Shiny alternative for Finance in the Classroom

Jayanth R. Varma
Vineet Virmani

W.P. No. 2017-03-05
March 2017

INDIAN INSTITUTE OF MANAGEMENT
AHMEDABAD – 380015
INDIA

Shiny alternative for Finance in the Classroom

Jayanth R. Varma
Vineet Virmani

Abstract

Despite the popularity of open-source languages like R and Python in modern empirical research and the data-science industry, spreadsheet programs like Microsoft Excel remain the data analysis software of choice in much of the business-school curriculum, including at IIMA. Even if instructors are comfortable with modern programming languages, they have to pitch their courses at the level of computer literacy prevalent among students. Excel then appears to be a natural choice given its popularity, but this choice constrains the depth of analysis that is possible and requires a certain amount of dumbing-down of the subject by the instructor. Recent software advances however make the ubiquitous web browser a worthy challenger to the spreadsheet. This article introduces one such browser-based tool called Shiny for bringing finance applications to the classroom and smart phones. Fueled by the availability of high-quality R packages in finance and statistics, Shiny brings together the power of HTML with the R programming language. It naturally creates an environment for the instructor to focus on the role of parameters and assumptions in analysis without the clutter of data, and allows the instructor to go beyond the toy problems that are necessitated by the nature of spreadsheets. The learning curve is short for an interested instructor with even a rudimentary exposure to programming in any language. The article ends with the discussion of a fully-worked out example of Shiny for teaching the mean variance efficient frontier in a basic investments course.

Keywords: Finance, Open-source computing, Pedagogy, R , Shiny, Statistics

1 Introduction

There is no denying the popularity of spreadsheets, and in particular Microsoft Excel, in financial reporting and modeling (Panko and Ordway, 2008). Excel is also one of the most popular software for data analysis courses in the classroom, and much of post-graduate finance and statistics teaching relies on it (Adams et al., 2013; Nash, 2006).

Examples and websites illustrating applications in Excel abound. There is no dearth of helpful material on Excel for topics ranging from time value of money (Zhang, 2015) to regression (Berenson, 2013) and portfolio analysis (McDermott, 2010) to solving complex problems of optimal control (Nævdal, 2003).

Barreto (2015) argues that Excel provides a "just right" balance of software for training in the classroom. With enough advanced features available, it offers the advantage of gradually moving up the learning curve to implementing more complicated models.

However, it is also true that Excel has been named an important culprit in high profile instances of operational losses, both financial and reputational (JP Morgan, 2013).

While some cases are more publicized than others because of the organizations involved,[1] the problem is more widespread. So much so that there exists a dedicated special interests group called the European Spreadsheet Risks Interest Group with the sole focus of highlighting dangers in use of spreadsheets in organizations. Even the Basel Committee on Banking Supervision explicitly warns financial institutions to build controls against errors arising due to 'manual processes' (Bank of International Settlements, 2013).

Notwithstanding the increase in awareness about errors caused due to spreadsheets, Excel remains popular with both students and instructors in business schools (Adams et al., 2013; Nash, 2006). With the generations of faculty themselves being trained and used to spreadsheets, and with Apple and Microsoft resellers bundling the Office suite virtually for free for students,[2] it is not surprising either.

This article, however, is not another attack against spreadsheets or Excel. The problems with Excel for applications in finance and statistics are well-known and written about at length (McCullough and Heiser, 2008; Yalta, 2008; Panko and Ordway, 2008). Our objective is to introduce an alternative way of approaching empirical finance in the classroom using the R programming language.

Recent software advances make the ubiquitous web browser a worthy challenger to the spreadsheet. We introduce one such browser-based tool called Shiny for bringing finance applications to the classroom and smart devices. Shiny brings together the power of HTML and Cascading Style Sheets (CSS) with the sophistication of the R programming language and contributed packages.

Shiny with R naturally facilitates an efficient separation of data (fixed), inputs (constants vs variables) and output (tables vs figures), and allows for focus to remain on important issues like the role of parameters and assumptions in models. Without having to deal with clutter of data and unwieldy formatting, the instructor can go beyond the toy problems that are necessitated by the nature of spreadsheets.

As we illustrate with examples, Shiny offers a full-fledged programming environment in which simple applications easily scale up to advanced models without any cost to the the end-user. The learning curve is short for an interested instructor with even a rudimentary exposure to programming in any language.

2 The R Environment

The R project and the programming language began in the 1990s as an offshoot of the S language by Robert Gentleman and Ross Ihaka at the University of Auckland. Its name is both a hat-tip to the developers' initials while also being a play on the name of the language on which it is based.[3]

[1] `http://www.eusprig.org/horror-stories.htm`, accessed March 27, 2017.

[2] `https://www.microsoft.com/en-in/education/students/deals/default.aspx`

[3] Ross Ihaka, "R: Past and Future History", `https://cran.r-project.org/doc/html/interface98-paper/paper.html`, 1998, accessed March 27, 2017.

Since the mid-2000s, R has come to be one of the most important languages and environment for empirical work in academia and businesses. It is also the fastest growing language for training in empirical methods at universities worldwide.[4] Since the rise of data science as an industry, R has come to be the main competitor of Python,[5] and according to IEEE, its rank has been consistently rising in the list of most important programming language for jobs in the last three years.[6]

While there is no dearth of tutorials and books available for R , the wealth of information available can be overwhelming for a beginner. Given the popularity of the language, today popular Massive Open Online Courses (MOOC) platforms like Coursera offer formal courses covering a variety of applications using R .[7] For a beginner, we however recommend starting with the official R manual and tutorials available at its home page[8] and then going from there depending on one's area of specialization. The installation instructions for different platforms are also available at the same page as the manuals.

