Chapter 1. Introduction
R Commentary

In the R Commentaries, we provide supplemental information supporting the sample R code within the QF Repository. First, we provide some insights on why the R language is an excellent entry point for finance professionals.

Why learn the R language?

Job postings

Many jobs available for quantitative financial analysts require some knowledge of computer programming and particularly R. For example, any search of quantitative finance job board and you will see that knowledge of computer programming is often required. Clearly, knowledge of R makes learning any other languages easier. Although the computer language of choice changes over time, R is an excellent language to learn initially.

History of computer programming languages

In the mid-1950s John Backus of IBM led a group of computer programmers to create the FORTRAN (formula translation) programming language. FORTRAN was the first high-level computer programming language. A programming language is said to be high level if it hides complex details of the software's interactions with the computer. Low-level programming languages, such as machine code and assembly languages, explicitly manage the complex details of the software's interactions with the computer.

With the development of the C language as well as the enhancements of C++, computer programming took a major step forward. C++ combines both 'high level' (easy to use) and 'low level' (powerful) features. C++ supports the object-oriented approach to programming; this approach permits the combination of data with methods allowing for higher levels of abstraction. Historically, early languages followed the procedural approach, designed as a collection of functions that manage and manipulate data. For example, the cumulative distribution function, denoted $N(d)$, can be viewed as an object that contains data (d) and methods (solving for $N()$). Viewed as an object containing both the data input and the methods, $N(d)$ can be incorporated into many different option valuation and risk management solutions.

Bjarne Stroustrup developed C++ in the late 1970s as a highly optimized, object-oriented language. Hence it is very fast, a feature attractive to the finance industry. Quantitative finance problems are suitable for object-oriented programming[1] as contrasted with sequential programming. Unfortunately, C++ remains difficult for many financial analysts. There have been numerous other languages that sought to simplify C++ while preserving its speed and versatility.

According to Wikipedia, R was created in the early 1990s with the first version available in 1995. R was based on another programming language known as S. Further, R has the capacity to link with C++ as well as other current popular languages such as Python. Hence, we rely on R here as an introductory programming language.

Rapid application development

One goal of quantitative analysts is to be able to implement their solutions within an organization rapidly. Hence, it is necessary to be able to both build applications fast, as well as implement them with ease. The object-oriented approach combined with R permits rapid application development (RAD). With many well developed objects prebuilt and rigorously tested, it is relatively simple to complete the implementation of a quantitative model in a very short period of time.

File types

R code is typically stored in files with the .R extension. The extension .RPROJ is a project file that aids in managing information related to a specific project. It is created in and used by RStudio. The extension

[1]Object-oriented programming (OOP) with C++ is typically identified with four characteristics: encapsulation, inheritance, polymorphism, and abstraction. Object-oriented programming is addressed further in chapter 2.

.RDATA is a data file that contains readable information for use in R. The extension .RHISTORY contains information related to prior work within a particular R project.

Why learn R this way?

Autonomous versus heteronomous

Most programming books and other training materials choose to allow the learner to be autonomous when it comes to the version (say Mac or PC) of the given language. These materials have the advantage of being applicable to any version, so long as it complies with the existing standards established for R.

For our objectives, allowing the learning to be autonomous (having freedom to act independently) versus heteronomous (subject to external standard) was a difficult choice. We chose to limit this material to a particular version (RStudio), rather than make the learning experience generic and applicable for any version.

The primary motive is to allow the financial analyst to be equipped to demonstrate his or her work in an efficient way. Within a very short period, you will be able to create your own R code that is user friendly. Our primary objective, however, is not to make you a professional programmer. To this end we seek the delicate balance between keeping the concepts as simple as possible while at the same time permitting the user to develop professional solutions.

For the autonomous among us, care is taken to separate the interface code (code focused on interactions with the end-user) and the implementation code (code focused on computing the solution). Because of this separation, it is easy to export the implementation code to other versions and there develop different graphical user interfaces (GUI).

