Chapter 3. Quantitative Finance Tools R Commentary

Introduction

In this chapter, we present a series of useful R modules that cover fundamental tools used often in quantitative finance applications.

Each of these R programs is presented as a separate module contained in a subdirectory. The goal is to review essential coding techniques that will prove useful in building quantitative finance. We recommend that you work through each module with an effort to understand both the financial concepts as well as illustrations of R coding.

First, we address some R code that will show up in almost all our sample code.

Basic housekeeping

At the top of each file, we will provide the file name as illustrated below. Most of the files that run modules will end with `Test.R` indicating that it is simply a test file that you are free to subsequently modify.

```
# File Name Test.R
```

The next several lines are designed to clean up prior runs of the program. These lines are helpful when debugging as residual errors may often remain even when you have fixed the code. Alternatively, it may appear the code is running correctly but only because variables that have now been removed are still in the environment. The next time you clean out the environment, the program would crash.

```
rm(list = ls())  # Take out the Environment "trash"
cat("\014") # Clear Console, making error checking easier.
while (!is.null(dev.list()))  dev.off() # Clear old plots
```

It is helpful to format the font of graphs to be consistent with other documents you are producing that may be independent of R.

```
par(family = 'Times New Roman') # Globally set fonts for graphs
```

The R programming language is powerful in its ability to access multiple packages and even design packages of your own. Most quantitative finance packages will use several available packages. The syntax below automatically installs packages even if they have not already been installed as well as make them accessible to this program. In this illustration, we assume there is a need for three packages, date, optimx, and openxlsx. At this point, all you need to know is that we need access to these packages, not what they provide. As you can see, it is helpful to leave comments reminding you and others why the packages are being installed.

```
# Libraries
#  date - functions for handling dates
#  optimx - general purpose optimization
#  openxlsx - manipulate spreadsheet files
Packages <- c("date", "optimx", "openxlsx")
if(length(setdiff(Packages, rownames(installed.packages()))) > 0) {
  install.packages(setdiff(Packages, rownames(installed.packages())))
} # Make sure libraries are installed on this computer
lapply(Packages, library, character.only=TRUE) # Load and attach libraries
rm(Packages)
```

It is important to note that because R is open source some packages may be inappropriate for your objectives. Also, when upgrading to a new version of R, certain older packages may not function correctly. If you maintain software over any length of time, you quickly realize that computer programming languages and the operating systems in which they reside are in constant flux.

Module 3.1: Managing the Calendar

See *Calendar and Compounding Test.R*. There is also a supplemental file exploring ancient day counting as well as a few other items in *Ancient Calendar Test.R* and an exploration of day counting with functions from the package jrvFinance is provided in *Calendar Test.R*. Although not discussed here, see *Ancient Calendar Test.R* for dealing with ancient historical dates.

Calendar and Compounding Test.R (Selected Excerpts and Outputs)

The date package contains several useful functions. The function `mdy.date()` converts calendar integers into a Julian number:

```
JulianStartDate = mdy.date(InputStartMonth,InputStartDay,InputStartYear,nineteen = FALSE)
> JulianStartDate
[1] 1Jan2020 # Printed as a date
> as.integer(JulianStartDate)
[1] 21915 # Stored as an integer
> class(JulianStartDate)
[1] "date"
```

The function date.mdy() converts a Julian number into a vector containing the calendar date:

```
StartDate = date.mdy(JulianStartDate)
> StartDate
$month
[1] 1

$day
[1] 1

$year
[1] 2020
```

The function paste() is useful for producing appropriately formatted data:

```
TVACT360 = paste("$",format(TVACT360, big.mark=","),sep="")
…
TVACT360; TVACT365; TVDifference
## [1] "$4,414,890"
## [1] "$4,325,988"
## [1] "$88,901.4"
```

Calendar Test.R (Selected Excerpts and Outputs)

