STA 6133: Homework 1

Ricardo Cortez & Ben Graf

Due 9 Feb 2021

1.

Consider approximating the function $sin(x)$ by its Taylor polynomial of degree $2n-1$:

$$p_n(x) = x - \frac{x^3}{3!} + \frac{x^5}{5!} + \cdots + (-1)^{n+1}\frac{x^{2n-1}}{(2n-1)!}$$

(a)

Write an R function to compute $p_n(x)$ for $x \in \mathbb{R}$ and $n \in \mathbb{N}$.

```
sinapprox <- function(x, n) {
  ## sin(x) approximation using Taylor polynomial of degree 2n-1
  taylor <- 0
  for (i in 1:n) {
    taylor <- taylor + (-1)^(i+1) * x^(2*i-1) / factorial(2*i-1)
  }
  return(taylor)
}
```

(b)

Use your function to approximate $sin(x)$ for $x = \pi$ and 5π, and $n =$ 10, 20, and 100. What lesson(s) can you draw from this example?

```
sin(pi)
```

```
## [1] 1.224647e-16
```

```
sin(5*pi)
```

```
## [1] 6.123234e-16
```

```
sinapprox(pi,10)
```

```
## [1] -5.289183e-10
```

```
sinapprox(pi,20)
```

```
## [1] 3.328057e-16
```

```
sinapprox(pi,100)
```

```
## [1] 3.328057e-16
```

```
sinapprox(5*pi,10)
```

```
## [1] -169520.9
```

```
sinapprox(5*pi,20)
```

```
## [1] -0.2886023
```

```
sinapprox(5*pi,100)
```

```
## [1] -4.578756e-11
```

Because the Taylor polynomial contains numerous terms with negative signs, there is a risk of subtractive cancellation. The true values of $sin(\pi)$ and $sin(5\pi)$ should be 0. Even the built-in R functions do not provide exactly 0, though their results are on the order of 10^{-16}, so very small. We get similarly small numbers for our Taylor approximations of $sin(\pi)$, though the version with $n = 10$ is not quite as small as the others. The problems really kick in with approximating $sin(5\pi)$. Because the numbers being subtracted are larger, we see major errors in the approximations. For $n = 10$, the approximation is roughly $-169,000$, nowhere near 0! With $n = 20$, it improves but is still around -0.29. Finally, with $n = 100$, the approximation nears 0. The big lesson here is to beware of Taylor expansions with alternating positive and negative terms!

2.

On your computer, find a root fo the quadratic equation $ax^2 + bx + c = 0$ using the formula

$$x = \frac{-b + \sqrt{b^2 - 4ac}}{2a}$$

with the values $b = 1$ and $a = c = 10^{-n}$. Assess the accuracy of your answers for $n = 1, 5, 10$ by seeing how closely the equation is satisfied. Now think of a clever way to avoid subtractive cancellation and improve your answers.

Solution:

Finding a root in a non-clever way:

```
# create a quadratic function since we'll be testing several n's and potentially diff
erent b's
quad <- function(n,b){
  a <- 10^-n
  c <- a

  root_num_right <- sqrt(b^2-(4*a*c))
  #splitting the equations into separate terms to better understand the subtractive c
ancellation
  root_num <- -b +root_num_right
  root_den <- 2*a

  quad_root <- root_num/root_den
  true_root <- Re(polyroot(c(c,b,a)))[1] # only take the negative root since that's w
hat we're dealing with


  #use Re to only take the real root of the complex number, checked them all, no imag
parts.

  df <- data.frame("TrueRoot" =true_root, "QuadRoot" = quad_root)

  #calculate Relative Error
  rel_err <- abs((true_root-quad_root)/true_root)
  df$rel_err <- rel_err
  cat(paste0("Relative Error for a=",a, ", b=",b,", c=",c, ", and n=",n," is ", rel_e
rr))

  return(df)
}

df1 = quad(1,1)
```

```
## Relative Error for a=0.1, b=1, c=0.1, and n=1 is 2.06063905221897e-15
```

