Statistical Modelling Tool

Author:
Nishant Kumar

Supervisor:
André Lange

January 19, 2018

Fraunhofer Institute for Integrated Circuits Division

Engineering of Adaptive Systems

Contents

1 Introduction

With the rise of widely available high-speed internet connections, web applications have become more an more popular. However, to create a web app, a lot of programming experience in HTML and JavaScript was needed. In this void, RStudio released it's product called Shiny. Shiny is a means of creating web applications entirely in R. The client-server communication, HTML, layout and JavaScript programming is entirely handled by Shiny. The research group "Quality and Reliability" at the Fraunhofer Institute for Integrated Circuits IIS, Division Engineering of Adaptive Systems EAS, Dresden, Germany focuses on considering imperfections during manufacturing and operating integrated circuits in the circuit design phase already. A particular effect of interest is variability, which is introduced by process variations and atomic-level fluctuations. Variability causes the characteristics of integrated circuits to be statistical quantities that can be modeled by multivariate random variables. One task to be solved is statistical modeling, i.e. mapping sample data to a feasible description method for multivariate random variables. A user-friendly tool is therefore been created using Shiny to support the modeling activities and to demonstrate this part of the Fraunhofer IIS/EAS portfolio to customers and project partners.

2 Shiny Package

Shiny is an R package that makes it easy to build interactive web applications (apps) straight from R. The package can be installed and executed in the application by running the following commands:

```
install.packages("shiny")
library(shiny)
```

2.1 Basic Structure of a Web Application

The structure of a web-application generally takes the form as shown in Fig.1 below. A typical web application consist of a number of user interface (UI) elements, say a button or checkbox. Each of these elements can be interacted with, for example push the button. This button push then triggers an action, running a piece of code. This can then change the state of the web application, for example retrieve a piece of information from the server

or draw a picture in the application. This style of programming is called event-driven programming. The piece of code that is executed based on an event is called an event handler.

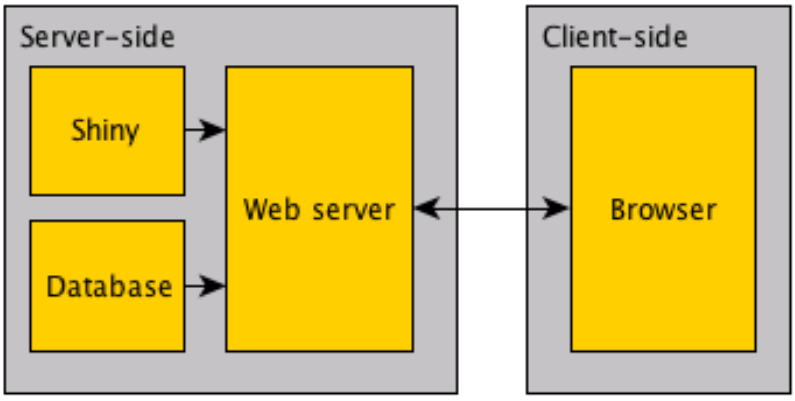

Figure 1: Structure of a Web Application

2.2 Structure of Shiny Apps

Shiny apps follow this typical structure of web applications. However, as a user you only have to specify which UI elements you want to show, and the underlying R code that draws a plot, shows some text, or a table. These pieces of information are stored in two scripts:

1. A user-interface script
2. A server script

The user-interface (UI) script controls the layout and appearance of the tool and the server script contains the instructions that the computer needs to build the tool.
Note: R session will be busy while the Statistical Modelling tool is active, so the user will not be able to run any other R commands. R is monitoring the tool and executing the tool's reactions. To get the R session back, hit escape or click the stop sign icon (found in the upper right corner of the RStudio console panel). As of version 0.10.2, Shiny supports single-file applications. The tool has been saved as a R file called shiny_tool.R that contains both

the server and UI components.
The basic template of a Shiny app is as follows:

```
library(shiny)
ui <- fluidPage()
server <- function(input, output){}
shinyApp(ui = ui, server = server)
```

3 Building a User Interface

3.1 pageWithSidebar()

Shiny UI script used the function **pageWithSidebar** to create a Shiny UI that contains a header with the application title, a sidebar for input controls, and a main area for output. The code below showcases it:

```
ui <- pageWithSidebar(
                headerPanel(),
                sidebarPanel(),
                mainPanel(),
                conditionalPanel()
                        )
```