Current platform

The current platform used during the production of this material is RStudio (Version 2023.06.1+524 (2023.06.1+524) on a Mac). Most likely, by the time you read this, it will be a different version. Often various packages cease to function correctly when you upgrade RStudio, so you may only update when you have time to fix new problems.

Deliverables

The main deliverables illustrated in this material are simple prototype R programs. If the goal is to produce fully implementable programs in real time, then the computer source code should be refined by professional software programmers. For example, no effort is made to exhaustively error trap inputs (for example, real numbers as opposed to alphabetic characters) and test for inputs that are out of range (for example, volatility equal to zero causing a division by zero).

Sample programs

1.1 Program Layout.R

In the remainder of this book there will be references to sample programs. Depending on the version, the repository directory will have the general structure that corresponds to chapters in this book.

Thus, the program illustrated below is in the folder, Ch 1 Introduction (1.1 Program Layout). To illustrate the process, we provide a simple program that runs a console application that does not do anything. The source code will be set in Courier type as follows.

```
# 1.1 Program Layout.R
# RStudio layout
# This page is the File Window, multiple files open at one time
# The page to the immediate right is the Environment Window,
#  with the Environment tab and History tab
# The page below is the Console Window, give interface history
# The page below to the right is the Information Window,
#  Files tab shows Project files (files within the Project folder),
#  Plots tab shows plots,
#  Packages show attached and some unattached packages
#  Help shows any requested help documentation
# Program Layout
help(base) # One way to get help, very cryptic
library(help = "base") # Control goes to the document
help("Arithmetic") # Details on arithmetic operators
library(base) # This line is not necessary but is the way to include libraries
# To load and attach add-on packages, either:
library(stats) # Statistics package is included
help(stats) # Help tab now has details on the R Stats package
```

```
# Or:
require(Rcpp) # Rcpp package is included
?"Rcpp" # Alternative way to have details on the R Rcpp package
# R does not have a formal entry point like C++ (e.g., int main())
x = 100
# When the line above is run, you should see Values x 100 in the Environment window
# and x = 100 in the Console window
# The next line closes R
quit()
```

The first line is a comment line (starts with #) and is ignored when the program is implemented. Within the text, we will direct you to the location of the functioning programs provided at http://www.robertebrooks.org/project/buildiingqfawr/.

Figure 1R.1 illustrates what you should see when Module 1.1 is loaded into RStudio. What you see is likely a bit different depending on your operating system and RStudio version. We point out just a few items. The very top line, in faint gray, is the path along with the name of the R code file in focus within the File Window. The tab, 1.1 Program Layout.R*, is in red and the asterisk indicates changes have been made before the last time it was saved. The Environment Window is in the top right. The Environment tab will contain variables and values while the program is running. The Console Window will provide information important while the program is running. Finally, the Information Window is in the bottom right. The Files tab shows the project files and any other files contained within its folder. Plots can be found at the Plots tab. If the Information Window is too small, then the plots will fail to appear. The Packages tab is important when it is necessary to include different packages that have already been installed in the past. For example, blogdown is a package used to build websites and would appear in the Packages tab if you have ever installed it in the past. The Help tab is very useful as you begin to learn different built-in functions.

