Fundamentals of Spatial Analysis in R

Marc Weber

2020-08-04

Contents

Chapter 1

Introduction

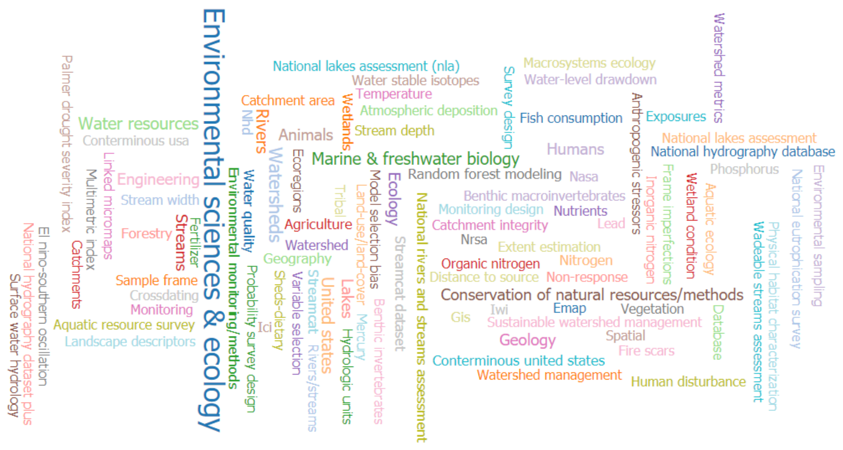

Environmental sciences & ecology
Marine & freshwater biology
Conservation of natural resources/methods
Water resources
Watersheds
Environmental monitoring/methods
National rivers and streams assessment
National lakes assessment (nla)
National hydrography database
National lakes assessment
Wadeable streams assessment
Agriculture
Streams
Rivers
Lakes
Ecology
Geology
Nitrogen
Nutrients
Phosphorus
Humans
Engineering
Forestry
Wetlands
Watershed management
Human disturbance
Conterminous united states
Catchment integrity
Random forest modeling

Figure 1.1: A bit about me

- Workshop agenda
  - Intro and quick review of basic R objects and methods
  - Getting spatial data into R
  - Learning ways to map and visualize spatial data
  - Understanding coordinate reference systems (CRS)!
  - Along the way: overview of select `tidyverse` packages and syntax
    - i.e. `ggplot2`, `dplyr`, `readr`, `tidyr`, the pipe operator %>%
  - Working with vector and raster data in R
  - If this is all new, don't sweat it - google things, we'll answer questions as we go
- This portion of workshop there is no expectation of experience with spatial

analysis in R - if you already have some experience, you are sure to pick up new tricks - if you don't, we'll cover the basics

- We have a limited time to cover a very broad topic, so I'll move quickly - ask questions as they come up, but if you get lost on steps we'll have time for discussion at the end and material will be available to peruse at your own speed later.

1.1 Workshop Packages and Data

- Packages you need installed
  - `sf`
  - `raster`
  - `stars`
  - `tmap`
  - `tmaptools`
  - `dplyr`
  - `devtools`
  - `palmerpenguins`
  - `tigris`
  - `tibble`
  - awra2020spatial - data for this workshop contained in package
- Downloading content via Github
- Using RMarkdown and RStudio

