MATH 3080 Lab Project 2

Jon Moses

01/15/2025

Contents

Problem 1

Write a function that takes a variable number of arguments and prints them with each of the inputs separated by a new line. (Hint: There is a special character, `\n`, that represents new lines. The following code should suggest what to do: `cat("hello", "awesome", "world", sep = "\n")`)

Name your function concat_func.

```
concat_func <- function(..., sep="\n") {
  cat(..., sep = "\n")
}
```

Don't change this next part; this is to see if your function worked!

```
writeLines("\nFirst test: ")
```

```
##
## First test:
```

```
concat_func("one", "two")
```

```
## one
## two
```

```
writeLines("\nSecond test: ")
```

```
##
## Second test:
```

```r
concat_func("one", "two", "three")
```

```
## one
## two
## three
```

Problem 2

Write an infix operator that represents logical XOR. In logic, ***x xor y*** *is true if only one of either* ***x*** *or* ***y*** *are true; if neither are true or both are true, then it's false. The following function implements XOR:*

```r
xor <- function(x, y) {
  (x | y) & (!x | !y)
}
```

Write the infix operator ***%xor%*** *that allows for the syntax* ***x %xor% y****.*

```r
# standard xor functionality, x or y, not both and not neither
`%xor%` = function(x, y) {(x | y) & (!x | !y)}
```

The following should work as anticipated:

```r
TRUE %xor% TRUE  # Should be FALSE
```

```
## [1] FALSE
```

```r
FALSE %xor% TRUE  # Should be TRUE
```

```
## [1] TRUE
```

```r
TRUE %xor% FALSE  # Should be TRUE
```

```
## [1] TRUE
```

```r
FALSE %xor% FALSE  # Should be FALSE
```

```
## [1] FALSE
```

Problem 3

Newton's method is a numerical root-finding technique; that is, given a function f, the objective of the method is to find an input x such that $f(x) = 0$. We call such an x a root*. The method is iterative. We start with an initial guess x_0. The algorithm then produces new approximations for the root x via the formula:*

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)}.$$

We need a rule for stopping the algorithm, and we could either stop at some fixed N or when $|x_{n+1} - x_n| < \epsilon$ for some user-selected $\epsilon > 0$. (This represents some tolerable numerical error.)

In this project you will write a function implementing Newton's method; call the function ***`newton_solver()`****. Based on the above description this function must take at least the following inputs:*

- *An initial x_0;*
- *A function f;*
- *The function's derivative f';*
- *A maximum number of iterations N; and*
- *A desired numerical tolerance ϵ.*

(One may think we need either ϵ or N but in practice we should always have N to ensure the algorithm terminates.)

We will add additional behavioral constraints to the function.

- *There will be a loop where the update algorithm is applied. This loop should terminate immediately if the numerical tolerance threshold is met; this can be achieved via an* ***`if`*** *statement and* ***`break`****. But if the loop hits N iterations, the function should throw a warning.*
- *f and f' should be functions. They should return univariate* ***`numeric`*** *values. If there ever comes a time where the input functions don't return a single number, then* ***`newton_solver()`*** *should throw an error.*
- *It's possible that $f'(x_n)$ could become zero and then a division-by-zero error will occur.* ***`newton_solver()`*** *should stop with an error informing the user that the derivative became zero.*
- *We could have our function return a list with detailed information not just with the obtained root but also with the value of f at the root or how many iterations of the algorithm went through. But instead, we will just have the function return the obtained root.*
- *The maximum number of iterations N should be a positive number; the same should be said for ϵ. If not, an error should be thrown.*

