Introduction to Spatial Data Programming - R

Lesson 01
The R environment

Michael Dorman
Geography and Environmental Development
dorman@post.bgu.ac.il

Last updated: 2018-12-20 17:00:46

Ben-Gurion University of the Negev

Contents

General information

- **Course number**: 128.1.0043
- **Time**: Sunday 16:10-19:00
- **Place**: Building 72, room 249
- **Instructor**: Michael Dorman (dorman@post.bgu.ac.il)
- **Grading**: 6 exercises (50%) + Exam (50%)
- **Requirements** -
  - Basic knowledge of GIS (e.g. "Intro to GIS" course)
  - Self study
- **Getting help** -
  - Forum on **Moodle** (http://moodle2.bgu.ac.il)
  - Meetings (Monday 15:00-16:00, schedule by e-mail)

Course aims

- General knowledge in programming
- Overview of spatial data processing and analysis in R

Course topics

Part I - Introduction to R programming

1. The R environment
2. Vectors
3. Time series + function definition
4. Tables + conditionals and loops

Part II - Processing and analysis of spatial data in R

5. Matrices and rasters
6. Raster algebra
7. Vector layers
8. Geometric operations with vector layers
9. Geometric operations with rasters
10. Working with spatio-temporal data
11. Combining rasters and vector layers
12. Spatial interpolation of point data

What is R?

- **R** is a **programming language** and **environment**, originally developed for **statistical computing** and **graphics**
- As of September 2018 there are >13,000 R **packages** in the official repository (CRAN)[1]
- **Advantages** -
  - A programming language, yet relatively simple
  - Over 100,000 functions from various areas of interest

[1] https://cran.r-project.org/web/packages/

Ranking

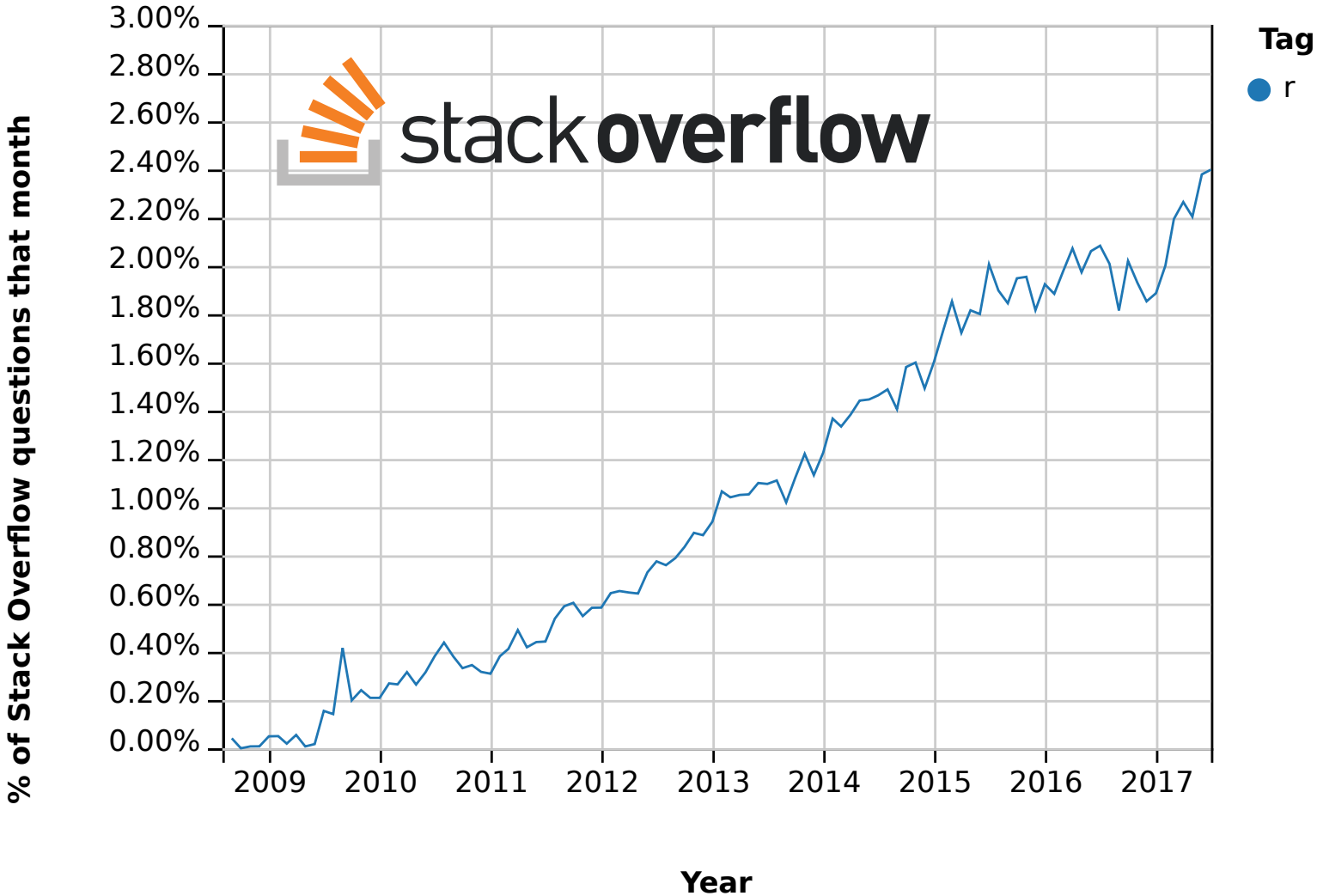


Figure 1: Stack Overflow Trend for the 'r' question tag[2]

[2] https://insights.stackoverflow.com/trends?tags=r

Ranking

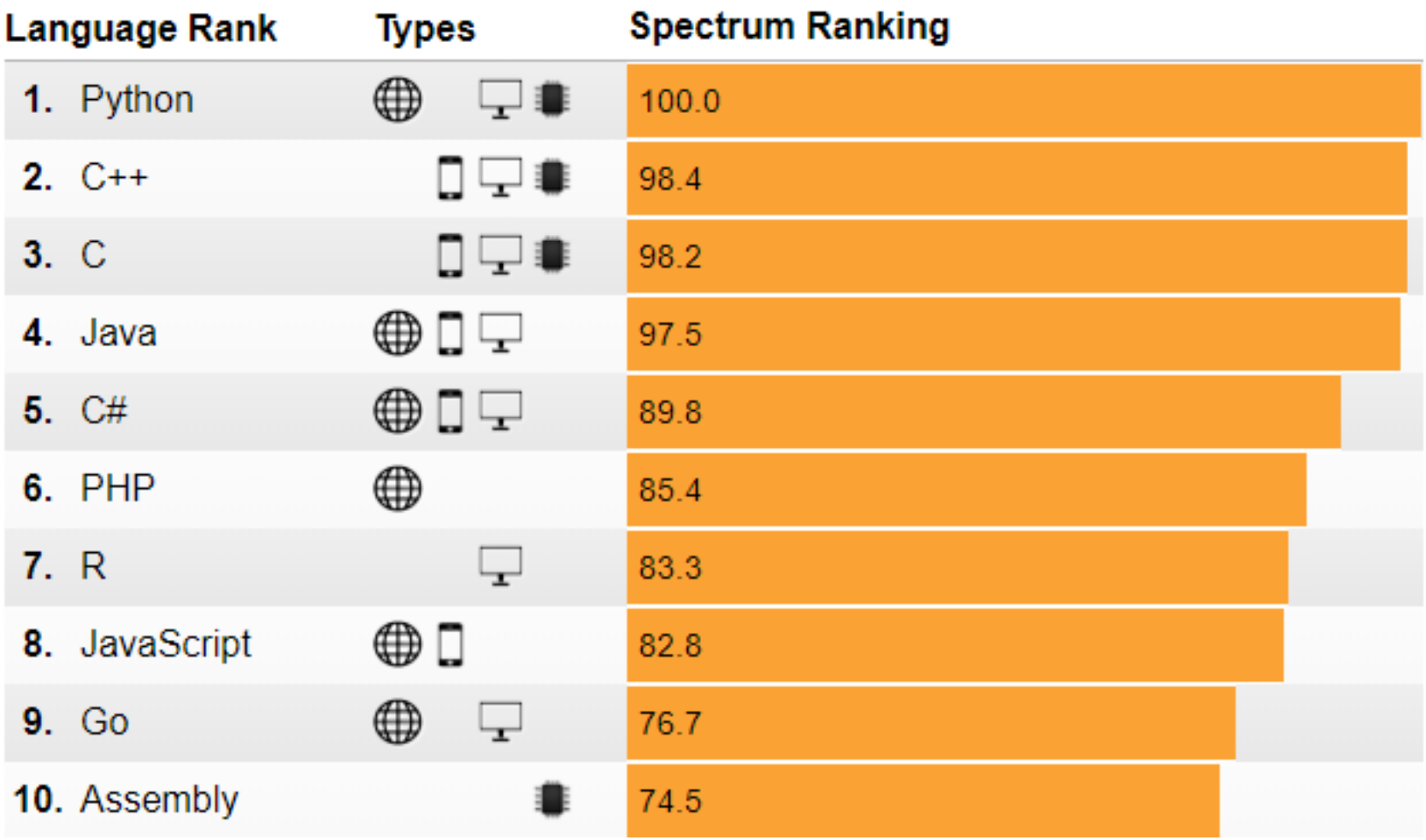


Figure 2: IEEE Language Rankings 2018[3]

[3] http://blog.revolutionanalytics.com/popularity/

Ranking

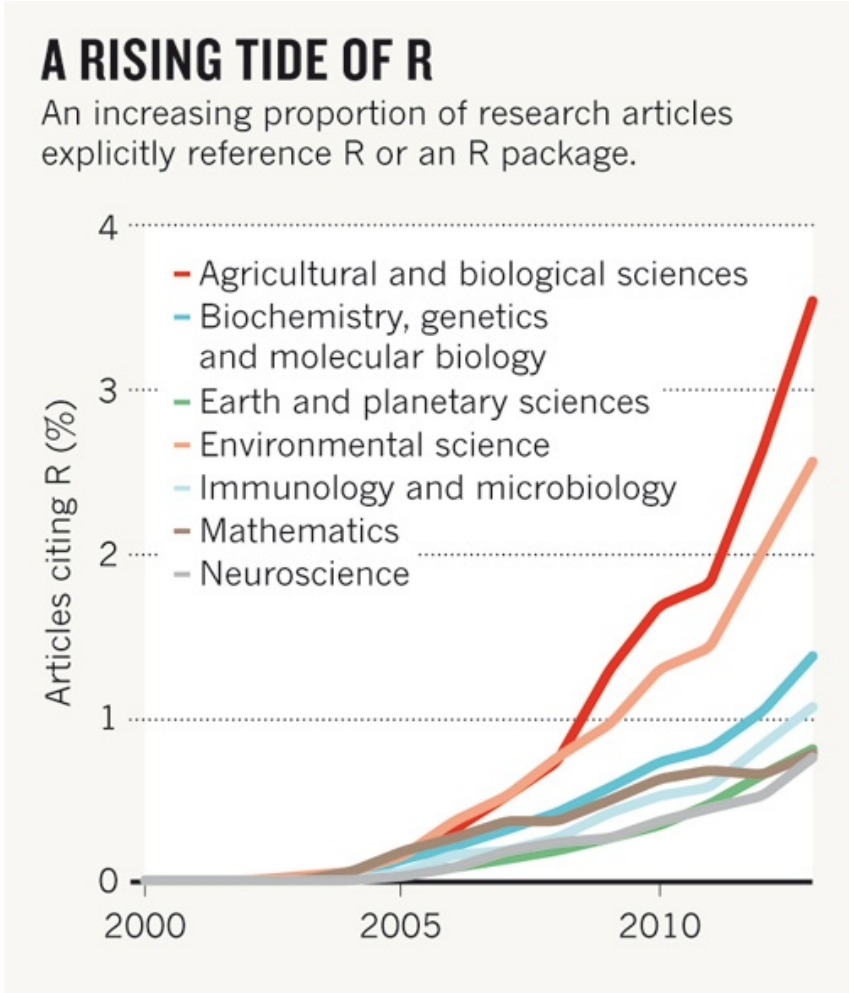


Figure 3: Proportion of research papers citing R[4]

[4] https://www.nature.com/news/programming-tools-adventures-with-r-1.16609

Task views

- https://cran.r-project.org/web/views/
- http://www.maths.lancs.ac.uk/~rowlings/R/TaskViews/

