Homework 2

Advanced Statistical Computing (STAT 6984)

Sumin Shen

Problem 2: Immutable objects (30 pts)

Consider the following swap function which swaps elements `i` and `j` of a vector `v`. The time needed to swap the first two elements in v is:

```
swap <- function(v, i, j)
  {
    tmp <- v[i]
    v[i] <- v[j]
    v[j] <- tmp
      print(v[1:5])
  }
```

```
pool = v <- 1:1e9
system.time(swap(v, i=1, j=2))
```

```
## [1] 2 1 3 4 5
```

```
##    user  system elapsed
##   0.568   0.252   0.817
```

Check the first five elements of v in the parent frame after the operation:

```
## [1] 1 2 3 4 5
```

A disadvantage of this implementation is that it copies the entire vector, `v`, in order to work with just two of its elements. Consider the following example.

part a

a. Report on how much time it takes to swap two elements (`i=1; j=2`) dircetly on the command line, i.e., without wrapping in a function.

```
v = pool

for (r in 1:3){
  print(system.time({
i = 1
j = 2
tmp <- v[i]
v[i] <- v[j]
v[j] <- tmp
}))
}
```

```
##    user  system elapsed 
##   0.552   0.244   0.796 
##    user  system elapsed 
##       0       0       0 
##    user  system elapsed 
##       0       0       0
```

Check the first five elements of v in the parent frame after the operation:

```
## [1] 2 1 3 4 5
```

After several tries, the swap in the parent frame is very fast.

part b

b. Write a new version of the `swap` function, called `swap.eval`, which uses `quote` and `eval` to perform the calculation just like in part a. but within the function environment and without copying `v` by working on `v` in the `parent.frame`. Although this is a toy example, a similar code might be useful if, say, indicies `i` and `j` required substantial pre-calculation within the function before the swap occurred. Demonstrate your `swap.eval` with `i=1; j=2` and report on the time.

```r
v = pool
swap.eval = function(){
 tt =  quote({
    tmp <- v[i];
  v[i] <- v[j];
  v[j] <- tmp ;
    print(v[1:5])
  })
  eval(tt, envir = parent.frame()) # look for arguments in the parent.frame
}

i = 1; j = 1
for (r in 1:3) print(system.time(swap.eval()))
```

```
## [1] 1 2 3 4 5
##    user  system elapsed 
##   0.540   0.280   0.821 
## [1] 1 2 3 4 5
##    user  system elapsed 
##   0.004   0.000   0.002 
## [1] 1 2 3 4 5
##    user  system elapsed 
##       0       0       0
```

Check the first five elements of v in the parent frame after the operation:

```
## [1] 1 2 3 4 5
```

After several tries, the swap in the parent frame is very fast.

c. Write a similar function named `swap.do` which can be called via `do.call` that similarly accesses `v` without copying it. Add a `print` statement at the end of `swap.do` to show the first five elements of `v` after the swap occurs. Demonstrate `swap.do` with `i=1; j=2` and report on the time. Are there any disadvantages to `swap.do` compared to `swap.eval`?

```
v = pool
swap.do = function(){
  tmp <- v[i]
  v[i] <- v[j]
  v[j] <- tmp
  print(v[1:5])
}

i = 1
j = 2
for (r in 1:3) print(system.time(do.call(swap.do,args = list(), envir = parent.frame())))
```

```
## [1] 2 1 3 4 5
##    user  system elapsed
##   0.588   0.232   0.821
## [1] 2 1 3 4 5
##    user  system elapsed
##   0.552   0.264   0.816
## [1] 2 1 3 4 5
##    user  system elapsed
##   0.536   0.268   0.808
```

Check the first five elements of v in the parent frame after the operation:

```
## [1] 1 2 3 4 5
```

After several tries, the swap through do.call in the parent frame is slow. The argument is not copied in the function swap.do, so it was supposed to be fast, but it is not that fast compared to the quote and eval. Maybe due to the function do.call, which costs time.

Problem 3: Bisection broadening (30 pts)

a. Incorporate bracketing into the function we coded. Note that broadening is *not guaranteed* to find x_l and x_r such that $f(x_l)f(x_r) \leq 0$, so you should include a limit on the number of times broadening is successively tried with a sensible default.

The script bisection.R is updated and the updated encoded Bisection broadening algorithm is in the algorithm table