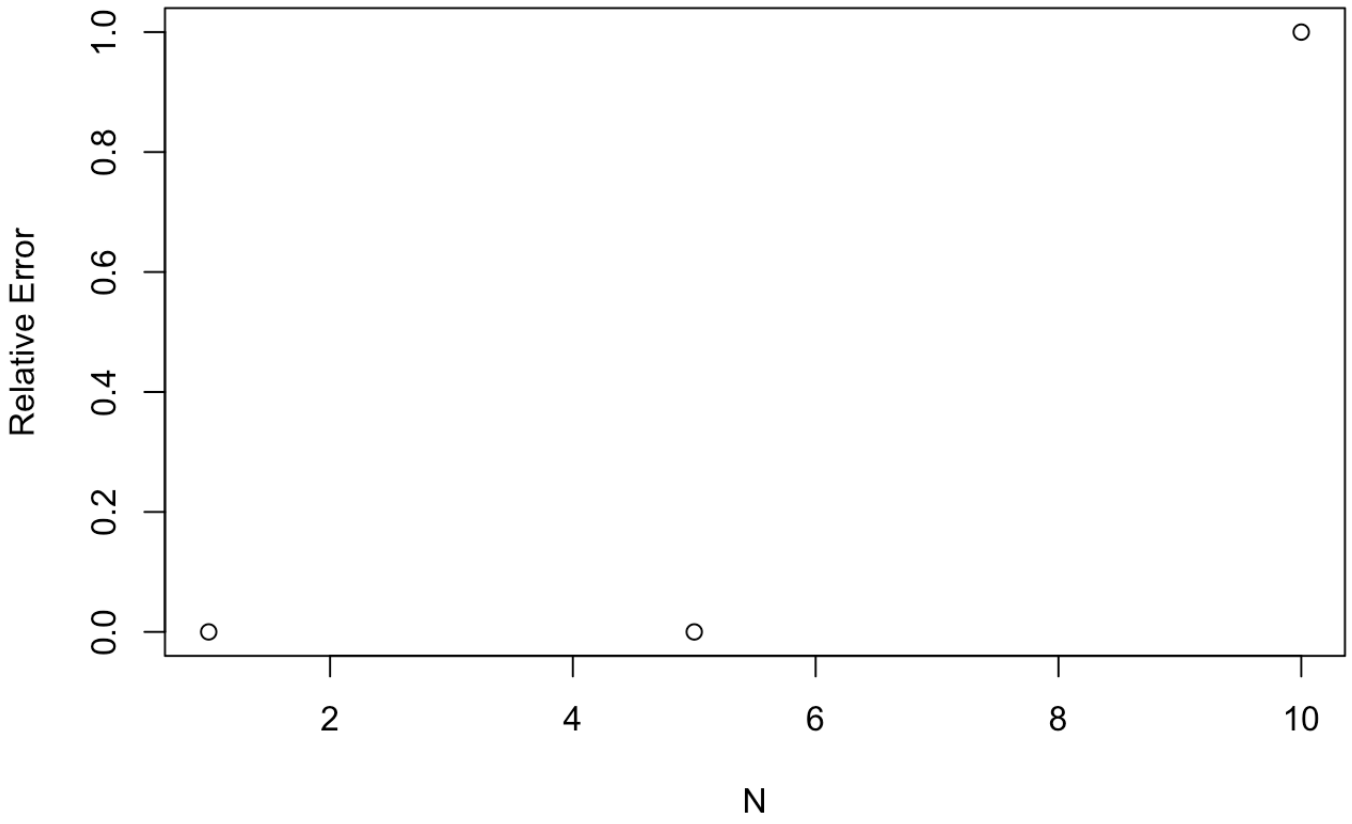
```
df2 = quad(5,1)
```

```
## Relative Error for a=1e-05, b=1, c=1e-05, and n=5 is 8.26403710463034e-08
```

```
df3 = quad(10,1)
```

```
## Relative Error for a=1e-10, b=1, c=1e-10, and n=10 is 1
```

```
plot(c(1,5,10),c(df1$rel_err,df2$rel_err,df3$rel_err), ylab = "Relative Error", xlab
= "N")
```



From the plot above, we can see as n increases the relative error increases to 1, which indicates that subtractive cancellation is present. A closer observation of the individual terms (not shown) indicated that the most aggressive cancellation occurred in between the square root term and the $-b$ term. A common way of handling this type of cancellation is to use known mathematical identities to eliminate the need for subtraction between those elements. In this problem we used:

$$x^2 - y^2 = (x - y)(x + y)$$

The derivation is as follows:

For simplicity, focus only on the numerator for now, the denominator can be added later.

