Building interactive dashboards

Aroon T. Chande[1,2], Lavanya Rishishwar[1,3,*]

[1] *Applied Bioinformatics Laboratory, Atlanta, GA, USA*
[2] *Seagen Inc., Seattle, WA, USA*
[3] *School of Biological Sciences, Georgia Institute of Technology, Atlanta, GA, USA*

(Dated May 18, 2021)

This material was designed for and delivered at the Georgia Institute of Technology's Quantitative Biosciences (QBioS) 2021 Hands-On Modeling Workshop (https://workshop2021.qbios.gatech.edu), organized by Professors Joshua Weitz and James C. Gumbart and co-organized with the Fall 2020 QBioS Cohort: Lynn Jin, Ethan Wold, Leo Wood, Disheng Tang, Aradhya Rajanala, Aaron Pfennig, and Tucker Lancester.

The last few years have witnessed a rise in the availability and use of dashboards. While used in various business operations for several years, the availability of COVID-19-related dashboards demonstrated their ability to easily communicate complex information to a much broader audience. In this session, we will begin with introducing you to dashboards and why are they useful, and then we will discuss how you can build your own dashboards using R Shiny. We will work towards building a dashboard akin to our Shield Immunity dashboard (https://shieldimmunity.biosci.gatech.edu/; Weitz et al. Nat Med. 26(6): 849-854).

I. Dashboards: Basics and applications

Dashboards are powerful analytical tools that can help their users keep track of key metrics. They are used in a wide variety of fields such as business intelligence/analytics, stock market, biological sciences, epidemiology, IT resource management, and so on. Dashboards help users stay informed through visual means, e.g., summarizing the prevalence of various SARS-CoV-2 lineages in your state/country over the last few weeks or assessing the risk of exposure to a COVID-19 positive individual if you are attending an event in your city.

While there is not any single list of criteria that defines an application as a dashboard, a typical dashboard has the following elements: (a) it is accessible through a web-browser, (b) it has a graphical element visualizing some key metric(s), (c) it has some level of interactivity that helps users make sense of the data, and (d) it is customizable.

Often, dashboards are intended to be the end-product of arduous data modeling and wrangling efforts. They provide their intended users/viewers a "5-minute pitch" of analysis results and allow them to interact with the results to better understand the effects of different factors. Such interfaces are incredibly useful in the age of data science where we are frequently dealing with data deluge. Data generation and collection has become easier and cheaper, thanks to significant advances in technology and availability of open data repositories. Relatively newer technologies, such as cloud infrastructures and rapid-web development libraries, have lowered the entry barrier to develop interactive dashboards.

*Electronic address: LRishishwar@ihrc.com; URL: https://abil.ihrc.com

There are several technologies available for building dashboards including, but not limited to, R Shiny, Python Dash, Node.js, and Tableau; each technology has their own pros and cons. A skilled developer can create identical applications in all these technologies. Ultimately, the choice between different technologies comes down to what resources you have access to, what your project budget is, and who is doing the development.

II. Building dashboards with R Shiny

A. Introduction to R and Shiny

R is one of the several, popular languages used for bioinformatics, statistical analysis, and data visualization. It is a high-level language, implying that it provides the user the ability to perform complex operations (from a computational point of view) with simple commands. There are several factors that contribute to its popularity: ease of learning and use, expansive set of packages and user base, ease of generating visualizations, and ease for interactive dashboard development. We will focus on its use for developing interactive dashboards.

Interactive dashboards within R are created through a library/package/module called Shiny. Shiny is a framework that allows developers to rapidly develop and deploy complex interactive applications purely in R. While knowledge of HTML, CSS, and JavaScript is very handy, Shiny makes it so that the developer is not required to understand these technologies to deploy their application. Shiny is great for developers of all experience levels. Its high-level syntax and relatively simpler structure allow novice developers to get started much more easily than other technologies, and its adherence to strong software engineering principles provides experienced users with a formal structure to develop their projects.

B. Simplified Shiny architecture

To help developers rapidly develop and deploy applications, Shiny abstracts much of the complexity behind simple functions and classes. Like most web technologies, Shiny has a front-end and back-end system. The front-end system, called "R ui" in Figure 1, is responsible for generating the webpage that the user sees. From a dashboard perspective, this webpage will typically contain some text and visual elements (e.g., a graph), and some interactive element that allow the user to interact with the website. The "ui" provides the mechanisms for interacting with the website, but the reactive response is handled in the back-end "R server" function.

We will explore the behavior of these two components through a series of incremental codebases.