Once one has understood the basic R syntax and practised examples given in the manual, the job of installing the right R packages is a cinch. The R core development team has come up with a suite of packages called Task Views (Zeileis, 2005).[9] So as one prepares to work in any given field, say, Finance or Econometrics, all one needs is to install the associated Task View. This will not only install all the commonly used and popular packages in the field, but also their dependencies.[10] For example, the Finance Task View is easily installed by running the following commands in R console:

```
install.packages("ctv")
library("ctv")
install.views("Finance")
```

An advanced user, of course, might want to be more selective about the packages being installed. The entire Task View could be an overkill if one is going to be working on only a select set of applications. However, given how light most packages are (with many of them written in C++), and easy availability of memory and hard disk, one need not fret about it much. In fact, one could customize one's own list of packages in a custom Task View too.[11] This is ideal convenient an instructor who foresees repeat usage of a specific set of R packages in teaching over multiple years. This has some initial set-up cost, but in the long-term it makes life easier not only for the instructor but also for the students.

For a beginner, if one is installing a package which is not part of the Task View, it might be useful to go through the package manual and find out about the package developers.

[4] R. A. Muenchen, "The Popularity of Data Science Software," February 28, 2017, `http://r4stats.com/articles/popularity/`, accessed March 28, 2017.

[5] `https://www.datacamp.com/community/tutorials/r-or-python-for-data-analysis`, accessed March 27, 2017.

[6] `http://spectrum.ieee.org/computing/software/the-2016-top-programming-languages`, accessed March 27, 2017.

[7] See, for example, the suite of courses and specializations offered at Coursera: `https://www.coursera.org/courses?languages=en&query=r+programming`, accessed March 27, 2017.

[8] `https://cran.r-project.org/manuals.html`

[9] The list of Task Views is available at: `https://cran.r-project.org/web/views/`. Click on a Task View to see the packages included in it.

[10] This might take a while though depending on the network speed and the Task View being installed.

[11] See `http://stackoverflow.com/questions/7265133/how-can-i-list-packages-not-included-in-any-r-task-view`, accessed March 28, 2017.

As a rule of thumb, if the manual is detailed and the developers are well-known in their fields (easily checked using search engines), one should feel safe about the accuracy and quality of implementation. Two highly useful and reliable resources which regularly publish reviews of new packages are *The R Journal* published by the R foundation itself[12] and the *Journal of Statistical Software*,[13] a free open-access journal published by the Foundation for Open Access Statistics.

3 The Shiny Package

The Shiny package was developed by RStudio,[14] the company behind the popular eponymous integrated development environment (IDE) for R . Shiny, or Shiny app as it is called,[15] is essentially an HTML document hosted on a computer running R (this could be one's personal computer or a remote server in the intranet or internet), which means any smart device running a modern browser can run the apps.

Like any app in the modern smartphone sense of the word, it can take clicks and keystrokes, interpret them in R and send the output back to the browser window. The input can be in the form of sliders, drop-downs, text fields or even mouse clicks, and it supports output in all familiar forms including figures, tables and summaries. Once developed, the apps can be shared via cloud, intranet or internet, and run on any browser-enabled smart phone, making it ideal for use within and outside the classroom.

Even though a Shiny app is an HTML document, and an advance user can easily enhance and embellish the apps by using HTML and JavaScript, it neither assumes nor requires any knowledge of HTML on part of the instructor. Pretty sophisticated apps can be built by Shiny functions alone, so an instructor need only be familiar with R and the Shiny environment.

3.1 Elements of a Shiny App

The way Shiny is designed, there are two components/files to every Shiny app:

1. The user interface script (`ui.R`): This component handles the user experience. It sets the page details (the way the app looks like), lists the input options and defines the output formats.
2. The server script (`server.R`): This component does all the R work, meaning it handles all the input calls and instructions given to the app and returns the output objects to be displayed on the browser.

Although, technically, both the components can reside in a single file named `app.R`, RStudio recommends that they be separate for reasons of tractability and ease of debugging.

[12] `https://journal.r-project.org/`
[13] `https://www.jstatsoft.org/`
[14] `https://www.rstudio.com/`
[15] `https://shiny.rstudio.com/`

3.1.1 A Simple Example

The first example at the Shiny Github demo page[16] plots an histogram for a well-known data on waiting time between eruptions of the Old Faithful geyser in Yellowstone National Park, USA.[17] The lesson is to illustrate the impact of bin size on the shape of the histogram. The app consists of the following `ui.R` and `server.R` files:

1. `ui.R`: The user-interface script controls
   a) The title of the app (`titlePanel`)
   b) How the app looks on the browser: In this case with a left hand side area (`sidebarLayout`) to collect inputs and the remaining area to display output
   c) How it handles the input: In this case a `sliderInput` (with minimum, maximum and the default value defined) for controlling the bin size for the histogram
   d) How it displays the output: In this case a plot or a figure (`plotOutput`)

```
# Define UI for application that draws a histogram
fluidPage(

  # Application title
  titlePanel("Hello Shiny!"),

  # Sidebar with a slider input for the number of bins
  sidebarLayout(
    sidebarPanel(
      sliderInput(inputId = "bins", label = "Number of bins:",
                  min = 1, max = 50, value = 30)
    ),

    # Show a plot of the generated distribution
    mainPanel(
      plotOutput("distPlot")
    )
  )
```

2. `server.R`: The server script reads the data, gets the number of bins from `ui.R` (`input$bins`) and tells R to plot a histogram (`renderPlot`) given the number of bins and other fixed histogram characteristics like color (`darkgray`) and border (`white`).

```
library(shiny)

# Define server logic required to draw a histogram
function(input, output) {

  # Expression that generates a histogram. The expression is
  # wrapped in a call to renderPlot to indicate that:
  #
  #  (1) It is "reactive" and therefore should be automatically
```

[16] `https://github.com/rstudio/shiny-examples/tree/master/001-hello`, accessed March 27, 2017.