This program illustrates some of the functionality of the jrvFinance package. The following snippet illustrates different issues related to day counting.

```
# Computing fraction of year
InputStartMonth <- 6
InputStartDay <- 1
InputStartYear <- 2020
JulianStartDate = mdy.date(InputStartMonth,InputStartDay,InputStartYear,nineteen = FALSE)
InputEndMonth <- 6
InputEndDay <- 1
InputEndYear <- 2021
JulianEndDate = mdy.date(InputEndMonth, InputEndDay, InputEndYear, nineteen = FALSE)
d1 <- JulianStartDate
d2 <- JulianEndDate
r1 <- mdy.date(1, 1, InputStartYear, nineteen = FALSE) # Beginning of year, not June
r2 <- mdy.date(1, 1, InputEndYear, nineteen = FALSE)
# 2020 is a leap year, see jrvFinance package help
F1 <- yearFraction(d1, d2, r1, r2, freq = 2, convention = "30/360") # 360/360
F2 <- yearFraction(d1, d2, r1, r2, freq = 1, convention = "ACT/ACT") # 366/365
F3 <- yearFraction(d1, d2, r1, r2, freq = 2, convention = "ACT/360") # 366/360
F4 <- yearFraction(d1, d2, r1, r2, freq = 2, convention = "30/360E") # 360/360 (semi ?)
F5 <- yearFraction(d1, d2, r1, r2, freq = 12, convention = "ACT/ACT") # Monthly
F6 <- daycount.actual(d1, d2, variant = "bond")
F7 <- daycount.30.360(d1, d2, variant = "US")
F1; F2; F3; F4; F5; F6; F7

> F1; F2; F3; F4; F5; F6; F7
[1] 1
[1] 0.9972678
[1] 1.013889
[1] 1
[1] 0.08310565
[1] 365
```

```
[1] 360
```
Often you need the current date on the computer system.
```
# Find today on the system
?'Sys.Date'
TodaysDate = Sys.Date() # But in unusable format
TodaysYear <- as.integer(format(TodaysDate, "%Y")) # year -- upper case
TodaysMonth <- as.integer(format(TodaysDate, "%m")) # month -- note case sensitive
TodaysDay <- as.integer(format(TodaysDate, "%d")) # day -- note case sensitive
JulianTodaysDate = mdy.date(TodaysMonth, TodaysDay, TodaysYear, nineteen = FALSE)
TodaysDate; TodaysYear; TodaysMonth; TodaysDay

[1] "2019-09-26"
[1] 2019
[1] 9
[1] 26
# Holidays: tis: Time Indexes and Time Indexed Series
if("tis" %in% rownames(installed.packages())==FALSE)install.packages("tis")
library(tis)
x <- 20190101
x <- mdy.date(1, 1, 2019, nineteen = FALSE)
```
One challenging problem is determining whether a particular date is a business holiday or finding particular dates for holidays.
```
> x <- mdy.date(1, 1, 2021, nineteen = FALSE)
> # See package tis: Time Indexes and Time Indexed Series
> board <- FALSE # Presidential inauguration is not a holiday
> nextBusinessDay(x, holidays = NULL, goodFriday = F, board = F, inaug = board)
[1] 20210104
class: ti
> previousBusinessDay(x, holidays = NULL, goodFriday = F, board = F, inaug = board)
[1] 20201231
class: ti
> isHoliday(x, goodFriday = TRUE, board = FALSE, inaug = board, businessOnly = TRUE)
[1] TRUE
> isBusinessDay(x)
[1] FALSE
> isGoodFriday(x)
[1] FALSE
> isEaster(x)
[1] FALSE
> years <- 2021
> holidays(years, goodFriday = F, board = F, inaug = board, businessOnly = T)
    NewYears       MLKing   GWBirthday     Memorial Independence        Labor
    20210101     20210118     20210215     20210531     20210705     20210906
    Columbus     Veterans Thanksgiving
    20211011     20211111     20211125
> federalHolidays(years, board = F, businessOnly = T)
    NewYears       MLKing   GWBirthday     Memorial Independence        Labor
    20210101     20210118     20210215     20210531     20210705     20210906
    Columbus     Veterans Thanksgiving
    20211011     20211111     20211125
> goodFriday(years)
GoodFriday
  20210402
> easter(years)
[1] 20210404
> inaugurationDay(years)
Inauguration
    20210120
```
The last bit of code illustrates how to build your own function. Although not robust, you should get the general idea of how to build a function is a separate file—see *Adjust Date.R*.
```
> source('Adjust Date.R')
> TestMonth <- 9
> TestDay <- 26 # Normal Tuesday
```

```
> TestYear <- 2019
> Convention <- "MBP" # Modified Business Following or MBP (Preceeding)
> AdjustDate(TestMonth, TestDay, TestYear, Convention)
[1] 26Sep2019
> TestDay <- 29 # Sunday
> AdjustDate(TestMonth, TestDay, TestYear, Convention)
[1] 27Sep2019
> Convention <- "MBF"
> AdjustDate(TestMonth, TestDay, TestYear, Convention)
[1] 30Sep2019
```

You now have a good introduction to the challenges of managing dates.

Module 3.2. Cumulative Normal Distribution Function and its Inverse

See program *CDF and Inverse CDF Test.R*.

3.2 CDF and Inverse CDF Test.R (Selected Excerpts and Output)

The function rnorm() generates a vector of normally distributed random numbers:

```
SampleDraw = rnorm(NumberOfObservations, EstimatedMean, EstimatedStandardDeviation)
```

The functions dnorm(), pnorm(), and qnorm() are all based on the normal distribution:

```
# Probability density function
PDF = dnorm(QuantileValue, EstimatedMean, EstimatedStandardDeviation, log = FALSE)
# Cumulative distribution function
CDF = pnorm(QuantileValue, EstimatedMean, EstimatedStandardDeviation,
  lower.tail = TRUE, log.p = FALSE)
# Quantile function
QF = qnorm(CDF, EstimatedMean, EstimatedStandardDeviation,
  lower.tail = TRUE, log.p = FALSE)
Difference = QuantileValue - QF
QuantileValue; PDF; CDF; QF
## [1] 0
## [1] 0.3989423
## [1] 0.5
## [1] 0
```

This program provides an illustration of the lack of precision when dealing with numeric calculations.

```
NumberOfObservations = 201
D <- c(1:NumberOfObservations)   # Quantile input
N <- c(1:NumberOfObservations)   # CDF input
N1 <- c(1:NumberOfObservations)  # First estimate of CDF
D1 <- c(1:NumberOfObservations)  # First estimate of quantile
DE <- c(1:NumberOfObservations)  # Quantile estimation error
N2 <- c(1:NumberOfObservations)  # Second estimate of CDF
D2 <- c(1:NumberOfObservations)  # Second estimate of quantile
NE <- c(1:NumberOfObservations)  # CDF estimation error
n <- c(1:NumberOfObservations)   # Estimated PDF
EstimatedMean = 0 # Mean
EstimatedStandardDeviation = 1 # Standard deviation
```