1. *Write* ***`newton_solver()`*** *based on the description above.*

```
#' Newton's Method for Finding Roots
#'
#' Implements Newton's method for finding roots of functions numerically
#'
#' This function implements Newton's method, a numerical root finding technique.
#' Given a function \eqn{f}, its derivative \eqn{f'}, and an initial guess for
#' the root \eqn{x_0}, the function finds the root via the iterative formula
#'
#' \deqn{x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)}}
#'
#' @param f The function for which a root is sought
#' @param fprime The function representing the derivative of \code{f}
#' @param x0 The initial guess of the root
#' @param N The maximum number of iterations
#' @param eps The tolerable numerical error \eqn{\epsilon}
#' @return The root of \code{f}
#' @examples
#' f <- function(x) {x^2}
#' fprime <- function(x) {2 * x}
#' newton_solver(f, fprime, x0 = 10, eps = 10^(-4))
newton_solver <- function(f, fprime, x0, N = 1000, eps = 10e-4) {
```

```
  counter = 0
  best_approximate = x0
  previous_approximate = 0
  while(counter < N) {
      if((abs(previous_approximate - best_approximate)) < eps) {
        best_approximate
        break
      }
      previous_approximate = best_approximate
      best_approximate <- best_approximate - (f(best_approximate)/fprime(best_approximate))
      #This print statement lets you see the evolution of the algorithms approximation
      print(paste("iteration", counter + 1, "best approximate: ", best_approximate))
      counter <- counter + 1
  }
  if(abs(previous_approximate - best_approximate) > eps) {
    warning("The error is very large")
  }
  print(best_approximate)


}
```

The following code tests whether `newton_solver()` *works as specified.* ***BEWARE: IF THIS CODE DOES NOT RUN AS ANTICIPATED OR TAKES LONGER THAN 10 SECONDS TO RUN, YOU WON'T RECEIVE CREDIT!***

```
# Test code for newton_solver(); DO NOT EDIT
f1 <- function(x) {x^2}
fprime1 <- function(x) {2 * x}
fprime2 <- function(x) {0}
fprime3 <- function(x) {1}
f2 <- function(x) {c(1, x^2)}
fprime4 <- function(x) {c(1, 2 * x)}
f3 <- function(x) {"oopsie!"}
fprime5 <- function(x) {"dasies!"}

# The following should execute without warning or error
newton_solver(f1, fprime1, x0 = 10, eps = 10e-4)   # Should be close to zero
```

```
## [1] "iteration 1 best approximate:  5"
## [1] "iteration 2 best approximate:  2.5"
## [1] "iteration 3 best approximate:  1.25"
## [1] "iteration 4 best approximate:  0.625"
## [1] "iteration 5 best approximate:  0.3125"
## [1] "iteration 6 best approximate:  0.15625"
## [1] "iteration 7 best approximate:  0.078125"
## [1] "iteration 8 best approximate:  0.0390625"
## [1] "iteration 9 best approximate:  0.01953125"
## [1] "iteration 10 best approximate:  0.009765625"
## [1] "iteration 11 best approximate:  0.0048828125"
## [1] "iteration 12 best approximate:  0.00244140625"
## [1] "iteration 13 best approximate:  0.001220703125"
## [1] "iteration 14 best approximate:  0.0006103515625"
```

```
## [1] 0.0006103516
```

```
newton_solver(f1, fprime1, x0 = -10, eps = 10e-4)  # Should be close to zero
```

```
## [1] "iteration 1 best approximate:  -5"
## [1] "iteration 2 best approximate:  -2.5"
## [1] "iteration 3 best approximate:  -1.25"
## [1] "iteration 4 best approximate:  -0.625"
## [1] "iteration 5 best approximate:  -0.3125"
## [1] "iteration 6 best approximate:  -0.15625"
## [1] "iteration 7 best approximate:  -0.078125"
## [1] "iteration 8 best approximate:  -0.0390625"
## [1] "iteration 9 best approximate:  -0.01953125"
## [1] "iteration 10 best approximate:  -0.009765625"
## [1] "iteration 11 best approximate:  -0.0048828125"
## [1] "iteration 12 best approximate:  -0.00244140625"
## [1] "iteration 13 best approximate:  -0.001220703125"
## [1] "iteration 14 best approximate:  -0.0006103515625"
## [1] -0.0006103516
```

```
# The following code should produce errors if the function was written correctly
newton_solver(f1, fprime2, x0 = 10)
```

```
## [1] "iteration 1 best approximate:  -Inf"
## [1] "iteration 2 best approximate:  -Inf"

## Error in `if ((abs(previous_approximate - best_approximate)) < eps) ...`:
## ! missing value where TRUE/FALSE needed
```

```
newton_solver(f2, fprime1, x0 = 10)
```

```
## [1] "iteration 1 best approximate:  9.95" "iteration 1 best approximate:  5"

## Error in `if ((abs(previous_approximate - best_approximate)) < eps) ...`:
## ! the condition has length > 1
```

```
newton_solver(f1, fprime4, x0 = 10)
```

```
## [1] "iteration 1 best approximate:  -90" "iteration 1 best approximate:  5"

## Error in `if ((abs(previous_approximate - best_approximate)) < eps) ...`:
## ! the condition has length > 1
```

```
newton_solver(f3, fprime1, x0 = 10)
```

```
## Error in `f(best_approximate) / fprime(best_approximate)`:
## ! non-numeric argument to binary operator
```

```r
newton_solver(f1, fprime5, x0 = 10)
```

```
## Error in `f(best_approximate) / fprime(best_approximate)`:
## ! non-numeric argument to binary operator
```

```r
# The following code should produce warnings if the function was written
# correctly
newton_solver(f1, fprime3, x0 = 10, N = 2)
```

```
## [1] "iteration 1 best approximate:  -90"
## [1] "iteration 2 best approximate:  -8190"

## Warning in newton_solver(f1, fprime3, x0 = 10, N = 2): The error is very large

## [1] -8190
```

