Data Science for Economists

Functions and Parallel Programming

Kyle Coombs
Bates College | ECON/DCS 368

Table of contents

Prologue

Prologue

- By the end of class you will:
    - Be able to write basic functions in R
    - Be able to iterate tasks serially and in parallel in R
    - Be able to bootstrap in parallel in R

Attribution

I pull most of this lecture from the textbook Data Science in R by James Scott

Functions

What is a function?

- In math, a function is a mapping from domain to range

$$\begin{aligned} f(x) &= x^2 \quad \text{Takes a number from the domain and returns its square in the range} \\ f(2) &= 4 \quad \text{The function applied to 2 returns 4} \end{aligned}$$

- In programming, a function is a mapping from input to output

```r
exponentiate <- function(x,p=2) {
  x^p
}

exponentiate(x=2) # Returns 4
```

```
## [1] 4
```

The functions: why and how

Why write functions?

- **Abstraction**
  - Summarize complex operations into single lines of code that are easier to remember
- **Automation**
  - Automate a task to happen many times without having to write the same code over and over
- **Documentation**
  - Well-written and named functions are "self-documenting," so you can remember what you did

How do I write a function?

In R, functions are defined using the `function` keyword

```
some_function <- function(positional_input1=1,positional_input2="two",keyword_inputs) {
  # Do something with these inputs
  # Create output or ouputs
  return(output) # Return the output
  # If you do not specify return, it returns the last object
}
```

`function` takes keyword inputs and positional inputs or "arguments." The order of the inputs is important unless you specify otherwise!

Control flow: If/else logic

```r
square_ifelse <- function(x = NULL) {
    if (is.null(x)) { ## Start multi-line IF statement with {
      x = 1 # Default value
      message("No input value provided. Using default value of 1.") ## Message to users:
      }               ## Close multi-line if statement with }
    x_sq = x^2
    d = data.frame(value = x, value_squared = x_sq)
    return(d)
  }
print(square_ifelse())
```

```
## No input value provided. Using default value of 1.
```

```
##   value value_squared
## 1     1             1
```

```r
print(square_ifelse(2))
```

```
##   value value_squared
## 1     2             4
```

This function has a default value of 1 for when you fail to provide a value.

Each step of bootstrap

```r
# library(tidyverse) # Already loaded
set.seed(1)
df ← tibble(x = rnorm(1000, mean = 0, sd = 1),
  y= x+rnorm(1000, mean = 0, sd = 1))

bootstrap_sample ← function(df) {
  # 1. Draw a random sample with replacement of size N from your sample.
  sample ← df %>% sample_frac(1, replace = TRUE)
  # 2. Perform the same analysis, here a median, on the new sample.
  return(coef(feols(y ~ x, data = sample))[2])
}

bootstrap_sample(df)
```

```
##         x
## 0.9671832
```

Aside: What's a seed?

- The `set.seed()` function sets the seed for the random number generator
- If you set the seed to the same number, you will get the same random numbers each time
- This is important for reproducibility

More on functions

- There is a lot more to functions!
- Check out Grant McDermott's Introductory and Advanced chapters on functions
- There are some incredible tips on how to:
  - Debug functions
  - Write functions that are easy to read
  - Catch errors
  - Cache or `memoise` big functions

Iteration

Iteration: For loops

- You've likely heard of for loops before![1]
- They're the most common way to iterate across programming languages
- In R, the syntax is fairly simple:

```r
for(i in 1:10) {
  print(exponentiate(i))
}
```

```
## [1] 1
## [1] 4
## [1] 9
## [1] 16
## [1] 25
## [1] 36
## [1] 49
## [1] 64
## [1] 81
## [1] 100
```

Bootstrapping for loop

To save output, you have to pre-define a list where you deposit the output

```
deposit ← vector("list",10) # preallocate list of 10 values
set.seed(1)
for (i in 1:10) {
  # perform bootstrap
  deposit[[i]] ← bootstrap_sample(df)
}

bootstrapped_for ← bind_rows(deposit)
head(bootstrapped_for)
```

```
## # A tibble: 6 × 1
##        x
##    <dbl>
## 1 1.02
## 2 0.987
## 3 0.997
## 4 0.987
## 5 0.947
## 6 0.999
```

Binding output

- Did you notice the `bind_rows()` function I called?
- After any iteration that leaves you a bunch of dataframes in a list, you'll want to put them together
- The `bind_rows` function is a great way to bind together a list of data frames
- Other options include:
  - `do.call(rbind, list_of_dataframes)`
  - `data.table::rbindlist()`

