

## Project Report- A shiny app

You can find our shiny app here: <https://github.com/YKakdas/Root-Finding-Shiny-App.git>

### Introduction

Root Finding is used in various fields such as Finance, Mathematics, or biological science. Not only the applications but also the algorithms to solve root-finding problems are diverse. The goal of our shiny application is to allow the user to test some of the fastest and most efficient algorithms to find the roots of any function. Thus, we chose to include six approaches in our application.

To develop the app, we used Shiny, an R package that makes it easy to build interactive web apps. R gives us the required flexibility and the opportunity to develop a clearly structured user interface.

### Methods description

Our main goal for the shiny application was to give the user as much flexibility as possible to test different algorithms and their solutions. Therefore, we created a dashboard where users can navigate the respective algorithm interfaces. The gifs give a better understanding of what each algorithm is doing.

We implemented a clear structure in our code. The respective R.files only contain the necessary information, and refer to another file for further description.

We defined seven files for various reasons:

- App.R: the environment of our shiny app with all the relevant information and the user interface to run the app
- Global.R: installs packages, loads them, and sources all necessary R.files
- Servers: creates servers for all of our algorithms and defines required in-/output
- Pages: defines the user interface within the websites
- Dashboard.R: defines the user interface and the menu of our main page
- Algorithms: describes the mathematical computation of every algorithm

The following describes our app's basic structure and the special features we implemented.

First, the app.R file refers to our global. R document. In our global file, we implemented “new packages” to automatically install all packages needed if a user wants to use our application. That helps us ensure that there are no files on the other end.

First, we load all the packages and then source the server of every algorithm with the “source” command.

The servers are defined in the “servers” file. “eventReactive” allows us to create a calculated value that only updates in response to an event. We also implemented a statement that states that the algorithm failed to converge if the result should be null. If needed, we also automatically evaluate the first and second derivatives of  $f$ . Therefore, we use the built-in function in R ( $D()$ ). Additionally, the user can define the function he wants to find the root of. That ensures more flexibility as they can use any function that the algorithm can calculate.

Additionally, we source the respective pages in our global. R file. In those, we have defined how our pages should look like.

The following “source” command in our global file is the `source('dashboard.R')`. Our dashboard is crucial as it defines our user interface's main features. We implemented a sidebar that allows the user to navigate by clicking on each algorithm on the left. The code “menuItem” will enable us to add all our algorithms. Furthermore, we added our main body with “tabItem” and gave all our algorithms their special boxes on the main page. Finally, the “dashboardPage” command defines our heading, the sidebar, and the body.

Our algorithms are defined under “algorithm.R.” So, you can, for example, find the Newton algorithm under `newton.R`. We used four algorithms that we already discussed in class because we saw their importance and advantages of each of them. We added two more algorithms to our shiny app to get an even better variety.

The first algorithm we added is the regula falsi or false position method. It is a numerical method for estimating the roots of polynomials. It is similar to the bisection method, except that the midpoint in the Bisection Method is replaced by a value  $x$ . It is one of the oldest approaches and was founded to speed up the conversion of the bisection method. The pros and cons are very similar to the Bisection Method as it still converges comparably slowly, can't find the root of some equations such as  $x^2$ , and has a linear convergence rate.

The second algorithm we added is Halley's method which is a root-finding algorithm used for functions of one real variable with a continuous second derivative. It can be seen as an extension of Newton's method that incorporates the second derivative of the target function. Thus, Halley's Method needs second-order information but converges fast.

## Project Structure & Technical Specs

Particular importance to organizing the code to make it fine-grained has been given so that the readability increases and debugging cost reduces since it is easier to find which fine-grained part is responsible for the bug. The following file structure shows the project's modularization system. The following section defines what each module is responsible for in detail.

