Programming in R for Hydrologists

R.D. Moore

2018-Sept-25

Contents

1 Introduction **2**

2 Grouped expressions **2**

3 Conditional execution **2**

4 Digression on logical operators **3**

5 Functions **4**
5.1 Example . . . 5
5.2 Components of a function . . . 5
5.3 Specifying arguments in a function call . . . 6
5.4 Default values for arguments . . . 7
5.5 Use of `return()` . . . 7
5.6 Nested functions . . . 9
5.7 Scope of objects within a function . . . 9
5.8 Passing functions as arguments to another function . . . 10

6 Looping **10**
6.1 Why loop? . . . 10
6.2 Types of loop . . . 11
6.3 `for` loops . . . 11
6.4 `while` and `repeat` loops . . . 13

7 Application of loops to iterative solutions **13**

8 Subset processing **15**
8.1 `for` loops . . . 15
8.2 `for` loops for graphing by data subsets . . . 17
8.3 The `aggregate()` function in base R . . . 19

9 Writing efficient code **23**

10 Debugging and error handling **25**
10.1 General considerations . . . 25
10.2 The `try()` function . . . 25

11 Algorithms **26**
11.1 Prose description . . . 26
11.2 Flow chart . . . 26
11.3 Pseudo-code . . . 27

1 Introduction

In this installment, we look at the fundamental building blocks for programming in R. We will only scratch the surface, but the material that is covered will take you a long way in your journey into data processing and analysis. I would highly recommend Hadley Wickham's *Advanced R* (http://adv-r.had.co.nz/) for those who want to pursue the topic more deeply.

As with the installment on *R Basics*, we will focus on functions within base R. In a later installment, we will move on to functions in contributed packages, particularly those in the `tidyverse` suite of packages.

2 Grouped expressions

Grouped expressions are sets of commands that are to be executed as a group; the result of a grouped expression is the result of the last expression in the group that was evaluated.

In R, grouped expressions are enclosed in braces and separated by semi-colons:

```
{command 1; command 2; command 3; ...; command n}
```

Alternatively, the commands can be placed on separate lines if it improves readability (in which case semi-colons are not required):

```
{
  command 1
  command 2
    ...
    command n
}
```

Here are two variations on an example.

```
{x = c(1, 2, 3, 4, NA); y = rev(x); x > y}
```

```
## [1]    NA FALSE FALSE  TRUE    NA
```

```
z = {x = c(1, 2, 3, 4, NA); y = rev(x); x > y}
z
```

```
## [1]    NA FALSE FALSE  TRUE    NA
```

3 Conditional execution

Conditional execution of program statements can be controlled by the `if ... else ...` construct, which has the following form.

```
if (expr_1) expr_2 else expr_3
```

where `expr_1` is an expression that evaluates to `TRUE` or `FALSE` and `expr_1` and `expr_2` are expressions that are executed depending on the value of `expr_1`. An example follows.

```
y = 2
if (y == 1) x = 2 else x = 1
```

What would be the result of the code shown above? What would the result for `y == 1`?

Unlike `ifelse()`, the `if ... else ...` construct does not allow the test condition, `expr_1`, to be a vector. If you try to use a vector expression for `expr_1`, R will use the first value and issue a warning. Compare the results of running the two sets of code below.

```
# if ... else with a vector test condition
y = seq(1, 4, 1)
if (y == 1) x = 2 else x = 1
x

# ifelse() equivalent
x = ifelse(y == 1, 2, 1)
x
```

The `if ... else ...` construct can be especially useful for situations in which different sets of operations should be executed, depending on one or more conditions. For example, suppose you wanted to want to generate a histogram only for data sets with at least 20 observations and a box plot for data sets with less than 20 observations. Try re-running the code chunk below, but after changing `n` to 10.

```
n = 30
x = runif(n, 0, 100)
if (n < 20) boxplot(x) else hist(x)
```

An `if ...` expression can be written without a corresponding `else ...` expression. In the following example, a histogram would be plotted if n were 20 or higher. For n $<$ 20, no action would be taken.

```
if (n >= 20) hist(x)
```

`if ... else ...` statements can be written with grouped expressions. Note the placement of the `{}` in the following, as recommended in the Google R style guide to aid readability.

```
if(expr_1) {
  # statements to execute if expr_1 is TRUE
} else {
  # statements to execute if expr_1 is FALSE
}
```

They can also accommodate more than two conditions, as in the following example.

```
if(expr_1) {
  # statements to execute if expr_1 TRUE
} else if(expr_2) {
  # statements to execute if expr_1 is FALSE and expr_2 is TRUE
} else {
  # statements to execute if both expr_1 and expr_2 are FALSE
}
```

After you gain more experience, you should investigate `switch()` and `cut()`, which can be useful in the case of multiple conditions.

4 Digression on logical operators

Two logical conditions, A and B, can be combined by two operations, AND and OR. The following code chunk and the resulting table summarize the possible outcomes.

```
A = c(TRUE, TRUE, FALSE, FALSE)
B = c(TRUE, FALSE, TRUE, FALSE)
AandB = A & B
AorB = A | B
AB = data.frame(A, B, AandB, AorB)
colnames(AB) = c("A", "B", "A AND B", "A OR B")
knitr::kable(AB)
```

A	B	A AND B	A OR B
TRUE	TRUE	TRUE	TRUE
TRUE	FALSE	FALSE	TRUE
FALSE	TRUE	FALSE	TRUE
FALSE	FALSE	FALSE	FALSE

In R, both `|` and `||` represent logical OR and `&` and `&&` represent logical AND. The difference is that the shorter forms (i.e., `|` and `&`) are vectorized operations, which operate elementwise on vectors, as in the code chunk above. For example, consider the following statements.

```
x = seq(1, 10, 1)
x > 2
```

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

```
x < 7
```

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

```
x > 2 & x < 7
```

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

```
x > 2 | x < 7
```

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

The longer forms (`||` and `&&`) operate from left to right and only evaluate the first element of a vector. Compare the following results with those from the shorter-form operators.

```
x > 2 && x < 7
```

```
## [1] FALSE
```

```
x > 2 || x < 7
```

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

An important feature of the longer form is that they "short-cut" the boolean expression. That is, they evaluate a sequence of logical relations only until an answer is reached. For example, suppose A, B and C are three logical conditions, and we wish to evaluate `A || B || C`. The entire statement is TRUE even if only one of the three conditions is TRUE. Thus, if A is TRUE, there is no need to evaluate B or C, which can reduce the run time for a program that requires the evaluation of many logical conditions.

Similarly, for `A && B && C`, the entire statement is FALSE if even one of the conditions is FALSE Thus, if A is FALSE, there is no need to evaluate further.

The upshot is that the shorter forms are appropriate for use in vectorized operations such as `ifelse()` or `x[x > 2 | x < 5]`, and the longer forms are appropriate in `if ... else ...` constructs.

5 Functions

In the previous installment on R basics, we saw the definition of a function having only one statement. We now look at the construction of functions that involve grouped expressions and multiple alternative conditions. The material presented here is sufficient to get you started. For a more detailed coverage, see http://adv-r.had.co.nz/Functions.html.

5.1 Example

As an example, consider the following code, which creates a lagged or advanced version of a vector. Note that there is a `lag()` function for `ts` objects, and also a `lag()` function in the `dplyr` package, but not one for vectors in base R.

```
xlag = function(x, k) {
  # x is a vector to be lagged/advanced
  # k is the lag (positive for lag, negative for advance
  # the function returns the lagged/advanced time series
  n = length(x)
  if (k > 0) { # lag time series
    c(rep(NA, k), x[1:(n - k)])
  } else if (k < 0) { # advance time series
    c(x[abs(k-1):n], rep(NA, abs(k)))
  } else { # k = 0, return original series
    x
  }
}
```

The `xlag()` function accepts two arguments, `x` and `k`, and returns a lagged version of `x`.