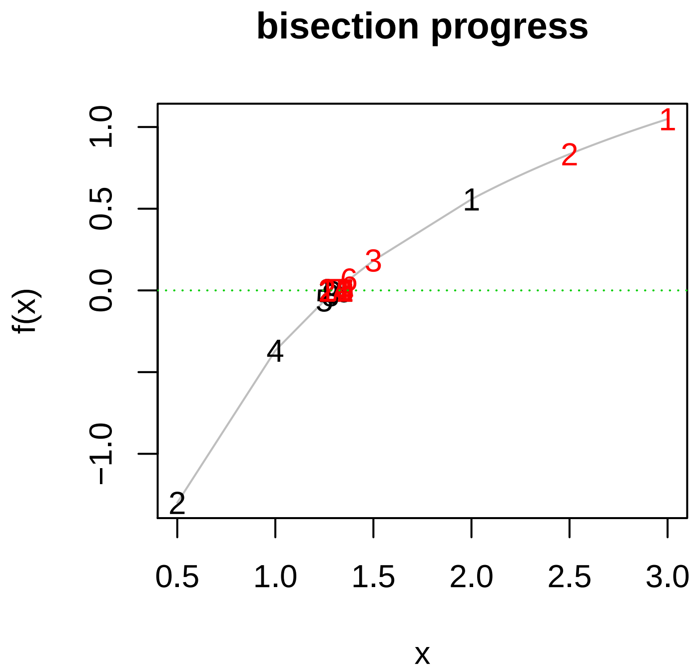
b. Use your modified function Bisection to find a root of the (original) function $f(x)$ we used in class, but with a different starting interval of $x_l = 2$ and $x_r = 3]$, i.e., not containing the root we found in class.

```
## Root of:
```

Algorithm 1

```
Input: f, xl, xr, tol, maxiter, verb
if xl > xr then
    stop
end if
fl = f(xl); fr = f(xr)
if fl = 0 then
    return xl
end if
if fr = 0 then
    return xr
end if
flag = fl * fr {comment: do the broadening}
if flag > 0 then
    for i in 1:maxiter do
        m = (xl + xr)/2
        w = xr - xl
        xl_tmp = m - w
        xr_tmp = m + w
        fl_tmp = f(xl_tmp)
        fr_tmp = f(xr_tmp)
        flag = fl_tmp * fr_tmp
        if flag <= 0 then {comment: do some updates}
            xl = xl_tmp
            xr = xr_tmp
            fl = fl_tmp
            fr = fr_tmp
            break
        else
            xl = xl_tmp
            xr = xr_tmp
        end if
    end for
end if
if flag = 0 then
    if fl = 0 then
        ans = xl
    else if fr = 0 then
        ans = xr
    end if
end if
if flag > 0 then
    print("max iter is reached before findind xl and xr s.t. fxl * fxr <= 0")
end if
if flag < 0 then
    do the bisection part in the provided script bisection.R
end if
output: f, tol, ans, initial
```

```
## function(x) log(x) - exp(-x)
## <bytecode: 0x283c538>
## in (2, 3) found after 29 iterations: 1.3098
## to a tolerance of 1.490116e-08
```




```
## Root of:
## function(x) log(x) - exp(-x)
## <bytecode: 0x283c538>
## in (2, 3) found after 29 iterations: 1.3098
## to a tolerance of 1.490116e-08
##
## Progress is as follows
##          xl       xr            fl           fr
## 1  2.000000 3.000000  5.578119e-01 1.048825e+00
## 2  0.500000 2.500000 -1.299678e+00 8.342057e-01
## 3  0.500000 1.500000 -1.299678e+00 1.823349e-01
## 4  1.000000 1.500000 -3.678794e-01 1.823349e-01
## 5  1.250000 1.500000 -6.336125e-02 1.823349e-01
## 6  1.250000 1.375000 -6.336125e-02 6.561414e-02
## 7  1.250000 1.312500 -6.336125e-02 2.787367e-03
## 8  1.281250 1.312500 -2.985381e-02 2.787367e-03
## 9  1.296875 1.312500 -1.342726e-02 2.787367e-03
## 10 1.304688 1.312500 -5.293741e-03 2.787367e-03
## 11 1.308594 1.312500 -1.246670e-03 2.787367e-03
## 12 1.308594 1.310547 -1.246670e-03 7.719731e-04
## 13 1.309570 1.310547 -2.369419e-04 7.719731e-04
## 14 1.309570 1.310059 -2.369419e-04 2.676172e-04
## 15 1.309570 1.309814 -2.369419e-04 1.536305e-05
## 16 1.309692 1.309814 -1.107831e-04 1.536305e-05
```

