Introduction to programming in R: coding best practices

Erik Spence

SciNet HPC Consortium

8 March 2023

Today's slides

To find today's slides, go to the "Introduction to programming in R" page, under Lectures, "Best practices".

`https://scinet.courses/1277`

Today's class

Today we will visit the following topics:

- Coding best practices.
- Modularity.
- Testing.
- Defensive programming.

What are coding best practices?

What are coding best practices?

- These are coding practices which have been discovered, over the course of decades, to produce code which is easy to write, read, debug, test, modify, share and use.
- Best practices are a set of rules which affect how you
  - design the code.
  - implement the code.
- Broadly speaking, coding best practices can be summed up thus:
  - write code which is modular.
  - don't write code which has already been written.
  - write code which is easy to read and understand.

We've discussed a number of these points during previous lectures, but today we will discuss them in detail.

What is modularity?

What do I mean when I say that I'm writing modular code? I'm writing code which:

- is separated into individual functions and procedures. Each of these performs a single, specific task.
- is separated into files, if the code is big enough, which contain related functionality.
- is written so as to enforce boundaries between sections of code.
- includes testing routines, to test the functions against known correct answers.

Why does modularity matter?

Who cares? Why should I write modular code?

- Scientific software can be large, complex and subtle.
- If each section of code uses the internal details of other sections you must understand the entire code at once to understand what the code in a particular section is doing.
- Interactions grow as (number of lines of code)^2.
- This makes finding bugs (mistakes) extremely difficult.
- It also makes writing testing routines for your code extremely difficult.

Bad code

```
pw <- function() {
q <- rep(0, n)
for (i in 1:n) for (j in 1:n) {
q[i] <- q[i] + m[i,j] * a[j]
} a <<- q
}

en <- function() {
q <- rep(0, n)
for (i in 1:n) for (j in 1:n) {
q[i] <- q[i] + m[i,j] * a[j]
} e <- 0
for (i in 1:n) {
e <- e + a[i] * q[i]
} return(e) }
```

It uses functions. Isn't that modular?
What's wrong with this code?

```
run.calc <- function() {
n <- 100
m <- matrix(rep(0, n * n), nrow = n)
a <- rep(0, n)

for (i in 1:n) a[i] <- 1

for (i in 1:n) for (j in 1:n) {
m[i,j] <- something(i,j) }

b <- 0
for (i in 1:n) {
if(m[i,i] > b) b <- m[i,i] }

for (i in 1:n) m[i,i] <- m[i,i] - b
for (i in 1:10) pw()

cat("Ground state energy is ", en(), '\n') }
```

What's wrong with this code?

While it is true that this is valid R code, there are many things that are wrong. Some serious, some less so.

- Global variables are being directly accessed within functions.
- Global variables are modified from within a function (using the '<<-' operator).
- All the code is in one file.
- Code has been copied and pasted.
- Doesn't use existing R functionality.
- No use of vectorization.
- No comments.
- No indentation of code blocks.
- Cryptic variable and function names.

Let's go through these problems, one at a time, and fix the code.

Global variables

We've discussed this before. Don't use global variables inside functions without passing them into the function explicitly through the function's argument list. Just don't.

- By default, in most programming languages, variables declared outside of functions are considered 'global', meaning they can be modified by any function.
- They're very tempting to use, since they make all information available everywhere, but they are terrible coding practise.
- Why? Because they totally destroy modularity. The domain of any function that uses global variables is the entire program. The function can't be tested separately from the rest of the program.
- If you have code which uses global variables, you should modify the code so that the variables are explicitly passed into the function, through the argument list.
- The "<<-" operator allows you to modify global variables from within functions. Don't use it. Ever.

Bad code, version 2

```
pw <- function(a, m) {
n <- length(a)
q <- rep(0, n)
for (i in 1:n) for (j in 1:n) {
q[i] <- q[i] + m[i,j] * a[j]
} return(q) }

en <- function(a, m) {
n <- length(a)
q <- rep(0, n)
for (i in 1:n) for (j in 1:n) {
q[i] <- q[i] + m[i,j] * a[j] }
e <- 0
for (i in 1:n) {e <- e + a[i] * q[i] }
return(e) }
```

The functions no longer use the global variables directly.

```
run.calc <- function(n = 100) {

a <- rep(0, n)
m <- matrix(rep(0, n * n), nrow = n)

for (i in 1:n) a[i] <- 1

for (i in 1:n) for (j in 1:n) {
m[i,j] <- something(i,j) }

b <- 0
for (i in 1:n) {
if(m[i,i] > b) b <- m[i,i] }

for (i in 1:n) m[i,i] <- m[i,i] - b
for (i in 1:10) a <- pw(a, m)

cat("Ground state energy is ", en(a, m), '\n') }
```

A single file

Code should be split up into multiple files which contain functions of similar functionality.

