10

Functional programming

R, at its heart, is a functional programming (FP) language. This means that it provides many tools for the creation and manipulation of functions. In particular, R has what's known as first class functions. You can do anything with functions that you can do with vectors: you can assign them to variables, store them in lists, pass them as arguments to other functions, create them inside functions, and even return them as the result of a function.

The chapter starts by showing a motivating example, removing redundancy and duplication in code used to clean and summarise data. Then you'll learn about the three building blocks of functional programming: anonymous functions, closures (functions written by functions), and lists of functions. These pieces are twined together in the conclusion which shows how to build a suite of tools for numerical integration, starting from very simple primitives. This is a recurring theme in FP: start with small, easy-to-understand building blocks, combine them into more complex structures, and apply them with confidence.

The discussion of functional programming continues in the following two chapters: Chapter 11 explores functions that take functions as arguments and return vectors as output, and Chapter 12 explores functions that take functions as input and return them as output.

Outline

- Section 10.1 motivates functional programming using a common problem: cleaning and summarising data before serious analysis.
- Section 10.2 shows you a side of functions that you might not have known about: you can use functions without giving them a name.
- Section 10.3 introduces the closure, a function written by another function. A closure can access its own arguments, and variables defined in its parent.

© 2015 by Taylor & Francis Group, LLC

- Section 10.4 shows how to put functions in a list, and explains why you might care.
- Section 10.5 concludes the chapter with a case study that uses anonymous functions, closures and lists of functions to build a flexible toolkit for numerical integration.

Prequisites

You should be familiar with the basic rules of lexical scoping, as described in Section 6.2. Make sure you've installed the pryr package with `install.packages("pryr")`

10.1 Motivation

Imagine you've loaded a data file, like the one below, that uses −99 to represent missing values. You want to replace all the −99s with `NA`s.

```
# Generate a sample dataset
set.seed(1014)
df <- data.frame(replicate(6, sample(c(1:10, -99), 6, rep = TRUE)))
names(df) <- letters[1:6]
df
#>    a  b c   d   e f
#> 1  1  6 1   5 -99 1
#> 2 10  4 4 -99   9 3
#> 3  7  9 5   4   1 4
#> 4  2  9 3   8   6 8
#> 5  1 10 5   9   8 6
#> 6  6  2 1   3   8 5
```

When you first started writing R code, you might have solved the problem with copy-and-paste:

```
df$a[df$a == -99] <- NA
df$b[df$b == -99] <- NA
df$c[df$c == -98] <- NA
df$d[df$d == -99] <- NA
```

© 2015 by Taylor & Francis Group, LLC

Downloaded by [The University of British Columbia Library] at 10:45 13 July 2016

Downloaded by [The University of British Columbia Library] at 10:45 13 July 2016

```
df$e[df$e == -99] <- NA
df$f[df$g == -99] <- NA
```

One problem with copy-and-paste is that it's easy to make mistakes. Can you spot the two in the block above? These mistakes are inconsistencies that arose because we didn't have an authorative description of the desired action (replace −99 with `NA`). Duplicating an action makes bugs more likely and makes it harder to change code. For example, if the code for a missing value changes from −99 to 9999, you'd need to make the change in multiple places.

To prevent bugs and to make more flexible code, adopt the "do not repeat yourself", or DRY, principle. Popularised by the "pragmatic programmers" (`http://pragprog.com/about`), Dave Thomas and Andy Hunt, this principle states: "every piece of knowledge must have a single, unambiguous, authoritative representation within a system". FP tools are valuable because they provide tools to reduce duplication.

We can start applying FP ideas by writing a function that fixes the missing values in a single vector:

```
fix_missing <- function(x) {
  x[x == -99] <- NA
  x
}
df$a <- fix_missing(df$a)
df$b <- fix_missing(df$b)
df$c <- fix_missing(df$c)
df$d <- fix_missing(df$d)
df$e <- fix_missing(df$e)
df$f <- fix_missing(df$e)
```

This reduces the scope of possible mistakes, but it doesn't eliminate them: you can no longer accidentally type -98 instead of -99, but you can still mess up the name of variable. The next step is to remove this possible source of error by combining two functions. One function, `fix_missing()`, knows how to fix a single vector; the other, `lapply()`, knows how to do something to each column in a data frame.

