MFE R Programming Workshop

Week 1

Brett Dunn

Fall 2016

Overview

Goals

- Learn to program in R.
- What does programming mean?
    - Language syntax.
    - Debugging.
    - Finding solutions.
    - Translating math to code.
- This is just the beginning, you'll develop these skills throughout the program.

R as a language

- R is object oriented.
  - Everything is an object and functions operate differently when passed different types of objects.
- R is functional.
  - You write fewer loops.
  - You write cleaner code.

R vs C++

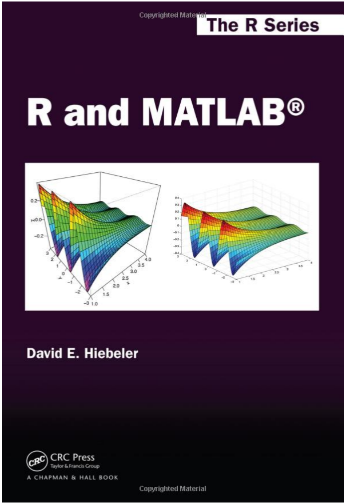
- Both are useful, and you will use both in the MFE program.
- R is an interpreted language.
  - Low programmer time.
  - A great tool for data munging, statistics, regressions, ect.
  - However, certain tasks in R can be slow (e.g. loops).
- C++ is very fast, but it takes longer to write programs.
- We can use both together!
- A good workflow:
  1. Write your program in R.
  2. If the program is too slow, benchmark your code.
  3. Try to speedup any bottlenecks in R.
  4. Convert any remaining bottlenecks to C++.

Jack of All Trades, Master of None

- You are better served by learning R and C++ very well, rather than trying to learn R, C++, MATLAB, Python, Julia, SAS, ect.
- The MFE program is just too short.
  - You also need to learn finance!
- Once you are proficent with R and C++, learning other languages is easy.
- Don't become a master of none!

MATLAB

- If you want to learn MATLAB after learning R, take a look at R and MATLAB by David Hiebeler.



Structure

- I will talk at the beginning of each class.
- For the remainder of the time you will break into your study groups and work on programming tasks.
- Tasks are designed to introduce you to the building blocks that will be used for course assignments throughout the MFE program.
- This course is a programming course with emphasis on methods for finance:
  - You will see finance terms and math.
  - You *may* not understand all of the finance, but you will learn it throughout the program.
- The key skills will be translating mathematical algorithms into code and developing the ability to find helpful resources.

Questions

Any questions before we start?

R Resources: Books

- Introductory:
  - R for Everyone by Jared P. Lander
  - R Cookbook by Paul Teetor (free at UCLA LearnIT)
  - R for Data Science by Hadley Wickham (free as well)
- Intermediate:
  - The Art of R Programming by Norman Matloff
- Advanced:
  - Software for Data Analysis by John Chambers
  - Extending R by John Chambers
  - Advanced R by Hadley Wickham