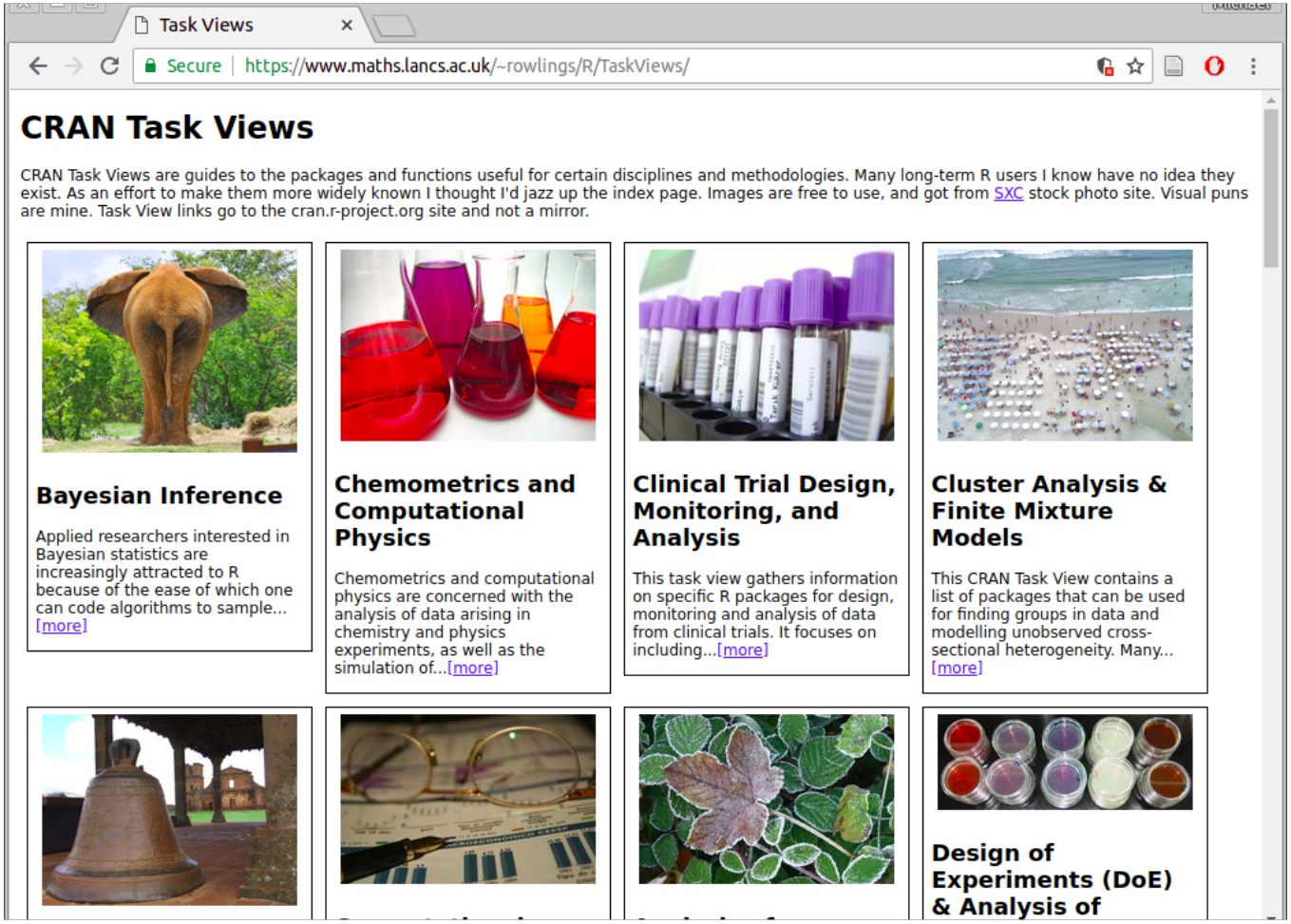


Figure 4: CRAN Task Views

R and analysis of spatial data

- Over time, there was an increasing number of **contributed packages** for handling and analyzing spatial data in R
- Today, spatial analysis is a **major functionality in R**
- As of September 2018, there are ~**180 packages**[5] specifically addressing spatial analysis in R

Figure 5: Books on Spatial Data Analysis with R

[5] https://cran.r-project.org/web/views/Spatial.html

R and analysis of spatial data

- pre-2003: Variable and incomplete approaches (`MASS`, `spatstat`, `maptools`, `geoR`, `splancs`, `gstat`, ...)
- 2003: Consensus that a package defining standard data structures should be useful; `rgdal` released on CRAN[6]
- 2005: `sp` released on CRAN; `sp` support in `rgdal`
- 2008: **Applied Spatial Data Analysis with R**, 1st ed.
- 2010: `raster` released on CRAN
- 2011: `rgeos` released on CRAN
- 2013: **Applied Spatial Data Analysis with R**, 2nd ed.
- 2016: `sf` released on CRAN
- 2018: `stars` released on CRAN
- 2019(?): **Geocomputation with R**[7], 1st ed.

[6] Comprehensive R Archive Network

[7] https://geocompr.robinlovelace.net/

R as a Geographic Information System (GIS)?

- **General** advantages of Command Line Interface (CLI) software
  - **Automation** - Doing otherwise unfeasible repetitive tasks
  - **Reproducibility** - Precise control of instructions to the computer
- **Strengths** of R as a GIS
  - R capabilities in **data processing** and **visualization**, combined with dedicated **packages for spatial data**
  - A **single environment** encompassing all analysis aspects - acquiring data, computation, statistics, visualization, Web, etc.
- Situations when **other tools** are needed
  - **Interactive editing or georeferencing** (but see `mapedit`[8])
  - Unique **GIS algorithms** (3D analysis, label placement, network routing, splitting lines at intersections, etc.)
  - Data that **cannot fit in RAM** (but R can connect to spatial databases[9])

[8] https://cran.r-project.org/package=mapedit

[9] https://cran.r-project.org/web/packages/sf/vignettes/sf2.html

Input and output of spatial data

- Reading spatial layers from a file into an R data structure, or writing the R data structure into a file, are handled by external libraries -
  - **OGR**[10] is used for reading/writing vector files, with `sf`
  - **GDAL**[11] is used for reading/writing raster files, with `raster`
  - **PROJ4**[12] is used for handling CRS, in both `sf` and `raster`
  - Working with specialized formats, e.g. **HDF** with `gdalUtils` or **NetCDF** with `ncdf4`

[10] http://www.gdal.org/ogr/
[11] http://www.gdal.org/
[12] http://trac.osgeo.org/proj/

sf: Geoprocessing Vector Layers

- **GEOS**[13] is used for geometric operations on **vector layers** with `sf` -
  - **Numeric operators** - Area, Length, Distance...
  - **Logical operators** - Contains, Within, Within distance, Crosses, Overlaps, Equals, Intersects, Disjoint, Touches...
  - **Geometry generating operators** - Centroid, Buffer, Intersection, Union, Difference, Convex-Hull, Simplification...

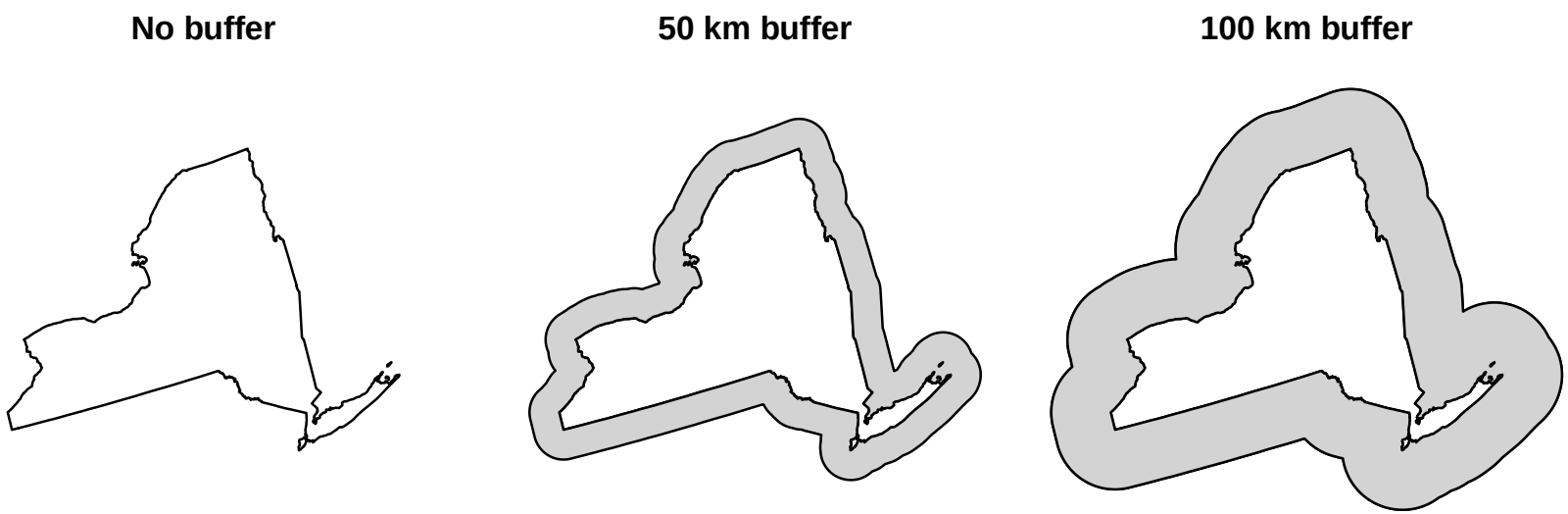


Figure 6: Buffer function

[13] http://trac.osgeo.org/geos/

`raster`: Geoprocessing Rasters

- Geometric operations on **rasters** can be done with package `raster` -
  - **Accessing cell values** - As vector, As matrix, Extract to points / lines / polygons, random / regular sampling, Frequency table, Histogram...
  - **Raster algebra** - Arithmetic (+, −, ...), Math (`sqrt`, `log10`, ...), logical (`!`, ==, >, ...), summary (`mean`, `max`, ...), Mask, Overlay...
  - **Changing resolution and extent** - Crop, Mosaic, (Dis)aggregation, Reprojection, Resampling, Shift, Rotation...
  - **Focal operators** - Distance, Direction, Focal Filter, Slope, Aspect, Flow direction...
  - **Transformations** - Vector layers <-> Raster...

`geosphere`: Geometric calculations on longitude/latitude

- Package `geosphere` implements **spherical trigonometry** functions for distance- and direction-related calculations on **geographic coordinates (lon-lat)**

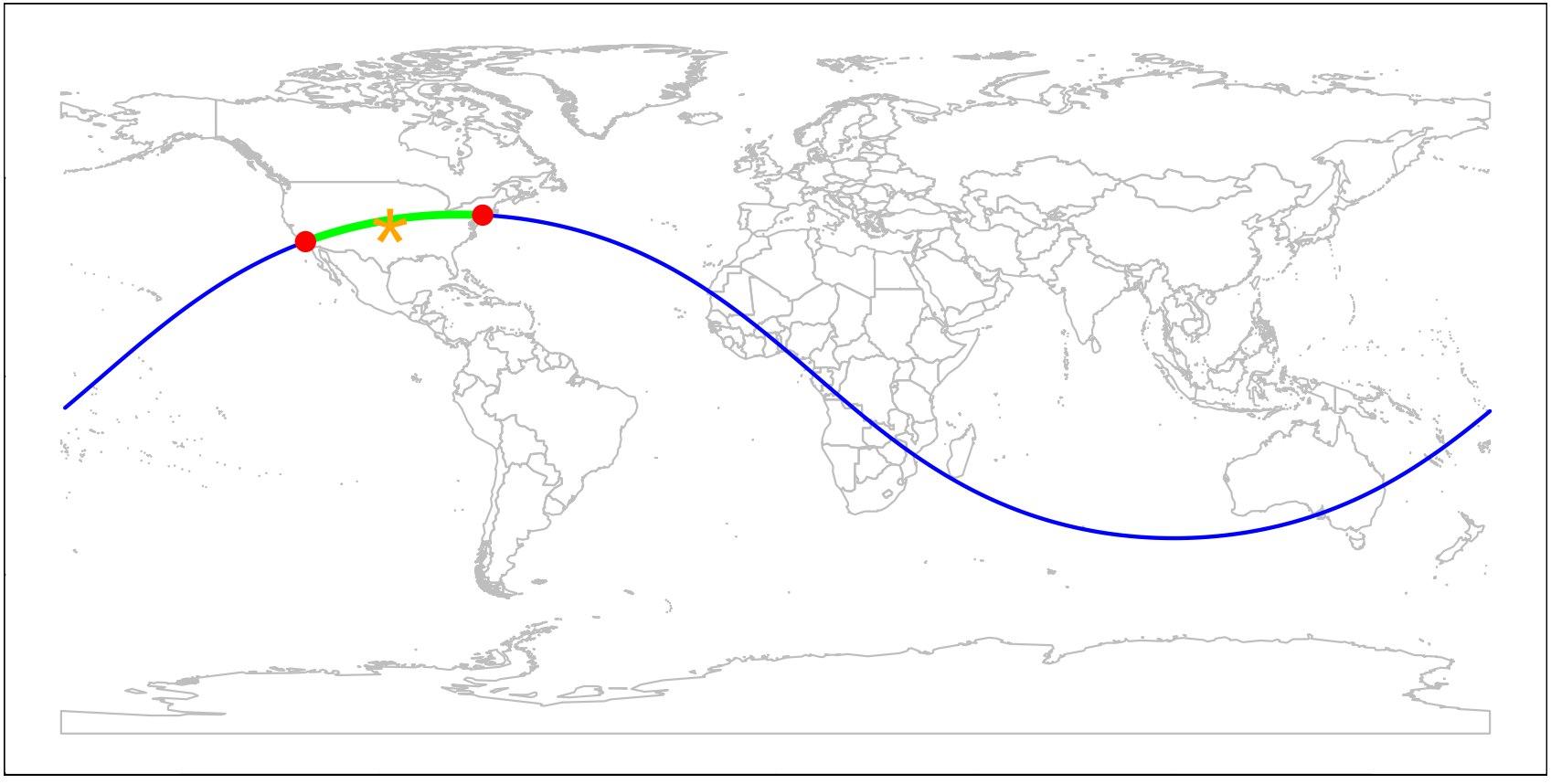


Figure 7: Points on Great Circle

`geosphere`: Geometric calculations on longitude/latitude

Figure 8: Visualizing Facebook Friends with `geosphere`[14]

[14] http://paulbutler.org/archives/visualizing-facebook-friends/

gstat: Geostatistical Modelling

- Univariate and multivariate geostatistics -
  - Variogram modelling
  - Ordinary and universal point or block (co)kriging
  - Cross-validation