Issues with for loops

- For loops are slow in R
- They clutter up your environment with extra variables (like the `i` indexer)
- They can also be an absolute headache to debug if they get too nested
- Look at the example below: this is a nested for loop that is hard to read and debug
- In some languages, this is all you have, but not in R!

```r
for (i in 1:5) {
  for (k in 1:5) {
    if (i > k) {
      print(i*k)
    }
    else {
      for (j in 1:5) {
        print(i*j*k)
      }
    }
  }
}
```

Tips on iterating

- Start small! Set your iteration to 1 or 2 and make sure it works
- Why?
    - You'll know faster if it broke
- Print where it is in the iteration (or use a progress bar with something like `pbapply`)

```r
for (i in 1:2) {
  print(i)
  # complex function
}
```

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

While loops

- I'm largely skipping while loops, but they're also important!
- While loops iterate until one or more conditions are met
    - Typically one condition is a max number of iterations
    - Another conditions is that the some value of the loop is within a small amount of a target value
- These are critical for numerical solvers, which are common in computational economics and machine learning

Iteration: apply family

- R has a much more commonly used approach to iteration: the `*apply` family of functions: `apply`, `sapply`, `vapply`, `lapply`, `mapply`
- The `*apply` family takes a function and applies it to each element of a list or vector
- `lapply` is the most commonly used and returns a list back

- `*apply` family is a little confusing at first
- Syntax is `*apply(list_or_vector, function, other_input)`
- The first input of the function will be the current element of the list/vector in the iteration
- `other_inputs` are next inputs passed to the function

```r
lapply(1:10, exponentiate,p=2)
```

```
## [[1]]
## [1] 1
##
## [[2]]
## [1] 4
##
## [[3]]
## [1] 9
##
## [[4]]
## [1] 16
##
## [[5]]
## [1] 25
##
## [[6]]
## [1] 36
##
## [[7]]
```

Bootstrapping lapply

- One trick: `*apply` insists on iterating over some sequence indexed `i` like a for-loop
- But you can ignore it by using `function(i)` and then not using `i` in the function

```r
set.seed(1)
lapply(1:10, function(i) bootstrap_sample(df=df)) %>%
  bind_rows()
```

```
## # A tibble: 10 × 1
##         x
##     <dbl>
##  1 1.02
##  2 0.987
##  3 0.997
##  4 0.987
##  5 0.947
##  6 0.999
##  7 0.966
##  8 0.983
##  9 0.987
## 10 0.987
```

Wrapper functions due to odd syntax

- Maybe you don't like the ugly syntax of `function(i)` and then not using `i` in the function
- Well you can write a wrapper function to get around that

```r
set.seed(1)
wrapper_bootstrap <- function(i, df) {
  bootstrap_sample(df)
}
lapply(1:10, wrapper_bootstrap, df=df) %>%
  bind_rows()
```

```
## # A tibble: 10 × 1
##         x
##     <dbl>
##   1 1.02
##   2 0.987
##   3 0.997
##   4 0.987
##   5 0.947
##   6 0.999
##   7 0.966
##   8 0.983
##   9 0.987
##  10 0.987
```

Iteration: map

- The **purrr** package introduces `map` functions, which are more intuitive with **tidyverse**
- The variant `map_df` is especially useful beause it automatically binds the output into a data frame
    - The same iteration syntax applies here too.

```
set.seed(1)
map_df(1:10, function(i) bootstrap_sample(df=df))
```

```
## # A tibble: 10 × 1
##         x
##     <dbl>
##   1 1.02
##   2 0.987
##   3 0.997
##   4 0.987
##   5 0.947
##   6 0.999
##   7 0.966
##   8 0.983
##   9 0.987
##  10 0.987
```

Parallel Programming

Parallel Programming

- Imagine you get home from the grocery store with 100 bags of groceries
- You have to bring them all inside, but you can only carry 2 at a time
- That's 50 trips back and forth, so how can you speed things up?

Parallel Programming

- Imagine you get home from the grocery store with 100 bags of groceries
- You have to bring them all inside, but you can only carry 2 at a time

- That's 50 trips back and forth, so how can you speed things up?

- Ask friends to carry to at a time with you (Parallel Programming)

- Get a cart and carry 10 at a time (more RAM and a better processor)

Parallel Programming

- Imagine you get home from the grocery store with 100 bags of groceries
- You have to bring them all inside, but you can only carry 2 at a time

- That's 50 trips back and forth, so how can you speed things up?