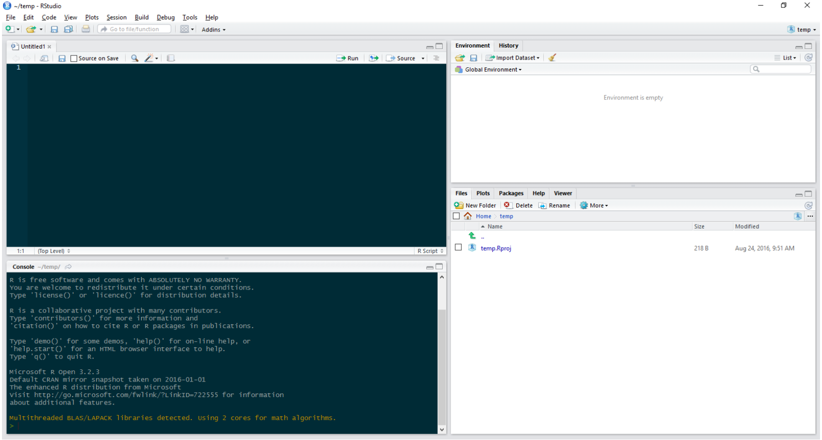
Other Resources

- Book series:
  - Use R! Springer series
    - FYI: Many Springer textbooks are just $25 through http://link.springer.com/. You need to be on campus or signed into the UCLA VPN. You can download the pdfs for free.
  - O'Reilly R Books (free at UCLA LearnIT)
- Built in documentation!
  - `?funcname`
- Journal of Statistical Software
- Data science courses on Coursera
- Data Camp
- https://www.r-bloggers.com/
- https://twitter.com/rstudiotips
- Google, Stack Overflow, ect.

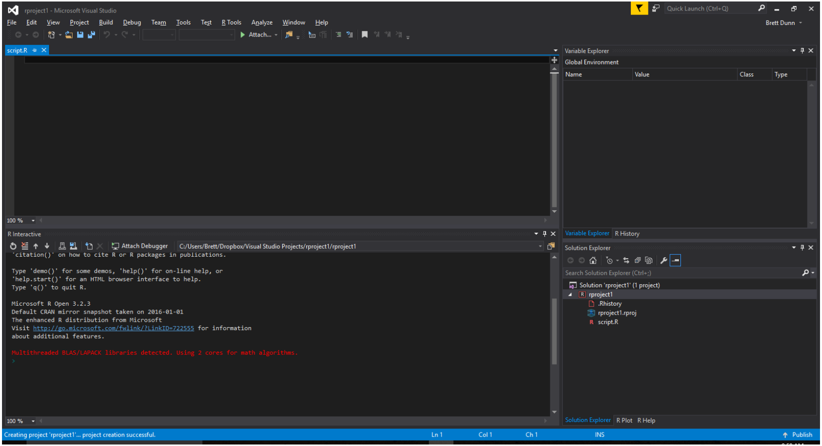
R Environment

- First, you need an R distribution.
  - I recommend Microsoft R Open.
  - `https://mran.revolutionanalytics.com/download/`
- Second, you need an integrated development environment (IDE) for R.
  - R Studio is a fantastic environment to interact with R.
  - Other options:
    - R Tools for Visual Studio if you use Visual Studio.
    - Emacs Speaks Statistics (ESS) if you use Emacs.
- I am going to assume that you have a working installation of R Studio and that you have a basic understanding of how it works.
- I will show you some Visual Studio.
- My focus is going to be on R programming.

RStudio

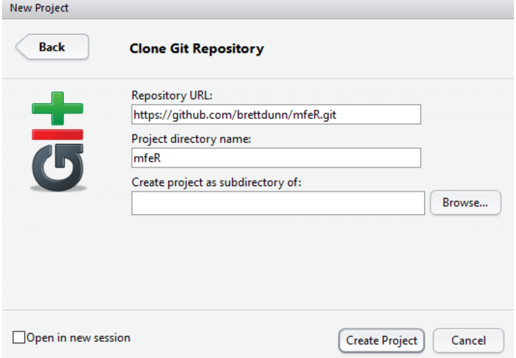

~/temp - RStudio
File Edit Code View Plots Session Build Debug Tools Help
Go to file/function
Addins
temp
Untitled1
Source on Save
Run
Source
1
1:1
(Top Level)
R Script
Console ~/temp/
R is free software and comes with ABSOLUTELY NO WARRANTY.
You are welcome to redistribute it under certain conditions.
Type 'license()' or 'licence()' for distribution details.
R is a collaborative project with many contributors.
Type 'contributors()' for more information and
'citation()' on how to cite R or R packages in publications.
Type 'demo()' for some demos, 'help()' for on-line help, or
'help.start()' for an HTML browser interface to help.
Type 'q()' to quit R.
Microsoft R Open 3.2.3
Default CRAN mirror snapshot taken on 2016-01-01
The enhanced R distribution from Microsoft
Visit http://go.microsoft.com/fwlink/?LinkID=722555 for information
about additional features.
Multithreaded BLAS/LAPACK libraries detected. Using 2 cores for math algorithms.
Environment History
Import Dataset
List
Global Environment
Environment is empty
Files Plots Packages Help Viewer
New Folder Delete Rename More
Home temp
Name Size Modified
temp.Rproj 218 B Aug 24, 2016, 9:51 AM