- The general model is to have a 'driver' program which invokes the 'utility' functions (or whatever you want to call them).
- You can write as many driver programs as you need to perform your different analyses.
- The utility functions, being separate from the driver programs, can be tested and debugged separately from any other code.
- Use the 'source' command to tell the driver program about the existence of the utility functions.

Bad code, version 3

```
# Hydrogen_Utilities.R

pw <- function(a, m) {
n <- length(a)
q <- rep(0, n)
for (i in 1:n) for (j in 1:n) {
q[i] <- q[i] + m[i,j] * a[j]
} return(q) }

en <- function(a, m) {
n <- length(a)
q <- rep(0, n)
for (i in 1:n) for (j in 1:n) {
q[i] <- q[i] + m[i,j] * a[j]
} e <- 0
for (i in 1:n) {
e <- e + a[i] * q[i]
} return(e) }
```

```
# Hydrogen.R
source("Hydrogen_Utilities.R")
run.calc <- function(n = 100) {

m <- matrix(rep(0, n * n), nrow = n)
a <- rep(1, n)

for (i in 1:n) for (j in 1:n) {
m[i,j] <- something(i,j) }

b <- 0
for (i in 1:n) {
if(m[i,i] > b) b <- m[i,i] }

for (i in 1:n) m[i,i] <- m[i,i] - b
for (i in 1:10) a <- pw(a, m)

cat("Ground state energy is ", en(a, m), '\n') }
```

Identical blocks of code

As we've discussed before, DO NOT copy-and-paste code.

- If you feel the need to copy-and-paste code, write a function which contains the code block.
- This prevents you from copy-and-pasting code which might be buggy.
- If you discover an error in the code, you only need to fix one block of code.
- It also makes the code easier to read.

Never ever have multiple copies of the same block of code!

Bad code, version 4

```
# Hydrogen_Utilities.R
pw <- function(a, m) {
n <- length(a)
q <- rep(0, n)
for (i in 1:n) for (j in 1:n) {
q[i] <- q[i] + m[i,j] * a[j]
} return(q)
}

en <- function(a, m) {
q <- pw(a, m)
e <- 0
for (i in seq_along(a)) {
e <- e + a[i] * q[i]
} return(e)
}
```

The 'en' function can use 'pw'.

```
# Hydrogen.R
source("Hydrogen_Utilities.R")

run.calc <- function(n = 100) {

m <- matrix(rep(0, n * n), nrow = n)
a <- rep(1, n)

for (i in 1:n) for (j in 1:n) {
m[i,j] <- something(i,j) }

b <- 0
for (i in 1:n) {if(m[i,i] > b) b <- m[i,i] }

for (i in 1:n) m[i,i] <- m[i,i] - b
for (i in 1:10) a <- pw(a, m)

cat("Ground state energy is ", en(a, m), '\n') }
```

Use existing functionality!

There's no need to re-code things when others have already done the work for you.

- If the functionality exists built-into R, you can feel comfortable using it. It works.
- Use vectorization whenever you can!
- If the functionality exists in a user-contributed package, find out if the package is well-regarded and commonly used.
- Talk to your colleagues about the R packages that are used in your field.

Don't bother solving a coding problem which has already been solved.

Bad code, version 5

```
# Hydrogen_Utilities.R

pw <- function(a, m) return(m %*% a)

en <- function(a, m) {
q <- pw(a, m)
return(sum(a * q))
}
```

- The 'pw' function just calculates a matrix multiplication.
- The 'en' function just returns the sum of the product of two vectors.
- The function 'diag' can be used to modify the values of matrix diagonals.

```
# Hydrogen.R
source("Hydrogen_Utilities.R")

run.calc <- function(n = 100) {

m <- matrix(rep(0, n * n), nrow = n)
a <- rep(1, n)

for (i in 1:n) for (j in 1:n) {
m[i,j] <- something(i,j) }

b <- max(diag(m))
diag(m) <- diag(m) - b
for (i in 1:10) a <- pw(a, m)

cat("Ground state energy is ", en(a, m), '\n')

}
```

Write clear code

When writing your code, make things clear. You don't get points for writing code which is hard to understand.