Fig.2 below shows the layout of the tool identifying each segments:

3.1.1 headerPanel

Creates a header panel containing an application title that should be displayed by the browser window.

```
headerPanel("CSV Viewer")
```

3.1.2 sidebarPanel

All the UI elements that needs to be part of the sidebar needs to be included on the sidebarpanel. The tool for statistical modelling which has been developed has the following features in sidebarpanel:

```
sidebarPanel(
            fileInput(),
```

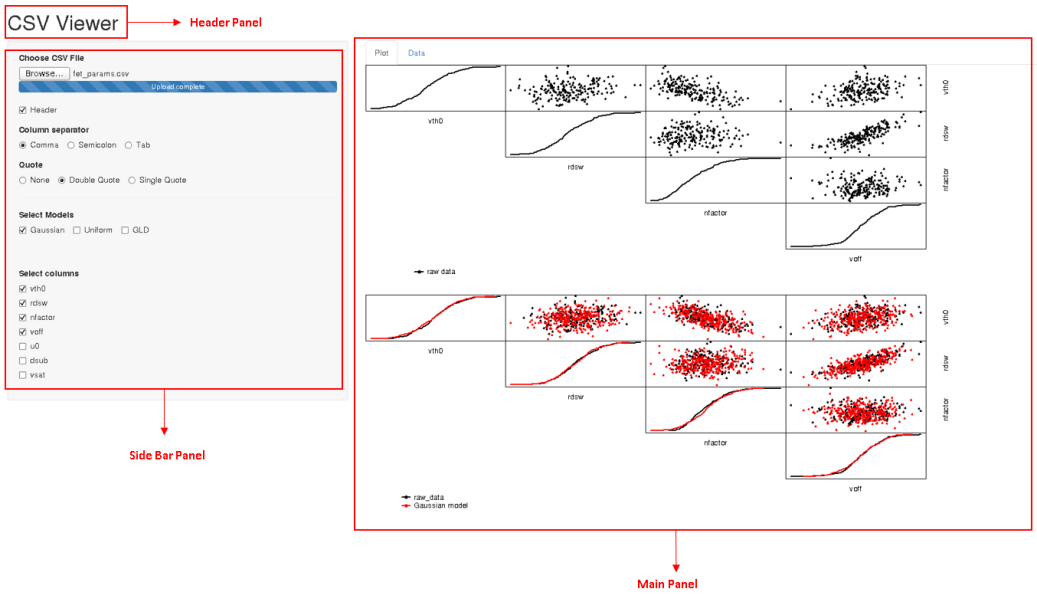

Figure 2: Tool Layout

```
checkboxInput(),
radioButtons(),
fluidRow(),
uiOutput()
  ),
```

- **fileInput** - Creates a file upload control that can be used to upload one or more files.In the tool the file which has been uploaded is a CSV File.

```
fileInput('file1',
          'Choose CSV File',
          accept=c('text/csv',,'.csv')
         )
```

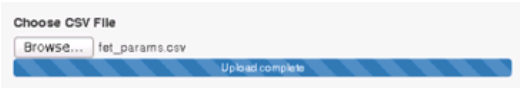

Figure 3: File Input

- **checkboxInput** - Creates a checkbox that can be used to specify logical values.It generates a checkbox control that can be added to a UI definition.

```
checkboxInput('header', 'Header', TRUE)
```

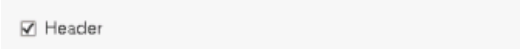

Figure 4: CheckboxInput Example

- **radioButtons** - Creates a set of radio buttons used to select an item from a list.This set of radio buttons has been added to the UI definition.

```
radioButtons('quote',
             'Quote',
             c(None='',
               'Double Quote'='"',
               'Single Quote'="'"),
             '"',
             inline=TRUE
            )
```

Figure 5: Radio Button Example

- **checkboxGroupInput** - Create a group of checkboxes that can be used to toggle multiple choices independently. The server will receive the input as a character vector of the selected values.

```
checkboxGroupInput(inputId='Models',
                   label='Select Models',
                   choices=c(Gaussian='Gaussian',
                             Uniform ='Uniform',
                             GLD='GLD'),
                   selected=c(),
                   inline=TRUE)
                   )
```

Figure 6: CheckBoxGroupInput Example

3.1.3 mainPanel

Create a main panel containing output elements that can in turn be passed to sidebarLayout. All output elements has been included in the main panel. They are as follows:

```
mainPanel(
          tabsetPanel(),
          conditionalPanel(),
          conditionalPanel()
          )
```

- **tabsetPanel** - Create a tabset that contains tabPanel elements. Tabsets are useful for dividing output into multiple independently viewable sections.
- **conditionalPanel** - Creates a panel that is visible or not, depending on the value of a JavaScript expression. The JS expression is evaluated once at start up and whenever Shiny detects a relevant change in input/output.