R Tools for Visual Studio



Course Materials

- https://github.com/brettdunn/mfeR
- The materials for this course were created in RStudio, using R Markdown.
- To create your own RStudio project:
  - File / New Project / Version Control / Git
  - Enter the URL



R Basics

Command Line Interface

- To run a command in `R`, type it into the console next to the > symbol and press the Enter key.

```
2 + 3
```

```
## [1] 5
```

- Up Arrow + Enter repeats the line of code.
- Esc (Windows/Mac) or Ctrl-C (Linux) interups a command.

RStudio

- To start, create a new R Script file.
  - File/New File/R Script
- You can type your commands in the R Script file and run them on the Console.
  - Easy way to save your work.
  - `Ctrl+Enter` sends the line at the cusor to the consule.
  - `Ctrl+Shift+S` runs the entite file.
  - Help/Keyboard Shortcuts lists all the available shortcuts.
    - Check out the multiple cursors.
- For larger tasks with many files, create an R project.
- Visual Studio is similar.

General Comments

- Make your code easy to read.
- Check out Google's R Style Guide
- Comment your code!

Google's R Style Guide

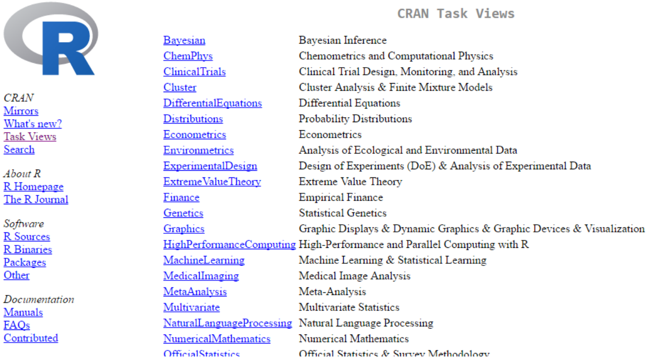
R is a high-level programming language used primarily for statistical computing and graphics. The goal of the R Programming Style Guide is to make our R code easier to read, share, and verify. The rules below were designed in collaboration with the entire R user community at Google.

Summary: R Style Rules

1. File Names: end in `.R`
2. Identifiers: `variable.name` (or `variableName`), `FunctionName`, `kConstantName`
3. Line Length: maximum 80 characters
4. Indentation: two spaces, no tabs
5. Spacing
6. Curly Braces: first on same line, last on own line
7. else: Surround else with braces
8. Assignment: use `<-`, not `=`
9. Semicolons: don't use them
10. General Layout and Ordering
11. Commenting Guidelines: all comments begin with `#` followed by a space; inline comments need two spaces before the `#`
12. Function Definitions and Calls
13. Function Documentation
14. Example Function
15. TODO Style: `TODO(username)`

R Packages

- A package is essentially a library of prewritten code designed to accomplish some task or a collection of tasks.
- R has a huge collection of user-contributed packages.
  - Warning: Not all packages are of the same quality.