Here's an exercise: Is the last condition of the *if-else* structure necessary? How might you re-write the *if-else* structure to avoid including the third condition?

To use a function, the code defining the function must first be executed (e.g., entered at the command prompt or run within a script). The function can then be called as needed.

```
y <- seq(1, 10, 1)
xlag(y, 4)
```

```
##  [1] NA NA NA NA  1  2  3  4  5  6
```

```
xlag(y, -4)
```

```
##  [1]  5  6  7  8  9 10 NA NA NA NA
```

```
ylag4 <- xlag(y, 4)
yadv4 <- xlag(y, -4)
ylag4
```

```
##  [1] NA NA NA NA  1  2  3  4  5  6
```

```
yadv4
```

```
##  [1]  5  6  7  8  9 10 NA NA NA NA
```

Although functions can accept multiple arguments, they can return only a single object, which is defined by the last quantity evaluated during the function's execution. For example, in the case of `xlag()`, the last statement to be evaluated for the case of k = 0 is `c(rep(NA, k), x[1:(n - k)])`, so that is the value that is returned.

If you need to return multiple objects, such as a set of vectors, you can combine them in a list and return that.

5.2 Components of a function

Functions have three components: (a) the body, which is the code inside the function, (b) the formals, which are the arguments used within the function, and (c) the environment, is the environment in which the function

was created and in which it stores local variables.

When you "print" a function (e.g., type the function name at the command prompt and hit "enter"), these components are shown. If the environment is not displayed, the function was created in the global environment.

```
xlag
```

```
## function(x, k) {
##   # x is a vector to be lagged/advanced
##   # k is the lag (positive for lag, negative for advance
##   # the function returns the lagged/advanced time series
##   n = length(x)
##   if (k > 0) { # lag time series
##     c(rep(NA, k), x[1:(n - k)])
##   } else if (k < 0) { # advance time series
##     c(x[abs(k-1):n], rep(NA, abs(k)))
##   } else { # k = 0, return original series
##     x
##   }
## }
## <bytecode: 0x00000000123b5268>
```

We can examine these components using the following commands.

```
body(xlag)
```

```
## {
##     n = length(x)
##     if (k > 0) {
##         c(rep(NA, k), x[1:(n - k)])
##     }
##     else if (k < 0) {
##         c(x[abs(k - 1):n], rep(NA, abs(k)))
##     }
##     else {
##         x
##     }
## }
```

```
formals(xlag)
```

```
## $x
##
##
## $k
```

```
environment(xlag)
```

```
## <environment: R_GlobalEnv>
```

Some built-in functions, like `apply()` and `uniroot()`, are coded in R, whereas others are coded in whole or in part in C for higher performance. Therefore, if you print or use `body()` for a built-in function, you may or may not see code that you recognize as having been written in R.

5.3 Specifying arguments in a function call

When calling a function, the arguments can be specified in one of three ways:

- by order
- by name
- by partial name

For example, the `seq()` function has the following arguments: `from`, `to`, `by`, `length.out` and `along.with`. In our usage of `seq()` to date, we have assigned the arguments by matching by order. For example, in `seq(1, 10, 1)`, the values of 1, 10 and 1 are assigned to `from`, `to` and `by`, respectively. The other arguments are assigned their default values (see next topic).

Suppose you wanted to generate a sequence of 31 evenly spaced values from 100 to 430. This situation could arise, for example, if you wanted to generate evenly spaced sampling locations along a section of river from 100 to 430 m upstream of some specified point. In this case, we would use the `length.out` argument, not `by`.

```
x_sample = seq(100, 430, length.out = 31)
```

In this example, we assigned the first two arguments by order and the third by name. It is good practice to use assignment by order only for the first one or two arguments, which are usually the most commonly used, and to assign by name for the rest.

As indicated above, you only need to provide part of the name of an argument in the call to the function. When interpreting a function call, R uses the `pmatch()` function, which does partial matching of character strings. This function takes a character string that may be only the first few characters of a name and then matches it against all of the possible names. Thus, we could use the following statement.

```
x_sample = seq(100, 430, len = 31)
```

You need to provide enough characters to provide a unique matching, or else the `pmatch()` function will return `NA`. For example, run the following code, in which "me" is the string to be matched, and c("mean", "median") are the values to be matched against.

```
pmatch("me", c("median", "mean"))
```

One potential pitfall of relying on partial matching is that a function may be modified. If new arguments were added, their names could possibly conflict with your partial matching. It is safest to use full argument names in your scripts, and restrict the use of abbreviated argument names to code that you type interactively at the command prompt.

5.4 Default values for arguments

Many functions will have default values for at least some of their arguments. You can find these by looking at the "usage" description in the built-in help.

```
?seq
```

In the case of `xlag()`, the most commonly used lag is 1. Thus, it would be reasonable to use that as a default value.

```
xlag = function(x, k = 1) {
  # code left out
}
```

In this case, the statement `xlag(y)` would return a version of y that had been lagged by one step.

5.5 Use of `return()`

The function `xlag()` uses the principle of returning the last evaluated statement within a grouped expression. Many programmers like to use a `return()` statement to identify explicitly the value to be returned, as in the following example.

```
xlag2 = function(x, k) {
  # x is a vector to be lagged/advanced
  # k is the lag (positive for lag, negative for advance
  # the function returns the lagged/advanced time series
  n = length(x)
  if (k > 0) {
    xlk <- c(rep(NA, k), x[1:(n - k)])
  } else if (k < 0) {
    xlk <- c(x[abs(k-1):n], rep(NA, abs(k)))
  } else {
    xlk <- x
  }
  return(xlk)
}
```

A third option is to use an "early return" approach.

```
xlag3 = function(x, k) {
  # x is a vector to be lagged/advanced
  # k is the lag (positive for lag, negative for advance
  # the function returns the lagged/advanced time series
  n = length(x)
  if (k > 0) {
    return(c(rep(NA, k), x[1:(n - k)]))
  } else if (k < 0) {
    return(c(x[abs(k-1):n], rep(NA, abs(k))))
  } else {
    return(x)
  }
}
```

The use of `return()` in functions is a controversial matter, especially in R. See, for example, the following exchange on the topic:

- https://stackoverflow.com/questions/11738823/explicitly-calling-return-in-a-function-or-not

Many R programmers, including Hadley Wickham (http://adv-r.had.co.nz/Functions.html#return-values), recommend that explicit `return()` statements be avoided in most cases, but note that they can be useful if there are conditions in which the function execution should terminate prior to running the rest of the code.

For example, suppose you want a function to return `NA` if either of two arguments, `a` and `b`, is `NA`, and that the rest of the code involves multiple blocks of code with different levels of indentation. This could be accomplished by the two approaches illustrated below.

