POIR 613: Computational Social Science

Pablo Barberá

University of Southern California
pablobarbera.com

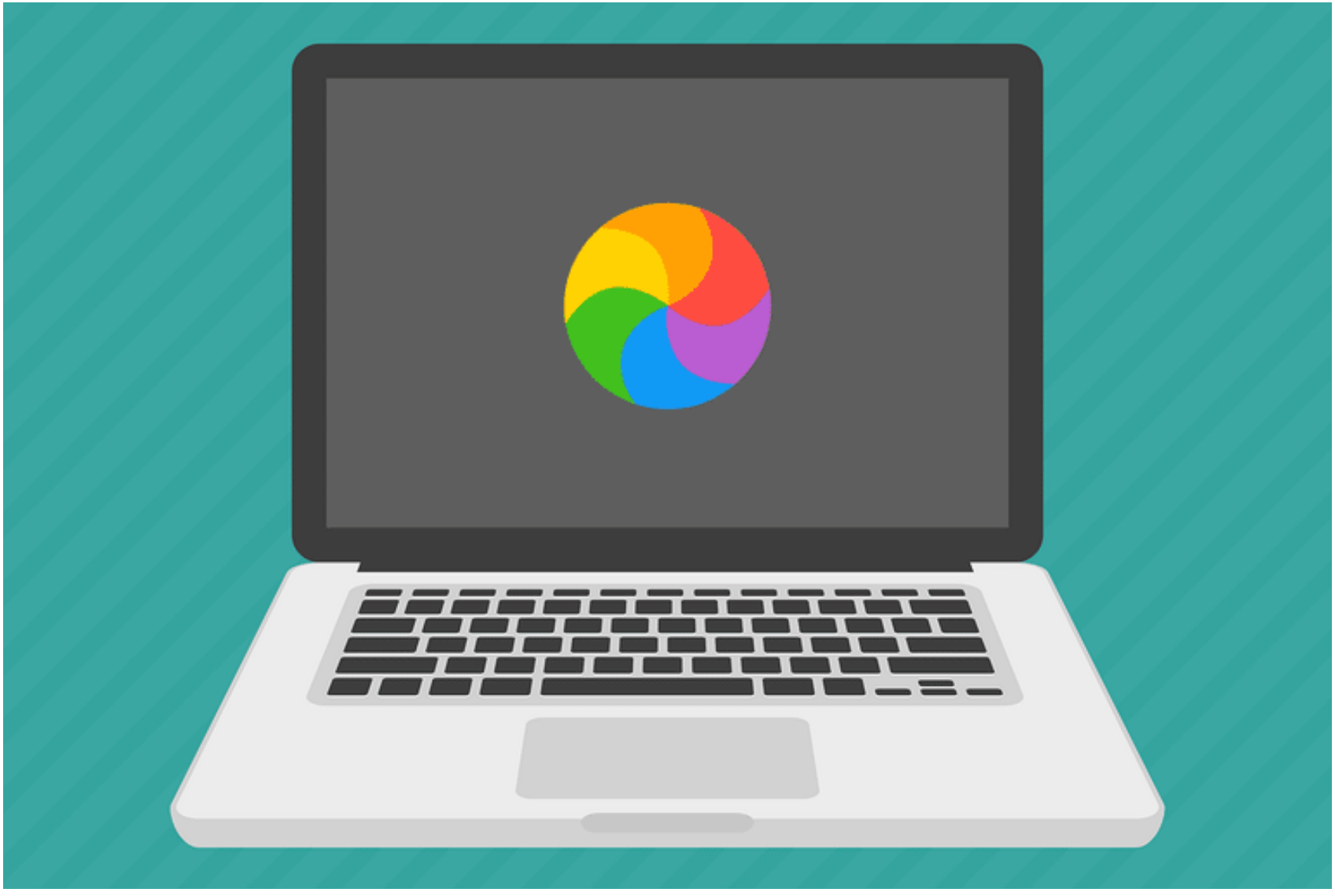
Course website:
pablobarbera.com/POIR613/

Outline

What we will discuss here:

- Efficient data analysis with R
- Guided coding session:
  - Loops and functions in R
  - Algorithm complexity
  - Examples of good coding practices in R

Efficient data analysis with R



Myths about R as programming language

1. R is an interpreted language, so it must be slow
   - Interpreted = executes code directly without compiling
   - Compiled code = code executed natively on CPU (fast!)
   - BUT: many functions are written in C and C++ and thus run in fast machine code
   - Slow code can be written more efficiently
2. All objects in R are stored in memory
   - You cannot open datasets larger than RAM
   - BUT: most laptops now have 8+ GB of RAM (+virtual mem)
   - `bigmemory` package: work with files on disk
   - Easy to work with large databases in the cloud
3. R only uses one core of your CPU
   - Unlike STATA, no multi-core computing out of the box
   - BUT: many functions and packages now take advantage of multi-core computers
   - Easy to write your own code to do parallel computing

My data is too big! My code is too slow!

What to do?

1. Buy a better computer or expand RAM memory
2. Write more efficient code
3. Use parallel computing – more on that later this semester
4. Move your code/data to the cloud
5. Use out-of-memory storage: SQL databases, bigmemory package, Hadoop...

Detour: algorithm complexity

- You can define the efficiency (*aka* complexity) of code (*algorithm*) in two different ways:
  1. Time: how long it takes to run
  2. Space: how much memory it uses
- These can be defined in terms proportional to the size of your data – "Big O" notation; e.g. $O(n)$, $O(\log n)$, $O(n^2)$
- Time and space complexity can be quite different! (more on this later)

Writing efficient R code (Part I)

- Conventional wisdom: avoid for loops at all costs!
- But simply rewriting loops will not make code faster
- Key: use vectorized functions instead of loops
- Why are vectorized function fast? They use vector filtering, which means loop is done in machine native code
  - Takes vector as input and return vector as output
  - Some vectorized functions: `ifelse()`, `which()`, `rowSums()`, `colSums()`, `sum()`, `any()`, `rnorm()`...

Writing efficient R code (Part II)

- A common bottleneck is memory re-allocation, e.g.:

```
result <- c()
for (i in 1:n){
   result[i] <- x[i] + y[i]
}
```

- In iteration, `R` re-sizes the vector and re-allocates memory
- For large operations (e.g. data frames), this can make your code really slow
- Solution: pre-allocate vector size:

```
result <- rep(NA, n)
for (i in 1:n){
   result[i] <- x[i] + y[i]
}
```