Functions, Loops,
& Automation

Announcements

- Class on April 8th will be pre-recorded to allow flexibility for those looking to see the last solar eclipse in the continental US for the next 20 years!
    - Next total solar eclipse in continental US is in **August of 2044**
- Spring Break next week
- Final Project Proposal Due on March 25th

1. Custom Functions

Custom Functions Overview

- Allows you to tailor a reproducible analysis
- Improves code readability and tidiness
- Essential for creating good automated workflows
- Can be flexibly called from a separate, dedicated file!
  - Example
    - > source(here(‘Scripts’, ‘Functions’, ‘Function1’))

Custom Function Basics

- You want to set the custom function to a variable
- Parameters are specified for the function
  - These act in the function's local environment
- Function General Syntax:
  - custom_function = **function**(parameter1, parameter2…) **{**
  - FUNCTION
  - **return**(OUTPUT) **}**
  - custom_function(parameterx, parametery)

Guidelines for Custom Functions - 1

- **Function Size**: Each function should perform one task or responsibility (ideally short length).
- **Readability**: Strive for readability. Sometimes, a more verbose but readable code is preferable over a compact, less intuitive solution.
- **Comments and Documentation**: Document your functions with comments. Explain what the function does, its parameters, return values, and any side effects.
- **Naming Conventions**: Use clear and descriptive names for functions and variables.

A word on naming conventions

- Function names should be descriptive to what the function does
  - Not too long, not too short!
- Do not overly abbreviate function names or parameter names!
  - Good: us_precip_23
  - Bad: p23
- Choose a naming style and stick with it, don’t mix and match
  - Good consistent variables
    - us_precip_23
    - us_precip_22
    - us_temp_23
  - Bad inconsistent variables
    - 23_precip_usa
    - precip22_us
    - us_t23

Guidelines for Custom Functions - 2

- **Parameters**: Functions should have well-defined parameters.
  - Use default parameter values where appropriate!
- **Avoid Global Variables**: Functions should rely on their input parameters and not on global variables! Take the time to setup an input parameter.
- **Return Values**: Functions should return values that are consistent in type and structure.
- **Error Handling**: Include error handling within your functions.
- **Testing**: Write tests for your functions to ensure they behave as expected.

Custom Function Example

```
# 1. Basic Function
poly_overlap <- function(points, polygon) {
  intersects <- st_intersects(points, polygon)
  overlap_logical <- lengths(intersects) > 0
  points$OverlapsWithPolygon <- overlap_logical
  return(points)
}
vp_prihab_overlap = poly_overlap(vp, prihab)
```

- Cool function, now let me read this one sec to figure out what it does…

Function Documentation

- Comments
- print() statements

Function Documentation - Good Example

```r
# 2. Basic Function w/ Comments
poly_overlap <- function(points, polygon) {
  # 1. Determine intersections of points & poly
  intersects <- st_intersects(points, polygon)
  # 2. Convert intersections to TRUE/FALSE values
  overlap_logical <- lengths(intersects) > 0
  # 3. Append a new column to points with it's intersection status
  points$OverlapsWithPolygon <- overlap_logical
  return(points)
}
vp_prihab_overlap = poly_overlap(vp, prihab)
```

Function Documentation - Better Example

```r
# 3. Basic Function w/ Comments + print() Statements
poly_overlap <- function(points, polygon) {
  # 1. Determine intersections of points & poly
  intersects <- st_intersects(points, polygon)
  print("Intersections calculated successfully!")
  # 2. Convert intersections to TRUE/FALSE values
  overlap_logical <- lengths(intersects) > 0
  # 3. Append a new column to points with it's intersection status
  points$OverlapsWithPolygon <- overlap_logical
  print("Overlap column successfully appended!")
  return(points)
}
```

```
> vp_prihab_overlap = poly_overlap(vp, prihab)
[1] "Intersections calculated successfully!"
[1] "Overlap column successfully appended!"
```