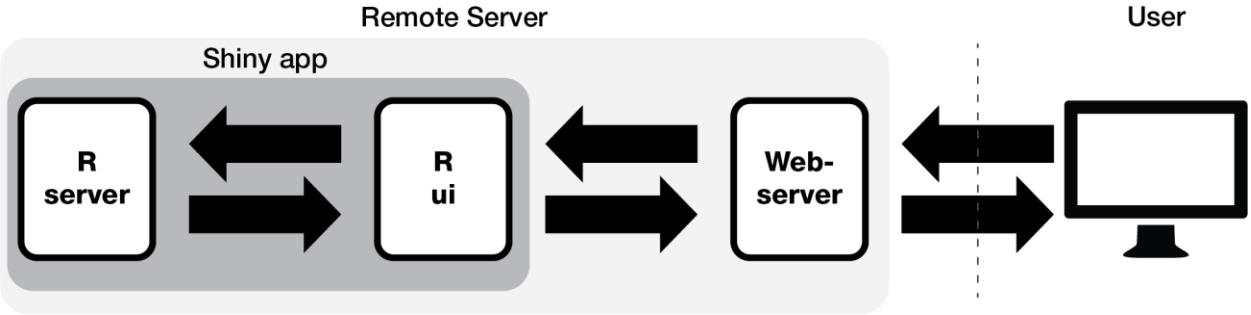


FIG. 1: Simple schematic of the R Shiny architecture. An R Shiny application can be broken down into two components: "server" and "ui". The "server" is intended to handle responses to interactivity and plotting. The "ui" is intended to provide the webpage and an interface for interactivity.

C. Making your first Shiny application

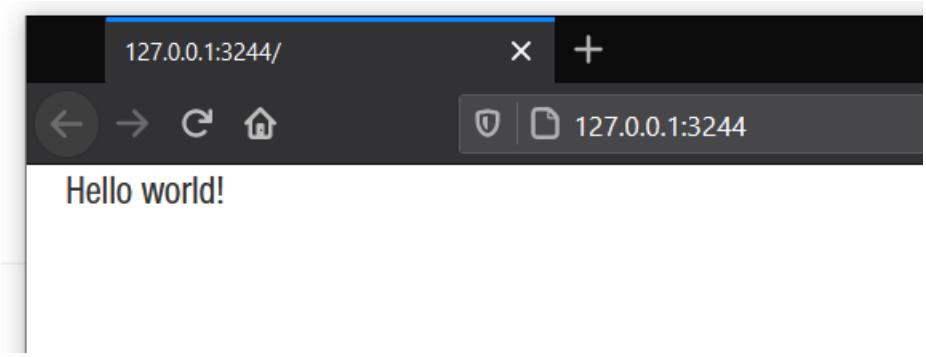
Our first codebase will be a "Hello world" example. This example will introduce you to the basic structure of a Shiny code and print "Hello world" on the front-end ui (coded through the fluidPage() function).

Shiny application #1: Hello world

A simple "Hello world" application

```
library(shiny)

ui <- fluidPage(
  "Hello world!"
)

server <- function(input, output){}

if(interactive()){
  shinyApp(ui, server)
}
```

Expected output:



The first line library(shiny) loads the Shiny library. The next piece of code ui <- fluidPage("Hello world") supplies the webpage markup text that is displayed by Shiny. The server function takes two arguments: input and output, however, this example contains no processing instructions. Finally, the results from fluidPage() (stored in the ui variable) and the server function are passed to the shinyApp function which creates our simple application.

In our next example, we will add an interactive element to our webpage: a slider that goes from 0 to 20 in steps of 0.2. We will define the slider to rest at a default value of 10. The slider will have the label "Strength of shielding" placed above it. We will call the slider "alpha"; this name will be used to retrieve

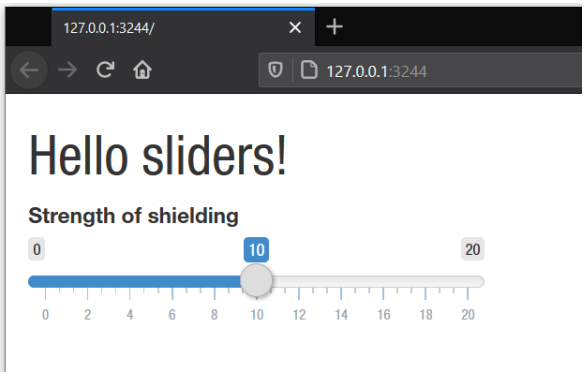
the slider position value each time a user interacts with it. Finally, we will also put a heading of level 1 (largest heading=h1) "Hello sliders!" on top of the page. This is done using the h1("Hello sliders!") line.

Shiny application #2: Hello sliders

An application with a simple slider

```
library(shiny)

ui <- fluidPage(
  h1("Hello sliders!"),
  sliderInput(inputId = "alpha",
              label = "Strength of shielding",
              value = 10,
              min = 0,
              max = 20,
              step = 0.2)
)

server <- function(input, output){}

shinyApp(ui, server)
```

Expected output:



Now having familiarized ourselves with fluidPage(), let us focus our attention to the server function. The next codebase will display a normal distribution on the screen. Comments in the code are shown in green and start with the # symbol.

