GIS and Spatial Analysis with R

Manny Gimond, Colby College

Maine GIS Educators Conference 2016, Belfast, ME

Table of Contents

# The workhorse: R

[R](http://cran.r-project.org/) is an open source data analysis and visualization programming environment whose roots go back to the [S programming language](https://en.wikipedia.org/wiki/S_%28programming_language%29) developed at Bell Laboratories in the 1970's by [John Chambers](https://en.wikipedia.org/wiki/John_Chambers_%28statistician%29). It's available for most operating systems including Windows and Macs.

You can download the latest version of R for Windows from <https://cran.r-project.org/bin/windows/base/> and for the latest version of R for Mac got to <https://cran.r-project.org/bin/macosx/>

# The friendly interface: RStudio

[RStudio](https://www.rstudio.com/products/RStudio/) is an integrated development environment (IDE) for R. It offers a user friendly interface to R by including features such as a source code editor (with colored syntax), data table viewer, git and github integration and markdown output. Note that RStudio is *not needed* to run R (which has its own IDE environment--albeit not as nice as RStudio's) but it makes using R far easier. RStudio is free software, but unlike R, it's maintained by a private entity which also distributes a commercial version of RStudio for businesses or individuals needing customer support.

You can download RStudio from this link <https://www.rstudio.com/products/rstudio/download3/#download>

# Command line vs. script file

## Command line

R can be run from a *R console* or a *RStudio* command line environment. For example, we can assign four numbers to the object x then have R read out the values stored in x by typing the following at a command line:

x <- c(1,2,3,4)  
x

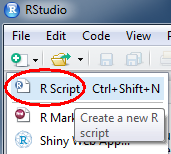
## [1] 1 2 3 4

<- is refered to as the assignment operator. Operations and functions to the right of the operator are stored in the object to the left.

## R script file

If you intend on typing more than a few lines of code in a command prompt environment, or if you wish to save a series of commands as part of a project's analysis, it is probably best that you type your commands in an a R script file. Such file is usually saved with a *.R* extension.

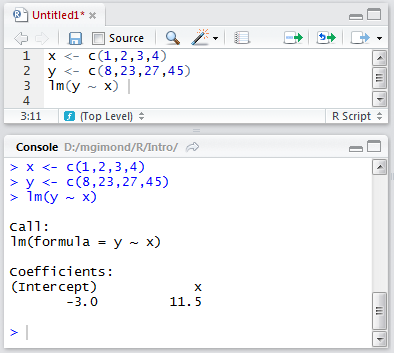
You create a new script by clicking on the upper-left icon and selecting *R Script*.



In RStudio, you can run (or *execute* in programming lingo) a line of code of an R script by placing a cursor anywhere on that line (while being careful not to highlight any subset of that line) and pressing the shortcut keys Ctrl+Enter (or Command+Enter on a Mac).

You can also run an entire block of code by selecting (highlighting) all lines to be run and pressing the shortcut keys Ctrl+Enter. Or, you can run the entire R script by pressing Ctrl+Alt+R.

In the following example, the R script file has three lines of code: two assignment operations and one regression analysis. The lines are run one at a time using the Ctrl+Enter keys and the output of each line is displayed in the console window.

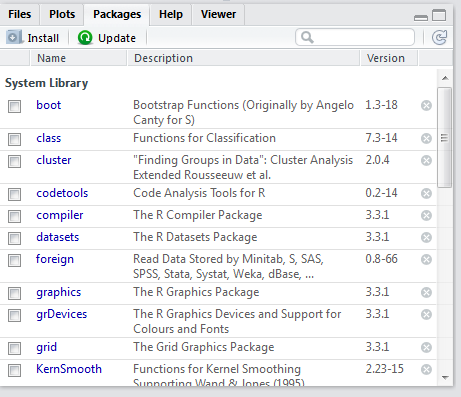


# Packages

One of R's attractive features is its rich collection of packages designed for specific applications and analysis techniques. Packages allow researchers and scientists to share R functions and data with other users. Some packages come already installed with R, others must be downloaded separately from a CRAN repository or other locations such as GitHub or a personal website.