Estimation iteration seeking to introduce machine error.

```
for(i in 1:NumberOfObservations){
  D[i] <- -5.0 + 0.05*as.double((i-1))
  D[i] <- D[i] * EstimatedStandardDeviation + EstimatedMean
  N1[i] <- pnorm(D[i], EstimatedMean, EstimatedStandardDeviation)
  n[i] <- dnorm(D[i], EstimatedMean, EstimatedStandardDeviation)
  D1[i] <- qnorm(N1[i], EstimatedMean, EstimatedStandardDeviation)
  DE[i] <- D1[i] - D[i]
  N[i] <- -0.005 + 0.005*as.double(i)
  D2[i] <- qnorm(N[i], EstimatedMean, EstimatedStandardDeviation)
  N2[i] <- pnorm(D2[i], EstimatedMean, EstimatedStandardDeviation)
  NE[i] <- N2[i] - N[i]
}
MaxNError <- max(abs(NE), na.rm=TRUE)
```

```
MaxDError <- max(abs(DE), na.rm=TRUE)
MaxNError; MaxDError
# Standard normal probability density function plot
MaxValue = max(n, na.rm=TRUE); MinValue = min(n, na.rm=TRUE); MaxValue; MinValue
ylim1 = c(1:2); ylim1[1] = MinValue; ylim1[2] = MaxValue
MaxValue = max(D, na.rm=TRUE); MinValue = min(D ,na.rm=TRUE); MaxValue; MinValue
xlim1 = c(1:2); xlim1[1] = MinValue; xlim1[2] = MaxValue
xTitle = "D"; yTitle = "n"
mTitle = "Standard Normal Probability Density Function"
plot(D, n, type = "l", main = mTitle, xlab = xTitle, ylab = yTitle, xlim = xlim1,
  ylim = ylim1, pch = 1, cex = 0.5)
# Standard normal cumulative distribution function plot
MaxValue = max(N1, na.rm=TRUE); MinValue = min(N1, na.rm=TRUE); MaxValue; MinValue
ylim1 = c(1:2); ylim1[1] = MinValue; ylim1[2] = MaxValue
MaxValue = max(D, na.rm=TRUE); MinValue = min(D ,na.rm=TRUE); MaxValue; MinValue
xlim1 = c(1:2); xlim1[1] = MinValue; xlim1[2] = MaxValue
xTitle = "D"; yTitle = "N"
mTitle = "Standard Normal Cumulative Distribution Function"
plot(D, N1, type = "l", main = mTitle, xlab = xTitle, ylab = yTitle, xlim = xlim1,
  ylim = ylim1, pch = 1, cex = 0.5)
# Estimation error in D
MaxValue = max(DE, na.rm=TRUE); MinValue = min(DE, na.rm=TRUE); MaxValue; MinValue
ylim1 = c(1:2); ylim1[1] = MinValue; ylim1[2] = MaxValue
MaxValue = max(D, na.rm=TRUE); MinValue = min(D ,na.rm=TRUE); MaxValue; MinValue
xlim1 = c(1:2); xlim1[1] = MinValue; xlim1[2] = MaxValue
xTitle = "D"; yTitle = "DE"
mTitle = "Estimation Error in D"
plot(D, DE, type = "p", main = mTitle, xlab = xTitle, ylab = yTitle, xlim = xlim1,
  ylim = ylim1, pch = 1, cex = 0.5)
# Estimation error in N
MaxValue = max(NE, na.rm=TRUE); MinValue = min(NE, na.rm=TRUE); MaxValue; MinValue
ylim1 = c(1:2); ylim1[1] = MinValue; ylim1[2] = MaxValue
MaxValue = max(N, na.rm=TRUE); MinValue = min(N ,na.rm=TRUE); MaxValue; MinValue
xlim1 = c(1:2); xlim1[1] = MinValue; xlim1[2] = MaxValue
xTitle = "N"; yTitle = "NE"
mTitle = "Estimation Error in N"
plot(N, NE, type = "p", main = mTitle, xlab = xTitle, ylab = yTitle, xlim = xlim1,
  ylim = ylim1, pch = 2, cex = 0.75)  # pch = 2 is triangle, cex is size
```

Module 3.3. Univariate Random Numbers

See *Univariate RNG Test.R*. There is also a supplemental file exploring large samples in *Univariate RNG Large Sample Test.R*.

Univariate RNG Test.R (Selected Excerpts and Output)

The following program explores various aspects of random number generation. The following code generates a vector of 100 uniformly distributed real numbers between 0 and 1 and calculates the mean and standard deviation (appropriately formatted for display).

```
y = runif(NumberOfObservations, LowerBound, UpperBound)
yMean = mean(y)
…
yMean <- format(yMean, trim = FALSE, digits = NULL, nsmall = 4, justify = "right")
yStdDev = sd(y)
yStdDev <- format(yStdDev, trim = FALSE, digits = NULL, nsmall = 4, justify = "right")
```

The following generates a vector of uniformly distributed *integer* numbers, adjusts for upper and lower bounds, and calculates the mean and standard deviation (appropriately formatted for display).

```
y = sample.int(NumberOfIntegers, size=NumberOfObservations, replace = TRUE, prob = NULL)
y = y + LowerBound - 1 # Adjust integer vector for lower bound, lower bound not possible
yMean = mean(y)
yMean <- format(yMean, trim = FALSE, digits = NULL, nsmall = 4, justify = "right")
yStdDev = sd(y)
yStdDev <- format(yStdDev, trim = FALSE, digits = NULL, nsmall = 4, justify = "right")
```

The following code generates a vector of uniformly distributed real numbers between a lower and upper bound and calculates the mean and standard deviation (appropriately formatted for display).

```
y = runif(NumberOfObservations, LowerBound, UpperBound)
yMean = mean(y)
yMean <- format(yMean, trim = FALSE, digits = NULL, nsmall = 4, justify = "right")
yStdDev = sd(y)
yStdDev <- format(yStdDev, trim = FALSE, digits = NULL, nsmall = 4, justify = "right")
```

The following code generates a vector of *normally* distributed numbers with mean 0 and standard deviation 1. The sample mean and standard deviation are calculated illustrating sample error.

```
PMean = 0
PStandardDeviation = 1
y = rnorm(NumberOfObservations, PMean, PStandardDeviation)
yMean = mean(y)
…
yStdDev = sd(y)
```

The following code generates a vector of *normally* distributed numbers with a population mean 10 and population standard deviation 30. The sample mean and standard deviation are calculated illustrating sample error.

```
PMean = 10.0
PStandardDeviation = 30.0
NumberOfObservations = 100
…
y = rnorm(NumberOfObservations, PMean, PStandardDeviation)
```

The following code generates a binary vector (Bernoulli distribution) based on a desired likelihood. The sample mean and standard deviation are calculated illustrating sample error.

```
DesiredLikelihood = 0.25 # Probability of success
NumberOfObservations = 100
LowerBound = 0 # Minimum
UpperBound = 1 # Maximum
x = c(1:NumberOfObservations)
y = c(1:NumberOfObservations)
y = runif(NumberOfObservations, LowerBound, UpperBound)
z = c(1:NumberOfObservations)
for(i in 1:NumberOfObservations){
  if(y[i] < DesiredLikelihood) y[i] = 1
  else y[i] = 0
  z[i] = DesiredLikelihood
}
yMean = mean(y)
yStdDev = sd(y)
```

