Part II. Continue

March 27, 2025

1 Part 2 (Continue): Interactive Web Applications with Shiny

1.0.1 Instructor: Dr. Maryam Movahedifar

`<img src="Uni_Logo.png" alt="Uni Logo" style="width: 100px; margin-right: 10px;">`
`<img src="DSC_Logo.png" alt="DSC Logo" style="width: 150px;">`

2 3 Basic Reactivity

2.1 3.1 Introduction

In Shiny, server logic is expressed using **reactive programming**. This paradigm is powerful and elegant but can feel disorienting at first since it differs significantly from traditional scripting.

The core idea of reactive programming is to define a **graph of dependencies**, ensuring that when an input changes, all related outputs update automatically. This simplifies app flow but requires some time to fully grasp.

This part provides a **gentle introduction** to reactive programming, covering the most common reactive constructs in Shiny. It includes:
- A detailed look at the **server function** and how `input` and `output` arguments work.
- Basic reactivity, where **inputs connect directly to outputs**.
- **Reactive expressions**, which help eliminate redundant calculations.
- Common pitfalls that new Shiny users often face.

2.2 3.2 The Server Function

As you have seen every Shiny app follows this basic structure:

[1]:
```r
library(shiny)

ui <- fluidPage(
  # front end interface
)

server <- function(input, output, session) {
  # back end logic
}

shinyApp(ui, server)
```

```
Listening on http://127.0.0.1:5763
```

2.3 Server Function Overview

The **ui** defines the front end of a Shiny app, presenting the same HTML to all users. In contrast, the **server** function is more complex because each user needs an independent session—ensuring that changes made by one user don't affect others.

To achieve this, Shiny calls the `server()` function for each new session, creating a **separate local environment**. This ensures that:
- Each session maintains its own unique state.
- Variables inside the function remain isolated.
- Most reactive programming occurs within the `server` function.

2.3.1 Server Function Parameters

The `server` function has three key parameters:
- **`input`** – Captures user inputs.
- **`output`** – Manages UI updates.
- **`session`** – Handles session-specific operations (covered later).

Since the `server` function is automatically called by Shiny, developers do not manually create these objects. The next sections will focus on `input` and `output`.

2.4 3.2.1 Input

The `input` argument is a **list-like object** that stores all user input data from the browser. Each input is named according to its **input ID**.

For example, if your UI includes a numeric input with an ID of `count`:

[2]:
```
ui <- fluidPage(
  numericInput("count", label = "Number of values", value = 100)
)
```

Once defined, you can access the input value using `input$count`. Initially, it will contain the default value (e.g., `100`), and it will automatically update as the user interacts with the UI.

Unlike regular lists, **`input` objects are read-only**. Attempting to modify an input value inside the server function will result in an error.

[3]:
```
server <- function(input, output, session) {
  input$count <- 10
}

shinyApp(ui, server)
#> Error: Can't modify read-only reactive value 'count'
```

```
Listening on http://127.0.0.1:5763
```

2.4.1 Important Note About `input`

One crucial rule about `input`: **it can only be read inside a reactive context**. Functions like `renderText()` or `reactive()` create these contexts. This constraint ensures that outputs update automatically when inputs change.

If you try to access `input` outside a reactive context, you'll get an error. For example, the following code will fail:

[4]:
```
server <- function(input, output, session) {
  value <- input$count  #  Error: Not in a reactive context
}
```

2.4.2 Summary

Shiny restricts access to `input` values—you can only read them inside **reactive contexts** like `renderText()` or `reactive()`. This rule ensures that outputs update automatically when inputs change.

2.5 3.2.2 Output

The `output` object is similar to `input`: it's a **list-like object** named according to the **output ID**. However, unlike `input`, which receives data from the user, `output` is used to **send data to the UI**.

To update an output, you must use it inside a **render function**. Here's a simple example:

[5]:
```
ui <- fluidPage(
  textOutput("greeting")
)

server <- function(input, output, session) {
  output$greeting <- renderText("Hello human!")
}
```

2.5.1 The Render Function

The **render function** plays two important roles:

1. **Creates a reactive context** – It automatically tracks which `input` values the `output` depends on, ensuring updates happen when inputs change.