4 Displaying Reactive Output

In order to give a live quality to the modelling tool , reactive output has been built. Reactive output automatically responds when the user toggles a widget such as checkboxes, radiobuttons and so on. Reactive output has been created with a two step process. They are as follows:

1. Adding an R object to the user-interface i.e UI script
2. Telling Shiny to build the object in server script. The object will be reactive if the code that builds it calls a widget value.

4.1 Step 1: Adding an R object to the UI

Shiny provides a family of functions that turn R objects into output for the user-interface. The functions used in Each function creates a specific type of output. The functions that have been used in the tool are:

1. **plotOutput** - This function creates a plot or image output element that has been included in a panel on the right hand side of the tool. For example:

   ```
   plotOutput('plot')
   ```

2. **tableOutput** - This function creates a table output element that has been included in a panel on the right hand side of the tool.

   ```
   tableOutput("contents")
   ```

3. **uiOutput** - The function uiOutput will tell Shiny where the controls such as renderUI declared in server script should be rendered. In the below, it should be rendered in uiColumnSelection

   ```
   uiOutput(outputId="uiColumnSelection")
   ```

4.2 Step 2: Provide R code to build the object in Server

Placing a function in ui script only tells Shiny where to display the object.Up to now, the user interface does not contain dynamic content: the option selection does nothing and the text is static. We can change this using the server. The most important to build the tool is to tell shiny how to build the object. This has been done by providing relevant R code that builds the object in server script. The code should go in the function that appears in the tool as follows:

```
server <- function(input,output,session) {}
```

This function plays a special role in the Shiny process; it builds a list-like object named **output** that contains all of the code needed to update the R objects in the tool. Each R object have its own entry in the list. An entry has been created within the function by defining a new element as shown below. The element name should match the name of the reactive element that has been created in ui part of script.

```
output$uiColumnSelection<-renderUI(uiColumnSelection());
```

Each entry to output contain the output of one of Shiny's render functions. These functions capture an R expression and do some light preprocessing on the expression. There is a render function corresponding to each type of reactive object declared in ui part of the script. Since reactive objects such as plotOutput , tableOutput and uiOutput has been declared in the ui, following are the render function that has been declared in the server part of the script:

1. **renderPlot** - specifically for plots. In the code segment below, Plotscatter function used to manipulate the current data. After performing the operation the result is sent to the UI which is plotOutput using the command output$plot.

   Each render function takes a single argument: an R expression surrounded by braces. The expression can be one simple line of text, or it can involve many lines of code, as if it were a complicated function call. For example, the following part of the code for renderPlot showcases it:

   ```
   output$plot<-renderPlot(
                {
                DATA <-list("raw_data"=CURRENT_DATA$plot)
                ## extend data depend on model selection
                PlotScatter(DATA=DATA)
                })
   ```

   Think of this R expression as a set of instructions that has been given to Shiny to store for later. Shiny will run the instructions when you first launch the tool, and then Shiny will re-run the instructions every time it needs to update the object.

   For this to work, the R expression should return the object (a piece of text, a plot, a data frame, etc). An error will be displayed if the expression does not return an object, or if it returns the wrong type of object.

2. **renderTable** - any printed output data frame, matrix, other table like structures.

3. **renderUI** - It is used in server script in conjunction with the uiOutput function in ui script, and it lets you generate calls to UI functions and make the results appear in a predetermined place in the UI. It can be also called a Shiny tag object or HTML.

```
output$uiColumnSelection <- renderUI(
                              uiColumnSelection(CURRENT_DATA)
                                                  );
```

4.3 reactive()

The tool has been made faster by modularizing the code with reactive expressions. A reactive expression takes input values, or values from other reactive expressions, and returns a new value. Reactive expressions save their results, and will only re-calculate if their input has changed. They can be created with reactive( ). Reactive expressions has been called with the name of the expression followed by parentheses (). Use reactive expressions from within other reactive expressions or render* functions.