`lapply()` takes three inputs: `x`, a list; `f`, a function; and `...`, other arguments to pass to `f()`. It applies the function to each element of the list and returns a new list. `lapply(x, f, ...)` is equivalent to the following for loop:

© 2015 by Taylor & Francis Group, LLC

```
out <- vector("list", length(x))
for (i in seq_along(x)) {
  out[[i]] <- f(x[[i]], ...)
}
```

The real `lapply()` is rather more complicated since it's implemented in C for efficiency, but the essence of the algorithm is the same. `lapply()` is called a **functional**, because it takes a function as an argument. Functionals are an important part of functional programming. You'll learn more about them in Chapter 11.

We can apply `lapply()` to this problem because data frames are lists. We just need a neat little trick to make sure we get back a data frame, not a list. Instead of assigning the results of `lapply()` to `df`, we'll assign them to `df[]`. R's usual rules ensure that we get a data frame, not a list. (If this comes as a surprise, you might want to read Section 3.3.) Putting these pieces together gives us:

```
fix_missing <- function(x) {
  x[x == -99] <- NA
  x
}
df[] <- lapply(df, fix_missing)
```

This code has five advantages over copy and paste:

- It's more compact.
- If the code for a missing value changes, it only needs to be updated in one place.
- It works for any number of columns. There is no way to accidentally miss a column.
- There is no way to accidentally treat one column differently than another.
- It is easy to generalise this technique to a subset of columns:

  ```
  df[1:5] <- lapply(df[1:5], fix_missing)
  ```

The key idea is function composition. Take two simple functions, one which does something to every column and one which fixes missing values, and combine them to fix missing values in every column. Writing

Downloaded by [The University of British Columbia Library] at 10:45 13 July 2016

© 2015 by Taylor & Francis Group, LLC

simple functions that can be understood in isolation and then composed is a powerful technique.

What if different columns used different codes for missing values? You might be tempted to copy-and-paste:

```
fix_missing_99 <- function(x) {
  x[x == -99] <- NA
  x
}
fix_missing_999 <- function(x) {
  x[x == -999] <- NA
  x
}
fix_missing_9999 <- function(x) {
  x[x == -999] <- NA
  x
}
```

As before, it's easy to create bugs. Instead we could use closures, functions that make and return functions. Closures allow us to make functions based on a template:

```
missing_fixer <- function(na_value) {
  function(x) {
    x[x == na_value] <- NA
    x
  }
}
fix_missing_99 <- missing_fixer(-99)
fix_missing_999 <- missing_fixer(-999)

fix_missing_99(c(-99, -999))
#> [1]   NA -999
fix_missing_999(c(-99, -999))
#> [1] -99  NA
```

Downloaded by [The University of British Columbia Library] at 10:45 13 July 2016

© 2015 by Taylor & Francis Group, LLC

Extra argument

In this case, you could argue that we should just add another argument:

```
fix_missing <- function(x, na.value) {
  x[x == na.value] <- NA
  x
}
```

That's a reasonable solution here, but it doesn't always work well in every situation. We'll see more compelling uses for closures in Section 11.5.

Now consider a related problem. Once you've cleaned up your data, you might want to compute the same set of numerical summaries for each variable. You could write code like this:

```
mean(df$a)
median(df$a)
sd(df$a)
mad(df$a)
IQR(df$a)

mean(df$b)
median(df$b)
sd(df$b)
mad(df$b)
IQR(df$b)
```

But again, you'd be better off identifying and removing duplicate items. Take a minute or two to think about how you might tackle this problem before reading on.

One approach would be to write a summary function and then apply it to each column:

```
summary <- function(x) {
  c(mean(x), median(x), sd(x), mad(x), IQR(x))
}
lapply(df, summary)
```

Downloaded by [The University of British Columbia Library] at 10:45 13 July 2016

© 2015 by Taylor & Francis Group, LLC

That's a great start, but there's still some duplication. It's easier to see if we make the summary function more realistic:

```
summary <- function(x) {
 c(mean(x, na.rm = TRUE),
   median(x, na.rm = TRUE),
   sd(x, na.rm = TRUE),
   mad(x, na.rm = TRUE),
   IQR(x, na.rm = TRUE))
}
```

All five functions are called with the same arguments (`x` and `na.rm`) repeated five times. As always, duplication makes our code fragile: it's easier to introduce bugs and harder to adapt to changing requirements.

