

Tutorial and Laboratories

11464 – 11524G AR/VR for Data Analysis and Communication

Week 3

Introduction

In this computer lab we will continue practice data visualisation using **interactive user interfaces and plots**, which are a great tool for live presentations and very relevant to the communication of results.

It is assumed that you are now familiar with base R to create plots, how to subset data from data frames, how to enter data or read data files, how to plot data; topics that were covered in our previous tutorials. It is recommended to complete the tutorials in week 2-3 before attempting this tutorial.

Skills Covered in this tutorial include:

- Shiny package.
- Interactive user interfaces.
- Data visualisation in Shiny.

Note: Do not copy-paste the commands. As you type each line, you will make mistakes and correct them, which make you think as you go along. Remember, that the objective is that you understand the commands and master the concepts, so you can reproduce their principles on your own later.

For this tutorial we will use the iris dataset in some of the examples, the dataset is available in R. We should be already familiar with the dataset, if you need more information about it, please refer to the Tutorial from week 9. To load the dataset, do the following:

```
# load dataset
data("iris")
# explore dataset
View(iris)
```

```
# create vectors for plots
x <- iris$Sepal.Length
y <- iris$Petal.Length
z <- iris$Sepal.Width
```

1. Interactive user interfaces (UI) and plots (also 3D plots) with *shiny*

Shiny is a package that allows to build interactive web applications (apps) with R. This package makes responsive (real time) applications that have outputs change instantly as the user modify the inputs, without reloading the data or function. Everything can be built in R without the need to use JavaScript to build. As a reference, you can use the Shiny cheat sheet available in Canvas.

In general, a shiny app is a web page (or UI) connected to a computer with a live R session (or server). Users can change the parameters in the “live” app, these changes are immediately reflected in the UI due to the real-time update made by the server. The first thing we need to do is to install the Shiny package, please to the following.

```
# install packages
install.packages("shiny")
# load package
library(shiny)
```

In order to create a Shiny app, we need to create two elements: the *UI* and the *Server*. The *UI* element includes all R functions that assemble an HTML user interface for your app, and controls the layout and appearance of the app. The *Server* is a function with instructions on how to build and rebuild the R objects displayed in the *UI*. Once we have created these two elements, we need to integrate them with the *shinyApp()* function to create the app. Below is a template that can be used when we need to create shiny apps, you can preview the app by running the code at the command line.

```
ui <- fluidPage( )           ← It needs: 1) a type of layout. 2) a type of input. 3) a type of output.
server <- function(input, output){ } ← It needs a render function.
shinyApp(ui = ui, server = server)
```

Exercise 1. Shiny apps can be included in a single script (e.g., myApp.R). The script is stored in a directory (e.g., workingdirectory/) and the app can be executed with the *runApp()* function. Let's create a simple app, so we can understand how it works. First, open a new script and save it as *app1.R*; it is a good practice to place this app in its own folder. The following example presents a sidebar UI (please refer to the cheat sheet). Please try the following (a short description of what each function does is on your right).