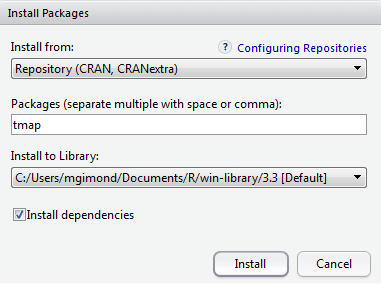
## Base packages

R comes installed with a set of default packages. A snapshot of a subset of the installed base packages is shown below:



## Installing packages from CRAN

There are thousands of R packages to choose from. Most can be accessed from the CRAN repository. To install a CRAN package from within RStudio, click on the *Packages* tab, select *Install* and choose *Repository (CRAN, CRANextra)* as the source location. In the following example, the package *tmap* is installed from CRAN.



Package installation from CRAN's repository can also be accomplished using the following line of code:

install.packages("tmap")

The installation is usually straightforward and if any other packages need to be installed, R will install those as well if the *Install dependencies* option is checked. In the previous example, tmap requires that more than two dozen or so packages be present on your computer (such as RColorBrewer , sp and rgdal)--all of which are automatically installed.

Note that R packages are installed in the user's home directory (C:/Users/...) by default. This is advantageous in that you do not need to have administrative privileges to install any package. But it can be a disadvantage in that if someone else logs on to the same computer where you installed a package, that person will not have access to it requiring that she install that same package in *her* home directory thereby duplicating an instance of that package on the same computer.

Packages used in this workshop that you might want to download now are: tmap, rgdal, raster, rgeos. Note that installing tmap forced R to install rgdal, raster and rgeos since tmap makes use of a subset of their functions, so these packages do not need to be installed explicitly.

## Using a package in a R session

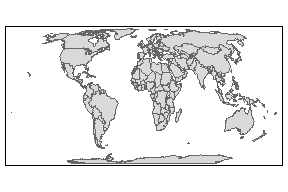
Just because a package is installed on your computer (in your home directory or in a directory accessible to you) does not mean that you have access to its functions in an R session. For example, after installing the tmap package you might want to access one of its datasets (World) and make use of one of its functions (qtm) to generate a choropleth map. This requires that tmap's function be loaded at the beginning of the session using the library() function. Here's what happens when you *don't load* the tmap package before using its World dataset and running its qtm() function:

data(World)   
qtm(World)

## Error in eval(expr, envir, enclos): could not find function "qtm"

To make the functions and/or data of a package available in an R session, use the library() function:

library(tmap)  
data(World)  
qtm(World)



# Getting a session's info

Reproducibility is a fundamental idea behind an open source analysis environment such as R. So it's only fitting that *all* aspects of your analysis environment be made available (along with your data and analysis results). This is because functions and programming environments may change in their behavior as versions evolve; this may be by design or the result of a bug in the code fixed in later versions. No pieces of software, open-source or commercial, are immune to this. It's therefore important that you publicize the R session in which parts of or all analyses were conducted. A simple way to do this is to call the sessionInfo() function.

sessionInfo()

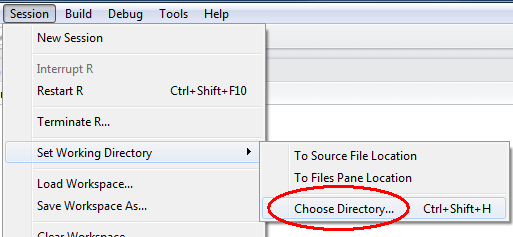
## R version 3.3.0 (2016-05-03)  
## Platform: x86\_64-w64-mingw32/x64 (64-bit)  
## Running under: Windows 7 x64 (build 7601) Service Pack 1  
##   
## locale:  
## [1] LC\_COLLATE=English\_United States.1252   
## [2] LC\_CTYPE=English\_United States.1252   
## [3] LC\_MONETARY=English\_United States.1252  
## [4] LC\_NUMERIC=C   
## [5] LC\_TIME=English\_United States.1252   
##   
## attached base packages:  
## [1] stats graphics grDevices utils datasets methods base   
##   
## other attached packages:  
## [1] tmap\_1.4-1  
##   
## loaded via a namespace (and not attached):  
## [1] Rcpp\_0.12.5 spdep\_0.6-5 formatR\_1.4   
## [4] RColorBrewer\_1.1-2 LearnBayes\_2.15 bitops\_1.0-6   
## [7] class\_7.3-14 tools\_3.3.0 boot\_1.3-18   
## [10] digest\_0.6.9 evaluate\_0.9 nlme\_3.1-127   
## [13] lattice\_0.20-33 Matrix\_1.2-6 yaml\_2.1.13   
## [16] rgdal\_1.1-10 e1071\_1.6-7 coda\_0.18-1   
## [19] osmar\_1.1-7 stringr\_1.0.0 knitr\_1.13.7   
## [22] raster\_2.5-8 gtools\_3.5.0 htmlwidgets\_0.6   
## [25] rgeos\_0.3-19 classInt\_0.1-23 leaflet\_1.0.1   
## [28] grid\_3.3.0 XML\_3.98-1.4 rmarkdown\_1.0.9001  
## [31] sp\_1.2-3 gdata\_2.17.0 deldir\_0.1-12   
## [34] magrittr\_1.5 gmodels\_2.16.2 htmltools\_0.3.5   
## [37] MASS\_7.3-45 splines\_3.3.0 geosphere\_1.5-5   
## [40] KernSmooth\_2.23-15 stringi\_1.1.1 RCurl\_1.95-4.8