Figure 1R.1. Illustration of RStudio

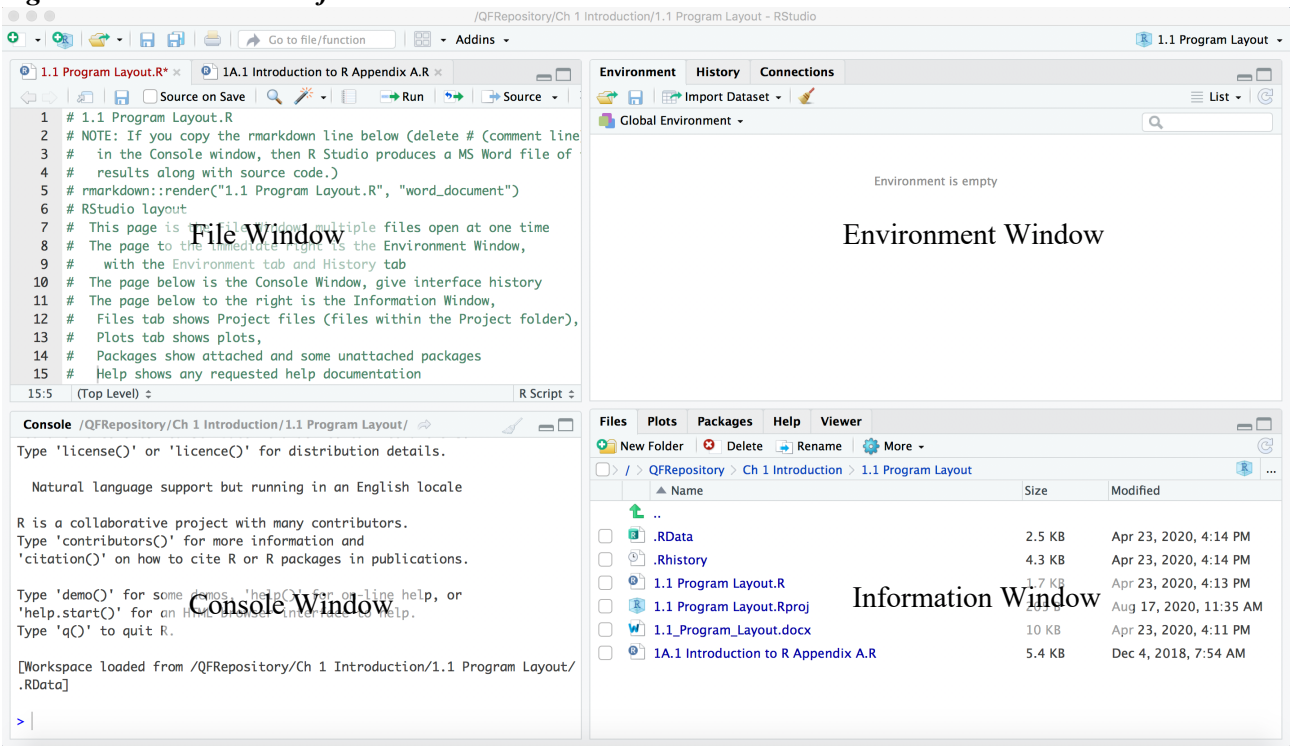
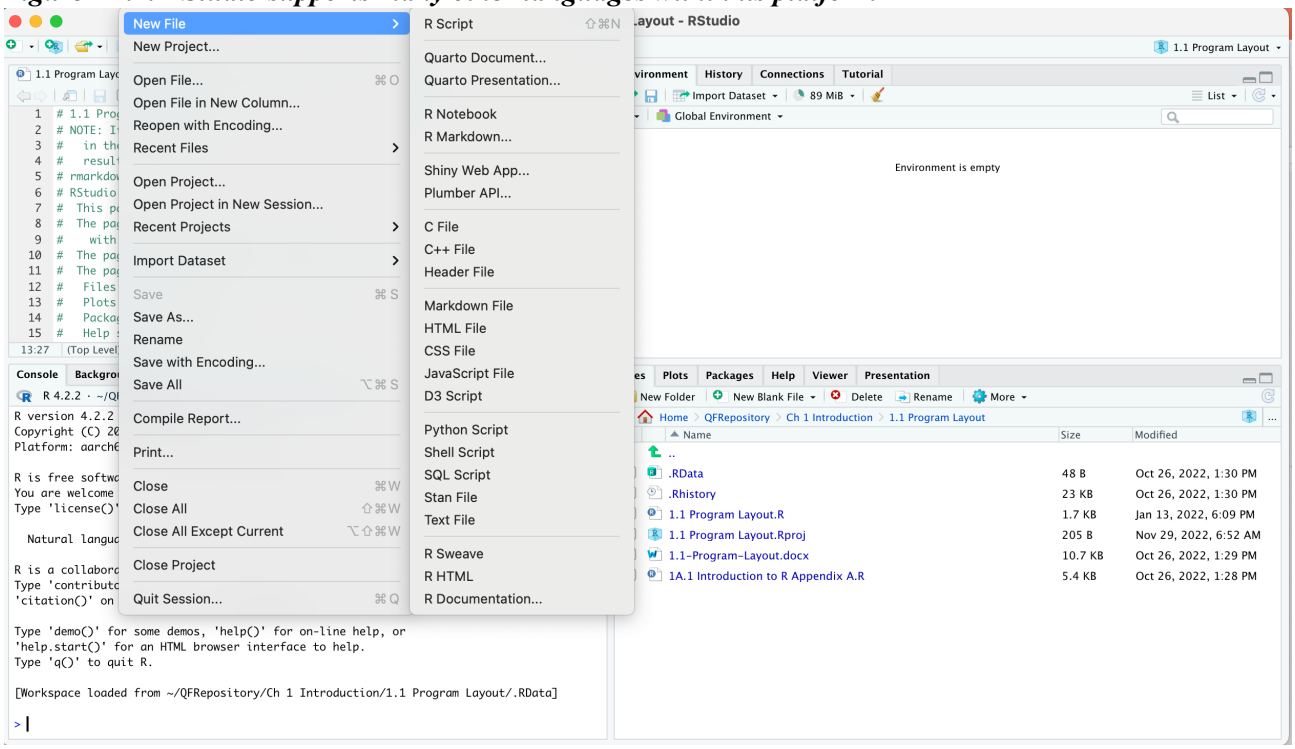