2. IF/ELSE

IF/ELSE

- IF/ELSE statements allow you to add logic to your functions
- They can also be used to do quality assurance checks on your data as a pre-processing method.
    - Ensures proper inputs and therefore predictable outputs
- IF/ELSE Syntax:

```
if (LOGICAL) {
  # Code if LOGICAL == TRUE
} else {
  # Code if LOGICAL == FALSE
}
```

IF/ELSE - Example

```r
# Define input number
number <- -5

# Use if else statement to check if the number is positive
if (number > 0) {
  print("The number is positive.")
} else {
  print("The number is not positive.")
}
```

```
[1] "The number is not positive."
```

IF/ELSE/ELSE IF - Example

```r
# Define input number
number <- 0

# Use if, else if, and else statements to check the number's status
if (number > 0) {
  print("The number is positive.")
} else if (number < 0) {
  print("The number is negative.")
} else {
  print("The number is zero.")
}
```

```
[1] "The number is zero."
```

Custom Functions - IF/ELSE

```
# 4. Basic Function w/ Documentation & IF/ELSE Verification
poly_overlap <- function(points, polygon) {
  # 0. Pre-Check your inputs to ensure they are ready for analysis
  if (!inherits(points, "sf") || !inherits(polygon, "sf") ||
      st_crs(points) != st_crs(polygon)) {
    stop("Ensure 'points' is a POINT sf object, 'polygon' is a POLYGON sf object,
         and both have the same CRS.")
  } else {print("Points & Polygons are both sf objects with consistent CRS!")}
  # 1. Determine intersections of points & poly
  intersects <- st_intersects(points, polygon)
  print("Intersections calculated successfully!")
  # 2. Convert intersections to TRUE/FALSE values
  overlap_logical <- lengths(intersects) > 0
  # 3. Append a new column to points with it's intersection status
  points$OverlapsWithPolygon <- overlap_logical
  print("Overlap column successfully appended!")
  return(points)
}
vp_prihab_overlap = poly_overlap(vp, prihab)
```

3. Loops

Loops

- Loops allow you to repeat lines of code for a specified amount of time.
- Depending on how you setup your loop, you can have a new input with each cycle of repetition the loop does.
- Loops use **for(index in __) { CODE }**
- The index of a loop is the logic that defines how many times, and what inputs will change with each iteration
    - Your index can be numbers, characters, booleans etc

Loops - Names Example

```
jnames_vec = c("Jack", "Jeremy", "James", "Jim")

for (names in jnames_vec) {
  print(names)
}
[1] "Jack"
[1] "Jeremy"
[1] "James"
[1] "Jim"
```

Index

Vector containing new value for each loop iteration.

- Notice how we looped through each name from jnames_vec
- names was my index in this case

Loops - Example

```
for (i in 1:5) {
  print(i)
}
```

```
[1] 1
[1] 2
[1] 3
[1] 4
[1] 5
```

Quick refresher of the : syntax

```
> 1:5
[1] 1 2 3 4 5
```

Loop Example - Break

```
for (i in 1:5) {
  if (i == 4) {
    break # Exit the loop when i is 4
  }
  print(i)
}
[1] 1
[1] 2
[1] 3
```

- We no longer get 4 or 5 because the loop stopped at 4

Loop Example - Next

```
for (i in 1:5) {
  if (i == 4) {
    next # Ignore 4 and continue loop
  }
  print(i)
}
[1] 1
[1] 2
[1] 3
[1] 5
```

- Unlike break, next will continue the loop instead of stopping it entirely

Loops - Nesting Example

```
for (i in 1:3) {
  for (j in letters[1:3]) {
    print(paste("Number:", i, "Letter:", j))
  }
}
```

```
[1] "Number: 1 Letter: a"
[1] "Number: 1 Letter: b"
[1] "Number: 1 Letter: c"
[1] "Number: 2 Letter: a"
[1] "Number: 2 Letter: b"
[1] "Number: 2 Letter: c"
[1] "Number: 3 Letter: a"
[1] "Number: 3 Letter: b"
[1] "Number: 3 Letter: c"
```

1. For the duration of the j loop, the i will be the same.
2. Once j loop is complete, the i loop will move to the next value
    - (1 ->2, 2->3)