[17] `https://stat.ethz.ch/R-manual/R-devel/library/datasets/html/faithful.html`, accessed March 27, 2017.

```
  #     re-executed when inputs change
  #  (2) Its output type is a plot

  output$distPlot ← renderPlot({
    x    ← faithful[, 2]  # Old Faithful Geyser data
    bins ← seq(min(x), max(x), length.out = input$bins + 1)

    # draw the histogram with the specified number of bins
    hist(x, breaks = bins, col = 'darkgray', border = 'white')
  })

}
```

3. Finally, the app is run by putting the two together as:[18]

```
shinyApp(ui = ui, server = server)
```

The reader would notice that in the app one is able to neatly describe the sensitivity of the histogram to the bin size, with both data and plotting functions hidden behind the scenes. The user plays around with only what is important for the task at hand. This is unlike Excel where tabulation of data, analysis and plots have no natural separation. However, it is desired to see the tab ala Excel, it is straightforward to do so in Shiny using, for example, a `tabsetPanel` where data could be kept/called on demand in one of the tabs. In fact, that's what makes browser-styled tabs so popular in Shiny - they are ideal for keeping space aside for different kind of outputs.

3.2 The Shiny Environment

From a pedagogical standpoint, when building Shiny apps it is useful to think in terms of inputs and outputs. Inputs provide values to the app (e.g. numbers, clicks), and outputs are R objects that are produced by the R code (e.g. tables and figures).

Shiny offers a variety of input and output functions in addition to various design elements that control the look and feel of the app (HTML wrapper functions essentially). In a sense, then, all one needs is a basic understanding of Shiny input and output functions and some idea on how to customize the user interface using in-built wrappers.

3.2.1 Input functions

Input functions control how the user interacts with the app, i.e. what kind of inputs the user can provide and in what format. For instance, in the example above on plotting histogram, the input function `sliderInput("bins", ...)` created a slider object to be displayed in the browser.[19] The slider recorded the user's input and passed it onto the variable `input$bins`.

In all, Shiny provides for the following input functions that can be used in the apps:

[18]This particular app with some more features (along with the user interface and server files) is available to be run at the RStudio gallery at `https://shiny.rstudio.com/gallery/faithful.html`, accessed March 27, 2017..

[19]`sliderInput` actually generates HTML code which in turned displays the slider in the browser

- `actionButton` and `actionLink`: They both have the same function, but look different in the app. The `actionButton` creates a 'button' in the app which when clicked calls for some changes to the app. The `actionLink` function creates a hyperlink to achieve the same.
- `checkBoxInput` and `checkBoxGroupInput`: `checkBoxInput` creates a check box, which when clicked specifies a logical value (whether a condition/case is TRUE or FALSE). The `checkBoxGroupInput` does the same for a set of logical conditions at the same time. It is a convenient method to toggle multiple choices independently. For example, if one wants to show only few columns of a table, a table header could be set to TRUE or FALSE using a check box input.
- `dateInput` and `dateRangeInput`: As the names suggest these allows for getting a specific date or a date range as input - tremendously useful in creating custom plots.
- `radioButtons` and `selectInput`: These allow for selection of inputs as radio buttons or via drop down menus.
- `fileInput`, `numericInput`, `passwordInput`: These three inputs respectively allow for getting data from a file, as a number or a string for authentication (protected on display as dots).
- `submitButton`: This input is often used in conjunction with other inputs to confirm their value. So the R code changes its state on input only after `submitButton` confirms it.

We'll have an opportunity to use quite a few of the input functions above in our detailed example later.

3.2.2 Output functions

Analogous to the different kind of input functions, the user interface creates output objects using output functions. In the earlier example, a plot (histogram) was created using `plotOutput`.

Shiny supports many others including `dataTableOutput` (for an interactive table), `htmlOutput` (for raw HTML), `imageOutput` (for an image), `tableOutput` (for a table), `textOutput` (for simple text), `uiOutput` (for a Shiny user-interface element) and `verbatimTextOutput` (for verbatim text).

While the output function specifies the name of the object,[20] the output object itself (in this case, an histogram) gets created in the server script which assembles inputs into outputs.

So in the server script in the histogram example (excerpt below), the bin size is kept track of in `input$bins`, the output object is accessed by `output$distPlot` and they are assembled together using the function `renderPlot({...})`.

[20]As with input functions, output functions also internally generate HTML code which sets aside space/-format in the user interface for the output object

```
function(input, output) {
  output$distPlot ← renderPlot({
    ...
    bins ← seq(... , length.out = input$bins + 1)

    # draw the histogram with the specified number of bins
    hist(... , breaks = bins , ...)
  })
}
```

For each output function type, there is an associated render function, namely: `renderPlot`, `renderDataTable`, `renderImage`, `renderPrint` (associated with verbatim text), `renderTable`, `renderText` and `renderUI`.

Shiny articles[21] is the definitive source to learn more about the input and output functions but it maybe easiest to learn Shiny by running the examples in the gallery[22] after going through the basic tutorials.[23]

While familiarity with all the input and output functions is useful (and happens over time), for a given application, say finance or statistics, realistically one would use one class of inputs and output functions more often than the others. So, to begin with one needs to get adept at only a subset of input and output functions. The source codes for the examples in the gallery act as useful templates to build upon one's own application without having to learn everything about it from scratch.

3.3 Reactivity

The power of Shiny lies in what the guys at RStudio call the 'reactive programming' framework. Reactivity is what allows the R program to 'react' to user inputs through clicks and keystrokes.

In the example above, the `sliderInput("bins", ....)` collects the value of `bins` from the user. R reads this input as `input$bins` and stores the location of the slider. In other words, the value of `input$bins` reflects the current value of the slider position. The `input$bins` object changes whenever user slides the slider. In general, the Shiny code is said to be reactive whenever the current value of the input is used to render an output object.

Technically speaking, `input$bins` is called a reactive value, and `renderPlot({...})` a reactive function. Reactive values work only with reactive functions. For example, if `hist(...)` is called in the server script outside `renderPlot({...})`, it is not reactive, and would throw up an error like "Operation is not allowed without an active reactive context."