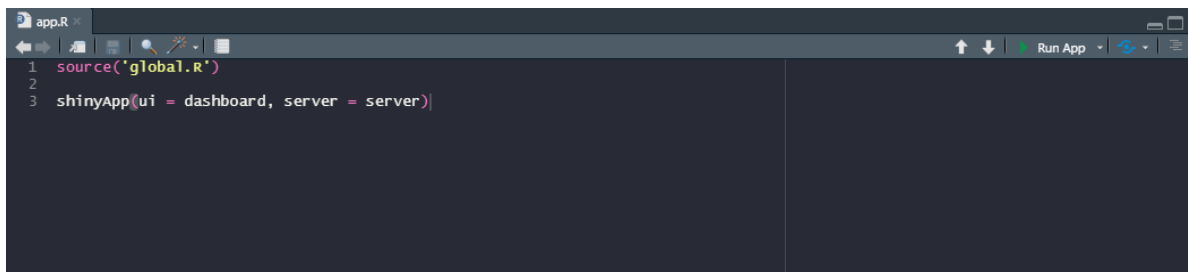
app

- └─ algorithms
  - └─ bisection.R
  - └─ fixedpoint.R
  - └─ halley.R
  - └─ newton.R
  - └─ regula.R
  - └─ secant.R
- └─ html
  - └─ bisection.html
  - └─ fixedpoint.html
  - └─ halley.html
  - └─ newton.html
  - └─ regula.html
  - └─ secant.html
- └─ jsHelper
  - └─ bisection\_js\_helper.R
  - └─ fixedpoint\_js\_helper.R
  - └─ halley\_js\_helper.R
  - └─ newton\_js\_helper.R
  - └─ regula\_js\_helper.R
  - └─ secant\_js\_helper.R
- └─ pages
  - └─ bisectionpage.R
  - └─ fixedpointpage.R
  - └─ halleypage.R
  - └─ homepage.R
  - └─ newtonpage.R
  - └─ regulapage.R
  - └─ reipage.R
  - └─ secantpage.R
- └─ rsconnect
  - └─ shinyapps.io
    - └─ tcqd7b-yasar0can-kakdas
      - └─ root-finding.dcf
- └─ servers
  - └─ bisectionserver.R
  - └─ dashboardserver.R
  - └─ fixedpointserver.R

- | | ├── halleyserver.R
- | | ├── newtonserver.R
- | | ├── regulaserver.R
- | | └── secantserver.R
- | ├── util
- | | ├── server\_util.R
- | | └── ui\_util.R
- | ├── www
- | | ├── bisection.gif
- | | ├── fixedpoint.gif
- | | ├── halleys.gif
- | | ├── newton.gif
- | | ├── regula.gif
- | | ├── rei.gif
- | | ├── rei\_smiling.jpg
- | | └── secant.gif
- | ├── app.R
- | ├── dashboard.R
- | └── global.R

## app.R

“app.R” is the file where the execution of the project starts and triggers the chain of the modules to make the project work. Due to the modularization, “app.R” has the most straightforward shape it could get. The following image shows the “app.R”. It loads the “global.R” script and starts Shiny App by referencing the corresponding UI and server components.



```
1 source('global.R')
2
3 shinyApp(ui = dashboard, server = server)
```

