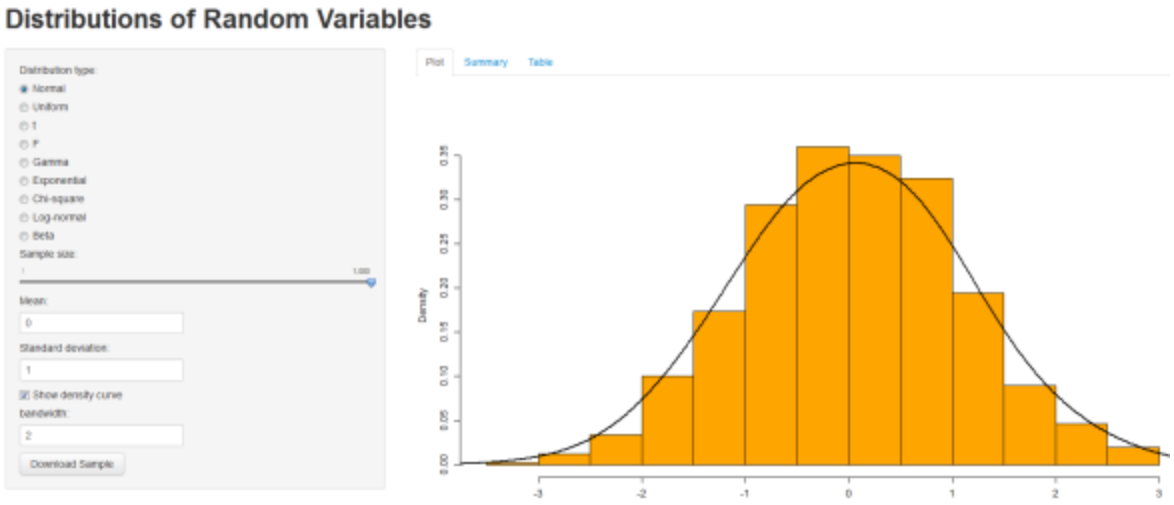
Introducing R Shiny web apps

Posted on **May 20, 2013** by **Matt Leonawicz**

Recently I've begun experimenting with the **R** Shiny package, which is a great way for interactively showcasing and sharing statistical analyses and results performed and generated in **R** on the web. Here I provide a simple example of how to use the shiny package to create and deploy an app to your browser using a local host. A Shiny app can also be hosted on a web server, but that is beyond the scope of this post.

Building upon a more basic app that appears in the RStudio Shiny tutorial, here we have an app that plots histograms of random samples drawn from any one of a set of common probability distributions. The user can select a distribution, a sample size, whether to overlay a smooth density curve, and only if shown, what bandwidth to use for the curve.



You can click the image to go to the app page.

This is the first of a four-part series where I add enhancements to the app in stages. Here are the other posts and corresponding versions of the app:
R sampling app version 2 [app version 2]
R sampling app version 3 [app version 3]
R sampling app version 4 [app version 4]

Apps can become much more complicated of course. The `shiny` package is a powerful tool for statisticians and data analysts who wish to share their work in an interactive fashion. Statistics is storytelling, and from start to finish – from the formulation of a problem or a question; through exploratory data analysis, looking at tables, charts, graphs and summary statistics; to the selection and application of appropriate advanced inferential methods; and finally the interpretation of results and dissemination of relevant information – a Shiny app can be used to tell the story interactively, in a way that is accessible to managers, clients, scientists, and anyone else who can interact with the app on a webpage but who may not

be statisticians or data analysts themselves. It is an excellent web presentation tool that allows you to share your work without being present and it permits clients to focus on what aspects of a project are most interesting and important to them.

An **R** Shiny app consists of two scripts, `ui.R` and `server.R`, the user-interface and server-side scripts, respectively, as well as any data or other files that may be needed by the app, depending on its complexity. In this example, we need no extra files. We will generate our own data as part of the app. Here is the `ui.R` script:

Ui.R script goes here

In the user-interface script you'll see the `shiny` package is loaded followed by some nested package functions. The purpose here is not to reproduce tutorials which already exist, but rather just to show an example. If you haven't explored Shiny before, I recommend you see the tutorial put together by the RStudio team and other online examples to get a better sense for how Shiny works, and the package documentation for a more thorough understanding of the package functions and examples of their varied uses. With at least some nominal exposure to those resources and some experience with **R**, it should be relatively straightforward to connect the code here to the behavior of the app and what it displays at the app image link above. The `ui.R` script is usually pretty simple and short even for fairly complex apps. The `server.R` script is the one that grows in size and complexity more linearly with your project. Here is the `server.R` script:

Server.R script goes here

In the `server.R` script, we run any **R** code that we need to up front. This is the one-time stuff. You run these commands when the app launches. The code may set up certain objects in your workspace. It may load a data set or a workspace (`.RData`) file. Basically, this is anything that you will only need to do one time in your **R** session and don't want to rerun wastefully every time a user changes some options on their screen.

After this, `shinyServer` is called. Within this function is where we achieve our reactivity. We use reactive expressions of various kinds to update the outputs displayed to the user based on the inputs the user selects. Reactive expressions are only evaluated when a user-initiated change to inputs is detected. These expressions then evaluated, and immediately update the outputs sent to the browser.

Things not to get caught up on. You may notice some odd commands at the beginning of this script, specifically the use of the `formals` function, and you may wonder why I appear to be assigning base **R** functions to new objects with altered default parameters. This is not a "Shiny thing". Ignore it. But if you must know, it's because in putting this app together, I found that the Shiny app did not behave well the way I had originally coded it. This turned out to be due to the fact that multiple **R** functions I was using, e.g. `rgamma` and `rexp`, or `rt` and `rchisq`, had formal arguments with the same name. Picking parameter values in the

app for one distribution, and then toggling over to another distribution in the browser which had a similarly named distribution parameter argument, would cause some annoying bugginess.

I got around this by making calls to my own wrapper functions which did not have this kind of argument name overlap. It's possible I don't need to do this any longer and it may have been due to the early version of Shiny, or my own lack of understanding of how Shiny worked at the time I was first experimenting with it. In any case, I got it to work. In more recent apps I have not encountered this kind of problem. Not to say that it does not remain an issue, but I have not been trying to recreate it. But if anyone can tell me what I may have been doing wrong or why this was an issue to begin with, please let me know. Moving along now…

For those of you who have dabbled in **R** Shiny a bit already, it is perhaps worth mentioning that in my experience I have had more success in terms of achieving complexity of reactivity and generalizability by using `uiOutput` in the `ui.R` script rather than specifying sidebar controls directly, in tandem with `renderUI` in the `server.R` script. I tend to define user controls for the sidebar in the latter script and have found so far that this style works better for my applications in general.

Now let's discuss launching an app locally. Just place your `ui.R` and `server.R` scripts in a directory together. Launch **R**. Make sure the `shiny` package is installed. Load the package. Then all you need to do is enter `runApp("C:/path/to/my/directory")` for example. **R** will launch your app in your default web browser. Done!

In my next post, R sampling app version 2 [app], I spiff up the plots a bit by adding probability distribution function (pdf) annotations using `plotmath` expressions. In total there are four versions of this sampling app, one building on the next, so that you can see various elements added in stages. I didn't have it all planned out when I began, but that's sometimes how it goes. I'll have more posts regarding **R** Shiny apps I've developed for various SNAP projects, to be hosted on our website like the one above. If you are interested in hosting your own apps on a web server as well, where users can interact with your apps on a webpage without having to deal with **R** or anything else, you'll have to follow instructions like these.