Loop Techniques - Pre-Allocate Storage

1. Create a storage dataframe / list
2. Append loop outputs to storage dataframe / list

- Pre-allocating storage is great because it streamlines loop processes

Loop Techniques - Pre-Allocate Storage Example

```r
# Pre-allocate an empty data frame for storage of loop output
df <- data.frame(Number = integer(),
                 Letter = character(),
                 stringsAsFactors = FALSE)

# Nested loop to append each combination of number and letter to the data frame
for (i in 1:3) {
  for (j in letters[1:3]) {
    # Dynamically append a new row to the data frame
    df <- rbind(df, data.frame(Number = i,
                               Letter = j,
                               stringsAsFactors = FALSE))
  }
}
```

Number	Letter
1	a
1	b
1	c
2	a
2	b
2	c
3	a
3	b
3	c

Loop Etiquette

- Use description index names
- Minimize work done inside of the loop
- Pre-Allocate a storage vector/dataframe/list etc

4. Apply Family

What is the Apply Family in R?

- A family a functions that aims to efficiently apply functions to multiple elements in R simultaneously.
- Sometimes a more efficient alternative to looping.
- Each function within the apply family is designed for a slightly different input and output scheme (vector vs matrix vs list).

apply() Function Family

- apply():
  - Use: Matrices.
  - Operation: Applies a function over margins (rows or columns).
  - Output: Array or list, depending on the function applied.
- sapply():
  - Use: Lists or vectors.
  - Operation: Applies a function element-wise.
  - Output: Simplified to vector or matrix if possible, else list.
- lapply():
  - Use: Lists or vectors.
  - Operation: Applies a function element-wise.
  - Output: List.

lapply()

- lapply() or “list apply” applies a function to each input X, and outputs a list

lapply(X, FUN, …)

  - X: The input list or vector. lapply() will iterate over each element of X.
  - FUN: The function to be applied to each element of X.
  - … : Additional parameters required by FUN
- Iteration: lapply() iterates over each element in the input list X.
- Application: It applies the function FUN to each element individually.
- Output: The results of applying FUN to each element are collected into a list.

Disclaimer

This is an example, and is by no means the best way to do this analysis.

Pay attention to the way things operate!

lapply() - Example

```r
# Load Data
points   = st_read(here('data', 'Adelges_tsugae.shp'))
polygon  = st_read(here('data', 'polygon.shp'))
polygon1 = st_read(here('data', 'polygon1.shp'))

# We want to process multiple polygons, and lapply() takes a list!
polygons = list(polygon, polygon1)

# Use lapply to apply st_intersects for each polygon
results_lapply = lapply(polygons, function(poly) {
  st_intersects(points, poly)
})
```

1. lapply() is going to apply each polygon in the polygons list to the input function
2. Function in this case determines points that intersect each polygon

lapply() - Example

Name	Type	Value
results_lapply	list [2]	List of length 2
[[1]]	list [3323 x 918] (S3: sgbp, list)	List of length 3323
[[2]]	list [3323 x 262] (S3: sgbp, list)	List of length 3323
[[1]]	integer [0]	
[[2]]	integer [0]	
[[3]]	integer [1]	16
[[4]]	integer [0]	
[[5]]	integer [0]	

PTS

OBJECTID of intersecting polygon

- lapply() returns a list, which can be weird to work with if unfamiliar

do.call()

- Allows you to call a function with arguments specified in typically a list form.
- do.call(what, args)
  - what: The function you want to call, do not use ()
    - Example: To do.call() the cbind() function, you would say do.call(cbind, …, …)
  - args: A list format input for the function to act on
- Example:
  - list = list(df1, df2, df3)
  - **combined_df = do.call(rbind, list)**
  - # Will call the rbind function with arguments df1, df2, and df3.
  - # Output: A dataframe with df1-3 combined

Combining lapply() & do.call()

- lapply() will output a list, but you may not always want a list output!
- do.call() can bridge the gap between lapply() and tidy dataframes

