R Shiny as a Simulation Interface

*Introduction*

This tutorial presents a method of using R Shiny as a user interface for non-R based workflows. In this use case we utilize R Shiny to run a Java based discrete event simulation by editing the configuration file, calling the model execution, and processing the results. This method significantly extends the power of R Shiny as a user interface by allowing us to employ any software or user generated code that is callable from a command line interface (CLI).

*Background*

The nexus for this tutorial is a recent project in which we assisted our sponsor with the analysis of a logistics systems under varied conditions. We wanted to provide the customer with an enduring deliverable that allowed them to explore the system performance under a variety of conditions. In one of the scenarios we found that the system performance was well represented by a linear regression model fitted to the results of our design of experiments outputs.

However, in the second scenario we were unable to find a meta-model that produced a sufficient degree of accuracy. To meet our objectives we developed an R Shiny application that allowed the user to provide the model inputs, call the underlying simulation model, and return the results.

This tutorial uses an M/M/1 Queue example to demonstrate this methodology. We use the M/M/1 closed form equations in place of our linear regression meta-model.

*Step 1 - Environment Configuration*

In this section we will set up the folder structure and download the necessary files to build the R Shiny application.

*Step 1.1 - Create a folder*

Somewhere on your computer create a folder titled **SimModel**. For this example the **SimModel** folder is on my desktop.

*Step 1.2 - Download files.*

This tutorial uses the Java based JaamSim discrete event simulation (DES) software as the underlying simulation modeling environment. We found this free, open-source DES software to be extremely user friendly and well documented.

1. Download the JaamSim .jar file: [JaamSim Download](https://github.com/jaamsim/jaamsim/releases/download/v2017-05/JaamSim2017-05.jar). This link is specifically to the JaamSim2017-05 release to maintain code stability.
2. The base configuration file for the M/M/1 Queue Model and the full setup files can be found in my Git hub directory. <https://github.com/Frisbie17/ShinySimulation>

One of the nice qualities of JaamSim is that the order of the configuration file does not matter. This allows us to take the configuration file for a model, delete the lines that correspond to those we intend to modify thru R, and save the remainder as a base configuration file. Later we will append this base file with the user defined parameters.

*Step 2 - Build the Global file.*

In this section we will cover the material that should be included in the *global.R* file of the Shiny application.

*Step 2.1 - Set the file paths.*

Adjust these as necessary to wherever you placed your **SimModel** folder.

JarFilePath <- "C:\\SimModel\\JaamSim2017-05.jar"

BaseCfgFilePath <- "C:\\SimModel\\MM1\_exp\_base.cfg"

OutputFilePath <- "C:\\SimModel\\MM1\_exp\_app.dat"

CfgOutputFilePath <- "C:\\SimModel\\MM1\_exp\_app.cfg"

*Step 2.2 - Configuration File Function.*

This function takes the file path to our base configuration file, the file path where we will write out the final configuration file, and the user defined parameters. In this example the user defined parameters are the mean inter-arrival time and the mean service time.

makeCfg <- **function**(baseInputFile, outputFileName, meanIAT, meanST){

outFile <- file(outputFileName, "w")

**for** (i **in** baseInputFile){

x <- readLines(i)

writeLines(x[1:(length(x))], outFile)

}

writeLines(paste("ArrivalTimeExpDist Mean { ", meanIAT, " s }"), outFile)

writeLines(paste("ServerTimeExpDist Mean { ", meanST, " s }"), outFile)

close(outFile)

}

*Step 2.3 - Model Call Function.*

This function makes a call to system to pass a formatted command line argument that runs the JaamSim model using the configuration file created in Step 2.2.

runJaamSim <- **function**(JarFilePath, CfgFilePath){

SetCommand <- paste("java -jar ", JarFilePath, " ", CfgFilePath, " -h")

system(SetCommand)

}

*Step 2.4. - Post-processing Function.*

This function takes the file path for the JaamSim .dat output file and returns a list object containing the mean queue length and the mean time in the queue.

readJaamSimOutput <- **function**(outputFilePath){

results.table <- test <- read.table(outputFilePath, header = FALSE, skip =2 , sep="\t")

result.means <- colMeans(results.table)

aveQueLength <- result.means[2]

aveQueTime <- result.means[3]\*60\*60 *#Convert from hours to seconds*

results <- c(aveQueLength, aveQueTime)

**return**(results)

}

*Step 2.5 - Closed form equation functions.*

These function take the same inputs as the simulation call (mean intra-arrival time and mean service time) to return the closed form solutions to the M/M/1 queue model. We present these as an example of using a formulaic representation of a system to provide performance information to a user.

*#L(q) AQL formula*

closedAQL <- **function**(mIAT, mST){

rho <- (1/mIAT)/(1/mST)

Lq <- (rho^2)/(1-rho)

**return**(Lq)

}

*#W(q) AQT formula*

closedAQT <- **function**(AQL, mIAT){

Wq <- AQL/(1/mIAT)

**return**(Wq)

}

*Step 3 - R Demonstration.*

We now have everything we need to run our M/M/1 queue model using R. Before we introduce the added complexities of Shiny we will demonstrate the functionality from R. In the code below we will: 1. Set our two input parameters, 2. Write our configuration file, 3. Run the simulation, 4. Get our results.