$$\sqrt{b^2 - 4ac} - b$$

Let $z = \sqrt{b^2 - 4ac}$ and multiply by a carefully chosen version of 1:

$$\frac{(z - b)(z + b)}{z + b} = \frac{z^2 - b^2}{z + b}$$

Plugging original values back in:

$$\frac{b^2 - 4ac - b^2}{\sqrt{b^2 - 4ac} + b}$$

Simplifying:

$$\frac{-4ac}{\sqrt{b^2 - 4ac} + b}$$

Adding back the original denominator:

$$\frac{-4ac}{2a * (\sqrt{b^2 - 4ac} + b)}$$

Simplify again to obtain the final equation:

$$\frac{-2c}{(\sqrt{b^2 - 4ac} + b)}$$

Modifying the earlier code:

```
quad_new <- function(n,b){
  a <- 10^-n
  c <- a

  quad_root = -2*c / (sqrt(b^2 - 4*a*c) + b)
  true_root = Re(polyroot(c(c,b,a)))[1] # only take the negative root since that's wh
at we're dealing with


  #use Re to only take the real root of the complex number, checked them all, no imag
parts.

  df <- data.frame("TrueRoot" =true_root, "QuadRoot" = quad_root)

  #calculate Relative Error
  rel_err <- abs((true_root-quad_root)/true_root)
  df$rel_err <- rel_err
  cat(paste0("Relative Error for a=",a, ", b=",b,", c=",c, ", and n=",n," is ", rel_e
rr))

  return(df)
}

df1_new <- quad_new(1,1)
```

```
## Relative Error for a=0.1, b=1, c=0.1, and n=1 is 0
```

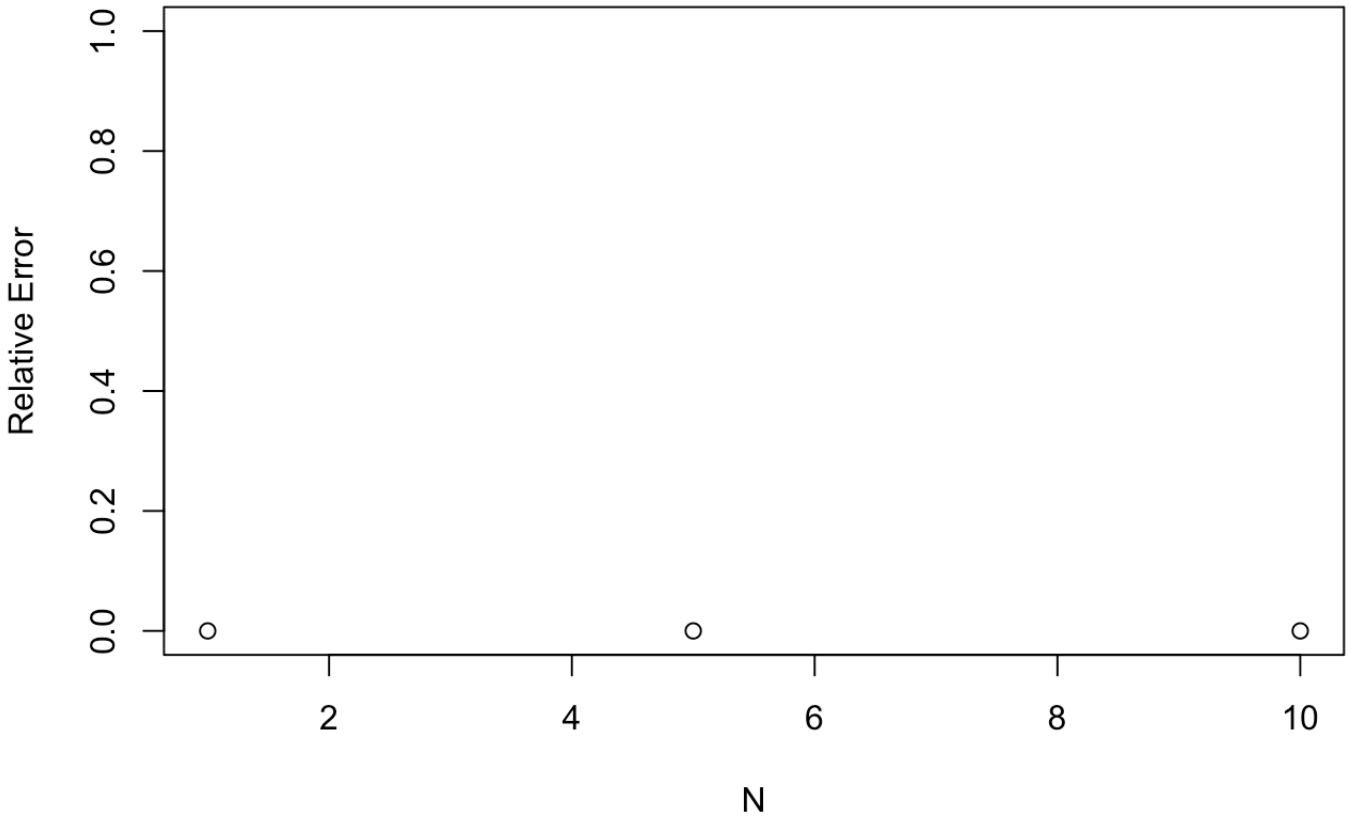
```
df2_new <-quad_new(5,1)
```

```
## Relative Error for a=1e-05, b=1, c=1e-05, and n=5 is 1.69406589433919e-16
```

```
df3_new <-quad_new(10,1)
```

```
## Relative Error for a=1e-10, b=1, c=1e-10, and n=10 is 1.29246970711411e-16
```

```
plot(c(1,5,10),c(df1_new$rel_err,df2_new$rel_err,df3_new$rel_err), ylab = "Relative E
rror", xlab = "N",ylim = c(0,1))
```