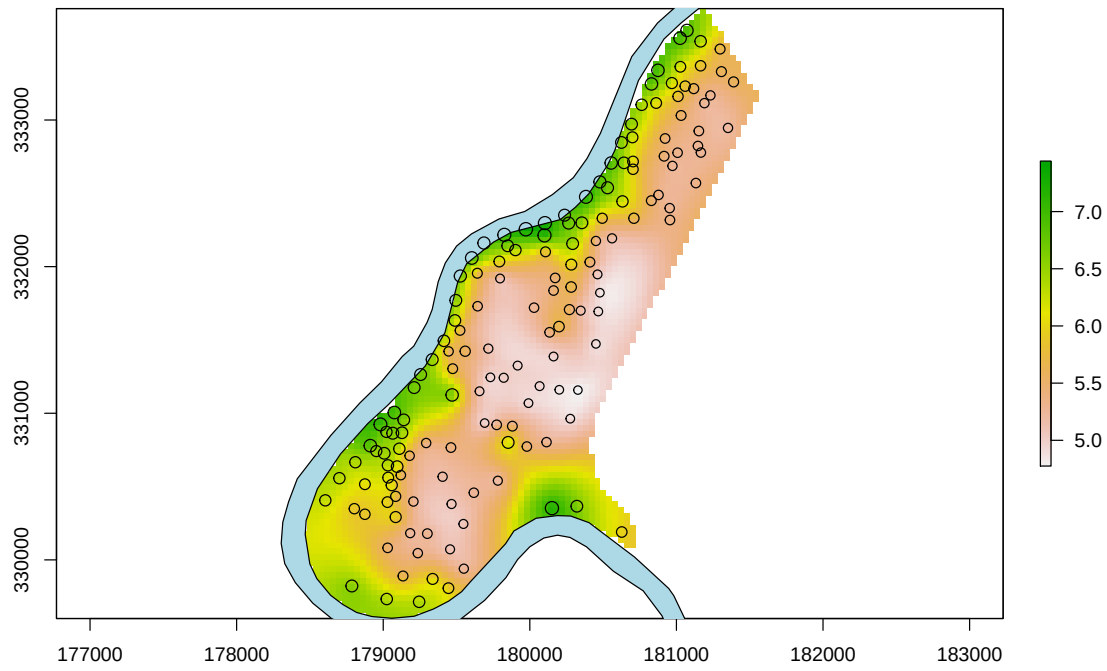


Figure 9: Predicted Zinc concentration, using Ordinary Kriging

`spdep`: Spatial dependence modelling

- Modelling with spatial weights -
  - Building neighbor lists and spatial weights
  - Tests for spatial autocorrelation for areal data (e.g. Moran's I)
  - Spatial regression models (e.g. SAR, CAR)

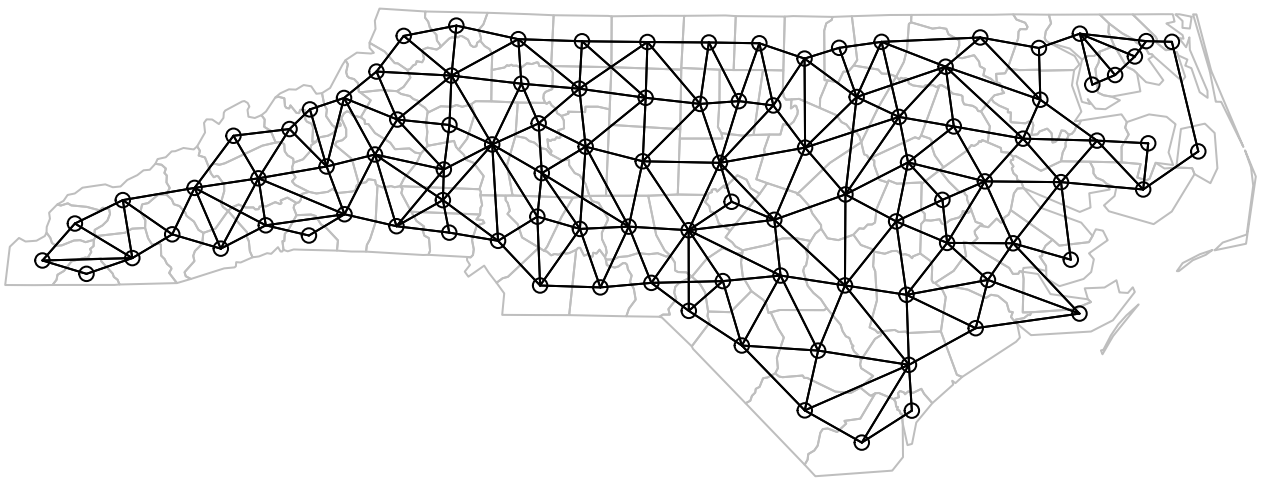


Figure 10: Neighbours list based on regions with contiguous boundaries

`spatstat`: Spatial point pattern analysis

- Techniques for statistical analysis of spatial point patterns, such as -
  - Kernel density estimation
  - Detection of clustering using Ripley's K-function
  - Spatial logistic regression

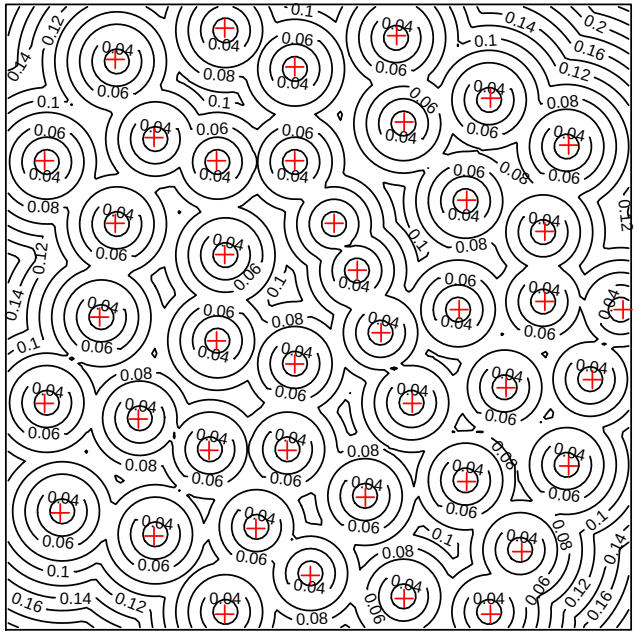


Figure 11: Distance map for the Biological Cells point pattern dataset

`osmdata`: Access to OpenStreetMap data

- Accessing **OpenStreetMap (OSM)** data using the **Overpass API**[15]

```
library(sf)
library(osmdata)
q = opq(bbox = "Beer-Sheva, Israel")
q = add_osm_feature(q, key = "highway")
dat = osmdata_sf(q)
lines = dat$osm_lines
pol = dat$osm_polygons
pol = st_cast(pol, "MULTILINESTRING")
pol = st_cast(pol, "LINESTRING")
lines = rbind(lines, pol)
lines = lines[, c("osm_id", "highway")]
lines = st_transform(lines, 32636)
plot(lines)
```

[15] http://wiki.openstreetmap.org/wiki/Overpass_API

`osmdata`: Access to OpenStreetMap data

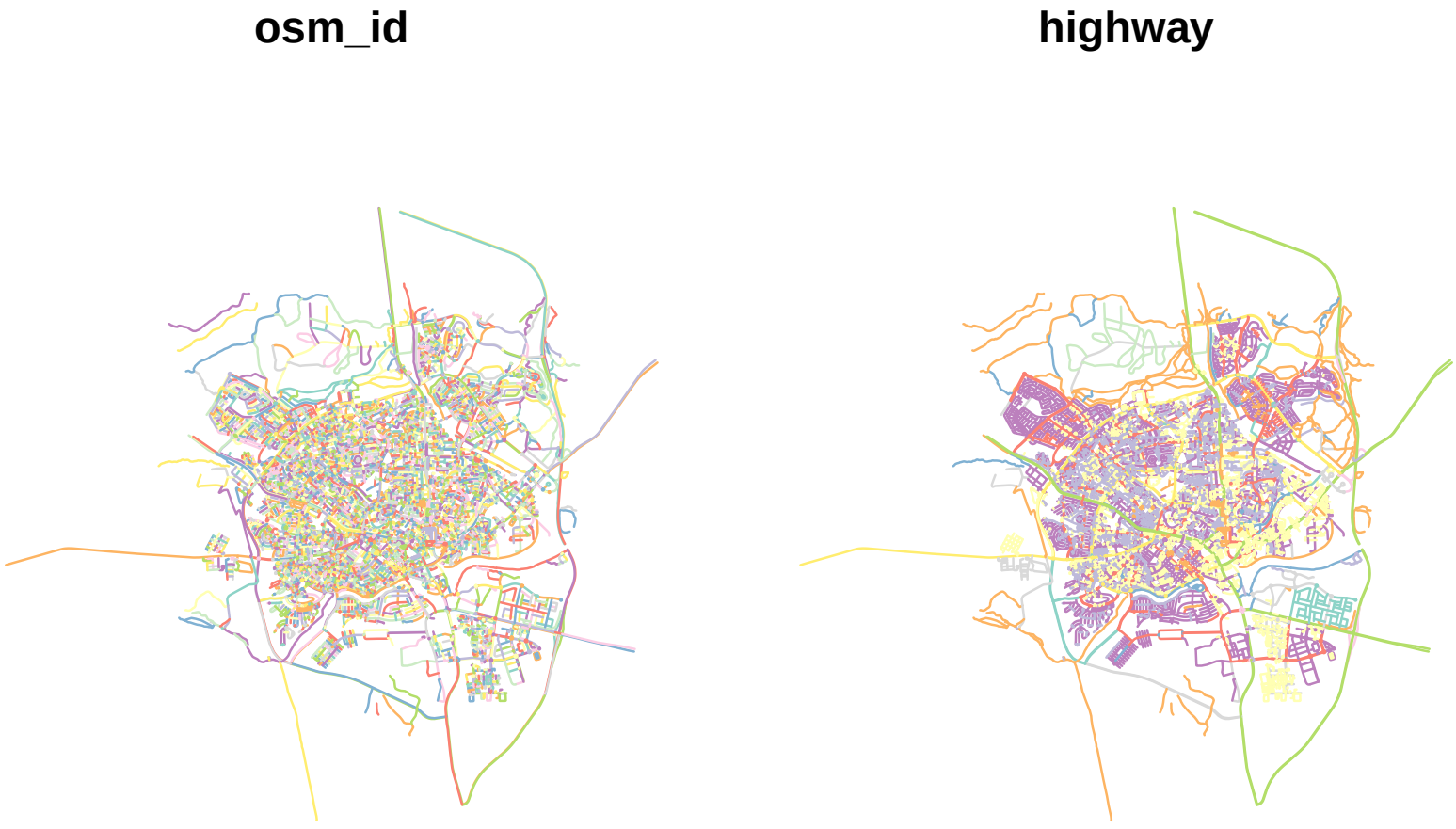


Figure 12: Beer-Sheva road network

ggplot2, ggmap: Visualization

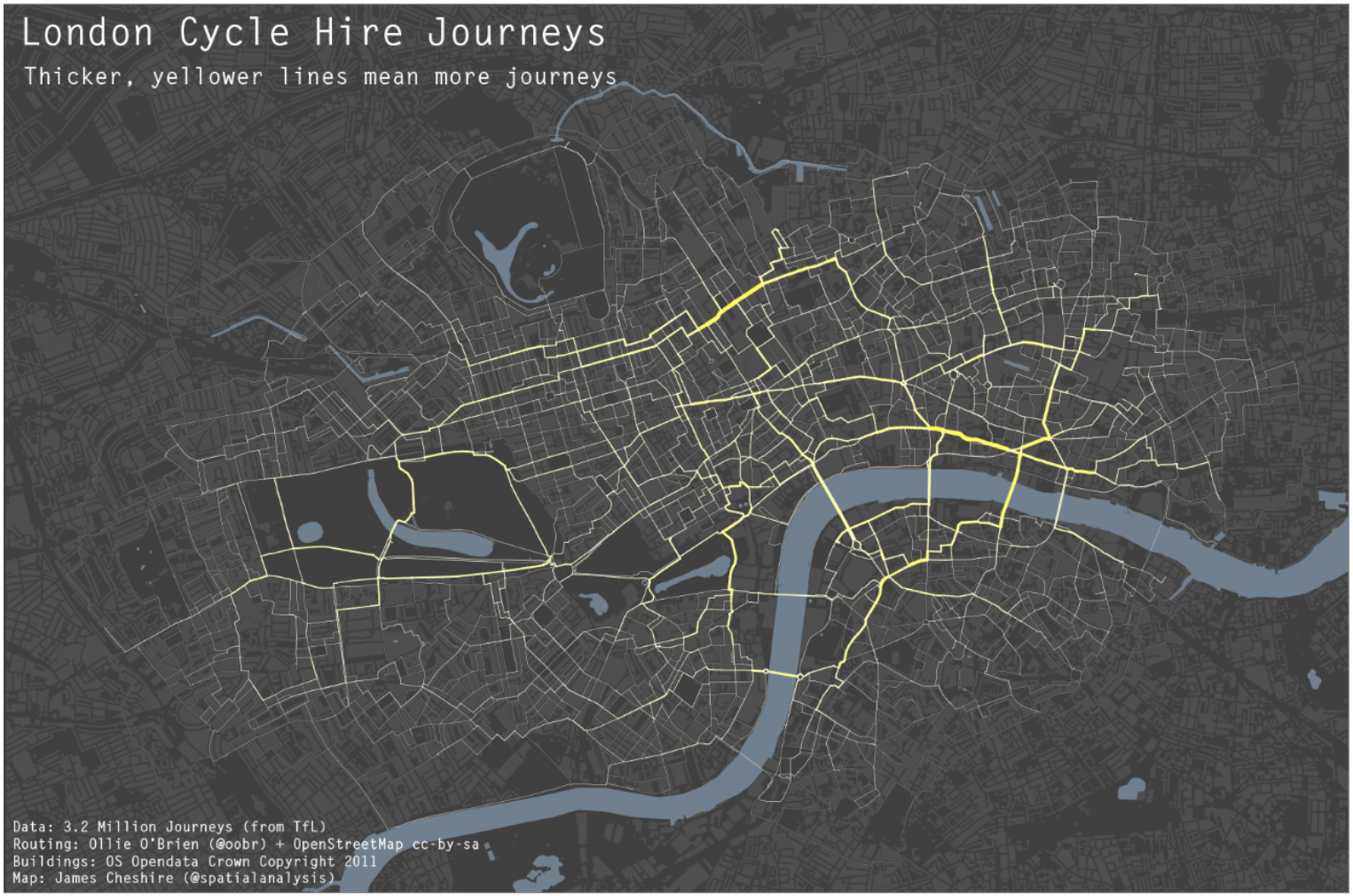


Figure 13: London cycle hire journeys with ggplot2[16]