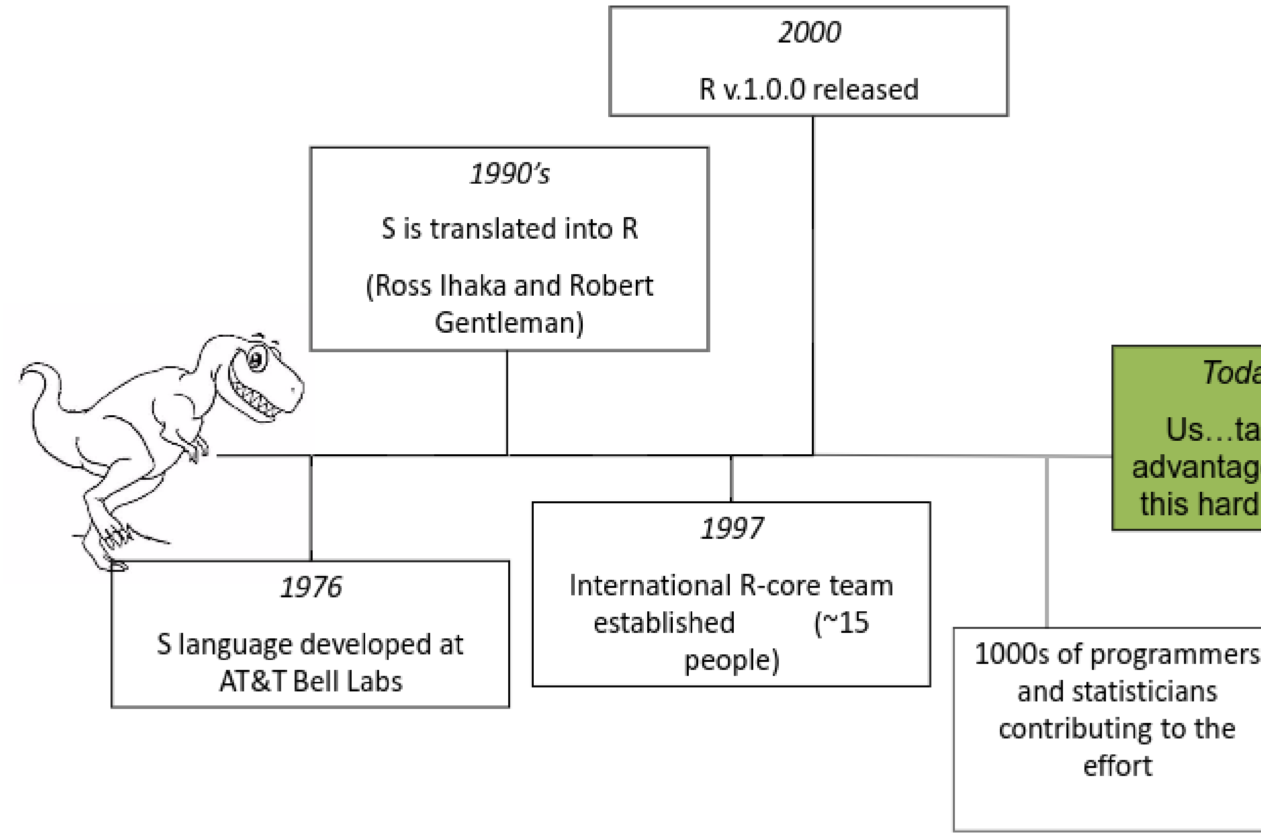
1.1.1 Overview

What is R and why should we use R for spatial analysis? Let's break that into two questions - first, what is R and why should we use it?

- A language and environment for statistical computing and graphics
- R is lightweight, free, open-source and cross-platform
- Works with contributed packages - currently 15,362 - extensibility
- Automation and recording of workflow (reproducibility)
- Optimized work flow - data manipulation, analysis and visualization all in one place
- R does not alter underlying data - manipulation and visualization in memory
- R is great for repetetive graphics

Second, why use R for spatial, or GIS, work?

- Spatial and statistical analysis in one environment
- Leverage statistical power of R (i.e. modeling spatial data, data visualization, statistical exploration)


2000
R v.1.0.0 released
1990's
S is translated into R
(Ross Ihaka and Robert Gentleman)
Toda
Us…ta
advantag
this hard
1997
International R-core team established (~15 people)
1976
S language developed at AT&T Bell Labs
1000s of programmers and statisticians contributing to the effort

of R.bb

Figure 1.2: History of R

- Can handle vector and raster data, as well as work with spatial databases and pretty much any data format spatial data comes in
- R's GIS capabilities growing rapidly right now - new packages added monthly - currently about 200 spatial packages (depending on how you categorize)

Some drawbacks to using R for GIS work

- R not as good for interactive use as desktop GIS applications like ArcGIS or QGIS (i.e. editing features, panning, zooming, and analysis on selected subsets of features)
- Explicit coordinate system handling by the user, no on-the-fly projection support
- In memory analysis does not scale well with large GIS vector and tabular data
- Steep learning curve
- Up to you to find packages to do what you need - help not always great

1.2 R Basics Review

```r
getwd()
```

Which should return something like:

```
[1] "/home/marc/GitProjects/AWRA_GIS_R_Workshop"
```