In the example of `f2()`, the function terminates execution as soon as it returns an `NA` value, and does not evaluate the rest of the code. This use of an explicit `return()` can keep the code simpler, with one less level of indentation than is the case for `f1()`.

```
f1 = function(a, b) {
  if (is.na(a) || is.na(b)) {
    NA
  } else {
  # multiple lines of code follow ...
  }
}

f2 = function(a, b) {
  if (is.na(a) || is.na(b)) return(NA)
```

```
  # multiple lines of code follow ...
}
```

5.6 Nested functions

The following example illustrates a function nested within a function. The function `f1()` computes the difference in the means of the positive values of two vectors, x and y. The function `g()` converts all zero and negative values in a vector to `NA`. Depending on the end goal, there are simpler ways to code this function. The code below is intended simply to illustrate the concept.

```
f1 = function(x, y) {
  g = function(x) ifelse(x <= 0, NA, x)
  mean(g(x), na.rm = TRUE) - mean(g(y), na.rm = TRUE)
}
```

In the preceding example, the function `g()` is not available outside the function `f1()`. If you want to use it in other places in your code, you should define it outside the function `f1()`.

```
g = function(x) ifelse(x <= 0, NA, x)
f2 = function(x, y) mean(g(x), na.rm = TRUE) - mean(g(y), na.rm = TRUE)
```

Now we will apply the two versions.

```
x = runif(20, -10, 10)
y = runif(20, -10, 10)
x
```

```
##  [1] -2.1217622  0.6710431  3.2390249  0.8530057 -7.9460693 -7.5889510
##  [7] -2.3357038  1.3874604  4.4121736  7.4676589 -6.3693285  4.1585967
## [13] -5.3807399 -8.7934699  0.2734849 -5.0768633 -1.0661765  0.9419391
## [19]  9.4972342  7.0230698
```

```
y
```

```
##  [1] -6.4682755  8.3498402  4.2279678 -7.9304565 -0.5874951 -7.9715838
##  [7]  5.9701938  9.4292949  9.7564058  6.6604442 -5.6130410 -4.9141107
## [13]  9.5062076  8.6691099  5.3302130  5.5274325  5.0251077 -0.7590692
## [19]  1.2557814 -2.1733248
```

```
f1(x, y)
```

```
## [1] -3.012816
```

```
f2(x, y)
```

```
## [1] -3.012816
```

5.7 Scope of objects within a function

"Scope" refers to the rules by which R looks for variables or functions that are referred to in code.

Many functions involve the calculation of intermediate quantities prior to computing the value to be returned (e.g., `n` in the function `xlag()`). These variables are local and temporary, and cease to exist after the function returns its value, and thus cannot be used outside the function.

If a variable is referred to in the function but not defined either by an argument passed to the function or by an assignment within the function, R will look a level higher. If a function is nested within another function,

R will first look in the higher level function, then in the global environment. For a non-nested function, R will look in the global environment.

5.8 Passing functions as arguments to another function

The ability to pass functions as arguments is a powerful aspect of the R programming language, which we will see incorporated into many of the functions in common use. A simple example is provided below, based on our earlier example.

```
g = function(x) ifelse(x <= 0, NA, x)

f3 = function(x, y, user.fun) {
  mean(user.fun(x), na.rm = TRUE) - mean(user.fun(y), na.rm = TRUE)
}

f3(x, y, user.fun = g)
```

```
## [1] -3.012816
```

In this example, the user can easily use different functions to be applied within the function `f3()` without recoding it.

6 Looping

"Looping" refers to the repeated application of a set of procedures until some condition is met. Writing loops is the key to working efficiently with multiple data sets or data sets with multiple subdivisions.

6.1 Why loop?

Many situations arise in which the same operation or set of operations must be applied multiple times. In hydrology and related areas, these situations commonly fall into four categories:

- performing similar operations on multiple subsets of data
- performing similar operations on multiple data sets
- solving an equation by iteration
- tracking the evolution of a system through time

This week's session will focus on the first, which are often called "split-apply-combine" operations because they involve *splitting* a data set into subsets, *applying* some functions to each subset, then *combining* the results into a summary. A simple example is computing the mean of daily air temperature data by year and month. We will also touch on solving equations by iteration. The second and fourth situations will be covered in later sessions.

Although loops provide a flexible and customizable approach to sub-group processing, they can be slow when coded in R and require many lines of code for "book-keeping" tasks like identifying the levels of variables. It is generally recommended that you avoid loops by using one of the "split-apply-combine" functions available in base R or a package like `dplyr`. These functions handle the book-keeping and execute the loops using efficient code. However, the following article raises some interesting arguments in favour of using loops in some code:

- Ligges, U. and Fox, J. 2008. How can I avoid this loop or make it faster? *R News* 8/1: 46-50. https://www.r-project.org/doc/Rnews/Rnews_2008-1.pdf

I highly recommend reading Ligges and Fox (2008) for their advice on how to improve the performance of code that includes loops.

6.2 Types of loop

There are three types of loop: `for`, `repeat` and `while`. In `for` loops, the number of times the code is executed can be specified *a priori*, whereas `repeat` and `while` loops execute the code until a specified condition occurs, at which point control exits the loop and the code following the loop is executed.

6.3 `for` loops

6.3.1 Syntax

The examples provided here are simply for illustration of the structure and syntax of `for` loops. It would be preferable to use a vectorized approach rather than a `for` loop for these cases.

The following example illustrates a typical format for a `for` loop. Note that the comments are for the benefit of novice R users. For more experienced users, the code is sufficiently self-evident that the comments would not be required.

```
# create a vector of 5 uniform random numbers between 0 and 100
x = runif(5, 0, 100)
x
```

```
## [1] 64.59679 61.73267 58.70421 53.58723 72.06596
```

```
n = length(x)
# print each value, including the value of "i"
for (i in 1:n) {
  print(c(i, x[i]))
}
```

```
## [1]  1.00000 64.59679
## [1]  2.00000 61.73267
## [1]  3.00000 58.70421
## [1]  4.00000 53.58723
## [1]  5.00000 72.06596
```

When the loop is executed, the code is executed `n` times, each time setting `i` to a value in the sequence beginning with `i = 1` and ending with `i = n`. The variable `i` is called an index, and **its value should not be modified within the loop**. Although I have used `i` for the index in this example, any valid variable name could be used.

If there is only a single line of code to be executed, the loop can be typed as follows:

```
# note additional argument in "print"
for (i in 1:n) print(c(i, x[i]), digits = 3)
```

```
## [1]  1.0 64.6
## [1]  2.0 61.7
## [1]  3.0 58.7
## [1]  4.0 53.6
## [1]  5.0 72.1
```

The range of the index can be expressed not only as a sequence, but also as a vector. In the latter case, the index is sequentially assigned the values within the vector, as in the following example.

```
i.range = 1:n
for (i in i.range) print(c(i, x[i]), digits = 3)
```

```
## [1]  1.0 64.6
## [1]  2.0 61.7
## [1]  3.0 58.7
## [1]  4.0 53.6
## [1]  5.0 72.1
```

The index range does not have to be a regular sequence, as shown in the next example.

```
i.range = c(1, 2, 5)
for (i in i.range) print(c(i, x[i]), digits = 3)
```

```
## [1]  1.0 64.6
## [1]  2.0 61.7
## [1]  5.0 72.1
```

A useful construction uses the `seq_along()` function. Using its default arguments, `seq_along(x)` would generate a sequence of digits from 1 to length(x).

```
for (i in seq_along(x)) print(c(i, x[i]), digits = 3)
```

```
## [1]  1.0 64.6
## [1]  2.0 61.7
## [1]  3.0 58.7
## [1]  4.0 53.6
## [1]  5.0 72.1
```

One can "nest" `for` loops. Inner loops are executed within each iteration of an outer loop. For example, consider the following code.

```
x = 1:3
y =  4:6
for (i in x) {
  for (j in y) {
    print(paste("x =", as.character(i), "   y =", as.character(j)))
  }
}
```

```
## [1] "x = 1    y = 4"
## [1] "x = 1    y = 5"
## [1] "x = 1    y = 6"
## [1] "x = 2    y = 4"
## [1] "x = 2    y = 5"
## [1] "x = 2    y = 6"
## [1] "x = 3    y = 4"
## [1] "x = 3    y = 5"
## [1] "x = 3    y = 6"
```

