Spatial computing in R

Tad Dallas

Reading for this week:

> Brunsdon and Comber (2020) Opening practice: supporting reproducibility and critical spatial data science. *Journal of Geographical Systems* https://doi.org/10.1007/s10109-020-00334-2

Spatial phenomena can generally be thought of as either discrete objects with clear boundaries or as a continuous phenomenon that can be observed everywhere, but does not have natural boundaries. Discrete spatial objects may refer to a river, road, country, town, or a research site. Examples of continuous phenomena, or "spatial fields", include elevation, temperature, and air quality.

Spatial objects are usually represented by vector data. Such data consists of a description of the *geometry* or *shape* of the objects, and normally also includes additional variables. For example, vector data can describe geographic border data while also containing information on attributes of bounded geographic locations (e.g., vector data describing borders of a country could include attribute data at smaller municipal locations like counties). In this lecture, we will take a fairly shallow dive into spatial data representation and manipulation in `R`.

This lecture would not have been possible without the help of Robert Hijmans and his work on Spatial Data Science (https://rspatial.org/raster/spatial/). For a much deeper dive into working with spatial data in R, please see that resource.

Vector data

The main vector data types are points, lines and polygons. In all cases, the geometry of these data structures consists of sets of coordinate pairs (x, y). Points are the simplest case. Each point has one coordinate pair, and n associated variables.

The geometry of lines is a just a little bit more complex. Lines are represented as ordered sets of coordinates (nodes). The actual line segments can be computed (and drawn on a map) by connecting the points. Thus, the representation of a line is very similar to that of a multi-point structure. The main difference is that the ordering of the points is important, because we need to know which points should be connected. A network (e.g. a road or river network), or spatial graph, is a special type of lines geometry where there is additional information about things like flow, connectivity, direction, and distance.

A polygon refers to a set of closed lines. The geometry is very similar to that of lines, but to close a polygon the last coordinate pair coincides with the first pair. A complication with polygons is that they can have holes (that is a polygon entirely enclosed by another polygon, that serves to remove parts of the enclosing polygon). For instance, consider a polygon of land mass which has multiple lakes within the polygon which describes the land mass. Valid polygons do not self-intersect, but multiple polygons can be considered as a single geometry. For example, Indonesia consists of many islands. Each island can be represented by a single polygon, but together then can be represent a single (multi-) polygon representing the entire country.

Raster data

Raster data is commonly used to represent spatially continuous phenomena such as elevation. A raster divides the world into a grid of equally sized rectangles (referred to as cells or, in the context of satellite remote sensing, pixels) that all have one or more values (or missing values) for the variables of interest. A raster cell value should normally represent the average (or majority) value for the area it covers. However, in some cases the values are actually estimates for the center of the cell (in essence becoming a regular set of points with an attribute).

In contrast to vector data, in raster data the geometry is not explicitly stored as coordinates. It is implicitly set by knowing the spatial extent and the number or rows and columns in which the area is divided. From the extent and number of rows and columns, the size of the raster cells (spatial resolution) can be computed. Think of rasters as a matrix of spatial data, where each cell of the matrix corresponds to a block of spatial data and some variable(s) measured at each matrix cell.

Simple representation of spatial data

The basic data types in R are numbers, characters, logical (TRUE or FALSE) and factor values. Values of a single type can be combined in vectors and matrices, and variables of multiple types can be combined into a data.frame. We can represent (only very) basic spatial data with these data types. Let's say we have the location (represented by longitude and latitude) of ten weather stations (named A to J) and their annual precipitation.

In the example below we make a very simple map. Note that a map is special type of plot (like a scatter plot, barplot, etc.). A map is a plot of geospatial data that also has labels and other graphical objects such as a scale bar or legend. The spatial data itself should not be referred to as a map.

Libraries

We'll use three out of the many many many spatial libraries that exist. We'll use `sp`, `raster`, and `rgdal`. You may run into installation issues, since at least `rgdal` relies on a backend library already existing. This is already a signal that reproducibility with spatial data is an inherent challenge. The last lecture, which went over docker and setting up a computational environment explicitly in a container, might seem a bit more worthwhile after hitting issues with spatial data packages.

```r
install.packages(c('sp', 'raster', 'rgdal'))
```

```
## Installing packages into '/home/tad/R/x86_64-pc-linux-gnu-library/4.1'
## (as 'lib' is unspecified)
```

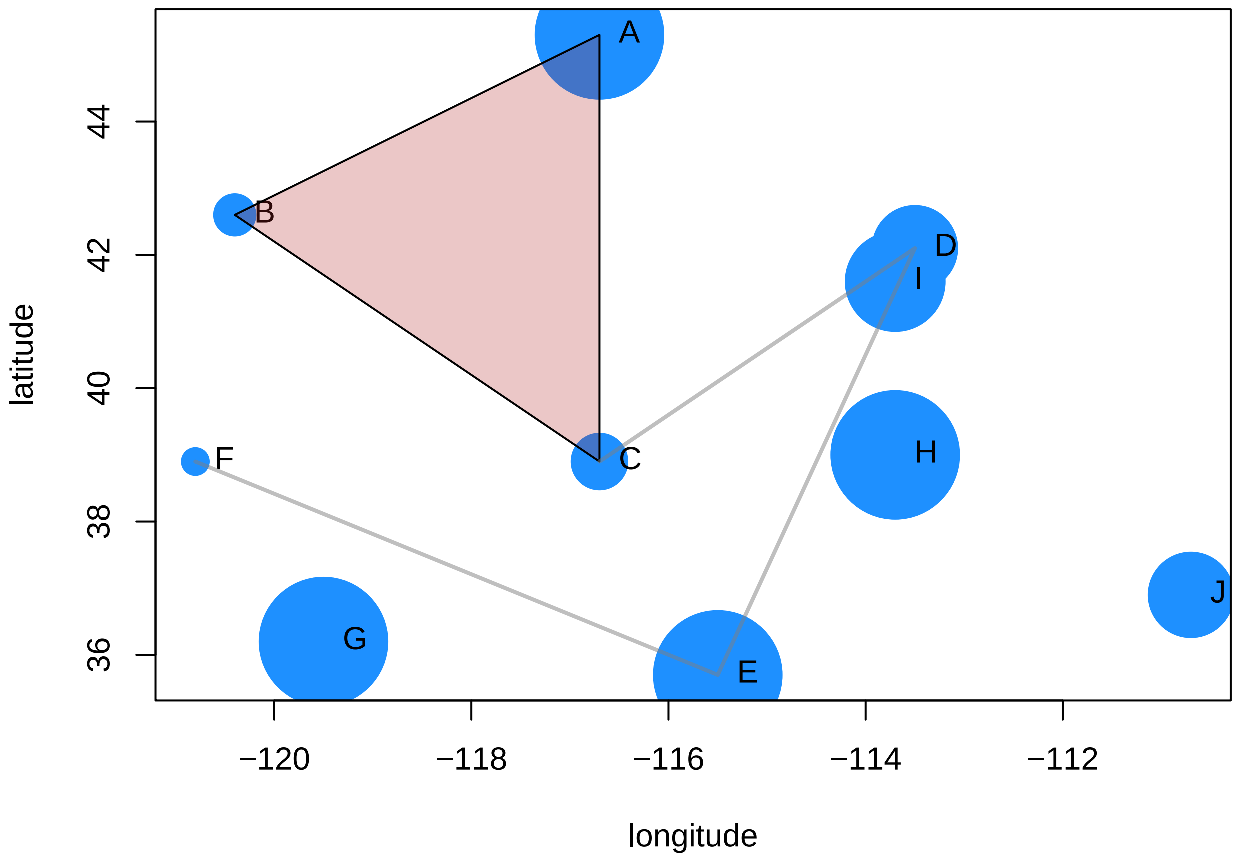
```
## also installing the dependency 'terra'
```

```r
name <- LETTERS[1:10]
longitude <- c(-116.7, -120.4, -116.7, -113.5, -115.5,
               -120.8, -119.5, -113.7, -113.7, -110.7)
latitude <- c(45.3, 42.6, 38.9, 42.1, 35.7, 38.9,
              36.2, 39, 41.6, 36.9)


# Simulated rainfall data
set.seed(0)
precip <- round((runif(length(latitude))*10))
```

```r
stations <- data.frame(longitude, latitude, name, precip)
```