To see what is in the directory:

```r
dir()
```

```
##  [1] "_after_body.html"          "_book"
##  [3] "_bookdown.yml"             "_bookdown_files"
##  [5] "_output.yml"               "02-crs.Rmd"
##  [7] "03-vector.Rmd"             "04-raster.Rmd"
##  [9] "05-geoprocessing.Rmd"      "06-references.Rmd"
## [11] "AWRA_2020_R_Spatial.log"   "AWRA_2020_R_Spatial.pdf"
## [13] "AWRA_2020_R_Spatial.Rmd"   "AWRA_2020_R_Spatial.Rproj"
## [15] "AWRA_2020_R_Spatial.tex"   "AWRA_2020_R_Spatial_files"
## [17] "book.bib"                  "css"
## [19] "docs"                      "images"
## [21] "index.Rmd"                 "js"
## [23] "packages.bib"              "preamble.tex"
## [25] "README.md"
```

To establish a different directory:

```
setwd("/home/marc/GitProjects")
```

1.2.0.1 Terminology: data structures

R is an interpreted language (access through a command-line interpreter) with a number of data structures (vectors, matrices, arrays, data frames, lists) and extensible objects (regression models, time-series, geospatial coordinates) and supports procedural programming with functions.

To learn about objects, become friends with the built-in `class` and `str` functions. Let's explore a new dataset - palmerpenguins - recently developed by Allison Horst as an alternative to the old R standby `iris` dataset:

```r
library(palmerpenguins)
class(penguins)
```

```
## [1] "tbl_df"     "tbl"        "data.frame"
```

```r
str(penguins)
```

```
## tibble [344 x 8] (S3: tbl_df/tbl/data.frame)
##  $ species          : Factor w/ 3 levels "Adelie","Chinstrap",..: 1 1 1 1 1 1 1 1 1 1 ...
##  $ island           : Factor w/ 3 levels "Biscoe","Dream",..: 3 3 3 3 3 3 3 3 3 3 ...
##  $ bill_length_mm   : num [1:344] 39.1 39.5 40.3 NA 36.7 39.3 38.9 39.2 34.1 42 ...
##  $ bill_depth_mm    : num [1:344] 18.7 17.4 18 NA 19.3 20.6 17.8 19.6 18.1 20.2 ...
##  $ flipper_length_mm: int [1:344] 181 186 195 NA 193 190 181 195 193 190 ...
##  $ body_mass_g      : int [1:344] 3750 3800 3250 NA 3450 3650 3625 4675 3475 4250 ...
##  $ sex              : Factor w/ 2 levels "female","male": 2 1 1 NA 1 2 1 2 NA NA ...
##  $ year             : int [1:344] 2007 2007 2007 2007 2007 2007 2007 2007 2007 2007 ...
```

`penguins` is a tibble and was created as a new alternative to the `iris` data set that has been used extensively for beginning tutorials on learning R. `penguins` as you can see is a tibble, which is a new `tidyverse` spin on `data frames`. Data frames consist of rows of observations on columns of values for variables of interest - they are one of the fundamental and most important data structures in R. `tibbles` make a few improvements / changes to `data frames` such as:

- Never converting strings to factors
- Never creating row names
- Updated print method that only shows first 10 rows, just the columns that fit on the screen, and the type of each column (just as we get using `str`)

We can easily convert objects from `tibble` to `data frame` and vice versa:

```r
library(tibble)
penguins <- as.data.frame(penguins)
penguins <- as_tibble(penguins)
```

But as we see in the result of `str(penguins)` above, following the information that `penguins` is a `tibble` with 344 observations of 8 variables, we get information on each of the variables, in this case that 2 are numeric, 2 are integers, and 3 are factors - factors encode categorical variables - and `str` gives us the number of levels in each factor.

First off, R has several main data types:

- logical
- integer
- double
- complex
- character
- raw
- list
- NULL
- closure (function)
- special
- builtin (basic functions and operators)
- environment
- S4 (some S4 objects)
- others you won't run into at user level

We can ask what data type something is using `typeof`:

```
typeof(penguins)
```

```
## [1] "list"
```

```
[1] "list"
```

```
typeof(penguins$bill_length_mm)
```

```
## [1] "double"
```

```
[1] "double"
```

```
typeof(penguins$specis)
```

```
## Warning: Unknown or uninitialised column: `specis`.
```

```
## [1] "NULL"
```

```
[1] "integer"
```

We see a couple interesting things here - `penguins`, which we just said is a `tibble`, is a data type of `list`. `bill_length_mm` is data type `double`, and in `str(penguins)` we saw it was numeric - that makes sense - but we see that `species` is data type `integer`, and in `str(penguins)` we were told this variable was a factor with three levels. What's going on here?