- Indent your code blocks.
- Use variable names which make sense.
- Use function names which make sense.
- Name the files which contain your functions sensibly.
- Name your scripts sensibly.
- Comment your code.

Six months from now you'll thank yourself for writing clear code. So will the next graduate student.

Good code

```
# Hydrogen_Utilities.R
# Functions used to calculate the
# hydrogen atomic energy states.

ham.applied <- function(a, ham) {
  # Calculates the Hamiltonian
  # applied to the state vector.
  # a - the state vector.
  # ham - Hamiltonian operator.
  # Returns a new state vector.

  # Return the matrix product of the
  # Hamiltonian and state vector.
  return(ham %*% a)
}
```

```
# Hydrogen_Utilities.R, continued

ground.energy <- function(a, ham) {
  # Calculates the hydrogen ground state energy.
  # a - the state vector.
  # ham - Hamiltonian operator.
  # Returns the ground state energy.

  # Apply the Hamiltonian.
  q <- ham.applied(a, ham)

  # Calculate the total energy, and return.
  return(sum(a * q))
}
```

Good code, continued

```r
# Hydrogen.R
# This function calculates the ground
# state atomic energy of hydrogen.

calc.ground.energy <- function(n = 100) {
  # Calculates and prints out the ground
  state energy of hydrogen.
  # n - the size of the problem.

  # File containing the ham.applied
  # and ground.energy functions.
  source("Hydrogen_Utilities.R")

  # Hamiltonian operator, and
  # initial state vector.
  ham <- matrix(rep(0, n * n), nrow = n)
  a <- rep(1, n)
```

```r
# Hydrogen.R, continued

  # Initialize the Hamiltonian.
  for (i in 1:n) for (j in 1:n) {
    ham[i,j] <- something(i,j) }

  # Get the value of the largest diagonal
  # element and remove it from all diagonals.
  max.diag <- max(diag(ham))
  diag(ham) <- diag(ham) - max.diag

  # Call ham.applied 10 times.
  for (i in 1:10) a <- ham.applied(a, ham)

  # Print out the answer.
  cat("Ground state energy is ",
    ground.energy(a, ham), '\n')
}
```

Testing

Ok, so you've written your awesome code, and it's been separated by functionality into various modules. Now you need to write testing routines.

Seriously?

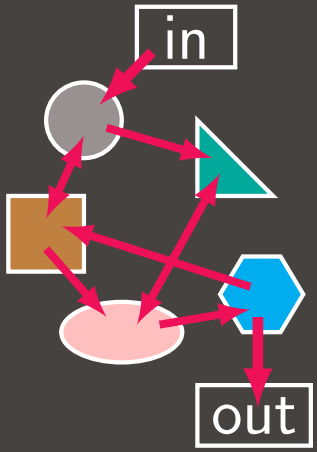
The purpose of these testing routines is to test the code against known situations, now and especially in the future. There are two broad categories of testing:

- Integrated testing: testing the whole gigantic program, integrating the outputs of your many modularized pieces.
- Unit testing: testing the pieces of your modules individually.

Integrated testing

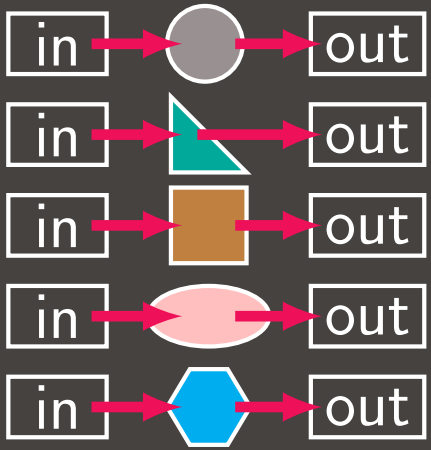
Integrated testing is important, but shouldn't be the only type of testing done.

- As you develop a large program, with many interacting parts, bugs will develop.
- It will be difficult to determine the source of the problem, if testing is only performed on the integrated whole.