Combining lapply() & do.call() - Example

```
# Extract outputs from list, and append to points
intersections = do.call(cbind, results_lapply_NAfix)
points = cbind(points, intersections)
```

V1	V2
NA	NA
NA	NA
1	16
NA	NA
NA	NA
NA	NA
NA	NA
1	16
NA	NA
NA	NA

→

X	Scientific.Name	Country	X1	X2	geometry
1	Adelges tsugae	US	NA	NA	POINT (-77.53387 39.8529)
2	Adelges tsugae	US	NA	NA	POINT (-74.08302 41.98527)
3	Adelges tsugae	US	1	16	POINT (-71.0758 42.44546)
4	Adelges tsugae	US	NA	NA	POINT (-77.945 42.67337)
5	Adelges tsugae	US	NA	NA	POINT (-80.28418 36.3993)
6	Adelges tsugae	US	NA	NA	POINT (-77.48811 43.24503)
7	Adelges tsugae	US	NA	NA	POINT (-77.4816 43.12731)
8	Adelges tsugae	US	1	16	POINT (-71.0814 42.44976)
9	Adelges tsugae	US	NA	NA	POINT (-76.76864 39.67685)
10	Adelges tsugae	US	NA	NA	POINT (-78.50717 37.99465)

Always check other functions before designing an analysis!

```
# How to do this analysis a bit more easily
points_with_poly  = st_join(points, polygon,
                            join = st_intersects)
points_with_poly_poly1 = st_join(points_with_polygon, polygon1,
                                 join = st_intersects)
```

Loops vs. Apply

- Loops
    - Good for:
        - Highly complicated logic
        - Readability
        - Refined iteration control is more straightforward
    - Cons:
        - Poor performance
        - More lines of code for simple operations
- Apply
    - Good for:
        - Performance
        - Conciseness
        - ~ Elegant ~
    - Cons:
        - Readability
        - Not as flexible as loops

5. A tidbit about design

5 minutes of planning will save you hours of coding

Ask yourself some questions before starting any complex analysis

1. What output do I want? A dataframe? A list? An sf or raster?
2. What is the problem that I need to solve, and
   do any functions already solve my problem?
3. If no functions solve my problem, which functions will help solve it?
   - st_area() for area, st_distance() for distance etc.
4. What approach is appropriate?
   - Stepwise analysis? Looping? Apply?

Tips for planning an analysis

- Try to map out the workflow before you start writing any code
- Use comments to set the groundwork for what you want to do
  - Example:
    - # 1. Load in Data
    - # 2. Prepare Data for Analysis
    - # 2.1 Subset Data
    - # 2.2 Filter Data
    - # 3. Calculate ____
    - # 4. Loop through ___ to get ___
    - # …
- This will help identify gaps in your analytical process, and make you think about what your methods are more deeply.

6. Efficiency

Disclaimer: Code Speed vs. Readability

- More often than not **readable slow code >>> unreadable fast code**
- Code speed is important though, so knowing how to make your code faster is important for computationally expensive analytics.

Maximizing Efficiency in R Automated Analytics

- Vectorization
- Avoid creating objects in a loop
- Avoid expensive reads and writes
- Find faster packages!
    - Tidyverse is great for this because it has efficiency at the forefront of it’s design
- Parallel Processing

Vectorization

- Vectorization is a technique that **applies operations to whole matrices or vectors at once** rather than iterating over elements individually.
- Examples include
  - Arithmetic operations (+, -, *, /)
  - Logical operations (<, >, ==)
  - Mathematical functions (log, exp, sin, cos).
- Easy ways to incorporate vectorization into your workflow
  1. **<u>Use vectorized packages</u>** like the tidyverse & the apply() family
  2. Avoid loops