First off, `class` refers to the abstract type of an object in R, whereas `typeof` or `mode` refer to how an object is stored in memory. So `penguins` is an object of class `tibble`, but it is stored in memory as a list (i.e. each column is an item in a list). Note that this allows tibbles and data frames to have columns of different classes, whereas a matrix needs to be all of the same mode.

For our `species` column, we see it's `mode` is numeric, it's `typeof` is `integer`, and it's class is `factor`. Nominal variables in R are treated as a vector of integers 1:k, where k is the number of unique values of that nominal variable and a mapping of the character strings to these integer values.

This allows us to quickly see see all the unique values of a particular nominal variable or quickly re-asign a level of a nominal variable to a new value - remember, everything in R is in memory, so don't worry about tweaking the data!

```
levels(penguins$species)
```

```
## [1] "Adelie"    "Chinstrap" "Gentoo"
```

```
levels(penguins$species)[1] <- 'adeliae'
```

See if you can explain how that re-asignment we just did worked.

To access particular columns in a `tibble` or `data frame`, as we saw above, we use the `$` operatoe. We can see the value of `species` for each observation in `penguins` as well as listing of all levels of the variable by running:

```
penguins$species
```

```
##   [1] adeliae   adeliae   adeliae   adeliae   adeliae   adeliae   adeliae
##   [8] adeliae   adeliae   adeliae   adeliae   adeliae   adeliae   adeliae
##  [15] adeliae   adeliae   adeliae   adeliae   adeliae   adeliae   adeliae
##  [22] adeliae   adeliae   adeliae   adeliae   adeliae   adeliae   adeliae
##  [29] adeliae   adeliae   adeliae   adeliae   adeliae   adeliae   adeliae
##  [36] adeliae   adeliae   adeliae   adeliae   adeliae   adeliae   adeliae
##  [43] adeliae   adeliae   adeliae   adeliae   adeliae   adeliae   adeliae
##  [50] adeliae   adeliae   adeliae   adeliae   adeliae   adeliae   adeliae
##  [57] adeliae   adeliae   adeliae   adeliae   adeliae   adeliae   adeliae
##  [64] adeliae   adeliae   adeliae   adeliae   adeliae   adeliae   adeliae
##  [71] adeliae   adeliae   adeliae   adeliae   adeliae   adeliae   adeliae
##  [78] adeliae   adeliae   adeliae   adeliae   adeliae   adeliae   adeliae
##  [85] adeliae   adeliae   adeliae   adeliae   adeliae   adeliae   adeliae
##  [92] adeliae   adeliae   adeliae   adeliae   adeliae   adeliae   adeliae
##  [99] adeliae   adeliae   adeliae   adeliae   adeliae   adeliae   adeliae
## [106] adeliae   adeliae   adeliae   adeliae   adeliae   adeliae   adeliae
## [113] adeliae   adeliae   adeliae   adeliae   adeliae   adeliae   adeliae
## [120] adeliae   adeliae   adeliae   adeliae   adeliae   adeliae   adeliae
## [127] adeliae   adeliae   adeliae   adeliae   adeliae   adeliae   adeliae
## [134] adeliae   adeliae   adeliae   adeliae   adeliae   adeliae   adeliae
## [141] adeliae   adeliae   adeliae   adeliae   adeliae   adeliae   adeliae
## [148] adeliae   adeliae   adeliae   adeliae   adeliae   Gentoo    Gentoo
## [155] Gentoo    Gentoo    Gentoo    Gentoo    Gentoo    Gentoo    Gentoo
## [162] Gentoo    Gentoo    Gentoo    Gentoo    Gentoo    Gentoo    Gentoo
## [169] Gentoo    Gentoo    Gentoo    Gentoo    Gentoo    Gentoo    Gentoo
## [176] Gentoo    Gentoo    Gentoo    Gentoo    Gentoo    Gentoo    Gentoo
## [183] Gentoo    Gentoo    Gentoo    Gentoo    Gentoo    Gentoo    Gentoo
## [190] Gentoo    Gentoo    Gentoo    Gentoo    Gentoo    Gentoo    Gentoo
## [197] Gentoo    Gentoo    Gentoo    Gentoo    Gentoo    Gentoo    Gentoo
## [204] Gentoo    Gentoo    Gentoo    Gentoo    Gentoo    Gentoo    Gentoo
## [211] Gentoo    Gentoo    Gentoo    Gentoo    Gentoo    Gentoo    Gentoo
## [218] Gentoo    Gentoo    Gentoo    Gentoo    Gentoo    Gentoo    Gentoo
## [225] Gentoo    Gentoo    Gentoo    Gentoo    Gentoo    Gentoo    Gentoo
## [232] Gentoo    Gentoo    Gentoo    Gentoo    Gentoo    Gentoo    Gentoo
## [239] Gentoo    Gentoo    Gentoo    Gentoo    Gentoo    Gentoo    Gentoo
## [246] Gentoo    Gentoo    Gentoo    Gentoo    Gentoo    Gentoo    Gentoo
## [253] Gentoo    Gentoo    Gentoo    Gentoo    Gentoo    Gentoo    Gentoo
## [260] Gentoo    Gentoo    Gentoo    Gentoo    Gentoo    Gentoo    Gentoo
## [267] Gentoo    Gentoo    Gentoo    Gentoo    Gentoo    Gentoo    Gentoo
## [274] Gentoo    Gentoo    Gentoo    Chinstrap Chinstrap Chinstrap Chinstrap
## [281] Chinstrap Chinstrap Chinstrap Chinstrap Chinstrap Chinstrap Chinstrap
## [288] Chinstrap Chinstrap Chinstrap Chinstrap Chinstrap Chinstrap Chinstrap
## [295] Chinstrap Chinstrap Chinstrap Chinstrap Chinstrap Chinstrap Chinstrap
## [302] Chinstrap Chinstrap Chinstrap Chinstrap Chinstrap Chinstrap Chinstrap
## [309] Chinstrap Chinstrap Chinstrap Chinstrap Chinstrap Chinstrap Chinstrap
## [316] Chinstrap Chinstrap Chinstrap Chinstrap Chinstrap Chinstrap Chinstrap
## [323] Chinstrap Chinstrap Chinstrap Chinstrap Chinstrap Chinstrap Chinstrap
## [330] Chinstrap Chinstrap Chinstrap Chinstrap Chinstrap Chinstrap Chinstrap
## [337] Chinstrap Chinstrap Chinstrap Chinstrap Chinstrap Chinstrap Chinstrap
## [344] Chinstrap
## Levels: adeliae Chinstrap Gentoo
```