## global.R

“global.R” is the file where all dependencies are defined so that no other scripts need to reference any script.

```

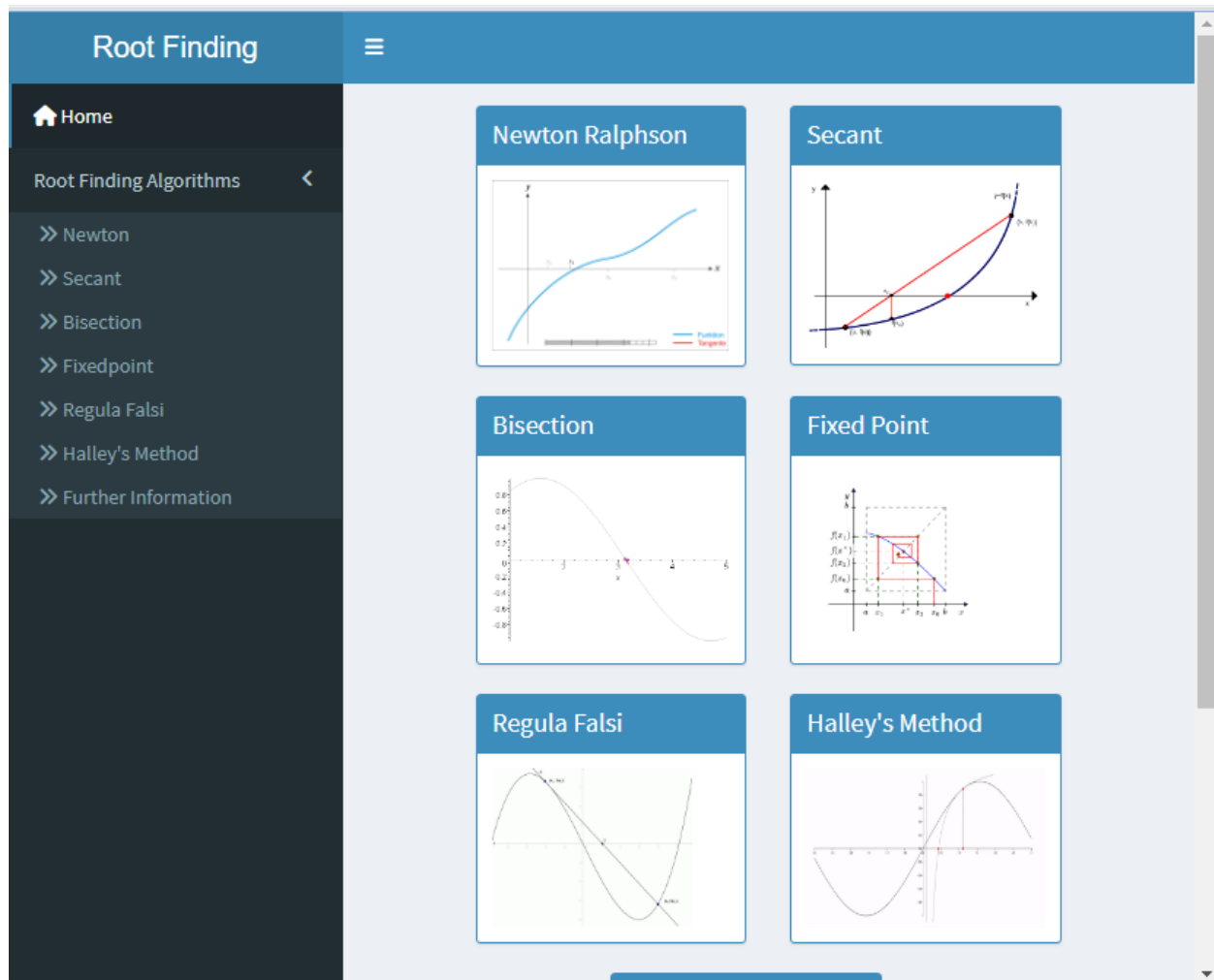
globalR X
app > globalR
1  list.of.packages <-
2    c(
3      "shiny",
4      "shinyjs",
5      "shinydashboard",
6      "systemfonts",
7      "shinyBS",
8      "shinyalert"
9    )
10
11  new.packages <-
12    list.of.packages[!(list.of.packages %in% installed.packages()[, "Package"])]
13  if (length(new.packages))
14    install.packages(new.packages)
15
16  library(shiny)
17  library(shinyjs)
18  library(shinydashboard)
19  library(spuRs)
20  library(shinyBS)
21  library(shinyalert)
22
23  source("util/server_util.R")
24  source("util/ui_util.R")
25
26  source("jsHelper/fixedpoint_js_helper.R")
27  source("jsHelper/bisection_js_helper.R")
28  source("jsHelper/regula_js_helper.R")
29  source("jsHelper/newton_js_helper.R")
30  source("jsHelper/secant_js_helper.R")
31  source("jsHelper/halley_js_helper.R")
32
33  server <- function(input, output, session) {
34    source('servers/fixedpointserver.R', local = TRUE)
35    source('servers/bisectionserver.R', local = TRUE)
36    source('servers/dashboardserver.R', local = TRUE)
37    source('servers/newtonserver.R', local = TRUE)
38    source('servers/secantserver.R', local = TRUE)
39    source('servers/regulaserver.R', local = TRUE)
40    source('servers/halleysserver.R', local = TRUE)
41  }
42
43
44  source("pages/homepage.R")
45  source("pages/fixedpointpage.R")
46  source("pages/bisectionpage.R")
47  source("pages/newtonpage.R")
48  source("pages/secantpage.R")
49  source("pages/regulapage.R")
50  source("pages/halleypage.R")
51  source("pages/reipage.R")
52
53  source('dashboard.R')
54
55  source("algorithms/bisection.R")
56  source("algorithms/fixedpoint.R")
57  source("algorithms/newton.R")
58  source("algorithms/secant.R")
59  source("algorithms/regula.R")
60  source("algorithms/halley.R")

```

“global.R” not only defines the libraries and dependencies but also checks whether a missing package needs to be installed. That eases the building and transferring phases of the project. Any person who clones the project for the first time just needs to click the “App Run” button and does not need to worry about the packages. Thanks to “global.R”, everything will be handled automatically.

## dashboard.R

The first screen that welcomes a user is called the dashboard. The dashboard page consists of two parts, 1) main panel where all root-finding algorithms are visualized by animated pictures (gif). 2) sidebar panel where a user can navigate through the root finding pages. It is also possible to navigate to a specific root-finding page by clicking the animated pictures in the main panel. A user may go back to the dashboard screen by pressing Home from the sidebar panel. The following figure shows the dashboard screen.



```

app > dashboard.R
1 sidebar <- dashboardSidebar(
2   tags$style(HTML(".sidebar-menu li a { font-size: 14px; }")),
3   sidebarMenu(
4     id = "tabs",
5     menuItem("Home", tabName = "home", icon = icon("home")),
6     menuItem(
7       "Root Finding Algorithms",
8       tabName = "rootfinding",
9       startExpanded = TRUE,
10      menuSubItem("Newton", tabName = "newton"),
11      menuSubItem("Secant", tabName = "secant"),
12      menuSubItem("Bisection", tabName = "bisection"),
13      menuSubItem("Fixedpoint", tabName = "fixedpoint"),
14      menuSubItem("Regula Falsi", tabName = "regula"),
15      menuSubItem("Halley's Method", tabName = "halley"),
16      menuSubItem("Further Information", tabName = "rei")
17    )
18  )
19 )
20
21
22 body <- dashboardBody(tabItems(
23   tabItem(tabName = "home",
24     home_page),
25
26   tabItem(tabName = "fixedpoint",
27     fixed_point_page),
28
29   tabItem(tabName = "newton",
30     newton_page),
31
32   tabItem(tabName = "bisection",
33     bisection_page),
34   tabItem(tabName = "secant",
35     secant_page),
36   tabItem(tabName = "regula",
37     regula_page),
38   tabItem(tabName = "halley", halley_page),
39   tabItem(tabName = "rei", rei_page)
40 )
41 )
42
43 dashboard <-
44   dashboardPage(dashboardHeader(title = "Root Finding Algorithms"),
45     sidebar,
46     body)
47
48