2. *Use* ***newton_solver()*** *to maximize the function* $g(x) = 1 - x^2$*. Simple calculus should reveal that the maximum is* $g(0) = 1$*. Maximizing* g *requires finding a root for* g'*, since the maxima/minima of differentiable functions occurs where* $g'(x) = 0$*. Compare the answer obtained by* ***newton_solver()*** *to the known analytical result.*

```r
#
gprime1 <- function(x) {2 * -x}
gdoubleprime1 <- function(x) {-2}
newton_solver(gprime1, gdoubleprime1, x0 = 5.5, eps = 10e-4)
```

```
## [1] "iteration 1 best approximate:  0"
## [1] "iteration 2 best approximate:  0"
## [1] 0
```

Did you get the same results as the analytical solution?

Yes, the Newtons method approximation is the same as the analytical solution

Why do we know that using newtons method on g'(x) returns the maximum for g(x), as opposed to the minimum?

For this g(x) since it is a 2nd order polynomial we know that it will have only one inflection point and through simple observation we know that this g(x) does not have a minimum, so g'(x) = 0 must be the maximum.

3. *Use* ***newton_solver()*** *to solve the equation:*

$$e^x = -x$$

(This equation doesn't have a known analytical solution.)

```r
# Moving terms around to allow the solver to find a root and storing the function and the derivative
f2 <- function(x) {exp(x) + x}
fprime2 <- function(x) {exp(x) + 1}
# running the solver to a very small epsilon
newton_solver(f2, fprime2, x0 = 1, eps = .00000000000001)
```

```
## [1] "iteration 1 best approximate:  0"
## [1] "iteration 2 best approximate:  -0.5"
## [1] "iteration 3 best approximate:  -0.566311003197218"
## [1] "iteration 4 best approximate:  -0.567143165034862"
## [1] "iteration 5 best approximate:  -0.567143290409781"
## [1] "iteration 6 best approximate:  -0.567143290409784"
## [1] -0.5671433
```

The newton solver found: -0.567143 to be a reasonable approximation for the root of $f(x) = e^x + x$, which is in line with other estimates