4.4 observe()

The object has been made reactive also by asking Shiny to call a widget value when it builds the object. Please consider the following code:

```
server <- function(input,output,session) {
observe({
          if(is.null(input$file1)) { }
          else
            { }
}
```

Take a look at the first lines of code above. It is noticeable that the function mentions two arguments, input and output. While we discussed that output is a list-like object that stores instructions for building the R objects in the tool.
input is a second list-like object. It stores the current values of all of the widgets in the tool. These values will be saved under the names that has been given to the widgets in ui part of the code . So for example, the tool has a widget, named "file1" (which accepts a file input from user). The

value of "file1" which is nothing but a text or csv will be saved in input as input$file1. Shiny automatically make the object reactive if the object uses an input value which is true in this case.

Shiny also tracks which outputs depend on which widgets. When the user will change a widget which is a CSV file in the tool, Shiny will rebuild all of the outputs that depend on the widget i.e CSV file, using the new value of the widget as it goes. As a result, the rebuilt objects will be completely up-to-date.

Therefore, by connecting the values of input to the objects in output, reactivity has been implemented in the tool with Shiny.

4.5 Difference between observe() and reactive()

An observer is like a reactive expression in that it can read reactive values and call reactive expressions, and will automatically re-execute when those dependencies change. But unlike reactive expressions, it doesn't yield a result and can't be used as an input to other reactive expressions. Thus, observers are only useful for their side effects (for example, performing I/O).

4.6 reactiveValues()

This function returns an object(CURRENT_DATA in the tool) for storing reactive values. It is similar to a list, but with special capabilities for reactive programming. When you read a value from it, the calling reactive expression takes a reactive dependency on that value, and when you write to it, it notifies any reactive functions that depend on that value. Note that values taken from the reactiveValues object are reactive, but the reactiveValues object itself is not. Take a look at the below code from the tool:

```
CURRENT_DATA <- reactiveValues(
                              all=data.frame(),
                              plot=data.frame()
                                    )
```

5 Using External R scripts, Data and Packages

In order to add various functionalities in the tool, external data, R Scripts, and packages has to be used in the tool. There are two types of external files/scripts which has been used in the tool:

1. **CSV files** - CSV is a simple file format i.e used to store tabular data, such as a spreadsheet or database. CSV stands for "comma-separated values". Data required for the tool is taken from this CSV file. In short, this file acted as the input to the tool.

2. **R scripts** - The R functions has been imported into the tool to manipulate the data from CSV file and perform the necessary task according to the specification of the tool. R functions has been imported into the tool using the **source** command and specifying the path where the file is located as shown below:

   ```
   source("/eas/../PlotScatter.R");
   ```

   The R function used in the tool are:

   - **PlotScatter**: R function to generate scatterplot matrix with information on marginal distributions on "main diagonal".
   - **FitGLD**: R function to fit data to GLD. It has been implemented because starship() from package 'gld' is too slow.

   Note: It is very important that all CSV files and R scripts exists in the same folder directory. In other words, the directory that you save PlotScatter.R will become the working directory of the Shiny tool.

3. **R packages**: Packages are collections of R functions, data, and compiled code in a well-defined format. The directory where packages are stored is called the library. R comes with a standard set of packages. Packages can be instantiated into the tool by invoking **require** command as shown below:

   ```
   require(MASS)
   ```

   The packages which has been used in the tool are:

- **shiny**: It is the web application framework for R. Makes it incredibly easy to build interactive web applications with R. Automatic "reactive" binding between inputs and outputs and extensive prebuilt widgets make it possible to build beautiful, responsive, and powerful application with minimal effort. This is the most important package for the tool.
- **MASS**: It supports functions and datasets for Venables and Ripley's MASS.
- **Matrix**: It invokes Sparse and Dense Matrix Classes and Methods using using 'LAPACK' and 'SuiteSparse'.
- **gld**: It is used for the estimation and use of the Generalised (Tukey) Lambda Distribution.