To remove this source of duplication, you can take advantage of another functional programming technique: storing functions in lists.

```
summary <- function(x) {
  funs <- c(mean, median, sd, mad, IQR)
  lapply(funs, function(f) f(x, na.rm = TRUE))
}
```

This chapter discusses these techniques in more detail. But before you can start learning them, you need to learn the simplest FP tool, the anonymous function.

10.2 Anonymous functions

In R, functions are objects in their own right. They aren't automatically bound to a name. Unlike many languages (e.g., C, C++, Python, and Ruby), R doesn't have a special syntax for creating a named function: when you create a function, you use the regular assignment operator to give it a name. If you choose not to give the function a name, you get an **anonymous function**.

You use an anonymous function when it's not worth the effort to give it a name:

Downloaded by [The University of British Columbia Library] at 10:45 13 July 2016

© 2015 by Taylor & Francis Group, LLC

```
lapply(mtcars, function(x) length(unique(x)))
Filter(function(x) !is.numeric(x), mtcars)
integrate(function(x) sin(x) ^ 2, 0, pi)
```

Like all functions in R, anonymous functions have `formals()`, a `body()`, and a parent `environment()`:

```
formals(function(x = 4) g(x) + h(x))
#> $x
#> [1] 4
body(function(x = 4) g(x) + h(x))
#> g(x) + h(x)
environment(function(x = 4) g(x) + h(x))
#> <environment: R_GlobalEnv>
```

You can call an anonymous function without giving it a name, but the code is a little tricky to read because you must use parentheses in two different ways: first, to call a function, and second to make it clear that you want to call the anonymous function itself, as opposed to calling a (possibly invalid) function *inside* the anonymous function:

```
# This does not call the anonymous function.
# (Note that "3" is not a valid function.)
function(x) 3()
#> function(x) 3()

# With appropriate parenthesis, the function is called:
(function(x) 3)()
#> [1] 3

# So this anonymous function syntax
(function(x) x + 3)(10)
#> [1] 13

# behaves exactly the same as
f <- function(x) x + 3
f(10)
#> [1] 13
```

You can call anonymous functions with named arguments, but doing so is a good sign that your function needs a name.

One of the most common uses for anonymous functions is to create

Downloaded by [The University of British Columbia Library] at 10:45 13 July 2016

© 2015 by Taylor & Francis Group, LLC

closures, functions made by other functions. Closures are described in the next section.

10.2.1 Exercises

1. Given a function, like `"mean"`, `match.fun()` lets you find a function. Given a function, can you find its name? Why doesn't that make sense in R?
2. Use `lapply()` and an anonymous function to find the coefficient of variation (the standard deviation divided by the mean) for all columns in the `mtcars` dataset.
3. Use `integrate()` and an anonymous function to find the area under the curve for the following functions. Use Wolfram Alpha (http://www.wolframalpha.com/) to check your answers.
   1. `y = x ^ 2 - x`, x in [0, 10]
   2. `y = sin(x) + cos(x)`, x in [-π, π]
   3. `y = exp(x) / x`, x in [10, 20]
4. A good rule of thumb is that an anonymous function should fit on one line and shouldn't need to use `{}`. Review your code. Where could you have used an anonymous function instead of a named function? Where should you have used a named function instead of an anonymous function?

10.3 Closures

"An object is data with functions. A closure is a function with data." — John D. Cook

One use of anonymous functions is to create small functions that are not worth naming. Another important use is to create closures, functions written by functions. Closures get their name because they **enclose** the environment of the parent function and can access all its variables. This is useful because it allows us to have two levels of parameters: a parent level that controls operation and a child level that does the work.

The following example uses this idea to generate a family of power functions in which a parent function (`power()`) creates two child functions (`square()` and `cube()`).

Downloaded by [The University of British Columbia Library] at 10:45 13 July 2016

© 2015 by Taylor & Francis Group, LLC

```
power <- function(exponent) {
  function(x) {
    x ^ exponent
  }
}

square <- power(2)
square(2)
#> [1] 4
square(4)
#> [1] 16

cube <- power(3)
cube(2)
#> [1] 8
cube(4)
#> [1] 64
```

When you print a closure, you don't see anything terribly useful:

```
square
#> function(x) {
#>     x ^ exponent
#>   }
#> <environment: 0x7fda09128f20>
cube
#> function(x) {
#>     x ^ exponent
#>   }
#> <environment: 0x7fda04748a60>
```

That's because the function itself doesn't change. The difference is the enclosing environment, `environment(square)`. One way to see the contents of the environment is to convert it to a list:

```
as.list(environment(square))
#> $exponent
#> [1] 2
as.list(environment(cube))
#> $exponent
#> [1] 3
```

