Making Health Economic Models Shiny: A Tutorial

Robert Smith[1] and Paul Schneider[1]

[1]**School of Health and Related Research, University of Sheffield, Regents Court, S1 4DA**

Corresponding author: Robert Smith (rasmith3@sheffield.ac.uk)

Abstract

Health economic evaluation models have traditionally been built in Microsoft Excel, but more sophisticated tools are increasingly being used as model complexity and computational requirements increase. Of all the programming languages, R is most popular amongst health economists because it has a plethora of user created packages and is highly flexible. However, even with an integrated development environment such as R Studio, R lacks a simple point and click user interface and therefore requires some programming ability. This might make the switch from Microsoft Excel to R seem daunting, and it might make it difficult to directly communicate results with decisions makers and other stakeholders.

The R package Shiny has the potential to resolve this limitation. It allows programmers to embed health economic models developed in R into interactive web browser based user interfaces. Users can specify their own assumptions about model parameters and run different scenario analyses, which, in case of regular a Markov model, can be computed within seconds. This paper provides a tutorial on how to wrap a health economic model built in R into a Shiny application. We use a 4 state Markov model developed by the Decision Analysis in R for Technologies in Health (DARTH) group as a case-study to demonstrate main principles and basic functionality.

A more extensive tutorial, all code, and data are provided in a GitHub repository.

Keywords

markov model, R, RShiny, Decision Science

Introduction

As the complexity of health economic models increase, there is growing recognition of the advantages of using high level programming languages to support statistical analysis (e.g. R, Python, C++, Julia). Depending on the model that is being used, Microsoft Excel can be relatively slow. Certain types of models (e.g. individual-level simulations) can take a very long time to run or become computationally infeasible, and some essential statistical methods can hardly be implemented at all (e.g. survival modelling, network meta analysis, value of sample information), or rely on exporting results from other programs (e.g. R, STATA, WinBUGs).

Of all the high level programming languages, R is the most popular amongst health economists [1]. R is open source and supported by a large community of statisticians, data scientists and health economists. There are extensive collections of (mostly free) online resources, including packages, tutorials, courses, and guidelines. Chunks of code, model functions, and entire models are shared by numerous authors, which allow R users to quickly adopt and adapt methods and code created by others. Importantly for the UK, R is also currently the only programming environment accepted by NICE for HTA submissions, the alternative submission formats Excel, DATA, Treeage, and WinBUGs are all software applications [2].

Despite the many strengths of the script based approach (e.g R) to decision modelling, an important limitation has been the lack of an easy-to-understand user-interface which would be useful as it "facilitates the development and communication of the model structure" (p.743) [1]. While it is common practice for 'spreadsheet models' to have a structured front tab, which allows decision makers to manipulate model assumptions and change parameters to assess their impact on the results, up until recently, R models had to be adapted within script files or command lines.

Released in 2012, Shiny is an R-package which can be used to create a graphical, web browser based interface. The result looks like a website, and allows users to interact with underlying R models without the need to manipulate the source code [3]. Shiny has already been widely adopted in many different areas and by various organisations to present the results of statistical analysis [4]. Within health economics Shiny is currently being used to conduct network meta analysis [5] and value of information analysis [6, 7].

Using Shiny, it is possible to create flexible user interfaces which allow users to specify different assumptions, change parameters, run underlying R code and visualise results. The primary benefit of this is that it makes script based computer models accessible to those with no programming knowledge - opening models up to critical inquiry from decision makers and other stakeholders [8]. Other benefits come from leveraging the power of R's many publicly available packages, for example allowing for publication quality graphs and tables to be downloaded, user specific data-files uploaded, open-access data automatically updated and, perhaps most importantly, to efficiently run comprehensive probabilistic sensitivity analyses in a fraction of the time that it would take in Microsoft Excel. Shiny web applications for R health economic models seem particularly useful in cases where model parameters are highly uncertain or unknown, and where analysis may want to be conducted with heterogeneous assumptions (e.g. for different populations). Once an R model and a Shiny application have been created, they can also be easily adapted, making it possible to quickly update the model when new information becomes available.

While, from a transparency perspective, it is preferable that models constructed in R are made open-access to improve replicability and collaboration, it is not a requirement [9]. Sensitive and proprietary data and/or models can be shared internally, or through password-protected web applications, negating the need to email zipped folders.

Several authors have postulated that there is considerable potential in using Shiny to support and improve health economic decision making. Incerti et al. (2019) identified web applications as being an essential part of modelling, stating that they "believe that the future of cost-effectiveness modeling lies in web apps, in which graphical interfaces are used to run script-based models" (p. 577) [10]. Similarly, Baio and Heath (2017) predicted that R Shiny web apps will be the "future of applied statistical modelling, particularly for cost-effectiveness analysis" (p.e5) [11]. Despite these optimistic prognoses, adoption of R in health economics has been slow and the use of Shiny seems to have been limited to only a few cases. A reason for this might be the lack of accessible tutorials tailored towards an economic modeller audience.