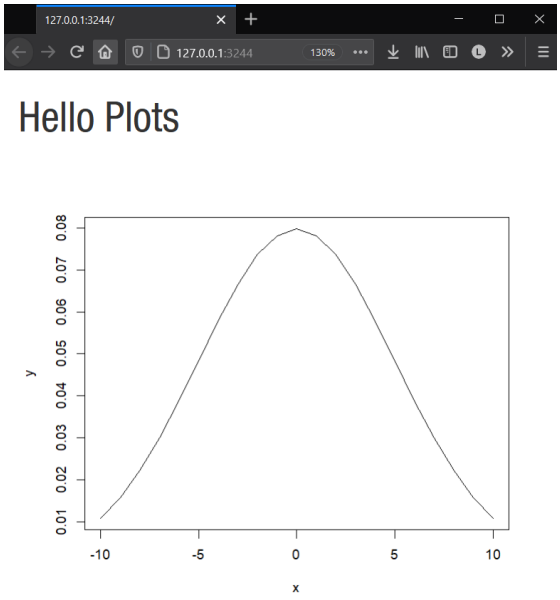
Shiny application #3: Hello plots

An application that displays a normal distribution.

```
library(shiny)

ui <- fluidPage(
  h1("Hello Plots"),
  plotOutput("plot")
)

server <- function(input, output){
  output$plot <- renderPlot({
    x = seq(-10, 10, 1)  # Generates the sequence -10, -9, -8, ..., 8, 9, 10
    y = dnorm(x, mean = 0, sd = 5) # Generates a normal density distribution
                                   # with x, mean of 0 and standard dev = 5
    plot(x, y, type = "l") # Plot x and y as a line plot (type="l")
  })
}

shinyApp(ui, server)
```

The plot is encapsulated within the renderPlot() function. The output of renderPlot() (the plot we want to display) will be attached to the "output$plot" variable. "output" is a special variable that contains the output from the server function. Were we to supply any inputs in this example, they will be stored in the "input" variable of the server function. "$plot" attaches a (sub-)variable to the "output" variable which is accessible through fluidPage() when it is called in the plotOutput("plot") line.

Expected output:



This brings us to how the "input" area and "output" area behave within Shiny. All the text that we supply in the beginning of our examples are placed as is on the website. The "sliderInput" that we used in Shiny application #2 is a type of "input". When a user interacts with the slider, it changes the slider's value. After each interaction, the new value is passed by the ui to the server function. The server function reruns its instructions and returns the output plot as part of the "output" variable. This variable is utilized within fluidPage() to replot the output plot.

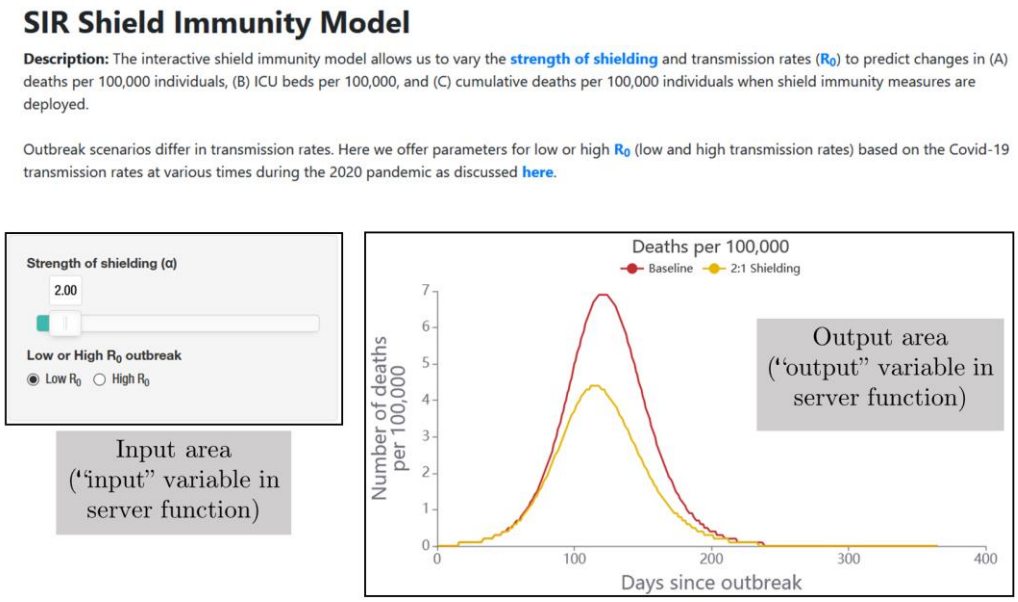


FIG. 2: Elements controlled by input and output variables within the server function.