- Ask friends to carry to at a time with you (Parallel Programming)

- Get a cart and carry 10 at a time (more RAM and a better processor)

One trip? Okay ,sure

A warning

- Parallel Programming is an incredibly exponentiateful tool, but it is full of pitfalls
- A friend of mine from the PhD said that he did not understand it until the 4th year of his PhD
- Many economists understand the intuition, but not the details until they have to
- That used to be me until I started teaching this class!
- So if it is hard, that's normal. But it is worth learning!

Parallel Programming: What?

- Your computer has multiple cores, which are like multiple brains
- Each of these is capable of doing the same tasks
- Parallel Programming is the act of using multiple cores to do the same task at the same time

Parallel Programming: What?

- Your computer has multiple cores, which are like multiple brains
- Each of these is capable of doing the same tasks
- Parallel Programming is the act of using multiple cores to do the same task at the same time
- Many coding tasks are "embarassingly parallel"
    - That means they can be broken up into many small tasks that can be done at the same time
    - Bootstrapping is one such example
- Some "serial" tasks are not "embarrassingly parallel"
    - Still, parts of these tasks may be possible to do in parallel
- R has many Parallel Programming packages:
    - **future.apply** - today
    - **furrr** - today
    - **parallel** - today
    - **future**
    - **pbapply**
    - **foreach**
    - **doParallel**

Parallel Programming: Why?

- Parallel Programming is a great way to speed up your code and often there are straight-forward ways to do it
- It is not always worth doing:
  - Theoretically, the gain should be linear: each additional node should speed up your code by the same amount
  - In practice, there are "overhead" costs to Parallel Programming that can slow things down
  - **Overhead costs**: reading in and subsetting data, tracking each node

Across computer clusters

- Parallel Programming is also a way to speed up your code across multiple computers
- This is called "distributed computing"
- It is a way to speed up your code when you have a lot of data and a lot of computers
- Imagine you have 1000 computers, each with 1/1000th of your data
- You can run the same code on each computer, and then combine the results
- Same logic, but the "overhead" costs are higher

How many cores are there?

- You can find out how many cores you have with the `parallel::detectCores()` function

```
parallel::detectCores()
```

```
## [1] 8
```

- The more cores, the more you can speed up your code
- But remember, there are diminishing returns to Parallel Programming
  - If a task takes 10 minutes on 1 core, it might take 6 minutes on 2 cores, but 4 minutes on 4 cores

Trivial example: square numbers

- Let's start with some trivial to understand examples

- Here is a function called `slow_square`, which takes a number and squares it, but after a pause.

```
## Emulate slow function
slow_square =
  function(x = 1) {
    x_sq = x^2
    d = data.frame(value = x, value_squared = x_sq)
    Sys.sleep(2) # literally do nothing for two seconds
    return(d)
    }
```

Let's time that quickly.

```
# library(tictoc) ## Already loaded

tic()
serial_ex = lapply(1:12, slow_square)
toc(log = TRUE)
```

```
## 24.83 sec elapsed
```

Now in parallel

- `plan` multisession tells R to use multiple cores

```
# library(future.apply)  ## Already loaded
# plan(multisession)     ## Already set above

tic()
future_ex = future_lapply(1:12, slow_square)
toc(log = TRUE)
```

```
## 10 sec elapsed
```

```
all.equal(serial_ex, future_ex)
```

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

Example: bootstrapping in parallel

- The future_lapply works the same, but now I have to set the seed inside the function with `future.seed`
- Why? Because each node is a separate R session, so they need to coordinate their random numbers

```r
set.seed(1)
tic()
serial_boot ← lapply(1:1e4, function(i) bootstrap_sample(df)) %>%
  bind_rows()
toc(log = TRUE)
```

```
## 220.22 sec elapsed
```

```r
tic()
parallel_boot ← future_lapply(1:1e4,
  function(i) bootstrap_sample(df),
  future.seed=1) %>%
  bind_rows()
toc(log = TRUE)
```

```
## 82.45 sec elapsed
```

Want to use `map`? Try **furrr**

The **furrr** package, i.e. future **purrrr** is a Parallel Programming version of **purrr**