Figure 1R.2 illustrates that RStudio supports many other languages, including C, C++, Python, Shiny (html), SQL, and many more.

Figure 1R.2. RStudio supports many other languages within its platform



1.2 Heating Degree Days.R

This module presents a simple program that reads in a comma separated file and produces a plot of the data. Heating degree days is a statistic used related to measuring temperature. At this point, the technical details are not important.

From Investopedia, we have the following definition: “The number of degrees that a day's average temperature is below 65 degrees Fahrenheit (approx. 18 degrees Celsius), the temperature below which buildings need to be heated. The price of weather derivatives traded in the winter is based on an index made up of monthly HDD values.”

The following is the HDD Test.R program.

```
# HDD Test.R
# rmarkdown::render("1.2 HDD Test.R", "word_document")
# HDD - heating degree day
#   "The number of degrees that a day's average temperature is below 65 degrees
#   Fahrenheit (approx. 18 degrees Celsius), the temperature below which
#   buildings need to be heated. The price of weather derivatives traded in the
#   winter is based on an index made up of monthly HDD values." Investopedia
rm(list = ls())  # Take out the Environment "trash"
# library(date)
# library(stats) # lm() - linear model
# Read a file and create data frame
```

The Atlanta airport (ATL) heating degree data for a particular year is in the file named ATLHDD.csv. The content of this file is imported into the R program based on the next line of code.