Here, we provide a simple example of a Shiny web app, using a general 4-state Markov model. The model is based on the 'Sick-Sicker model', which has been described in detail in previous publications [12, 13] and in open source teaching materials by the DARTH workgroup [14]. The model was slightly adapted to implement probabilistic sensitivity analysis.

Methods

While the focus of this tutorial is on the application of Shiny for health economic models, below we provide a brief overview of the "Sick-Sicker model". For further details, readers are encouraged to consult previous publications by the DARTH group [12, 13, 15] and the DARTH group website [14].

The Sick-Sicker model is a 4 state (Healthy, Sick, Sicker or Dead) time-independent Markov model. The cohort progresses through the model in cycles of equal duration, with the proportion of those in each health state in the next cycle being dependant on the proportion in each health state in the current cycle and a time constant transition probability matrix.

The analysis incorporates probabilistic sensitivity analysis (PSA) by creating a data-frame of PSA inputs (one row

being one set of model inputs) based on cost, utility and state transition probability distributions using the function *f_gen_psa* and then running the model for each set of PSA inputs using the model function *f_MM_sicksicker*. We therefore begin by describing the two functions *f_gen_psa* and *f_MM_sicksicker* in more detail before moving on to demonstrate how to create a user-interface. Note that we add to the coding framework from Alarid-Escudero et al. (2019) to use the *f_* prefix for functions [12].

Functions

The *f_gen_psa* function (see Listing. 6 in the appendix) returns a data-frame of probabilistic sensitivity analysis inputs: transition probabilities between health states using a beta distribution, hazard rates using a log-normal distribution, costs using a gamma distribution and utilities using a truncnormal distribution. It relies on two inputs, the number of simulations (PSA inputs), and the cost (which takes a fixed value). We set the defaults to 1000 and 50 respectively.
NOTE: in order to use the *rtruncnorm* function the user must first install and load the *truncnorm* package using *install.packages()* and *library()*.

Running the model for a specific set of PSA inputs

The function *f_MM_sicksicker* (see Listing 7 in the appendix) makes use of the *with* function which applies an expression (in this case the rest of the code) to a data-set (in this case params, which will be a row of PSA inputs). It uses the *params* (one row of PSA inputs) to create a transition probability matrix *m_P*, and then moves the cohort through the simulation one cycle at a time, recording the proportions in each health state in a Markov trace *m_TR* and applying the transition matrix to calculate the proportions in each health state in the next period *m_TR[t+1,]*. The function returns a vector of five results: Cost with no treatment, Cost with treatment, QALYs with no treatment and QALYs with treatment and an ICER. In this simple example treatment only influences utilities and costs, not transition probabilities

Creating the model wrapper

When using a web application it is likely that the user will want to be able to change parameter inputs and rerun the model. In order to make this simple, we recommend wrapping the entire model into a function. We call this function *f_wrapper*, using the prefix *f_* to denote that this is a function.
The wrapper function has as its inputs all the parameters which we may wish to vary using R-Shiny. We set the default values to those of the base model in any report/publication. The model then generates PSA inputs using the *f_gen_psa* function, creates an empty table of results, and runs the model for each set of PSA inputs (a row from *df_psa*) in turn. The function then returns the results in the form of a data-frame with n=5 columns and n=psa rows. The columns contain the costs and QALYs for treatment and no treatment for each PSA run, as well as an ICER for that PSA run.

Listing 1. Model wrapper function

```
f_wrapper <- function(

#-- User adjustable inputs --#

# age at baseline
n_age_init = 25,
# maximum age of follow up
n_age_max  = 110,
# discount rate for costs and QALYS
d_r      = 0.035,
# number of simulations
n_sim    = 1000,
# cost of treatment
c_Trt    = 50

 ){


#-- Unadjustable inputs --#

#  number of cycles
n_t <- n_age_max - n_age_init
# the 4 health states of the model:
v_n <- c("H", "S1", "S2", "D")
# number of health states
n_states <- length(v_n)

#-- Create PSA Inputs --#

df_psa <- f_gen_psa(n_sim = n_sim,
                    c_Trt =  c_Trt)

#--  Run PSA  --#

# Initialize matrix of results outcomes
m_out <- matrix(NaN,
                nrow = n_sim,
                ncol = 5,
                dimnames = list(1:n_sim,
                c("Cost_NoTrt", "Cost_Trt",
                  "QALY_NoTrt", "QALY_Trt",
                  "ICER")))

# run model for each row of PSA inputs
for(i in 1:n_sim){

 # store results in row of results matrix
 m_out[i,] <- f_MM_sicksicker(df_psa[i, ])

  } # close model loop


#-- Return results --#

# convert matrix to dataframe (for plots)
 df_out <- as.data.frame(m_out)

# output the dataframe from the function
 return(df_out)

 } # end of function
```

