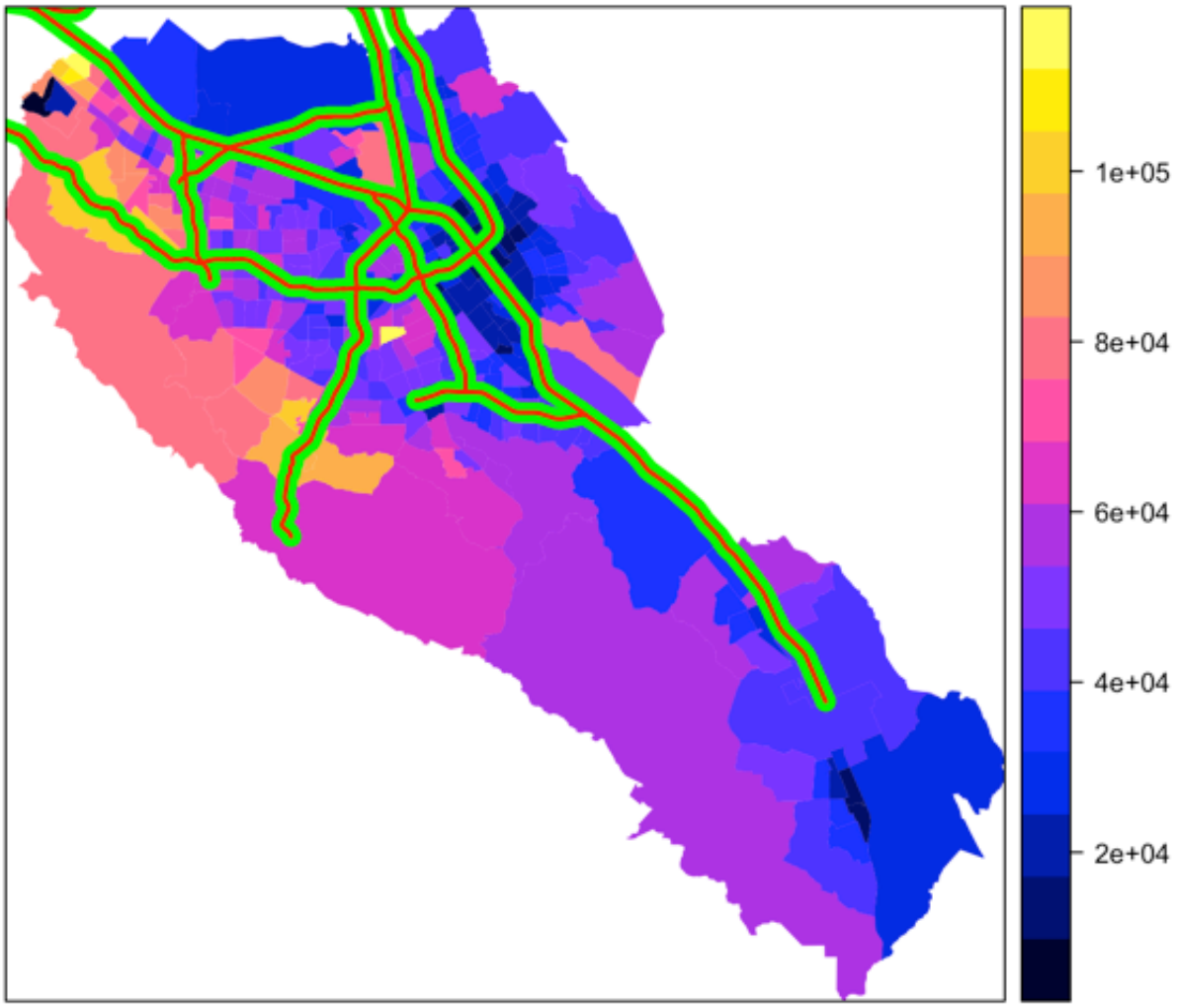
```
# Create a layer-list
freeway.layer <- list("sp.lines", freeways, col = "green")

# Plot with layer-list passed to `sp.layout`
spplot(palo_alto, "PrCpInc", sp.layout = freeway.layer, col = "transparent")
```