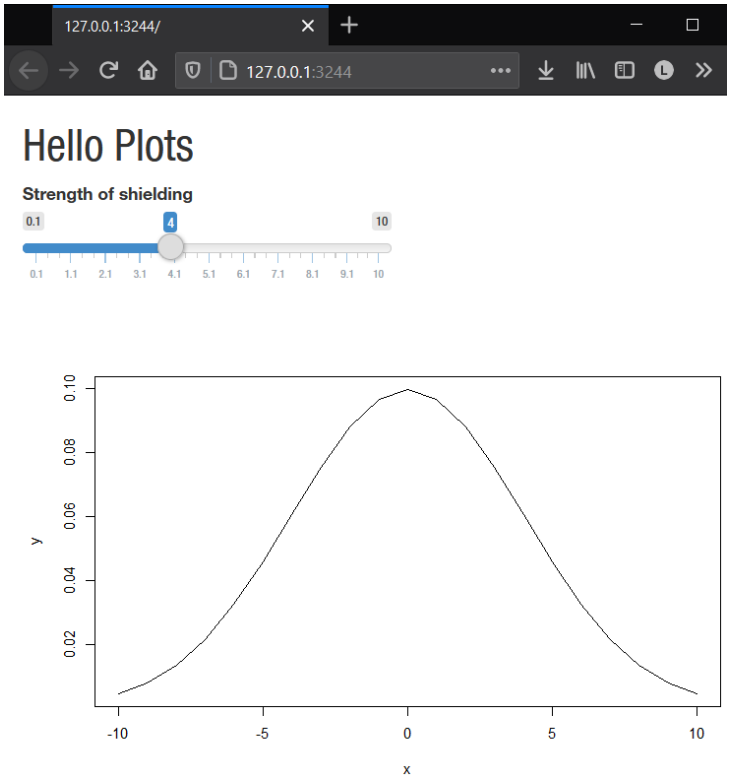
Let us see how this works with an example.

Shiny application #4: Hello interactions

An application that combines an interactive slider with a plot

```
library(shiny)

ui <- fluidPage(
  h1("Hello Plots"),
  sliderInput(
    inputId = "alpha",
    label = "Strength of shielding",
    value = 1,
    min = .1,
    max = 10,
    step = 0.1
  ),
  plotOutput("simulation")
)

server <- function(input, output){
  output$simulation <- renderPlot({
    x = seq(-10, 10, 1)
    y = dnorm(x, mean = 0, sd = input$alpha)
    plot(x, y, type = "l")
  })
}

shinyApp(ui, server)
```

Expected output:



D. Developing complex Shiny applications

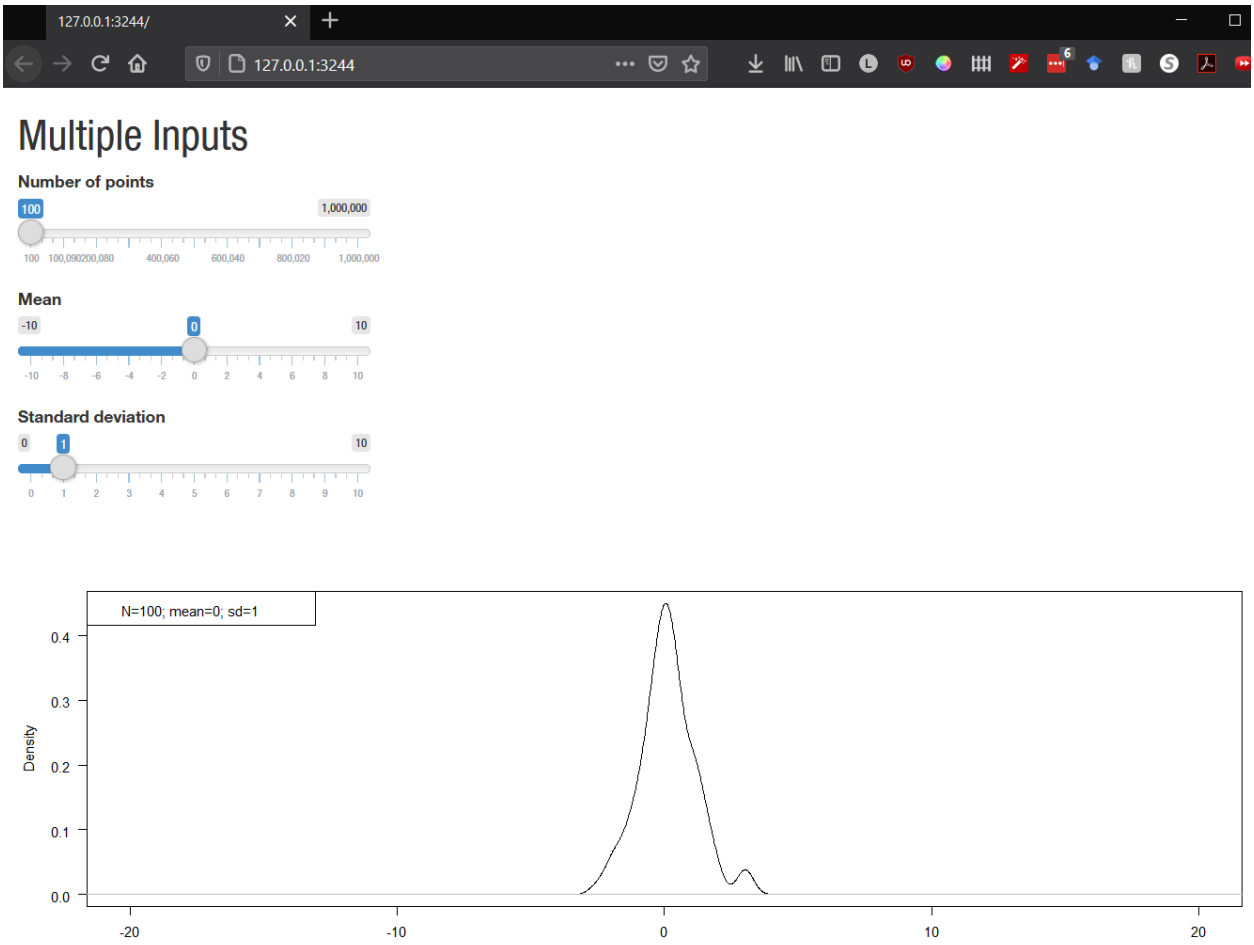
Having explored the basic building blocks and how they are used to create simple Shiny applications, we will now create more complex applications by repeating the components we have explored previously. The following application employs three sliderInput(), one for number of points, one for mean, and one for standard deviation. We also make our plot a little more complex by adding a legend to the plot on the "topleft".