To access particular columns or rows of a data frame, we use indexing:

```
penguins[1,3] # the 1st row and the 3rd column
```

```
## # A tibble: 1 x 1
##   bill_length_mm
##            <dbl>
## 1           39.1
```

```
[1] 39.1
```

```
penguins[4,1] # the 4th row and the 1st column
```

```
## # A tibble: 1 x 1
##   species
##   <fct>
## 1 adeliae
```

```
<fct>
1 Adelie
```

A handy function is `names`, which you can use to get or to set data frame variable names:

```
names(penguins)
```

```
## [1] "species"           "island"            "bill_length_mm"
## [4] "bill_depth_mm"     "flipper_length_mm" "body_mass_g"
## [7] "sex"               "year"
```

```
names(penguins)[3] <- 'Bill Length'
```

Explain what this last line did

A little example of tidy evaluation and piping to do the same thing - we'll go into more:

```
penguins <- penguins %>%
  dplyr::rename('Bill_Length'='Bill Length')
# check it
names(penguins)
```

```
## [1] "species"           "island"            "Bill_Length"
## [4] "bill_depth_mm"     "flipper_length_mm" "body_mass_g"
## [7] "sex"               "year"
```

1.2.0.2 Review of Classes and Methods

- Class: object types
  - `class()`: gives the class type
  - `typeof()`: information on how the object is stored
  - `str()`: how the object is structured
- Method: generic functions
  - `print()`
  - `plot()`
  - `summary()`

1.3 Spatial Data in R

We can represent spatial data as discrete locations (points, lines or polygons) or as a grid of values rendered on a map as pixels. We typically represent the former type of data (discrete locations) as *vector* data, with an associated geometry or shape, and some attributes with information about the locations. Examples are:

- state boundaries with state name and population
- rivers with their flow volume and names
- polygons of watersheds with their names and associated landscape information

We represent the latter type of data (a grid of values as pixels) with *rasters*. Rasters can be continous (i.e. elevation, precipitation, atmospheric deposition) or they can be categorical (i.e. land use, soil type) - they can also be image based rasters, and they can be single band or multi-band.