If you want to add multiple layers, just combine the layer-lists as follows. Note that order of items in the list will determine plotting order! Here's an example where I plot freeways twice so I can get a

different color scheme.

```
# Create a layer-list
freeway.layer <- list("sp.lines", freeways, col = "green", lwd = 10)
freeway2.layer <- list("sp.lines", freeways, col = "red", lwd = 2)


# Plot with layer-list passed to `sp.layout`
spplot(palo_alto, "PrCpInc", sp.layout = list(freeway.layer, freeway2.layer),
    col = "transparent")
```



Accoutrements

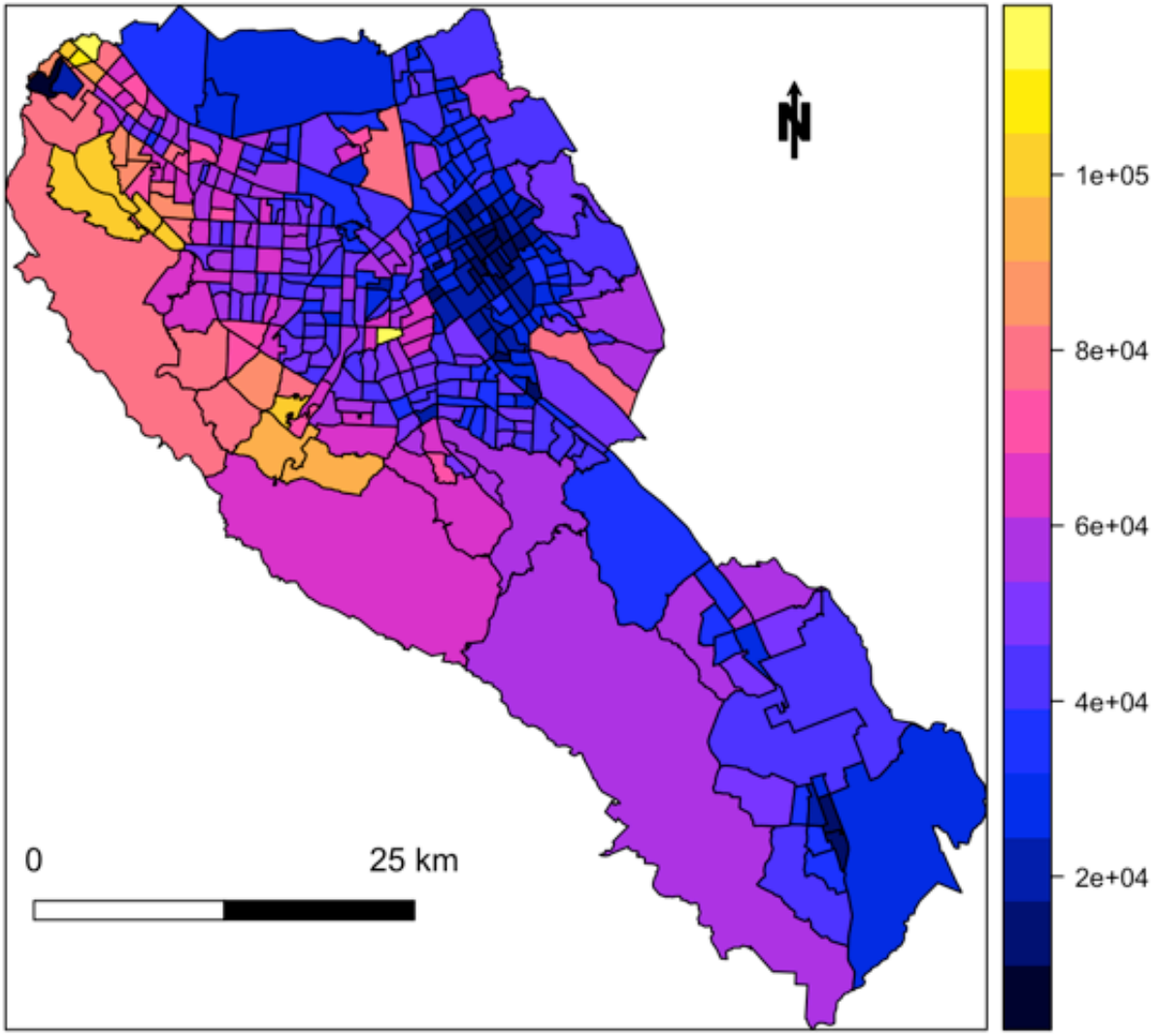
You can also use the `sp.layout` option to add other things, like compass arrows or scales. In most cases, the key to this is to set "offset", which defines the location of the bottom left hand corner of what you're working with. Getting offsets right will kinda drive you crazy.