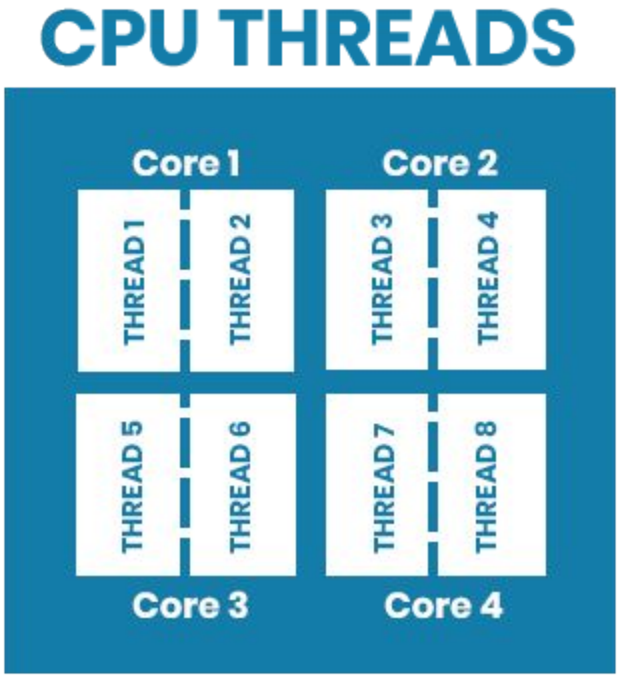
Avoiding Loops

- apply()
- purr from the tidyverse

Parallel Processing - Background

- **CPU** (Central Processing Unit): The brain of the computer where most calculations take place. It's like an office that can perform tasks (calculations).
- **Core**: Imagine a core as an individual worker in the office (CPU). A CPU can have multiple cores (workers), and each core (worker) can do tasks.
- **Thread**: A thread is like a single task assigned to a worker. A core can work on one or more threads at a time.
- **Multicore Architecture**: CPUs have multiple cores (workers), allowing the computer to perform multiple tasks simultaneously more efficiently.

Parallel Processing - Background cont.



https://emeraldforhome.com/cpu-cores-vs-threads/

- Each core can work on task/process.
- A thread can work on parts of a process.
- A single process can contain multiple threads, all of which may run tasks concurrently, called **multithreading**.
- A single process can also utilize multiple cores, called **multicore processing**.

Parallel Processing - Fundamentals

- By default most R operations are single core processes, which is a fraction of your computer's full capabilities.
- Parallel processing enables utilization of **Multiple Cores** by distributing tasks across the CPU, allowing for simultaneous computation.
- Types of Parallelism
  - **Task** parallelism: executing different tasks at the same time
    - Each Core/Thread is working on a separate task/process
  - **Data** parallelism: splitting data into chunks and processing those chunks simultaneously
    - Each Core/Thread is working on the same task, but breaking it up for efficiency
    - Multiple cores can be working on the same task in this way as well!

Parallel Processing - How to Apply

- You have two options
  1. Use packages which leverage parallel processing (avoid those that don't)
  2. Enable parallel processing with an external package
- Parallel enabling Packages
  - **parallel**, foreach, future, and doParallel
- My recommendation is to only look into enabling parallel processing with an external package if your code doesn't run well within the limits of practicality.
  - It will be normal for some analytics to take a LONG time, even with parallel processing

Code Benchmarking

- Benchmarking code in R is essential for optimizing performance, especially when dealing with large datasets or complex computations.
- Several packages can help you measure and compare the execution time of R code snippets.
- Packages
  - microbenchmark
  - rbenchmark
  - bench

Code Benchmarking - Example

```
require(microbenchmark)
benchmark_result <- microbenchmark(
  lapply_process = {
    polygons = list(polygon, polygon1)
    results_lapply = lapply(polygons, function(poly) st_intersects(points, poly))
    results_lapply_NAfix = lapply(results_lapply, function(sublist) {
      lapply(sublist, function(x) if(length(x) == 0) NA else x)
    })
    intersections = do.call(cbind, results_lapply_NAfix)
  },
  times = 5 # Number of times to repeat the test for averaging
)
```

```
> print(benchmark_result)
Unit: milliseconds
           expr      min       lq     mean   median       uq     max neval
 lapply_process 320.1164 327.6615 338.4764 328.6283 331.3268 384.649     5
```