Integrating into R-Shiny

The next step is to integrate the model function into a Shiny web-app. This is done within a single R file, which we call *app.R*. This can be found within the GitHub repository here.
The app.R script has three main parts, each are addressed in turn below: - set-up (getting everything ready so the user-interface and server can be created) - user interface (what people will see) - server (R code running in the background)

Initial set-up

The set-up is relatively simple, load the R-Shiny package from your library so that you can use the *shinyApp* function. The next step is to use the *source* function in baseR to run the script which creates the *f_wrapper* function, being careful to ensure your relative path is correct ('./wrapper.R' should work if the wrapper.R file is in the same folder as the app.R file).

Listing 2. Code initialization (within app.R)

```
# install 'shiny' if haven't already.
# install.packages("shiny")

# Load 'shiny from the library.
 library(shiny)

# source the wrapper function.
 source("./wrapper.R")
```

Creating the User Interface function

The user interface is extremely flexible, we show the code for a very simple structure (fluidpage) with a sidebar containing inputs and a main panel containing outputs. We have done very little formatting in order to minimize the quantity of code while maintaining basic functionality. In order to get an aesthetically pleasing application we recommend much more sophisticated formatting, relying on CSS, HTML and Javascript.
The example user interface displayed in Figure 1 and online on this website. The user interface is a *fluidpage* in a *sidebarLayout* (other types of layout are available). The *sidebarLayout* is made up of two components, a titlepanel and a sidebar layout display (which itself is split into a sidebar and a main panel). This is a basic structure used for teaching purposes, there are a plethora of templates available online.
The title panel contains the title "Sick Sicker Model in Shiny", the sidebar panel contains two numeric inputs and a slider input ("Treatment Cost", "PSA runs", "Initial Age") and an Action Button ("Run / update model").
The values of the inputs have ID tags (names) which are recognised and used by the server function, we denote these with the prefix "SI" to indicate they are 'Shiny Input' objects (*SI_c_Trt*, *SI_n_sim*, *SI_n_age_init*). Note that this is an addition of the coding framework provided by Alarid-Escudero et al., (2019).
The action button also has an id, this is not an input into the model wrapper *f_wrapper* so we leave out the SI and call it *run_model*.
The main panel contains two objects which have been output from the server: *tableOutput("SO_icer_table")* is a table of results, and *plotOutput("SO_CE_plane")* is a cost-effectiveness plane plot. It is important that the format (e.g. *tableOutput*) matches the format of the object from the server (e.g. *SO_icer_table*). Again, the *SO* prefix reflects the fact that these are Shiny Outputs. The two *h3()* functions are simply headings which appear as "Results Table" and "Cost-effectiveness Plane".

Listing 3. Shiny user interface function

```
ui <- fluidPage(    # creates empty page

  # title of app
  titlePanel("Sick Sicker Model in Shiny"),

  # layout is a sidebar-layout
  sidebarLayout(

    sidebarPanel( # open sidebar panel

# input type numeric
    numericInput(inputId = "SI_c_Trt",
                 label = "Treatment Cost",
                 value = 200,
                 min = 0,
                 max = 400),

    numericInput(inputId = "SI_n_sim",
                 label = "PSA runs",
                 value = 1000,
                 min = 0,
                 max = 400),

# input type slider
    sliderInput(inputId = "SI_n_age_init",
                label = "Initial Age",
                value = 25,
                min = 10,
                max = 80),

# action button runs model when pressed
    actionButton(inputId = "run_model",
                 label   = "Run model")

                ),  # close sidebarPanel

# open main panel
  mainPanel(

# heading (results table)
    h3("Results Table"),

# tableOutput id = icer_table, from server
    tableOutput(outputId = "SO_icer_table"),

# heading (Cost effectiveness plane)
    h3("Cost-effectiveness Plane"),

# plotOutput id = SO_CE_plane, from server
    plotOutput(outputId = "SO_CE_plane")

            ) # close mainpanel
```

```
      ) # close sidebarlayout

  ) # close UI fluidpage
```

Creating the Server function

The server is marginally more complicated than the user interface. It is created by a function with inputs and outputs. The observe event indicates that when the action button *run_model* is pressed the code within the curly brackets is run. The code will be re-run if the button is pressed again.

The first thing that happens when the *run_model* button is pressed is that the model wrapper function *f_wrapper* is run with the user interface inputs (*SI_c_Trt*, *SI_n_age_init*, *SI_n_sim*) as inputs to the function. The *input* prefix indicates that the objects have come from the user interface. The results of the model are stored as the data-frame object *df_model_res*.