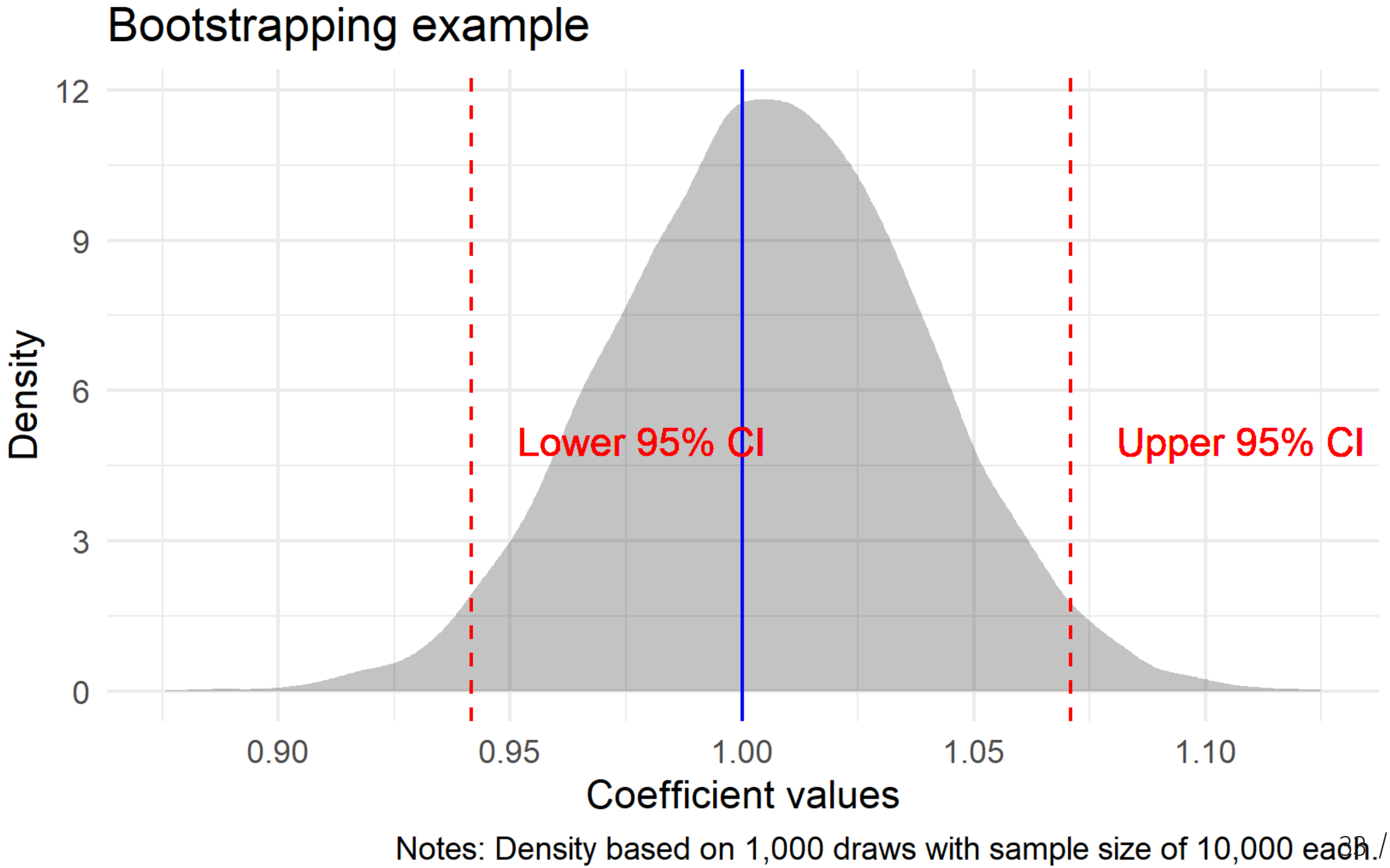
- Again, set the seed inside the function with `.options`.

```
tic()
furrr_boot = future_map_dfr(1:1e4,
  function(i) bootstrap_sample(df),
  .options = furrr_options(seed=1))
toc(log = TRUE)
```

```
## 92.55 sec elapsed
```

Get standard errors from results

- Now that we have a bunch of estimates, we can get the standard error of our estimates



Many R packages use Parallel

- Many R packages already use Parallel Programming
- `feols()` from **fixest** uses Parallel Programming to speed up regressions
  - You can control how using the `nthreads` input
- **data.table** uses Parallel Programming to speed up data wrangling
- **boot** and **sandwich** can use Parallel Programming to speed up bootstrapping
- And many others do the same

What next?

- Go try how to bootstrap in R!
- Better yet, learn to do it in parallel
- Navigate to the lecture activity 13a-bootstrapping-functions-practice

Next lecture: Machine Learning Intro

Parallel Programming vocab

The vocab for Parallel Programming can get a little confusing:

- **Socket**: A socket is a physical connection between a processor and the motherboard
- **Core**: A core is a physical processor that can do computations
- **Process**: A process is a task that is being done by a core (Windows users may know this from Task Manager)
- **Thread**: A thread is a subtask of a process that can be done in parallel and share memory with other threads
- **Cluster**: A cluster is a group of computers that can be used to do Parallel Programming
- **Node**: One computer within a cluster