```

The figure above shows how the UI components of the dashboard are defined.

www

This folder contains all images that are used in the dashboard. This is the default path where shiny searches the reference images.

rsconnect

The “rsconnect” folder is automatically generated when the application is deployed to the “shinyapp.io”. The hosted application of the project may be reached by the following URL: <https://sci-comp.shinyapps.io/root-finding/>

## General Structure of Root-Finding Pages

Each different root-finding algorithm has its own page that consists of 3 main components.

1. Description field by HTML
2. Input fields (Input for function, input for initial value(s), slider for maximum iteration, slider for tolerance)
3. Output fields (Table that shows results, and plot if possible)
4. Calculate & Reset buttons

The following image demonstrates the HTML description component and input fields for root-finding pages.

**Info**

### Secant Root Finding

The Secant Method is an iterative algorithm for finding the root for the polynomials by successive approximation. For given two points, a new value is approximated by using secant line of the given points.

**Running Instructions**

Provide an input function and an initial guesses (start,end values) by following the rules given below and hit the **Calculate** button. The program will output a table with results and a plot if algorithm converges. You may enlarge and download the plot by clicking on it. You may click on the **Reset** button for clearing the input fields and outputs.

**Tips**

- Multiplication should be entered by using "asterisk(\*)" such as  $3 * x$  (Do not type  $3x$ )
- Any trigonometric function should be used such as:  $\cos(x)$ ,  $\sin(x)$ ,  $\tan(x)$ ,...
- Power of Euler Number(e) should be entered using "exp" such as  $\exp(x)$
- Any other exponentials can be typed using "^" such as  $x^{-5}$
- An **initial value**( $x_0$ ) should be given for the algorithm. Please only enter a numeric value.
- Example query: Function:  $\log(x) - \exp(-x)$  Initial Guesses: (1,2)

**Function**

Function Input(x)

Enter function...

**Initial Values**

First Initial Guess

Enter first...

Second Initial Guess

Enter second...

**Number of Maximum Iterations**

Select the max iteration value

100 1,000

100 190 280 370 460 550 640 730 820 910 1,000

**Tolerance**

Select the tolerance value

-12 -11 -10 -9 -8 -7 -6



## Generic UI & Server Components

Since all the pages almost share the same view, reusable components were implemented instead of copying-pasting features over different pages. Even though the UI looks identical, the algorithms must be called by other parameters. Algorithms were also designed to preserve the singleton structure with the same parameters. Each algorithm is of format: function (ftn, params) where ftn is the input function and params is the list of required parameters for the specific algorithm. Moreover, each algorithm returns the same predefined list of values to make the return types generic.

The following two images show “fixedpoint” and “halley” algorithms, respectively,

```
1 fixedpoint <- function (ftn, params)
2 {
3   with(as.list(c(params)), {
4     # Copy given arguments into local function definition
5     xold <- x0
6     xnew <- ftn(xold)
7     iter <- 1
8     while ((abs(xnew - xold) > tol) && (iter < max.iter)) {
9       xold <- xnew
10      xnew <- ftn(xold)
11      iter <- iter + 1
12    }
13    if (abs(xnew - xold) > tol) {
14      return(list(
15        status = "FAIL",
16        root = NA,
17        iteration = iter,
18        tolerance = tol
19      ))
20    }
21    else {
22      return(list(
23        status = "SUCCESS",
24        root = xnew,
25        iteration = iter,
26        tolerance = xnew - xold
27      ))
28    }
29  })
30
31 }
32
33
34
```

```

halley <- function(ftn, params) {
  with(as.list(c(params)), {
    x <- x0 # initialize
    fx <- ftn(x)
    iter <- 0
    # continue iterating until stopping conditions are met
    while ((abs(fx[1]) > tol) && (iter < max.iter)) {
      x <- x - 2 * fx[1] * fx[2] / (2 * fx[2] ^ 2 - fx[1] * fx[3])
      fx <- ftn(x)
      iter <- iter + 1
    }
    if (is.nan(fx[1]) || abs(fx[1]) > tol) {
      return(list(
        status = "FAIL",
        root = NA,
        iteration = iter,
        tolerance = tol
      ))
    } else {
      return(list(
        status = "SUCCESS",
        root = x,
        iteration = iter,
        tolerance = abs(fx[1])
      ))
    }
  })
}

```

As seen from the figures above, two very different algorithms have the same structure regarding the parameters and the return types.