6 Wrapping UI and Server scripts

shinyApp command has to be written in the tool to combine ui and server parts of the code into a functioning app. Below is the code which showcases this:

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

7 How to use the tool

Following are the steps that needs to be followed to work with the tool:

1. Open the shiny_tool.R script in RStudio and select the code as shown below:

2. Copy the script in the Console Window of the RStudio and then press Enter. The tool now starts running and if there were no errors in the script, a new html page opens with a layout of tool and all its functionalities.

   The layout as shown in Fig.4 gets displayed in the tool. It is observable that there is a Browse button. Please select an input CSV File by using this button.

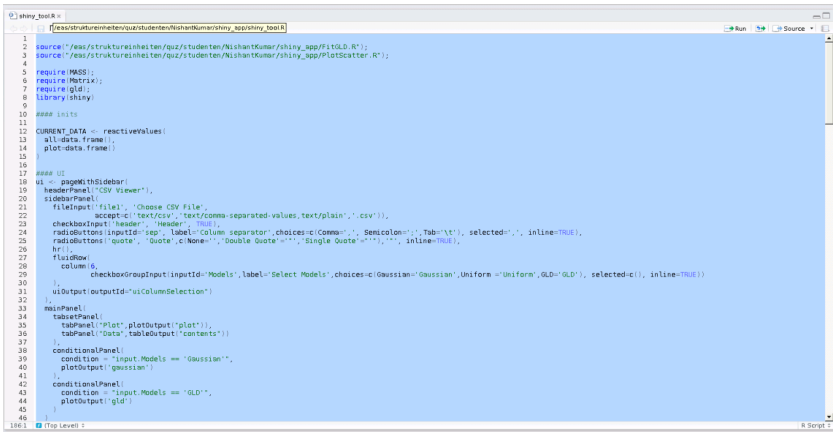

Figure 7: shiny_tool.R in RStudio

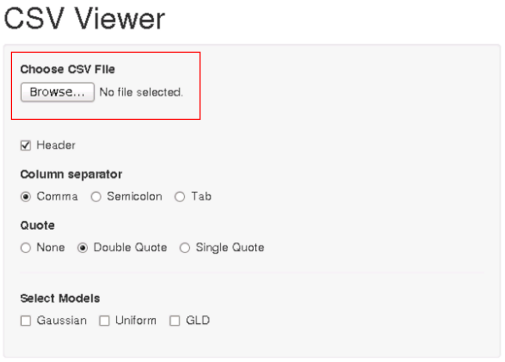

Figure 8: Browse Button in Tool

3. If a CSV File is selected, the tool automatically reads the columns inside the CSV file and updates the layout with checkboxes showcasing all the valid columns.

4. Once the columns gets updated in the tool. Any combination of columns can be selected in the checkbox to get the plot as shown in Fig.6 below. It should be noted that the plot that is generated is CDF graph and the number of selections of columns shown in the figure is 1 and 4 respectively.

5. Different models can also be plotted inside the plot of the main panel. The checkbox concerned with models does this work. When the user

Figure 9: Columns of the CSV File updated in the tool

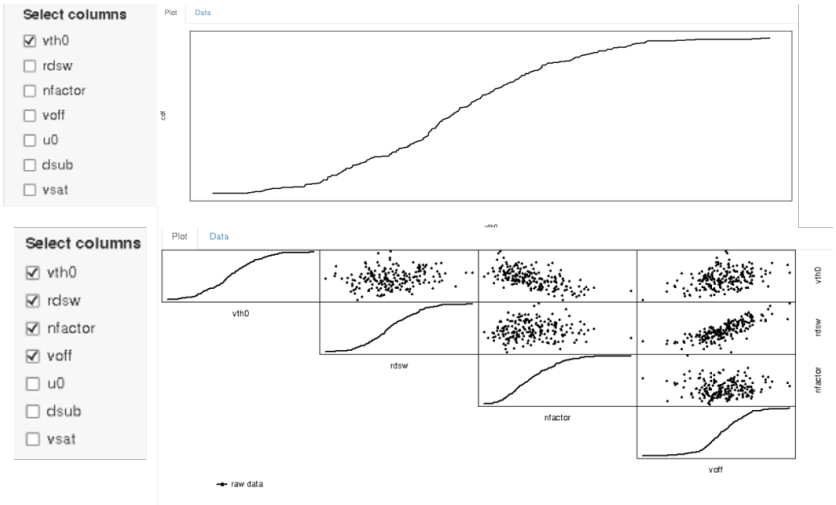

Figure 10: Column selections and CDF Plot

selects a model by ticking one of the checkbox, a plot of that model superimposing on the CDF plot appears in the panel. Following figure showcases it:

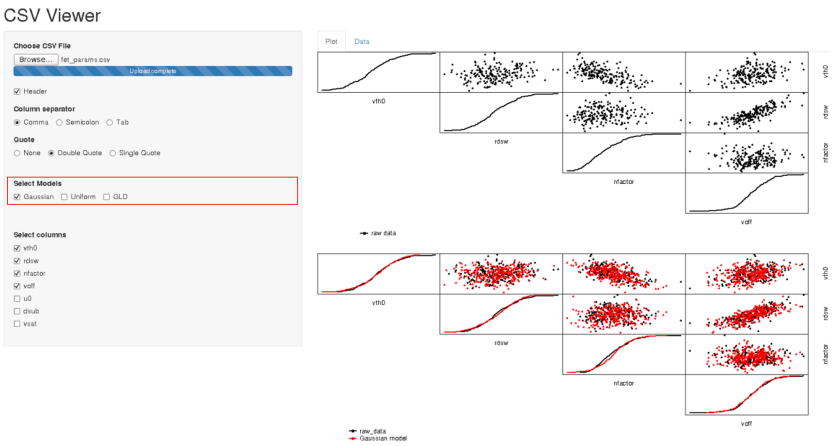

Figure 11: Model Selection and Plot

References

[1] https://shiny.rstudio.com/tutorial/

[2] http://shiny.rstudio.com/images/shiny-cheatsheet.pdf

Appendices

Following is the tool source code:

```
source("/eas/projekte/varimod-2_180564/work/nkumar/Shiny_App/FitGLD.R");
source("/eas/projekte/varimod-2_180564/work/nkumar/Shiny_App/PlotScatter.R");

require(MASS);
require(Matrix);
require(gld);
library(shiny)

#### inits

CURRENT_DATA <- reactiveValues(
                                all=data.frame(),
                                plot=data.frame()
                                 )

#### UI
ui <- pageWithSidebar(
  headerPanel("CSV_Viewer"),
  sidebarPanel(
    fileInput('file1', 'Choose_CSV_File',
              accept=c('text/csv','text/comma-separated-values,text/plain','.csv')),
    checkboxInput('header', 'Header', TRUE),
    radioButtons(inputId='sep',
                 label='Column_separator',
                 choices=c(Comma=',', Semicolon=';',Tab='\t'),
                 selected=',', inline=TRUE),
    radioButtons('quote',
                 'Quote',
                 c(None='','Double_Quote'='"','Single_Quote'="'"),
                 '"',
                 inline=TRUE),
    hr(),
    fluidRow(
      column(6,
             checkboxGroupInput(inputId='Models',
                                label='Select___Models',
                                choices=c(Gaussian='Gaussian',
                                          Uniform ='Uniform',
                                          GLD='GLD'),
                                selected=c(),
                                inline=TRUE))
           ),
    uiOutput(outputId="uiColumnSelection")
  ),
  mainPanel(
    tabsetPanel(
          tabPanel("Plot",plotOutput("plot")),
          tabPanel("Data",tableOutput("contents"))
    ),
    conditionalPanel(
          condition = "input.Models_==_'Gaussian'",
          plotOutput('gaussian')
    ),
    conditionalPanel(
          condition = "input.Models_==_'GLD'",
          plotOutput('gld')
    )
  )
)

#### dynamic part of UI
uiColumnSelection <- function(CURRENT_DATA) {
                       if( any(dim(CURRENT_DATA$all)== 0) ) { list(NULL) }
                       else {
                            list(
                               hr(),
                               fluidRow(
                               column(6,
                                      checkboxGroupInput(inputId='Columns',
                                                         label='Select_columns',
```

```
                choices=colnames(CURRENT_DATA$all), selected=colnames(CURRENT_DATA$plot)))
                                    )
                                )
                            }
                                                }

#### server
server <- function(input,output,session) {

  ## render UI for column selection
  output$uiColumnSelection <- renderUI(uiColumnSelection(CURRENT_DATA));

  ## read data from selected file
  observe({
    if( is.null(input$file1) ) {
                          ## no input file selected
                          CURRENT_DATA$all  <<- data.frame();
                          CURRENT_DATA$plot <<- data.frame();
    } else {
        isolate({
        if( file.exists(input$file1$datapath) )
        {