This is vector data.

```r
par(mar=c(4,4,0.5,0.5))

# adding points
plot(stations[,1:2],
    cex=stations$precip,
    pch=16,
    col='dodgerblue')
text(stations, name, pos=4)

# adding polygons
polygon(stations[1:3, 1:2],
    col=adjustcolor('firebrick', 0.25))

# adding lines
lines(stations[3:6, 1:2], lwd=2, col=grey(0.5,0.5))
```



Polygons need to *closed*, that is, the first point must coincide with the last point, but the polygon function took care of that for us.

sp data

Package sp is the central package supporting spatial data analysis in R. sp defines a set of classes to represent spatial data. A class defines a particular data type. The data.frame is an example of a class. Any particular data.frame you create is an object (instantiation) of that class.

The main reason for defining classes is to create a standard representation of a particular data type to make it easier to write functions (also known as 'methods') for them. In fact, the sp package does not provide many functions to modify or analyze spatial data; but the classes it defines are used in more than 100 other R packages that provide specific functionality. See Hadley Wickham's Advanced R or John Chambers' Software for data analysis for a detailed discussion of the use of classes in R).

We will be using the sp package here. Note that this package will eventually be replaced by the newer sf package — but sp is still more commonly used.

Package sp introduces a number of classes with names that start with Spatial. For vector data, the basic types are the SpatialPoints, SpatialLines, and SpatialPolygons. These classes only represent geometries. To also store attributes, classes are available with these names plus DataFrame, for example, SpatialPolygonsDataFrame and SpatialPointsDataFrame. When referring to any object with a name that starts with Spatial, it is common to write Spatial*

```r
#install.packages('sp')
library(sp)

pts <- sp::SpatialPoints(stations[,1:2])


typeof(pts)
```

```
## [1] "S4"
```

```r
class(pts)
```

```
## [1] "SpatialPoints"
## attr(,"package")
## [1] "sp"
```

```r
str(pts)
```

```
## Formal class 'SpatialPoints' [package "sp"] with 3 slots
##   ..@ coords     : num [1:10, 1:2] -117 -120 -117 -114 -116 ...
##   .. ..- attr(*, "dimnames")=List of 2
##   .. .. ..$ : NULL
##   .. .. ..$ : chr [1:2] "longitude" "latitude"
##   ..@ bbox       : num [1:2, 1:2] -120.8 35.7 -110.7 45.3
##   .. ..- attr(*, "dimnames")=List of 2
##   .. .. ..$ : chr [1:2] "longitude" "latitude"
##   .. .. ..$ : chr [1:2] "min" "max"
##   ..@ proj4string:Formal class 'CRS' [package "sp"] with 1 slot
##   .. .. ..@ projargs: chr NA
```

Coordinate reference systems.

There is also a "proj4string". This stores the coordinate reference system ("crs", discussed in more detail later). We did not provide the crs so it is unknown (NA). That is not good, so let's recreate the object, and now provide a crs.

```
pts <- sp::SpatialPoints(pts, proj4string=CRS("+proj=longlat +datum=WGS84"))
ptsdf <- sp::SpatialPointsDataFrame(pts, data=stations[,3:4])
```

Manipulating vector data

```
str(ptsdf)
```

```
## Formal class 'SpatialPointsDataFrame' [package "sp"] with 5 slots
##   ..@ data       :'data.frame':  10 obs. of  2 variables:
##   .. ..$ name  : chr [1:10] "A" "B" "C" "D" ...
##   .. ..$ precip: num [1:10] 9 3 4 6 9 2 9 9 7 6
##   ..@ coords.nrs : num(0)
##   ..@ coords     : num [1:10, 1:2] -117 -120 -117 -114 -116 ...
##   .. ..- attr(*, "dimnames")=List of 2
##   .. .. ..$ : NULL
##   .. .. ..$ : chr [1:2] "longitude" "latitude"
##   ..@ bbox       : num [1:2, 1:2] -120.8 35.7 -110.7 45.3
##   .. ..- attr(*, "dimnames")=List of 2
##   .. .. ..$ : chr [1:2] "longitude" "latitude"
##   .. .. ..$ : chr [1:2] "min" "max"
##   ..@ proj4string:Formal class 'CRS' [package "sp"] with 1 slot
##   .. .. ..@ projargs: chr "+proj=longlat +datum=WGS84 +no_defs"
##   .. .. ..$ comment: chr "GEOGCRS[\"unknown\",\n    DATUM[\"World Geodetic System 1984\",\n        E
```

```
head(ptsdf@data)
```

```
##   name precip
## 1    A      9
## 2    B      3
## 3    C      4
## 4    D      6
## 5    E      9
## 6    F      2
```

```
head(ptsdf)
```

```
##       coordinates name precip
## 1 (-116.7, 45.3)    A      9
## 2 (-120.4, 42.6)    B      3
## 3 (-116.7, 38.9)    C      4
## 4 (-113.5, 42.1)    D      6
## 5 (-115.5, 35.7)    E      9
## 6 (-120.8, 38.9)    F      2
```