Using the new solution provides a better estimate of the roots when subtractive cancellation is present in the original equation, it is clear to see in the plot above that the relative error stays near 0 for all N's.

3.

The following data are an i.i.d. sample from the $Cauchy(\theta, 1)$ distribution ($\theta \in \mathbb{R}$):

1.77	−0.23	2.76	3.80	3.47	56.75
−1.34	4.24	−2.44	3.29	3.71	−2.40
4.53	−0.07	−1.05	−13.87	−2.53	−1.75
0.27	43.21				

(a)

Plot the log–likelihood function. Then find the MLE of θ using Newton's method. Try all the following starting values: –11, –1, 0, 1.5, 4, 4.7, 7, 38. Discuss your results. Is the sample mean a good starting point?

The density for $Cauchy(\theta, \sigma)$ is:

$$f(x \mid \theta, \sigma) = \frac{1}{\pi\sigma} \frac{1}{1 + \left(\frac{x-\theta}{\sigma}\right)^2}$$

Because $\sigma = 1$, we get:

$$f(x \mid \theta, \sigma) = \frac{1}{\pi} \frac{1}{1 + (x-\theta)^2} \quad , \quad -\infty < \theta < \infty \quad , \quad -\infty < x < \infty$$

The joint density is then:

$$f(\mathbf{x} \mid \theta, 1) = \prod_{i=1}^{n} \frac{1}{\pi} \frac{1}{1 + (x_i - \theta)^2} = \pi^{-n} \prod_{i=1}^{n} \frac{1}{1 + (x_i - \theta)^2}$$

The log-likelihood is then:

$$l(\theta) = -n \ln \pi - \sum_{i=1}^{n} \ln\left(1 + (x_i - \theta)^2\right)$$

Its first derivative is:

$$l'(\theta) = \sum_{i=1}^{n} \frac{2\,(x_i - \theta)}{1 + (x_i - \theta)^2}$$

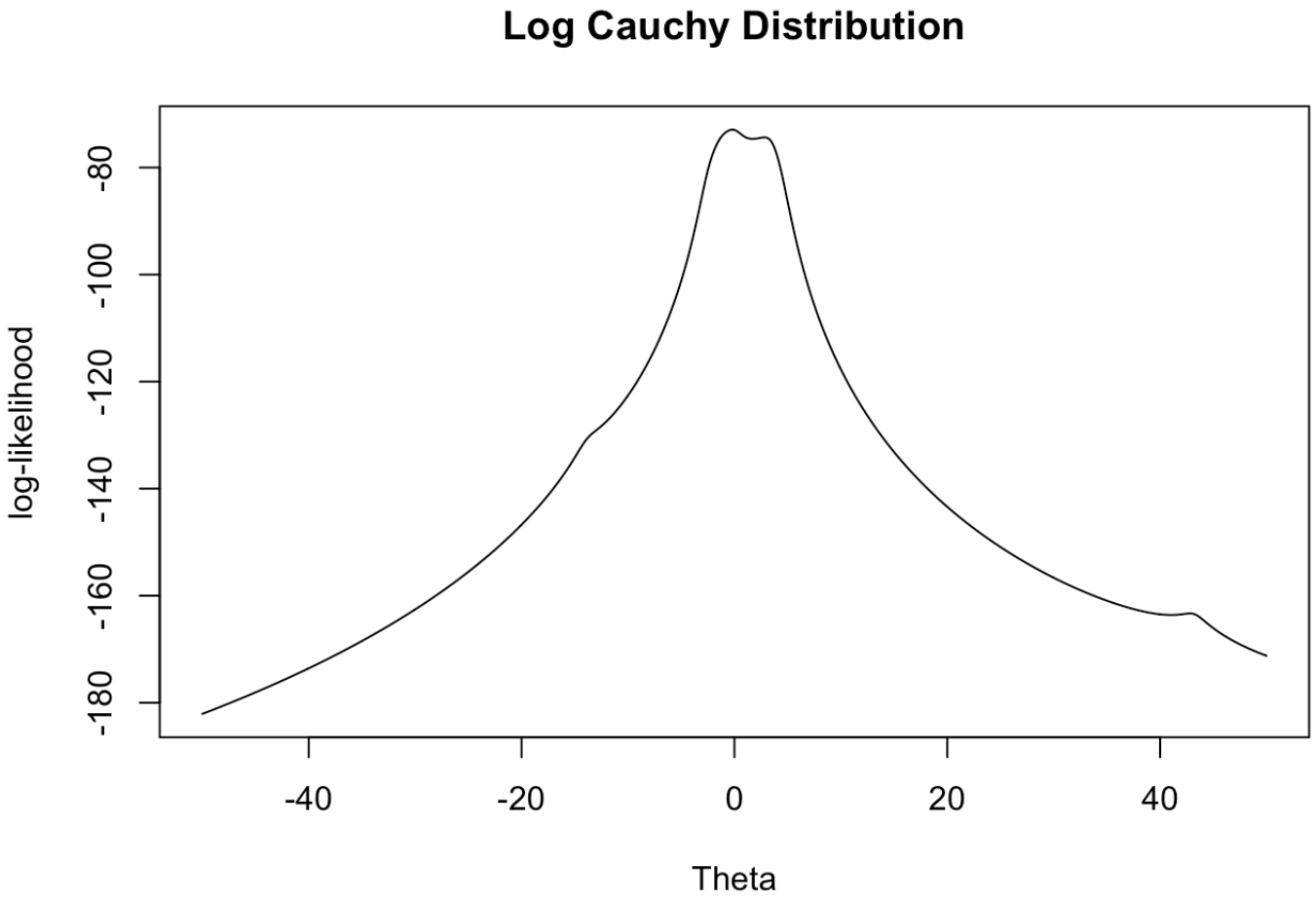
And its second derivative is:

$$l''(\theta) = \sum_{i=1}^{n} \frac{\left(1 + (x_i - \theta)^2\right)(-2) - 2\,(x_i - \theta)\,(-2\,(x_i - \theta))}{\left(1 + (x_i - \theta)^2\right)^2}$$

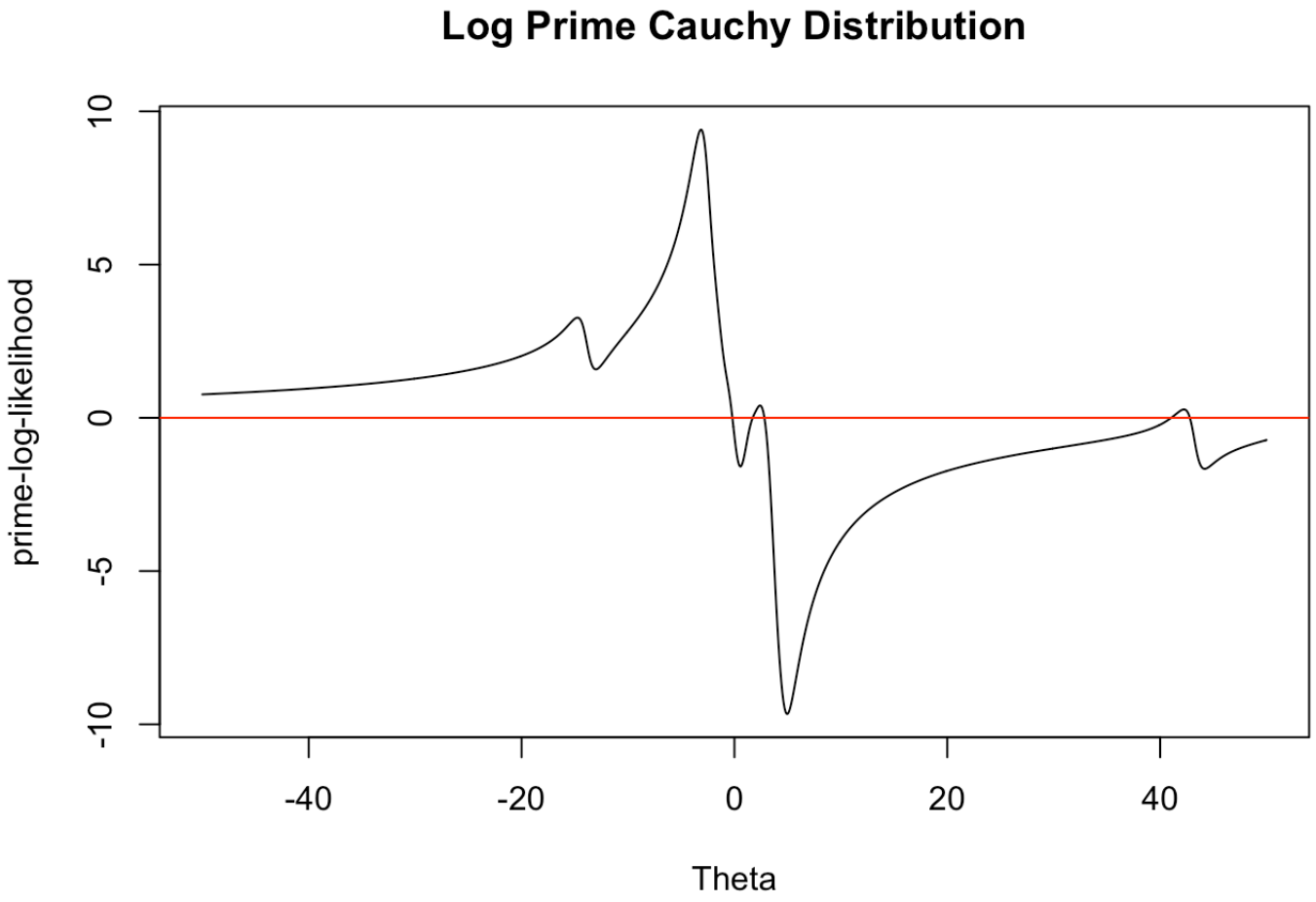
$$= \sum_{i=1}^{n} \frac{-2\left(1+(x_i-\theta)^2\right)+4(x_i-\theta)^2}{\left(1+(x_i-\theta)^2\right)^2} = \sum \frac{2\left(-1+(x_i-\theta)^2\right)}{\left(1+(x_i-\theta)^2\right)^2}$$