Shiny application #5: More interactions

An application with multiple input sliders affecting the output plot

```
library(shiny)

ui <- fluidPage(
  h1("Multiple Inputs"),
  sliderInput(
    inputId = "num_pts",
    label = "Number of points",
    value = 100,
    min = 100,
    max = 1000000,
    step = 10
  ),
  sliderInput(
    inputId = "mean",
    label = "Mean",
    value = 0,
    min = -10,
    max = 10,
  ),
  sliderInput(
    inputId = "sdev",
    label = "Standard deviation",
    value = 1,
    min = 0,
    max = 10,
    step = 0.1
  ),
  plotOutput("simulation")
)

server <- function(input, output){
  output$simulation <- renderPlot({
    # Randomly draw total of num_points from a normal distribution
    sim_dist = rnorm(n = input$num_pts, mean = input$mean, sd = input$sdev)
    # Plot the density distribution of the points
    plot(density(sim_dist), las=1, main = NA, xlab = NA, xlim = c(-20, 20))
    # Display to users what parameters we used for the plot
    legend("topleft", legend = paste0(
      "N=", input$num_pts, "; mean=", input$mean, "; sd=", input$sdev),
      cex - 0.8
    )
  })
}

shinyApp(ui, server)
```

Expected output:



This output achieves the desired behavior of multiple inputs, but it does not look aesthetically appealing. Let us rearrange the inputs so that they appear in a single row. To do this, we will use the fluidRow() and column() functions together. A fluidRow() lets us define a single row in the webpage. Each row can have an integer width between 1-12, with the constraint that the summation of all widths across all columns should equal 12. This constraint comes from the bootstrap layout, which R Shiny utilizes. The bootstrap layout provides fluidity to the webpage so that it displays nicely across all devices and window sizes.

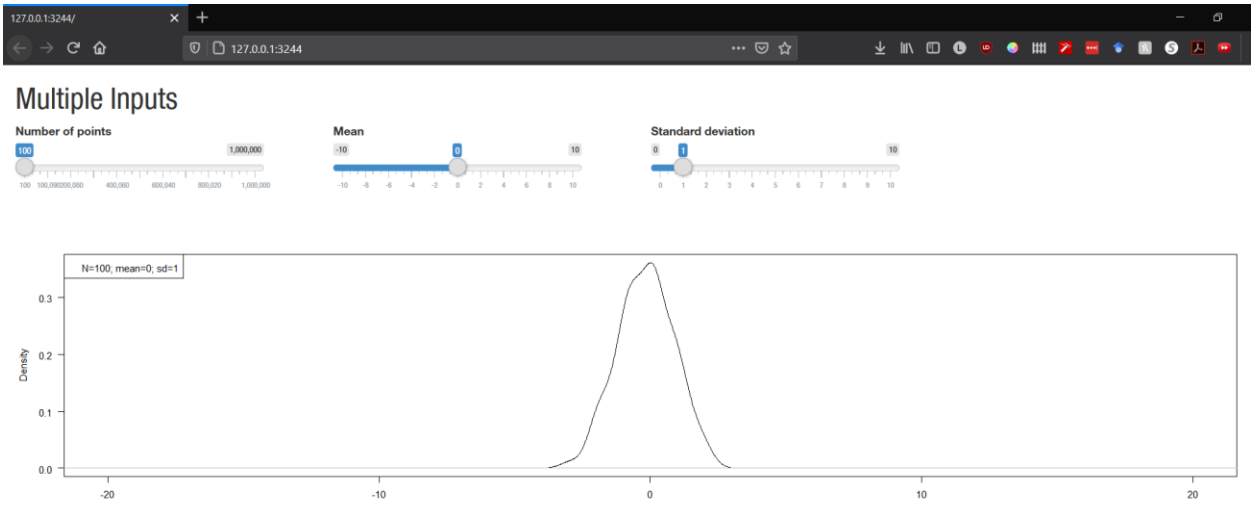
This addition to our code is shown below:

Shiny application #6: Organizing UI

An application that organizes different components into a layout

```
library(shiny)

ui <- fluidPage(
  h1("Multiple Inputs"),
  fluidRow(
    column(width = 3,
      sliderInput(
        inputId = "num_pts",
        label = "Number of points",
        value = 100,
        min = 100,
        max = 1000000,
        step = 10
      )
    ),
    column(width = 3,
      sliderInput(
        inputId = "mean",
        label = "Mean",
        value = 0,
        min = -10,
        max = 10,
      )
    ),
    column(width = 3,
      sliderInput(
        inputId = "sdev",
        label = "Standard deviation",
        value = 1,
        min = 0,
        max = 10,
        step = 0.1
      )
    )
  ),
  plotOutput("simulation")
)

server <- function(input, output){
  output$simulation <- renderPlot({
    sim_dist = rnorm(n = input$num_pts, mean = input$mean, sd = input$sdev)
    plot(density(sim_dist), las=1, main = NA, xlab = NA, xlim = c(-20, 20))
    legend("topleft", legend = paste0(
      "N=", input$num_pts, "; mean=", input$mean, "; sd=", input$sdev),
      cex - 0.8
    )
  })
}

shinyApp(ui, server)
```

Expected output:



E. Adding navigation bars to our applications

When your application becomes more complex with multiple graphs, you should consider reorganizing it as either a multi-page or multi-tab application. This is beneficial from both a visual and code-organization standpoint.

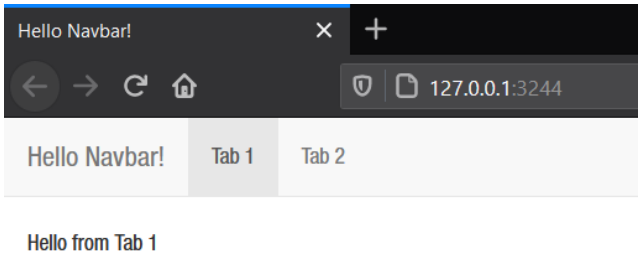
If you desire to create a multi-tab application, Shiny provides you with an in-built function that will handle the multiple tabs and add a navigation bar. This function, called navbarPage(), is used like fluidPage() that we explored previously. The difference this time is that we place our tab-specific content inside tabPanel() functions:

Shiny application #7: Hello Navbar (navigation bar)

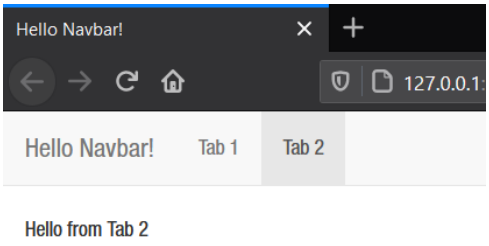
An application with multiple tabs

```
library(shiny)

ui <- navbarPage(
  title = "Hello Navbar!",
  id = "navbar",
  tabPanel(
    "Tab 1",
    "Hello from Tab 1"
  ),
  tabPanel(
    "Tab 2",
    "Hello from Tab 2"
  )
)

server <- function(input, output){}

shinyApp(ui, server)
```

Tab 1:

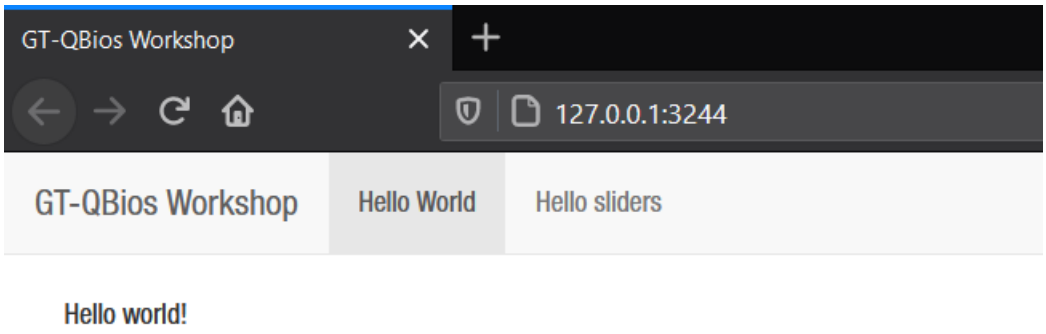


Tab 2:

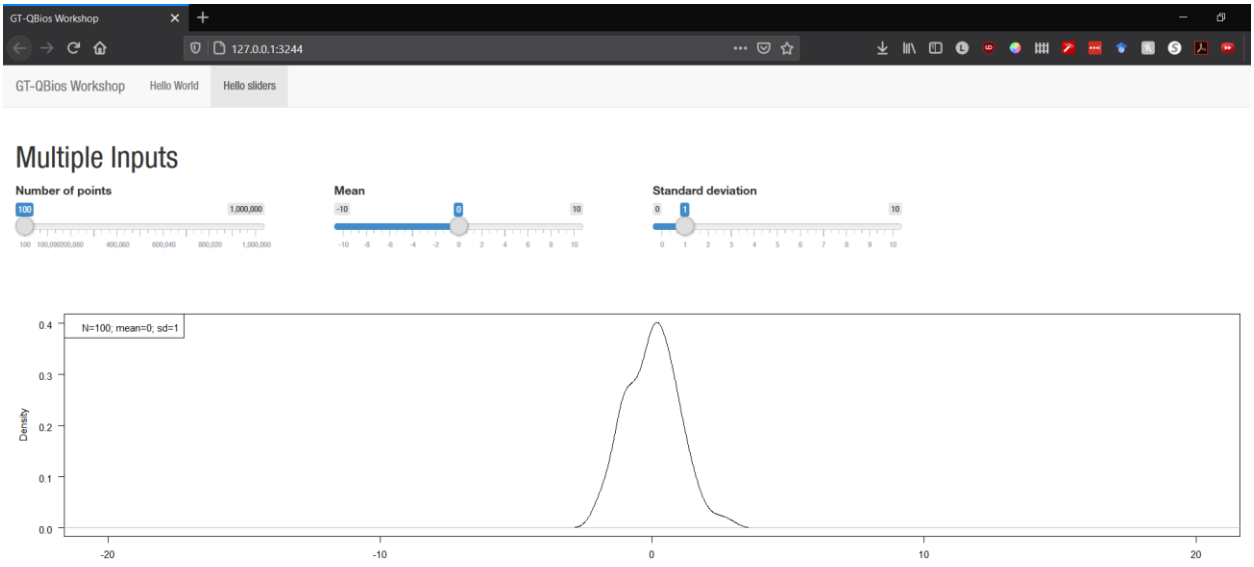


We can re-use our code from previous exercises to have the first tab contain text ("Hello world!" will do for now) and the second tab contain the complex multi-input application. This is achieved by using fluidRow() within tabPanel().

Tab 1:



Tab 2:



Shiny application #8: Navbar app

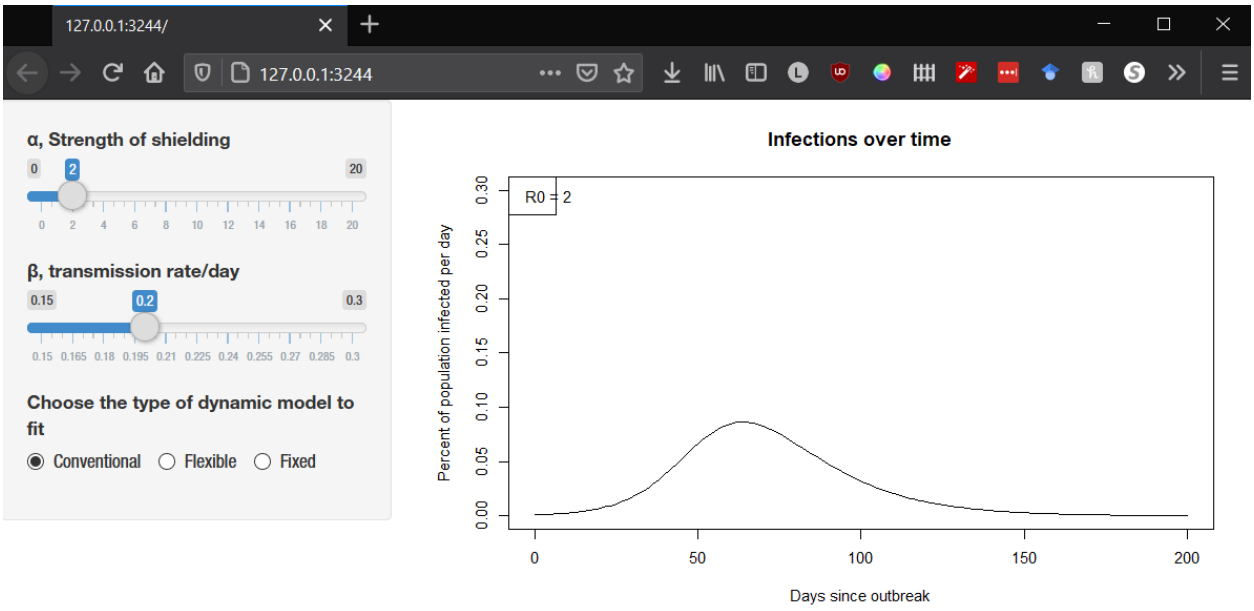
Combining applications #6 and #7

```
library(shiny)

ui <- navbarPage(
  title = "GT-QBios Workshop",
  id = "navbar",
  tabPanel(
    "Hello World",
    fluidPage(
      "Hello world!"
    )
  ),
  tabPanel(
    "Hello sliders",
    h1("Multiple Inputs"),
    fluidRow(
      column(width = 3,
             sliderInput(
               inputId = "num_pts",
               label = "Number of points",
               value = 100,
               min = 100,
               max = 1000000,
               step = 10
             )
      ),
      column(width = 3,
             sliderInput(
               inputId = "mean",
               label = "Mean",
               value = 0,
               min = -10,
               max = 10,
             )
      ),
      column(width = 3,
             sliderInput(
               inputId = "sdev",
               label = "Standard deviation",
               value = 1,
               min = 0,
               max = 10,
               step = 0.1
             )
      )
    ),
    plotOutput("simulation")
  )
)

server <- function(input, output){
  output$simulation <- renderPlot({
    sim_dist = rnorm(n = input$num_pts, mean = input$mean, sd = input$sdev)
    plot(density(sim_dist), las=1, main = NA, xlab = NA, xlim = c(-20, 20))
    legend("topleft", legend = paste0(
      "N=", input$num_pts, "; mean=", input$mean, "; sd=", input$sdev),
      cex - 0.8
    )
  })
}


shinyApp(ui, server)
```

F. Advanced code comprehension challenge: building an app for a SIR model

The pieces of code we have covered so far will help you get started in building your own dashboards. Keep in mind, we have barely scratched the surface in this short session. There are many of complex applications that you can develop with R Shiny. As na example of what more you can do with Shiny, we leave you here with the following application. From a code-standpoint, this application has several concepts that we were not able to go over in this session. We leave this optional section of the material to the reader to explore more on their own time and see if they can decipher how this application behaves. The full-scale application can be accessed from: https://shieldimmunity.biosci.gatech.edu/models-poc.html.

Expected output from the code below:



Shiny application #9: SIR app

A complex SIR application with layouts, sliders, radio buttons and output

```
packages = c("shiny", "deSolve")
install.packages(setdiff(packages, rownames(installed.packages())))
library(shiny)
library(deSolve)

source("https://raw.githubusercontent.com/appliedbinf/GaTech-Weitz-ShieldImmunity
/master/sir/pocmodel.R")

ui <- sidebarLayout(
  sidebarPanel(
    sliderInput(
      inputId = "alpha",
      label = HTML("&alpha;, Strength of shielding"),
      min = 0,
      max = 20,
      value = 2,
      step = .1
    ),
    sliderInput(
      inputId = "beta",
      label= HTML("&beta;, transmission rate/day"),
      min = 0.15,
      max = .3,
      value = .2,
      step=.005
    ),
    radioButtons(
      "model",
      "Choose the type of dynamic model to fit",
      choices = c(
        "Conventional" = "core",
        "Flexible" = "soft",
        "Fixed" = "hard"
      ),
      selected = "core",
      inline = T
    )
  ),

  mainPanel(fluidPage(
    fluidRow(htmlOutput('descText')),
    fluidRow(div(plotOutput("p_model")))
  ))
)

server <- function(input, output, ...){
  pars = list()
  pars['gamma'] = 1 / 10
  y0 = c('S' = 0.999, 'I' = 0.001, 'R' = 0) # Supplied by JSW
  t = 0:200 # Day index, 0 to 200
  output$p_model <- renderPlot({
    pars['alpha'] = input$alpha
    pars['beta'] = input$beta
    pars['R0'] = pars$beta / pars$gamma
    par_model = input$model
    # Generate SIR from user input
    model_output = SIR_shield(t, y0, shield = par_model, pars)
    plot(x = seq(0, 200), y=model_output$I, type = "l",
         main= "Infections over time",
         xlab = "Days since outbreak",
         ylab = "Percent of population infected per day",
         ylim = c(0, 0.3))
    legend("topleft",
           legend = paste0("R0 = ", pars['R0'])
    )
  })
}

shinyApp(ui, server)
```