[16] http://spatial.ly/2012/02/great-maps-ggplot2/

ggplot2, ggmap: Visualization

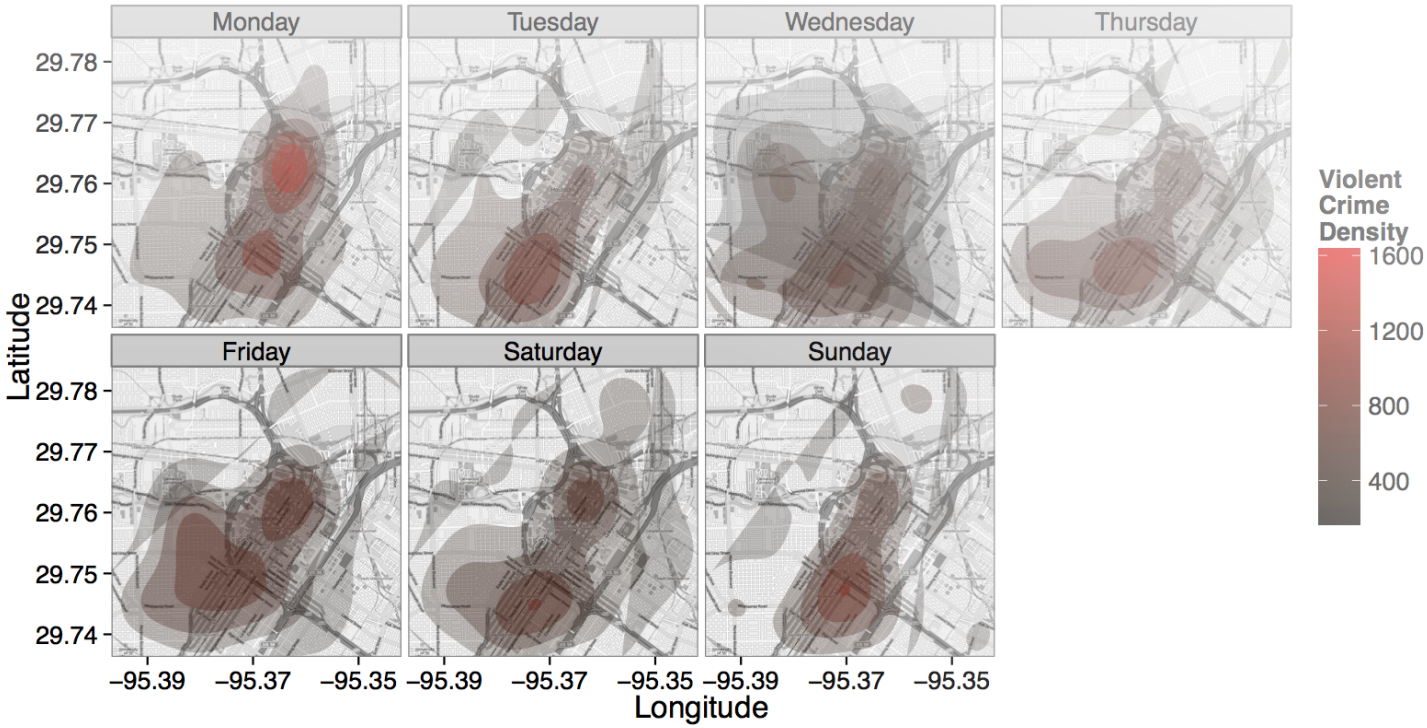


Figure 14: Crime density by day with ggplot2[17]

[17] http://spatial.ly/2012/02/great-maps-ggplot2/

`leaflet`, `mapview`: Web mapping

- Packages `leaflet` and `mapview` provide methods to produce **interactive maps** using the Leaflet JavaScript library[18]

- Package `leaflet`[19] gives more low-level control
- Package `mapview`[20] is a wrapper around `leaflet`, automating addition of useful features -
  - Commonly used **basemaps**
  - **Color scales** and **legends**
  - **Labels**
  - **Popups**

[18] http://leafletjs.com/
[19] https://rstudio.github.io/leaflet/
[20] https://r-spatial.github.io/mapview/

`mapview`: Example

- **Function `mapview`** produces an interactive map given a spatial object
  - `zcol="..."` specifies the **attribute** used for symbology
  - `legend=TRUE` adds a **legend**

```
library(sf)
library(mapview)
states = st_read("USA_2_GADM_fips.shp")
mapview(states, zcol = "NAME_1", legend = TRUE)
```

mapview: Example

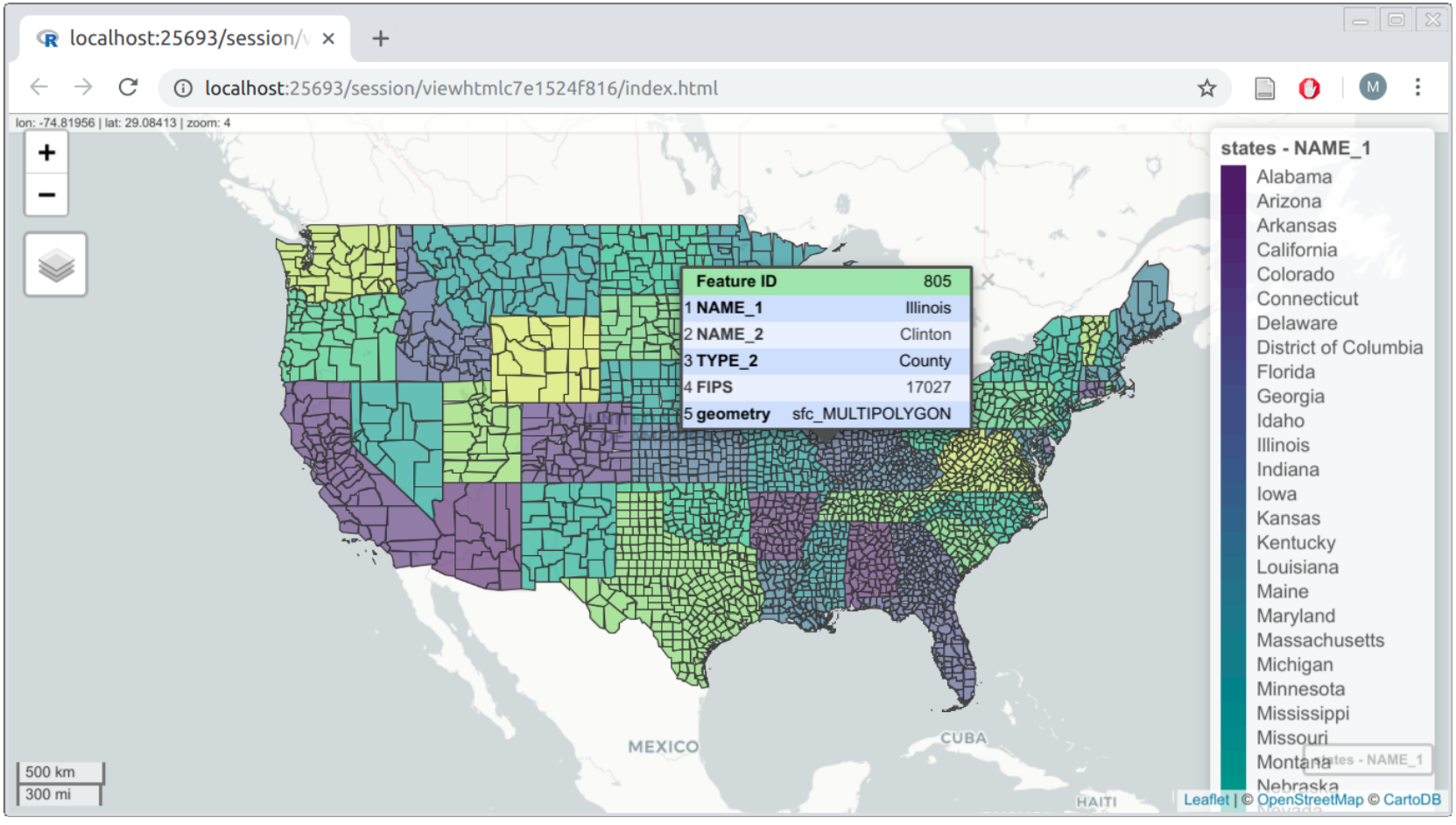


Figure 15: Intractive map made with `mapview`

Books

- **Hierarchical Modeling and Analysis for Spatial Data** (1st ed 2003, 2nd ed. 2014)
- **Model-based Geostatistics** (2007)
- **Applied Spatial Data Analysis with R** (1st ed. 2008, 2nd ed. 2013)
- **A Practical Guide for Geostatistical Mapping** (2009)
- **Spatial Data Analysis in Ecology and Agriculture using R** (2012)
- **Displaying Time Series, Spatial, and Space-Time Data with R** (1st ed. 2014, 2nd ed. 2018)
- **Learning R for Geospatial Analysis** (2014)
- **An Introduction to R for Spatial Analysis and Mapping** (2015)
- **Spatial Point Patterns: Methodology and Applications with R** (2015)
- **Geocomputation with R** (2019)[21]

[21] https://geocompr.robinlovelace.net/

Online courses and tutorials

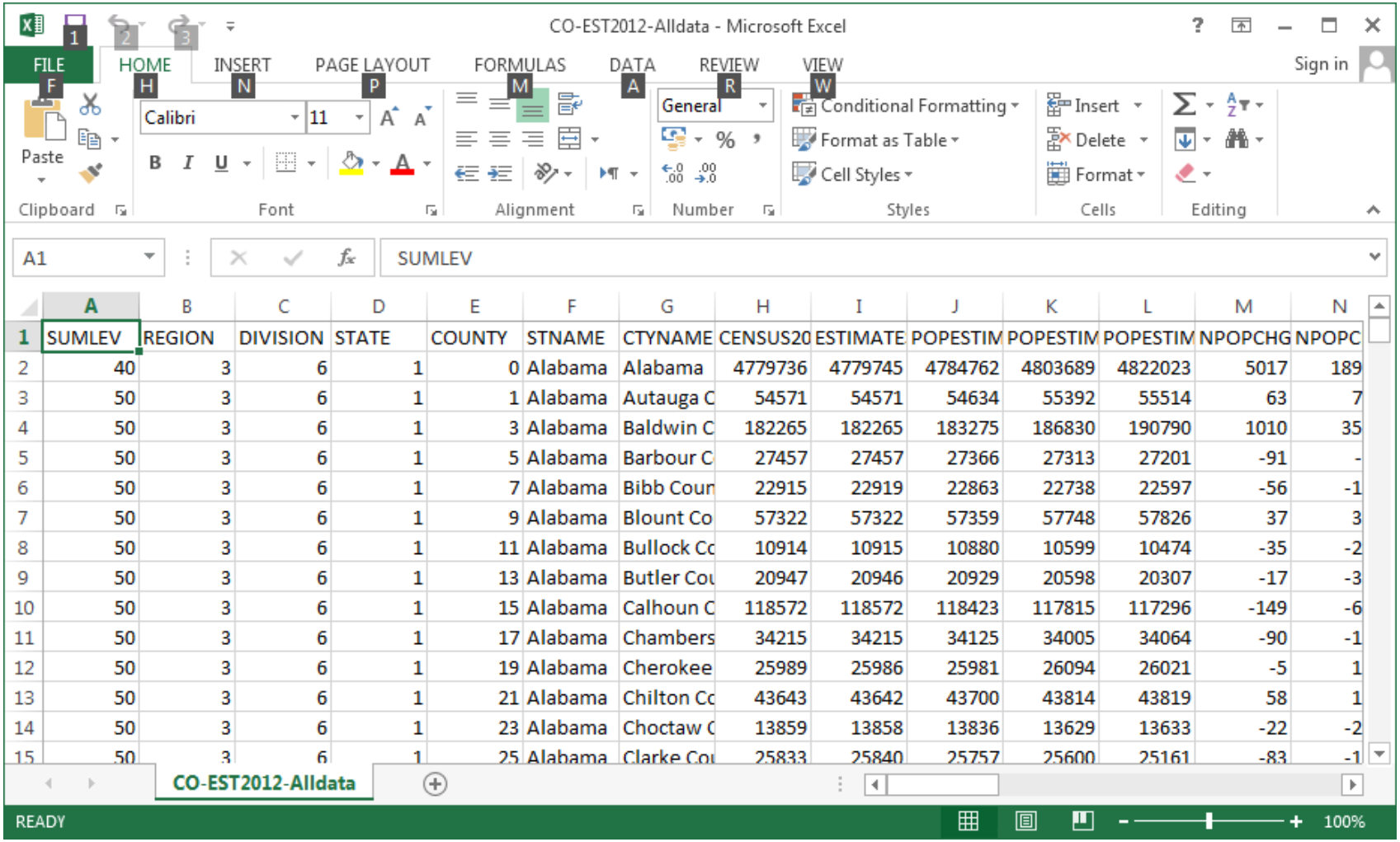
- Courses
  - https://mgimond.github.io/Spatial/index.html
  - http://adamwilson.us/SpatialDataScience/
  - http://geog.uoregon.edu/bartlein/courses/geog490/index.html
  - http://132.72.155.230:3838/r/ (This course)
- Tutorials
  - https://datacarpentry.org/lessons/#geospatial-curriculum
  - http://rspatial.org/
  - http://www.nickeubank.com/gis-in-r/
  - https://www.neonscience.org/resources/data-tutorials

Why is programming necessary?

- Is this a **Microsoft Excel** spreadsheet?
- The file has an Excel **icon**, and it opens in Excel on **double-click**...

Figure 16: CSV file

Why is programming necessary?