2. **Formats output for the web** – It converts R output into **HTML**, making it suitable for display in the UI.

Common Mistake: Forgetting the Render Function Just like `input`, `output` has strict usage rules. If you forget to use a **render function**, you'll encounter an error.

For example, this will **fail**:

[6]:
```
server <- function(input, output, session) {
  output$greeting <- "Hello human"
}
shinyApp(ui, server)
#> Error: Unexpected character object for output$greeting
#>   Did you forget to use a render function?
```

```
Listening on http://127.0.0.1:5763
```

Another mistake: You attempt to read from an **output**:

[7]:
```
server <- function(input, output, session) {
  message("The greeting is ", output$greeting)
}
shinyApp(ui, server)
#> Error: Reading from shinyoutput object is not allowed.
```

```
Listening on http://127.0.0.1:5763
```

2.6 3.3 Reactive Programming

An app isn't very useful if it only has inputs or only has outputs. The **real power of Shiny** comes from combining both, allowing outputs to dynamically respond to user inputs.

Here's a simple example of how reactivity works in Shiny:

[8]:
```
ui <- fluidPage(
  textInput("name", "What's your name?"),
  textOutput("greeting")
)

server <- function(input, output, session) {
  output$greeting <- renderText({
    paste0("Hello ", input$name, "!")
  })
}
```

2.6.1 Summary

Shiny's **reactive programming** enables dynamic interactions between inputs and outputs. When an input changes, **Shiny automatically updates dependent outputs**—creating an interactive experience.

2.6.2 Note

Shiny uses **lazy evaluation**, meaning code runs **only when needed**. Unlike regular R scripts that execute **line by line**, Shiny determines execution order based on **reactive dependencies**. Instead of running code sequentially, Shiny only updates outputs **when necessary**, improving efficiency and keeping the app responsive.

2.7 3.4 Reactive Expressions

2.7.1 What Are Reactive Expressions?

A **reactive expression** is a piece of code that automatically updates when the input values on which it depends change. It combines the roles of both **inputs** (what the user provides) and **outputs** (what is shown to the user).

Shiny ensures only the necessary calculations are done, making your app more efficient.

2.7.2 Why Are Reactive Expressions Important?

1. **Efficiency** – They prevent unnecessary recalculations, making apps faster.

2. **Organized** – They help manage dependencies and make the app easier to understand.

2.7.3 A More Complex Example

Next, we will define some simple functions that will help build a more complex Shiny app.

2.8 3.4.1 The Motivation

Imagine you want to **compare two simulated datasets** using:
1. A **plot** to visualize their distributions.
2. A **hypothesis test** to compare their means.

After some experimentation, we create two helper functions:

- `freqpoly()` → Visualizes the two distributions using **frequency polygons**.

- `t_test()` → Performs a **t-test** to compare means and returns a summary string.

These functions will form the foundation of our Shiny app, where we'll use **reactive expressions** to make the app more efficient and structured.

[9]:
```
library(ggplot2)

freqpoly <- function(x1, x2, binwidth = 0.1, xlim = c(-3, 3)) {
  df <- data.frame(
    x = c(x1, x2),
    g = c(rep("x1", length(x1)), rep("x2", length(x2)))
  )
```

```r
  ggplot(df, aes(x, colour = g)) +
    geom_freqpoly(binwidth = binwidth, size = 1) +
    coord_cartesian(xlim = xlim)
}

t_test <- function(x1, x2) {
  test <- t.test(x1, x2)

  # use sprintf() to format t.test() results compactly
  sprintf(
    "p value: %0.3f\n[%0.2f, %0.2f]",
    test$p.value, test$conf.int[1], test$conf.int[2]
  )
}
```