Built-in functionality to work with base R syntax is present for most of these types of data structures. For instance, the addition of a vector of data can be done in the same manner as you would add a variable to a data.frame.

```
ptsdf$random <- runif(nrow(ptsdf))
ptsdf
```

```
##       coordinates name precip     random
## 1  (-116.7, 45.3)    A      9 0.06178627
## 2  (-120.4, 42.6)    B      3 0.20597457
## 3  (-116.7, 38.9)    C      4 0.17655675
## 4  (-113.5, 42.1)    D      6 0.68702285
## 5  (-115.5, 35.7)    E      9 0.38410372
## 6  (-120.8, 38.9)    F      2 0.76984142
## 7  (-119.5, 36.2)    G      9 0.49769924
## 8    (-113.7, 39)    H      9 0.71761851
## 9  (-113.7, 41.6)    I      7 0.99190609
## 10 (-110.7, 36.9)    J      6 0.38003518
```

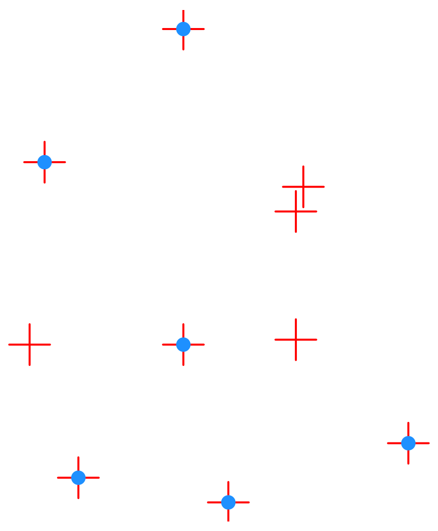
This SpatialPointsDataFrame can be subset like any data.frame or matrix

```
ptsdf2 <- ptsdf[which(ptsdf$random < 0.5), ]
ptsdf2
```

```
##       coordinates name precip     random
## 1  (-116.7, 45.3)    A      9 0.06178627
## 2  (-120.4, 42.6)    B      3 0.20597457
## 3  (-116.7, 38.9)    C      4 0.17655675
## 5  (-115.5, 35.7)    E      9 0.38410372
## 7  (-119.5, 36.2)    G      9 0.49769924
## 10 (-110.7, 36.9)    J      6 0.38003518
```

Then we can visualize the points that are included in both data.frames (the whole `ptsdf` and the subset `ptsdf` where our random variable is less than 0.5).

```
plot(ptsdf, col='red', cex=2)
points(ptsdf2, pch=16, col='dodgerblue')
```



Raster data

The `raster` package has functions for creating, reading, manipulating, and writing raster data. The package provides, among other things, general raster data manipulation functions that can easily be used to develop more specific functions. For example, there are functions to read a chunk of raster values from a file or to convert cell numbers to coordinates and back. The package also implements raster algebra and many other functions for raster data manipulation.

First, we will install the `raster` package and create our first simple raster. We give it an "extent" which mirrors latitude and longitude coordinates. An extent is just the bounding box of values which the raster, as a whole, covers. This is important to make the dimensions of the raster mean something, as otherwise we would simply assume that each raster cell is n units wide by m units tall.

```r
#install.packages('raster')
library(raster)

rast <- raster(ncol=100, nrow=100,
    xmx=-80, xmn=-150, ymn=20, ymx=60)

values(rast) <- runif(ncell(rast))

rast
```

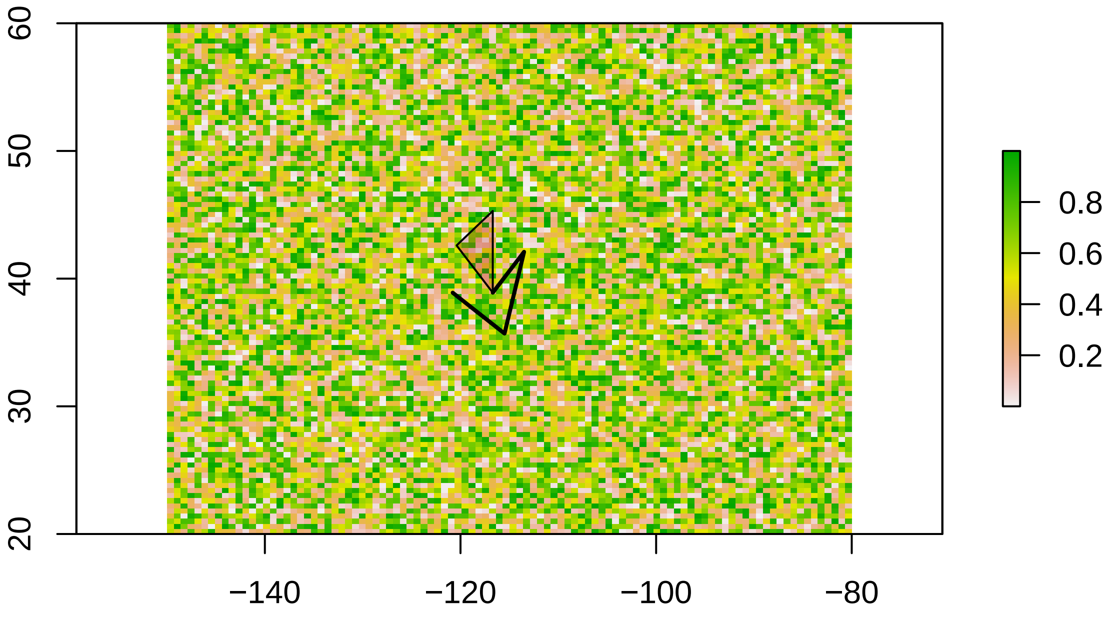
```
## class      : RasterLayer
## dimensions : 100, 100, 10000  (nrow, ncol, ncell)
## resolution : 0.7, 0.4  (x, y)
## extent     : -150, -80, 20, 60  (xmin, xmax, ymin, ymax)
## crs        : +proj=longlat +datum=WGS84 +no_defs
## source     : memory
## names      : layer
## values     : 0.0001064336, 0.9999306  (min, max)
```

```r
str(rast)
```

```
## Formal class 'RasterLayer' [package "raster"] with 12 slots
##   ..@ file    :Formal class '.RasterFile' [package "raster"] with 13 slots
##   .. .. ..@ name        : chr ""
##   .. .. ..@ datanotation: chr "FLT4S"
##   .. .. ..@ byteorder   : chr "little"
##   .. .. ..@ nodatavalue : num -Inf
##   .. .. ..@ NAchanged   : logi FALSE
##   .. .. ..@ nbands      : int 1
##   .. .. ..@ bandorder   : chr "BIL"
##   .. .. ..@ offset      : int 0
##   .. .. ..@ toptobottom : logi TRUE
##   .. .. ..@ blockrows   : int 0
##   .. .. ..@ blockcols   : int 0
##   .. .. ..@ driver      : chr ""
##   .. .. ..@ open        : logi FALSE
##   ..@ data    :Formal class '.SingleLayerData' [package "raster"] with 13 slots
##   .. .. ..@ values    : num [1:10000] 0.777 0.935 0.212 0.652 0.126 ...
##   .. .. ..@ offset    : num 0
##   .. .. ..@ gain      : num 1
##   .. .. ..@ inmemory  : logi TRUE
##   .. .. ..@ fromdisk  : logi FALSE
##   .. .. ..@ isfactor  : logi FALSE
##   .. .. ..@ attributes: list()
##   .. .. ..@ haveminmax: logi TRUE
##   .. .. ..@ min       : num 0.000106
##   .. .. ..@ max       : num 1
##   .. .. ..@ band      : int 1
##   .. .. ..@ unit      : chr ""
##   .. .. ..@ names     : chr ""
##   ..@ legend  :Formal class '.RasterLegend' [package "raster"] with 5 slots
##   .. .. ..@ type      : chr(0)
##   .. .. ..@ values    : logi(0)
##   .. .. ..@ color     : logi(0)
##   .. .. ..@ names     : logi(0)
##   .. .. ..@ colortable: logi(0)
##   ..@ title   : chr(0)
##   ..@ extent  :Formal class 'Extent' [package "raster"] with 4 slots
##   .. .. ..@ xmin: num -150
##   .. .. ..@ xmax: num -80
##   .. .. ..@ ymin: num 20
##   .. .. ..@ ymax: num 60
##   ..@ rotated : logi FALSE
##   ..@ rotation:Formal class '.Rotation' [package "raster"] with 2 slots
##   .. .. ..@ geotrans: num(0)
##   .. .. ..@ transfun:function ()
##   ..@ ncols   : int 100
##   ..@ nrows   : int 100
##   ..@ crs     :Formal class 'CRS' [package "sp"] with 1 slot
##   .. .. ..@ projargs: chr "+proj=longlat +datum=WGS84 +no_defs"
##   .. .. ..$ comment: chr "GEOGCRS[\"unknown\",\n    DATUM[\"World Geodetic System 1984\",\n        E
##   ..@ history : list()
##   ..@ z       : list()
```