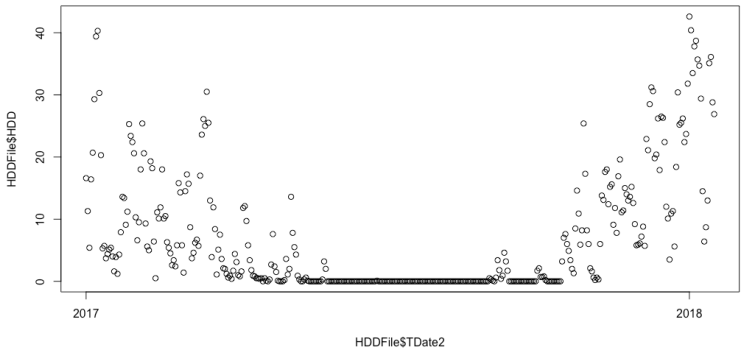
```
HDDFile <- read.table("ATLHDD.csv", header = TRUE, sep = ",")
head(HDDFile, 5) # Show first 5 lines in Console
tail(HDDFile, 5) # Show first 5 lines in Console
is.data.frame(HDDFile) # TRUE: HDDFile is a data frame
is.numeric(HDDFile$Date) # FALSE: Imported as a factor
is.factor(HDDFile$Date) # TRUE: A categorical variable, idealy limited in number (not here)
is.numeric(HDDFile$HDD) # TRUE: Numeric data suitable for mathematical calculations
is.numeric(HDDFile$Estimated) # TRUE
is.ordered(HDDFile$Date) # FALSE: It is stored as a factor variable
is.character(HDDFile$Date) # FALSE
max(HDDFile$Estimated) # Estimated variable is not relevant here, returns the max
min(HDDFile$Estimated) # Minimum reported to the console
# Often data sets have columns that need to be deleted
```

```
HDDFile$Estimated <- NULL # Estimated column is removed
head(HDDFile, 5)
HDDFile$TDate <- as.character(HDDFile$Date) # Convert factor to character
is.factor(HDDFile$TDate) # Now FALSE
is.character(HDDFile$TDate) # Factor Date converted to character TDate
# Convert character date (TDate) to Julian date but display in calendar format
#   Note default output format is YYYY-MM-DD, format = "..." denotes input
HDDFile$TDate2 <- as.Date(HDDFile$TDate, format = "%m/%d/%y")
# Insights on dates
head(HDDFile, 5)
# Insights on dates
as.integer(HDDFile$Date) # Nonsense and definitely not a Julian date
as.integer(HDDFile$TDate) # Cannot convert character string to integer, NAs
as.integer(HDDFile$TDate2) # Julian dates, base = 1/1/1970
BaseDate = as.Date("1970-01-01")
as.integer(BaseDate)
head(HDDFile, 5)
# Some simple plots
# plot(HDDFile$Date, HDDFile$HDD) # x-axis not formatted correctly, nonsense
# plot(HDDFile$TDate, HDDFile$HDD) # TDate not formatted for plotting
```

As we will explore in detail in module 3.1, managing calendar dates will always be challenging.

```
plot(HDDFile$TDate2, HDDFile$HDD)
# Thus, when programming, managing dates is tricky, and you must be careful
```

The plot is reproduced here.



Appendix 1R.A: R Coding Preferences

Learning objectives

- Determine preferences related to expressing and formatting R code
- Provide guidance for naming variables, methods, classes, and files

Introduction

Writing R code is part mundane implementation of mathematical ideas and part free-spirited, written expressions of abstract art. The goal here is not to squelch your artistic side, but to provide a consistent framework for you to begin to express your quantitative solutions using R.

Coding preferences

Accuracy and readability are vital for any successful R implementation of a quantitative finance problem. Highly readable source code that is inaccurate is nonetheless fatally flawed. It is not, however, difficult to fix readable code. It is highly recommended that you comment your code extensively so that you are able to retrace your steps as well as document what you have done in each piece of code for both debugging and reuse of code. Although accurate but illegible code will work, it cannot be easily maintained.