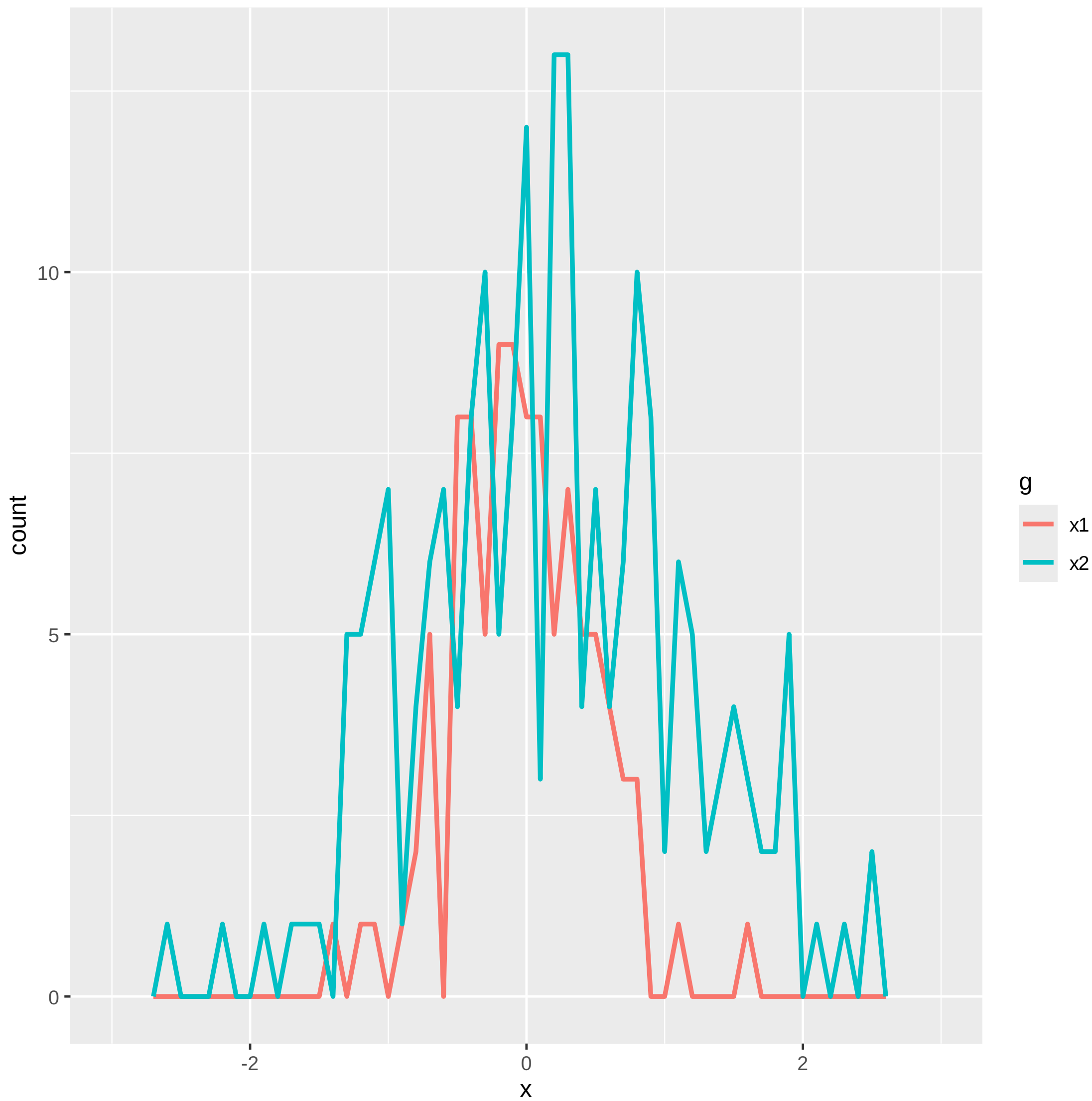
If I have some simulated data, I can use these functions to compare two variables:

[10]:
```r
x1 <- rnorm(100, mean = 0, sd = 0.5)
x2 <- rnorm(200, mean = 0.15, sd = 0.9)

freqpoly(x1, x2)
#> Warning: Using `size` aesthetic for lines was deprecated in ggplot2 3.4.0.
#>   Please use `linewidth` instead.
#> This warning is displayed once every 8 hours.
#> Call `lifecycle::last_lifecycle_warnings()` to see where this warning was
#> generated.
cat(t_test(x1, x2))
#> p value: 0.386
#> [-0.24, 0.09]
```

```
Warning message:
"Using `size` aesthetic for lines was deprecated in ggplot2 3.4.0.
 Please use `linewidth` instead."
```

```
p value: 0.023
[-0.35, -0.03]
```



2.9 3.4.2 The App

To efficiently explore multiple simulations, a **Shiny app** is ideal—it allows interactive adjustments **without** manually modifying and re-running R code.