Let’s compare methods

Long, impractical method benchmark

FASTER!

```
> print(benchmark_result)
Unit: milliseconds
           expr      min       lq     mean   median       uq     max neval
 lapply_process 320.1164 327.6615 338.4764 328.6283 331.3268 384.649     5
```

Quick st_join() method benchmark

```
> print(benchmark_result_join)
Unit: milliseconds
         expr      min       lq     mean   median       uq      max neval
 join_process 480.2103 486.2634 548.1355 493.1964 598.8912 682.1164     5
```

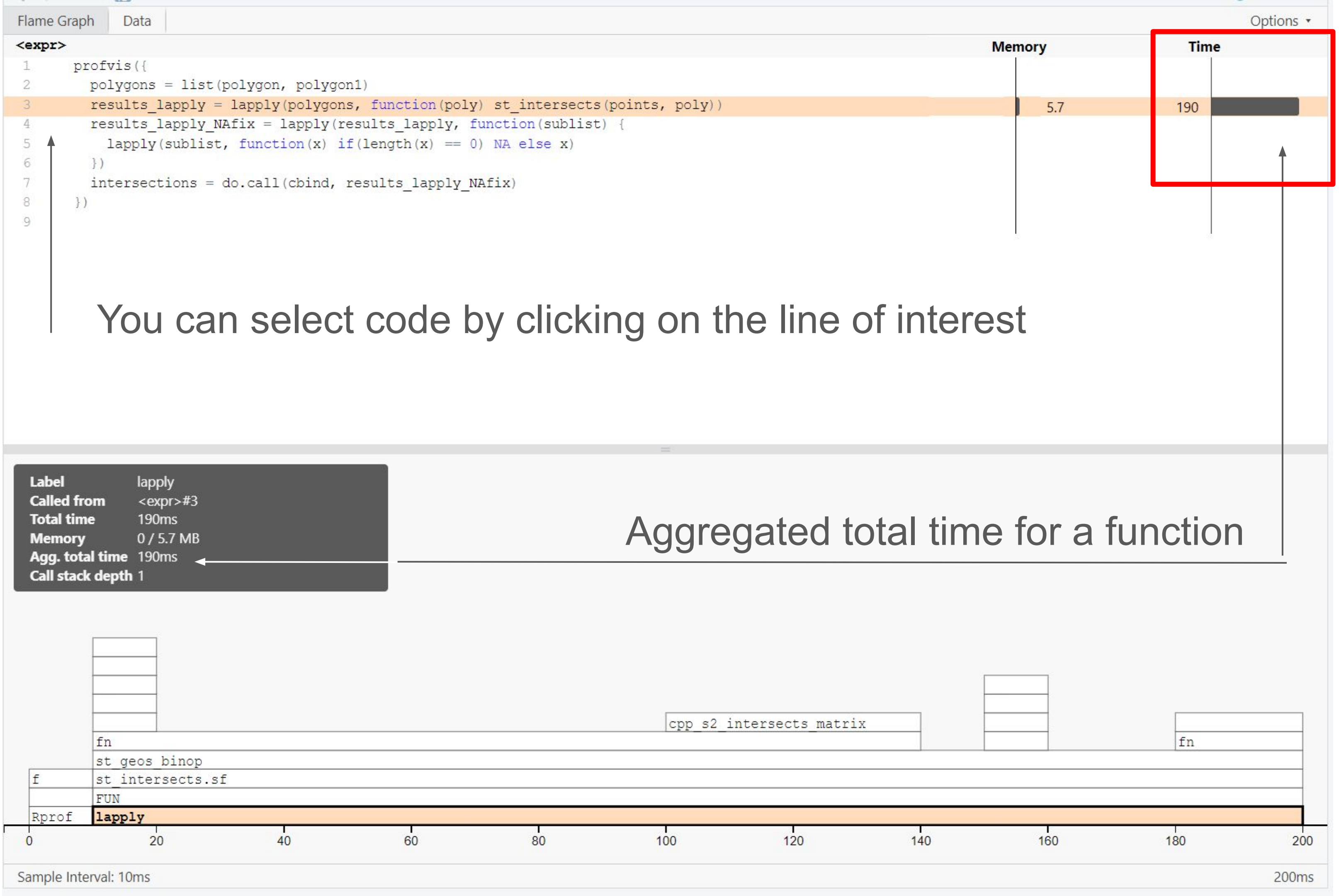
Code Profiling

- Profiling code in R is essential for identifying bottlenecks and optimizing the performance of your scripts or applications.
- Will tell you which parts of your code took the longest to run, and allows you to target that code for adjustment.
- Packages
  - utils::RProf
    - A part of base R
  - **Profvis**
  - microbenchmark
  - bench