	A	B	C	D	E	F	G	H	I	J	K	L	M	N
1	SUMLEV	REGION	DIVISION	STATE	COUNTY	STNAME	CTYNAME	CENSUS20	ESTIMATE	POPESTIM	POPESTIM	POPESTIM	NPOPCHG	NPOPC
2	40	3	6	1	0	Alabama	Alabama	4779736	4779745	4784762	4803689	4822023	5017	189
3	50	3	6	1	1	Alabama	Autauga C	54571	54571	54634	55392	55514	63	7
4	50	3	6	1	3	Alabama	Baldwin C	182265	182265	183275	186830	190790	1010	35
5	50	3	6	1	5	Alabama	Barbour C	27457	27457	27366	27313	27201	-91	-
6	50	3	6	1	7	Alabama	Bibb Coun	22915	22919	22863	22738	22597	-56	-1
7	50	3	6	1	9	Alabama	Blount Co	57322	57322	57359	57748	57826	37	3
8	50	3	6	1	11	Alabama	Bullock Cc	10914	10915	10880	10599	10474	-35	-2
9	50	3	6	1	13	Alabama	Butler Cou	20947	20946	20929	20598	20307	-17	-3
10	50	3	6	1	15	Alabama	Calhoun C	118572	118572	118423	117815	117296	-149	-6
11	50	3	6	1	17	Alabama	Chambers	34215	34215	34125	34005	34064	-90	-1
12	50	3	6	1	19	Alabama	Cherokee	25989	25986	25981	26094	26021	-5	1
13	50	3	6	1	21	Alabama	Chilton Cc	43643	43642	43700	43814	43819	58	1
14	50	3	6	1	23	Alabama	Choctaw C	13859	13858	13836	13629	13633	-22	-2
15	50	3	6	1	25	Alabama	Clarke Cou	25833	25840	25757	25600	25161	-83	-1

Figure 17: CSV file opened in Excel

Why is programming necessary?

- However, this is in fact a **plain-text** file in the **Comma Separated Values** (CSV) format, and can be opened in various other software, such as Notepad

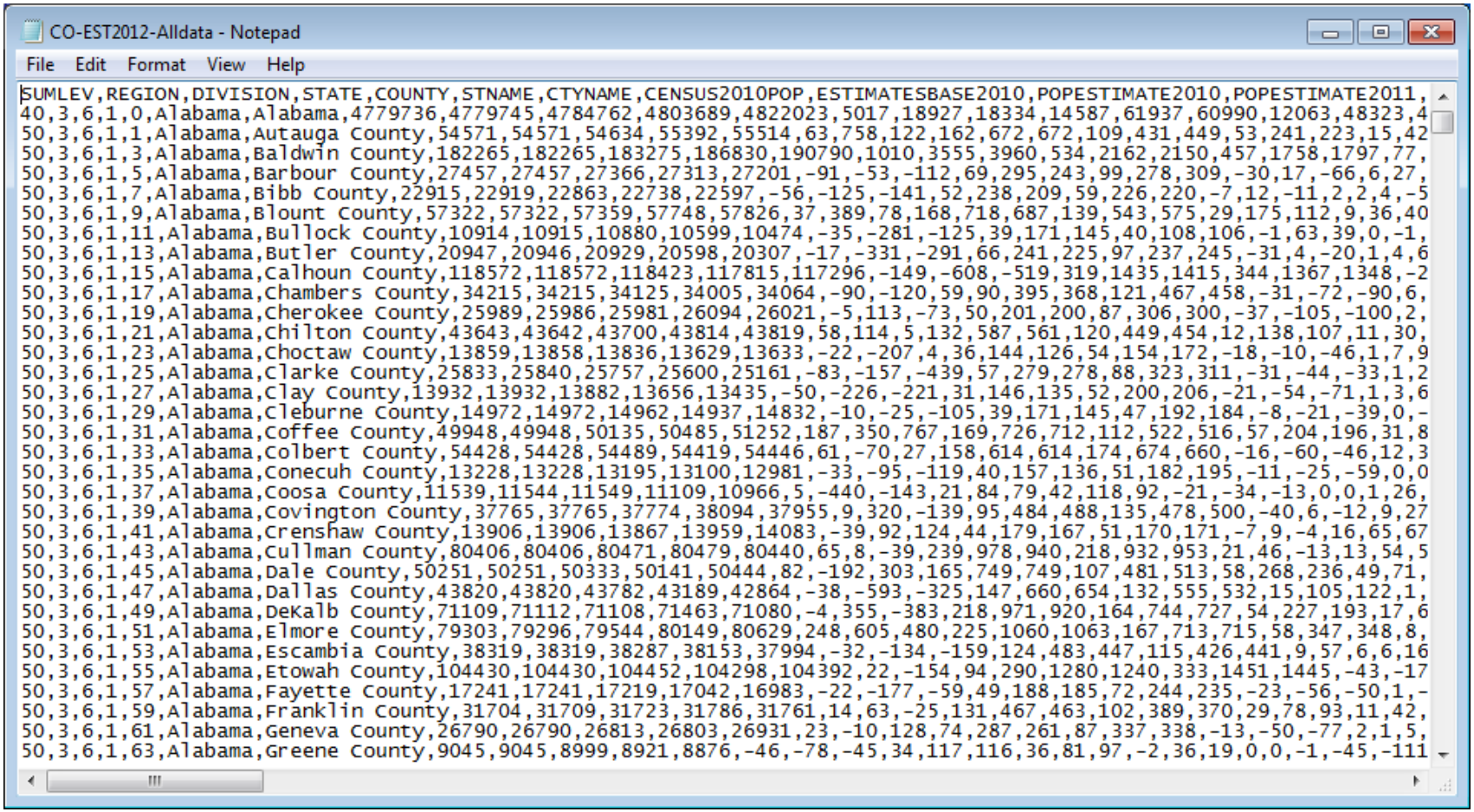


Figure 18: CSV file opened in Notepad

Why is programming necessary?

- The graphical interface **"protects"** us from the little details -
  - Hiding the `.csv` file **extension**
  - Displaying an Excel **icon**
  - Automatically **opening** the file in Excel
- Is this a **bad** thing?
  - We can be unaware of the fact that the file can be opened in software **other** than Excel
  - In general - the "ordinary" interaction with the computer is **limited** to clicking on links, selecting from menus and filling dialog boxes
  - The latter approach suggests there are **"boundaries"** set by the computer interface for the user who wishes to accomplish a given task
  - Of course the opposite is true - the user has full **control**, and can tell the computer exactly what he wants to do

Why is programming necessary?

- Question: how can we **change** the value of a particular raster **cell**, such as the `[120, 120]` cell?

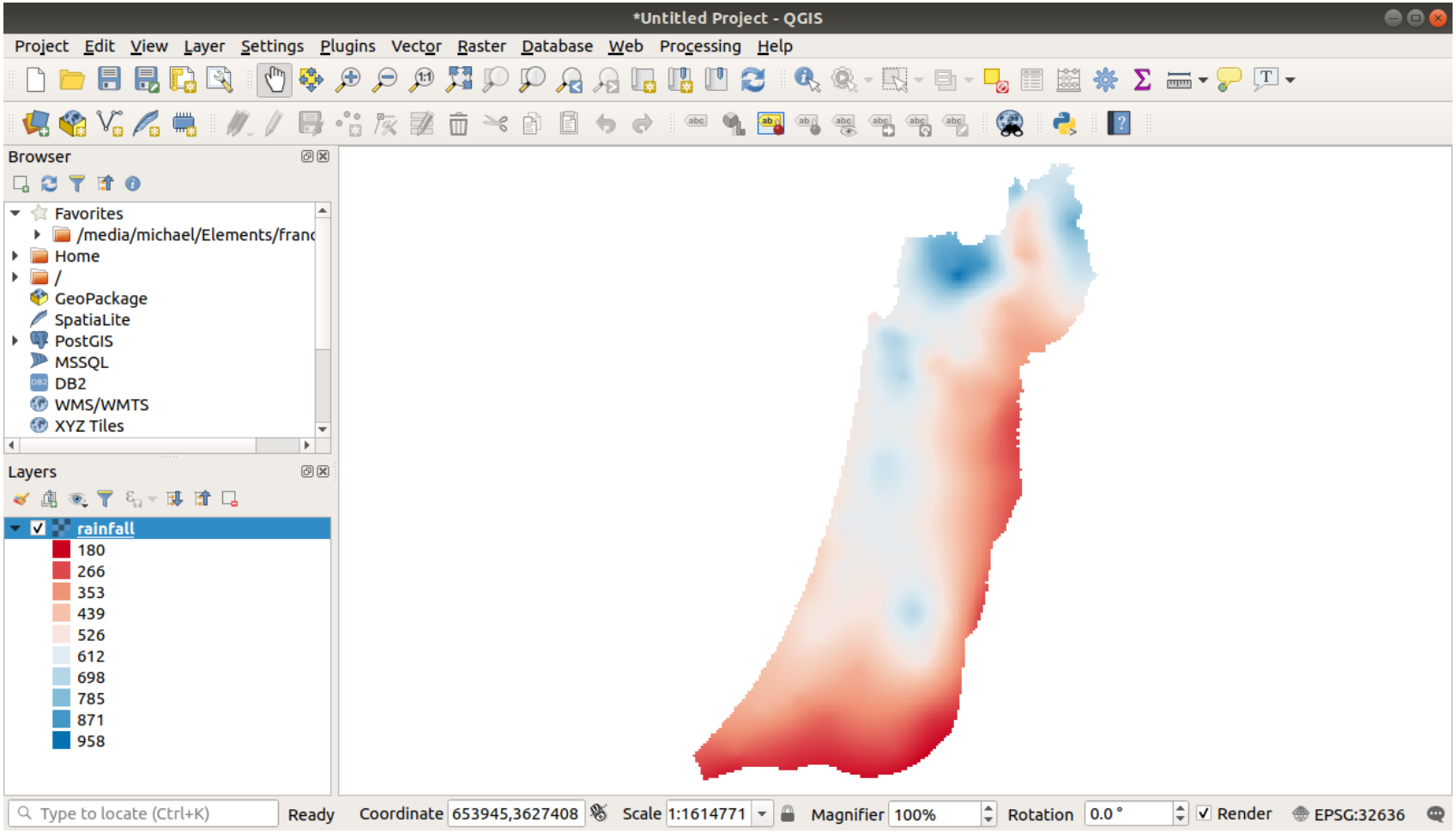


Figure 19: The `rainfall.tif` raster

Why is programming necessary?

- In **ArcGIS** -
  - Open the raster with “Add Data”
  - Convert the raster to points
  - Calculate row and column indices
  - Locate the point we want to change and edit its attribute
  - Convert the points to back to a raster, using the same extent and resolution and setting a snap raster
  - Export the raster

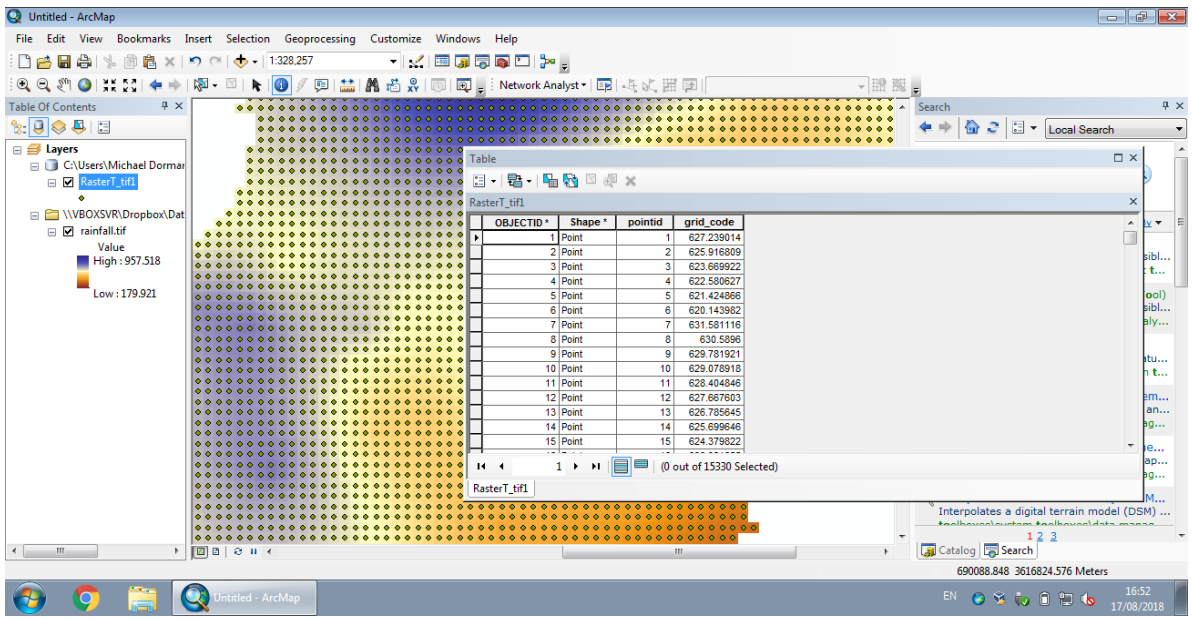


Figure 20: Raster to points in ArcGIS

Why is programming necessary?

- In **R** -
  - Loading the `raster` package
  - Reading the `rainfall.tif` raster
  - Assigning a new value to the `[120, 120]` cell
  - Writing the raster to disk

```
library(raster)
r = raster("rainfall.tif")
r[120, 120] = 1000
writeRaster(r, "rainfall2.tif")
```

Why is programming necessary?