6.3.2 Interrupting the loop: `break` and `next`

In some cases, it may be appropriate not to complete all steps within a current iteration of a loop and to jump to the next iteration. In these cases, the `next` statement is useful. For example, suppose you are conducting an analysis on subsets in which only subsets with a specific minimum sample size (n.min) will be analysed. In the example below, suppose that a data frame `df` contains a factor variable `site`, with multiple observations of some variable `x` at each site.

```
sites = levels(df$site)
ns = length(sites)
for (i in 1:ns) {
  ss = subset(df, site == sites[i])
  n.ss = nrow(ss)
  if (n.ss < n.min) next
  # more code to analyse data
}
```

Occasionally, there may be a need to break out of a loop before all the pre-specified iterations have been completed. For this purpose, use the `break` function.

```
for (i in 1:n) {
  statement_1
  statement_2
  # more statements ...
  if (break_condition) break
  # more statements ...
}
```

When used in nested loops, `break` and `next` only affect the current level of loop, and those nested within it. They do not affect the operation of outer loops.

6.4 `while` and `repeat` loops

`while` loops test for a condition at the beginning of loop. The loop is entered if the condition is initially TRUE; if the condition is initially FALSE, the loop is not entered. After statements within the loop are executed, the test condition is checked again. The loop ends once the condition is FALSE, and the program flow moves to the first statement following the end of the loop. The syntax of a `while` loop is as follows.

```
while (condition) {
  statement_1
  statement_2
  ...
}
```

In the case of a `repeat` loop, the loop is initially entered and all statements in the loop executed without checking an exit condition. If the exit condition is FALSE, all statements are executed again, followed by a check of the exit condition. The loop repeats until the exit condition is TRUE The syntax is as follows.

```
repeat {
  statement_1
  statement_2
  ...
  if (exit_condition) break
}
```

The main difference is that a `repeat` loop is always executed at least once, regardless of whether or not the condition is initially TRUE, whereas the statements in a `while` loop will not be executed even once if the condition is initially FALSE.

7 Application of loops to iterative solutions

Many analyses involve iterative solutions in which, through a series of steps, the calculated solution gets closer and closer to the true solution. A classic example is finding the root of an equation, which can be

framed as finding the value of x such that $f(x) = 0$. For a simple example, consider finding the square root of a given value S. This can be framed as finding the value of x such that $f(x) = x^2 - S = 0$.

A range of approaches have been developed for root finding. One that is typically taught in calculus classes and works well for many functions is *Newton's method*. This method involves finding an initial "guess" at the solution, x_0, and then iteratively refining it as follows:

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)}$$

where x_n is the current estimate, x_{n+1} is the updated estimate, and $f'(x_n)$ indicates the first derivative of $f(x)$, evaluated at $x = x_n$.

For the case of a square root, Newton's method becomes

$$x_{n+1} = \frac{1}{2}(x_n + \frac{S}{x_n})$$

An important decision is when to end the iteration. For example, one might decide that the answer is "close enough" if the difference between x_n and x_{n+1} is less than, say, $1 \cdot 10^{-6}$. One might also decide to impose a maximum number of iterations to avoid wasting time on problems that are not converging quickly to a solution.

The following code combines the programming techniques we have covered so far to implement Newton's method for finding a square root. It uses $0.5 \cdot x$ as a starting value, based on the observation that, for a positive number, the square root must lie between 0 and S.

```
my.sqrt = function(S, delta = 1e-6, i.max = 100) {
  # delta is the tolerance for stopping iterations
  # i.max is the maximum number of iterations
  x = 0.5*S
  count = 0
  repeat {
    count = count + 1
    x_new = 0.5*(x + S/x)
    dx = x_new - x
    x = x_new
    if (abs(dx) < delta || count >= i.max) break
  }
  if (abs(dx) < delta) {
    msg = "convergence reached"
  } else {
    msg = "convergence failed"
  }
  list(solution = x, dx = dx, iter = count, msg = msg)
}

# find square root of 20
x = my.sqrt(20)
str(x)
```

```
## List of 4
##  $ solution: num 4.47
##  $ dx      : num -3.77e-13
##  $ iter    : num 6
##  $ msg     : chr "convergence reached"
```

```
x$solution
```

```
## [1] 4.472136
```

The example above is provided merely as an illustration. Numerical analysis, including finding the roots of equations, is a complicated topic, and it is easy to go horribly wrong if you try to do it yourself without specialized knowledge. Functions like `uniroot()` in base R provide flexible and robust tools for root finding.

8 Subset processing

8.1 `for` loops

Although there are alternative, and usually better, ways to process subsets in R, we will introduce the use of `for` loops here, which can have advantages in some applications.

We will use the Mauna Loa CO_2 data set, and will use `for` loops to calculate mean CO_2 concentrations by decade and month. First, we set up the data vectors by extracting information from the `co2` time series object and then creating a character variable, `decade`. Note that there are a range of approaches one could use to generate the `decade` variable. An alternative approach will be used later for illustration.

```
CO2 = as.numeric(co2)
month = rep(1:12, times = 39)
year = rep(1959:1997, each = 12)
decades = c("1950s", "1960s", "1970s", "1980s", "1990s")
dec.code = floor(year/10) - 194
decade = decades[dec.code]
summary(decade)
```

```
##    Length     Class      Mode
##       468 character character
```

Now we find the "levels" of the variables using the `unique()` function.

```
month.vals = unique(month)
decade.vals = unique(decade)

month.vals
```

```
##  [1]  1  2  3  4  5  6  7  8  9 10 11 12
```

```
decade.vals
```

```
## [1] "1950s" "1960s" "1970s" "1980s" "1990s"
```

```
n.m = length(month.vals)
n.d = length(decade.vals)
```

Now we loop over the values of `month.vals`, compute the mean and number of data values, and store them in a data frame. We then specify column names (variable names).

```
# find means by month; first create a data frame to hold results
co2m.df = data.frame(ncol = 3, nrow = n.m)
for (i in 1:n.m) {
  co2m.df[i, 1] = month.abb[i]
  co2m.df[i, 2] = mean(CO2[month == i])
  co2m.df[i, 3] = length(CO2[month == i])
}
```

```
# specify variable names in data frame
colnames(co2m.df) = c("month", "CO2mean", "n")
co2m.df
```

```
##    month  CO2mean  n
## 1    Jan 336.4308 39
## 2    Feb 337.2033 39
## 3    Mar 338.0546 39
## 4    Apr 339.2944 39
## 5    May 339.8821 39
## 6    Jun 339.3282 39
## 7    Jul 337.9164 39
## 8    Aug 335.9579 39
## 9    Sep 334.2428 39
## 10   Oct 334.1692 39
## 11   Nov 335.4679 39
## 12   Dec 336.6946 39
```

Now we loop over the decades to extract decadal means.

```
# find means by decade; first create a data frame to hold results
co2d.df = data.frame(ncol = 3, nrow = n.d)
for (i in 1:n.d) {
  co2d.df[i, 1] = decade.vals[i]
  co2d.df[i, 2] = mean(CO2[decade == decades[i]])
  co2d.df[i, 3] = length(CO2[decade == decades[i]])
}

# specify variable names in data frame
colnames(co2d.df) = c("decade", "CO2mean", "n")
co2d.df
```

```
##   decade  CO2mean   n
## 1  1950s 315.8258  12
## 2  1960s 320.1277 120
## 3  1970s 330.7275 120
## 4  1980s 345.1609 120
## 5  1990s 358.6376  96
```