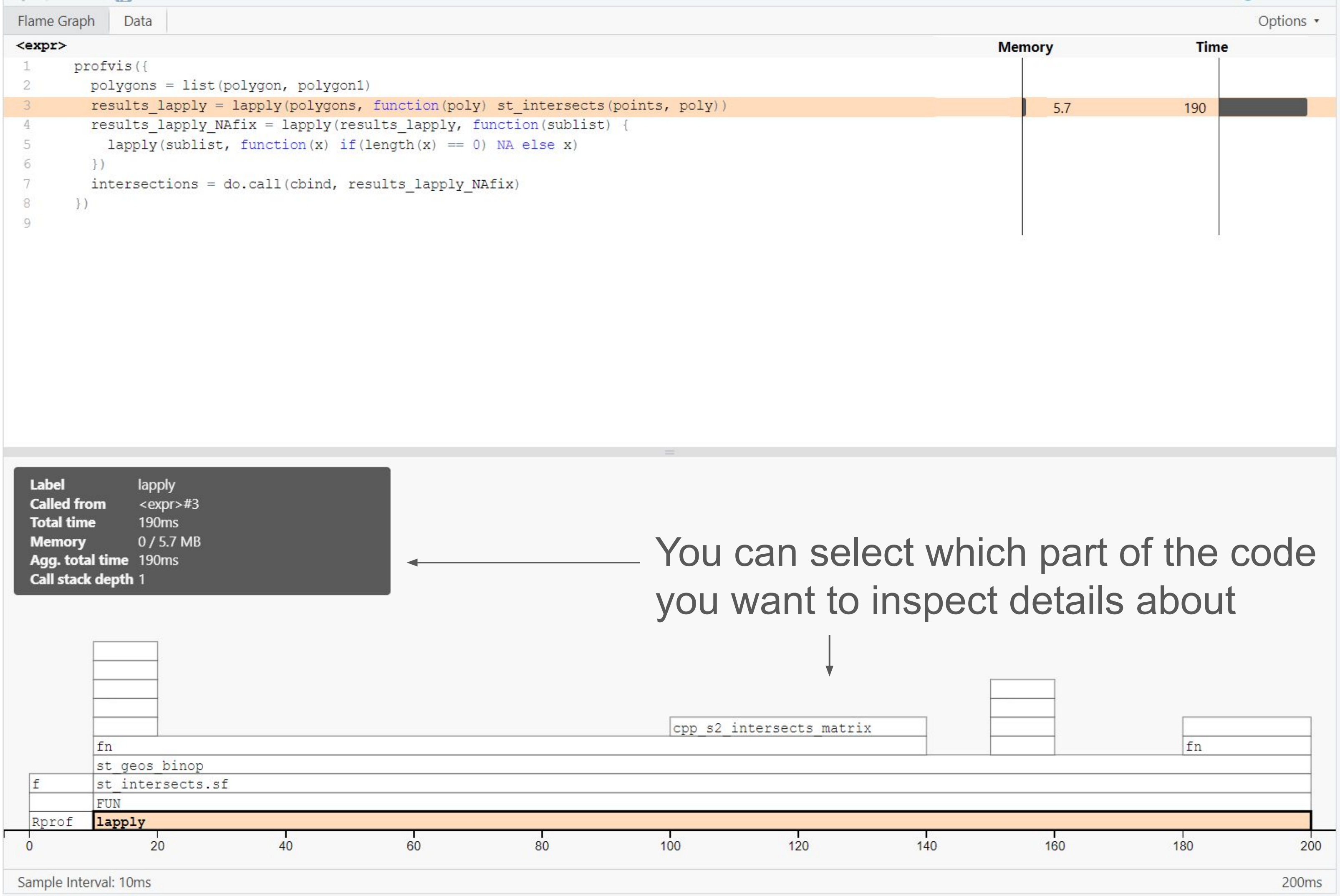
Code Profiling - Example

```
require(profvis)
profvis({
  polygons = list(polygon, polygon1)
  results_lapply = lapply(polygons, function(poly) st_intersects(points, poly))
  results_lapply_NAfix = lapply(results_lapply, function(sublist) {
    lapply(sublist, function(x) if(length(x) == 0) NA else x)
  })
  intersections = do.call(cbind, results_lapply_NAfix)
})
```