CRAN
Mirrors
What's new?
Task Views
Search

About R
R Homepage
The R Journal

Software
R Sources
R Binaries
Packages
Other

Documentation
Manuals
FAQs
Contributed

CRAN Task Views

Bayesian	Bayesian Inference
ChemPhys	Chemometrics and Computational Physics
ClinicalTrials	Clinical Trial Design, Monitoring, and Analysis
Cluster	Cluster Analysis & Finite Mixture Models
DifferentialEquations	Differential Equations
Distributions	Probability Distributions
Econometrics	Econometrics
Environmetrics	Analysis of Ecological and Environmental Data
ExperimentalDesign	Design of Experiments (DoE) & Analysis of Experimental Data
ExtremeValueTheory	Extreme Value Theory
Finance	Empirical Finance
Genetics	Statistical Genetics
Graphics	Graphic Displays & Dynamic Graphics & Graphic Devices & Visualization
HighPerformanceComputing	High-Performance and Parallel Computing with R
MachineLearning	Machine Learning & Statistical Learning
MedicalImaging	Medical Image Analysis
MetaAnalysis	Meta-Analysis
Multivariate	Multivariate Statistics
NaturalLanguageProcessing	Natural Language Processing
NumericalMathematics	Numerical Mathematics
OfficialStatistics	Official Statistics & Survey Methodology

R Packages

- Installing a packages:
  - `Ctrl+7` in RStudio accesses the packages pane
  - You can also type `install.packages("packageName")`
- Uninstalling a package:
  - `remove.packages("packageName")`
- Loading packages:
  - `require(packageName)` or `library(packageName)` loads a package into R
  - The difference is that `require` returns `TRUE` if the package loads or `FALSE` if it doesn't.
- Unloading packages
  - `detach(package:packageName)`
- If two packages have the same function name use two colons: -`package1::func` or `package2::func`

Variables

- Unlike C++, R does not require variable types to be decleared.
- A variable can take on any data type.
- A variable can also hold any R object such as a function, the result of an analysis, a plot, ect.
- Variable assignment is done with `<-`.
  - = works, but there are reasons to prefer `<-`.
- We can remove variables (e.g. to free up memory) with the `rm` function. `gc()` runs garbage collection.

```
x <- 2  # x is a pointer
x  # the same output as print(x)
```

```
## [1] 2
```

```
rm(x)  # removes x
```

Data Types

- There are many different data types in R.
- The four main types of data most likely to be used are:

  1. `numeric`
  2. `character` (string)
  3. `Date/POSIXct` (time-based)
  4. `logical (TRUE/FALSE)`

- The data type can be checked with the `class` function

```
x <- as.Date("2010-12-21")
class(x)
```

```
## [1] "Date"
```

Casting

```
x <- "2010-12-21"
class(x)
```

```
## [1] "character"
```

```
x
```

```
## [1] "2010-12-21"
```

```
x <- as.Date(x)
class(x)
```

```
## [1] "Date"
```

```
x
```

```
## [1] "2010-12-21"
```

More Casting

```r
x <- as.numeric(x)
class(x)
```

```
## [1] "numeric"
```

```r
is.numeric(x)
```

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

```r
x  # number of days since Jan 1, 1970
```

```
## [1] 14964
```

Even More Casting

```r
x <- as.integer(x)  # x <- 14964L assigns an integer
class(x)
```

```
## [1] "integer"
```

```r
is.integer(x)
```

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

```r
is.numeric(x)  # R promotes int to numeric as needed
```

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

```r
4L / 5L
```

```
## [1] 0.8
```

Logicals

```
# TRUE == 1 and FALSE == 0
x <- TRUE  # TRUE, FALSE, T, F are logicals
is.logical(x)
```

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

```
5 == 5  # != tests for inequality
```

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

```
"a" < "b"  # works on characters as well
```

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

Vectors

Vectors

- A `vector` is a collection of elements, all of the same type.
- We will learn about:
  - Recycling
    - The automatic lengthening of vectors.
  - Filtering
    - The extraction of subsets of vectors.
  - Vectorization
    - Where functiions are applied element-wise to vectors.

Vectors and Assignment

- Assigning values to variables can be done with <-
- Create vectors of numbers using the function `c()`

```
myvector <- c(1,2,3,4)
samevector <- 1:4
vectorsubset <- myvector[1:3]
vectorsubset
```

```
## [1] 1 2 3
```

c() function

- `c()`, which stands for concatenate is a very flexible function.

```
myvec1 <- 2:4
myvec2 <- c(1,3,5,myvec1)
myvec2
```

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

Accesing elements of vectors

- Elements can be accessed using []

```
myvec <- c(2,4,6,8)
myvec[4]
```

```
## [1] 8
```

```
myvec[c(1,3)]
```

```
## [1] 2 6
```