2.9.1 UI Structure

The app layout consists of:
1. **First row** → Three columns for input controls:
- Distribution 1
- Distribution 2
- Plot controls
2. **Second row** →

- A **wide column** for the plot.
- A **narrow column** for the hypothesis test results.

The layout uses `fluidRow` and `column(n)` functions to organize the UI elements:

- `fluidRow` is used to define a row that can hold multiple columns.

- `column(n)` is used within a `fluidRow` to define columns that take up a specified width (out of 12 total units).

By structuring the app this way, we can easily adjust input parameters and observe changes in real time.

[11]:
```
ui <- fluidPage(
  fluidRow(
    column(4,
      "Distribution 1",
      numericInput("n1", label = "n", value = 1000, min = 1),
      numericInput("mean1", label = "µ", value = 0, step = 0.1),
      numericInput("sd1", label = " ", value = 0.5, min = 0.1, step = 0.1)
    ),
    column(4,
      "Distribution 2",
      numericInput("n2", label = "n", value = 1000, min = 1),
      numericInput("mean2", label = "µ", value = 0, step = 0.1),
      numericInput("sd2", label = " ", value = 0.5, min = 0.1, step = 0.1)
    ),
    column(4,
      "Frequency polygon",
      numericInput("binwidth", label = "Bin width", value = 0.1, step = 0.1),
      sliderInput("range", label = "range", value = c(-3, 3), min = -5, max = 5)
    )
  ),
  fluidRow(
    column(9, plotOutput("hist")),
    column(3, verbatimTextOutput("ttest"))
  )
)
```

The server function combines calls to `freqpoly()` and `t_test()` functions after drawing from the specified distributions:

[12]:
```
server <- function(input, output, session) {
  output$hist <- renderPlot({
    x1 <- rnorm(input$n1, input$mean1, input$sd1)
    x2 <- rnorm(input$n2, input$mean2, input$sd2)

    freqpoly(x1, x2, binwidth = input$binwidth, xlim = input$range)
  }, res = 96)
```

```
  output$ttest <- renderText({
    x1 <- rnorm(input$n1, input$mean1, input$sd1)
    x2 <- rnorm(input$n2, input$mean2, input$sd2)

    t_test(x1, x2)
  })
}
```

2.10 3.5 Controlling Timing of Evaluation

In this section, we'll explore **two advanced techniques** to **control how often** a reactive expression runs—either **increasing** or **decreasing** its execution frequency.

Later, in **Chapter 15**, we'll dive deeper into their underlying implementations.

2.10.1 Simplifying the Simulation App

To focus on timing control, we simplify the previous app:
- Use a **single-parameter** distribution.
- Force both samples to share the same **n**.
- Remove **plot controls** to reduce complexity.

This results in a **smaller UI** and **simpler server function**, making it easier to understand the key concepts of timing control in Shiny.

[13]:
```
ui <- fluidPage(
  fluidRow(
    column(3,
      numericInput("lambda1", label = "lambda1", value = 3),
      numericInput("lambda2", label = "lambda2", value = 5),
      numericInput("n", label = "n", value = 1e4, min = 0)
    ),
    column(9, plotOutput("hist"))
  )
)
server <- function(input, output, session) {
  x1 <- reactive(rpois(input$n, input$lambda1))
  x2 <- reactive(rpois(input$n, input$lambda2))
  output$hist <- renderPlot({
    freqpoly(x1(), x2(), binwidth = 1, xlim = c(0, 40))
  }, res = 96)
}
```

In this example, instead of manually recalculating the values each time the inputs change, reactive expressions allow the app to automatically re-evaluate and update the outputs when the inputs (`n`, `lambda1`, `lambda2`) change. This makes the app more efficient and dynamic.