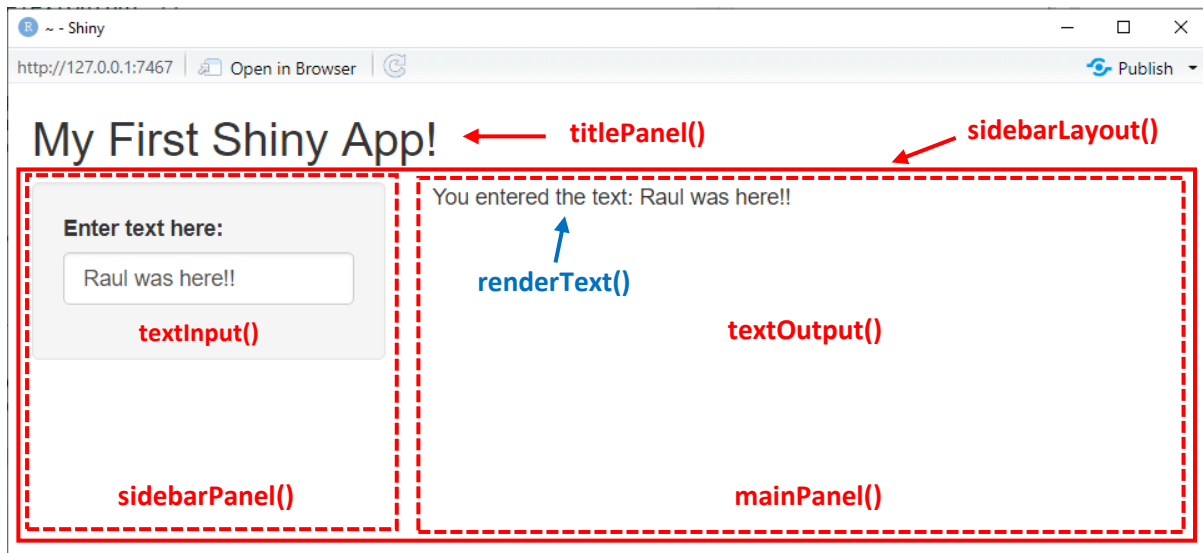
```
ui <- fluidPage(
  titlePanel("My First Shiny App!"),
  sidebarLayout(
    sidebarPanel(
      textInput("myText", "Enter text here:",
        "Raul was here")
    ),
    mainPanel(
      textOutput("niceTextOutput")
    )
  )
)

server <- function(input, output){
  output$niceTextOutput<-renderText({
    paste("You entered the text:", input$myText)
  })
}

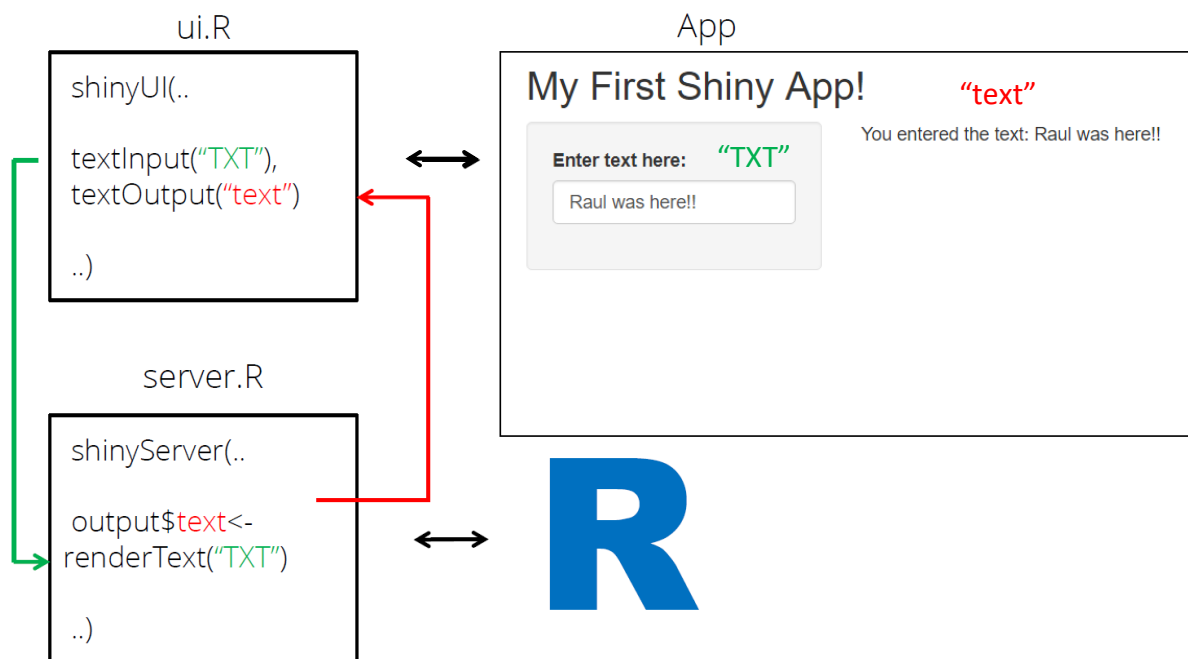
shinyApp(ui = ui, server = server)
```