Downloaded by [The University of British Columbia Library] at 10:45 13 July 2016

© 2015 by Taylor & Francis Group, LLC

Downloaded by [The University of British Columbia Library] at 10:45 13 July 2016

Another way to see what's going on is to use `pryr::unenclose()`. This function replaces variables defined in the enclosing environment with their values:

```
library(pryr)
unenclose(square)
#> function (x)
#> {
#>     x^2
#> }
unenclose(cube)
#> function (x)
#> {
#>     x^3
#> }
```

The parent environment of a closure is the execution environment of the function that created it, as shown by this code:

```
power <- function(exponent) {
  print(environment())
  function(x) x ^ exponent
}
zero <- power(0)
#> <environment: 0x7fda04cc0b18>
environment(zero)
#> <environment: 0x7fda04cc0b18>
```

The execution environment normally disappears after the function returns a value. However, functions capture their enclosing environments. This means when function a returns function b, function b captures and stores the execution environment of function a, and it doesn't disappear. (This has important consequences for memory use, see Section 18.2 for details.)

In R, almost every function is a closure. All functions remember the environment in which they were created, typically either the global environment, if it's a function that you've written, or a package environment, if it's a function that someone else has written. The only exception is primitive functions, which call C code directly and don't have an associated environment.

Closures are useful for making function factories, and are one way to manage mutable state in R.

© 2015 by Taylor & Francis Group, LLC

10.3.1 Function factories

A function factory is a factory for making new functions. We've already seen two examples of function factories, `missing_fixer()` and `power()`. You call it with arguments that describe the desired actions, and it returns a function that will do the work for you. For `missing_fixer()` and `power()`, there's not much benefit in using a function factory instead of a single function with multiple arguments. Function factories are most useful when:

- The different levels are more complex, with multiple arguments and complicated bodies.
- Some work only needs to be done once, when the function is generated.

Function factories are particularly well suited to maximum likelihood problems, and you'll see a more compelling use of them in Section 11.5.

10.3.2 Mutable state

Having variables at two levels allows you to maintain state across function invocations. This is possible because while the execution environment is refreshed every time, the enclosing environment is constant. The key to managing variables at different levels is the double arrow assignment operator (`<<-`). Unlike the usual single arrow assignment (`<-`) that always assigns in the current environment, the double arrow operator will keep looking up the chain of parent environments until it finds a matching name. (Section 8.4 has more details on how it works.)

Together, a static parent environment and `<<-` make it possible to maintain state across function calls. The following example shows a counter that records how many times a function has been called. Each time `new_counter` is run, it creates an environment, initialises the counter `i` in this environment, and then creates a new function.

```
new_counter <- function() {
  i <- 0
  function() {
    i <<- i + 1
    i
  }
}
```

Downloaded by [The University of British Columbia Library] at 10:45 13 July 2016

© 2015 by Taylor & Francis Group, LLC

The new function is a closure, and its enclosing environment is the environment created when `new_counter()` is run. Ordinarily, function execution environments are temporary, but a closure maintains access to the environment in which it was created. In the example below, closures `counter_one()` and `counter_two()` each get their own enclosing environments when run, so they can maintain different counts.

```
counter_one <- new_counter()
counter_two <- new_counter()

counter_one()
#> [1] 1
counter_one()
#> [1] 2
counter_two()
#> [1] 1
```

The counters get around the "fresh start" limitation by not modifying variables in their local environment. Since the changes are made in the unchanging parent (or enclosing) environment, they are preserved across function calls.

What happens if you don't use a closure? What happens if you use `<-` instead of `<<-`? Make predictions about what will happen if you replace `new_counter()` with the variants below, then run the code and check your predictions.

```
i <- 0
new_counter2 <- function() {
  i <<- i + 1
  i
}
new_counter3 <- function() {
  i <- 0
  function() {
    i <- i + 1
    i
  }
}
```

Modifying values in a parent environment is an important technique because it is one way to generate "mutable state" in R. Mutable state is normally hard because every time it looks like you're modifying an

© 2015 by Taylor & Francis Group, LLC

Downloaded by [The University of British Columbia Library] at 10:45 13 July 2016

object, you're actually creating and then modifying a copy. However, if you do need mutable objects and your code is not very simple, it's usually better to use reference classes, as described in Section 7.4.