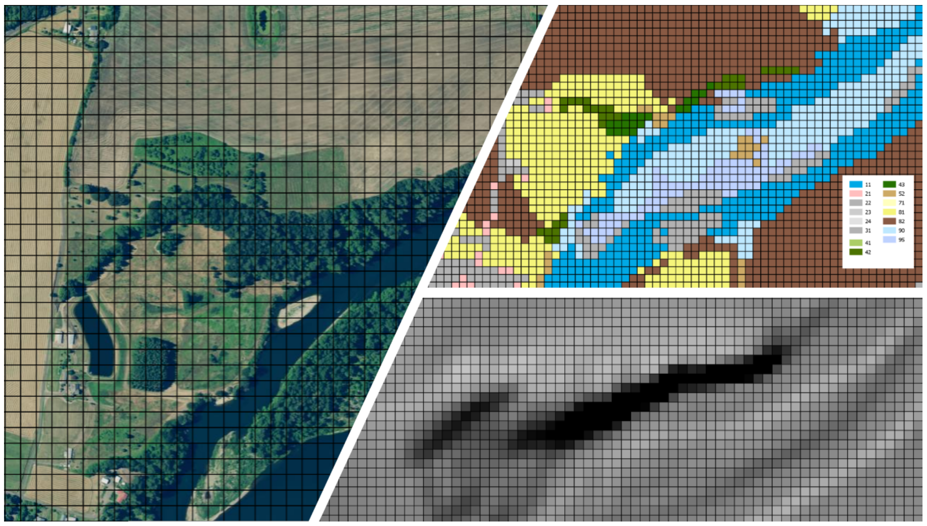


Figure 1.3: Raster Data

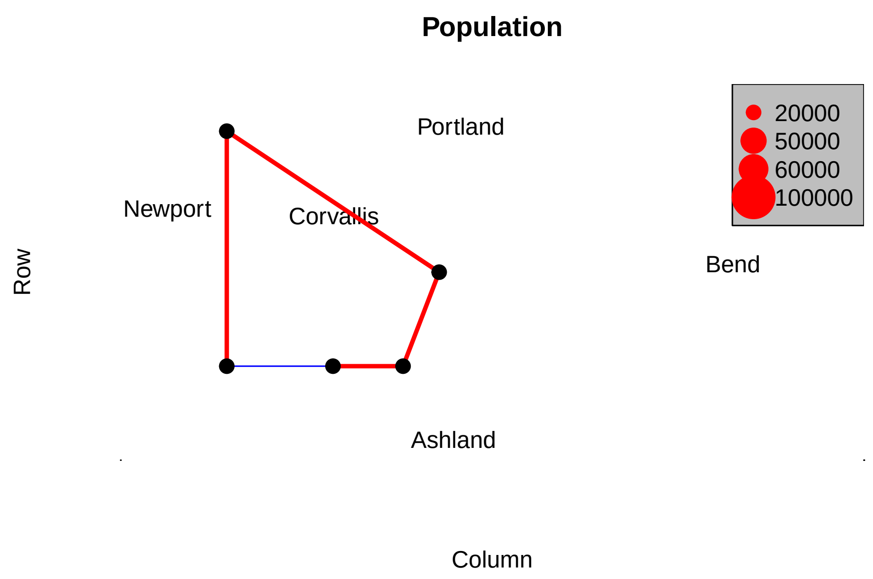
We will delve into working with each of these types of data in their own sections, but let's go over how these spatial data types are handled in R briefly.

Basic data structures in R can represent spatial data - all we need is some vectors with location and attribute information - below we generate cites with population, add a polygon, and make a map with a legend - take a minute to run this code in your own R session and make sure you understand what each line is doing.

```
cities <- c('Ashland','Corvallis','Bend','Portland','Newport')
longitude <- c(-122.699, -123.275, -121.313, -122.670, -124.054)
latitude <- c(42.189, 44.57, 44.061, 45.523, 44.652)
population <- c(20062,50297,61362,537557,9603)
locs <- cbind(longitude, latitude)
plot(locs, cex=sqrt(population*.0002), pch=20, col='red',
  main='Population', xlim = c(-124,-120.5), ylim = c(42, 46))
text(locs, cities, pos=4)

# Add a legend
breaks <- c(20000, 50000, 60000, 100000)
options(scipen=3)
legend("topright", legend=breaks, pch=20, pt.cex=1+breaks/20000,
  col='red', bg='gray')
```

```
# Add polygon
lon <- c(-123.5, -123.5, -122.5, -122.670, -123)
lat <- c(43, 45.5, 44, 43, 43)
x <- cbind(lon, lat)
polygon(x, border='blue')
lines(x, lwd=3, col='red')
points(x, cex=2, pch=20)
```