R code presentation

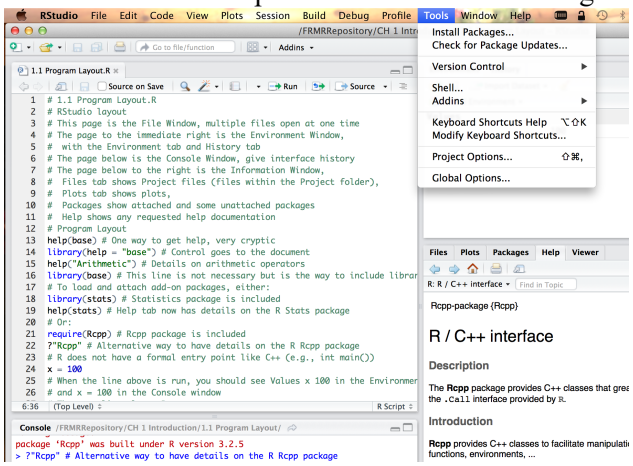
The first decision when expressing R code relates to spacing. It is important to keep track of how many brackets are open. It is also important to keep the number of pages printed when debugging to a minimum. Hence we recommend the following rule illustrated with the following snippet of code:

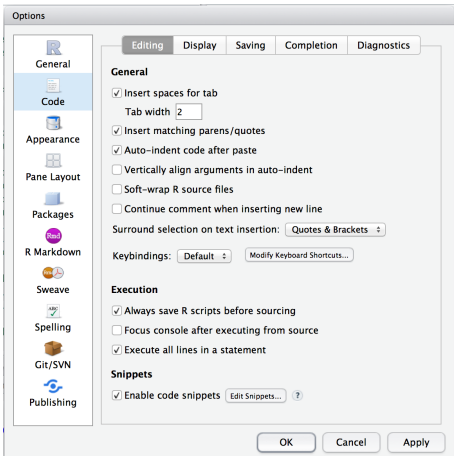
1) *Indent two spaces (even for wrapped lines) after each curly bracket ({), place first curly bracket on same line, and close (}) on a new line*

```
for (i in 1:LengthSPY){
  if(i==1){
    dfSPY$TotalReturn[i]=1
  } else {
    dfSPY$TotalReturn[i] = dfSPY$TotalReturn[i-1]*(1.0 + dfSPY$DailyReturn[i])
  }
}

PV1 = function(Maturity, Rate){
  return(exp( -(Rate/100.0) * Maturity) )
}
...
```

It is easier to set tab spaces to 2 based on selecting Tools as illustrated in the follow two screen shots.





Setting aside what this code does, the two-space indention makes the code compact and readable. Although tempting, you should use blank lines very rarely, even though you may find occasional lapses in the code presented. Hence the second rule is illustrated with the following snippet of code:

2) *Use blank lines very rarely. If you need a space, include a comment line*

```
}
# Calculate Rolling Standard Deviation using zoo
Return <- dfSPY$DailyReturn
...
```

It is helpful to place your name within your source code both to show pride in your work, as well as give contact information in case things go wrong. Hence, the third rule of illustration:

3) *Your name should be on the first line of each file's code as well as a brief description of what is contained in the file*

```
#**********************Robert Brooks*********************************
# BSMOVM Functions.R
#
#  INPUT STRUCTURE
#  BSMInputData - list of inputs with associated names; percent, not decimal
#  BSMInputData <- list(inputStockPrice, inputStrikePrice, inputInterestRate,
#    inputDividendYield, inputVolatility, inputTimeToMaturity, inputType)
#  names(BSMInputData) <- c("StockPrice", "StrikePrice", "InterestRate", "DividendYield",
#    "Volatility", "TimeToMaturity", "Type")
#
# Available functions
#  PV1(Maturity, Rate) - present value of $1
#  B = BSMInputData
#  d1(B) - value of d1
#  d2(B) - value of d2
#  n(d) - standard normal PDF, given scalar d
#  N(d) - standard normal CDF, given scalar d
```

```
#  BSMOptionValue(B) - option value, type = 1 is call, type = 2 is put
#  OptionLowerBound(B) - option lower bounds
#
```

Remember, your ultimate goal is to produce a large repository of quantitative methods that will improve your vocational abilities. Writing a brief description after the code is finished will be time consuming initially. If you have a long run perspective, however, it will save you countless hours in the future as you will not have to reinvent modules and you will quickly recall the code's purpose.