The ICER table is then created and output (note the prefix *output*) in the object *SO_icer_table*. See previous section on the user interface and note that the *tableOutput* function has as an input *SO_icer_table*. The function *renderTable* rerenders the table continuously so that the table always reflects the values from the data-frame of results created above. In this simple example we have created a table of results using code within the script. Normally we would generally use a custom function which creates a publication quality table which is aesthetically pleasing. There are numerous packages which provide this functionality [16, 17, 12].

The cost-effectiveness plane is created in a similar process, using the *renderPlot* function to continuously update a plot which is created using baseR plot function using incremental costs and QALYs calculated from the results dataframe *df_model_res*. For aesthetic purposes we recommend this is replaced by a ggplot2 or plotly plot which have much improved functionality [18, 19]. As with the results table, there are also numerous health economic modelling specific R packages which have plotting features [16, 17, 12].

Listing 4. Shiny server function

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

# when action button pressed ...
 observeEvent(input$run_model,
              ignoreNULL = F, {

# Run model function with Shiny inputs
 df_model_res = f_wrapper(
        c_Trt = input$SI_c_Trt,
        n_age_init = input$SI_n_age_init,
        n_sim = input$SI_n_sim)

#-- CREATE COST EFFECTIVENESS TABLE --#

# renderTable continuously updates table
 output$SO_icer_table <- renderTable({

 df_res_table <- data.frame( # create dataframe
 Option =  c("Treatment","No Treatment"),

 QALYs  =  c(mean(df_model_res$QALY_Trt),
             mean(df_model_res$QALY_NoTrt)),

 Costs  =  c(mean(df_model_res$Cost_Trt),
             mean(df_model_res$Cost_NoTrt)),

 Inc.QALYs = c(mean(df_model_res$QALY_Trt) -
               mean(df_model_res$QALY_NoTrt),
               NA),

 Inc.Costs = c(mean(df_model_res$Cost_Trt) -
               mean(df_model_res$Cost_NoTrt),
               NA),

 ICER = c(mean(df_model_res$ICER), NA)

 ) # close data-frame

# round the data-frame to two digits
df_res_table[,2:6] = round(
        df_res_table[,2:6],digits = 2)

# print the results table
  df_res_table

  }) # table plot end.


#--  CREATE COST EFFECTIVENESS PLANE --#

# render plot repeatedly updates.
output$SO_CE_plane <- renderPlot({

# calculate incremental costs and qalys
df_model_res$inc_C <- df_model_res$Cost_Trt -
                      df_model_res$Cost_NoTrt

df_model_res$inc_Q <- df_model_res$QALY_Trt -
                      df_model_res$QALY_NoTrt

# create cost effectiveness plane plot

plot(
# x y are incremental QALYs Costs
x = df_model_res$inc_Q,
y = df_model_res$inc_C,

# label axes
xlab = "Incremental QALYs",
ylab = "Incremental Costs",

# set x-limits and y-limits for plot.
xlim = c( min(df_model_res$inc_Q,
              df_model_res$inc_Q*-1),
          max(df_model_res$inc_Q,
              df_model_res$inc_Q*-1)),

ylim = c( min(df_model_res$inc_C,
              df_model_res$inc_C*-1),
          max(df_model_res$inc_C,
              df_model_res$inc_C*-1)),