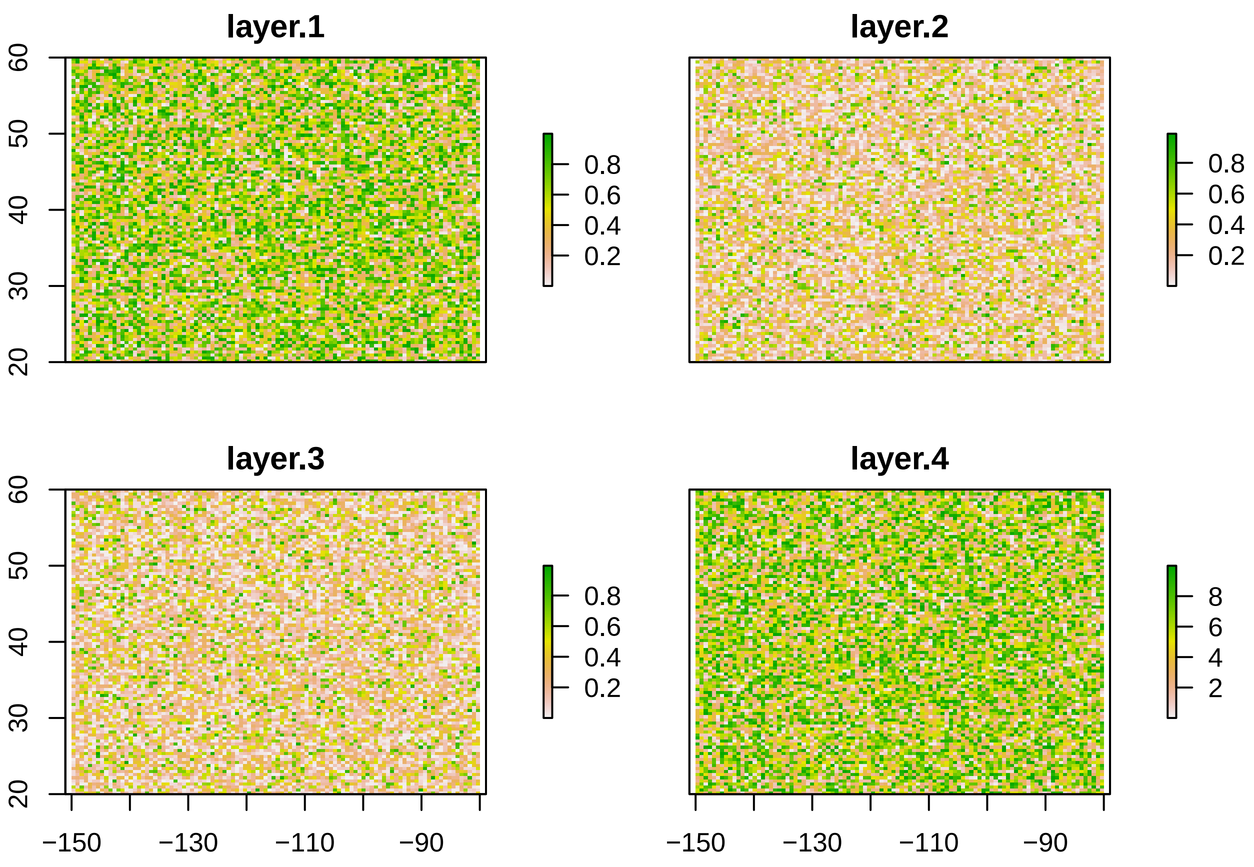
By delineating the extent of the raster, we can make this gridded data comparable with point, line, and polygon data. For instance, the station data we created earlier based on latitude and longitude can be plotted onto this raster object.

```
plot(rast)
# adding polygons
polygon(stations[1:3, 1:2],
    col=adjustcolor('firebrick', 0.25))

# adding lines
lines(stations[3:6, 1:2], lwd=2, col=1)
```



Raster data, as plotted above, correspond to a single z value or covariate (e.g., mean annual temperature), but often we have information on multiple climatic layers. We can keep these layers together by forming what is claled a “stack”.

```r
rastStack <- stack(rast,
    rast*runif(100*100),
    rast*runif(100*100),
    rast*10)

plot(rastStack)
```



This is useful because raster operations can now be done across all the rasters at the same time, much in the way forming objects into a list allows the use of `lapply` and similar functions.