Univariate RNG Large Sample Test.R (Selected Excerpts and Output)

The following program explores various aspects of random number generation when the sample size is very large. A data.frame is defined (FRMDSTATS) and output is returned from functions within this data.frame. The first illustration is a uniformly distributed real vector. Note simulation time is recorded.

```
FRMDSTATS <- data.frame(SampleMean, SampleStandardDeviation, PopulationMean,
  PopulationStandardDeviation, SimulationTimeInSeconds)
#
# Example 1: Uniform Real
#
FRMUniformReal <- function(FRMDSTATS, SampleSize, RealLowerBound, UpperBound) {
# Return CPU (Central Processing Unit) times that the expression () used
  Time <- system.time(Draw<-runif(SampleSize, LowerBound, UpperBound), gcFirst = TRUE)
  FRMDSTATS[1] = mean(Draw, na.rm = TRUE) # Sample mean
  FRMDSTATS[2] = sd(Draw, na.rm = TRUE) # Sample standard deviation
  FRMDSTATS[3] = (LowerBound + UpperBound)/2.0 # Population mean
  FRMDSTATS[4] = (((UpperBound - LowerBound)^2)/12.0)^0.5 # Pop. standard deviation
  FRMDSTATS[5] = Time[3] # Simulation Time In Seconds
  return(FRMDSTATS)
}
# Test the function
```

```
SampleSize = 1000000
UpperBound = 100
LowerBound = -100
USS <- FRMUniformReal(FRMDSTATS, SampleSize, RealLowerBound, UpperBound)
USS
##   SampleMean SampleStandardDeviation PopulationMean
## 1     0.1961                  57.73              0
##   PopulationStandardDeviation SimulationTimeInSeconds
## 1                       57.74                  0.029
```

The code below generates is a uniformly distributed integer vector.

```
UpperBound = 100
LowerBound = 0
Draw <- runif(SampleSize,0,1)
for (i in 1:SampleSize){
  Draw[i] = as.integer(Draw[i] * (UpperBound - LowerBound + 1)) + LowerBound
}
SampleMean = mean(Draw, na.rm = TRUE)
SampleStandardDeviation = sd(Draw, na.rm = TRUE)
PopulationMean = (LowerBound + UpperBound)/2.0
PopulationStandardDeviation = (((UpperBound - LowerBound + 1)^2 - 1.0)/12.0)^0.5
Time <- system.time(runif(SampleSize, LowerBound, UpperBound), gcFirst = TRUE)
SimulationTimeInSeconds = Time[3] # Simulation Time In Seconds
USS <- FRMDSTATS
USS[1] = PopulationMean
USS[2] = PopulationStandardDeviation
USS[3] = SampleMean
USS[4] = SampleStandardDeviation
USS[5] = SimulationTimeInSeconds
USS
##   SampleMean SampleStandardDeviation PopulationMean
## 1         50                  29.15          50.03
##   PopulationStandardDeviation SimulationTimeInSeconds
## 1                       29.14                  0.031
```

The code below generates is a vector of binary values (0 or 1) to estimate likelihood.

```
FRMLikelihood <- function(FRMDSTATS, SampleSize, Likelihood) {
  Time <- system.time(Draw <- runif(SampleSize, 0, 1), gcFirst = TRUE)
  for(i in 1:SampleSize){
    if(Draw[i] <= DesiredLikelihood) Draw[i] = 1.0
    else Draw[i] = 0.0
  }
  FRMDSTATS[1] = mean(Draw, na.rm = TRUE) # Sample mean
  FRMDSTATS[2] = sd(Draw, na.rm = TRUE) # Sample standard deviation
  FRMDSTATS[3] = DesiredLikelihood # Population mean
  FRMDSTATS[4] = sqrt(DesiredLikelihood-(DesiredLikelihood^2.0)) # Population standard
deviation
  FRMDSTATS[5] = Time[3] # Simulation Time In Seconds
  return(FRMDSTATS)
}
# Test the function
SampleSize = 1000000
DesiredLikelihood = 0.5
USS <- FRMLikelihood(FRMDSTATS, SampleSize, Likelihood)
USS
##   SampleMean SampleStandardDeviation PopulationMean
## 1     0.5004                     0.5            0.5
##   PopulationStandardDeviation SimulationTimeInSeconds
## 1                         0.5                  0.029
```

The code below generates is a normally distributed vector.

```
FRMNormal <- function(FRMDSTATS, SampleSize, NMean, NSD) {
  Time <- system.time(Draw <- rnorm(SampleSize, NMean, NSD), gcFirst = TRUE)
  FRMDSTATS[1] = mean(Draw, na.rm = TRUE) # Sample mean
  FRMDSTATS[2] = sd(Draw, na.rm = TRUE) # Sample standard deviation
  FRMDSTATS[3] = NMean # Population mean
  FRMDSTATS[4] = NSD # Population standard deviation
```

```
  FRMDSTATS[5] = Time[3] # Simulation Time In Seconds
  return(FRMDSTATS)
}
# Test the function
SampleSize = 10000000
NMean = 15.0
NSD = 30.0
USS <- FRMNormal(FRMDSTATS, SampleSize, NMean, NSD)
USS
##   SampleMean SampleStandardDeviation PopulationMean
## 1      15.02                  30.01            15
##   PopulationStandardDeviation SimulationTimeInSeconds
## 1                          30                  0.857
```

Module 3.4. The LSC Model: Curve Fitting Using Linear Regression

See *Regression Test.R* and *LSC Curve Fitting Test.R*. There is also an input data file LSCInputData.dat. See also *LSC Coefficient Sensitivity Test.R* and *LSC Regression Independent Variables Test.R*.