Length

- `length()` returns the vector length

```r
myvec <- 1:23
length(myvec)
```

```
## [1] 23
```

Recycling

- Vectors are recycled when an operation acts elementwise
- Be careful with and aware of this behavior!
- In some cases it is useful, others confusing

```r
vec1 <- 1
vec2 <- 1:4
vec3 <- 1:2
vec2+vec1
```

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

```r
vec2+vec3
```

```
## [1] 2 4 4 6
```

seq and rep

- Useful functions for generating vectors
- See ?seq and ?rep for details

```
myvec1 <- seq(1,10,2)
myvec1
```

```
## [1] 1 3 5 7 9
```

```
myvec2 <- rep(c(1,2),3)
myvec2
```

```
## [1] 1 2 1 2 1 2
```

```
rep(c(1,2), each=2)
```

```
## [1] 1 1 2 2
```

NULL and NA

- `NULL` is the non-existent value in R
- `NA` is the missing place hold

```
myvec1 <- 5:8
myvec1[2] <- NA
myvec1
```

```
## [1]  5 NA  7  8
```

```
myvec2 <- NULL
length(myvec2)
```

```
## [1] 0
```

Filtering (1)

- Select subsets using vectors of logicals

```
vec1 <- 1:5
vec2 <- c(TRUE,FALSE,TRUE,FALSE,TRUE)
vec1[vec2]
```

```
## [1] 1 3 5
```

Filtering (2)

- Select subsets using vectors of logical

```
vec1 <- 1:6
vec2 <- vec1 > 3
vec2
```

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

```
vec1[vec2]
```

```
## [1] 4 5 6
```

```
vec1[vec1>3]
```

```
## [1] 4 5 6
```

Assigning to filter

- You can assign to the subsets

```
vec1 <- 1:6
vec1[vec1<2] <- NA
vec1
```

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

Functions on vectors

- Functions typically operate on vectors

```
x <- 1:1000
mean(x)
```

```
## [1] 500.5
```

names

- You can give names to elements of vectors

```
myvec <- 1:3
names(myvec) <- c("A","B","C")
myvec["B"]
```

```
## B
## 2
```

Matrices

Intro

- Matrices are vectors with a number of rows and number of columns attribute

```r
myvec <- 1:10
mymat <- matrix(myvec, nrow=2, ncol=5)
mymat
```

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

```r
mymat[1,3]
```

```
## [1] 5
```

Matrix operations

- Many matrix operations are surrounded by % signs

```
mymat1 <- matrix(1:4, nrow=2)
mymat2 <- matrix(5:8, nrow=2)
mymat1 %*% mymat2
```

```
##      [,1] [,2]
## [1,]   23   31
## [2,]   34   46
```

```
mymat1 + mymat2
```

```
##      [,1] [,2]
## [1,]    6   10
## [2,]    8   12
```

apply

- `apply` allows you to apply a function across a dimension of a matrix
- The third argument is a function!

```
mymat1 <- matrix(1:10, nrow=2)
# mean across rows
apply(mymat,1,mean)
```

```
## [1] 5 6
```

cbind and rbind

- Column bind and Row bind

```
mymat1 <- matrix(1:4,nrow=2)
mymat2 <- matrix(6:9,nrow=2)
mymat3 <- matrix(10:11,ncol=2)
cbind(mymat1,mymat2)
```

```
##      [,1] [,2] [,3] [,4]
## [1,]    1    3    6    8
## [2,]    2    4    7    9
```

```
rbind(mymat1,mymat3)
```

```
##      [,1] [,2]
## [1,]    1    3
## [2,]    2    4
## [3,]   10   11
```

Lists

Intro

- Lists are where you really start to see the advantages of R as a statistics and data manipulation language

```
element1 <- 1:5
element2 <- matrix(1:6,nrow=3)
mylist <- list(el1=element1,el2=element2)
mylist[["el1"]]
```