Output includes all loaded base packages and external packages (e.g. *tmap* in this working example) as well as their version.

# Exploring vector data

The following exercise assumes that your data files are stored in a folder called Data and that the R session working folder is at the the same directory level as the Data folder.

You change an R session's working directory via the following pulldown:



## Loading a shapefile

Shapefiles are easy to work with in R. Many packages will read them into an R session including rgdal.

library(rgdal)  
me <- readOGR("./Data", "Income")

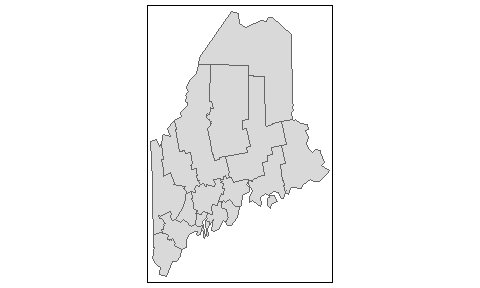
## OGR data source with driver: ESRI Shapefile   
## Source: "./Data", layer: "Income"  
## with 16 features  
## It has 2 fields

R stores the contents of the shapefile as a SpatialPolygonsDataFrame object in an object called me in your current R session.

## Create a simple map

To check that the geometry was properly loaded, use the qtm function from the tmap package.

qtm(me)



## Exploring the attributes table

The attributes table can be accessed as follows;

me@data

## NAME Income  
## 0 Aroostook 21024  
## 1 Somerset 21025  
## 2 Piscataquis 21292  
## 3 Penobscot 23307  
## 4 Washington 20015  
## 5 Franklin 21744  
## 6 Oxford 21885  
## 7 Waldo 23020  
## 8 Kennebec 25652  
## 9 Androscoggin 24268  
## 10 Hancock 28071  
## 11 Knox 27141  
## 12 Lincoln 27839  
## 13 Cumberland 32549  
## 14 Sagadahoc 28122  
## 15 York 28496

The attribute names can be accessed as follows:

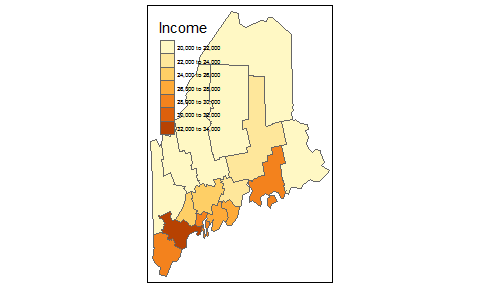
names(me)

## [1] "NAME" "Income"

There are two attributes (fields) in our table: NAME and Income. Note that R is case sensitive so when referencing theses field names, you must be mindful of the case.

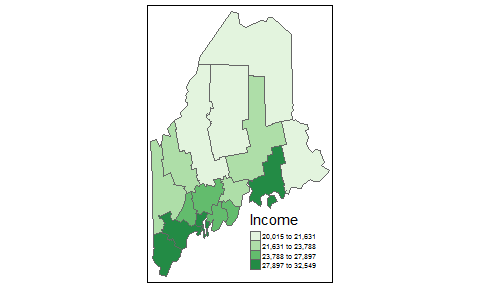
Let's symbolize the polygons using the Income attribute.

qtm(me, fill="Income")



This is a good first start, but it would be nice to control the color scheme as well as the legend placement.