Now we loop over both decade and month using nested `for` loops. This version generates output in a 'wide' format.

```
# find means by both decade and month; store results in a "wide" data frame
co2.dm.df = data.frame(ncol = n.m, nrow = n.d)
for (i in 1:n.d) {
  for (j in 1:n.m) {
    co2.dm.df[i, j] = mean(CO2[decade == decade.vals[i] & month == month.vals[j]])
  }
}

# specify row and column names
row.names(co2.dm.df) = decade.vals
names(co2.dm.df) = month.abb[month.vals]
co2.dm.df
```

```
##           Jan      Feb      Mar      Apr      May      Jun      Jul
```

```
## 1950s 315.420 316.3100 316.5000 317.5600 318.1300 318.000 316.3900
## 1960s 319.690 320.3040 321.0810 322.2340 322.8630 322.350 321.1100
## 1970s 330.046 330.8440 331.6810 332.8710 333.4020 332.934 331.6280
## 1980s 344.378 345.1960 346.1430 347.4980 348.0810 347.512 346.0060
## 1990s 358.030 358.8975 359.8225 361.1112 361.7262 360.980 359.3637
##            Aug      Sep      Oct      Nov      Dec
## 1950s 314.6500 313.6800 313.1800 314.6600 315.4300
## 1960s 319.0890 317.5170 317.2680 318.4490 319.5770
## 1970s 329.8030 328.0740 327.9580 329.1670 330.3220
## 1980s 344.0610 342.2700 342.2550 343.6220 344.9090
## 1990s 357.2725 355.3975 355.5763 357.0263 358.4475
```

In many applications, "long" format is more convenient for analysis. The following code generates the means in long format. We will look at "long" versus "wide" data frames in more detail in a later installment on data wrangling.

```
# find means by both decade and month; store results in a "long" data frame
co2.dm.df = data.frame(ncol = 3, nrow = n.d*n.m)
rownum = 0
for (i in 1:n.d) {
  for (j in 1:n.m) {
    rownum = rownum + 1
    co2.dm.df[rownum, 1] = decade.vals[i]
    co2.dm.df[rownum, 2] = month.vals[j]
    co2.dm.df[rownum, 3] = mean(CO2[decade == decade.vals[i] & month == month.vals[j]])
  }
}

# specify variable names in data frame
colnames(co2.dm.df) = c("decade", "month", "CO2mean")
head(co2.dm.df)
```

```
##   decade month CO2mean
## 1  1950s     1  315.42
## 2  1950s     2  316.31
## 3  1950s     3  316.50
## 4  1950s     4  317.56
## 5  1950s     5  318.13
## 6  1950s     6  318.00
```

8.2 `for` loops for graphing by data subsets

Data with multiple subsets can be plotted in two ways:

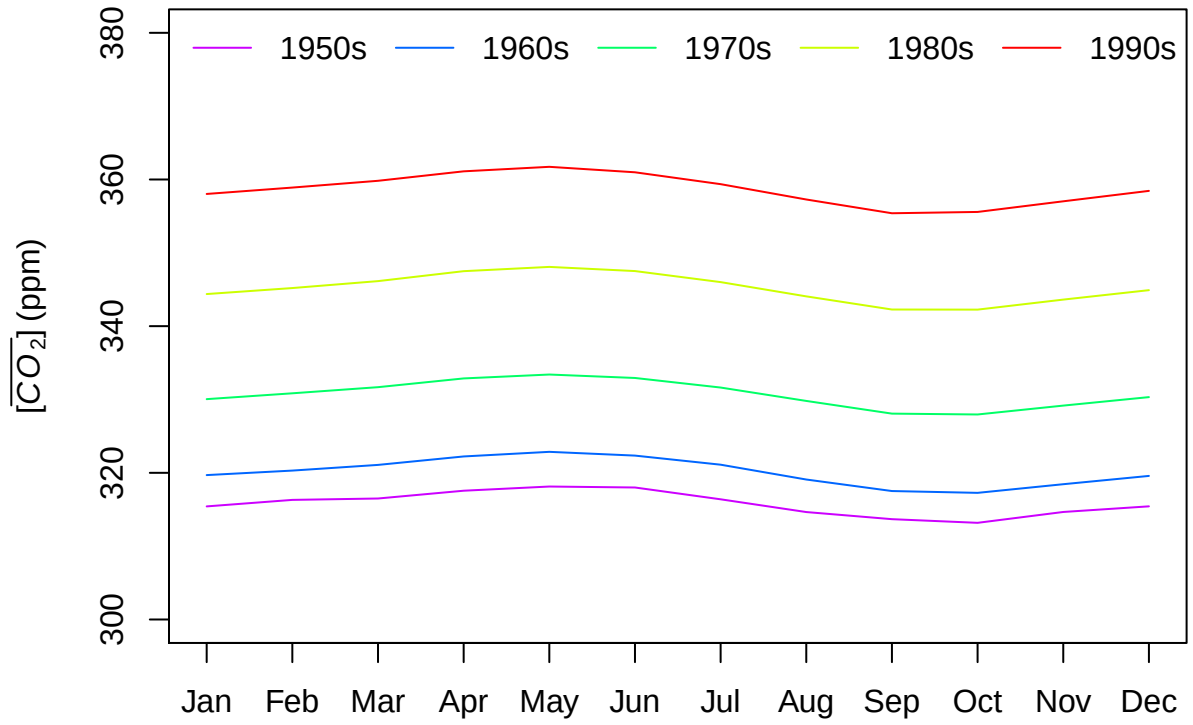
- in a single plot, with each data set represented by a different symbol and/or different line style or colour; or
- in a multi-panel plot, with each panel containing a subset.

In the terminology used in `Lattice` graphics and `ggplot2`, the former can be referred to as "grouped" by subset, and the latter referred to as "conditioned" by subset. We will look at the implementation of these concepts in `Lattice` and `ggplot2` graphics in a later installment.

As an example of a grouped plot, the code below generates a plot of the monthly mean CO_2 concentrations grouped by decade. Note the use of `type = "n"` to suppress plotting of the data in the `plot()` command. Also note the `rainbow()` function to generate the line colours.

```r
par(mar = c(5, 5, 1, 1))
plot(co2.dm.df$month, co2.dm.df$CO2mean, type = "n",
  xlab = "", xaxt = "n",
  ylim = c(300, 380),
  ylab = expression("["*italic(bar(CO[2]))*"]" ~ "(ppm)")
)
axis(side = 1, at = 1:12, labels = month.abb)

linecol = rev(rainbow(5))
for (i in 1:n.d) {
  ss = subset(co2.dm.df, decade == decade.vals[i])
  lines(ss$month, ss$CO2mean, col = linecol[i])
}
legend("top", bty = "n", legend = decade.vals, col = linecol, lty = 1, ncol = 5)
```