Here's a small handfull:

```
palo_alto_proj <- spTransform(palo_alto, CRS("+init=EPSG:32611"))  # Can't make a scale if not projected!
palo_alto_proj@bbox  # Check dimensions to help guide offset choices
```

```
         min       max
x   39130.08  103617.3
y 4092787.57 4160027.3
```

```
scale = list("SpatialPolygonsRescale", layout.scale.bar(), scale = 25000, fill = c("transparent",
    "black"), offset = c(41000, 4100000))

# The scale argument sets length of bar in map units
text1 = list("sp.text", c(41000, 4104000), "0")
text2 = list("sp.text", c(66000, 4104000), "25 km")

arrow = list("SpatialPolygonsRescale", layout.north.arrow(), offset = c(90000,
    4150000), scale = 5000)


spplot(palo_alto_proj, "PrCpInc", sp.layout = list(scale, text1, text2, arrow))
```



Exercise 2

1. Using the same city data you used for Exercise 1, overlay the freeway shapefile over your city. Make the freeways red.
2. Add a compass somewhere reasonable on the map.
3. BONUS: Add a scale bar.

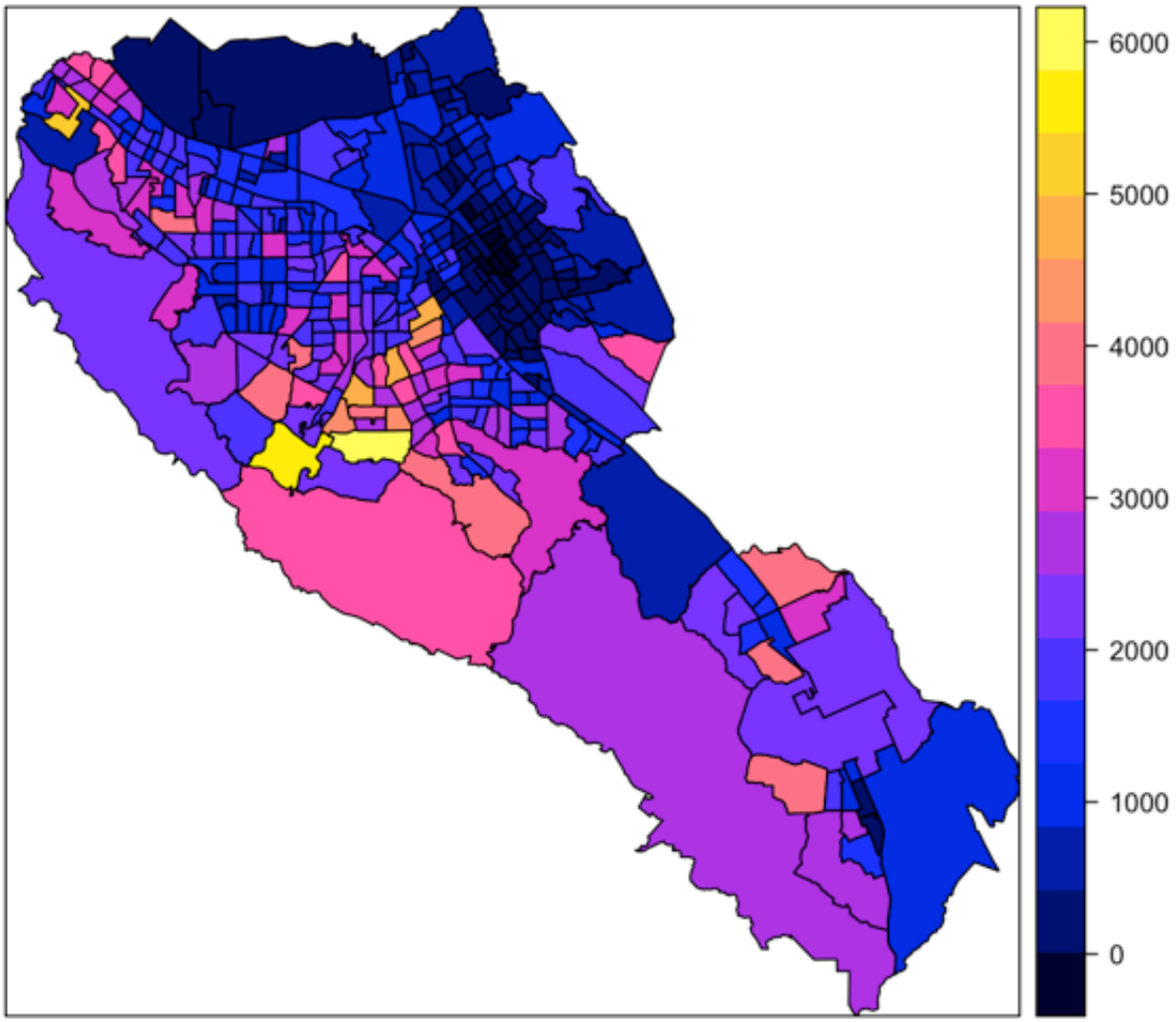
3. Dot-Density Plots

Dot-Density plots are a favorite these days, and they can be made relatively easily with the use of `spplot` and the `dotsInPolys` tool from the `maptools` library. Not much to explain here, so here's just an example of making a dot-density plot for Santa Clara county from census polygons!