Manipulating raster data

Raster data, as noted above, are in the form of a 2-dimensional array (e.g., a matrix). But we say when we examined the `str()` of a raster object that there was a bit more nuance to this. The developers of `raster` have made it so that the functionality of interacting with matrices in R (at least in terms of indexing) also corresponds to indexing raster objects.

```
rast[1:4]
```

```
## [1] 0.7774452 0.9347052 0.2121425 0.6516738
```

```
rast[1:3,1:3]
```

```
## [1] 0.7774452 0.9347052 0.2121425 0.6401010 0.9918386 0.4955936 0.3399792
## [8] 0.2624741 0.1654539
```

```
# keep the matrix structure
rast[1:3,1:3, drop=FALSE]
```

```
## class       : RasterLayer
## dimensions : 3, 3, 9  (nrow, ncol, ncell)
## resolution : 0.7, 0.4  (x, y)
## extent     : -150, -147.9, 58.8, 60  (xmin, xmax, ymin, ymax)
## crs        : +proj=longlat +datum=WGS84 +no_defs
## source     : memory
## names      : layer
## values     : 0.1654539, 0.9918386  (min, max)
```

Raster-specific commands for data manipulation:

A property representing the bounds of a raster object is the `extent` (mentioned briefly above). This property of the raster can be modified to focus on a particular area of the raster or to simply view the inherited extent of your raster object.

```
extent(rast)
```

```
## class      : Extent
## xmin       : -150
## xmax       : -80
## ymin       : 20
## ymax       : 60
```

```
extent(rast, 1,10,1,10)
```

```
## class      : Extent
## xmin       : -150
## xmax       : -143
## ymin       : 56
## ymax       : 60
```

This can be useful if we want to focus on a specific region of the raster, and specifically if we know which rectangular extent we wish to focus on. The cropping of a raster object is made possible with the `crop` function, which takes the input raster object `x` and an extent function `y`. This means that the second argument (the `y`) must be in the form of an extent object (created usin gthe `extent()` function described above).

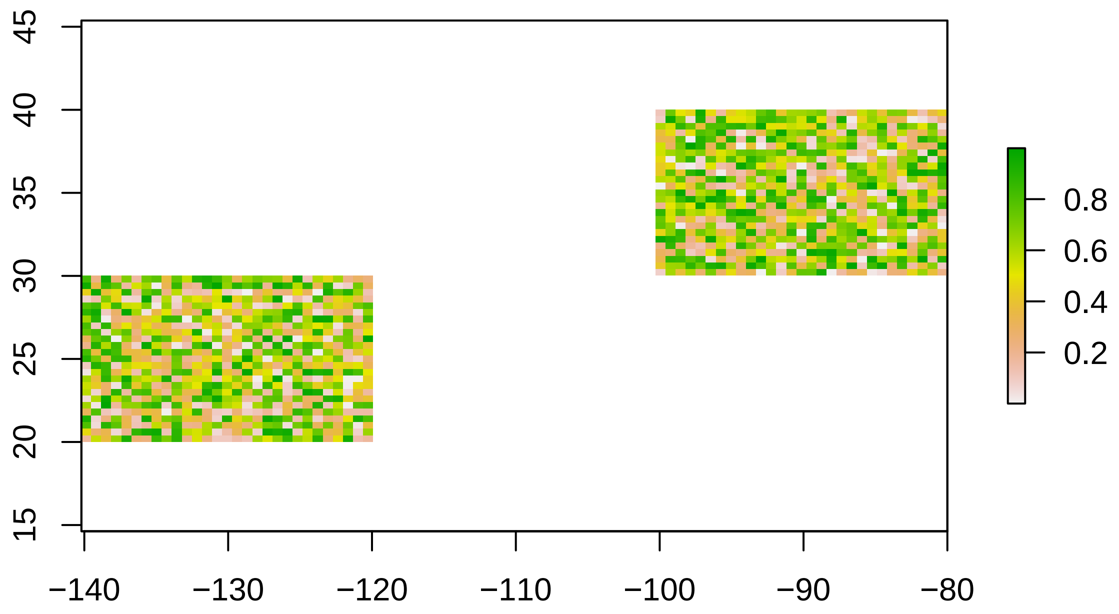
```
crast <- rast
raster::extent(crast)
```

```
## class      : Extent
## xmin       : -150
## xmax       : -80
## ymin       : 20
## ymax       : 60
```

```
crast <- raster::crop(crast, extent(-150,-100,20,60))
```

But while this is incredibly useful for subsetting data down to a specific region or bounding box, we may want to consider merging raster files of different extents. First, it's important to make sure that the units match. This is not *as* big of a deal for raster data as it is for *true* spatial data which has a clear coordinate reference system, as this controls how spatial data are treated and visualized. Recall that raster data are gridded data, so comparing raster data simply requires that they have comparable units for their extents.

```r
r1 <- raster::crop(rast, extent(-140,-120, 0, 30))
r2 <- raster::crop(rast, extent(-100, -80, 30, 40))
m <- raster::merge(r1, r2)
plot(m)
```



A common situation related to spatial data in R is one in which the researcher wants to extract a set of values from a continuous surface of some covariate, where the continuous surface is most often represented as a raster and the set of values is typical a set of coordinates. There are a few different ways to do this, with varying levels of complexity.

Rasters can be subset by their row and column indices:

```r
cells <- raster::cellFromRowCol(rast, 50, 35:39)
cells
```

```
## [1] 4935 4936 4937 4938 4939
```

```r
rast[cells]
```

```
## [1] 0.9400302 0.0569107 0.3174507 0.5291132 0.0458825
```

Or, perhaps more simply, using the `extract` function within the `raster` package. Here, `xyFromCells` provides raster coordinates that correspond to certain cell ids (set above using the `cellFromRowCol` function). Here, `extract` will pull out the values in a raster corresponding to a given set of x and y values. To frame this in a context, this could correspond to extracting values for a specific set of latitude and longitude coordinates from a raster of mean annual temperature.

```r
raster::extract(rast, cells)
```

```
## [1] 0.9400302 0.0569107 0.3174507 0.5291132 0.0458825
```

```
xy <- raster::xyFromCell(rast, cells)
xy
```

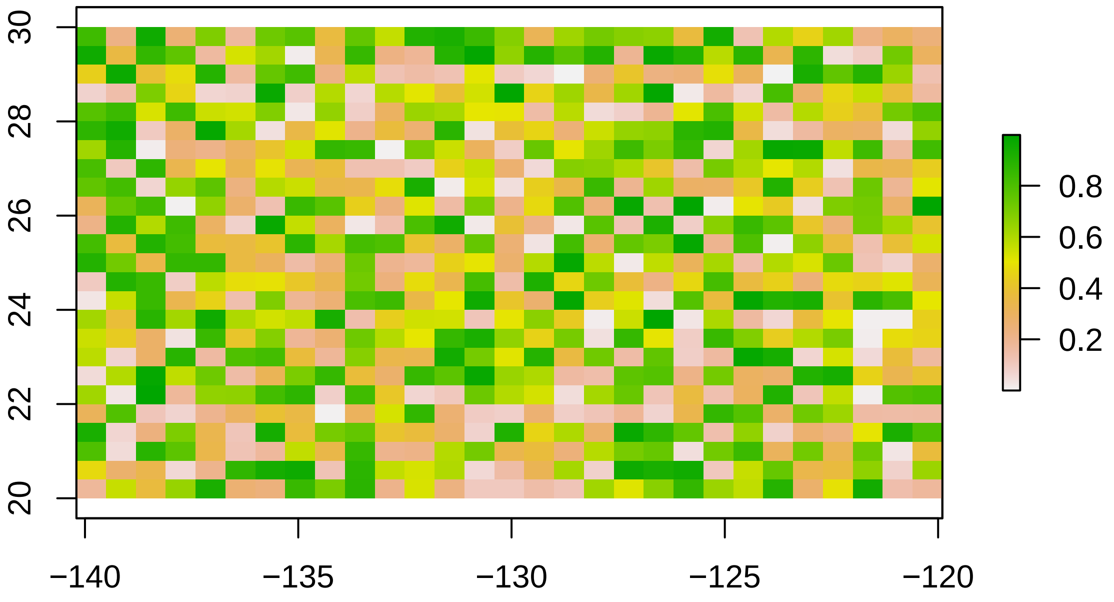
```
##             x    y
## [1,] -125.85 40.2
## [2,] -125.15 40.2
## [3,] -124.45 40.2
## [4,] -123.75 40.2
## [5,] -123.05 40.2
```

```
raster::extract(rast, xy)
```

```
## [1] 0.9400302 0.0569107 0.3174507 0.5291132 0.0458825
```