How are these algorithms called by just using a single generic function?

For generic server functions, we have a script named “server\_util.R” for handling calls. “calculate\_root\_finding” function of this script has been defined as follows:

```

calculate_root_finding <-
function(input_func,
         root_finding_method,
         params,
         values,
         is_newton = F,
         is_halley = F) {
  tryCatch(f

```

This function takes the input function provided by the user and parses it. The only part that cannot be generalized is the needing to take derivatives for the algorithms “Newton” and “Halley”. The reference of the implementation of the root-finding algorithm is also passed as an argument to that function (root\_finding\_method) along with parameters (params). Lastly, the values hold the reactive values that must be carried between modules. Since all the algorithms have the exact function definition, the call “root\_finding\_method(ftn, params)” is enough for any root-finding algorithm. Therefore, we can run six completely different algorithms using a single function.

Although the pages look similar, each root-finding solution page still needs its own UI components with a unique identifier. We created a way to create UI elements by just providing a prefix of the algorithm. The following figure shows how all UI IDs are created by using a prefix,

```

populate_id <-
function(prefix,
  text_func_height,
  single_init,
  logify,
  onload,
  is_secant = F) {
  names <-
    list(
      popup = paste0(prefix, "_popup"),
      popup_plot = paste0(prefix, "_popup_plot"),
      download_plot = paste0(prefix, "_download_plot"),
      plot_name = paste0(toupper(substr(prefix, 1, 1)), substr(prefix, 2, nchar(prefix)), sep =
        "",
      logify = logify,
      onload = onload,
      html_id = paste0(prefix, "_html"),
      html_path = paste0("html/", prefix, ".html"),
      func_text_id = paste0(prefix, "_text_function"),
      func_text_height = text_func_height,
      single_init = single_init,
      init_value = paste0(prefix, "_init_value"),
      max_iter = paste0(prefix, "_max_iter_value"),
      tolerance = paste0(prefix, "_tolerance_value"),
      output_solution = paste0(prefix, "_solution"),
      output_plot = paste0(prefix, "_plot"),
      calculate = paste0(prefix, "_calculate_button"),
      reset = paste0(prefix, "_reset_button"),
      is_secant = is_secant
    )
  if (!single_init) {
    temp <-
      list(
        init_value_start = paste0(prefix, "_init_value_start"),
        init_value_end = paste0(prefix, "_init_value_end")
      )
    names <- append(names, temp)
  }
  if (is_secant) {
    temp <-
      list(
        first_title = "First Initial Guess",
        first_placeholder = "Enter first...",
        second_title = "Second Initial Guess",
        second_placeholder = "Enter second..."
      )
    names <- append(names, temp)
  }
  return(names)
}

```

Thanks to this generalization, our UI scripts are concise and readable. This function creates unique id names for each page. The following two figures show how this function is called to generate ids for the “fixedpoint” and “halley”.

```
fixed_point_page <-
  create_page(populate_id(
    "fixedpoint",
    173,
    T,
    fixedpoint_js_logify,
    fixedpoint_js_onload
  ))
```

```
halley_page <-
  create_page(populate_id(
    "halley",
    173,
    T,
    halley_js_logify,
    halley_js_onload
  ))
```

After creating IDs, UI components are made by using a similar logic. The following figure series shows how the UI is generated using these populated IDs.

```
create_page <- function(names) {
  fluidPage(
    useShinyjs(),
    create_popup_window(names$popup, names$popup_plot, names$download_plot, names$plot_name),
    tags$head(tags$script(HTML(names$logify))),
    tags$head(tags$script(HTML(names$onload))),
    fluidRow(column(
      12,
      create_box_for_html(names$html_id, names$html_path)
    )),
    fluidRow(
      column(6, create_box_for_function_text(names$func_text_id, names$func_text_height), align = "center"),
      if (names$single_init) {
        column(6,
          create_box_for_single_initial_value(names$init_value),
          align = "center"
        )
      } else {
        if (names$is_secant) {
          column(6, create_box_for_multiple_initial_values(names$init_value_start, names$init_value_end, names$first_title,
            names$first_placeholder, names$second_title, names$second_placeholder), align = "center")
        } else {
          column(6, create_box_for_multiple_initial_values(names$init_value_start, names$init_value_end), align = "center")
        }
      }
    ),
    fluidRow(
      column(6, create_box_for_max_iter(names$max_iter), align = "center"),
      column(6, create_box_for_tolerance(names$tolerance), align = "center")
    ),
    fluidRow(
      column(6, uioutput(names$output_solution), align = "center"),
      column(6, uioutput(names$output_plot), align = "center")
    ),
    fluidRow(
      column(6, create_action_button_for_calculation(names$calculate), align = "right"),
      column(6, create_action_button_for_reset(names$reset), align = "left")
    )
  )
}
```

```
create_box_for_function_text <- function(text_input_id, height) {
  box(
    title = "Function",
    width = 12,
    height = height,
    solidHeader = TRUE,
    status = "primary",
    textInput(text_input_id, h3("Function Input(x)"),
      placeholder = "Enter function..."),
  )
}
```

```

create_box_for_tolerance <- function(slider_id) {
  box(
    title = "Tolerance",
    width = 12,
    height = "50%",
    solidHeader = TRUE,
    status = "primary",
    sliderInput(
      slider_id,
      h4('Select the tolerance value'),
      min = -12,
      max = -6,
      value = -9
    )
  )
}

