Big Data and Economics

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

Questions

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

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

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

```r
square <- function(x) {
  x^2
}

square(2) # Returns 4
```

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

Why use functions?

- **Abstraction**
  - They allow you to summarize complex details into a single line of code, so you only need to understand them once (instead of repeating yourself)
- **Automation**
  - Automate a task to happen many times without having to write the same code over and over
- **Documentation**
  - Well-written functions codify the steps you take to do something, so you can easily 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. It does not require a specific order for these unlike in Python. But generally, position comes first.

Control flow: If/else logic

Functions make great use of if/else logic

```
square =
  function(x = NULL) {

    if (is.null(x)) { ## Start multi-line IF statement with {
      x = 1
      ## Message to users:
      message("No input value provided. Using default value of 1.")
      }               ## Close multi-line if statement with }

    x_sq = x^2
    d = data.frame(value = x, value_squared = x_sq)

    return(d)
  }
square()
```

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

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

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 %>% slice_sample(n = nrow(df), 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
```

Wrapping up a function

- You can wrap functions inside of other functions
- This is a great way to make your code more readable and modular
- Also useful for various iteration tasks that need to take an iterated input

```
wrapper_bootstrap <- function(i,df) {
  # print(i) # if you want to visualize the i.
  bootstrap_sample(df)
}
wrapper_bootstrap(1,df)
```

```
##         x
## 0.9824952
```

More on functions

- There is a lot more to functions than we can cover today
- Check out Grant McDermott's Introudctory 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!
- They're the most common way to iterate across programming languages
- In R, the syntax is fairly simple: you iterative over a vector or list of values, and do stuff with those values

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

Bootstrapping for loop

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

```r
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

```r
lapply(1:10, square)
```

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

`*apply` syntax

- The `*apply` family is a little confusing at first, but it's very powerful
- The syntax is `*apply(list_or_vector, function, other_arguments)`
- The `function` is a function that takes arguments like any other
  - The first argument will be the element of the list or vector
- The `other_arguments` are arguments that are passed to the function

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 to get around `*apply`

- 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)
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

- Sometimes the `*apply` syntax is a little confusing
- The **purrr** package in the tidyverse has more intuitive syntax for iteration: `map`
- 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

Motivating example: Parallel

- 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
- How can you speed things up?

Motivating example: Parallel

- 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

- 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)

A warning

- Parallel Programming is an incredibly powerful 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 and only do it if absolutely necessary
- That used to be me until I started teaching this class!
- So if it is hard, that's normal. But it is worth learning!

"One trip?" okay

One trip? Okay ,sure

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 tasks are not embarrassingly parallel
    - These are called "serial" tasks
    - Parts of these tasks may be possible to do in parallel

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

Parallel Programming in R

- In R, there are many ways to parallel process, I'll introduce you to the **future.apply** package
- There are many Parallel Programming packages in R, but **future.apply** follows the `*apply` family syntax

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

```r
# 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
```

```r
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

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

```
## 171.47 sec elapsed
```

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

```
## 86.38 sec elapsed
```

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

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

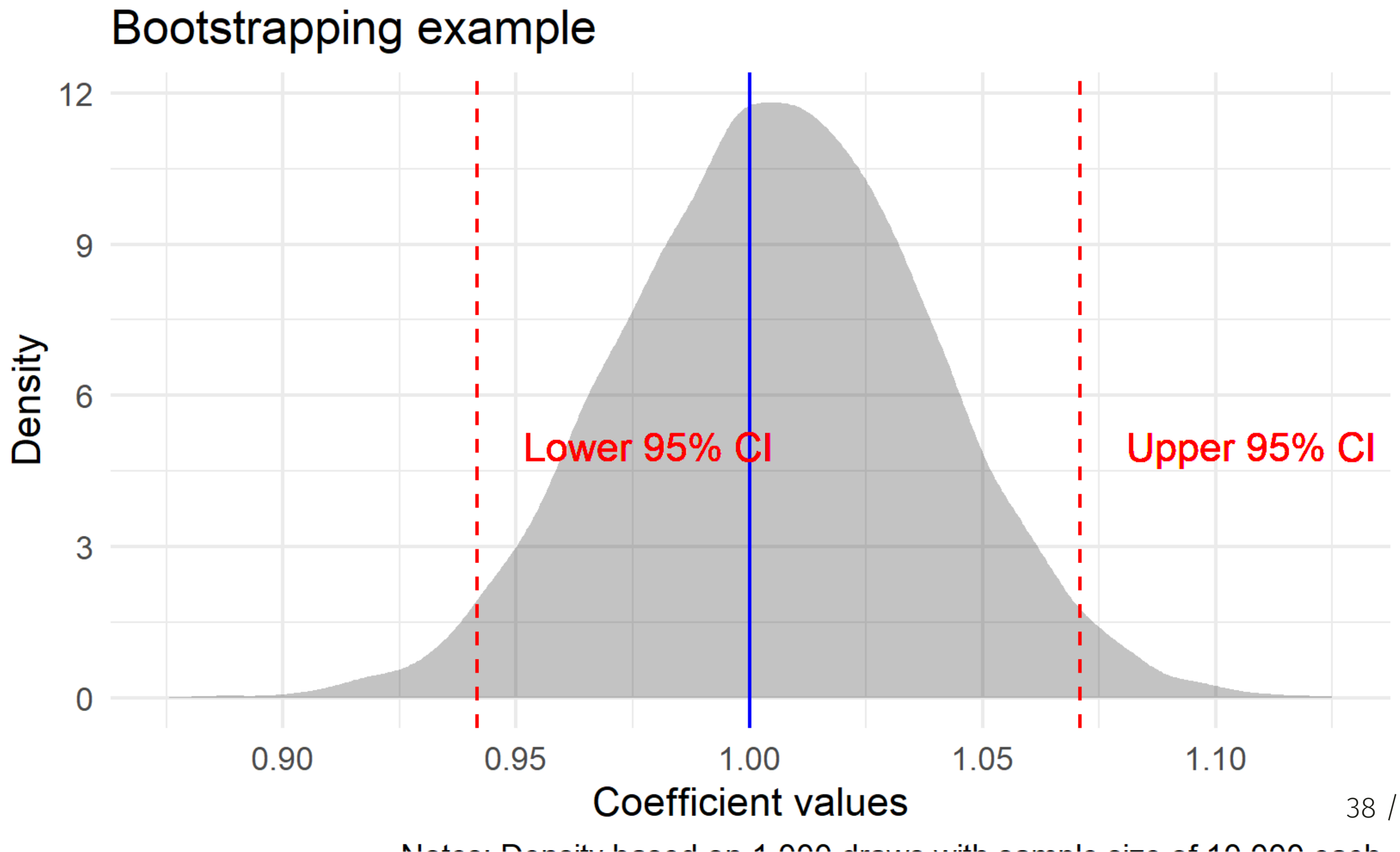
- Again, the syntax is the same, but you have to set the seed inside the function with `.options`.

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

```
## 71.95 sec elapsed
```

Get standard errors from results

- Now that we have a bunch of estimates, we can get the standard error of our estimates
- The 95% confidence interval is just the 2.5th and 97.5th percentile of the sampling distribution



R packages that use Parallel

- Many R packages already use Parallel Programming
- `feols()` from **fixest** uses Parallel Programming to speed up OLS estimation
  - You can control how using the `nthreads` argument
- **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

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

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