There are circumstances when a reactive value needs to be called multiple times within the same code, and then it is often convenient to work with reactive expressions. Called as functions, reactive expressions cache their values to the most recent calculation

[21] `https://shiny.rstudio.com/articles/`
[22] The Shiny gallery at `https://shiny.rstudio.com/gallery/` contains many useful examples along with the source code on their Github page at `https://github.com/rstudio/shiny-examples`
[23] `https://shiny.rstudio.com/tutorial/`

avoiding unnecessary computation. They help make the code modular and easier to read/modify.

One could also make the app respond to any changes in the input by forcing an `eventReactive` call. In this case a reactive expression gets updated only when the user confirms by, say, clicking on an `actionButton`.

An Excel user might think of reactivity as a spreadsheet with automatic refresh turned on. Unlike Excel, however, Shiny offers a lot more control on what is to be recalculated and when. For example, if one does not want the app to respond to any specific changes in the app (say if one wants to keep certain inputs or texts frozen), one can easily 'isolate' it as, for example, `isolate(input$bins)`. `isolate()` makes objects non-reactive, and then they behave like normal R values.[24]

3.4 Customizing Shiny

The look and feel of Shiny apps is controlled by layouts, panels and HTML tags.

Layouts. Layout functions are used to position different elements of the app (inputs and output) on the page. So in the histogram example, to get the slider in a small 'left panel' leaving a large space for plot in the 'right panel', a `sidebarLayout` was used. One could achieve the same objective 'manually' by dividing the empty `fluidPage()` into a number of rows and columns using `fluidRow` and `column` functions.

Panels. After the layout is created, one can group multiple elements into a single unit with its own properties. Such single units are called panels in Shiny. In the histogram example this was achieved by having a `sidebarPanel` for the bin size input and a `mainPanel` for the histogram.

Other available panels in Shiny include: `absolutePanel`, `conditionalPanel`, `fixedPanel`, `headerWithPanel`, `inputPanel`, `mainPanel`, `navlistPanel`, `tabPanel`, `tabsetPanel`, `titlePanel` and `wellPanel`.

Pages. While the `fluidPage()` is one of the most commonly used default/empty user interface, two other popular ones include the `navbarPage` (with the associated `navlistPanel` and `navbarMenu`) and `dashboardPage`. `navbarPage` layout faiclitates browser-styled tabs facilitating different kind of outputs/data to appear in different tabs.[25]

Shiny HTML tags. Once the basic app functionality and the layout is ready, one can use HTML wrappers available within Shiny to further format and decorate the app keeping in mind the in-class usage.

Some of the wrappers available in Shiny for facilitating text formatting/design include:

- Hyperlinks: `a(href = "www.iima.ac.in", "IIM Ahmedabad")`

[24]For more see `https://shiny.rstudio.com/articles/reactivity-overview.html`

[25]This can also be achieved by `tabsetPanel`. See the Layouts and UI chapter at `https://shiny.rstudio.com/articles/`

- Headings: `h1("First")` for a first level heading. One can go up to `h6`.
- New line: `p("New line starts here")`
- Text formatting: `em("This would be in italics")` and `strong("This would be in bold")`
- Code: `code("This is in monospace / verbatim / shaded code")`

Note that wrappers work without having to call the HTML `tags$` prefix. In addition to these commonly used functions, Shiny can also take raw HTML and import custom CSS for building further sophistication.[26]

4 A Finance Application: The Efficient Frontier

Most courses on portfolio analysis and investments in business schools begin with mean-variance optimization (Markowitz, 1952). While in general it involves solving a quadratic programming problem, the basic insight of Markowitz (1952) is that the risk of a security as part of a portfolio is very different from its standalone risk. In most textbooks and classroom the intuition of this idea is usually illustrated by working with two securities.

4.1 The Two Securities Case

If there are two securities, say, `A` and `B` with a mean or expected return (μ) as 10% and 15% respectively, and a standard deviation (σ) of 20% and 30% respectively, the Markowitz's insight is that risk (standard deviation) of the portfolio of the two securities is not minimized by putting all of one's money in security `A` (the one with a lower standard deviation). Shifting a small amount of money to the "riskier" security `B` reduces the portfolio standard deviation as long as the return of the two securities is not perfectly correlated.

With two securities, this idea of diversification ("don't put all eggs in one basket") is easy to illustrate with a spreadsheet. This begins by plotting the portfolio mean and portfolio standard deviation against the portfolio weight in one of the security, say `B` (w_B). Students quickly see that, as expected, the portfolio mean (σ_P) rises linearly from 10% to 15% when the weight in security B is increased from 0% to 100%.

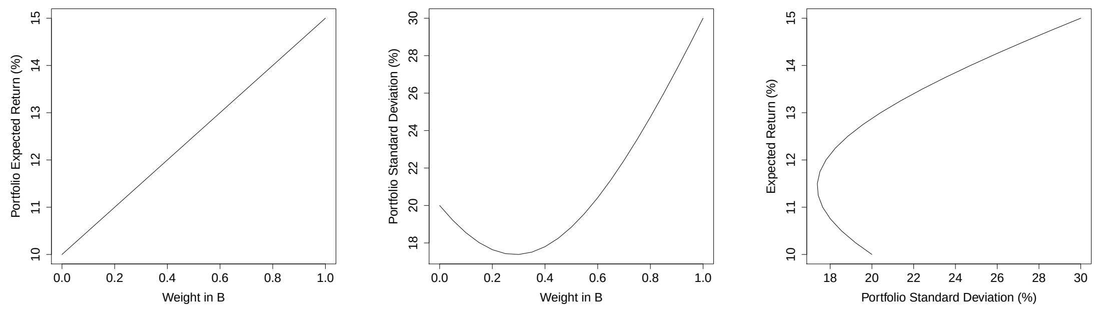
The portfolio standard deviation (σ_P) being a non-linear function of standard deviations of the constituents behaves in a more complex manner. It goes from 20% to 30% as expected, but not in a straight line: it falls below 20% before beginning to rise. This non-linearity follows from the elementary statistics formula for variance of a sum of random variables:

$$\sigma_P^2 = w_A^2\sigma_A^2 + w_B^2\sigma_B^2 + 2\rho w_A w_B \sigma_A \sigma_B$$

[26]For more on customizing user interface with HTML see `shiny.rstudio.com/articles/html-ui.html` and on using CSS see `https://shiny.rstudio.com/articles/css.html`

where, ρ is the correlation between the return of two securities.