```

On the server side, which is the only part we cannot reduce to a single instance due to the need to reference components by using IDs, we still have minimal code by using the util scripts. The following figure shows how the root finding algorithm is called for “fixedpoint” and “bisection” through server scripts.

```

fixedpoint_solution <-
  eventReactive(
    input$fixedpoint_calculate_button,
    calculate_root_finding(
      input$fixedpoint_text_function,
      fixedpoint,
      c(
        x0 = as.numeric(input$fixedpoint_init_value),
        max.iter = input$fixedpoint_max_iter_value,
        tol = 10 ^ input$fixedpoint_tolerance_value
      ),
      fixedpoint_reactive_values
    )
  )

```

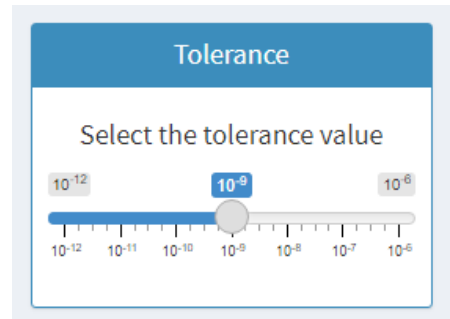
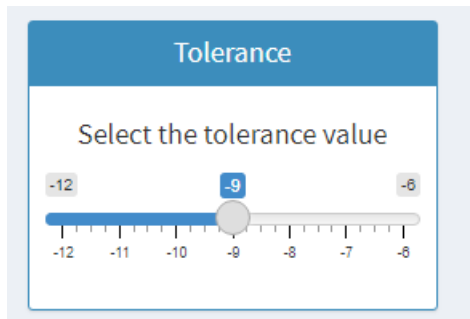
```

bisection_solution <-
  eventReactive(
    input$bisection_calculate_button,
    calculate_root_finding(
      input$bisection_text_function,
      bisection,
      c(
        a = as.numeric(input$bisection_init_value_start),
        b = as.numeric(input$bisection_init_value_end),
        max.iter = input$bisection_max_iter_value,
        tol = 10 ^ input$bisection_tolerance_value
      ),
      bisection_reactive_values
    )
  )