- It really is as simple as running a function with the code you want inside of it
- If you have multiple lines, utilize a { CODE } format like what is shown above



Flame Graph
Data
Options
<expr>
Memory
Time
profvis({
polygons = list(polygon, polygon1)
results_lapply = lapply(polygons, function(poly) st_intersects(points, poly))
results_lapply_NAfix = lapply(results_lapply, function(sublist) {
lapply(sublist, function(x) if(length(x) == 0) NA else x)
})
intersections = do.call(cbind, results_lapply_NAfix)
})
5.7
190
You can select code by clicking on the line of interest
Label lapply
Called from <expr>#3
Total time 190ms
Memory 0 / 5.7 MB
Agg. total time 190ms
Call stack depth 1
Aggregated total time for a function
cpp_s2_intersects_matrix
fn
fn
st_geos_binop
f
st_intersects.sf
FUN
Rprof
lapply
0
20
40
60
80
100
120
140
160
180
200
Sample Interval: 10ms
200ms



You can select which part of the code you want to inspect details about

Flame Graph | Data | Options

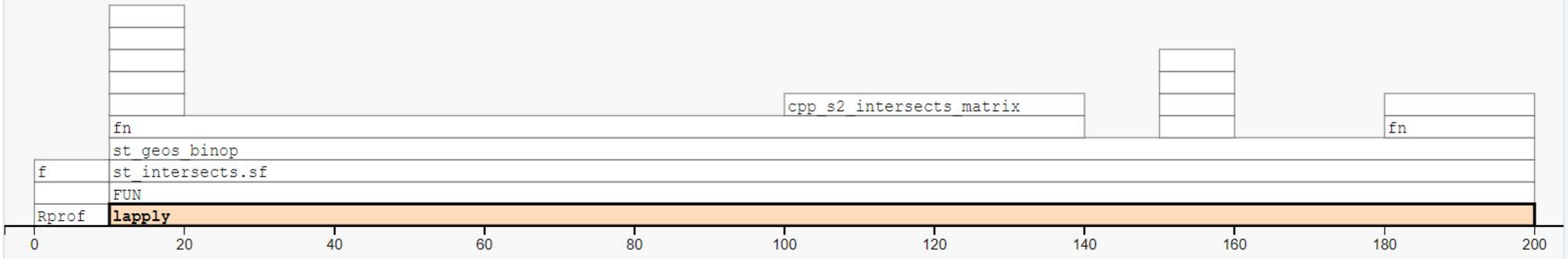
<expr> — Memory — Time

```r
profvis({
  polygons = list(polygon, polygon1)
  results_lapply = lapply(polygons, function(poly) st_intersects(points, poly))
  results_lapply_NAfix = lapply(results_lapply, function(sublist) {
    lapply(sublist, function(x) if(length(x) == 0) NA else x)
  })
  intersections = do.call(cbind, results_lapply_NAfix)
})
```

Line 3: Memory 5.7, Time 190

These parts with no time reported or memory usage were run under the sample interval of 10ms

Label	lapply
Called from	<expr>#3
Total time	190ms
Memory	0 / 5.7 MB
Agg. total time	190ms
Call stack depth	1



Sample Interval: 10ms 200ms

Good luck!