*#1. Set Inputs*

meanIntraArrivalTime <- 30

meanServiceTime <- 25

*#2. Write the cfg file*

makeCfg(BaseCfgFilePath, CfgOutputFilePath, meanIntraArrivalTime, meanServiceTime)

*#3. Run the simulation.*

runJaamSim(JarFilePath, CfgOutputFilePath)

*#4. Get the results.*

sim.results <- readJaamSimOutput(OutputFilePath)

sim.meanQueLength <- sim.results[1]

sim.meanQueTime <- sim.results[2]

print(sim.meanQueLength)

## V2

## 4.119611

print(sim.meanQueTime)

## V3

## 123.0141

Now lets compare our simulation results [Note: the simulation configuration is set to do 500 runs.] with the closed form solutions.

closed.meanQueLength <- closedAQL(meanIntraArrivalTime, meanServiceTime)

closed.meanQueTime <- closedAQT(closed.meanQueLength, meanIntraArrivalTime)

print(closed.meanQueLength)

## [1] 4.166667

print(closed.meanQueTime)

## [1] 125

As expected, the simulation and closed form solutions are reasonably close. Again, this comparison shows that in certain circumstances it may be sufficient, or even preferable, to use formulaic model representations over simulation based ones.

*Step 4 - R Shiny.*

In this section we provide the code needed to create a local R Shiny application as a user interface. As a reminder, all of the code snippets from the Step 2 sections should be pasted into a file called **global.R** and saved in the **SimModel** folder. Alternatively, a copy of the **global.R** file is included in the full download referenced in Step 1.2.

The code block below should be pasted into a file called **app.R** and saved in the **SimModel** folder.

**library**(shiny)

**library**(shinydashboard)

**source**("global.R")

*# Define UI for application that draws a histogram*

ui <- dashboardPage(skin = "green",

dashboardHeader(title = "M/M/1 Queue DES Demo"),

dashboardSidebar(

sidebarMenu(id = "tabs",

numericInput('meanIAT',

'Mean Intra-Arrival Time (secs)', 30, 0, 1000, 1),

numericInput('meanST', 'Mean Service Time (secs)', 25, 0, 1000, 1),

actionButton("setModel", "Set Model"),

hr(),

actionButton("runModel", "Run Model"),

hr(),

actionButton("getOutput", "Get Results")

) *#End SidebarMenu*

), *#End dashboardSidebar*

dashboardBody(

h2("Output"),

h4(textOutput("simQLA")),

h4(textOutput("simAQT")),

h4(textOutput("cldQLA")),

h4(textOutput("cldAQT"))

) *#End Body*

) *#End UI*

*# Define server logic required to draw a histogram*

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

*##Download Function for Code Base*

output$download <- downloadHandler(

filename = "MM1\_Example.zip",

*# content is a function with argument file. content writes the plot to the device*

content = **function**(file) {

file.copy("MM1\_Example.zip", file)

},

contentType = "application/zip"

) *#End download function*

*###Make the cfg file*

observeEvent(input$setModel, {

makeCfg(BaseCfgFilePath, CfgOutputFilePath, input$meanIAT, input$meanST)

}) *#END Observe Event make Cfg file*

observeEvent(input$runModel, {

withProgress(message = 'Simulation in progress', value = 0,{

**for** (i **in** 1:20){

incProgress(1/20)

Sys.sleep(.25)

}

})

runJaamSim(JarFilePath, CfgOutputFilePath)

}) *#END Observe Event Run JaamSim Model*

observeEvent(input$getOutput, {

result <- readJaamSimOutput(OutputFilePath)

sQLA <- result[1]

sAQT <- result[2]

cQLA <- closedAQL(input$meanIAT, input$meanST)

cAQT <- closedAQT(cQLA, input$meanIAT)

output$simQLA <- renderText(paste0("Simulation mean queue length is: ",

round(sQLA, 1), "."))

output$simAQT <- renderText(paste0("Simulation mean time in queue is: ",

round(sAQT, 2), " seconds."))

output$cldQLA <- renderText(paste0("Closed form equation mean queue length is: ",

round(cQLA, 1), "."))

output$cldAQT <- renderText(paste0("Closed form equation mean time in queue is: ",

round(cAQT, 2), " seconds."))

})*#End Observe Event Get Model Output*

} *#END SERVER*

*# Run the application*

shinyApp(ui = ui, server = server)

*Conclusion*

In this tutorial we’ve demonstrated how one can use R Shiny as a user interface for a Java based discrete event simulations as a specific use case of a non-R based workflow. This option expands the realm of the possible for R Shiny applications and allows use to incorporate and leverage many tools beyond R and R packages to meet customer requirements.