Finally, it is possible to plot the mean against the standard deviation (return versus risk) to obtain the classical parabolic shape of the Markowitz efficient frontier. The usual graphs that come about then look something like these:



To be fair, Excel works perfectly for the two asset case. One can build a the portfolio of the two assets with varying weights in two columns, and portfolio return and risk can easily be evaluated using the formulas above in different columns. The three graphs reproduced above result quite easily from the columns of weight in one of the securities, portfolio return and standard deviation.

4.2 Multiple Securities Case

The difficulty in using spreadsheets lie in extending the analysis beyond two securities. Traditionally, the approach in the classroom has been simply to indulge in some kind of hand-waving, and persuading the students by citing important results (Merton, 1972) that the efficient frontier has the same shape when there are a larger number of securities.

One difficulty is that with two securities, all portfolios lies along the parabola (including the “inefficient” lower half), and there are no portfolios in the interior of the parabola. With more than two securities, most portfolios are in the interior and by restricting portfolio choice to the efficient frontier, we are able to rule out a very large fraction of feasible but inefficient portfolios. This key insight of modern portfolio theory cannot be demonstrated with two securities, and is therefore either very hard or cumbersome to show with a spreadsheet.

By using R , the problem goes away, because it is very easy using widely available packages to handle a large number of securities. Also by using Shiny, the instructor can use this power of R without intimidating the students who might not be familiar with this or any other programming language, or even Excel. Students only need to point their browser to the Shiny server and interact with it. They can see the plots change instantly as they change any of the input parameters.

Now we discuss how one would implement a mean-variance efficient frontier app in Shiny.

4.3 A Shiny App for Efficient Frontier

The R package `fPortfolio` (also part of the Finance Task View) provides functions for computing the efficient frontier and plotting the frontier as well as the portfolio composition along the frontier.

1. `ui.R`: Given the understanding that a finance instructor would have about the inputs required to compute the frontier, this should appear quite straightforward. It is designed in the same `sidebarLayout` as the earlier example.

   The `sidebarPanel` collects all inputs including the number of securities (`numericInput`), the names of these securities (`textInput`), their means, standard deviations and the correlations (all `numericInput`). In the same panel, the user also chooses whether to plot the efficient frontier or the portfolio composition (`selectInput`). If the frontier is chosen, there are further choices: whether to plot the two asset frontiers, and whether to plot random portfolios and if so how many (`conditionalPanel`).

   Output objects are created in a `mainPanel` and collected in separate tabs (`tabsetPanel`), with the plot `plotOutput` on one tab and the input data `tableOutput` in another. In this paper, we focus only on the first tab containing the plots.

```
library(shiny)

fluidPage(
    sidebarLayout(
        sidebarPanel(
            numericInput('k', 'No of securities', 3),
            textInput('names', 'Security Names', 'Alpha Beta Gamma'),
            textInput('mean', 'Expected Returns (%)', '10 15 20'),
            textInput('sigmas', 'Standard Deviations (%)', '20 25 30'),
            textInput('correl',
                      'Correlations (R2,1 .. Rk,1 R3,2.. Rk,2 ... Rk,k-1)',
                      '0.3 0.2   0.1'),
            selectInput('type', 'Plot type',
                        list('Efficient Frontier',
                             'Portfolio Composition'),
                        selected = 'Efficient Frontier'),
            conditionalPanel(
                condition = "input.type == 'Efficient Frontier'",
                checkboxInput('frontier.twoasset',
                              'Plot All Two Asset Frontiers',
                              value = TRUE),
                checkboxInput('montecarlo', 'Plot Random Portfolios',
                              value = FALSE),
                numericInput('npoints', 'No of Random Portfolios', 200)
                )
            ) ,

        mainPanel(
            h1('Efficient Frontier'),
            tabsetPanel(
                tabPanel("Plot", plotOutput("plot")),
                tabPanel("Input Data",
                         h4('Means and Standard Deviations'),
                        tableOutput("mu.sd"),
                        h4('Correlation Matrix'),
                        tableOutput("rho")
                        )
                )
            )
        ),
    title ="Efficient Frontier"
)
```

2. `server.R`: The server script requires a bit more work. It needs to read the inputs, compute the correlation matrix and render the output. That said, the reactivity part of the code remains simple as in the case of the histogram. It calls the following functions from `fPortfolio`:

    - `portfolioFrontier` computes the frontier, and `weightsPlot` plots the portfolio composition (weights)
    - `twoAssetsLines` plots all two-asset efficient frontiers. If there are three securities, `A`, `B` and `C`, the efficient frontier drawn by `portfolioFrontier` contains portfolios including all three stocks. `twoAssetsLines` draws three additional efficient frontiers taking two securities at a time: the frontiers with only `A` and `B`, with only `A` and `C` and with only `B` and `C`.
    - `monteCarloPoints` plots the mean and standard deviation of a number of random portfolios. Most of these would be inefficient portfolios, but they map out the entire feasible region in mean – standard deviation space. The efficient frontier produced by `portfolioFrontier` appears as the upper boundary of the feasible region.