sidebarLayout() – create a layout with a sidebar (1/3 of space area) and main area (2/3 space area).
 sidebarPanel() – defines the sidebar area.
 textInput () – defines the type of input showed by the UI.
 mainPanel () – defines the main area.
 textOutput () – defines the type of output showed by the UI.
 "niceTextOutput" - name of variable that stores the output and to be used for the server.
 renderText() – type of graphic displayed as output.
 render*() and *Output() functions work together to add R output to the UI.
 input\$<inputId> – access the current value of an input object.
 output\$<outputId> – access the developed output of an output object.
 shinyApp() – combines both the UI and the server into an app.

Below is the resulting user interface produced by the code above. The image also contains the name of the functions related to each particular visual output.




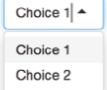

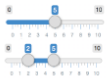

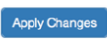

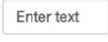


In the previous example we could see the interaction between both the *ui* and the *server*. On the *ui* side, we have the input (“myText”) that captures what the user typed and the output (“niceTextOutput”) that displays (on the output side of the *ui*) the string of characters typed by the user. On the server side, we have the input variable (“myText”) that serves as input for any operation within the *server*; we also have the output variable (“niceTextOutput”) that captures any operation performed within the server (all operations are processed in your R environment), this variable is internally updated to reflect any changes in the *ui*’s output. The following image summarises this relationship.



1.1 Inputs withing the UI framework

In the previous example, we used a `textInput()` input to capture data from the user. Shiny allows to build different types of inputs that serve different purposes. The figure below exhibits the different input functions available in the Shiny package.

Input	Description	Input	Description
	<code>actionButton(inputId, label, icon, ...)</code>		<code>numericInput(inputId, label, value, min, max, step)</code>
Link	<code>actionLink(inputId, label, icon, ...)</code>		<code>passwordInput(inputId, label, value)</code>
<input checked="" type="checkbox"/> Choice 1	<code>checkboxGroupInput(inputId, label, choices, selected, inline)</code>	<input checked="" type="radio"/> Choice A	<code>radioButtons(inputId, label, choices, selected, inline)</code>
<input checked="" type="checkbox"/> Choice 2		<input type="radio"/> Choice B	
<input type="checkbox"/> Choice 3	<code>checkboxInput(inputId, label, value)</code>	<input type="radio"/> Choice C	<code>selectInput(inputId, label, choices, selected, multiple, selectize, width, size) (also selectizeInput())</code>
<input checked="" type="checkbox"/> Check me			
	<code>dateInput(inputId, label, value, min, max, format, startview, weekstart, language)</code>		<code>sliderInput(inputId, label, min, max, value, step, round, format, locale, ticks, animate, width, sep, pre, post)</code>
	<code>dateRangeInput(inputId, label, start, end, min, max, format, startview, weekstart, language, separator)</code>		<code>submitButton(text, icon)</code> (Prevents reactions across entire app)
	<code>fileInput(inputId, label, multiple, accept)</code>		<code>textInput(inputId, label, value)</code>