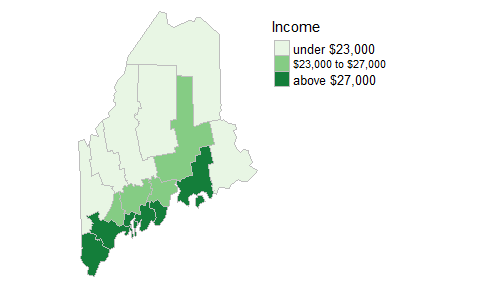
qtm(me, fill="Income", fill.style="quantile",   
 fill.n=4,  
 fill.palette="Greens",  
 legend.text.size = 0.5,  
 layout.legend.position = c("right", "bottom"))



Here, we use a *quantile* classification scheme with four classes (fill.n=4), a green hue following Cindy Brewer color scheme recommendations, a font size half the size of a default font (legend.text.size = 0.5) and a legend placement in the lower right corner of the layout (layout.legend.position = c("right", "bottom")).

tmap offers us additional control over the layout by breaking up the different mapping elements into individual functions tied together by +. If you have used the ggplot2 package in the past, this approach to tying mapping elements should be familiar. Here, we customize the legend box, remove the bounding box and place the legend outside of the bounding box.

tm\_shape(me) +   
 tm\_fill("Income", style="fixed", breaks=c(0,23000 ,27000,100000 ),  
 labels=c("under $23,000", "$23,000 to $27,000", "above $27,000"),  
 palette="Greens") +  
 tm\_borders("grey") +  
 tm\_legend(outside = TRUE, text.size = .8) +  
 tm\_layout(frame = FALSE)



## Joining attribute tables

In this exercise, we'll join a CSV file to the me spatial object. First, we'll load the contents of the CSV file into an oject we'll call dat.

dat <- read.csv("./Data/Crime2012.csv")

Next, we'll make use of dplyr's left\_join function to join the table to the spatial object me by county name.

library(dplyr)  
me@data <- left\_join(me@data, dat, by=c("NAME" = "County"))

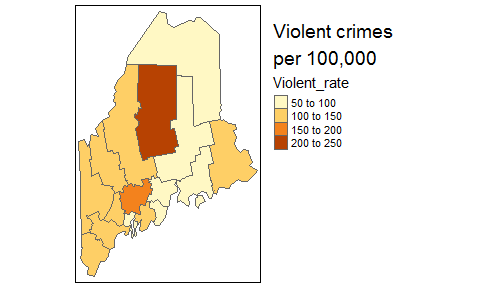
Here, we are accessing the attributes table (stored in the data component) via me@data. In essence, the above code is joining the data table dat to the attributes table me@data and overwriting the existing attributes table in the process. The function is also provided with a join key that dictates which columns from each table are to be used to match the records. Here, we are matching me's NAME column to dat's County column.

The joined table reports the number of property and violent crimes within each county. We will create a new *violent crime* rate field (per 100,000) from the violent crime count and population fields.

me$Violent\_rate <- me$Violent / me$Population \* 100000

Now let's map the violent crime rate.

qtm(me, fill="Violent\_rate", fill.style="pretty", title="Violent crimes \nper 100,000", legend.outside=TRUE)



# Exploring raster data

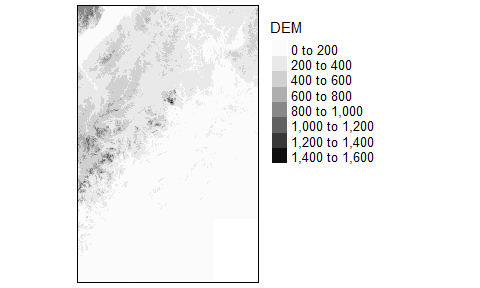
## Loading a raster file

Let's load a DEM raster of Maine. We could use the readOGR() function for this, but we will make use of another package called raster. In addition to reading many raster formats, raster has many wonderful map algebra operations capability.

library(raster)  
dem <- raster("./Data/DEM.img")

Let's map the raster.

tm\_shape(dem) + tm\_raster(palette = "Greys", n=8) +  
 tm\_legend(outside = TRUE, text.size = .8)

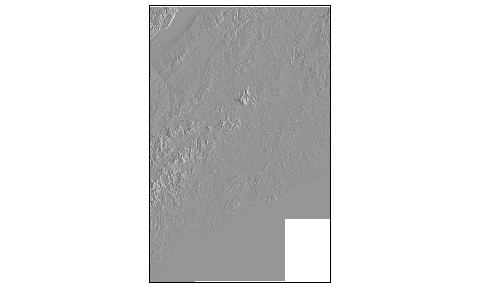


Note the use of the tm\_shape function. The word *shape* does not refer to a shapefile but references the feature to be mapped (vector or raster).