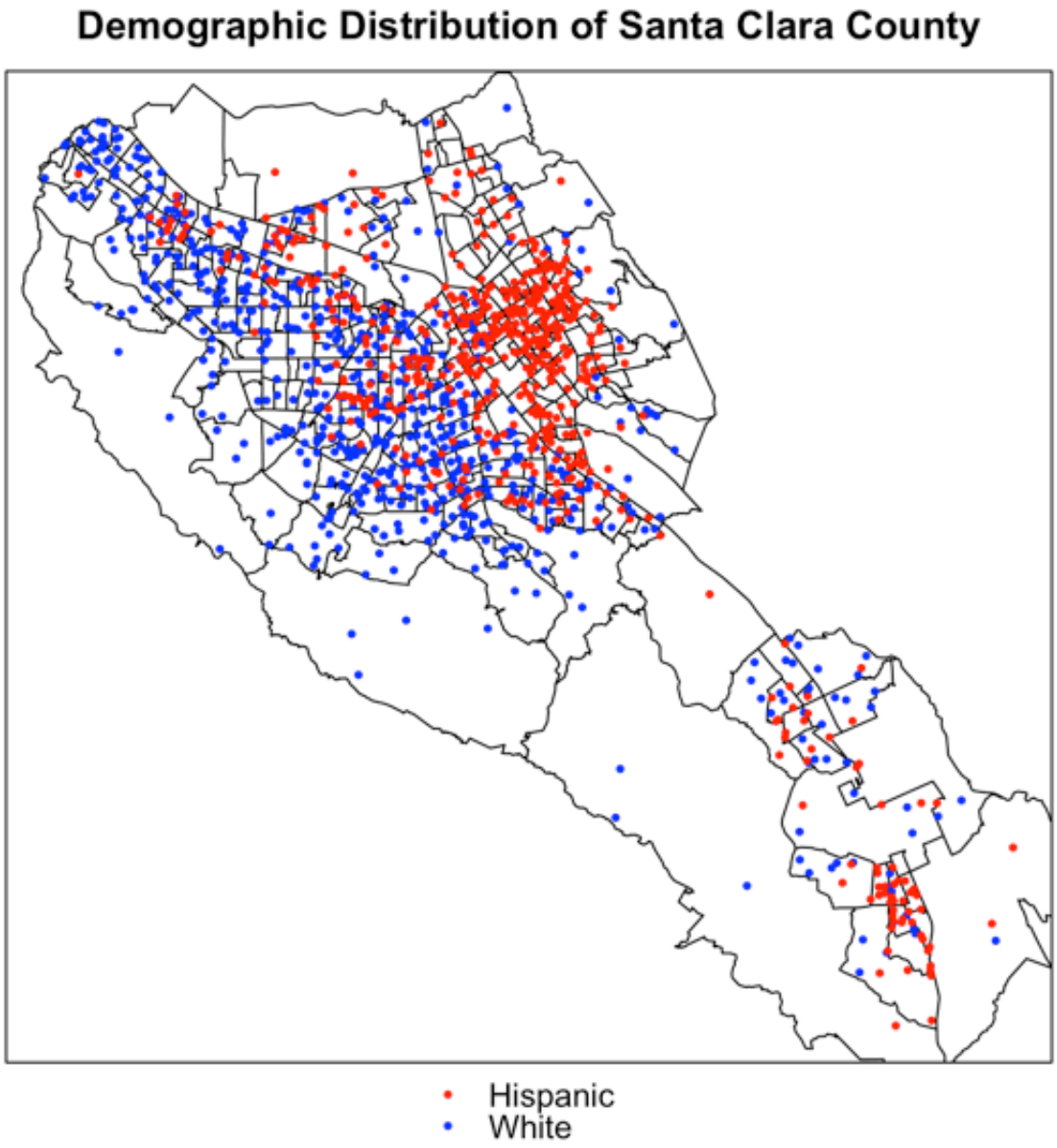
```
library(maptools)

# Get census polygons
census <- readOGR("RGIS3_data/palo_alto", "palo_alto")
```

```
OGR data source with driver: ESRI Shapefile
Source: "RGIS3_data/palo_alto", layer: "palo_alto"
with 371 features
It has 5 fields
```

```
# Get feel for data with white population (largest group)
spplot(census, "White")
```



```
# Setting a seed is a good idea -- since points are random, it's helpful for
# replication to make sure this code will always make the same result.
set.seed(47)

# Create a fixed number of points at random locations within each polygon
# based on a polygon variable.  Since the field values here are the number
# of people, we can get one dot per 300 people as follows:
people.per.dot = 700

dots.w <- dotsInPolys(census, as.integer(census$White/people.per.dot))
dots.w$ethnicity <- "White"

dots.h <- dotsInPolys(census, as.integer(census$hispanc/people.per.dot))
dots.h$ethnicity <- "Hispanic"

# Gather all the dots into a single SpatialPoints
dots.all <- rbind(dots.w, dots.h)
proj4string(dots.all) <- CRS(proj4string(palo_alto))

# Since ethnicity is a string, order is alphabetical. You can change if you
# want by making these categoricals!
my.palette <- c("red", "blue")
point.size <- 0.5

# Make sure stored as a factor
dots.all$ethnicity <- as.factor(dots.all$ethnicity)


# Make sp.layout list for the actually boundaries
census.tract.layer <- list("sp.polygons", census)

spplot(dots.all, "ethnicity", sp.layout = census.tract.layer, col.regions = my.pal
ette,
    cex = point.size, main = "Demographic Distribution of Santa Clara County")
```



Exercise 3

1. Make a dot-density plot for your own city!
2. Change the number of people per dot and the size of dots.
3. Overlay your freeway data. How do freeways affect segregation?

4. Basemaps

Both `plot` and `spplot` can suppot overlaying Spatial* objects over basemaps, with differing degrees of difficulty.

Before getting into specifics, one very important caveat: basemaps are generally just rasters, but not all functions for adding basemaps to your maps will include information on the projection of the basemap raster.

Warning: Almost all basemaps you will get from common sources (like google maps) use a very special projection – WGS84 Web Mercator (EPSG:3857). If you use a tool like `ggmap`, be very careful with this, as the maps it generates are in latitudes and longitudes (which makes you think it's regular WGS84), but if you treat the data as WGS84, you will have serious alignment issues. (http://rstudio-pubs-static.s3.amazonaws.com/16660_7d1ab1b355344578bbacb0747fd485c8.html) ALl tools presented here will address that issue.