Exercise 2. Now that we have seen some of the different inputs, let's use some of these to create a *ui* with multiple inputs and outputs in a single *ui*. This *ui* will capture the user's input and then will visualise the inputs in the output part of the *ui*. Please try the following.

```
ui <- fluidPage(
  titlePanel("My Second Shiny App!"),
  sidebarLayout(
    sidebarPanel( # INPUTS
      textInput("myTextInput", "Enter text here:"),
      numericInput("myNumberInput",
        "Select a number:",
        value = 50, min = 0, max = 100, step = 1),
      selectInput("mySelectInput",
        "Select from the dropdown:",
        choices = LETTERS[1:10])),
    mainPanel( # OUTPUTS
      h4("Using HTML in Shiny"),
      p("This is a paragraph of text included in our panel.",
        strong("This text will be in bold.")),
      textOutput("niceTextOutput"),
      textOutput("niceNumberOutput"),
      textOutput("niceSelectOutput"))
  )
)

server <- function(input, output){
  output$niceTextOutput<-renderText({
    paste("You entered text: ", input$myTextInput)})
  output$niceNumberOutput<-renderText({
    paste("You selected the number: ", input$myNumberInput)})
  output$niceSelectOutput<-renderText({
    paste("You selectedoption:", input$mySelectInput)})
}

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

My Second Shiny App!

Enter text here:

Select a number:

Select from the dropdown:

Using HTML in Shiny

This is a paragraph of text included in our panel. **This text will be in bold.**

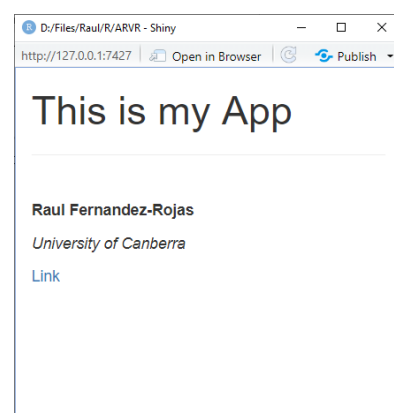
You entered text:
 You selected the number: 50
 You selected option: A

As you can see in the previous example, a Shiny app will only ever do the minimal amount of work needed to update the output controls that you can currently see. In Shiny, code is only run when needed depending on how inputs and outputs are connected. In the previous example, we connect an input to an output whenever the output accesses the input. Therefore, *niceTextOutput* will need to be recomputed only when *myTextInput* is changed. This kind of interaction is often described as *niceTextOutput* has a reactive dependency on *myTextInput*.

Another interesting feature of Shiny is that an app's *ui* is actually an HTML (Hypertext Markup Language) document. However, you don't need to use HTML tags, Shiny include a series of equivalent function. These equivalent functions create an HTML tag that you can use to layout your Shiny app. If you are not familiar with HTML tags, you can access a glossary of Shiny HTML tags here: <https://shiny.rstudio.com/articles/tag-glossary.html>

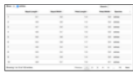

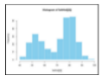

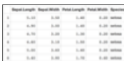

Exercise 3. Let's try to use some of the HTML elements to create a website. In this example, you will use h1 tags to define the size of the headers, p tags to define paragraphs, and a tags to add links. Please do the following.