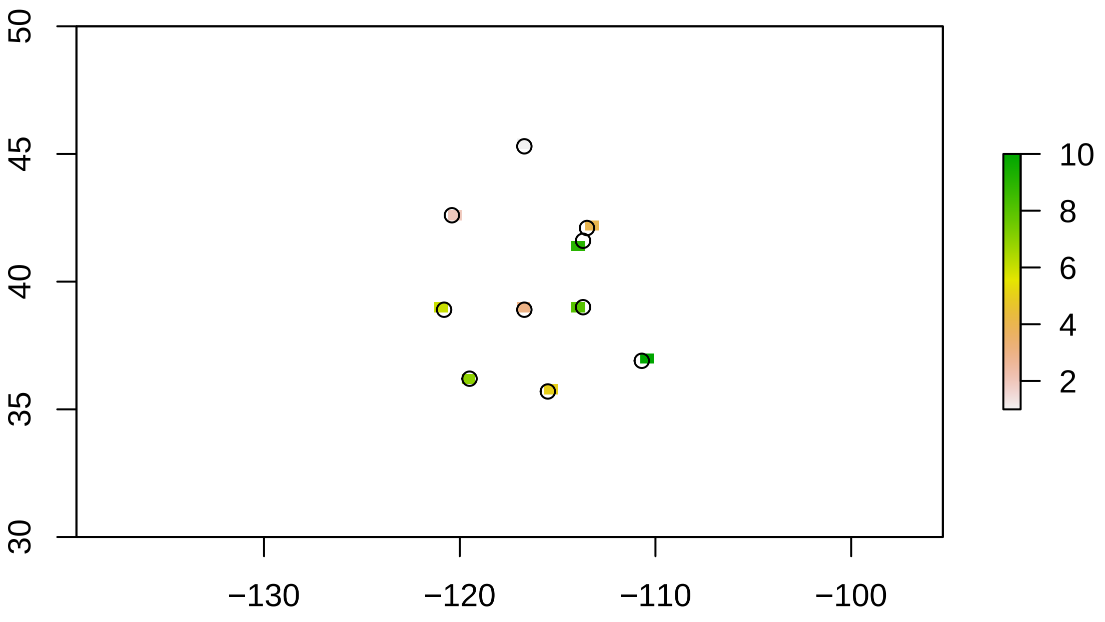
Raster objects may be converted into `SpatialPointsDataFrame` or `SpatialPolygonsDataFrame` objects, which is incredibly useful, but also may introduce some odd resolution issues, as it enforces the resolution of the raster (a lattice) into a continuous data structure.

```
r1sp <- as(r1, 'SpatialPolygonsDataFrame')

plot(r1)
```



Related to this point, in terms of efforts to bridge point/polygon data to raster data, are two sets of functions. First, `rasterize` overlays a lattice on point, line, or polygon data to enforce a raster structure onto more "continuous" spatial data.

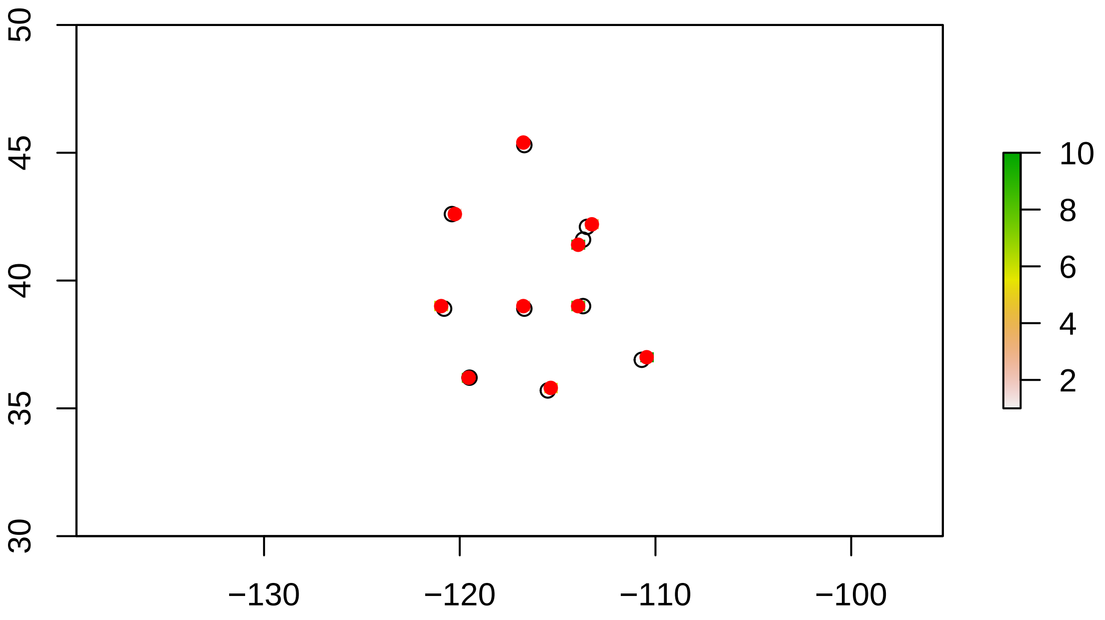
```
rpts <- rasterize(pts, rast)


plot(rpts, xlim=c(-130,-105), ylim=c(30,50))
points(pts)
```



Notice the slight jitter in the above plot. This is because forcing the raster structure on points data influences the relative position of points according to the resolution of the raster (a finer resolution would produce more agreement between point data and rasterized output).