The power of closures is tightly coupled with the more advanced ideas in Chapter 11 and Chapter 12. You'll see many more closures in those two chapters. The following section discusses the third technique of functional programming in R: the ability to store functions in a list.

10.3.3 Exercises

1. Why are functions created by other functions called closures?
2. What does the following statistical function do? What would be a better name for it? (The existing name is a bit of a hint.)

   ```
   bc <- function(lambda) {
     if (lambda == 0) {
       function(x) log(x)
     } else {
       function(x) (x ^ lambda - 1) / lambda
     }
   }
   ```

3. What does `approxfun()` do? What does it return?
4. What does `ecdf()` do? What does it return?
5. Create a function that creates functions that compute the ith central moment (http://en.wikipedia.org/wiki/Central_moment) of a numeric vector. You can test it by running the following code:

   ```
   m1 <- moment(1)
   m2 <- moment(2)

   x <- runif(100)
   stopifnot(all.equal(m1(x), 0))
   stopifnot(all.equal(m2(x), var(x) * 99 / 100))
   ```

6. Create a function `pick()` that takes an index, `i`, as an argument and returns a function with an argument `x` that subsets `x` with `i`.

   ```
   lapply(mtcars, pick(5))
   # should do the same as this
   lapply(mtcars, function(x) x[[5]])
   ```

© 2015 by Taylor & Francis Group, LLC

Downloaded by [The University of British Columbia Library] at 10:45 13 July 2016

Downloaded by [The University of British Columbia Library] at 10:45 13 July 2016

10.4 Lists of functions

In R, functions can be stored in lists. This makes it easier to work with groups of related functions, in the same way a data frame makes it easier to work with groups of related vectors.

We'll start with a simple benchmarking example. Imagine you are comparing the performance of multiple ways of computing the arithmetic mean. You could do this by storing each approach (function) in a list:

```
compute_mean <- list(
  base = function(x) mean(x),
  sum = function(x) sum(x) / length(x),
  manual = function(x) {
    total <- 0
    n <- length(x)
    for (i in seq_along(x)) {
      total <- total + x[i] / n
    }
    total
  }
)
```

Calling a function from a list is straightforward. You extract it then call it:

```
x <- runif(1e5)
system.time(compute_mean$base(x))
#>    user  system elapsed
#>   0.000   0.000   0.001
system.time(compute_mean[[2]](x))
#>    user  system elapsed
#>       0       0       0
system.time(compute_mean[["manual"]](x))
#>    user  system elapsed
#>   0.071   0.004   0.082
```

To call each function (e.g., to check that they all return the same results), use `lapply()`. We'll need either an anonymous function or a new named function, since there isn't a built-in function to handle this situation.

© 2015 by Taylor & Francis Group, LLC

```
lapply(compute_mean, function(f) f(x))
#> $base
#> [1] 0.499
#>
#> $sum
#> [1] 0.499
#>
#> $manual
#> [1] 0.499

call_fun <- function(f, ...) f(...)
lapply(compute_mean, call_fun, x)
#> $base
#> [1] 0.499
#>
#> $sum
#> [1] 0.499
#>
#> $manual
#> [1] 0.499
```

To time each function, we can combine `lapply()` and `system.time()`:

```
lapply(compute_mean, function(f) system.time(f(x)))
#> $base
#>    user  system elapsed
#>   0.001   0.000   0.000
#>
#> $sum
#>    user  system elapsed
#>       0       0       0
#>
#> $manual
#>    user  system elapsed
#>   0.054   0.002   0.058
```

Another use for a list of functions is to summarise an object in multiple ways. To do that, we could store each summary function in a list, and then run them all with `lapply()`:

```
x <- 1:10
funs <- list(
```

Downloaded by [The University of British Columbia Library] at 10:45 13 July 2016

© 2015 by Taylor & Francis Group, LLC

```
  sum = sum,
  mean = mean,
  median = median
)
lapply(funs, function(f) f(x))
#> $sum
#> [1] 55
#>
#> $mean
#> [1] 5.5
#>
#> $median
#> [1] 5.5
```

What if we wanted our summary functions to automatically remove missing values? One approach would be make a list of anonymous functions that call our summary functions with the appropriate arguments:

```
funs2 <- list(
  sum = function(x, ...) sum(x, ..., na.rm = TRUE),
  mean = function(x, ...) mean(x, ..., na.rm = TRUE),
  median = function(x, ...) median(x, ..., na.rm = TRUE)
)
lapply(funs2, function(f) f(x))
#> $sum
#> [1] 55
#>
#> $mean
#> [1] 5.5
#>
#> $median
#> [1] 5.5
```