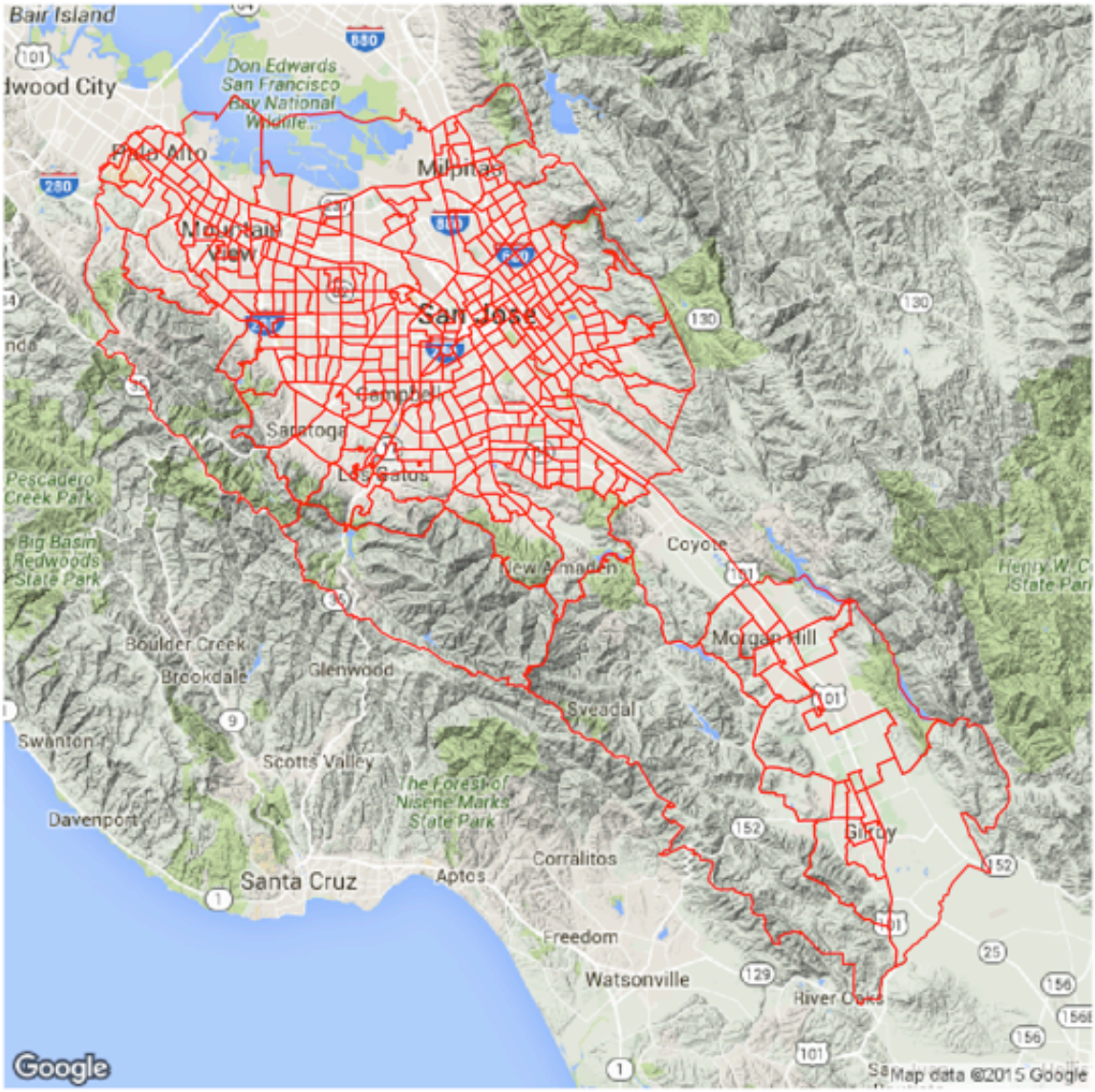
4.1 Basemaps with plot

If you don't need to color your plots according to an attribute of your Spatial* data, the easiest way to plot basemaps is with the `gmap` command from the `dismo` library. `gmap` returns a google map, and you can modify the `type` argument to get different types of google maps (it will accept 'roadmap', 'satellite', 'hybrid', 'terrain').

