# Chapter 1

Introduction

1

## Book Overview

- Introduction (this presentation)
- R Preliminaries (in appendix)
- Quantitative finance tools
- Instrument valuation
- Static risk management
- Dynamic market risk management
- Selected portfolio issues

2

## Book Design

- Modular (after chapter 2)
- Features of each module
  - Central finance concepts
    - Non-quantitative
    - Case studies, examples, illustrations
  - Quantitative finance materials
    - Very technical materials
    - Derivations and mathematical representations
  - R commentaries (in separate file)

3

## Quants Must Have Humor!

Programming in R may simply give you a more sophisticated way to harm yourself. R plus wisdom is extremely powerful.


Source: Lost.



4

## Introduction to Introduction

- Purpose: Not to provide state-of-the-art R programming techniques (provide selected)
- Purpose: Not to provide state-of-the-art quantitative finance techniques (provide selected advanced materials illustrating cutting edge analysis)
- Purpose: Provide as simple an approach as possible to learn prototype implementation code
- Facilitate implementation of quantitative finance ideas in R

5

## Finance as a Social Science

- Both mechanism and agency important
- Quant infrastructure needs flexibility
  - Manage rapid business expansion and contraction
  - Manage changing quantitative solutions
- Interchangeable valuation and risk management frameworks
- Continuous process

6

## Case for Computer Language

- Conformist versus non-conformist
- Clear and crisp understanding of model
- Build versus buy
- Computer language (CL) or spreadsheets
- CL or symbolic languages
- Improved communication with internal software developers
- More efficient debugging
- Decomposition

7

## Conformist vs. Non-Conformist

- Unique language
- Methods of expression
  - Client presentations
  - Valuation methodologies
  - Data collected
- R expands means of expression
- Improved software tools speeds the process

8

## Quants as Artists

9

## Clear / Crisp Understanding

- Computer program – 99% correct is 100% wrong
- Books often provide only a 30,000-foot perspective of financial models
- Example: Plain vanilla interest rate swap
  - Quarterly, ACT/360 FLT; Semi, 30/360 FIX
  - Payment frequency and day count have significant value

10

## Build Versus Buy

- Build
  - Takes time and energy, risk errors but …
  - Fully understood by someone within firm
  - Easy to modify as conditions change (social science)
- Buy
  - Pay immodest fee, someone else liable, fast but …
  - No one internal understands model nuances
  - Black box (only know inputs and interpretation of outputs)

11

## Spreadsheets

- Analysis only compliant with spreadsheet framework (all other solutions not considered)
- Large, complex problems difficult in spreadsheets
- Slow and cumbersome
- CL: Decompose large, complex problems
- CL: Fast and flexible

12

## Symbolic Languages

- Not portable
- Rely on existence of symbolic language provider
- CL: very portable
- CL: rely only on existence of language, not specific compiler

13

## Improved Communication

- Internal software developers do not understand finance language
- Advanced quantitative finance applications often are fraught with nuances (social science)
- Financial analyst can better communicate with internal software developers if understand computer language

14

## More efficient debugging

- Quantitative finance models are complex (speed, real time data, multidimensional, advanced math)
- "Bugs" are rampant 
- Financial analyst is well-suited to efficiently debug

15

## Decomposition

- Breaking down problem into smaller manageable pieces
- One goal here is to help you develop skills to achieve optimal level of decomposition

16

## Summary

- Finance is a social science (thus, mechanism and agency)
- Case made for learning a computer language
- Embrace perspective of being an artist within a quantitative social science

17

## R Commentary
## Why Learn the R Language?

- Jobs
- History of computer programming languages
  - Low level language: machine level code, powerful
  - High level language: easy to use
  - mid-1950: FORTRAN (formula translation), high level language
  - C++: combines high level (easy to use) and low level (powerful), fast, object-oriented
  - R: Easy and can link with C++
- Rapid application development

18

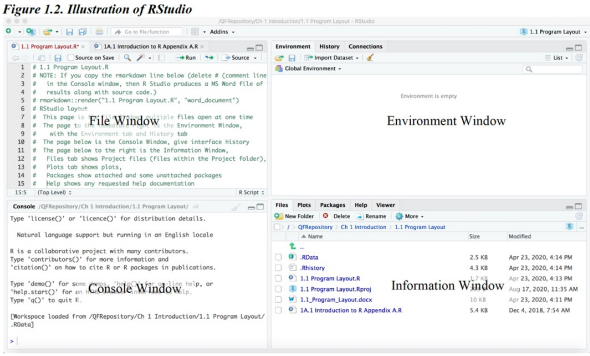
## Learning R

- Autonomous versus heteronomous
  - Autonomous: Freedom to act independently, training materials are compiler independent
  - Heteronomous: Subject to external standard, training materials specific to one compiler
- Deliverables: Simple prototype programs
  - Actual implementation requires exhaustive error-trapping, real time data
  - Goal: Deployable code