# include y and y axis lines.
abline(h = 0,v=0)

 ) # CE plot end

 }) # renderplot end

 }) # Observe event end
```

```
} # Server end
```

Running the app

The app can be run within the R file using the function *shinyApp* which depends on the *ui* and *server* which have been created and described above. Running this creates a Shiny application in the local environment (e.g. your desktop). It is also possible to deploy the application onto the web from RStudio using the shinyapps.io server (using the publish button in the top right corner of the R-file in R-Studio). Alternatively apps can be hosted on private servers and integrated into existing websites. A step by step guide to the process of publishing applications can be found on the R-Shiny website or other online resources [20, 3].

Listing 5. Running the app

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

Additional Functionality

The example Sick-Sicker web-app which has been created is a simple, but functional, R-Shiny user interface for a health economic model. There are a number of additional functionalities, many of which are covered in an online book by Hadley Wickham [20].

- fully customised user interface aesthetics. Since the user interface is translated into HTML and CSS it is possible to customise all components (such as colors, fonts, graphics, layouts and backgrounds) [21, 3].
- leverage many popular R packages to visualise model inputs (e.g. distributions) and outputs (e.g. plots and results tables) [18, 19, 22].
- upload files containing input parameters and data to the app [20].
- download specific figures and tables from the app [20].
- create a downloadable full report including model inputs and outputs [20].
- send model results/report to an email address once the model has finished running [23].

It is also possible to integrate all of the steps of health economic evaluation into one program. After selecting a subgroup of studies to use as inputs for a network meta-analysis, and economic model assumptions, the user would be required to simply click a 'run' button. They would then be presented with results of the network meta-analysis, economic model and value of information analysis in one simple user-interface. The app user would then also be able to download a report (or have it sent to an email address) with the model results and appropriate visualisations updated to reflect their assumptions. We believe this is the future of health economics.

Discussion

In this paper, we demonstrated how to generate a user-friendly interface for an economic model programmed in R, using the Shiny package. This tutorial shows that the process is relatively simple and requires limited additional programming knowledge than that required to build a decision model in R.

The movement towards script based health economic models with web based user interfaces is particularly useful in situations where a general model structure has been created with a variety of stakeholders in mind, each of which may have different assumptions (input parameters) and wish to conduct sensitivity analysis specific to their decision. For example the World Health Organisation Department of Sexual and Reproductive Health and Research recently embedded a Shiny application into their website. The application runs a heemod model [16] in R in an external server, and allows users to select their country and specify country specific assumptions (input parameters), run the model and display results. The process of engagement, the ability to 'play' with the model and test the extremes of the decision makers' assumptions gives stakeholders more control over models, making them feel less like black boxes. While there is a danger that a misinformed stakeholder may make a mistake in their choice of parameter, it is simple to limit the range that parameter inputs can take a feasible range.

The authors' experience of creating user-interfaces for decision models has led to the conclusion that the most efficient method is to work iteratively, starting with a very simple working application, and adding functionality step by step, testing the app at each iteration to ensure it works as intended.

There are several challenges that exist with the movement toward script based models with web-based user-interfaces. The first is the challenge of up-skilling health economists used to working in Microsoft Excel. We hope that this tutorial provides a useful addition to previous tutorials demonstrating how to construct decision models in R [13]. A second, and crucial challenge to overcome, is a concern about deploying highly sensitive data and methods to an external server. While server providers such as ShinyIO provide assurances of SSR encryption and user authentication clients with particularly sensitive data may still have concerns. This problem can be avoided in two ways: firstly if clients have their own server and the ability to deploy applications they can maintain control of all data and code, and secondly the application could simply not be deployed, and instead simply created during a meeting using code and data shared in a zip file. Finally, a challenge (and opportunity) exists to create user-interfaces that are most user-friendly for decision makers in this field, this is an area of important research which can be used to inform teaching content for years to come.

Conclusion

The creation of web application user interfaces for health economic models constructed in high level programming languages should improve their usability, allowing stakeholders and third parties with no programming knowledge to conduct their own sensitivity analysis remotely. This tutorial provides a reference for those attempting to create a user interface for a health economic model created in R. Further work is necessary to better understand how to design interfaces which best meet the needs of different decision makers.

Author contributions

R.S. and P.S. contributed equally to all sections of this publication.

Competing interests

R.S. and P.S. have no competing interests.

Grant information

R.S. and P.S. are joint funded by the Wellcome Trust Doctoral Training Centre in Public Health Economics and Decision Science [108903] and the University of Sheffield.

Acknowledgements

We would like to thank Gianluca Baio, Nathan Green and all participants in the pilot tutorial held at the University of Sheffield for comments. All errors are the responsibility of the authors.

References

[1] Hawre Jalal, Petros Pechlivanoglou, Eline Krijkamp, Fernando Alarid-Escudero, Eva Enns, and MG Myriam Hunink. An overview of r in health decision sciences. *Medical decision making*, 37(7):735–746, 2017.

[2] National Institute for Health and Care Excellence (Great Britain). *Guide to the processes of technology appraisal*. National Institute for Health and Care Excellence, 2014.

[3] Chris Beeley. *Web application development with R using Shiny*. Packt Publishing Ltd, 2016.

[4] Jay Gendron. *Introduction to R for Business Intelligence*. Packt Publishing Ltd, 2016.

[5] Rhiannon K Owen, Naomi Bradbury, Yiqiao Xin, Nicola Cooper, and Alex Sutton. Metainsight: An interactive web-based tool for analyzing, interrogating, and visualizing network meta-analyses using r-shiny and netmeta. *Research synthesis methods*, 2019.

[6] Mark Strong, Jeremy E Oakley, and Alan Brennan. Estimating multiparameter partial expected value of perfect information from a probabilistic sensitivity analysis sample: a nonparametric regression approach. *Medical Decision Making*, 34(3):311–326, 2014.

[7] Gianluca Baio, Andrea Berardi, and Anna Heath. Bceaweb: A user-friendly web-app to use bcea. In *Bayesian Cost-Effectiveness Analysis with the R package BCEA*, pages 153–166. Springer, 2017.

[8] Jeroen P Jansen, Devin Incerti, and Mark T Linthicum. Developing open-source models for the us health system: practical experiences and challenges to date with the open-source value project. *PharmacoEconomics*, 37(11):1313–1320, 2019.

[9] Anthony J Hatswell and Fleur Chandler. Sharing is caring: the case for company-level collaboration in pharmacoeconomic modelling. *PharmacoEconomics*, 35(8):755–757, 2017.

[10] Devin Incerti, Howard Thom, Gianluca Baio, and Jeroen P Jansen. R you still using excel? the advantages of modern software tools for health technology assessment. *Value in Health*, 22(5):575–579, 2019.

[11] Gianluca Baio and Anna Heath. When simple becomes complicated: why excel should lose its place at the top table, 2017.

[12] Fernando Alarid-Escudero, Eline M Krijkamp, Petros Pechlivanoglou, Hawre Jalal, Szu-Yu Zoe Kao, Alan Yang, and Eva A Enns. A need for change! a coding framework for improving transparency in decision modeling. *PharmacoEconomics*, 37(11):1329–1339, 2019.

[13] Fernando Alarid-Escudero, Eline M Krijkamp, Eva A Enns, MG Hunink, Petros Pechlivanoglou, and Hawre Jalal. Cohort state-transition models in r: From conceptualization to implementation. *arXiv preprint arXiv:2001.07824*, 2020.

[14] DARTH Workgroup. Decision analysis in r for technologies in health, 2020. URL `http://darthworkgroup.com/`.

[15] Eline M Krijkamp, Fernando Alarid-Escudero, Eva A Enns, Hawre J Jalal, MG Myriam Hunink, and Petros Pechlivanoglou. Microsimulation modeling for health decision sciences using r: a tutorial. *Medical Decision Making*, 38 (3):400–422, 2018.

[16] A Filipovic-Pierucci, K Zarca, and I Durand-Zaleski. Markov models for health economic evaluation modelling in r with the heemod package. *Value in Health*, 19(7): A369, 2016.

[17] Baio, G, Berardi, A, Heath, and A. *Bayesian Cost-Effectiveness Analysis with the R package BCEA*. Springer, New York, NY, Jul 2017. ISBN 978-3-319-55718-2. doi: 10.1007/978-3-319-55718-2. URL `http://www.springer.com/us/book/9783319557168`.

[18] Carson Sievert. *plotly for R*, 2018. URL `https://plotly-r.com`.

[19] Hadley Wickham. *ggplot2: Elegant Graphics for Data Analysis*. Springer-Verlag New York, 2016. ISBN 978-3-319-24277-4. URL `https://ggplot2.tidyverse.org`.

[20] Hadley Wickham. Mastering shiny, 2020. URL `https://mastering-shiny.org/index.html`.

[21] Shiny. Build your entire ui with html, 2020. URL `https://shiny.rstudio.com/articles/html-ui.html`.

[22] Matt Dowle and Arun Srinivasan. *data.table: Extension of 'data.frame'*, 2019. URL `https://CRAN.R-project.org/package=data.table`. R package version 1.12.8.

[23] Jim Hester. *gmailr: Access the 'Gmail' 'RESTful' API*, 2019. URL `https://CRAN.R-project.org/package=gmailr`. R package version 1.0.0.

Appendix

Listing 6. Function to Generate PSA Inputs