## Creating a hillshade

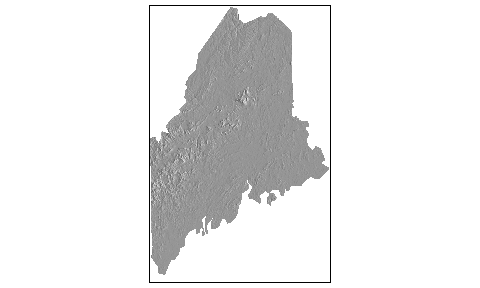
In this example, we make use of raster's hillshade function to create a hillshade rendering of the DEM. This requires that we first create a slope an aspect raster from the data using the terrain function.

slope <- terrain(dem, opt='slope')  
aspect <- terrain(dem, opt='aspect')  
hill <- hillShade(slope, aspect, 40, 270)  
tm\_shape(hill) + tm\_raster(palette = "Greys", style="cont", contrast = c(0,.8)) +  
 tm\_legend(show=FALSE)



We can clip/mask the hillshade raster to the Maine polygon outline using the mask function (equivalent to the *Extract by Mask* tool in ArcGIS).

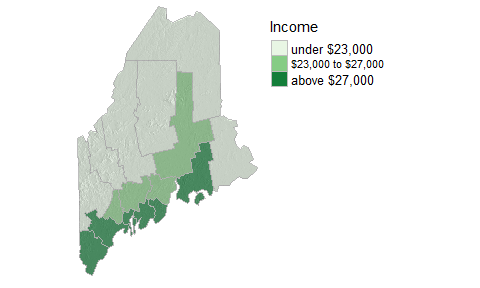
hill\_me <- mask(hill, me)  
tm\_shape(hill\_me) + tm\_raster(palette = "Greys", style="cont", contrast = c(0,.8)) +  
 tm\_legend(show=FALSE)



## Overlaying raster layers with vector layers

The tmap package allows you to overlay different features by simply appending tm\_ functions. In the following example, we map me then overlay it with a semi-transparent rendering of the hillshade.

tm\_shape(me) +   
 tm\_fill("Income", style="fixed", breaks=c(0,23000 ,27000,100000 ),  
 labels=c("under $23,000", "$23,000 to $27,000", "above $27,000"),  
 palette="Greens") +  
 tm\_borders("grey") +  
 tm\_legend(outside = TRUE, text.size = .8) +  
 tm\_layout(frame = FALSE) +  
 tm\_shape(hill\_me) + tm\_raster(palette = "Greys", style="cont",   
 contrast = c(0,.8), alpha=0.4,   
 legend.show = FALSE)

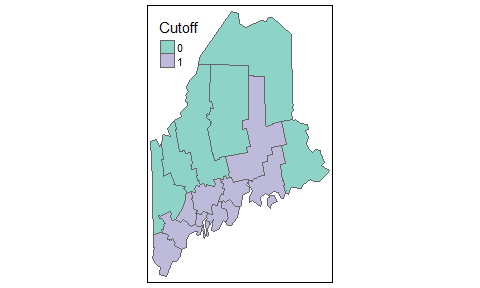


# Vector data manipulation

## Dissolving on a variable

Let's create a binary vector layer where a value of 1 is assigned to all counties having a value greater than $23,000 and a value of 0 for those below that amount. We'll create a new column called Cutoff to store this value.