- In **Python** -

```
import gdalnumeric
r = gdalnumeric.LoadFile("rainfall.tif")
r[119, 119] = 1000
gdalnumeric.SaveArray(
    r,
    "rainfall2.tif",
    format = "GTiff",
    prototype = "rainfall.tif"
)
```

What is programming

- A **computer program** is a sequence of text instructions that can be "understood" by a computer and executed
- A **programming language** is a machine-readable artificial language designed to express computations that can be performed by a computer
- **Programming** is the preferred way for giving instructions to the computer because that way -
  - We break free from the **limitations** of the graphical interface, and are able to perform tasks that are unfeasible or even impossible
  - We can keep the code for **editing** and **re-use** in the future, and as a reminder to ourselves of what we did in the past
  - Sharing a **precise** record of our analysis with others, making our results reproducible

Computer hardware

- The **Central Processing Unit (CPU)** performs (simple) calculations very fast
- The **Random Access Memory (RAM)** is a short-term fast memory
- **Mass Storage** (e.g. hard drive) is long-term and high-capacity memory, but slow
- A **Keyboard** is an example of an input device
- A **Screen** is an example of an output device

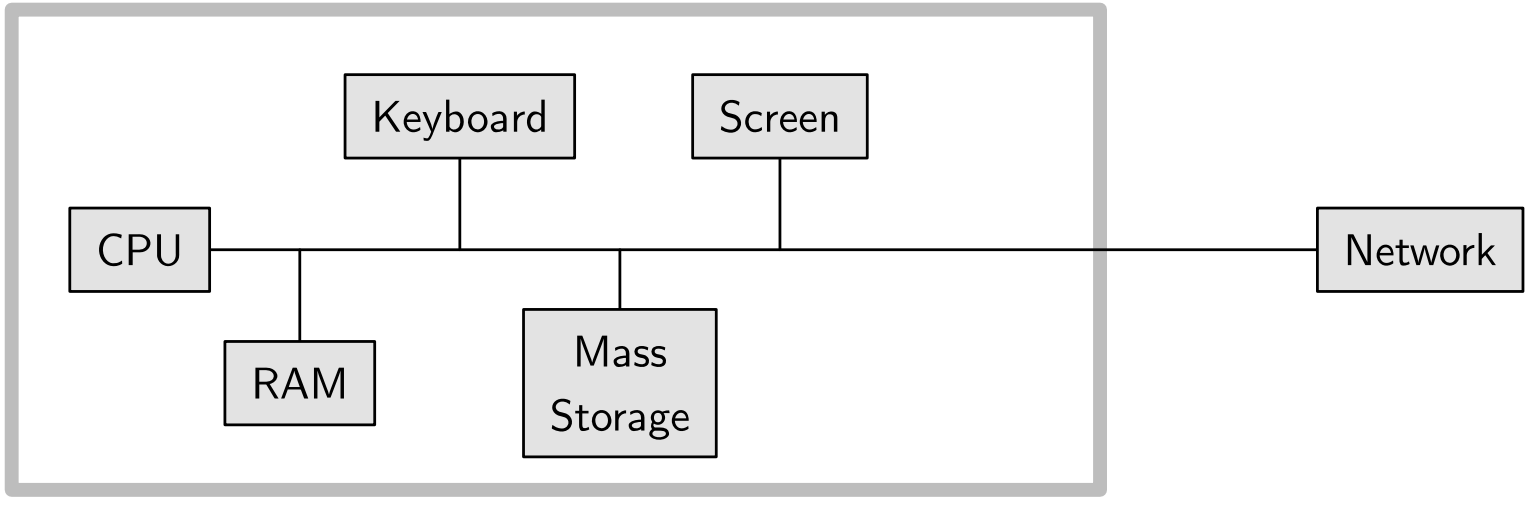


Figure 21: Components of a computing environment

Abstraction

- Programming languages differ in their level of **abstraction** and **execution models**
- **Abstraction** is the presentation of data and instructions which hide implementation details
  - **Low-level** programming languages provide little or no abstraction
    - **Advantage**: efficient memory use and therefore fast
    - **Disadvantage**: difficult to use because of the many technical details the programmer needs to know
  - **High-level** programming languages provide more abstraction and automatically handle various aspects such as memory use
    - **Advantage**: more "understandable" and easier to use
    - **Disadvantage**: Less efficient and therefore slower

Abstraction

- **Abstraction** lets the programmer focus on the task at hand, ignoring the small technical details

Assembly

```
 .486p
         .model   flat,STDCALL
include  win32.inc

extrn            MessageBoxA:PROC
extrn            ExitProcess:PROC

.data

HelloWorld db "Hello, world!",0
msgTitle db "Hello world program",0

.code
Start:
         push     MB_ICONQUESTION + MB_APPLMODAL + MB_OK
         push     offset msgTitle
         push     offset HelloWorld
         push     0
         call     MessageBoxA

         push 0
         call ExitProcess
ends
end Start
```

C++

```
#include <iostream>
using namespace std;

int main()
{
  cout << "Hello, World!\n";
  return 0;
}
```

R

```
print("Hello , world!")
```

Figure 22: Increasing abstraction from Assembly to C++ to R

Execution models

- **Execution models** are systems for execution of programs written in a given programming language
  - **Compiled** - before being executed, the code needs to be compiled into **executable** machine code
  - **Interpreted** - the code is executed directly, using an **interpreter**
    - **Advantage**: easier to develop and use
    - **Disadvantage**: lower efficiency

Execution models

- In **compiled** execution models, the code is first translated to an executable file

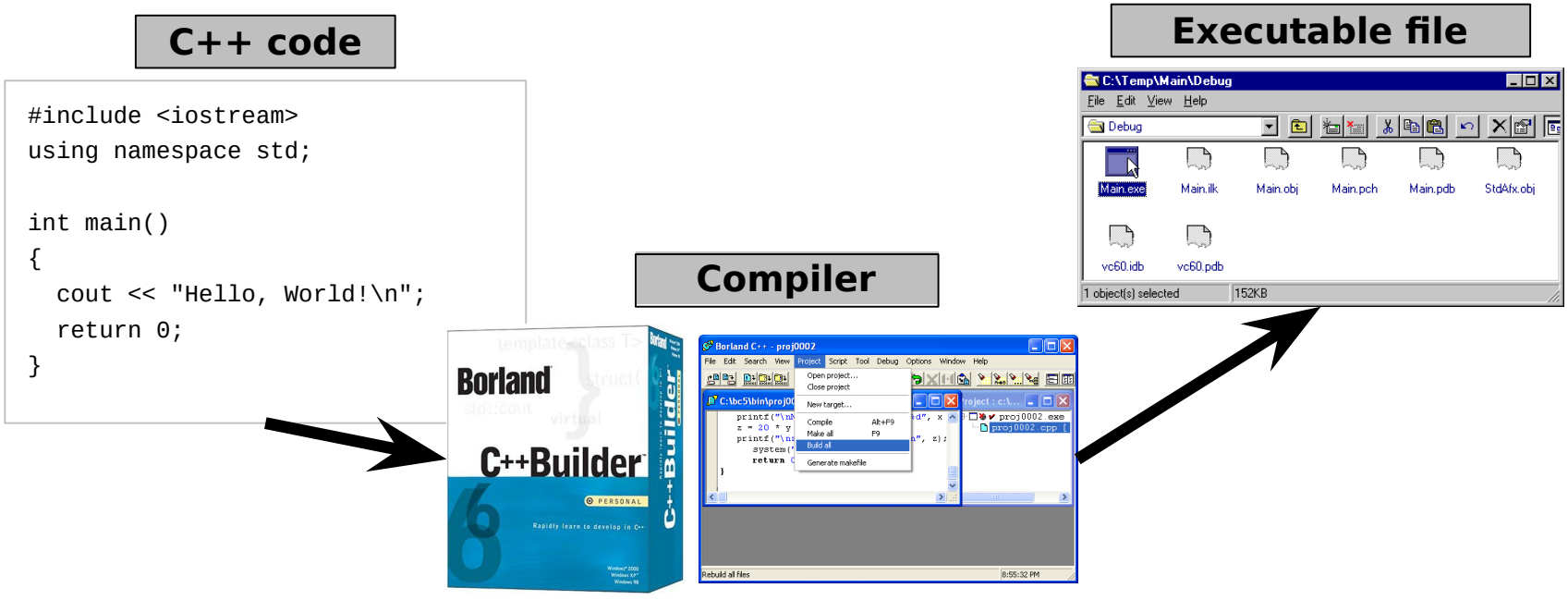


Figure 23: Compilation of C++ code

Execution models

- The executable file can then be run

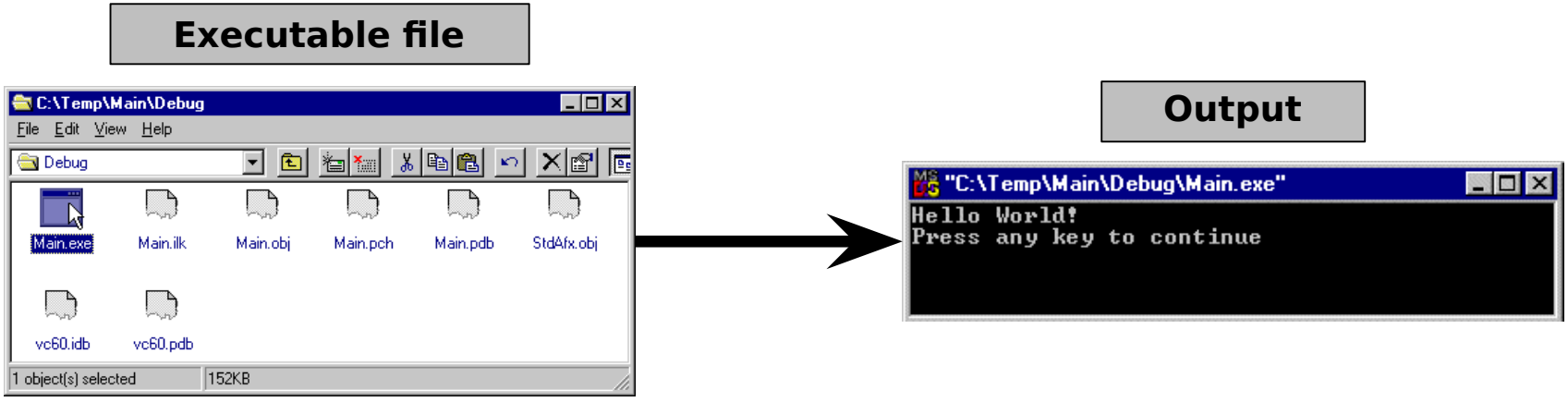


Figure 24: Running an executable file

Execution models

- In **interpreted** execution models, the code can be run directly using the interpreter

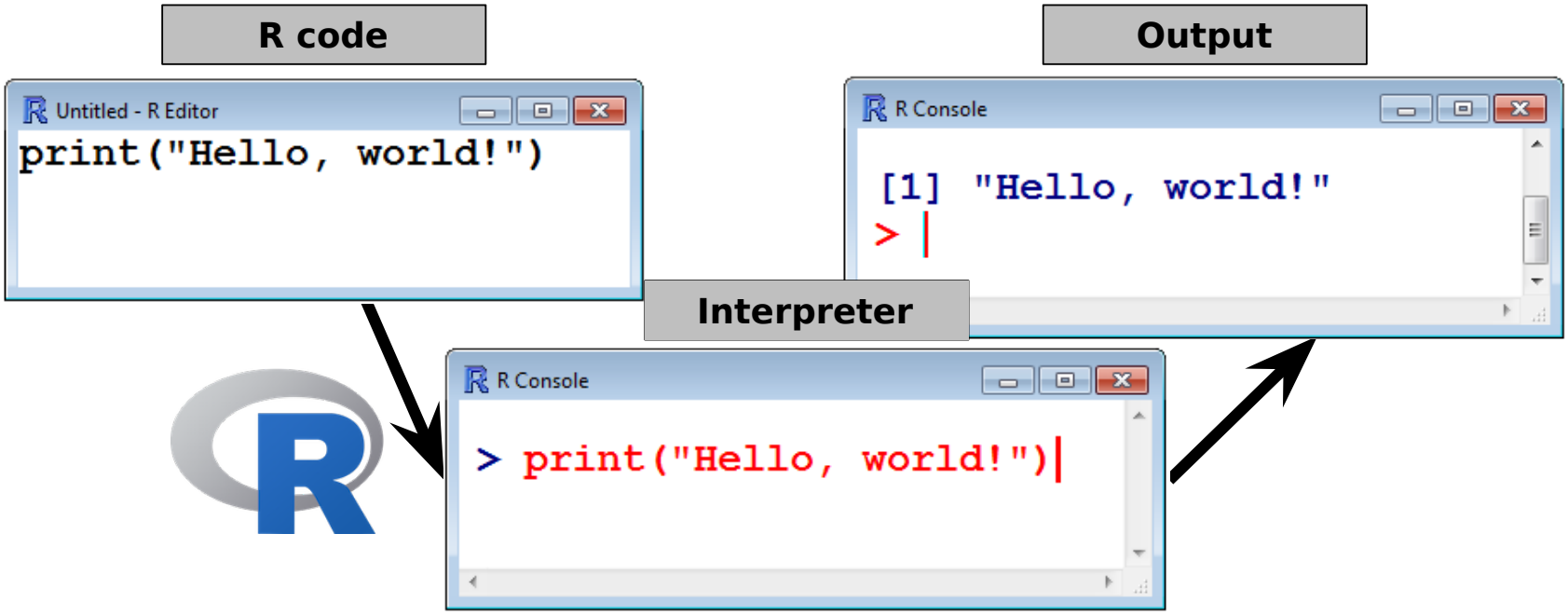


Figure 25: Running R code

Programming languages

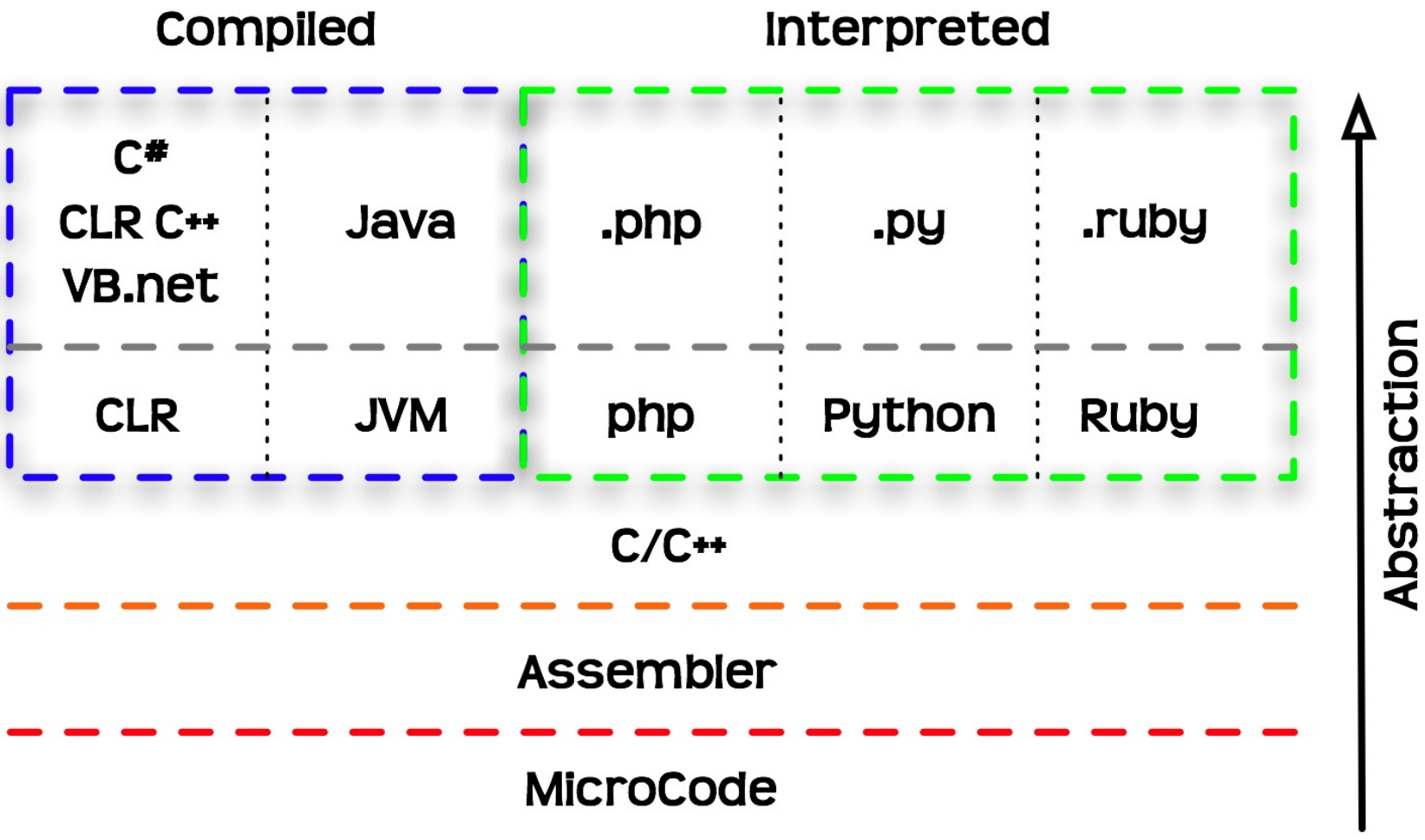


Figure 26: Programming languages classification based on abstraction level and execution model