Note that unlike the somewhat more popular `get_map` function in the `ggmap` library, the `gmap` function in `dismo` creates a RasterLayer complete with correct projection information!

```
library(dismo)
library(raster)

# Pass object whose extent you plan to plot as first argument and map-type
# as second.
base.map <- gmap(palo_alto, type = "terrain")

reprojected.palo_alto <- spTransform(palo_alto, base.map@crs)

plot(base.map)
```

```
plot(reprojected.palo_alto, add = T, border = "red", col = "transparent")
```

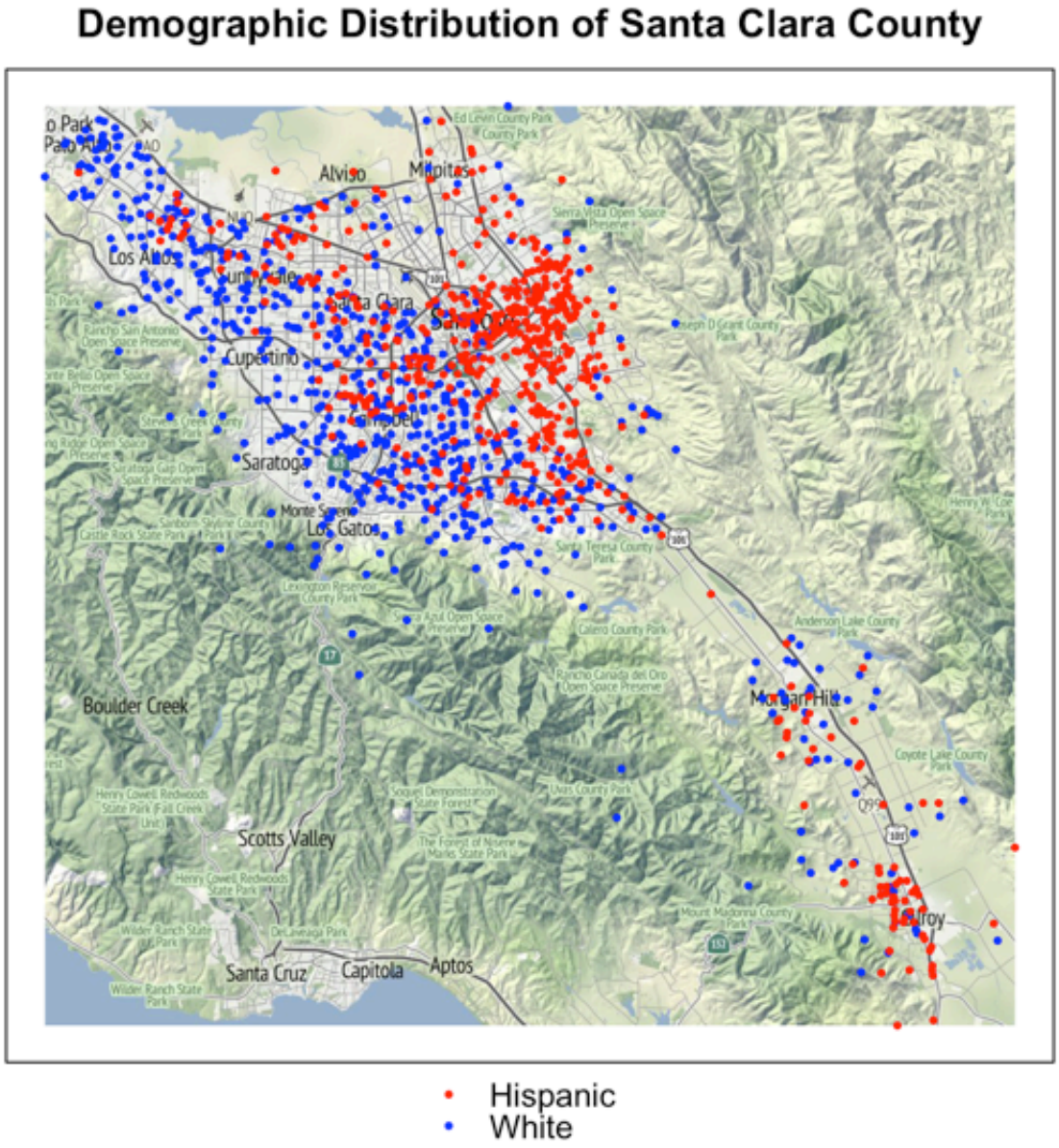


4.2 Basemaps with spplot

Unfortunately, adding a basemap to an `spplot` is much more complicated for a few reasons. Most of the code below can just be copy-pasted to make basemaps.