LSC Curve Fitting Test.R (Selected Excerpts and Output)

This code is an implementation of the LSC model. We first need selected global parameters.

```
> # Inputs
> NumberOfMaturities <- 9
> NumberOfFactors <- 3
> N <- NumberOfFactors - 2
> Tau <- c(1:N)
> Tau[1] <- 2.0
> NumberOfDates <- 5
```

Need to fill the Maturity and Dates vectors.

```
> Maturity <- c(1:NumberOfMaturities)
> Maturity[1] <- LSCData$V2[1]
> Maturity[2] <- LSCData$V3[1]
> Maturity[3] <- LSCData$V4[1]
> Maturity[4] <- LSCData$V5[1]
> Maturity[5] <- LSCData$V6[1]
> Maturity[6] <- LSCData$V7[1]
> Maturity[7] <- LSCData$V8[1]
> Maturity[8] <- LSCData$V9[1]
> Maturity[9] <- LSCData$V10[1]
> Dates <- c(1:NumberOfDates)
> for(i in 1:NumberOfDates){
+   Dates[i] = as.character(LSCData$V1[i+1])
+ }
```

Place input rates in easy to understand matrix.

```
> # Place input rates in matrix
> Rates <- matrix(nrow = NumberOfDates, ncol = NumberOfMaturities)
> for(i in 1:NumberOfDates){
+   Rates[i,1] <- LSCData$V2[i+1]
+   Rates[i,2] <- LSCData$V3[i+1]
+   Rates[i,3] <- LSCData$V4[i+1]
+   Rates[i,4] <- LSCData$V5[i+1]
+   Rates[i,5] <- LSCData$V6[i+1]
+   Rates[i,6] <- LSCData$V7[i+1]
+   Rates[i,7] <- LSCData$V8[i+1]
+   Rates[i,8] <- LSCData$V9[i+1]
+   Rates[i,9] <- LSCData$V10[i+1]
+ }
```

Compute the appropriate factor values.

```
> Factors <- matrix(nrow = NumberOfFactors - 1, ncol = NumberOfMaturities)
> Factors # Note Factors is filled with NAs
     [,1] [,2] [,3] [,4] [,5] [,6] [,7] [,8] [,9]
```

```
[1,]   NA   NA   NA   NA   NA   NA   NA   NA   NA
[2,]   NA   NA   NA   NA   NA   NA   NA   NA   NA
> for (j in 1:NumberOfFactors-1) {
+   for (i in 1:NumberOfMaturities) {
+     if (j == 1) Factors[j,i] = (1.0 - exp(-Maturity[i]/Tau[j]))/(Maturity[i]/Tau[j])
+     else Factors[j, i] = (1.0 - exp(-Maturity[i]/Tau[j-1]))/(Maturity[i]/Tau[j-1]) -
+       exp(-Maturity[i]/Tau[j-1])
+   }
+ }
```

Compute LSC parameters and place in vectors for future use.

```
> Intercept <- c(1:NumberOfDates) # Vectors for output
> Slope <- c(1:NumberOfDates)
> Curvature <- c(1:NumberOfDates)
> # OLS regressions for each date
> for (i in 1:NumberOfDates){ # Cross-section analysis of each date
+   LSC <- lm(formula = Rates[i,]~Factors[1,]+Factors[2,])
+   Betas <- LSC$coefficients # Make clear grabbing beta coefficients
+   Intercept[i] <- Betas[1]
+   Slope[i] <- Betas[2]
+   Curvature[i] <- Betas[3]
+ }
```

Compute fitted rates based on LSC output.

```
> # Work on fitted data for plots
> FittedRates <- matrix(nrow = NumberOfDates, ncol = NumberOfMaturities)
> for (i in 1:NumberOfDates) { # Fitted values for each date
+   for (j in 1:NumberOfMaturities){
+     FittedRates[i,j] = Intercept[i] + Slope[i] *
+       ((1.0 - exp(-Maturity[j]/Tau[1]))/(Maturity[j]/Tau[1])) +
+       Curvature[i]*((1.0 - exp(-Maturity[j]/Tau[1]))/(Maturity[j]/Tau[1]) -
+       exp(-Maturity[j]/Tau[1]))
+   }
+ }
```

Format and generate a separate plot for each date.

```
> # Plots
> x <- Maturity
> y1 <- Rates        # Helps to check output
> y2 <- FittedRates
> MinXValue = 0; MaxXValue = max(x)
> xlim1 = c(1:2); xlim1[1] = MinXValue; xlim1[2] = MaxXValue
> MinYValue = min(y1, y2); MaxYValue = max(y1, y2)
> ylim1 = c(1:2); ylim1[1] = MinYValue; ylim1[2] = MaxYValue
> legtxt = c("Actual Rates","Fitted Rates")
> mTitle = "Swap Rates"
> xTitle = "Maturity"
> yTitle = "Rates"
> lTitle <- "Parameter"
> legtxt = c("Actual","Fitted")
> for (i in 1:NumberOfDates) {
+   plot(x, y1[i,], type="b", main=mTitle, sub=Dates[i], xlab=xTitle,
+     ylab=yTitle, col="blue", xlim = xlim1, ylim = ylim1, pch = 1, cex = 1.0)
+   lines(Maturity, FittedRates[i,], type="b", col="red", xlim = xlim1,
+     ylim = ylim1, pch = 2, cex = 1.0)
+   legend("topleft", legtxt, cex = 1.0, lwd = c(1, 1), lty = c(1, 1),
+     col = c("blue","red"), pch = c(1,2), bty = "n", title = lTitle)
+ }
```

Module 3.5. Sorting Data

See *Sorting Test.R*. There are also two input data files, SortingData.dat and SO.PRN. SortingData.dat contains 100 real numbers and SO.PRN contains a variety of variables related to Southern Company's daily stock prices.