          CURRENT_DATA$all <<- read.csv(file=input$file1$datapath,
                                        header=input$header,
                                        sep=input$sep,
                                        quote=input$quote)
          if( ncol(CURRENT_DATA$all) > 4 ) {
                                    CURRENT_DATA$plot <<- CURRENT_DATA$all[,1:4]
                                        }
          else {
                                    CURRENT_DATA$plot <<- CURRENT_DATA$all
               }
        }
        else
        {
          ## file cannot be accessed
          CURRENT_DATA$all <<- data.frame();
          CURRENT_DATA$plot <<- data.frame();
        }
      })
    }
  })

  ## observe checkbox group input
  observe({
    if( ! is.null(input$Columns) ) {
 #browser()
 isolate({
        if( length(input$Columns)!=ncol(CURRENT_DATA$plot) ) {
          if( length(input$Columns)==0 ) {
            ## nothing selected
            CURRENT_DATA$plot <<- data.frame()
          } else {
            for( i in 1:length(input$Columns) ) {
              if( i==1 ) {
                TMP_DF <- CURRENT_DATA$all[[input$Columns[i]]];
              } else {
                TMP_DF <- cbind(TMP_DF,CURRENT_DATA$all[[input$Columns[i]]]);
              }
            }
            TMP_DF <- data.frame(TMP_DF); colnames(TMP_DF) <- input$Columns
            CURRENT_DATA$plot <<- TMP_DF
          }
        } else {
          ## selection unchanged --> nothing to do here
        }
      })
    } else {
      ## nothing here since dynamic UI part not yet created
    }
  })

  #### plot
  output$plot <- renderPlot({
    DATA <- list("raw_data"=CURRENT_DATA$plot)
    ## extend data depend on model selection
    PlotScatter(DATA=DATA)
```

```
  })

  ##Gaussian
  output$gaussian = renderPlot({
    DATA <- CURRENT_DATA$plot
    sample_size <- 300;
    MODELS <- list();
    X <- DATA;
    MODELS$covmat <- cov(as.matrix(X));
    MODELS$Pearson <- cor(X, method="pearson");
    MODELS$Spearman <- cor(X, method="spearman");
    MODELS$means <- apply(DATA,2,mean);
    SAMPLED <- list();
    SAMPLED$Gaussian <- mvrnorm(n=sample_size,mu=MODELS$means,Sigma=MODELS$covmat);
    DATA_G <- list(
      "raw_data" = DATA,
      "Gaussian_model"=SAMPLED$Gaussian
    )

    if (!is.null(DATA_G[[1]])){
      PlotScatter(DATA=DATA_G)
    }
  })


  ##GLD
  output$gld = renderPlot({
    DATA <- CURRENT_DATA$plot
    sample_size <- 300;
    MODELS <- list();
    X <- DATA;
    MODELS$covmat <- cov(as.matrix(X));
    MODELS$Pearson <- cor(X, method="pearson");
    MODELS$Spearman <- cor(X, method="spearman");
    MODELS$means <- apply(DATA,2,mean);
    MODELS$gld_coeffs <- apply(DATA,2,FitGLD);
    SAMPLED <- list();
    SAMPLED$Gaussian <- mvrnorm(n=sample_size,mu=MODELS$means,Sigma=MODELS$covmat);
    C <- nearPD(2*sin(pi/6*MODELS$Spearman))$mat;
    Z <- mvrnorm(n=sample_size,mu=rep(0,nrow(C)),Sigma=C);
    U <- pnorm(q=Z, mean=0, sd=1)
    SAMPLED$GLD <- SAMPLED$Gaussian;
    for( i in 1:ncol(SAMPLED$GLD) ) {
      SAMPLED$GLD[,i] <- qgl(p=U[,i], lambda1=MODELS$gld_coeffs[,i])
    }
    DATA_F <- list(
      "raw_data" = DATA,
      "Gaussian_model"=SAMPLED$Gaussian,
      "GLD_model"=SAMPLED$GLD
    )
    if (!is.null(DATA_F[[1]])){
      PlotScatter(DATA=DATA_F)
    }
  })

}

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