The reason – for those of you who are interested – is that the library that creates basemaps in a format that work with `spplot` wants to know about the bounds of the figure in latitude and longitude. The problem is that the projection these basemaps use does not use latitude and longitude as it's native coordinates. Thus you have to convert your image into the right projection (WGS84 Web Mercator), find the dimensions of your plot in those units, pass some of those back to the basemap function as longitudes and latitudes, then tell spplot what the dimensions of the basemap are in the WGS84 Web Mercator units. If you don't do this, you'll get this problem. (http://rstudio-pubs-static.s3.amazonaws.com/16660_7d1ab1b355344578bbacb0747fd485c8.html)

```
library(ggmap)
# REPROJECT YOUR DATA TO EPSG 3857
to.plot.web.merc <- spTransform(dots.all, CRS("+init=EPSG:3857"))


# COPY AND PASTE SEGEMENT 1 Series of weird conversions to deal with
# inconsistencies in units for API.
box <- to.plot.web.merc@bbox

midpoint <- c(mean(box[1, ]), mean(box[2, ]))
left.bottom <- c(box[1, 1], box[2, 1])
top.right <- c(box[1, 2], box[2, 2])

boundaries <- SpatialPoints(rbind(left.bottom, top.right))
proj4string(boundaries) <- CRS("+init=EPSG:3857")
boundaries.latlong <- c(t(spTransform(boundaries, CRS("+init=EPSG:4326"))@coords))

# END COPY-PASTE SEGMENT 1


# SET MAP TYPE HERE, LEAVE OTHER PARAMETERS AS THEY ARE
gmap <- get_map(boundaries.latlong, maptype = "terrain", source = "stamen",
    crop = TRUE)

# COPY-PASTE SEGMENT 2 Create object that sp.layout likes.
long.center <- midpoint[1]
lat.center <- midpoint[2]
height <- box[2, 2] - box[2, 1]
width <- box[1, 2] - box[1, 1]

sp.raster <- list("grid.raster", gmap, x = long.center, y = lat.center, width = wi
dth,
    height = height, default.units = "native", first = TRUE)
# END COPY-PASTE SEGMENT 2


# NORMAL PLOTTING TRICKS - HAVE FUN HERE!

# Housecleaning and set colors
to.plot.web.merc$ethnicity <- as.factor(to.plot.web.merc$ethnicity)

my.palette <- c("red", "blue")
point.size <- 0.5

# Plot!

spplot(to.plot.web.merc, "ethnicity", sp.layout = sp.raster, col.regions = my.pale
tte,
    cex = point.size, main = "Demographic Distribution of Santa Clara County")
```



Exercise 4

1. Plot the census tracts of your city over a base-map using the `plot` command.
2. Plot your demographic dot-plot with a basemap using the `spplot` command.

5. A final note on ggplot

The `ggplot2` library is a popular graphing library, and it can be used for mapping spatial data. The main trick is that `ggplot` will only accept DataFrames as inputs, so you have to first convert your spatial data into DataFrames using the `fortify` command from the `ggplot2` library.

I am not a frequent `ggplot` user, so I'll just include some workable code (taken almost verbatum from Claudia Engel's excellent tutorials) as guidance to interested parties.

```
library(ggplot2)

# create a unique ID for the later join
palo_alto$id = rownames(as.data.frame(palo_alto))


# turn SpatialPolygonsDataframe into a data frame
# (note that the rgeos library is required to use fortify)

palo_alto.pts <- fortify(palo_alto, region="id") #this only has the coordinates
palo_alto.df <- merge(palo_alto.pts, palo_alto, by="id", type='left') # add the at
tributes back


# calculate even breaks breaks
palo_alto.df$qt <- cut(palo_alto.df$PrCpInc, 5)


# plot
ggplot(palo_alto.df, aes(long,lat,group=group, fill=qt)) + # the data
  geom_polygon() + # make polygons
  scale_fill_brewer("Per Cap Income", palette = "OrRd") + # fill with brewer color
s
  theme(line = element_blank(),  # remove the background, tickmarks, etc
        axis.text=element_blank(),
        axis.title=element_blank(),
        panel.background = element_blank()) +
  coord_equal()
```