19

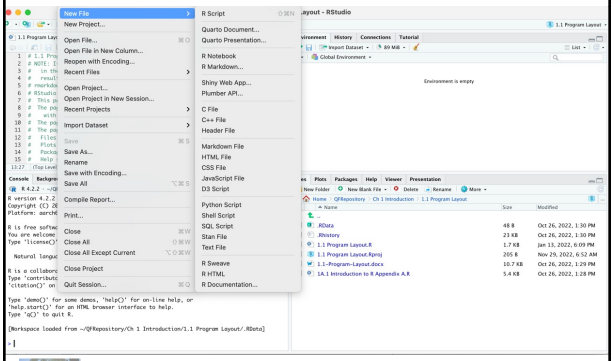
*Figure 1.2. Illustration of RStudio*






20

RStudio supports C, C++, Python, Shiny (html), SQL, and more.

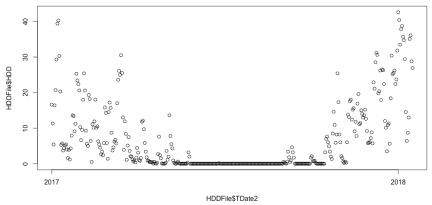




21

## 1.2 Heating Degree Days

- HDD: The number of degrees that a day's average temperature is below 65 degrees Fahrenheit






22

## Appendix: Building a Repository

- Managing subdirectories for source code
  - C:\QFRepository
- Managing files
  - Modular development
  - Each module independent



23

## Appendix: Coding Preferences

1) Indent two spaces (even for wrapped lines) after each curly bracket ({), open ({) on same line and close (}) on new line
2) Use blank lines very rarely. If you need a space, include a comment line
3) Your name should be on the first line of each file



24

## Coding Preferences Continued

4) R function code should be separated from other code
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



25

## Example Code: Name on Top

```
#                       Robert Brooks
# 5.4 GBMOVM Test.R
# Illustrating functions in R (function definitions in separate file)
# rmarkdown::render("5.4 GBMOVM Test.R", "word_document")
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
par(family = 'Times New Roman') # Globally set fonts for graphs
# Generic test inputs
inputStockPrice = 100.0
inputStrikePrice = 100.0
inputInterestRate = 5.0        # In percent
```



26

## Indent, No Extra Lines

```
PutTimeValue <- c(1:NumberOfObservations)
for(i in 1:NumberOfObservations){
  StockPrice[i] <- as.double(LowerBound + (i-1)*StepSize)
  GBMInputData$StockPrice = StockPrice[i]
  GBMInputData$Type = 1
  CallLowerBound[i] <- OptionLowerBound(GBMInputData)
  CallUpperBound[i] <- OptionUpperBound(GBMInputData)
  CallValue[i] <- GBMOptionValue(GBMInputData)
  CallTimeValue[i] <- CallValue[i] - CallLowerBound[i]
  GBMInputData$Type = -1
  PutLowerBound[i] <- OptionLowerBound(GBMInputData)
  PutUpperBound[i] <- OptionUpperBound(GBMInputData)
  PutValue[i] <- GBMOptionValue(GBMInputData)
  PutTimeValue[i] <- PutValue[i] - PutLowerBound[i]
}
```



27

## Functions Separated

5.4 GBMOVM Test.R | GBMOVM Functions.R

Source on Save | Run | Sou

```
#
# Functions for GBMOVM-related calculations
#
PV1 = function(Maturity, Rate){
  return(exp( -(Rate/100.0) * Maturity) )
}
# d - functions used in GBMOVM
# with: Evaluates R expressions based on a set of data (B here)
d1 = function(B){
  with(B, {
    Num = ( ((InterestRate - DividendYield) / 100) + ((Volatility/100)^2)/2) *
      TimeToMaturity
    Num = log(StockPrice/StrikePrice) + Num
    Den = (Volatility/100)*sqrt(TimeToMaturity)
    return( Num/Den )
  })
}
```



28