```
f_gen_psa <- function(n_sim = 1000, c_Trt = 50){

  df_psa <- data.frame(

# Transition probabilities (per cycle)

    # prob Healthy -> Sick
    p_HS1   = rbeta(n = n_sim,
                    shape1 =  30,
                    shape2 =  170),

    # prob Sick    -> Healthy
    p_S1H   = rbeta(n = n_sim,
                    shape1 =  60,
                    shape2 =  60) ,

    # prob Sick    -> Sicker
    p_S1S2  = rbeta(n = n_sim,
                    shape1 =  84,
                    shape2 =  716),

    # prob Healthy -> Dead
    p_HD    = rbeta(n = n_sim,
                    shape1 =  10,
                    shape2 =  1990),

    # rate ratio death S1 vs healthy
    hr_S1   = rlnorm(n = n_sim,
                     meanlog =  log(3),
                     sdlog =   0.01),

    # rate ratio death S2 vs healthy
    hr_S2   = rlnorm(n = n_sim,
                     meanlog =  log(10),
                     sdlog =  0.02),

# Cost vectors with length n_sim

    # cost p/cycle in state H
    c_H   = rgamma(n = n_sim,
                   shape = 100,
                   scale = 20),

    # cost p/cycle in state S1
    c_S1  = rgamma(n = n_sim,
                   shape = 177.8,
                   scale = 22.5),

    # cost p/cycle in state S2
    c_S2  = rgamma(n = n_sim,
                   shape = 225,
                   scale = 66.7),