    `my.sample()` part of the code is explained later. It is there because of a quirk in the way `fPortfolio` package uses sample moments.

```
library(shiny)
library(fPortfolio)

function(input, output) {
  ## Code to read inputs and generate sample is omitted ...
  ## ... and is implemented before renderPlot({...}) here ...
  ## ...
  output$plot <- renderPlot({
      fp <- portfolioFrontier(as.timeSeries(my.sample()))
      if(input$type == 'Portfolio Composition'){
          weightsPlot(fp)
      }else{
          frontierPlot(fp, col = c('blue', 'red'), pch = 20)
          if(input$frontier.twoasset)
              twoAssetsLines(fp, col = 'green')
          if(input$montecarlo)
              monteCarloPoints(fp, input$npoints,
                                   col = 'grey', pch = 20, cex=0.3)
      }
  })
}
```

3. Preparation work in `server.R`: The remaining part of the server script (before `renderPlot`) consists in parsing the input data, and computing correlations. In this particular case, the mean input also needs to be converted from percent to decimal:[27]

   ```
   mu ← reactive(scan(text=input$mean, quiet = TRUE) / 100)
   ```

   Note that while the `scan` function is standard in R for reading text data, reactivity requires that this be called as a reactive expression so that it can respond to any changes in the data input by the user.

   The correlation matrix computation part of the code (not reproduced here) takes a little more effort because only the lower triangular part of the correlation matrix is typed by the user, and the code has to copy this into the upper triangular part and also fill the diagonal with ones. This requires about a dozen lines of R code easily understood by anyone with a basic understanding of correlation matrix. The full source code is available on the first author's Github page.[28]

4. `my.sample()` object in `server.R`: There is a quirk in the `fPortfolio` package in that it does not allow direct input of mean and covariances, but insists on computing these from a time series of returns. We have no choice but to generate an appropriate sample of returns and feed these returns to the package.

   Now, R has no difficulty generating a sample with specified population mean vector and variance-covariance matrix. The problem is that the sample mean and variance of this sample will not be *exactly* equal to that of the population parameters. One option is to choose a large sample and ignore the small differences between the sample and population values.

   This is one of the things that can often get 'hand-waved' when using spreadsheets in the class. In R there is no such problem. With a bit of re-centering and re-scaling, one can ensure that the sample means and covariances are exactly equal to the desired population values. A non-programmer audience needn't worry about all this, but when they input a desired population moment that is what gets used.

   One could argue that it is the use of the `fPortfolio` package that causes the problem so one should avoid it. But it is one of the most reliable and widely used packages for portfolio analysis in the R universe. And given that the source code is completely transparent about its methods, it is easier to deal with any issue than working with a black-box.

   After the `iid.sample()` code draws uncorrelated multivariate normal variates, and forces the mean to 0, standard deviation to 1 and removes sample correlation, the server script deals with scaling of means, variance and correlation as follows:

   - Means and variances: R provides a function `scale` that can be used to recenter and rescale a sample to change the univariate means and variances. One can also do this directly by dividing the data with sample standard deviation and multiplying the resulting new series by the population standard deviation.

[27]Input format and its use in the code is something that needs to be kept in mind during development.
[28]`https://github.com/jrvarma`

- Correlation: Getting the correlations right takes more effort. Essentially, we need a "square root" of the correlation matrix, and the tool needed for this is Cholesky factorization. The R function `chol` does the job.
- Validation: It also includes some validation to ensure basic requirements (for example, all standard deviations should be positive).

`iid.sample()` and `my.sample()` part of the codes are excerpted from `server.R` below. Ignoring `validate()` functions and annotated comments, the rest of the `my.sample()` code is merely 5 lines.

```
iid.sample <- reactive({
    k <- input$k
    N <- k + 30 # ensure adequate degrees of freedom
    x <- rnorm(N*k) # iid standard normals (in population)
    ## Sample moments of x can differ from population moments.
    ## We now force sample mean = 0 and sample std deviation =
    ## 1 by simple centering and rescaling
    x <- scale(matrix(x, N, k))
    ## To remove sample correlations, we multiply by the
    ## inverse of the cholesky factor of the sample
    ## covariance. The cholesky factor is like the square root
    ## of the matrix
    x %*% solve(chol(cov(x)))
})
## Generate sample with desired sample moments
my.sample <- reactive({
    k <- input$k
    validate(
        need(length(mu()) == k,
             'Wrong number of elements for Expected Returns'),
        need(length(sigmas()) == k,
             'Wrong number of elements for Standard Deviations'),
        need(length(correl.vector()) == k * (k - 1) / 2,
             'Wrong number of elements for Correlation Matrix'),
        need(length(names()) == k,
             'Wrong number of elements for Security Names')
        )
    ## print(eigen(correl())$values)
    validate(
        need(all(eigen(correl())$values > 0),
             'Correlation matrix not positive definite'),
        need(all(sigmas() > 0),
             'Standard deviation not positive')
        )
    ## To replicate the required correlation matrix we
    ## multiply the iid sample by the cholesky factor of the
    ## sample covariance. The cholesky factor is like the
    ## square root of the matrix
    y <- iid.sample()  %*% chol(correl())
    ## We then rescale the variables to achieve the required
    ## standard deviations
    y <- scale(y, center = FALSE, scale = 1/sigmas())
    ## Finally, we then re-center the variables to achieve the
    ## required means
    y <- scale(y, center = - mu(), scale = FALSE)
    dimnames(y)[[2]] <- names()
    y
})
```

4.4 Shiny-R Advantages for Finance

While prima-facie the steps involved in building Shiny apps may seem intimidating, an instructor who needs to use or build similar applications regularly for teaching or research the benefits far outweigh the cost in our opinion.

Reusability. Once the apps are created, they can be used across for different audience that one encounters in a business school. While this is shared with Excel for simple/toy examples to an extent, as argued earlier, Shiny keeps the focus on the elements crucial for a session. The layout of the app can be designed such that the same app can be used for different groups by using tabs to separate different outputs by difficulty level. Select data can also be made non-reactive by using `isolate()` as need be.