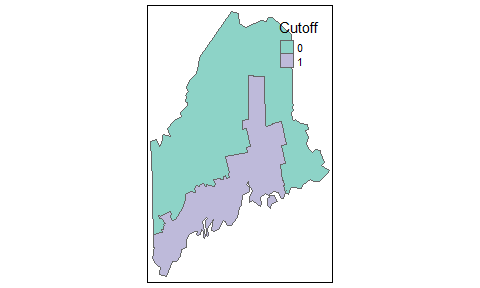
me$Cutoff <- ifelse( me$Income > 23000, 1, 0)  
qtm(me, fill="Cutoff", fill.style="cat")



In the above example, we force the numeric values to categorical values with the call to fill.style="cat". This is done to prevent qtm from attempting to created *ranges* of value such as -0.5 to 0.5 and 0.5 to 1.5.

Next, we'll dissolve the polygons based on the Cutoff value using raster's aggregate function.

me\_dis <- aggregate(me, by="Cutoff")  
qtm(me\_dis, fill="Cutoff", fill.style="cat")



## Transforming coordinate systems

In this example, we'll load a point layer representing Augusta whose coordinate pair is stored in a geographic (lat/long) coordinate system. We'll then project it into a UTM NAD83 Zone 19N coordinate system.

Aug\_ll <- readOGR("./Data", "Augusta\_ll")

## OGR data source with driver: ESRI Shapefile   
## Source: "./Data", layer: "Augusta\_ll"  
## with 1 features  
## It has 1 fields

To extract the coordinate system information type,

proj4string(Aug\_ll)

## [1] "+proj=longlat +datum=WGS84 +no\_defs +ellps=WGS84 +towgs84=0,0,0"

Coordinate systems are defined in R using a syntax created by the (PROJ4)[<http://proj4.org/>] project. It consists of a list of parameters, each prefixed with the + character. For example, Aug\_ll's CS is in a lat/long coordinate system (+proj=longlat) built on a WGS 1984 datum (+datum=WGS84) which is itself built on top of a WGS84 ellipsoid.

To change the coordinate system, we must create the projection string. There are several options. If you know the EPSG index for your coordinate system, you can reduce the coordinate string to +init=epsg:26919 as in,

Aug\_UTM <- spTransform(Aug\_ll, CRS("+init=epsg:26919"))

If you don't have the EPSG value or would rather manually define the coordinate system, you could use the following syntax:

Aug\_UTM <- spTransform(Aug\_ll, CRS("+proj=utm +zone=19 +datum=NAD83 +units=m +no\_defs +ellps=GRS80 +towgs84=0,0,0"))

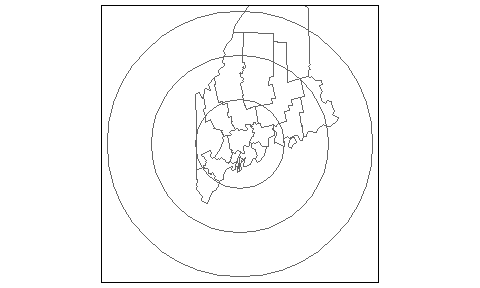
Note that the proj4string and spTransform functions are part of the sp package. We did not need to manually load this package since it was automatically loaded when we loaded the rgdal package earlier in this exercise.

The CRS projection string is very versatile--allowing you to customize your own projection. For a comprehensive list of projection types and parameters, visit the [proj4.org](http://proj4.org/) website.

## Example: Calculating average income as a distance from Augusta

First, we'll buffer the Augusta point using three different distances: 100km, 200km and 300km. Note that the map units are in meters which will require that we encode the distances in meters and not kilometers. We'll make use of raster's buffer function.

d100 <- buffer(Aug\_UTM, width=100000, quadsegs=10)  
d200 <- buffer(Aug\_UTM, width=200000, quadsegs=10)  
d300 <- buffer(Aug\_UTM, width=300000, quadsegs=10)  
  
# Let's check the circles on a Maine outline  
tm\_shape(d300) +tm\_borders() +tm\_shape(d200) +tm\_borders() + tm\_shape(d100) +tm\_borders() +tm\_shape(me) +tm\_borders()



The quadsegs=10 parameter defines the number of line segments to use per quarter circle. In our example, we are using 10 segments per circle for a total of 40 line segments.

Next, we will combine the circles to create a single layer defining the different annuli using the union function. We will do this twice since the function can only accept two layers at a time.

dAll <- union(d200, d100)  
dAll <- union(d300, dAll)

Not that the union operation will not allow polygons to overlap so any overlapping area of the polygons are merged into a single one.