    # cost p/cycle in state D
    c_D   = 0,

    # cost p/cycle of treatment
    c_Trt = c_Trt,

# Utility vectors with length n_sim

    # utility when healthy
    u_H   = rtruncnorm(n = n_sim,
                       mean = 1,
                       sd = 0.01,
                       b = 1),

    # utility when sick
    u_S1  = rtruncnorm(n = n_sim,
```

```
                      mean = 0.75,
                      sd = 0.02,
                      b = 1),

    # utility when sicker
    u_S2  = rtruncnorm(n = n_sim,
                      mean = 0.50,
                      sd = 0.03,
                      b = 1),
    # utility when dead
    u_D   = 0,

    # utility when being treated
    u_Trt = rtruncnorm(n = n_sim,
                      mean = 0.95,
                      sd = 0.02,
                      b = 1)
  )

  return(df_psa)
}
```

Listing 7. Model function

```
f_MM_sicksicker <- function(params) {

# run following code with a set of data
  with(as.list(params), {

# rate of death in healthy
r_HD    = - log(1 - p_HD)
# rate of death in sick
r_S1D   = hr_S1 * r_HD
# rate of death in sicker
r_S2D   = hr_S2 * r_HD
# probability of death in sick
p_S1D   = 1 - exp(-r_S1D)
# probability of death in sicker
p_S2D   = 1 - exp(-r_S2D)

# calculate discount weight for each cycle
v_dwe <- v_dwc <- 1 / (1 + d_r) ^ (0:n_t)

#transition probability matrix for NO treatment
m_P <- matrix(0,
              nrow = n_states,
              ncol = n_states,
              dimnames = list(v_n, v_n))

# fill in the transition probability array

### From Healthy
m_P["H", "H"]  <- 1 - (p_HS1 + p_HD)
m_P["H", "S1"] <- p_HS1
m_P["H", "D"]  <- p_HD

### From Sick
m_P["S1", "H"]  <- p_S1H
m_P["S1", "S1"] <- 1 - (p_S1H + p_S1S2 + p_S1D)
m_P["S1", "S2"] <- p_S1S2
m_P["S1", "D"]  <- p_S1D

### From Sicker
m_P["S2", "S2"] <- 1 - p_S2D
m_P["S2", "D"]  <- p_S2D

### From Dead
m_P["D", "D"] <- 1

# create empty Markov trace
m_TR <- matrix(data = NA,
               nrow = n_t + 1,
               ncol = n_states,
               dimnames = list(0:n_t, v_n))

# initialize Markov trace
m_TR[1, ] <- c(1, 0, 0, 0)

############ PROCESS ####################

for (t in 1:n_t){ # throughout the number of cycles
# estimate next cycle (t+1) of Markov trace
 m_TR[t + 1, ] <- m_TR[t, ] %*% m_P
  }

########## OUTPUT  ####################

# create vectors of utility and costs for each state
v_u_trt    <- c(u_H, u_Trt, u_S2, u_D)

v_u_no_trt <- c(u_H, u_S1, u_S2, u_D)

v_c_trt    <- c(c_H, c_S1 + c_Trt,
                    c_S2 + c_Trt, c_D)

v_c_no_trt <- c(c_H, c_S1, c_S2, c_D)

# estimate mean QALYs and costs
 v_E_no_trt <- m_TR %*% v_u_no_trt

 v_E_trt    <- m_TR %*% v_u_trt

 v_C_no_trt <- m_TR %*% v_c_no_trt

 v_C_trt    <- m_TR %*% v_c_trt

### discount costs and QALYs
#   1x31 %*% 31x1 -> 1x1

te_no_trt <- t(v_E_no_trt) %*% v_dwe

 te_trt    <- t(v_E_trt) %*% v_dwe

 tc_no_trt <- t(v_C_no_trt) %*% v_dwc

 tc_trt    <- t(v_C_trt)    %*% v_dwc

 results <- c(
   "Cost_NoTrt" = tc_no_trt,
   "Cost_Trt"   = tc_trt,
   "QALY_NoTrt" = te_no_trt,
   "QALY_Trt" = te_trt,
   "ICER" = (tc_trt - tc_no_trt)/
               (te_trt - te_no_trt)
   )