We can also consider outputting the raster data to xy point data or polygon data, which we would use `rasterToPoints` or `rasterToPolygons` functions.

```
pts2 <- raster::rasterToPoints(rpts)

plot(rpts, xlim=c(-130,-105), ylim=c(30,50))
points(pts)
points(pts2, pch=16, col='red')
```



There are also ways to examine the shared overlap between raster objects. However, these methods require the user to have R packages `rgdal` and `rgeos`, both of which require backend libraries `gdal` and `geos`, respectively. It doesn't take long to see that spatial data analyses may suffer from some of the reproducibility concerns that we've gone into thus far.

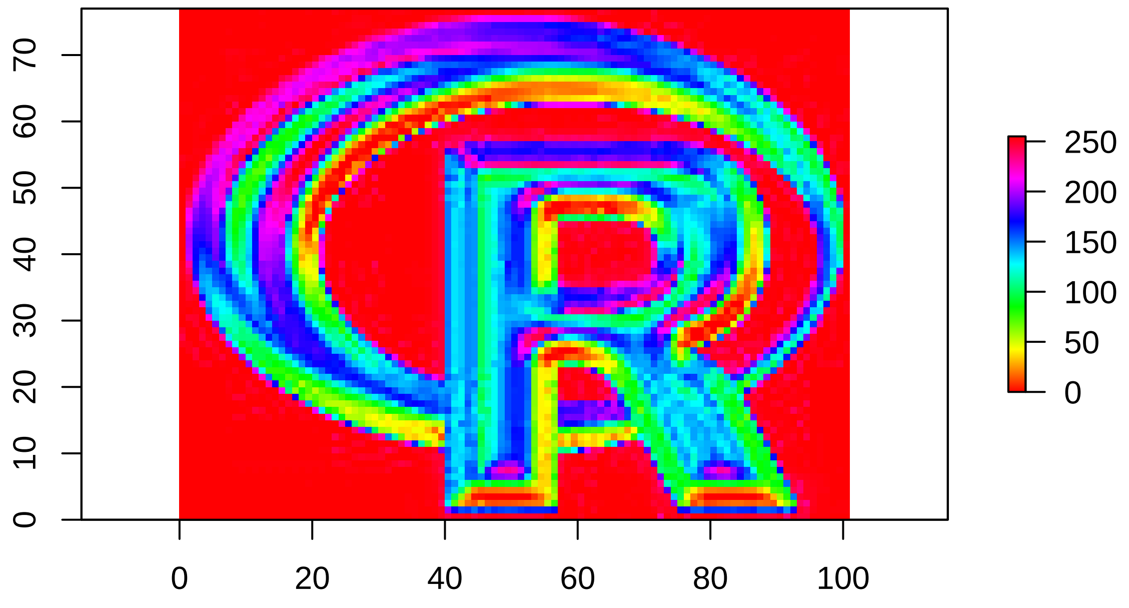
Working with external data

Instead of creating point, line, polygon, or raster data, let us now look at times where we pull data in from external sources. To do this, we will need the package `rgdal`, which may cause some errors when downloading, but this is just how using `R` as a GIS system goes sometimes.

```
#install.packages('rgdal')
library(rgdal)
```

```
## Please note that rgdal will be retired by the end of 2023,
## plan transition to sf/stars/terra functions using GDAL and PROJ
## at your earliest convenience.
##
## rgdal: version: 1.5-28, (SVN revision 1158)
## Geospatial Data Abstraction Library extensions to R successfully loaded
## Loaded GDAL runtime: GDAL 3.0.4, released 2020/01/28
## Path to GDAL shared files: /usr/share/gdal
## GDAL binary built with GEOS: TRUE
## Loaded PROJ runtime: Rel. 6.3.1, February 10th, 2020, [PJ_VERSION: 631]
## Path to PROJ shared files: /usr/share/proj
## Linking to sp version:1.4-6
## To mute warnings of possible GDAL/OSR exportToProj4() degradation,
## use options("rgdal_show_exportToProj4_warnings"="none") before loading sp or rgdal.
```

```
#reading data
f <- raster(system.file("external/rlogo.grd", package="raster"))

plot(f, col=rainbow(1000))
```



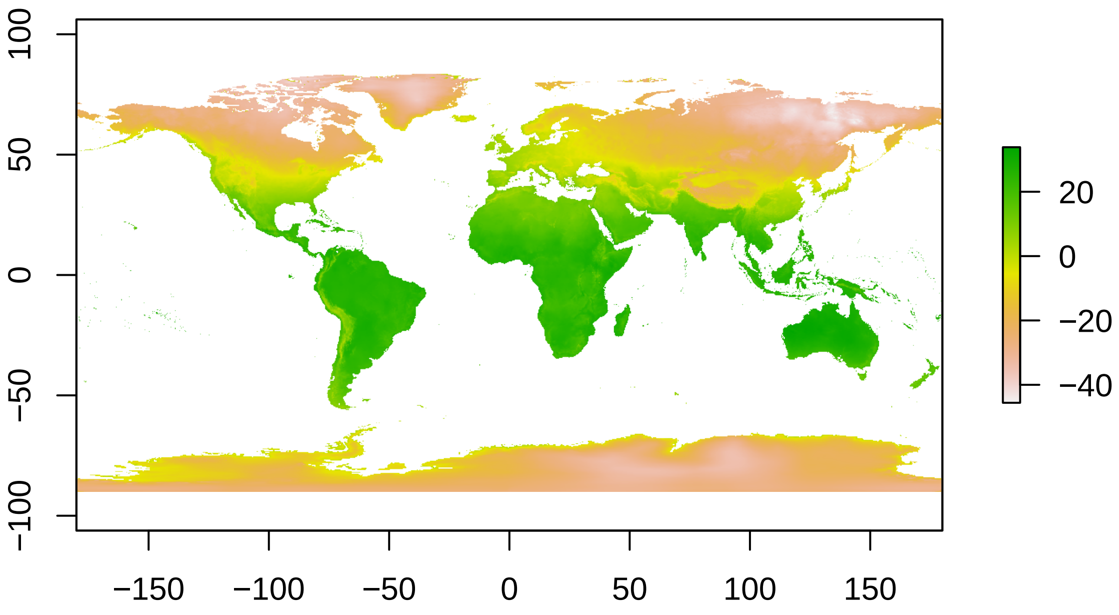
```r
#writing data
raster::writeRaster(f, 'output.tif', overwrite=TRUE)
```