This, however, leads to a lot of duplication. Apart from a different function name, each function is almost identical. A better approach would be to modify our `lapply()` call to include the extra argument:

```
lapply(funs, function(f) f(x, na.rm = TRUE))
```

10.4.1 Moving lists of functions to the global environment

From time to time you may create a list of functions that you want to be available without having to use a special syntax. For example, imagine

© 2015 by Taylor & Francis Group, LLC

Downloaded by [The University of British Columbia Library] at 10:45 13 July 2016

you want to create HTML code by mapping each tag to an R function. The following example uses a function factory to create functions for the tags `<p>` (paragraph), `<b>` (bold), and `<i>` (italics).

```
simple_tag <- function(tag) {
  force(tag)
  function(...) {
    paste0("<", tag, ">", paste0(...), "</", tag, ">")
  }
}
tags <- c("p", "b", "i")
html <- lapply(setNames(tags, tags), simple_tag)
```

I've put the functions in a list because I don't want them to be available all the time. The risk of a conflict between an existing R function and an HTML tag is high. But keeping them in a list makes code more verbose:

```
html$p("This is ", html$b("bold"), " text.")
#> [1] "<p>This is <b>bold</b> text.</p>"
```

Depending on how long we want the effect to last, you have three options to eliminate the use of `html$`:

- For a very temporary effect, you can use `with()`:

  ```
  with(html, p("This is ", b("bold"), " text."))
  #> [1] "<p>This is <b>bold</b> text.</p>"
  ```

- For a longer effect, you can `attach()` the functions to the search path, then `detach()` when you're done:

  ```
  attach(html)
  p("This is ", b("bold"), " text.")
  #> [1] "<p>This is <b>bold</b> text.</p>"
  detach(html)
  ```

- Finally, you could copy the functions to the global environment with `list2env()`. You can undo this by deleting the functions after you're done.

  ```
  list2env(html, environment())
  #> <environment: R_GlobalEnv>
  p("This is ", b("bold"), " text.")
  #> [1] "<p>This is <b>bold</b> text.</p>"
  rm(list = names(html), envir = environment())
  ```

Downloaded by [The University of British Columbia Library] at 10:45 13 July 2016

© 2015 by Taylor & Francis Group, LLC

I recommend the first option, using `with()`, because it makes it very clear when code is being executed in a special context and what that context is.

10.4.2 Exercises

1. Implement a summary function that works like `base::summary()`, but uses a list of functions. Modify the function so it returns a closure, making it possible to use it as a function factory.
2. Which of the following commands is equivalent to `with(x, f(z))`?
   (a) `x$f(x$z)`.
   (b) `f(x$z)`.
   (c) `x$f(z)`.
   (d) `f(z)`.
   (e) It depends.

10.5 Case study: numerical integration

To conclude this chapter, I'll develop a simple numerical integration tool using first-class functions. Each step in the development of the tool is driven by a desire to reduce duplication and to make the approach more general.

The idea behind numerical integration is simple: find the area under a curve by approximating the curve with simpler components. The two simplest approaches are the **midpoint** and **trapezoid** rules. The midpoint rule approximates a curve with a rectangle. The trapezoid rule uses a trapezoid. Each takes the function we want to integrate, `f`, and a range of values, from `a` to `b`, to integrate over. For this example, I'll try to integrate `sin x` from 0 to π. This is a good choice for testing because it has a simple answer: 2.

```
midpoint <- function(f, a, b) {
  (b - a) * f((a + b) / 2)
}
```

Downloaded by [The University of British Columbia Library] at 10:45 13 July 2016

© 2015 by Taylor & Francis Group, LLC

```
trapezoid <- function(f, a, b) {
  (b - a) / 2 * (f(a) + f(b))
}

midpoint(sin, 0, pi)
#> [1] 3.14
trapezoid(sin, 0, pi)
#> [1] 1.92e-16
```