   return(results)
 }) end with function

} # end f_MM_sicksicker function
```

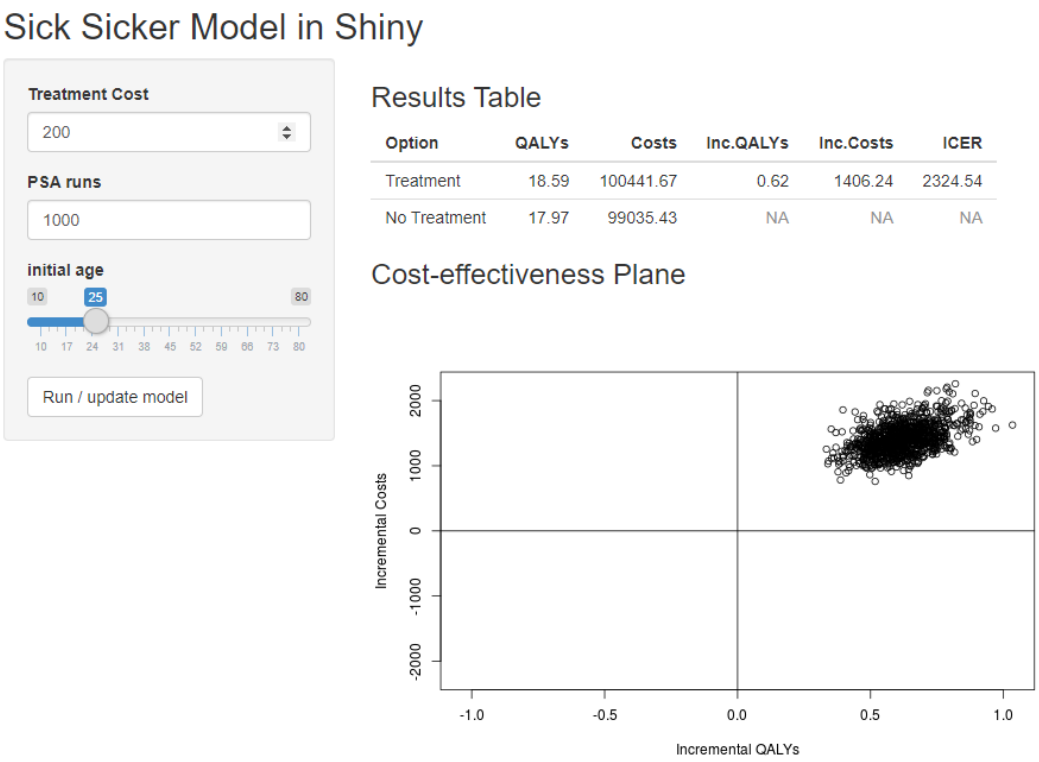

Figure 1. Screen-print of Sick-Sicker model user interface

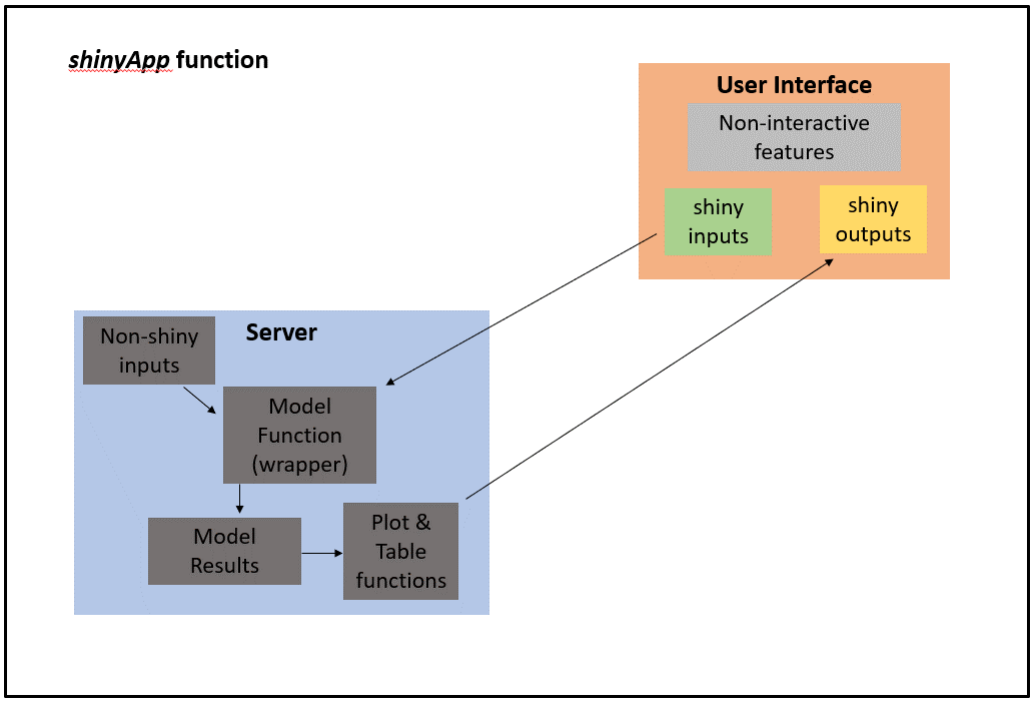

Figure 2. Diagram depicting how the Sick-Sicker app is structured