Methodological preferences

The goal is to create source code that is easily reused. For example, the particular module that computes N(d), may be used in a wide variety of R programs. Thus, the R code functions should be separated from the a particular application. Hence, the fourth rule:

4) In R, function-based code should be separated from other R code

Here is where the function is called in the R code—see 3.1 BSMOVM in the repository.

```
CallValue = BSMOptionValue(BSMInputData)
```

Here is where the function is defined and developed in the function code—see 3.1 BSMOVM in the repository.

```
# BSMOVM
BSMOptionValue = function(B){
  OptionValue = B$Type * B$StockPrice * PV1(B$TimeToMaturity, B$DividendYield) * N(B$Type * d1(B)) -
    B$Type * B$StrikePrice * PV1(B$TimeToMaturity, B$InterestRate) * N(B$Type * d2(B))
  LowerBound = B$Type * B$StockPrice * PV1(B$TimeToMaturity, B$DividendYield) -
    B$Type * B$StrikePrice * PV1(B$TimeToMaturity, B$InterestRate)
  return( max(OptionValue, LowerBound) )
}
```

The names attached to variables, functions, classes, and so forth are very important. Consistent naming will lead to much greater efficiency when maintaining a large repository of source code. The following naming conventions are adopted and illustrated below:

5) Naming conventions adopted here

a. Names of variables: lower case or both upper and lower case, err on longer name

b. Names of functions: begin with upper case for each word

```
# Test inputs
inputStockPrice = 100       # Need "input" as using variable names below
inputStrikePrice = 100
inputInterestRate = 5.0        # In percent
inputDividendYield = 0.0       # In percent
inputVolatility = 30.0         # In percent
inputTimeToMaturity = 1.0
inputType = 1 # 1 for call, -1 for put
...
CallValue = BSMOptionValue(BSMInputData)
...
```

In the code above, `CallValue` is a numeric variable. One approach to identifying your source code is to use the initials of your name. The benefit is ease of identifying who developed various files.

Appendix 1R.B: Building Your Own Code Repository

Learning objectives

- Understand one approach to managing implementation code
- Explain detailed steps for managing R code
- Emphasize the importance of organizing your quantitative ideas when coding

Introduction

The goal of this appendix is to specify one approach to managing source code. The primary goal is to maximize code reuse; ideally, you will build a module that will be used over and over again. The secondary goal is to build detailed organization in your development of quantitative ideas. Often the exercise of

implementing a new idea in a computer language will result in numerous improvements to the idea itself. Coding is also much easier if you are detail oriented and very organized.

Repository

We will make use of a central repository for R functions. Thus, multiple programs can access the same R functions. For example, consider the estimation function of the cumulative normal distribution (N(d)) used in many option valuation routines, as well as risk management calculations. The approximation method deployed here is accurate to about the ninth decimal place. Suppose this routine is used in 15 different programs that you have to support. If you discover a more accurate estimation method or a bug in the existing N(d) calculation without a repository, you will have to fix 15 different programs. With a repository, you fix one file.

Managing subdirectories for source code (recommended when first starting)

The method illustrated here is not the only one, but it is very simple. In the root directory of the operating system (or wherever you wish), create a new subdirectory. The one illustrated for this material is QFRepository.[2]

All R code with functions is placed in the QFRepository (and not in a subdirectory within QFRepository). For efficient R code management, all the R code for various specific purpose programs is contained in subdirectories within QFRepository. For example, all the specific purpose programs covered in this material is placed in subdirectories of the following subdirectory.

```
C:\QFRepository
```

[2]Note that the source code provided for this book assumes the repository is located at C:\QFRepository. If it is located anywhere else, the paths must be change.