Sorting Data Test.R (Selected Excerpts and Output)

This code illustrates several uses of sorting data. First, the data.table library has a fast inputting function, fread().

```
SortData <- fread("SortingData.dat", header = FALSE, sep = " ")
head(SortData,5)
##       V1
## 1: 29.82
## 2: 71.51
## 3:  3.30
## 4: 87.44
## 5: 53.42
tail(SortData,5)
##       V1
## 1: 25.01
## 2: 82.09
## 3: 89.67
## 4: 57.08
## 5: 38.27
```

The base library has a simple, single vector sorting function, sort().

```
DataVectorSorted <- sort(DataVector, na.last = NA, method = "quick")
```

The comma delimited data contained in SO.PRN is read in with read.delim or fread. The contents of this file is identified with sapply(). We may want to examine just subsets of the entire dataset, hence we identify upper and lower bounds.

```
LowerDate <- 19001231 # No bounds
UpperDate <- 25001231
# LowerDate <- 20061231 # Pre and post crisis
# UpperDate <- 20201231
# SortData <- read.delim("SO.PRN", header = TRUE, sep = ",")
Company <- "Southern Company" # Used in plot titles, change is new data used
SortData <- fread("SO.PRN", sep = ",") # Input variables faster, data.table
sapply(SortData, class) # Way too much stuff
```

After a bit of data manipulation, the file is sorted based on first differences of the stock price (vector FD within SortData). The resulting sort is illustrated with a histogram overlaid with the sample parameters applied to the normal distribution.

```
SortDataFD <- SortData[order(SortData$FD), ]
…
y <- SortDataFD$FD
…
hist(y, main=mTitle, breaks=50, freq=FALSE, col="red", labels = FALSE,
  sub=sTitle, xlab=xTitle, ylab=yTitle, plot=TRUE, axes=TRUE, density=10)
curve(dnorm(x, yMean, yStdDev), add=TRUE, col="darkblue", lwd=4)
```

To illustrate the distributional implications of first differences when compared to percentage changes, the dataset is sorted by rate of return and the results are graphically illustrated.

```
SortDataR <- SortData[order(SortData$R), ]
…
y <- SortDataR$R*100 # Express as percent
…
hist(y, main=mTitle, breaks=50, freq=FALSE, col="red", labels = FALSE,
  sub=sTitle, xlab=xTitle, ylab=yTitle, plot=TRUE, axes=TRUE, density=10)
curve(dnorm(x, yMean, yStdDev), add=TRUE, col="darkblue", lwd=4)
```

Next we illustrate multiple sorting by year, then month, and then day. The closing price, first differences, and rates of return are plotted. Note in the last couple of plots, the y-axis is highly limited so as to illustrate the influence of decimal prices.

```
SortDataDate <- SortData[order(SortData$Year, SortData$Month, SortData$Day), ]
```

To gain better understanding of the unstable nature of financial data, run this program with different five year subperiods.

Module 3.6. Embedded Parameters

See *Embedded Functions Test.R*.

Embedded Functions Test.R (Selected Excerpts and Output)

This code illustrates several functions as well as the optimize() function. After inputting specific values of bond parameters, a function is created to calculate the bond value (with accrued interest).

```
FRMBondValue <- function(tempYieldToMaturity, tempCouponRate, tempParValue,
  tempYearsToMaturity){
  PV = 0.0 # Present value variable
  for (i in 1:tempYearsToMaturity){
    PV = PV + ((tempCouponRate/100.0)*tempParValue) /
      ((1.0 + (tempYieldToMaturity/100.0))^i)
  }
  return(PV + tempParValue /
    ((1.0 + (tempYieldToMaturity/100.0))^tempYearsToMaturity))
}
# Test the function
BondValue = FRMBondValue(YieldToMaturity, CouponRate, ParValue,
  YearsToMaturity)
```

To solve for yield to maturity, the goal is the find the yield to maturity such that the model value equals the market price as illustrated in the following function. This function receives a temporary yield to maturity and computes the difference between the actual market price and the model value.

```
FRMPriceDifference <- function(tempYieldToMaturity, tempCouponRate,
  tempParValue, tempYearsToMaturity, tempActualPrice){
  PV = 0.0 # Present value variable
  for (i in 1:tempYearsToMaturity){
    PV = PV + ((tempCouponRate/100.0)*tempParValue) /
      ((1.0 + (tempYieldToMaturity/100.0))^i)
  }
  return(abs(tempActualPrice - (PV + tempParValue
    / ((1.0 + (tempYieldToMaturity/100.0))^tempYearsToMaturity))))
}
# Test the function -- should be 0 if using BondValue from calculation above
BondValue = 90
TestDifference = FRMPriceDifference(YieldToMaturity, CouponRate, ParValue,
  YearsToMaturity, BondValue)
```