A less trivial example would be working with large-scale climatic data, something which many of us will find ourselves doing at some point. Here, I downloaded a single layer of the WorldClim data corresponding to average global temperature at 10 minute resolution (a coarse resolution corresponding to around 18km cell length and width at the equator).

```r
tavg1 <- raster('tavg/wc2.1_10m_tavg_01.tif')
str(tavg1)
```

```
## Formal class 'RasterLayer' [package "raster"] with 12 slots
##   ..@ file    :Formal class '.RasterFile' [package "raster"] with 13 slots
##   .. .. ..@ name        : chr "/media/tad/sanDisk1TB/Teaching/reproResearchUSC/website/content/code/
##   .. .. ..@ datanotation: chr "FLT4S"
##   .. .. ..@ byteorder   : chr "little"
##   .. .. ..@ nodatavalue : num -Inf
##   .. .. ..@ NAchanged   : logi FALSE
##   .. .. ..@ nbands      : int 1
##   .. .. ..@ bandorder   : chr "BIL"
##   .. .. ..@ offset      : int 0
##   .. .. ..@ toptobottom : logi TRUE
##   .. .. ..@ blockrows   : int 1
##   .. .. ..@ blockcols   : int 2160
##   .. .. ..@ driver      : chr "gdal"
##   .. .. ..@ open        : logi FALSE
##   ..@ data    :Formal class '.SingleLayerData' [package "raster"] with 13 slots
##   .. .. ..@ values    : logi(0)
##   .. .. ..@ offset    : num 0
##   .. .. ..@ gain      : num 1
##   .. .. ..@ inmemory  : logi FALSE
##   .. .. ..@ fromdisk  : logi TRUE
##   .. .. ..@ isfactor  : logi FALSE
##   .. .. ..@ attributes: list()
##   .. .. ..@ haveminmax: logi TRUE
##   .. .. ..@ min       : num -45.9
##   .. .. ..@ max       : num 34
##   .. .. ..@ band      : int 1
##   .. .. ..@ unit      : chr ""
##   .. .. ..@ names     : chr "wc2.1_10m_tavg_01"
##   ..@ legend  :Formal class '.RasterLegend' [package "raster"] with 5 slots
##   .. .. ..@ type      : chr(0)
##   .. .. ..@ values    : logi(0)
##   .. .. ..@ color     : logi(0)
##   .. .. ..@ names     : logi(0)
##   .. .. ..@ colortable: logi(0)
##   ..@ title   : chr(0)
##   ..@ extent  :Formal class 'Extent' [package "raster"] with 4 slots
##   .. .. ..@ xmin: num -180
##   .. .. ..@ xmax: num 180
##   .. .. ..@ ymin: num -90
##   .. .. ..@ ymax: num 90
##   ..@ rotated : logi FALSE
##   ..@ rotation:Formal class '.Rotation' [package "raster"] with 2 slots
##   .. .. ..@ geotrans: num(0)
##   .. .. ..@ transfun:function ()
##   ..@ ncols   : int 2160
##   ..@ nrows   : int 1080
##   ..@ crs     :Formal class 'CRS' [package "sp"] with 1 slot
##   .. .. ..@ projargs: chr "+proj=longlat +datum=WGS84 +no_defs"
##   .. .. ..$ comment: chr "GEOGCRS[\"WGS 84 (with axis order normalized for visualization)\",\n    DA
##   ..@ history : list()
##   ..@ z       : list()
```

```
plot(tavg1)
```

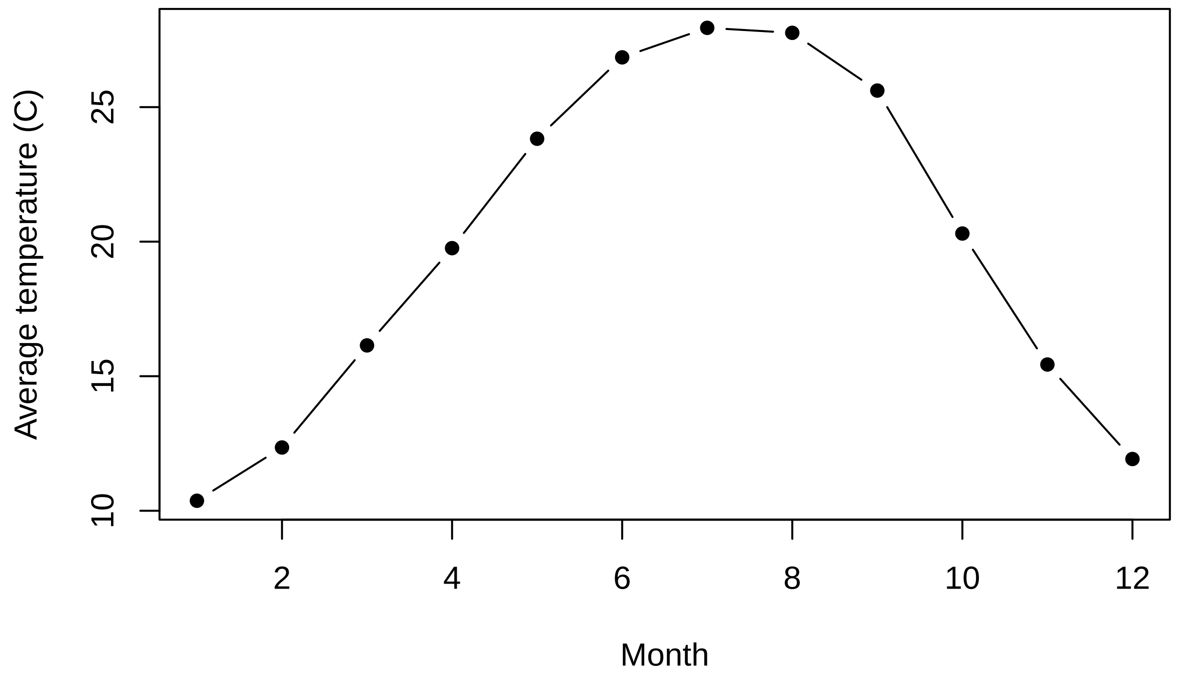


Using a raster stack in a realistic setting.

```
tavg <- raster::stack(
    raster('tavg/wc2.1_10m_tavg_01.tif'),
    raster('tavg/wc2.1_10m_tavg_02.tif'),
    raster('tavg/wc2.1_10m_tavg_03.tif'),
    raster('tavg/wc2.1_10m_tavg_04.tif'),
    raster('tavg/wc2.1_10m_tavg_05.tif'),
    raster('tavg/wc2.1_10m_tavg_06.tif'),
    raster('tavg/wc2.1_10m_tavg_07.tif'),
    raster('tavg/wc2.1_10m_tavg_08.tif'),
    raster('tavg/wc2.1_10m_tavg_09.tif'),
    raster('tavg/wc2.1_10m_tavg_10.tif'),
    raster('tavg/wc2.1_10m_tavg_11.tif'),
    raster('tavg/wc2.1_10m_tavg_12.tif')
)
```

For instance, we can extract specific values from the raster stack of mean temperatures. In this case, we use the `extract` function from the `raster` package to extract temperature data for Baton Rouge, Louisiana. It is hot here.

```r
brTemp <- raster::extract(tavg, data.frame(lon=-91.1, lat=30.47))

plot(1:12, brTemp,
    xlab='Month',
    ylab='Average temperature (C)',
    pch=16, type='b')
```


Average temperature (C)
Month