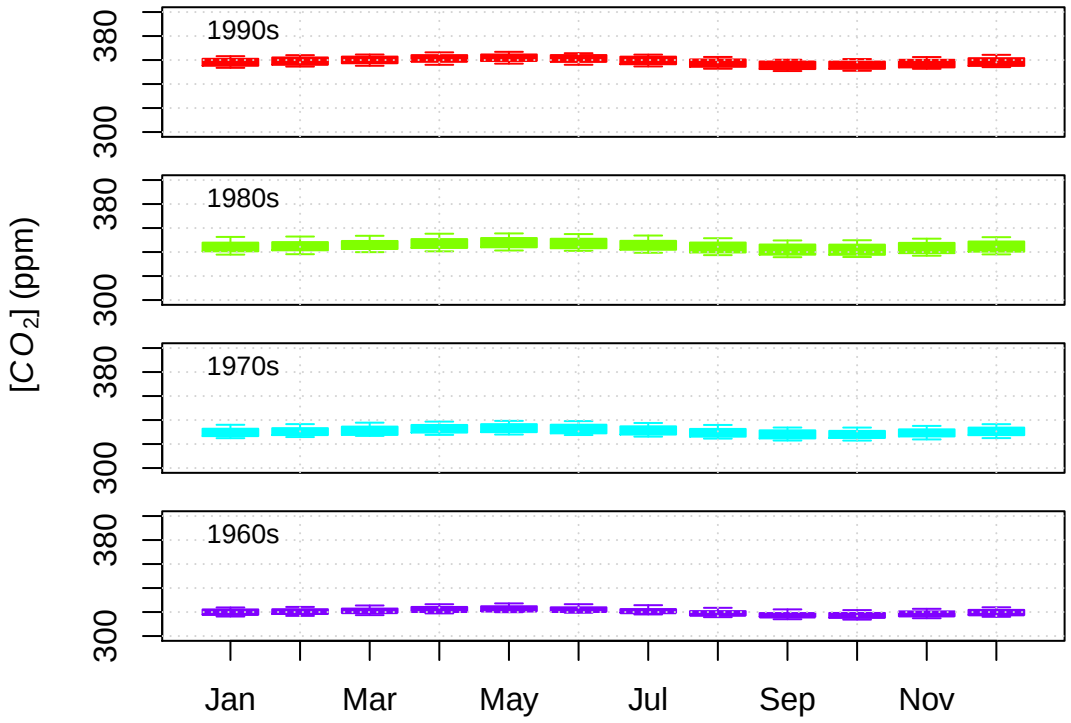


The code below generates boxplots of CO_2 by month, conditioned by decade. The 1950s are excluded due to the small number of observations.

The panels are arranged in one column and `nd` rows as controlled by `mfrow = c(nd, 1)`. Note that the `for` loop runs from `nd:1` – that is, in descending order. This will result in the most recent decade being plotted at the top and the earliest at the bottom.

```r
df = data.frame(year, month, CO2)
df2 = subset(df, year >= 1960)
df2$decade = paste0(floor(df2$year/10), "0s")
decs = unique(df2$decade)
nd = length(decs)
```

```
boxcol = rev(rainbow(nd))
par(mfrow = c(nd, 1), mar = c(0, 3, 1, 1), oma = c(5, 5, 0, 0), cex = 1)
for (i in nd:1) {
  ss = subset(df2, decade == decs[i])
  boxplot(ss$CO2 ~ ss$month,
          xaxt = "n", xlab = "n",
          col = boxcol[i],
          border = boxcol[i],
          ylim = c(300, 400)
         )
  grid()
  legend(x = 0, y = 360, bty = "n", decs[i], xjust = 0, yjust = 0, cex = 0.8)
}
axis(1, at = 1:12, labels = month.abb, outer = TRUE, line = 0)
mtext(side = 2, outer = TRUE, expression("["*italic(CO[2])*"]" ~ "(ppm)"))
```



8.3 The `aggregate()` function in base R

In this section, we look at the `aggregate()` function, which is a flexible tool for split-apply-combine operations. If you compare the code below to the earlier code that used `for` loops to compute subgroup means, the disadvantages of `for` loops for these types of data summaries are clear.

In a later installment, we will look at the `summarize()` function in the `dplyr` package, which is part of a set of packages known as the `tidyverse`. We will look at the use of `dplyr` and other packages in some detail in a later installment on data wrangling.

For a first example, we will generate mean CO_2 concentrations by month. Note that we are using vectors as arguments, and the result is a data frame.

```
co2m = aggregate(CO2, by = list(month), FUN = mean)
str(co2m)
```

```
## 'data.frame':    12 obs. of  2 variables:
##  $ Group.1: int  1 2 3 4 5 6 7 8 9 10 ...
##  $ x      : num  336 337 338 339 340 ...
```

```
co2m
```

```
##    Group.1        x
## 1        1 336.4308
## 2        2 337.2033
## 3        3 338.0546
## 4        4 339.2944
## 5        5 339.8821
## 6        6 339.3282
## 7        7 337.9164
## 8        8 335.9579
## 9        9 334.2428
## 10      10 334.1692
## 11      11 335.4679
## 12      12 336.6946
```

We can rename the variables in `co2m` to be more meaningful.

```
colnames(co2m) = c("month", "CO2")
co2m
```

```
##    month      CO2
## 1      1 336.4308
## 2      2 337.2033
## 3      3 338.0546
## 4      4 339.2944
## 5      5 339.8821
## 6      6 339.3282
## 7      7 337.9164
## 8      8 335.9579
## 9      9 334.2428
## 10    10 334.1692
## 11    11 335.4679
## 12    12 336.6946
```

We can also use a formula in place of the `by =` argument. Note that the resulting data frame has the column names equivalent to those in the `aggregate()` function call.

```
co2m = aggregate(CO2 ~ month, FUN = mean)
head(co2m)
```

```
##   month      CO2
## 1     1 336.4308
## 2     2 337.2033
## 3     3 338.0546
## 4     4 339.2944
## 5     5 339.8821
## 6     6 339.3282
```

If all of the variables are in the same data frame, it is useful to use the following form, in which the `data =` argument is provided to specify the data frame containing the vectors. We use `df2`, which excludes data from the 1950s.

```
co2m = aggregate(CO2 ~ month, data = df2, FUN = mean)
head(co2m)
```

```
##   month      CO2
## 1     1 336.9837
## 2     2 337.7532
## 3     3 338.6218
## 4     4 339.8663
## 5     5 340.4545
## 6     6 339.8895
```

Now, we will generate means by decade.

```
co2d = aggregate(CO2 ~ decade, data = df2, FUN = mean)
co2d
```

```
##   decade      CO2
## 1  1960s 320.1277
## 2  1970s 330.7275
## 3  1980s 345.1609
## 4  1990s 358.6376
```

Finally, we average by both decade and month. This time, we will use the formula notation in place of the `by =` argument. Note the effect of changing the order of "Decade" and "Month."

```
# decade first, then month
co2dm = aggregate(CO2 ~ decade + month, data = df2, FUN = mean)
head(co2dm)
```

```
##   decade month     CO2
## 1  1960s     1 319.690
## 2  1970s     1 330.046
## 3  1980s     1 344.378
## 4  1990s     1 358.030
## 5  1960s     2 320.304
## 6  1970s     2 330.844
```

```
# month first, then decade
co2dm = aggregate(CO2 ~ month + decade, data = df2, FUN = mean)
co2dm[1:20, ]
```

```
##    month decade     CO2
## 1      1  1960s 319.690
## 2      2  1960s 320.304
## 3      3  1960s 321.081
## 4      4  1960s 322.234
## 5      5  1960s 322.863
## 6      6  1960s 322.350
## 7      7  1960s 321.110
## 8      8  1960s 319.089
## 9      9  1960s 317.517
## 10    10  1960s 317.268
## 11    11  1960s 318.449
## 12    12  1960s 319.577
```

```
## 13     1  1970s 330.046
## 14     2  1970s 330.844
## 15     3  1970s 331.681
## 16     4  1970s 332.871
## 17     5  1970s 333.402
## 18     6  1970s 332.934
## 19     7  1970s 331.628
## 20     8  1970s 329.803
```