This difference function is used to solve for the yield to maturity using the optimize routine. The program concludes with an overlay plot of model bond values for three different bonds.

```
solution = optimize(FRMPriceDifference, tempCouponRate = CouponRate,
  tempParValue = ParValue, tempYearsToMaturity = YearsToMaturity,
  tempActualPrice = ActualPrice, interval = c(0,1000),
  tol = .Machine$double.eps^0.25)
solution
# Print YieldToMaturity that equates actual and model bond prices
BondYieldToMaturity = solution$minimum
BondYieldToMaturity
# Data for plots
YieldToMaturity = c(1:NumberOfObservations)
BondValue30 <- c(1:NumberOfObservations)
BondValue15 <- c(1:NumberOfObservations)
BondValue1 <- c(1:NumberOfObservations)
YieldToMaturityEst30 <- c(1:NumberOfObservations)
YieldToMaturityEst15 <- c(1:NumberOfObservations)
YieldToMaturityEst1 <- c(1:NumberOfObservations)
YieldToMaturityError30 <- c(1:NumberOfObservations)
YieldToMaturityError15 <- c(1:NumberOfObservations)
YieldToMaturityError1 <- c(1:NumberOfObservations)
```

```
for(i in 1:NumberOfObservations){
  YieldToMaturity[i] <- as.double(YieldLowerBound + (i-1)*YieldStepSize)
  YearsToMaturity = 1
```
First, take yield to maturity and compute bond value.
```
  BondValue1[i] = FRMBondValue(YieldToMaturity[i], CouponRate, ParValue,
    YearsToMaturity)
```
Second, take bond value and solve for yield to maturity.
```
 solution = optimize(FRMPriceDifferenceWFunction, tempCouponRate = CouponRate,
    tempParValue = ParValue, tempYearsToMaturity = YearsToMaturity,
    tempActualPrice = BondValue1[i], interval = c(0,1000),
    tol = .Machine$double.eps^0.25)
  YieldToMaturityEst1[i] = solution$minimum
```
Third, compare original yield to maturity with the estimate and appraise errors.
```
 YieldToMaturityError1[i] = YieldToMaturityEst1[i] - YieldToMaturity[i]
  YearsToMaturity = 15
  BondValue15[i] = FRMBondValue(YieldToMaturity[i], CouponRate, ParValue,
    YearsToMaturity)
  solution = optimize(FRMPriceDifferenceWFunction, tempCouponRate = CouponRate,
    tempParValue = ParValue, tempYearsToMaturity = YearsToMaturity,
    tempActualPrice = BondValue15[i], interval = c(0,1000),
    tol = .Machine$double.eps^0.25)
  YieldToMaturityEst15[i] = solution$minimum
  YieldToMaturityError15[i] = YieldToMaturityEst15[i] - YieldToMaturity[i]
  YearsToMaturity = 30
  BondValue30[i] = FRMBondValue(YieldToMaturity[i], CouponRate, ParValue,
    YearsToMaturity)
  solution = optimize(FRMPriceDifferenceWFunction, tempCouponRate = CouponRate,
    tempParValue = ParValue, tempYearsToMaturity = YearsToMaturity,
    tempActualPrice = BondValue30[i], interval = c(0,1000),
    tol = .Machine$double.eps^0.25)
  YieldToMaturityEst30[i] = solution$minimum
  YieldToMaturityError30[i] = YieldToMaturityEst30[i] - YieldToMaturity[i]
}
# Note: Maximum error is all the same (when ytm is zero, see graph below)
max(abs(YieldToMaturityError1))
max(abs(YieldToMaturityError15))
max(abs(YieldToMaturityError30))
# Simple plot
MaxXValue = max(YieldToMaturity)
MinXValue = min(YieldToMaturity)
xlim1 = c(1:2); xlim1[1] = MinXValue; xlim1[2] = MaxXValue
MaxYValue = max(BondValue1, BondValue15, BondValue30)
MinYValue = min(BondValue1, BondValue15, BondValue30)
ylim1 = c(1:2); ylim1[1] = MinYValue; ylim1[2] = MaxYValue
legtxt = c("30 Year","15 Year","1 Year")
Title1 = "Bond Price-Yield Relation"
xTitle = "Yield To Maturity"
yTitle = "Bond Value"
# Plot footers
TC = paste0('Coupon = ', CouponRate, '%')
TPar = paste0(', Par = $', ParValue)
sTitle = paste0(TC, TPar)
lTitle = "Maturity"
plot(YieldToMaturity, BondValue30, type="b", main=Title1,
  sub=sTitle, xlab=xTitle, ylab=yTitle, col="black", xlim = xlim1,
  ylim = ylim1, pch = 1, cex = 0.5, lty = 1)
lines(YieldToMaturity,BondValue15, type="b", col="black", xlim = xlim1,
  ylim = ylim1, pch = 2, cex = 0.5, lty = 2)
lines(YieldToMaturity,BondValue1, type="b", col="black", xlim = xlim1,
  ylim = ylim1, pch = 3, cex = 0.5, lty = 3)
legend("topright", legtxt, cex = 0.75, lwd = c(1,1,1), lty = c(1,2,3),
  col = c("black","black","black"), pch = c(1, 2, 3), bty = "n",
  title = lTitle)
# Simple plot of errors
```

```
MaxXValue = max(YieldToMaturity)
MinXValue = min(YieldToMaturity)
xlim1 = c(1:2); xlim1[1] = MinXValue; xlim1[2] = MaxXValue
MaxYValue = max(YieldToMaturityError1, YieldToMaturityError15, YieldToMaturityError30)
MinYValue = min(YieldToMaturityError1, YieldToMaturityError15, YieldToMaturityError30)
ylim1 = c(1:2); ylim1[1] = MinYValue; ylim1[2] = MaxYValue
legtxt = c("30 Year","15 Year","1 Year")
Title1 = "Estimation Error by Yield"
xTitle = "Yield To Maturity"
yTitle = "Estimation Error"
# Plot footers
TC = paste0('Coupon = ', CouponRate, '%')
TPar = paste0(', Par = $', ParValue)
sTitle = paste0(TC, TPar)
lTitle = "Maturity"
plot(YieldToMaturity, YieldToMaturityError30, type="b", main=Title1,
     sub=sTitle, xlab=xTitle, ylab=yTitle, col="black", xlim = xlim1,
     ylim = ylim1, pch = 1, cex = 0.5, lty = 1)
lines(YieldToMaturity,YieldToMaturityError15, type="b", col="black", xlim = xlim1,
      ylim = ylim1, pch = 2, cex = 0.5, lty = 2)
lines(YieldToMaturity,YieldToMaturityError1, type="b", col="black", xlim = xlim1,
      ylim = ylim1, pch = 3, cex = 0.5, lty = 3)
legend("topright", legtxt, cex = 0.75, lwd = c(1,1,1), lty = c(1,2,3),
       col = c("black","black","black"), pch = c(1, 2, 3), bty = "n",
       title = lTitle)
```

Module 3.7: Numerical Integration and the Lognormal Distribution

See *Numerical Integration Test.R*. We extensively review the normal and lognormal distribution properties as they are widely used in quantitative finance. See also 3.7 Numerical Double Integration Test.R that illustrates double integrals of both the normal and lognormal distributions. Double integral solution tools are useful particularly for spread options and compound options.