Object-oriented programming

- The interaction with the computer takes places through **objects**
- Each object belongs to a **class**: an abstract structure with certain properties
- Objects are in fact **instances** of a class
- The class comprises a template which sets the **properties** and **methods** each object of that class should have, while an object contains specific **values** for that particular instance
- For example -
  - All cars we see in the parking lot are **instances** of the "car" class
  - The "car" class has certain **properties** (manufacturer, color, year) and **methods** (start, drive, stop)
  - Each "car" object has specific **values** for the properties (Suzuki, brown, 2011)

Object-oriented programming

Object	Properties	Methods
	car.name = Fiat	car.start()
	car.model = 500	car.drive()
	car.weight = 850kg	car.brake()
	car.color = white	car.stop()

Figure 27: An object[22]

[22] https://www.w3schools.com/js/js_objects.asp

Object-oriented programming

```
r = raster("rainfall.tif")
r
```

```
## class       : RasterLayer
## dimensions  : 236, 155, 36580  (nrow, ncol, ncell)
## resolution  : 1000, 1000  (x, y)
## extent      : 615964.7, 770964.7, 3456430, 3692430  (xmin, xmax, ymin, ymax)
## coord. ref. : +proj=utm +zone=36 +datum=WGS84 +units=m +no_defs +ellps=WGS84
## data source : /home/michael/Dropbox/Data2/rainfall.tif
## names       : rainfall
## values      : 179.9213, 957.5177  (min, max)
```

Inheritance

- One of the implications of object-oriented programming is **inheritance**
- Inheritance makes it possible for one class to **"extend"** another class, by adding new properties and/or new methods
- For example -
  - A "taxi" class is an **extension** of the "car" class
  - A "taxi" has **new** properties (taxi company name), and new methods (switching the taximeter on and off)

Inheritance

```
str(r)
```

```
## Formal class 'RasterLayer' [package "raster"] with 12 slots
##   ..@ file    :Formal class '.RasterFile' [package "raster"] with 13 slots
##   .. .. ..@ name        : chr "/home/michael/Dropbox/Data2/rainfall.tif"
##   .. .. ..@ datanotation: chr "FLT8S"
##   .. .. ..@ byteorder   : chr "little"
##   .. .. ..@ nodatavalue : num -Inf
##   .. .. ..@ NAchanged   : logi FALSE
##   .. .. ..@ nbands      : int 1
##   .. .. ..@ bandorder   : chr "BIL"
##   .. .. ..@ offset      : int 0
##   .. .. ..@ toptobottom : logi TRUE
##   .. .. ..@ blockrows   : int 6
##   .. .. ..@ blockcols   : int 155
##   .. .. ..@ driver      : chr "gdal"
##   .. .. ..@ open        : logi FALSE
##   ..@ data    :Formal class '.SingleLayerData' [package "raster"] with 13 sl
##   .. .. ..@ values    : logi(0)
##   .. .. ..@ offset    : num 0
##   .. .. ..@ gain      : num 1
##   .. .. ..@ inmemory  : logi FALSE
##   .. .. ..@ fromdisk  : logi TRUE
##   .. .. ..@ isfactor  : logi FALSE
##   .. .. ..@ attributes: list()
```

Installing R and RStudio

R

- R can be downloaded from the R-project website
  - https://www.r-project.org/
- Current version is 3.5.1
  - https://cloud.r-project.org/bin/windows/base/R-3.5.1-win.exe

RStudio

- RStudio can be downloaded from the company website
  - https://www.rstudio.com/
- Current version is 1.1.456
  - https://download1.rstudio.org/RStudio-1.1.456.exe

RGui - How to start

- Start → All Programs → R → R x64 3.5.1

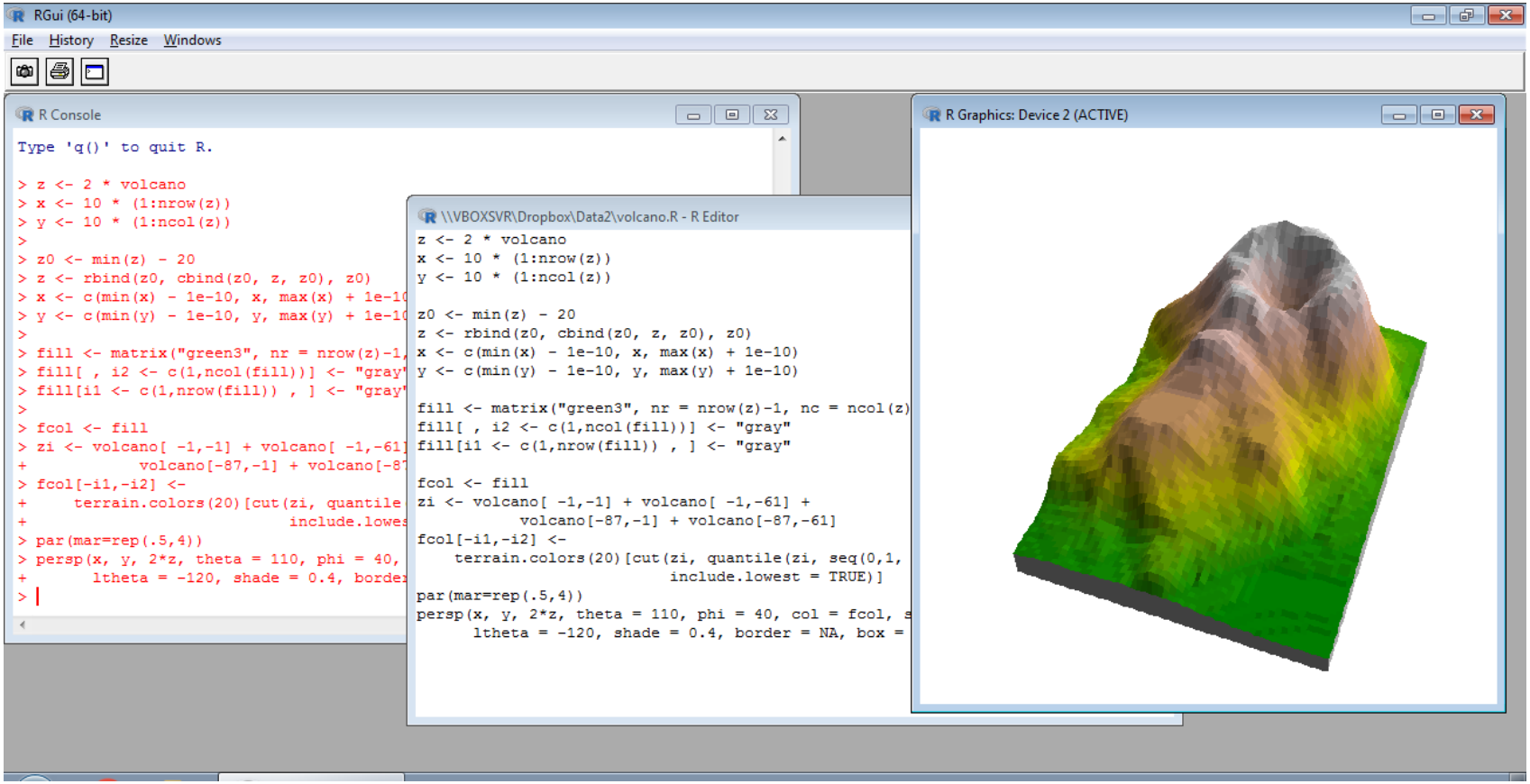


Figure 28: RGui

RStudio - How to start

- Start → All Programs → RStudio → RStudio

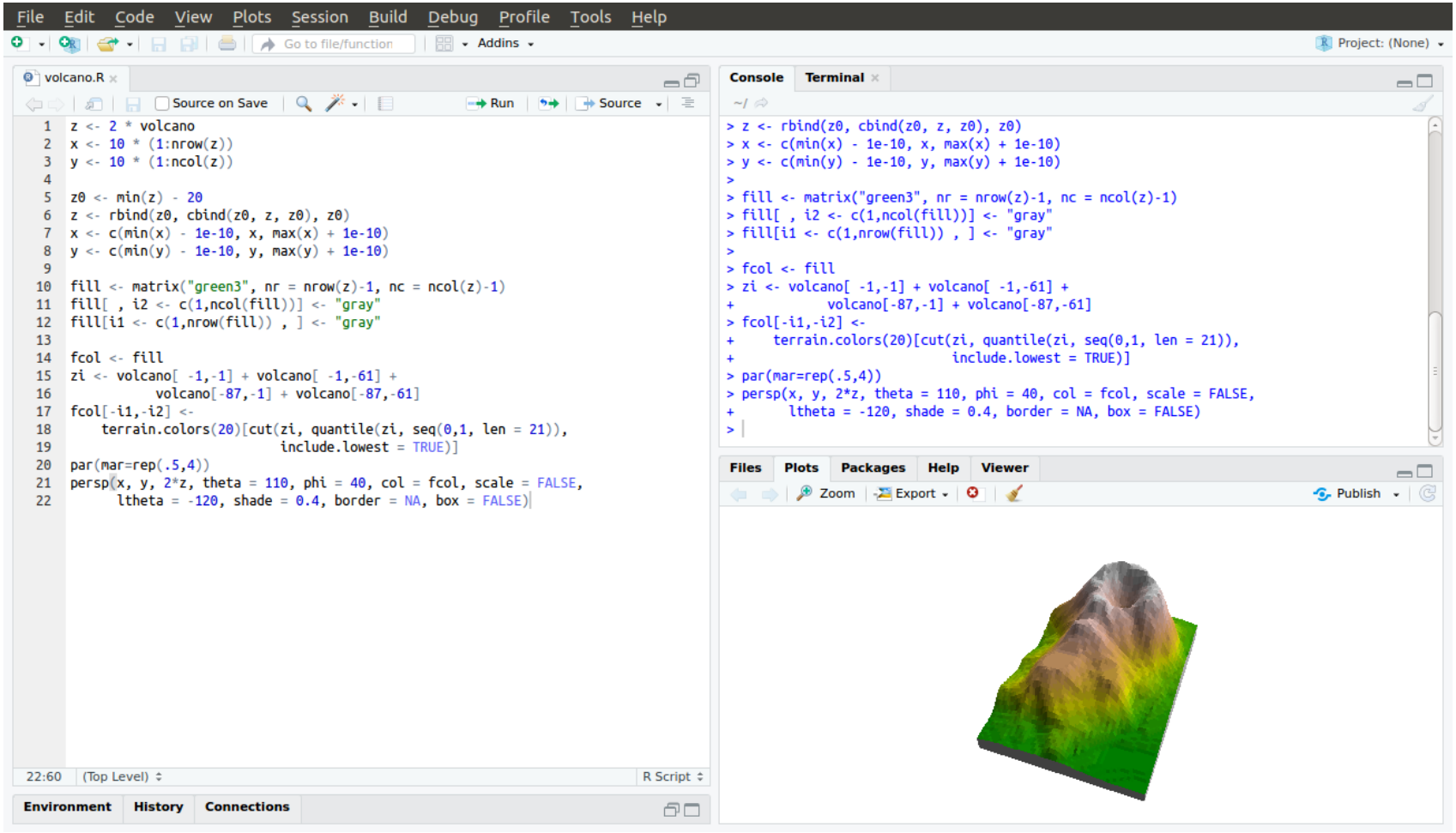


Figure 29: RStudio

RStudio - Interface components

- In this lesson we will only work with the **console**, i.e. the command line
- In the following lessons we will also work with other RStudio panels