Unit testing

- Test the pieces of the code individually.
- This means writing code that will test the pieces of code in question.
- Test against easy solutions, typical solutions, edge cases, special cases.
- This enormously speeds up the detection of bugs.
- If you are given a code which does not come with testing routines, just assume that it's wrong.



This won't work if you don't write modular code.

Testing, types of tests

So we're writing our testing code. What kind of tests should we perform?

- Comparison to analytic solutions:
  - Solutions tend to be for simple situations - not hard tests of the computation.
  - If your code doesn't get these right you've got serious problems.
- Benchmarking: comparing the results of your code to other codes which solve the same problem, in the same parameter regime.
  - Does not demonstrate that either solution is correct.
  - Can show that at last one code or version has a problem, or that something has caused changes.
  - Is more powerful if different algorithm types are used.
  - Save the results of benchmarks in your testing directory.
- Testing against reality:
  - If your code calculates something that can be compared against reality, do it as one of your tests.
  - Assume that reality is correct.

Testing the good code

```r
# Hydrogen_Utilities_Tests.R
# Testing routines for the functions
# in Hydrogen_Utilities.R.
source("Hydrogen_Utilities.R")

# Tests for ham.applied.

my.test <- function(a, ham, cond, test.name) {
  n <- length(a)
  if(sum(ham.applied(a, ham) == cond) == n) {
    cat("Passed", test.name, "test.\n")
  } else {
    stop("Failed", test.name, "test.")
  }
}
```

```r
# Hydrogen_Utilities_Tests.R, continued

run.hydrogen.utils.tests <- function(n = 100) {
  # Identity test.
  # Ham is an identity matrix.
  ham <- diag(n)
  a <- rep(1, n)
  my.test(a, ham, a, "identity")

  # Linear test.
  for (i in 1:n) ham[i,] <- i
  my.test(a, ham, 1:n * n, "linear")

  # Back-diagonal test.
  ham[,] <- 0
  for (i in 1:n) ham[n - i + 1, i] <- i
  my.test(a, ham, n:1, "back-diagonal")
}
```

Testing, continued

```
>
> source("Hydrogen.Utilities.Tests.R")
>
> run.hydrogen.utils.tests()
[1] "Passed identity test."
[1] "Passed linear test."
[1] "Passed back-diagonal test."
>
```

Every so often, especially when editing your routines, re-run your test suite.

There also exist more-advanced testing frameworks to build into your R libraries. We won't cover those here.

Defensive programming

What is 'defensive programming'? Programming to protect the code from the user (usually yourself). This means checking that function arguments, or script command-line arguments, meet certain criteria.

How this is accomplished depends upon the context, and the programming language, but there are some common situations.

- Check to make sure that numbers are not negative (when they shouldn't be).
- Check to make sure that arguments are of the correct type.
- Use 'tryCatch()'.

Train yourself to put defensive coding blocks at the start of your functions.

Defending the good code

```r
# Hydrogen_Utilities.R
# Functions used to calculate the
# hydrogen atomic energy states.
ham.applied <- function(a, ham) {

  # The second dimension of ham must
  # be the length of a.
  if(dim(ham)[2] != length(a)) {
    stop("Problem in ham.applied: bad ham
      dimensions.")
  }

  # Return the matrix product of the
  # Hamiltonian and state vector.
  return(ham %*% a)
}
```

```r
# Hydrogen_Utilities.R, continued
ground.energy <- function(a, ham) {

  # The first dimension of ham must
  # be the length of a.
  if(dim(ham)[1] != length(a)) {
    stop("Problem in ground.energy: bad ham
      dimensions.")
    }

  # Apply the Hamiltonian.
  q <- ham.applied(a, ham)

  # Calculate the total energy, and return.
  return(sum(a * q))
}
```

Documentation of your code

You must document your code. Must. Not optional. Documentation of code comes in many forms:

- sensible variable, function, class, and module names.
- comments in the code.
- help commands, docstrings, or other built-in feedback.

Six months from now you're not going to remember the motivation for writing that function. So write it down in the code somewhere.

Enough to get started

This is a good summary of coding best practices.

Starting immediately, we expect you to use best practices (comments, function documentation, indentation, defensive programming, etc.) with all of your homework assignments. You will be docked marks if you do not.

We will not be expecting testing routines with your homework assignments, unless explicitly requested.