The corresponding R functions are:

```
#Import the data
data <- c(1.77, -0.23, 2.76, 3.80, 3.47, 56.75, -1.34, 4.24, -2.44, 3.29,
          3.71, -2.40, 4.53, -0.07, -1.05, -13.87, -2.53, -1.75, 0.27, 43.21)

n <- length(data)

#log likelihood cauchy function
log_cauchy <- function(theta){
  return(-n*log(pi)-sum(log(1+(data-theta)^2)))
}
#first derivative of the log-likelihood
log_cauchy_prime <- function(theta){
return(sum(2*(data-theta)/(1+(data-theta)^2)))
}

#second derivative of the log-likelihood
log_cauchy_2prime <- function(theta){
  return(sum(2*(-1+(data-theta)^2) / (1+(data-theta)^2)^2))
}

#function for log-cauchy and log-cauchy prime
cauchy <- function(theta){
  return(c(log_cauchy(theta),
           log_cauchy_prime(theta)))
}
#function for log-cauchy prime and 2nd prime of log-cauchy
cauchy_primes <- function(theta){
  return(c(log_cauchy_prime(theta),log_cauchy_2prime(theta)))
}

theta_list <- seq(-50,50,.1)
values <- sapply(theta_list, cauchy)

log_cauchy_values <- values[1,]
log_cauchy_prime_values <- values[2,]

plot(theta_list,log_cauchy_values, type = "l", main = "Log Cauchy Distribution",xlab
= "Theta", ylab = "log-likelihood")
```




```
plot(theta_list,log_cauchy_prime_values, type = "l", main = "Log Prime Cauchy Distrib
ution", xlab = "Theta", ylab = "prime-log-likelihood")
abline(h = 0, col = "red")
```


Log Prime Cauchy Distribution
prime-log-likelihood
10
5
0
-5
-10
-40
-20
0
20
40
Theta

```
# Newton's Method function provided by Dr. DeOliveira
newtonraphson <- function(ftn, x0, tol = 1e-9, max.iter = 100) {
  ## Newton_Raphson algorithm for solving ftn(x)[1] == 0
  ## It is assumed that ftn is a function of a single variable that returns
  ## the function value and its first derivative as a vector of length 2.
  ## x0 is the initial guess of the root.
  ## The algorithm terminates when the function value is within distance tol of 0,
  ## or the number of iterations exceeds max.iter, whichever happens first
  # initialise
  x <- x0
  fx <- ftn(x)
  iter <-  0
  # continue iterating until stopping conditions are met
  while ((abs(fx[1]) > tol) & (iter < max.iter)) {
    x <- x - fx[1]/fx[2]
    fx <- ftn(x)
    iter <- iter+1
    #cat("At iteration", iter, "value of x is:", x, "\n")
  }
  # output depends on success of algorithm
  if (abs(fx[1]) > tol) {
    #cat("Algorithm failed to converge\n")
    return(NULL)
  } else {
    #cat("Algorithm converged\n")
    return(c(root = x, f.val = fx[1]))
  }
}
```