spatial data structures vector-1.bb

We can see in this toy example that numeric vectors can represent locations in R for simple mapping. Points just need to be a pair of numbers in cartesian space, and lines and polygons are just a number of these points (note that polygons are closed by having their first point coincide with last point which the polygon function in base R graphics takes care of).

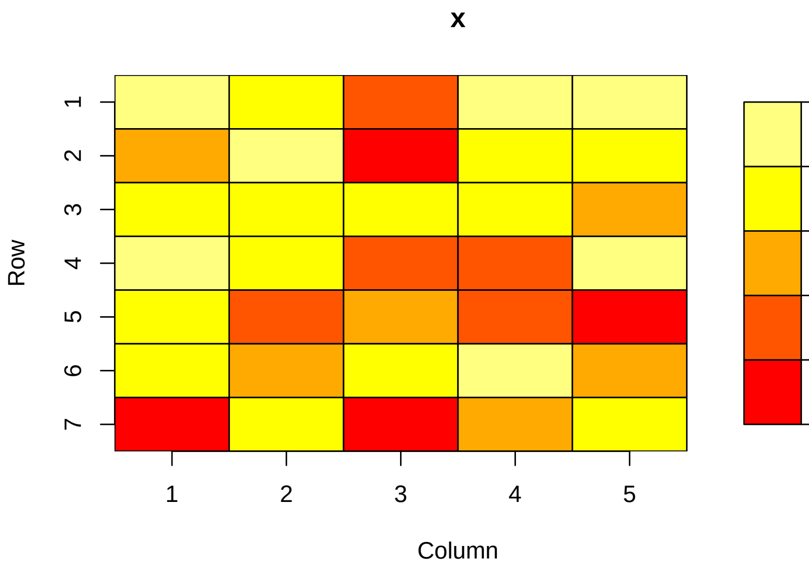
What about raster data? A raster in essence is simply a matrix of values for raster cells

```
library('plot.matrix')
# numeric matrix
x <- matrix(runif(35), ncol=5) # create a numeric matrix object
class(x)
```

```
## [1] "matrix" "array"
```

```
#> [1] "matrix"
par(mar=c(5.1, 4.1, 4.1, 4.1)) # adapt margins
```

```r
plot(x)
```



spatial data structures raster-1.bb

We have a ‘raster’ of cell values in columnns and rows - but what is lacking?

You can do simple things with these spatial representations using basic R structures, but it breaks down quickly if you want to ask any spatial questions - for instance using the first example above, how would we figure out the nearest city to Corvallis? Or imagine the polygon is a county and we wanted to know what cities are within the county? Or how would we superimpose our cities on our raster cells? Or extract the value of a cell at the location of a city?