Neither of these functions gives a very good approximation. To make them more accurate using the idea that underlies calculus: we'll break up the range into smaller pieces and integrate each piece using one of the simple rules. This is called **composite integration**. I'll implement it using two new functions:

```
midpoint_composite <- function(f, a, b, n = 10) {
  points <- seq(a, b, length = n + 1)
  h <- (b - a) / n

  area <- 0
  for (i in seq_len(n)) {
    area <- area + h * f((points[i] + points[i + 1]) / 2)
  }
  area
}

trapezoid_composite <- function(f, a, b, n = 10) {
  points <- seq(a, b, length = n + 1)
  h <- (b - a) / n

  area <- 0
  for (i in seq_len(n)) {
    area <- area + h / 2 * (f(points[i]) + f(points[i + 1]))
  }
  area
}

midpoint_composite(sin, 0, pi, n = 10)
#> [1] 2.01
midpoint_composite(sin, 0, pi, n = 100)
#> [1] 2
trapezoid_composite(sin, 0, pi, n = 10)
#> [1] 1.98
```

© 2015 by Taylor & Francis Group, LLC

Downloaded by [The University of British Columbia Library] at 10:45 13 July 2016

```
trapezoid_composite(sin, 0, pi, n = 100)
#> [1] 2
```

You'll notice that there's a lot of duplication between `midpoint_composite()` and `trapezoid_composite()`. Apart from the internal rule used to integrate over a range, they are basically the same. From these specific functions you can extract a more general composite integration function:

```
composite <- function(f, a, b, n = 10, rule) {
  points <- seq(a, b, length = n + 1)

  area <- 0
  for (i in seq_len(n)) {
    area <- area + rule(f, points[i], points[i + 1])
  }

  area
}

composite(sin, 0, pi, n = 10, rule = midpoint)
#> [1] 2.01
composite(sin, 0, pi, n = 10, rule = trapezoid)
#> [1] 1.98
```

This function takes two functions as arguments: the function to integrate and the integration rule. We can now add even better rules for integrating over smaller ranges:

```
simpson <- function(f, a, b) {
  (b - a) / 6 * (f(a) + 4 * f((a + b) / 2) + f(b))
}

boole <- function(f, a, b) {
  pos <- function(i) a + i * (b - a) / 4
  fi <- function(i) f(pos(i))

  (b - a) / 90 *
    (7 * fi(0) + 32 * fi(1) + 12 * fi(2) + 32 * fi(3) + 7 * fi(4))
}

composite(sin, 0, pi, n = 10, rule = simpson)
#> [1] 2
```

Downloaded by [The University of British Columbia Library] at 10:45 13 July 2016

© 2015 by Taylor & Francis Group, LLC

```
composite(sin, 0, pi, n = 10, rule = boole)
#> [1] 2
```

It turns out that the midpoint, trapezoid, Simpson, and Boole rules are all examples of a more general family called Newton-Cotes rules (http://en.wikipedia.org/wiki/Newton%E2%80%93Cotes_formulas). (They are polynomials of increasing complexity.) We can use this common structure to write a function that can generate any general Newton-Cotes rule:

```
newton_cotes <- function(coef, open = FALSE) {
  n <- length(coef) + open

  function(f, a, b) {
    pos <- function(i) a + i * (b - a) / n
    points <- pos(seq.int(0, length(coef) - 1))

    (b - a) / sum(coef) * sum(f(points) * coef)
  }
}

boole <- newton_cotes(c(7, 32, 12, 32, 7))
milne <- newton_cotes(c(2, -1, 2), open = TRUE)
composite(sin, 0, pi, n = 10, rule = milne)
#> [1] 1.99
```

Mathematically, the next step in improving numerical integration is to move from a grid of evenly spaced points to a grid where the points are closer together near the end of the range, such as Gaussian quadrature. That's beyond the scope of this case study, but you could implement it with similar techniques.

10.5.1 Exercises

1. Instead of creating individual functions (e.g., `midpoint()`, `trapezoid()`, `simpson()`, etc.), we could store them in a list. If we did that, how would that change the code? Can you create the list of functions from a list of coefficients for the Newton-Cotes formulae?
2. The trade-off between integration rules is that more complex rules are slower to compute, but need fewer pieces. For `sin()`

© 2015 by Taylor & Francis Group, LLC

Downloaded by [The University of British Columbia Library] at 10:45 13 July 2016

in the range $[0, \pi]$, determine the number of pieces needed so that each rule will be equally accurate. Illustrate your results with a graph. How do they change for different functions? `sin(1 / x^2)` is particularly challenging.

Downloaded by [The University of British Columbia Library] at 10:45 13 July 2016

© 2015 by Taylor & Francis Group, LLC