```

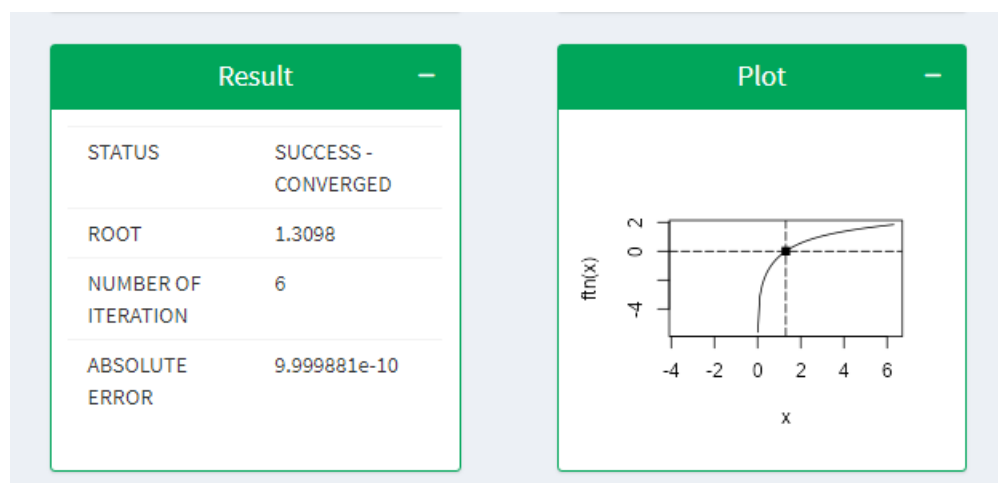
## Javascript Components

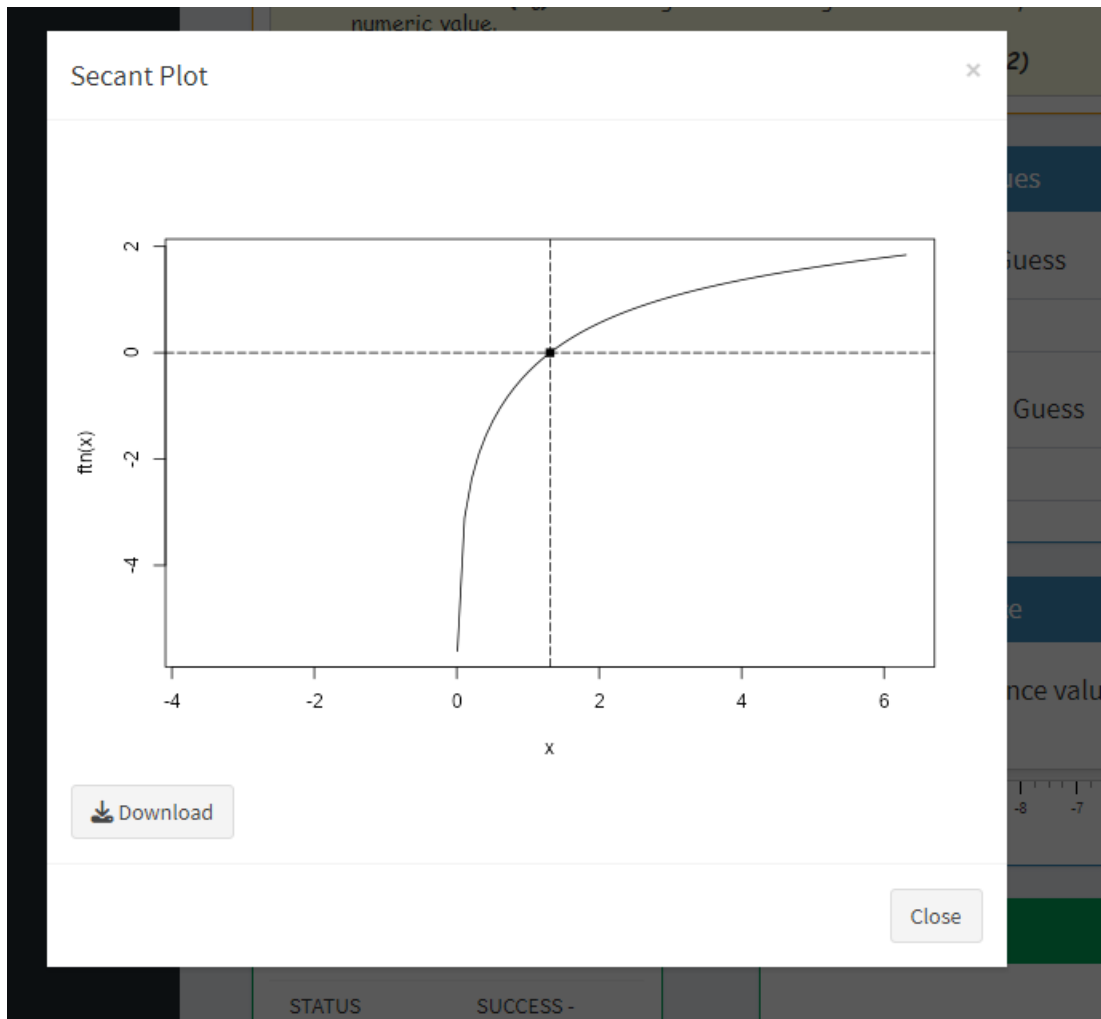
Shiny provides an easy way to build interactive apps. However, it has some limitations for specific needs. For instance, there is no support for changing the tick labels of slider components. At this point, we have used javascript functionalities to solve the issue. Changing tick labels are needed for the tolerance slider component. The following two figures show the same component without javascript and with javascript.