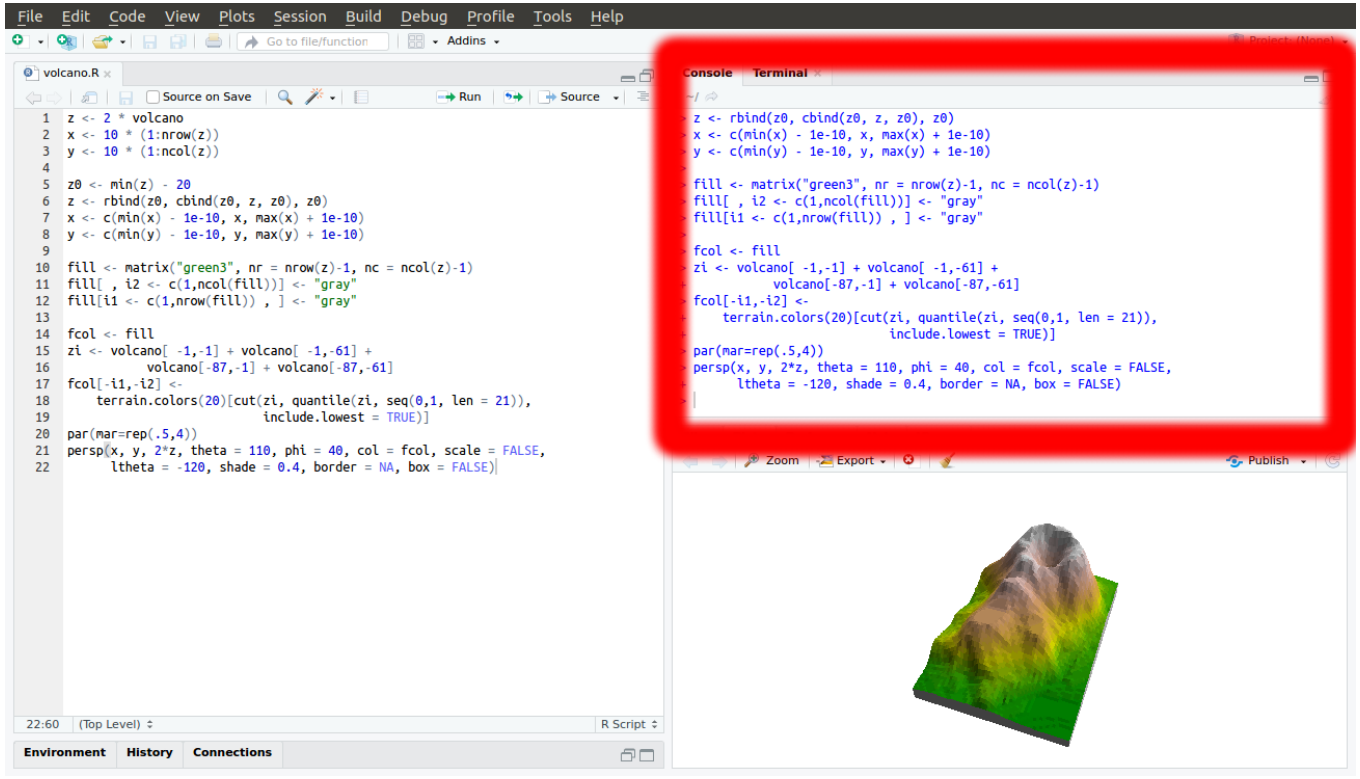


Figure 30: RStudio console

Expressions

- We can type expressions at the command line and press **Enter**
- For example, let's type the expression 1+3+5+7 -

```
1 + 3 + 5 + 7
## [1] 16
```

- The expression 1+3+5+7 was sent to the processor, and the result 16 was printed in the console
- (Later on we will discuss the `[1]` part)
- Note: the value 16 is not kept in in the RAM or Mass Storage, just printed on screen

Figure 31: In memory

Expressions

- The input and output appears like this in the slides -

```
1 + 3 + 5 + 7
## [1] 16
```

- And here is how it appears in RStudio -

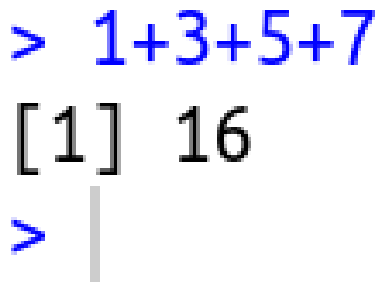


Figure 32: RStudio console input and output

Expressions

- We can type a number, the number itself is returned -

```
600
## [1] 600
```

- We can type text inside single ' or double " quotes -

```
"Hello"
## [1] "Hello"
```

- Both of these are constant values, `numeric` or `text`, the simplest type of expressions in R

Arithmetic operators

- Through interactive use of the command line we can experiment with basic operators in R
- For example, R includes the standard **arithmetic operators**

Table 1: Arithmetic operators

Operator	Meaning
+	Addition
-	Subtraction
*	Multiplication
/	Division
^	Exponent

Arithmetic operators

- For example -

```
5 + 3
## [1] 8
4 - 5
## [1] -1
1 * 10
## [1] 10
1 / 10
## [1] 0.1
10 ^ 2
## [1] 100
```

Note: We can use the **up** $\uparrow$ and **down** $\downarrow$ keys to scroll through the executed expressions history

Exponential notation

- Note that very large or very small numbers are formatted in **exponential** notation -

```
1 / 1000000 # 1*10^-6
## [1] 1e-06
```

```
7 * 100000  # 7*10^5
## [1] 7e+05
```

Infinity

- **Infinity** is treated as a special numeric value `Inf` or `-Inf` -

```
1 / 0
## [1] Inf
```

```
-1 / 0
## [1] -Inf
```

```
Inf + 1
## [1] Inf
```

```
-1 * Inf
## [1] -Inf
```

Expressions

- We can control operator precedence with **brackets**, just like in math
- This is recommended for clarity even where not strictly required

```
2 * 3 + 1
## [1] 7
```

```
2 * (3 + 1)
## [1] 8
```

Comments

- The interpreter ignores everything to the right of **number sign** # -

```
1 * 2 # * 3
## [1] 2
```

- It is therefore used for **code comments** -

```
# Multiplication example
5 * 5
## [1] 25
```

Expressions

- The interpreter ignores **spaces**, so the following expressions are treated exactly the same way -

```
1 + 1
## [1] 2
```

```
1+1
## [1] 2
```

```
1+           1
## [1] 2
```

Expressions

- We can type **Enter** in the middle of an expression and keep typing on the next line
- The interpreter displays the + symbol, which means that the expression is **incomplete** -

```
5 *
2
## [1] 10
```

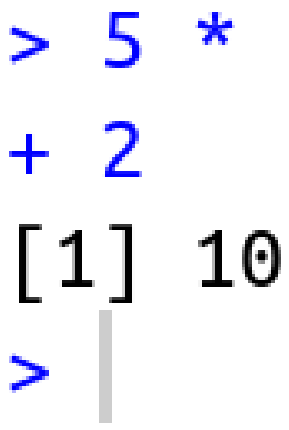


Figure 33: Incomplete expression

Conditional operators

- **Conditions** are expression that use **conditional operators** and have a yes/no result, i.e. the condition can be either true or false
- The result of a condition is a **logical** value, `TRUE` or `FALSE`
  - `TRUE` means the expression is true
  - `FALSE` means the expression is false
  - (`NA` means it is unknown)

Conditional operators

Table 2: Conditional operators

Operator	Meaning
==	Equal
>	Greater than
>=	Greater than or equal
<	Less than
<=	Less than or equal
!=	Not equal
&	And
|	Or
!	Not

Conditional operators

- For example, we can use **conditional operators** to compare numeric values -

```
1 < 2
## [1] TRUE
1 > 2
## [1] FALSE
2 > 2
## [1] FALSE
2 >= 2
## [1] TRUE
2 != 2
## [1] FALSE
```

Conditional operators

- **“Equal”** == and **“not equal”** != are opposites of each other: a pair of values can be either equal or not

```
1 == 1
## [1] TRUE
1 == 2
## [1] FALSE
```

```
1 != 1
## [1] FALSE
1 != 2
## [1] TRUE
```

Conditional operators

- The **"and"** & and **"or"** | operators are used to create more complex conditions
- **"And"** & returns `TRUE` when **both** sides are `TRUE`

```
(1 < 10) & (10 < 100)
## [1] TRUE
(1 < 10) & (10 > 100)
## [1] FALSE
```

- **"Or"** | returns `TRUE` when **at least one** of the sides is `TRUE` -

```
(1 < 10) | (10 < 100)
## [1] TRUE
(1 < 10) | (10 > 100)
## [1] TRUE
```

Conditional operators

- The last conditional operator is **"not"** `!`, which reverses `TRUE` to `FALSE` and `FALSE` to `TRUE` -

```
1 == 1
## [1] TRUE
!(1 == 1)
## [1] FALSE
```

```
(1 == 1) & (2 == 2)
## [1] TRUE
(1 == 1) & !(2 == 2)
## [1] FALSE
```

Conditional operators

- Question: run the following expression and explain their result -

```
FALSE == FALSE
## [1] TRUE
```

```
!(TRUE == TRUE)
## [1] FALSE
```

```
!(!(1 == 1))
## [1] TRUE
```

Special values

Table 3: Special values in R

Value	Meaning
`Inf`	Infinity
`NA`	Not Available
`NaN`	Not a Number
`NULL`	Empty object

Special values

- For example -

```
1/0
## [1] Inf
```

```
NA + 3
## [1] NA
```

```
0/0
## [1] NaN
```

```
NULL
## NULL
```

Functions

- In math, a **function** is a relation that associates each element x of a set X, to a single element y of another set Y
- For example, the function $y = 2x$

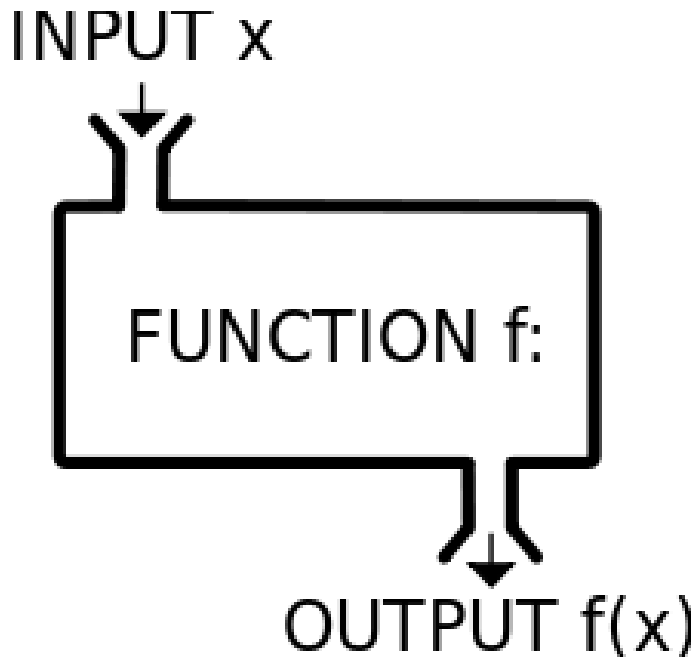


Figure 34: A function

Functions

- The concept of functions in programming is similar -
  - A function is a code piece that "knows" how to do a certain **task**
  - Executing the function is known as a **function call**
  - The function accepts zero or more objects as **input** (e.g. 2) and returns a single object as output (e.g. 4), possibly also doing other things known as **side effects**
  - The number and type of inputs the function needs are determined in the function definition; these are known as the function **parameters** (e.g. a single number)
  - The objects the function received in practice, as part of a particular function call, are known as **arguments** (e.g. 2)

Functions

- A function is basically a set of pre-defined **instructions**
- There are thousands of **built-in** functions in R
- Later on we will learn to **define** our own functions

> To understand computations in R, two slogans are helpful:
>
> - Everything that exists is an object.
> - Everything that happens is a function call.

```
`+`(5, 5)
## [1] 10
```

[22] Chambers 2014, Statistical Science (https://arxiv.org/pdf/1409.3531.pdf)

Functions

- A **function call** is composed of the function name, followed by the arguments inside brackets `()` and separated by commas `,` -

```
sqrt(4)
## [1] 2
```

- The `sqrt` function received a single argument `4` and returned its square root 2

Error messages

- Consider the following expressions -

```
sqrt(16)
## [1] 4
```

```
sqrt("a")
## Error in sqrt("a"): non-numeric argument to mathematica
```

```
sqrt(a)
## Error in eval(expr, envir, enclos): object 'a' not foun
```

Error messages

- In the previous slide we got two **error messages**, because the last two expressions were illegal, i.e. not in agreement with the syntax rules of R
- The **first error** was occurred because we tried to run a mathematical operation `sqrt` on a text value `"a"`
- The **second error** occurred because we tried to use a non-existing object `a`. And text without quotes is treated as a name of an object, i.e. a label for an actual object stored in RAM. Since we don't have an object named `a` we got an error.

Pre-loaded objects

- When starting R, a default set of **objects** is loaded into the RAM, such as `TRUE`, `FALSE`, `sqrt` and `pi`
- Type `pi` and see what happens -

```
pi
## [1] 3.141593
```

Decimal places

- Is the value of PI stored in memory really equal to `3.141593`?

```
pi == 3.141593
## [1] FALSE
```

- If not, what is the difference?

```
pi - 3.141593
## [1] -3.464102e-07
```

- The reason is that by default R prints only the first 7 digits

```
options()$digits
## [1] 7
```

Case-sensitivity

- R is **case-sensitive**, it distinguishes between lower-case and upper-case letters
- For example, `TRUE` is a logical value, but `True` and `true` are undefined -

```
TRUE
## [1] TRUE
```

```
True
## Error in eval(expr, envir, enclos): object 'True' not f
```

```
true
## Error in eval(expr, envir, enclos): object 'true' not f
```

Classes

- R is an object-oriented language, where each object belongs to a **class**
- The `class` functions accepts an object and returns the **class name** -

```
class(TRUE)
## [1] "logical"
class(1)
## [1] "numeric"
class(pi)
## [1] "numeric"
class("a")
## [1] "character"
class(sqrt)
## [1] "function"
```

Classes

- Question: explain the returned value of the following expressions -

```
class(1 < 2)
## [1] "logical"
```

```
class("logical")
## [1] "character"
```

```
class(1) == class(2)
## [1] TRUE
```

Classes

- Question: explain the returned value of the following expressions -

```
class(class)
## [1] "function"
```

```
class(class(sqrt))
## [1] "character"
```

```
class(class(1))
## [1] "character"
```

Using help files

- Every built-in object is associated with a **help document**, which can be accessed using the `help` function or the `?` operator -

```
help(class)
?class
?TRUE
?pi
```