The output of the union operation did not create an attributes table. We will therefore create one by add an id value (1 thorugh 3) to the polygons.

dAll$ID <- 1:length(dAll)

Next, we will clip the distance bands to the state of Maine (this will be important for subsequent steps).

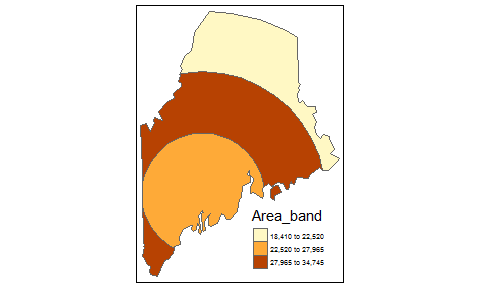
dAllme <- crop(dAll, me)

Next, we'll compute the area in each bacd using the gArea function from the rgeos package.

library(rgeos)  
dAllme$Area\_band <- gArea(dAllme, byid=TRUE) / 1000000 # Compute area in km2

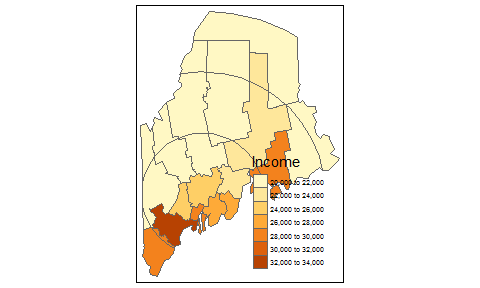
Let's map the distance bands

qtm(dAllme, fill="Area\_band")



Next we'll intersect the distance bands with the income layers.

clp1 <- intersect(me, dAllme)  
tm\_shape(clp1) + tm\_fill(col="Income") + tm\_borders()



Next, we'll compute the area for each polygon in this new layer.

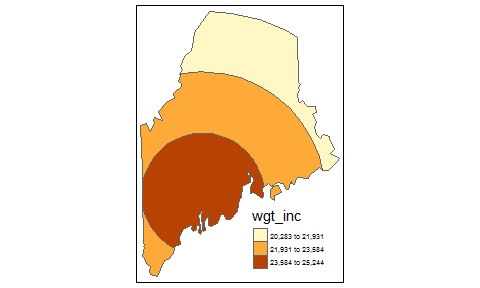
clp1$Area\_sub <- gArea(clp1, byid=TRUE) / 1000000

Each polygon now has two area attributes: the area associated with the distance band and each polygon that was created from the intersection between the income layer and the distance band layer. The next step is to calculate the fraction of the distance band area covered by the polygon, then to weigh each polygon's income value using that fraction.

# Compute the polygon's fraction vis-a\_vis the distance band's area  
clp1$frac\_area <- clp1$Area\_sub / clp1$Area\_band  
  
# Mutiply income by area fraction--this is the weighted income   
# within each distance band  
clp1$wgt\_inc <- clp1$Income \* clp1$frac\_area

Finally, we will dissolve all polygons (using the aggregate function) by distance band and sum the weighted income values to give us average income values per distance.

dist\_inc <- aggregate(clp1, by="ID",sums= list(list(sum, "wgt\_inc")))  
qtm(dist\_inc, fill="wgt\_inc")



# Additional resources

* I have additional *GIS with R* tutorials in the Appendix of my ES214 course site at <https://mgimond.github.io/Spatial>. The Appendix also offers an extensive write-up on (raster) map algebra operations in R; this includes local, focal, zonal ang global operations.
* For a general introduction to R and basic (aspatial) data manipulation and visualization you might want to skim through my *Exploratory Data Analysis with R* course page: [https://mgimond.github.io/ES218](https://mgimond.github.io/ES218/index.html).
* To stay abreast of new spatially related packages in R, follow Roger Bivand's *CRAN Task View* at <https://cran.r-project.org/web/views/Spatial.html>
* I know of only one book dedicated to GIS data manipulation in R: *An Introduction to R for Spatial Analysis & Mapping* by Chris Brunsdon and Lex Comber. The authors make use of another R package called GISTools. Chris Brunsdon also offers an online course in GIS data manipulation in R on the *Statistics.com* website.
* If you are interested in spatial analysis and statistics, you will want to check out *Applied Spatial Analysis with R* by R. Bivand, E. Pebesma and V. Gomez-Rubio. They make heavy use of a package called sp.