## Plots

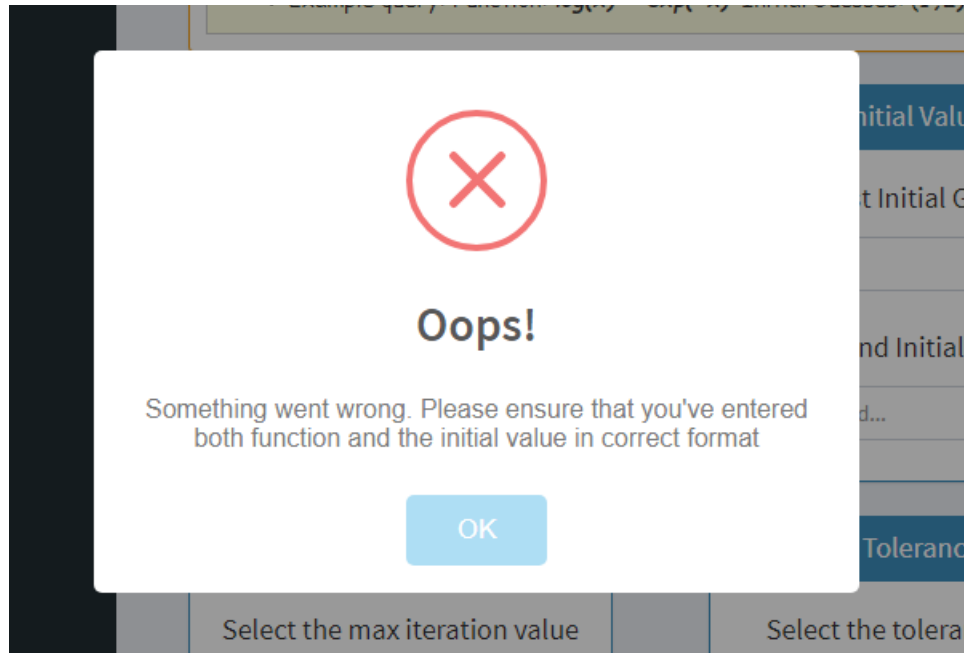
If the algorithm converges, we output a plot by marking the root. However, since the UI has limited space to place the plot, we added functionality where a user may enlarge the plot by clicking on it and may download it. The following figures show this feature:





## Error Handling

The core functions were encapsulated by a try-catch block to prevent unexpected crashes. If an error is being fired, we handle that, show an informative popup error window, and let the user continue using the app without interruptions. The following figure shows the UI when an error occurs.



## Conclusions/Future Work

We tried to make our application as flexible as possible and implemented a clear structure in our code. Our application allows us to make quick changes and analyze mistakes if they occur.

We created a shiny app that is easy to use and gives the user various algorithms. Furthermore, the user gets the opportunity to define their function. Compared to the option to choose out of a pool of functions, it adds enormous value to the application.

For future work, we could implement another page that automatically chooses the fastest or most precise algorithm for each function. That would ensure the best result for any function. Furthermore, a comparison page could be added that includes aspects such as time, number of evaluations, and accuracy of the result. Thus, the user could compare all algorithms and see what fits best for their function.



## Appendix

Code to ensure the installation of all packages and loading them:

```
1 list.of.packages <- c("shiny", "shinyjs", "shinydashboard", "spuRs")
2 new.packages <- list.of.packages[!(list.of.packages %in% installed.packages()[, "Package"])]
3 if(length(new.packages)) install.packages(new.packages)
4
5 library(shiny)
6 library(shinyjs)
7 library(shinydashboard)
8 library(spuRs)
```

### EventReactive – Implementation

```
1 solution_bisection <- eventReactive(input$calculate_button_bisection,
2   {
3     ftn <- function(x) {
4       exp <- parse(text = as.character(input$text_function_bisection))
5       return(eval(exp))
6     }
7     fun_result <-
8       fixedpoint(ftn, input$init_value_bisection)
9
10  })
```

The shinyjs package “onclick” in our dashboard server allows the user to run the r expression by clicking an element.

```
1 shinyjs::onclick("newton", updateTabItems(session, "tabs", "newton"))
2 shinyjs::onclick("bisection", updateTabItems(session, "tabs", "bisection"))
3 shinyjs::onclick("secant", updateTabItems(session, "tabs", "secant"))
4 shinyjs::onclick("fixedpoint", updateTabItems(session, "tabs", "fixedpoint"))
5 shinyjs::onclick("halley", updateTabItems(session, "tabs", "halley"))
6 shinyjs::onclick("rei", updateTabItems(session, "tabs", "rei"))
```

### Regula Falsi Method

$$c = b - \frac{f(b)(b - a)}{f(b) - f(a)} = \frac{af(b) - bf(a)}{f(b) - f(a)}$$

Halley’s Method for root-finding

$$x_{n+1} = x_n - \frac{2f(x_n)f'(x_n)}{2[f'(x_n)]^2 - f(x_n)f''(x_n)}$$

---

## Sources

<https://shiny.rstudio.com>

[https://en.wikipedia.org/wiki/Regula\\_falsi](https://en.wikipedia.org/wiki/Regula_falsi)

<http://www2.lv.psu.edu/ojj/courses/cmpsc-201/numerical/regula.html>

[https://en.wikipedia.org/wiki/Halley%27s\\_method](https://en.wikipedia.org/wiki/Halley%27s_method)

<https://mathworld.wolfram.com/HalleysMethod.html>