```
ui <- fluidPage(
  tags$h1("This is my App"),
  tags$hr(),
  tags$br(),
  tags$p(strong("Raul Fernandez-Rojas")),
  tags$p(em("University of Canberra")),
  tags$a(href="https://www.canberra.edu.au/", "Link")
)
server <- function(input,output){}
shinyApp(ui=ui,server=server)
```



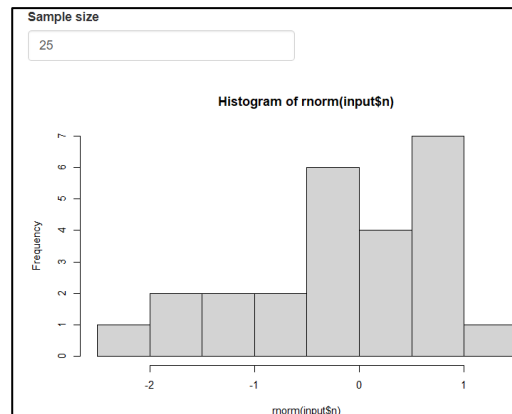
1.2 Outputs withing the UI and Server framework

So far, we have just output text, Shiny also allows us to output graphics, data, and images. We have to define the output in the *ui* and the *server* scripts using different functions. There are different types of outputs in both *ui* and *server* scripts, each type of output in the *server* (*render*()*) has a corresponding type of output (**Output()*) in the *ui* side. Below are the most common types of outputs and their counterpart in the *ui/server* script.

	DT::renderDataTable (expr, options, callback, escape, env, quoted)	works with	dataTableOutput (outputId, icon, ...)
	renderImage (expr, env, quoted, deleteFile)		imageOutput (outputId, width, height, click, dblclick, hover, hoverDelay, hoverDelayType, brush, clickId, hoverId, inline)
	renderPlot (expr, width, height, res, ..., env, quoted, func)		plotOutput (outputId, width, height, click, dblclick, hover, hoverDelay, hoverDelayType, brush, clickId, hoverId, inline)
	renderPrint (expr, env, quoted, func, width)		verbatimTextOutput (outputId)
	renderTable (expr, ..., env, quoted, func)		tableOutput (outputId)
foo	renderText (expr, env, quoted, func)		textOutput (outputId, container, inline)
	renderUI (expr, env, quoted, func)		uiOutput (outputId, inline, container, ...) & htmlOutput (outputId, inline, container, ...)

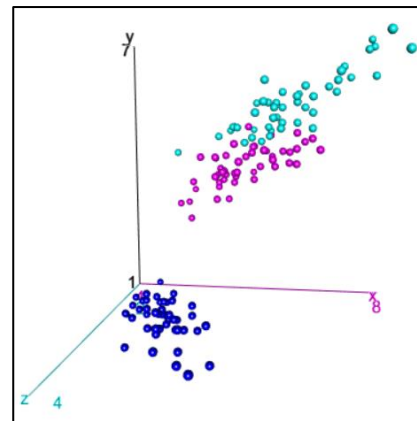
Exercise 4. Let's try a different type of ui, a histogram plot. In this example, we will have a `numericInput()` as an input in our UI. Try different values for each parameter and observe the results. Please refer to the cheat sheet and identify the functions that we use.

```
ui <- fluidPage(
  numericInput(inputId = "n",
    "Sample size", value = 25),
  plotOutput(outputId = "hist")
)
server <- function(input, output) {
  output$hist <- renderPlot({
    hist(rnorm(input$n))
  })
}
shinyApp(ui = ui, server = server)
```



Exercise 5. Now let's try a 3D example using the iris dataset, change the values of the parameters and observe the results.

```
library(rgl)
library(car)
ui <- fluidPage(
  rglwidgetOutput("plot", width = 800, height = 600)
)
server <- (function(input, output) {
  output$plot <- renderRglwidget({
    rgl.open(useNULL=T)
    scatter3d(x, y, z,
      groups = iris$Species,
      col = as.numeric(iris$Species), surface=FALSE)
    rglwidget()
  })
})
shinyApp(ui = ui, server = server)
```



Note: if an error occurs, you might need to install an updated version of rgl using this [LINK](#). To install the latest development version, try the following.

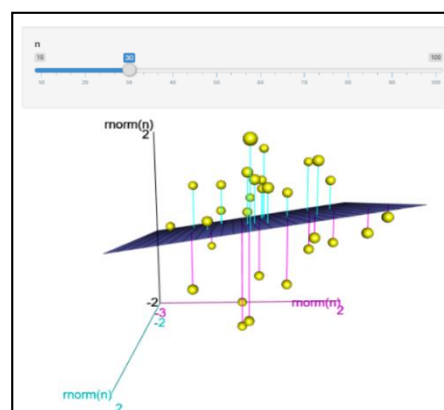
```
# Install development version from Github
remotes::install_github("dmurdoch/rgl")
```

It'll have a prompt to choose an item from a list, **chose 1** which installs/updates everything

Exercise 6. Let's try with another 3D example, but this time lets generate n number of random samples. In order to define n , the user can use a slider input to generate the samples. Please do the following.