Object-oriented and vectorized. Many problems in asset pricing and investment analysis are naturally cast in terms of linear algebra. This makes R an almost-perfect fit for the task, given that indexing, operators and functions in R closely resemble the algebra of matrices. Its vectorization capabilities are often leveraged to speed up the code (Wang et al., 2015).

Graphics. Spreadsheets are particularly infamous for the poor quality of their charting utility for communicating scientific results. In comparison, R offers a modern approach to data visualization using the layered grammar of graphics with a package called `ggplot2` (Wickham, 2010). Coupled with the interactivity that Shiny offers, `ggplot2` is leaps ahead of any spreadsheet software in exploring and visualizing complex data.

Extensions The biggest advantage of working in the R and Shiny environment is possibilities afforded for extensions to the server script by including more advanced versions of basic models. Some possible extensions include:

- Non-Normality of asset returns, say, by bringing in other elliptical or stable distribution using, for example, the `fBasics` package
- Black-Litterman's approach (Black and Litterman, 1992) for introducing investor views and opinions in the Markowitz problem using the `BLCOP` package. For many such nonlinear problems, spreadsheets are known to be particularly unsuitable (Almiron et al., 2010)
- Bring in stylized facts (Cont, 2001) and time series methods to the class using `fGarch`, `rugarch` and `rmgarch` packages.
- Bring in live financial and macroeconomic data from important financial markets directly into the R workspace using packages like `quantMod` and `quandl`.
- Animate apps as GIF or flash movie using the `animate` package (Xie, 2013)

These are only a few obvious examples. Richness of R allows for customization and extensions in different directions.

Of course, the context decides, but depending on the audience, the apps can be as basic or as sophisticated as need be. While investment required on part of the instructor differs depending on the nature of the app, the user experience at all difficulty levels remains the same. All Shiny apps essentially involve interacting with the browser, however simple or complex the app may be. It's a bit like an Android smart phone. A high-end Android phone may have more features (so costs more) than a low-end one, but both run on the same kernel and offer the same basic functionality.

5 Sharing and Deploying Shiny Apps

5.1 Sharing the component files

If the users are familiar with R and have all the necessary data available and R packages are installed on their computers, sharing a Shiny app is as easy sharing the component files and instructions to run the app. It is crude, but it works and may work for a small class.

This may not be ideal with a non-programmer audience though, as the users may inadvertently damage the app by tinkering with the user interface or server files. If the objective, however, is to teach R and Shiny, then an instructor could share component files for an incomplete app and have students finish them as homework. Other than simply sharing the two files over a network or otherwise, they can also be hosted on Github or Gist and run directly from there.[29]

5.2 Hosting a Shiny server

If the users do not have R installed or are not keen to, the utility of Shiny really comes through as an app hosted on cloud. All the users need is a browser and instructions for running the app.

RStudio provides a paid service to do so at `https://www.shinyapps.io/`. They also have a basic free/trial version. An alternative to using RStudio is to deploy apps using a Docker-based technology[30] at the service called ShinyProxy which only relies on the open-source Shiny package without any dependency on the server version of Shiny or RStudio.[31]

Those with some experience in web hosting or a helpful IT department may either set up a Shiny server locally[32] or on any cloud hosting service, be it, e.g., Amazon EC2, Google Cloud or Digital Ocean.[33]

Given the popularity of free Amazon EC2, in Appendix A we provide steps to install and run the efficient frontier Shiny app discussed earlier on Amazon Web Services (AWS).

[29]`https://shiny.rstudio.com/articles/deployment-local.html`, accessed March 27, 2017.
[30]`https://www.docker.com/`
[31]More details at `https://www.shinyproxy.io/`
[32]`https://github.com/rstudio/shiny-server/`
[33]For an example, see `http://deanattali.com/2015/05/09/setup-rstudio-shiny-server-digital-ocean/`, accessed March 27, 2017.

6 Conclusion

As the data science and analytics industry has grown, so has the popularity of open-source languages like Python and R . While the former has gained in importance due to its versatility, R has fast grown into one of the most popular languages for empirical research in social science. There are not many competitors for R when it comes to applications in statistical computing in practice.

For beginning data analysis courses at business schools, however, the software of choice remains Microsoft Excel despite the known problems for its usage in serious finance and statistics applications. It's not merely inertia, but a combination low barriers to entry and lack of exposure of business school students to programming. Even if instructors are adept at using R , they are forced to use Excel given its popularity. The nature of spreadsheets often significantly constrains the depth of analysis possible in the class, often even requiring dumbing-down of the subject.

Recent software advances have brought browser-based tools to fore as capable alternatives to spreadsheets. In this article we have introduced one such tool called Shiny. Since a Shiny app is essentially an HTML document hosted on a computer running R , it is able to leverage the power of HTML and CSS with the sophistication of the R programming language. Once developed, the apps can be shared via cloud and run on any browser-enabled device, making it ideal for teaching. With the detailed documentation available for both R and Shiny, an interested instructor can get up to speed in quick time.

After describing the design philosophy of Shiny and elements of a Shiny app with a toy example, we have provided detailed steps for building an app for explaining mean-variance efficient frontier with arbitrary number of securities. The app is not only amenable to extensions for more advanced courses, it is also straightforward to set it up on the cloud for easy accessibility by any modern browser.

References

Adams, W. C., Infeld, D. L., and Wulff, C. M. (2013). Statistical software for curriculum and careers. *Journal of Public Affairs Education*, 19(1):173–188.

Almiron, M., Lopes, B., Oliveira, A., Medeiros, A., and Frery, A. (2010). On the numerical accuracy of spreadsheets. *Journal of Statistical Software*, 34(1):1–29.

Bank of International Settlements (2013). Principles for effective risk data aggregation and risk reporting. Technical report, Bank of International Settlements.

Barreto, H. (2015). Why Excel? *The Journal of Economic Education*, 46(3):300–309.

Berenson, M. L. (2013). Using Excel for White's Test - An Important Technique for Evaluating the Equality of Variance Assumption and Model Specification in a Regression Analysis. *Decision Sciences Journal of Innovative Education*, 11(3):243–262.