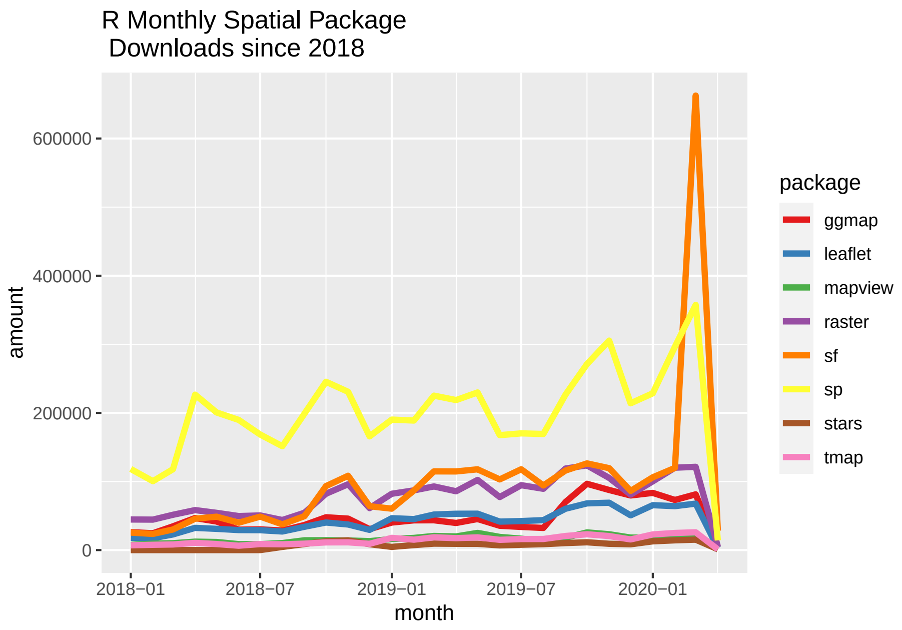
1.3.1 Challenge

What information do we need to properly define spatial vector data and perform spatial operations?

1.3.2 Answer

- A coordinate reference system (CRS)
- A bounding box or extent
- Methods for storing and accessing spatial attributes of data
- ?

1.3.3 R Spatial Package Landscape



packages-1.bb

1.3.4 How did we make this figure?

Look at answer below, run code in your own code editor, see if it makes sense and ask questions

1.3.5 Answer

```
# devtools::install_github("metacran/cranlogs")
library(cranlogs, quietly = T)
library(ggplot2, quietly = T)
library(lubridate, quietly = T)
library(dplyr, quietly = T)
options(scipen=3)
stats <- cran_downloads(from = "2018-01-01", to = "2020-04-01",packages = c("sp", "sf", "raster",

monthly_stats <- stats %>%
  group_by(month=floor_date(date, "month"), package) %>%
  summarize(amount=sum(count))
ggplot(monthly_stats, aes(x=month, y=amount, group = package, colour = package)) + geom_line(size
```

Figure 1.4: Moving on from ‘sp‘

1.3.6 Primary R spatial packages

`sp` was the core vector spatial data package in R for a number of years, and while still used and while many packages still depend on the package, we will make a clean break in this workshop and focus entirely on the new `sf` package for working with vector data.

This portion of the workshop will focus primarily on the following core packages for working with spatial data in R: - `sf` - The core package for working with vector data in R - `raster` - Still the primary spatial package for working with raster data in R - `mapview` - this is a wrapper package for R `leaflet` package and I find simpler and more intuitive - `ggolot` and `tmap` - static plotting and thematic maping - `dplyr` - Not a spatial package, but `sf` is “tidy-compliant” and we will follow “tidy” workflows in this portion of the workshop for much or our examples

The R Spatial Task View page provides current and comprehensive information on the ecosystem of R packages for working with spatial data - there are hundreds out there for specific tasks - we’ll touch on several others besides core packages above this morning.

1.4 Quick examples

Here is just a sampling of few quick spatial tasks.

1.4.1 Geocoding example with tmaptools using open street map

Here we'll use the `tmap` package and `tmaptools` to 'geocode' a named feature in OpenStreetMap import it into our R session as an `sf` feature.

```
# uses OSM
library(tmap)
library(tmaptools)
library(dplyr)
tex_cap <-tmaptools::geocode_OSM("Texas Capital",
        as.sf = TRUE) %>%
  glimpse()
```

```
## Rows: 1
## Columns: 9
## $ query   <chr> "Texas Capital"
## $ lat     <dbl> -31.46748
## $ lon     <dbl> -64.22844
## $ lat_min <dbl> -31.46748
## $ lat_max <dbl> -31.46748
## $ lon_min <dbl> -64.22995
## $ lon_max <dbl> -64.22723
## $ bbox    <POLYGON [°]> POLYGON ((-64.22995 -31.467...
## $ point   <POINT [°]> POINT (-64.22844 -31.46748)
```

1.4.2 Run and examine code chunk above

1. What is the double colon doing?
2. What is the `geocode_OSM` function doing?
3. Explain how the code runs together using the `%>%` chaining operator
4. What is `glimpse`? Is it useful compared to `head` function?

1.4.3 Answer

1. It specifies using `geocode_OSM` from the `tmaptools` package. R gives namespace preference to packages in order loaded; some packages share function names; so it's good practice to disambiguate your functions with the double-colon
2. It is looking up a named feature in OpenStreetMap and returning the coordinates (bonus - we'll delve into more in next section - what coordinate reference system are coordinates in and how to you find out?)
3. You would translate code using the `>%>` operator from:

- do this `%>%` do that `%>%` do that

To