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

```
mylist[["el2"]]
```

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

Subsetting lists

- Subsets of lists are with single []

```
mylist <- list(A=1,B=2,C=3,D=4)
# this returns a list because of the single []
mylist[c(1,3)]
```

```
## $A
## [1] 1
##
## $C
## [1] 3
```

lapply

- `lapply` implicitly loops over each list element and applies a function

```
mylist <- list(A=1:10,B=2:17,C=745:791)
lapply(mylist,mean)
```

```
## $A
## [1] 5.5
##
## $B
## [1] 9.5
##
## $C
## [1] 768
```

lapply example

```r
g <- c("M","F","F","I","M","M","F")
lapply(c("M","F","I"),function(gender) which(g==gender))
```

```
## [[1]]
## [1] 1 5 6
##
## [[2]]
## [1] 2 3 7
##
## [[3]]
## [1] 4
```

Data Frames

Intro

- In my mind, `data.frame` is the core data type in R
- The nicest part is that they can hold different types
- Each column must be the same length

```
dat1 <- 1:4
dat2 <- rep(c("A","B"),each=2)
myframe <- data.frame(col1=dat1,col2=dat2)
myframe
```

```
##   col1 col2
## 1    1    A
## 2    2    A
## 3    3    B
## 4    4    B
```

Subsets

- Subsetting is just like a matrix

```
myframe[1,2]
```

```
## [1] A
## Levels: A B
```

```
myframe[1,]
```

```
##   col1 col2
## 1    1    A
```

```
myframe[,2]
```

```
## [1] A A B B
## Levels: A B
```

Reading in data

- Reading in data typically gives you a `data.frame`
- `read.table` is the basic function to read in tabular data
- `read.csv` is a special cast of `read.table`
- As usual see `?read.table`
- Often you want to set `stringsAsFactors = FALSE`

Adding columns

```r
dat1 <- 1:4
dat2 <- rep(c("A","B"),each=2)
myframe <- data.frame(col1=dat1,col2=dat2)
myframe$col3 <- 5:8
myframe
```

```
##   col1 col2 col3
## 1    1    A    5
## 2    2    A    6
## 3    3    B    7
## 4    4    B    8
```

names

- column names in data.frames are specificed by `names()`
- this is because data.frames are actually lists with special attributes
- that means that the usual list functions work on data.frames
- lapply, etc

Long example

```r
all2006 <- read.csv("2006.csv",header=TRUE,as.is=TRUE)

# exclude hourly-wagers
all2006 <- all2006[all2006$Wage_Per=="Year", ]

# exclude weird cases
all2006 <- all2006[all2006$Wage_Offered_From > 20000,]

all2006$rat <- all2006$Wage_Offered_From
                  / all2006$Prevailing_Wage_Amount

se2006 <- all2006[grep("Software Engineer", all2006),]
```

Control Statements

For loops (1)

```r
x <- c(1:5)
y <- NULL
for(i in 1:length(x)) {
   y[i] <- x[i] + 2
}
y
```

```
## [1] 3 4 5 6 7
```

For loops (2)

- or another nice way to make a for loop

```
x <- c(1:3)
for(element in x) {
   print(element + 2)
}
```

```
## [1] 3
## [1] 4
## [1] 5
```

While loops

```r
x <- c(1:5)
y <- NULL
i <- 1
while(i<=length(x)) {
   y[i] <- x[i] + 2
   i <- i+1
}
y
```

```
## [1] 3 4 5 6 7
```

Conditional Statements

```r
x <- -10
myabs <- x
if(x<0) {
    myabs <- -x
}
myabs
```

```
## [1] 10
```

Functions

Function definitions

- Note that the last value evaluated is what is returned by the function

```
myfunc <- function(x) x^2
myfunc(10)
```

```
## [1] 100
```

Scope rules

- Variables defined inside a function are local to that function

```
myfunc <- function(x) {
    N <- 10
    N*x^2
}
myfunc(10)
```

```
## [1] 1000
```

```
# You can't access N out here
```

Lab 1

Let's work on Lab 1.