Black, F. and Litterman, R. (1992). Global portfolio optimization. *Financial Analysts Journal*, 48(5):28–43.

Cont, R. (2001). Empirical properties of asset returns: stylized facts and statistical issues. *Quantitative Finance*, 1(2):223–236.

JP Morgan (2013). Report of JPMorgan Chase & Co. Management Task Force Regarding 2012 CIO Losses. `http://files.shareholder.com/downloads/ONE/2272984969x0x628656/4cb574a0-0bf5-4728-9582-625e4519b5ab/Task_Force_Report.pdf`. Accessed March 27, 2017.

Markowitz, H. (1952). Portfolio selection. *The Journal of Finance*, 7(1):77–91.

McCullough, B. and Heiser, D. A. (2008). On the accuracy of statistical procedures in Microsoft Excel 2007. *Computational Statistics & Data Analysis*, 52(10):4570 – 4578.

McDermott, J. (2010). Returns-Based Style Analysis: An Excel-Based Classroom Exercise. *Journal of Education for Business*, 85:107–113.

Merton, R. C. (1972). An analytic derivation of the efficient portfolio frontier. *The Journal of Financial and Quantitative Analysis*, 7(4):1851–1872.

Nash, J. C. (2006). Spreadsheets in statistical practice—another look. *The American Statistician*, 60(3):287–289.

Nævdal, E. (2003). Solving Continuous-Time Optimal-Control Problems with a Spreadsheet. *The Journal of Economic Education*, 34(2):99–122.

Panko, R. R. and Ordway, N. (2008). Sarbanes-oxley: What about all the spreadsheets? *arXiv:0804.0797 [cs.SE]*, abs/0804.0797.

Wang, H., Padua, D., and Wu, P. (2015). Vectorization of apply to reduce interpretation overhead of r. *SIGPLAN Not.*, 50(10):400–415.

Wickham, H. (2010). A Layered Grammar of Graphics. *Journal of Computational and Graphical Statistics*, 19(1):3–28.

Xie, Y. (2013). animation: An r package for creating animations and demonstrating statistical methods. *Journal of Statistical Software*, 53(1):1–27.

Yalta, A. T. (2008). The accuracy of statistical distributions in Microsoft Excel 2007. *Computational Statistics & Data Analysis*, 52(10):4579 – 4586.

Zeileis, A. (2005). CRAN Task Views. *R News*, 5(1):39–40.

Zhang, C. (2015). Incorporating Powerful Excel Tools into Finance Teaching. *Journal of Financial Education*, 40:87–113.

A Deploying the Efficient Frontier Shiny App on Amazon Web Services

The following steps sets up the efficient frontier Shiny app on a Free Tier EC2 instance on Amazon Web Services (AWS).

A.1 Creating the instance

The basic setup takes only a couple of minutes:

- Create an Amazon account (or use an existing account)
- Log into the AWS console `https://console.aws.amazon.com/`
- Launch an EC2 Instance: choose the `Ubuntu Server 16.04 LTS` Amazon Machine Image (AMI), select a t2.micro instance type
- Create new key pair, download it as say `my-key.pem` and change the file permissions to give all rights only to the owner (Unix permission 400).
- From the AWS Console, determine the public DNS (hostname) of the instance say `my-ec2.amazonaws.com`

To connect to a running instance one needs to use secure shell (SSH) as:

```
ssh -i my-key.pem ubuntu@my-ec2.amazonaws.com
```

A.2 Installing R on the instance

We then proceed to install R and all requisite packages to install the `fPortfolio` R package:

```
# Some packages require newer version of R, so we use the RStudio
    repository
sudo apt-key adv --keyserver keyserver.ubuntu.com --recv-keys
   E298A3A825C0D65DFD57CBB651716619E084DAB9
sudo add-apt-repository 'deb [arch=amd64,i386] https://cran.
   rstudio.com/bin/linux/ubuntu xenial/'
sudo apt-get update
sudo apt-get upgrade
sudo apt-get install r-base
# install all packages required for fPortfolio and its
   dependencies
sudo apt-get install libcurl4-openssl-dev
sudo apt-get install libxml2-dev
sudo apt-get install coinor-symphony coinor-libsymphony-dev
   coinor-libcgl-dev
sudo apt-get install xorg-dev libglu1-mesa-dev
sudo apt-get install  glpk-utils libglpk-dev
```

We then run R as `root` (`sudo R`) and execute the following commands:

```
install.packages('fPortfolio')
install.packages('shiny')
```

A.3 Setting the Efficient Frontier App

We now copy the files `ui.R` and `server.R` to the EC2 instance using `scp` on our local machine:

```
scp -i my-key.pem /path/to/my/efficient-frontier/*.R
ubuntu@my-ec2.amazonaws.com:
```

This command will upload the files from the local machine to the home folder of user `Ubuntu`, but they can be moved into the right folder later.

Coming back to the SSH terminal, we execute the following commands on the EC2 instance:

```
sudo apt-get install gdebi-core
wget https://download3.rstudio.org/ubuntu-12.04/x86_64/shiny-
   server-1.5.1.834-amd64.deb
sudo gdebi shiny-server-1.5.1.834-amd64.deb
sudo mkdir /srv/shiny-server/efficient-frontier
sudo mv *.R  /srv/shiny-server/efficient-frontier/
sudo chmod 755 /srv/shiny-server/efficient-frontier/*
```

At this point, we can terminate the `ssh` connection to the EC2 instance. Now in the AWS Console (in the browser), navigate to Security Group, edit and add rule for inbound traffic. For `Type`, choose `Custom TCP Rule`, for `Port` select `3838` and for `Source`, accept the default `0.0.0.0/0, ::/0`.

A.4 Running the App

To run the app, we point the web browser to
`http://my-ec2.amazonaws.com:3838/efficient-frontier/`.

This completes the steps for setting up the app on Amazon AWS.