We test using the recommended starting values:

```
mle_values <- roots <- test_values <- c(-11,-1,0,1.5,4,4.7,7,38, mean(data))

for(i in 0:length(test_values)){
  root <- newtonraphson(cauchy_primes,test_values[i])[1]
  roots[i] <- root
  mle_values[i] <- log_cauchy(root)
}
roots
```

```
## [1] -5.381711e+10 -1.922866e-01 -1.922866e-01  1.713587e+00  2.817472e+00
## [6] -1.922866e-01  4.104085e+01  4.279538e+01  5.487662e+01
```

```
 ### max value is at -1
cat(paste0("The maximum log-likelihood value of ", test_values[which.max(mle_values)]
, " occurs when theta = ", roots[which.max(mle_values)]))
```

```
## The maximum log-likelihood value of -1 occurs when theta = -0.192286613229646
```

$\hat{\theta}_{MLE} = -0.1923$.

Utilizing different starting points resulted in getting several different root values, which, given the execution pattern of the Newton-Raphson algorithm and the log-prime Cauchy distribution plot above, would be expected. Using the sample mean for a Cauchy distribution is not advantageous because the Cauchy distribution does not have a population mean. In fact, the sample mean as a starting point does *not* lead to the MLE.

(b)

Apply the bisection method with bracketing interval [–1, 1]. Comment on the result. Now run the bisection method with another bracketing interval that makes the algoritm converge to the 'wrong root', i.e., to a local maximum.

```
# Bisection Method function provided by Dr. DeOliveira
bisection <- function(f, x1, x2, maxit = 1000, tol = 1e-7, stop.fval = TRUE) {
  # it uses absolute error as stopping criterion when stop.fval = FALSE
  f1 <- f(x1)
  if (abs(f1) < tol)
    return(x1)
  f2 <- f(x2)
  if (abs(f2) < tol)
    return(x2)
  if (f1 * f2 > 0)
    stop("f has equal sign at endpoints of initial interval")
  if (f1 > 0) { # swap x1 and x2
    tmp <- x1 ; x1 <- x2 ; x2 <- tmp
  }
  n <- 0 # counter
  x <- x1
  repeat {
    n <- n + 1
    x.mid <- (x1 + x2) / 2
    f.mid <- f(x.mid)
    if(stop.fval == TRUE) {
      if(abs(f.mid) < tol | n == maxit)
        break }
    else if(abs(x.mid - x) < tol | n == maxit)
      break
    if(f.mid<0){ x1 <- x.mid ; x <- x1}
    else { x2 <- x.mid ; x <- x2 }
  }
  return(list(root = x.mid, f = f.mid, iter = n))
}

bisection(log_cauchy_prime, -1, 1, tol = 1e-9)
```

```
## $root
## [1] -0.1922866
##
## $f
## [1] 5.166203e-10
##
## $iter
## [1] 31
```

The Bisection took 31 iterations to converge with the given interval. The Newton-Raphson at worst took about the same number of iterations but often took fewer than 10 iterations. Below we see a poor choice of bracketing inveral results in converging to a local maximum.

```
bisection(log_cauchy_prime, 1, 2, tol = 1e-9)   #This bracket converges to a local ma
ximum, not the global
```

```
## $root
## [1] 1.713587
##
## $f
## [1] 3.384803e-10
##
## $iter
## [1] 29
```

(c)

Apply a fixed–point iteration with starting value −1 and scaling choices $\alpha = 1, 0.64, 0.25$. Discuss your results.

```
fixedpoint <- function(f, starting, alpha, maxit = 1000, tol = 1e-9) {
  # g(x) = x + alpha * f(x)
  x <- starting
  fx <- f(starting)
  iter <- 0
  while ((abs(fx) > tol) & (iter < maxit))  {
    x <- x + alpha*fx
    fx <- f(x)
    iter <- iter + 1
    #cat("At iteration", iter, "value of x is:", x, "\n")
  }
  if (abs(fx) > tol) {
    cat("Algorithm failed to converge\n")
    return(NULL)
  } else {
    cat("Algorithm converged\n")
    return(list(root = x, f.val = fx, iter = iter))
  }
}

fixedpoint(log_cauchy_prime, -1, 1)
```

```
## Algorithm failed to converge
```

```
## NULL
```

```
fixedpoint(log_cauchy_prime, -1, 0.64)
```

```
## Algorithm converged
```

```
## $root
## [1] -0.1922866
##
## $f.val
## [1] 9.973122e-10
##
## $iter
## [1] 700
```

```
fixedpoint(log_cauchy_prime, -1, 0.25)
```

```
## Algorithm converged
```

```
## $root
## [1] -0.1922866
##
## $f.val
## [1] 4.31729e-10
##
## $iter
## [1] 16
```

```
fixedpoint(log_cauchy_prime, -1, 0.2)
```

```
## Algorithm converged
```

```
## $root
## [1] -0.1922866
##
## $f.val
## [1] 5.361135e-10
##
## $iter
## [1] 24
```