An important feature is that `aggregate()` can apply a user-defined function, as in the following example. In this case, the user-defined function uses the `lm()` (linear model) function to compute a linear regression between the CO_2 concentration and an index representing time, and then the `coef()` method to extract the regression coefficients as a vector (in which the first element is the intercept and the second the slope).

```
x_trend = function(x) {
  n = length(x)
  t_i = 1:n
  # fit linear regression and store result in an object named "reg"
  reg = lm(x ~ t_i)
  # extract slope
  coef(reg)[2]
}

co2.trend = aggregate(CO2 ~ month + decade, data = df2, FUN = x_trend)

colnames(co2.trend) = c("month", "decade", "CO2trend")
head(co2.trend)
```

```
##   month decade  CO2trend
## 1     1  1960s 0.8192727
## 2     2  1960s 0.8038788
## 3     3  1960s 0.8135152
## 4     4  1960s 0.8133333
## 5     5  1960s 0.7651515
## 6     6  1960s 0.7717576
```

```
tail(co2.trend)
```

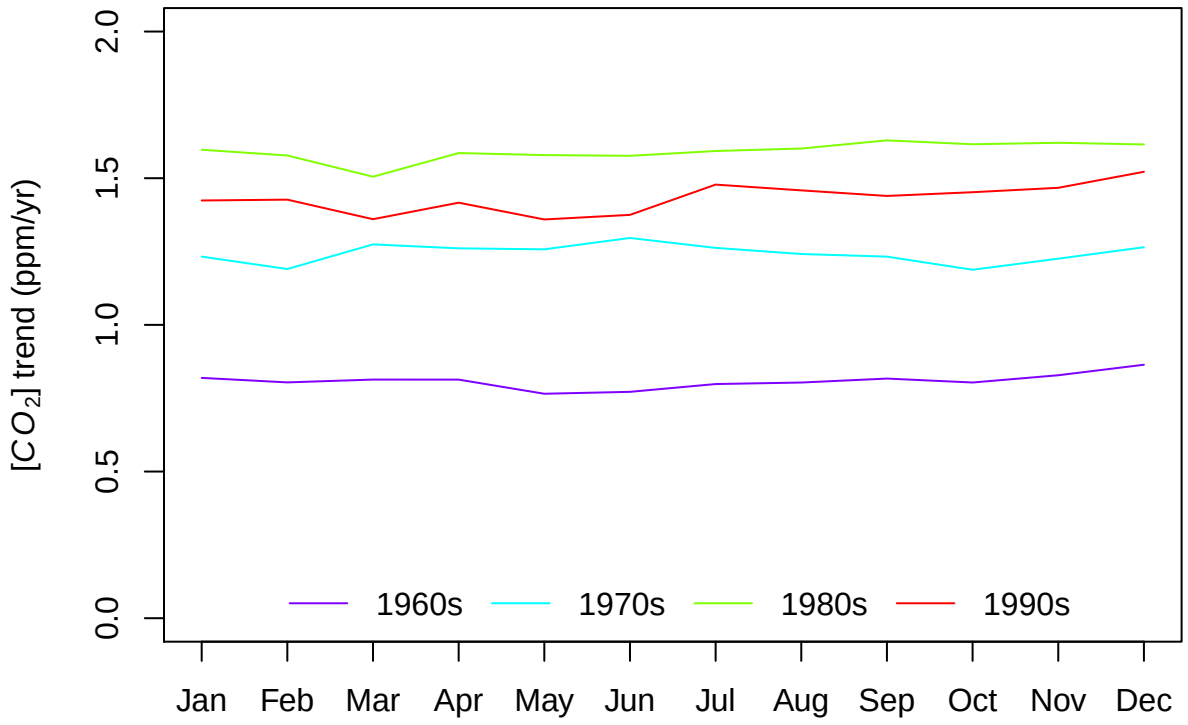
```
##    month decade CO2trend
## 43     7  1990s 1.478214
## 44     8  1990s 1.458571
## 45     9  1990s 1.439524
## 46    10  1990s 1.452262
## 47    11  1990s 1.467262
## 48    12  1990s 1.521905
```

Just for interest, we can plot these trends. As seen below, there does not appear to be any systematic seasonal pattern, but the strongest trends occurred in the 1980s.

```
decs = unique(co2.trend$decade)
n.d = length(decs)
par(mar = c(5, 5, 1, 1))
plot(co2.trend$month, co2.trend$CO2trend, type = "n",
  xlab = "", xaxt = "n",
  ylim = c(0, 2),
  ylab = expression("["*italic(CO[2])*"]" ~ "trend (ppm/yr)")
)
```

```r
axis(side = 1, at = 1:12, labels = month.abb)

linecol = rev(rainbow(n.d))
for (i in 1:n.d) {
  ss = subset(co2.trend, decade == decs[i])
  lines(ss$month, ss$CO2trend, col = linecol[i])
}
legend("bottom", bty = "n", legend = decs, col = linecol, lty = 1, ncol = n.d)
```



9 Writing efficient code

In programming, there are usually many different ways to accomplish the same end goal. However, these can differ greatly in the speed with which the program is executed even if they involve similar amounts of code.

When working with small to medium-sized data sets, it is often not critical to be careful about making code highly efficient. However, when working with large data sets and highly complex computations, speed may become of great concern.

One simple but valuable tool for evaluating the efficiency of your code is the `system.time()` function, which monitors how long a particular piece of code takes to execute. It is illustrated below to compare three approaches to computing the element-wise sums of two vectors:

- `f1()` uses vector addition;
- `f2()` creates a vector `z` of the appropriate length prior to entering the loop, and then uses a loop to compute each element; and

- `f3()` creates an empty vector prior to entering the loop, then uses a loop to extend the vector in each iteration by adding `x[i] + y[i]` using `c()`.

```
n = 1e5
x = runif(n, 0, 1)
y = runif(n, 0, 1)

f1 = function(x, y) x + y
system.time(f1(x, y))
```

```
##    user  system elapsed
##       0       0       0
```

```
f2 = function(x, y) {
  n = length(x)
  z = numeric(n)
  for (i in 1:n) {
    z[i] = x[i] + y[i]
  }
  z
}
system.time(f2(x, y))
```

```
##    user  system elapsed
##    0.03    0.00    0.03
```

```
f3 = function(x, y) {
  n = length(x)
  z = numeric()
  for (i in 1:n) {
    z = c(z, x[i] + y[i])
  }
  z
}
system.time(f3(x, y))
```

```
##    user  system elapsed
##   11.18   10.58   21.84
```

As the results of `system.time()` show, `f1()` is faster than `f2()`, although the times are so small for `n = 1e5` that it is difficult identify how much faster `f1()` is. However, it is clear that `f3()` is substantially slower than both.

To evaluate the differences between `f1()` and `f2()`, change `n` to 1e7 and re-run them. Do not run `f3()` with `n = 1e7` or you will be waiting for a while (half an hour on my computer).

While looking at the performance of individual parts of a code is interesting, it is more important to look at an entire program to determine which parts are the bottlenecks, then work on optimizing those parts. Code profiling is an important topic, and one that you should go back to as you get more experienced with programming and begin writing more complex programs. A great starting point is the following:

- http://adv-r.had.co.nz/Profiling.html

A few suggestions to bear in mind as you develop your programming skills follow.

- When writing code that uses loops,
  - vectorize as many calculations as possible
  - if a calculated quantity does not change from iteration to iteration, compute its value prior to the start of the loop

  - pre-define data objects to have the final size before the loop rather than adding new results to an existing data structure (e.g., expanding a vector using `c()`)
  - see Ligges and Fox (2008) for more details.