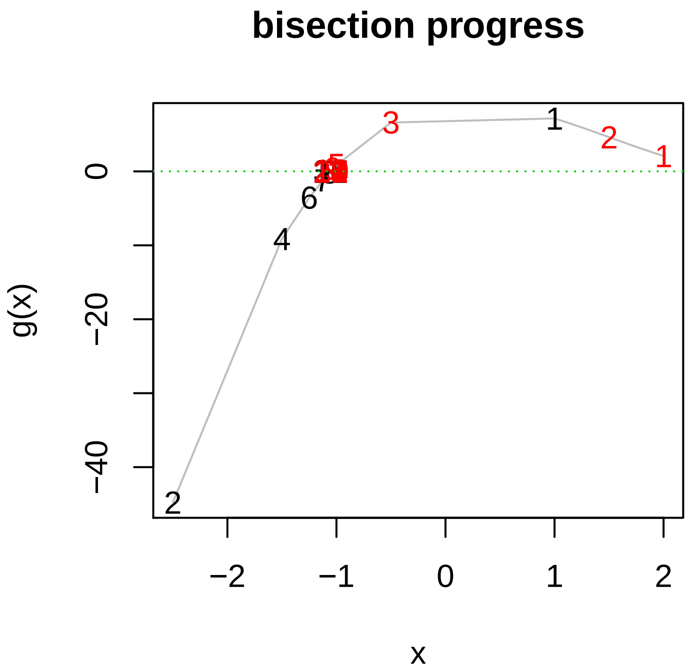
```
## 17 1.309753 1.309814 -4.770843e-05 1.536305e-05
## 18 1.309784 1.309814 -1.617229e-05 1.536305e-05
## 19 1.309799 1.309814 -4.045236e-07 1.536305e-05
## 20 1.309799 1.309807 -4.045236e-07 7.479287e-06
## 21 1.309799 1.309803 -4.045236e-07 3.537388e-06
## 22 1.309799 1.309801 -4.045236e-07 1.566434e-06
## 23 1.309799 1.309800 -4.045236e-07 5.809554e-07
## 24 1.309799 1.309800 -4.045236e-07 8.821597e-08
## 25 1.309799 1.309800 -1.581538e-07 8.821597e-08
## 26 1.309800 1.309800 -3.496891e-08 8.821597e-08
## 27 1.309800 1.309800 -3.496891e-08 2.662353e-08
## 28 1.309800 1.309800 -4.172689e-09 2.662353e-08
## 29 1.309800 1.309800 -4.172689e-09 1.122542e-08
```

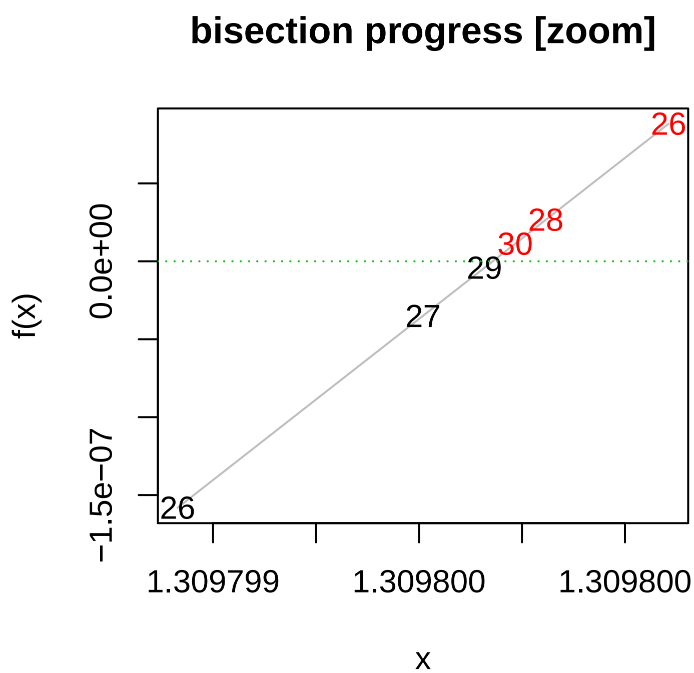
c. Use your modified function find the root of

$$h(x) = (x-1)^3 - 2x^2 + 10 - \sin(x),$$

starting with $x_l = 1$ and $x_r = 2$.

```
## Root of:
## function(x) (x-1)^3 - 2 * x^2 + 10 - sin(x)
## <bytecode: 0x12ad480>
## in (1, 2) found after 30 iterations: -1.052896
## to a tolerance of 1.490116e-08
```





For full credit you must keep everything in the S3 environment with appropriate modifications to your generic methods, etc. You will be judged on style here, in terms of code, `S3` behavior, and writeup/demonstration. You may have a separate `bisection.R` file with your `S3` library functions, however your writeup must verbally describe how those functions have changed. I will check for `bisection.R` in your repository against your description.

I tried to do the modifications on the generaic mothods: print.bisection, summary.bisection, plot.bisection, but when I used the provided functions, they are pretty good.

Problem 4: R scripts from the Unix prompt (25 pts)

R provides two commands to execute scripts:

- 1 R CMD BATCH
- 2 Rscript

For both commands, the plots are saved as Rplots.pdf in the directory.

For R CMD BATCH, the text output will be saved in a file filename.Rout, which includes the warnings and errors if any. The objects in the workspace is saved in the file .RData

```
R CMD BATCH test.R
```

If I want to make sure all the output should be saved, I could add the command `save.image()` in the R script.

For Rscript, the text output including the warning and error messages will go to the screen, no additional text output file is generated unless some function, maybe sink(), is specified in the R script. At default Rscript does not save objects in the workspace, but add the option '–save' will save the workspace at the end of the session.

```
Rscript --save test.R
```

A good reference is found in the cran r project

The single Unix command that is called from the command prompt to cause the .html and .R file to be generated from the Rmd file is

```
Rscript -e 'expr'
```

which is a front end for scripting with R. the option -e is followed by the expression to be evaluated.