```
library(rgl)
set.seed(100)

ui <- fluidPage(
  sidebarLayout(
    sidebarPanel(
      sliderInput("n", label = "n",
        min = 10, max = 100,
        value = 10, step = 10)),
    mainPanel(
```



```

    rglwidgetOutput("myPlot",
      width = 800,
      height = 600))
  )
)
server <- (function(input, output) {
  output$myPlot <- renderRglwidget({
    n <- input$n
    rgl.open(useNULL=T)
    scatter3d(rnorm(n), rnorm(n), rnorm(n))
    rglwidget())
  })
})
shinyApp(ui = ui, server = server)

```

Exercise 7. Let's try another 3D example, but this time let's use an example from the *threejs* package. In this example we will also learn about *reactive expressions*. Reactive expressions let us control which parts of our app update when, which prevents unnecessary computation that can slow down the app. For example, a very simple reactive expression called *up_to_x* that generates a sequence of numbers based on *input\$x* that uses the *seq_len()* function to generate a sequence of increasing numbers starting to whatever number you pass it; .e.g, *seq_len(3)* returns *c(1L,2L,3L)*. The full example would look like:

```
up_to_x <- reactive({ seq_len(x(3)) })
```

This means that this sequence of numbers is available for retrieval, by calling *up_to_x()* like it is a function. In this sense, creating a reactive expression is comparable to declaring a function, nothing occurs until we call the function. Please do the following.

```
library("shiny")
library("threejs")
```

```

ui <- fluidPage(
  titlePanel("Relative population of world cities from the R maps package"),
  sidebarLayout(
    sidebarPanel(
      sliderInput("N", "Number of cities to plot",
        value=5000, min = 100,
        max = 10000, step = 100),
      hr(), #horizontal rule
      p("Use the mouse zoom to zoom in/out."),
      p("Click and drag to rotate.")
    ),
    mainPanel(
      globeOutput("globe")
    )
  )
)

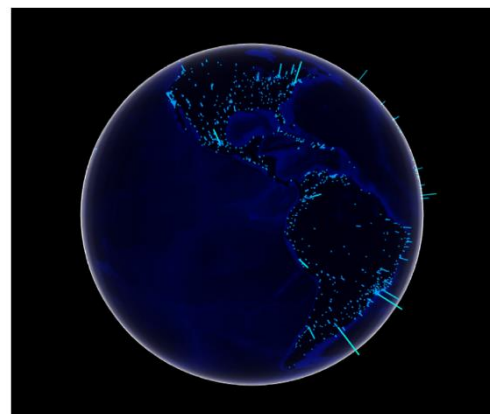
```

```

data(world.cities, package="maps")
earth_dark <- list(img=system.file("images/world.jpg",

```

Relative population of world cities from the R maps package



```

        package="threejs"),
        bodycolor="#0011ff",
        emissive="#000010",
        lightcolor="#99ddff")

server <- function(input, output)
{
  options(warn=-1) # to avoid warnings
  h <- 100 # height of the bar

  cull <- reactive({
    world.cities[order(world.cities$pop, decreasing=TRUE)[1:input$N], ]
  })

  values <- reactive({
    cities <- cull()
    value <- h * cities$pop / max(cities$pop)
    col <- rainbow(10, start=2.8 / 6, end=3.4 / 6)
    names(col) <- c()
    # Extend palette to data values
    col <- col[floor(length(col) * (h - value) / h) + 1]
    list(value=value, color=col, cities=cities)
  })

  output$globe <- renderGlobe({
    v <- values()
    p <- input$map
    args <- c(earth_dark, list(lat=v$cities$lat, long=v$cities$long, value=v$value, color=v$color,
atmosphere=TRUE))
    do.call(globejs, args=args)
  })
}

shinyApp(ui = ui, server = server)

```

Exercise 8. For more great examples, you can inspect the examples included in the Shiny package. Try any of the examples and see their results, please do the following.

```

# list all the available examples
runExample()

# Print the directory containing the code for all examples
system.file("examples", package="shiny")

# run one of the examples (one at a time)
runExample("01_hello")
runExample("02_text")
runExample("03_reactivity")

```

For more interesting examples, please visit <https://shiny.rstudio.com/gallery/>.

2. Take Home Exercises

2.1 Iris dataset. Create an app that displays the names of the variables in the dataset (example below). The user should select the name of the variable to plot from a drop-down list (hint: `selectInput()` with `choices=colnames()`). The output should be a histogram reflecting the distribution of data from the selected variable.