- When performing calculations row-wise or column-wise on a rectangular data structure (e.g., computing sums of each row),
  - use `apply()` or one of its variations rather than working row by row or column by column
  - use a matrix rather than a data frame, as extracting parts of a matrix is faster than extracting parts of a data frame

Notwithstanding the importance of writing efficient code, it is also important to consider the balance between the time it takes to write code and the time the code takes to run. If you are only going to run a piece of code one or a few times, it may not make sense to devote considerable time trying to optimize it beyond following simple guidelines like those mentioned above.

10 Debugging and error handling

10.1 General considerations

Writing code that either does not run or, even worse, runs but generates incorrect results is a universal experience in programming. The cause of errors can be as simple as incorrect syntax – e.g., typing `)` instead of `]` – or reflect an error in the underlying logic. An advantage of R Studio is that it will highlight a number of syntax errors, such as mis-matched brackets. Errors in the underlying logic are more difficult to diagnose, and require careful, step-by-step hand-checking of the code, running each line and checking to see that its output conforms with what the programmer intended.

As you develop your programming skills, you should always have in mind the concept of "defensive programming," which involves trying to anticipate where errors may arise (e.g., insufficient sample size in a data subset to perform some analysis) and including code to check for and accommodate those situations. For further information, consult http://adv-r.had.co.nz/Exceptions-Debugging.html.

10.2 The `try()` function

When running code that automates repetitive operations, it is not uncommon for an error to occur during one of the iterations. When this happens, the loop will terminate. This situation can be particularly frustrating when you have code that takes hours or days to run.

A useful approach to ensure that a loop will continue running after an error occurs is the `try()` function. The first argument of the function is an expression to be evaluated, which can be a user-defined function. The second argument controls whether the error message is printed to the console. The default is `silent = FALSE`. To suppress the error message being printed to the console, change the argument to `silent = TRUE`.

The example below is drawn from the function's documentation. The code creates a vector containing 50 values randomly generated from a normal distribution. In each iteration of the loop, the function `doit()` is applied. The `doit()` function creates a new vector by sampling, with replacement, the input vector. If there are fewer than 30 unique values in the sampled vectors, the function executes the `stop()` function, which generates an error action.

```
set.seed(123)
x <- stats::rnorm(50)

# execute loop with error trapping
doit <- function(x) {
  x <- sample(x, replace = TRUE)
```

```
  if(length(unique(x)) > 30) mean(x)
    else stop("too few unique points")
}

res1 <- vector("list", 100)
for (i in 1:100) res1[[i]] <- doit(x)
i # iteration in which the error occurred

res2 <- vector("list", 100)
for (i in 1:100) res2[[i]] <- try(doit(x), TRUE)
```

Try running the code chunk and then look at the values of `res1` and `res2`. Then try running the code with `silent = FALSE` in the `try()` function.

11 Algorithms

The term "algorithm" can be defined as a self-contained, step-by-step set of operations to be performed for calculation, data processing, and automated reasoning. Many data processing work flows are linear sequences of processing steps. Others, however, may involve complex sequences of branches based on different conditions. Especially in the latter case, it is important to develop a clear algorithm to guide the writing of the code and to communicate to others how the data were processed.

There are three approaches to describing algorithms:

- prose description
- flow chart (http://en.wikipedia.org/wiki/Flowchart)
- pseudocode (http://en.wikipedia.org/wiki/Pseudocode)

These will be illustrated below using the example of applying Newton's method to find the square root of a number.

11.1 Prose description

1. Input S, the number whose square root is desired, the maximum number of iterations (i_{max}), a tolerance to decide when to terminate iterations, and an initial guess, x.
2. Set $count = 0$.
3. Add 1 to *count*.
4. Update the initial guess as follows:
$$x_{new} == \frac{1}{2}(x + \frac{S}{x})$$
5. Compute $\Delta x = |x_{new} - x|$.
6. Set $x = x_{new}$.
7. If $\Delta x <$ tolerance or *count* greater than or equal to *i_max*, go to step 8; otherwise go to step 3.
8. If $\Delta x <$ tolerance, set msg = "convergence reached"; otherwise set msg = "convergence failed".
9. Return x, Δx, *count*, msg.

11.2 Flow chart

An explanation of flow chart symbols and the construction of flow charts can be found via the following link.

- https://www.programiz.com/article/flowchart-programming

The flow chart below is a graphic representation of the algorithm. It has been simplified slightly to keep it compact (it leaves out setting `msg`).

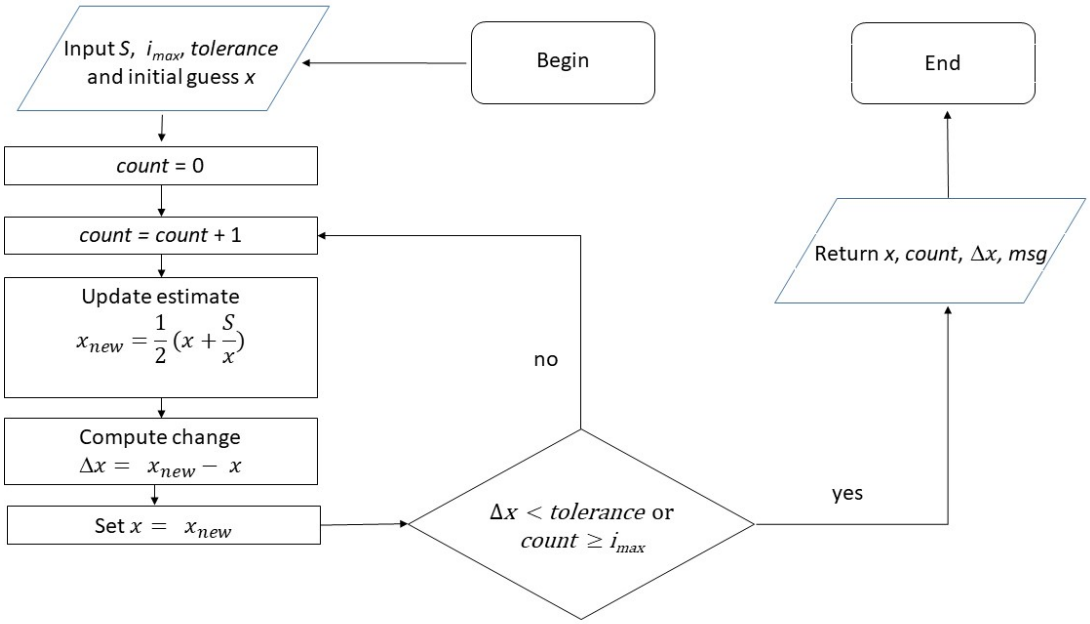


Figure 1: Flow chart illustrating Newton's method for finding a square root.

11.3 Pseudo-code

Pseudo-code aims to represent algorithms at a level of abstraction somewhere between a prose description and the actual code itself. The goal is to provide clarity of the algorithm to programmers working in any language, while making it relatively straightforward to translate the pseudo-code into an actual programming language. There is no uniform standard, and programmers working with different languages would likely base the structure and much of the syntax of their pseudo-code on the language they are programming within. The example below is loosely based on Pascal.

```
BEGIN
  Input S, x, tolerance, i_max
  Set count = 0
  REPEAT
    count = count + 1
    x_new = 0.5*(x + S/x)
    dx = x_new - x
    x = x_new
  UNTIL abs(dx) < tolerance or count = i_max
  IF abs(dx) < tolerance THEN set msg = "convergence reached"
    ELSE set msg = "convergence failed"
  Return x, dx, count, msg
END
```