2.11 3.5.1 Timed Invalidation

Imagine you want to **animate** the plot by constantly resimulating the data. This would help reinforce that you're working with **simulated data**. To achieve this, we can **increase the frequency** of updates using a special function: **`reactiveTimer()`**.

2.11.1 How `reactiveTimer()` Works

`reactiveTimer()` is a **reactive expression** that depends on a hidden input: the **current time**. It allows a reactive expression to **invalidate** and **recalculate** itself more frequently than usual.

For example, the following code sets an interval of **500 ms**, meaning the plot will update twice every second:

[14]:
```
server <- function(input, output, session) {
  timer <- reactiveTimer(500)

  x1 <- reactive({
    timer()
    rpois(input$n, input$lambda1)
  })
  x2 <- reactive({
    timer()
    rpois(input$n, input$lambda2)
  })

  output$hist <- renderPlot({
    freqpoly(x1(), x2(), binwidth = 1, xlim = c(0, 40))
  }, res = 96)
}
```

2.11.2 Summary

`reactiveTimer()` is used to create frequent updates by invalidating a reactive expression based on time. It's useful for animations or continuously updating data in Shiny apps.

2.12 3.5.2 On Click

In the previous scenario, consider what would happen if the **simulation** took **1 second** to run. Since we're performing the simulation every **0.5 seconds**, Shiny would continuously accumulate tasks, resulting in a **backlog**. This backlog would prevent Shiny from responding to new events, creating a poor **user experience**.

A similar problem can occur when users **click buttons rapidly** in your app, triggering expensive computations. Shiny might not be able to catch up, leading to delays and unresponsiveness.

2.12.1 Solution: Using `actionButton()`

To prevent this issue, you can allow the user to **opt-in** to perform expensive calculations. Instead of automatically triggering computations, **require the user to click a button** to start the process.

This approach controls when the computation happens and avoids creating unnecessary backlogs.

Here's an example using an `actionButton()` to trigger a simulation:

[15]:
```
ui <- fluidPage(
  fluidRow(
    column(3,
      numericInput("lambda1", label = "lambda1", value = 3),
      numericInput("lambda2", label = "lambda2", value = 5),
      numericInput("n", label = "n", value = 1e4, min = 0),
      actionButton("simulate", "Simulate!")
    ),
    column(9, plotOutput("hist"))
  )
)
```

[16]:
```
server <- function(input, output, session) {
  x1 <- reactive({
    input$simulate
    rpois(input$n, input$lambda1)
  })
  x2 <- reactive({
    input$simulate
    rpois(input$n, input$lambda2)
  })
  output$hist <- renderPlot({
    freqpoly(x1(), x2(), binwidth = 1, xlim = c(0, 40))
  }, res = 96)
}
```

2.13 Solving the Problem with `eventReactive()`

The previous solution using `actionButton()` didn't fully achieve our goal. Although it added a button to trigger the simulation, **x1()** and **x2()** still had dependencies on **lambda1**, **lambda2**, and **n**. This means they would **update automatically** when any of these inputs changed, which isn't what we want. We want to **replace** the existing dependencies, not just add to them.

2.13.1 Solution: Using `eventReactive()`

To achieve this, we can use **`eventReactive()`**. This function has two arguments:

1. The **first argument** specifies the input(s) that should trigger the computation.

2. The **second argument** specifies the computation to run when the event is triggered.

Using `eventReactive()`, we can ensure that **x1()** and **x2()** are only computed **when the "simulate" button is clicked**, and not when other inputs change.

Here's how to implement it:

[17]:

```r
server <- function(input, output, session) {
  x1 <- eventReactive(input$simulate, {
    rpois(input$n, input$lambda1)
  })
  x2 <- eventReactive(input$simulate, {
    rpois(input$n, input$lambda2)
  })

  output$hist <- renderPlot({
    freqpoly(x1(), x2(), binwidth = 1, xlim = c(0, 40))
  }, res = 96)
}
```