Numerical Integration Test.R (Selected Excerpts and Output)

In two separate files, we build functions for selected parameters related to the normal and lognormal distributions. See Normal Distribution Functions.R and Lognormal Distribution Functions.R.
This code starts with test data and testing of several of these functions.

```
# Test Data
StockPrice = 100.0
StrikePrice = 100.0
InterestRate = 12.0
DividendYield = 0.0
Volatility = 30.0
TimeToMaturity = 1.0
source('Normal Distribution Functions.R')
source('Lognormal Distribution Functions.R')
#
# Function tests
#
NMean <- NormalMean(StockPrice, InterestRate, DividendYield, TimeToMaturity,
  Volatility)
NSD <- NormalStandardDeviation(Volatility, TimeToMaturity)
NSKewness <- NormalSkewness() # Known to be zero
NExcessKurtosis <- NormalExcessKurtosis() # Known to be zero
NEntropy <- NormalEntropy(Volatility, TimeToMaturity)
LNMean <- LognormalMean(StockPrice, InterestRate, DividendYield,
  TimeToMaturity, Volatility)
LNSD <- LognormalStandardDeviation(StockPrice, InterestRate, DividendYield,
  TimeToMaturity, Volatility)
LNSKewness <- LognormalSkewness(Volatility, TimeToMaturity)
LNExcessKurtosis <- LognormalExcessKurtosis(Volatility, TimeToMaturity)
LNEntropy <- LognormalEntropy(StockPrice, InterestRate, DividendYield,
```

```
  TimeToMaturity, Volatility)
#
# Test normal PDF via integration
#
d = 0
LowerBound = -Inf
NMean <- NormalMean(StockPrice, InterestRate, DividendYield, TimeToMaturity,
  Volatility)
NSD <- NormalStandardDeviation(Volatility, TimeToMaturity)
UpperBound = NMean
Results = integrate(NormalPDF, LowerBound, UpperBound, NMean, NSD)
N = Results$value
N
[1] 0.5
```

Integration is illustrated with the integrate() function related to N(d) in numerous option valuation models.

```
NMean <- NormalMean(StockPrice, InterestRate, DividendYield, TimeToMaturity,
  Volatility)
NSD <- NormalStandardDeviation(Volatility, TimeToMaturity)
d2 = (NMean - log(StrikePrice)) / NSD
LowerBound = -Inf
UpperBound = d2
Results = integrate(NormalPDF, LowerBound, UpperBound, 0, 1)
Nd2V = Results$value
Nd2V
[1] 0.5987063
…
Nd1V = Nd1(StockPrice, StrikePrice, InterestRate, DividendYield,
  TimeToMaturity, Volatility)
Nd2V = Nd2(StockPrice, StrikePrice, InterestRate, DividendYield,
  TimeToMaturity, Volatility)
```

The probability, under the equivalent martingale measure, of an option being in the money is related to $N(d_2)$.

```
CallITMProb = Nd2(StockPrice, StrikePrice, InterestRate, DividendYield,
  TimeToMaturity, Volatility)
PutITMProb <- 1.0 - Nd2(StockPrice, StrikePrice, InterestRate, DividendYield,
  TimeToMaturity, Volatility)
TotalProb = CallITMProb + PutITMProb
CallITMProb; PutITMProb; TotalProb
[1] 0.5987063
[1] 0.4012937
[1] 1
```

With this set-up, several interesting observations can be made. First, we explore further the influence of volatility on both the normal and lognormal PDFs and CDFs. See Density and Distribution Study.R. Based on the following loop, we illustrate the following PDFs and CDFs.

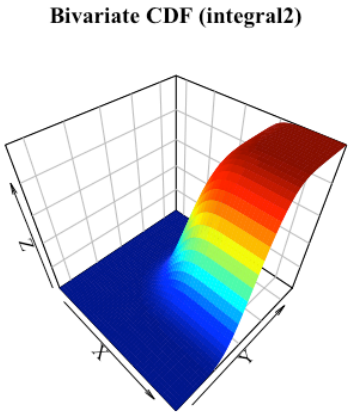
```
OriginalVolatility <- Volatility
Increment <- 50
for(i in 1:5){
  Volatility <- OriginalVolatility + (i-1)*Increment
  source('Density and Distribution Study.R')
}
Volatility <- OriginalVolatility
```

Numerical Double Integration Test.R (Selected Excerpts and Output)

There are several ways to integrate either the normal or lognormal distribution. We illustrate just a few in R here. First, simply taking the double integral of the normal distribution is illustrated in a three dimensional plot based on a standard bivariate normal.

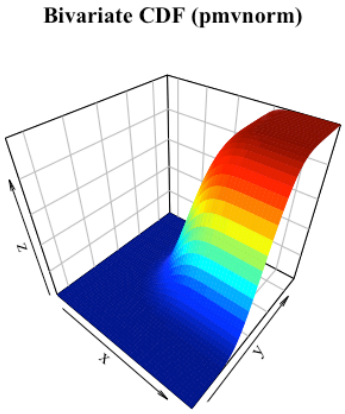
```
    N2CDFV1[i, j] <- as.numeric(integral2(Normal2PDF, xmin = MinX1, xmax = MaxX1,
      ymin=MinX2, ymax = MaxX2, reltol = 1e-6, Mu1 = Mu1, Mu2 = Mu2,
      SD1 = SD1, SD2 = SD2, rho = rho)[1])
```

Figure 3R.7.6 Bivariate Normal CDF based on double integral



Alternatively, we can use the mvtnorm package to compute the same bivariate normal CDF.

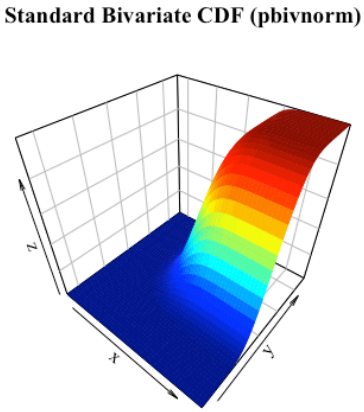
Figure 3R.7.7 Bivariate Normal CDF based on mvtnorm package



Finally, we can used the pbivnorm package to compute the standard normal bivariate CDF.

```
    N2CDFV3[i, j] <- pbivnorm(x = MaxX1, y = MaxX2, rho = rho, recycle = TRUE)
```

Figure 3R.7.8 Bivariate Normal CDF based on pbivnorm package



Whenever using integration packages, it is always suggested to evaluate the differences between various calculation methodologies. We see below that the maximum absolute error is small.

```
> MaxError12 <- max(abs(Error12))
> MaxError13 <- max(abs(Error13))
> MaxError23 <- max(abs(Error23))
> MaxError12; MaxError13; MaxError23
[1] 1.688078e-08
[1] 5.901631e-07
[1] 5.73303e-07
```