For an α of 1, the fixed-point algorithm failed to converge. Reducing α to 0.64 does achieve convergence, but it takes 700 iterations. Further reducing α to 0.25 takes a mere 16 iterations! Reducing α any lower with the same starting point begins increasing the number of iterations again.

4.

There were 46 crude oil spills of at least 1000 barrels from tankers in U.S. waters during 1974–1999. The web page http://www.stat.colostate.edu/computationalstatistics/ (http://www.stat.colostate.edu/computationalstatistics/) contains the following data:
N_i = number of spills in year i;
b_{i1} = amount of oil shipped through U.S. waters from import/export shipments in year i (in billions of barrels [Bbbl]);
b_{i2} = amount of oil shipped through U.S. waters U.S. from domestic shipments in year i.
Suppose we use the Poisson process model that assumes $N_i|b_{i1}, b_{i2} \sim Poisson(\alpha_1 b_{i1} + \alpha_2 b_{i2})$. The volume of oil shipped is a measure of spill risk, so α_1 and α_2 represent the rates of spill occurrences per Bbbl during import/export and domestic shipments, respectively.

(a)

Derive Newton's iterative algorithm for finding the MLE of $\boldsymbol{\alpha} = (\alpha_1, \alpha_2)$, and write an R function to implement this algorithm. Then run this iterative algorithm to compute the MLE of $\boldsymbol{\alpha}$ based on the above dataset.

The joint density of $\mathbf{N}$ is:

$$f(\mathbf{N} \mid \boldsymbol{\alpha}) = \prod_{i=1}^{n} e^{-(\alpha_1 b_{i1} + \alpha_2 b_{i2})} \frac{(\alpha_1 b_{i1} + \alpha_2 b_{i2})^{N_i}}{N_i!}$$

The log-likelihood of $\boldsymbol{\alpha}$ is therefore:

$$l(\boldsymbol{\alpha}) = \sum_{i=1}^{n} \left(-(a_i b_{i1} + \alpha_2 b_{i2}) + N_i \ln(\alpha_1 b_{i1} + \alpha_2 b_{i2}) - \ln N_i! \right)$$

Its first derivatives are:

$$\frac{\partial l}{\partial \alpha_1} = \sum_{i=1}^{n} \left(-b_{i1} + N_i \frac{b_{i1}}{\alpha_1 b_{i1} + \alpha_2 b_{i2}} \right)$$

$$\frac{\partial l}{\partial \alpha_2} = \sum_{i=1}^{n} \left(-b_{i2} + N_i \frac{b_{i2}}{\alpha_1 b_{i1} + \alpha_2 b_{i2}} \right)$$

The gradient is defined as:

$$\nabla l(\boldsymbol{\alpha}) = \begin{pmatrix} \frac{\partial l}{\partial \alpha_1} \\ \frac{\partial l}{\partial \alpha_2} \end{pmatrix}$$

The second derivatives are:

$$\frac{\partial^2 l}{\partial \alpha_1^2} = \sum_{i=1}^{n} \frac{-N_i b_{i1}^2}{(\alpha_1 b_{i1} + \alpha_2 b_{i2})^2}$$

$$\frac{\partial^2 l}{\partial \alpha_1 \partial \alpha_2} = \frac{\partial^2 l}{\partial \alpha_2 \partial \alpha_1} = \sum_{i=1}^{n} \frac{-N_i b_{i1} b_{i2}}{(\alpha_1 b_{i1} + \alpha_2 b_{i2})^2}$$

$$\frac{\partial^2 l}{\partial \alpha_2^2} = \sum_{i=1}^{n} \frac{-N_i b_{i2}^2}{(\alpha_1 b_{i1} + \alpha_2 b_{i2})^2}$$

The Hessian matrix is defined as:

$$H_l(\boldsymbol{\alpha}) = \begin{bmatrix} \frac{\partial^2 l}{\partial \alpha_1^2} & \frac{\partial^2 l}{\partial \alpha_1 \partial \alpha_2} \\ \frac{\partial^2 l}{\partial \alpha_2 \alpha_1} & \frac{\partial^2 l}{\partial \alpha_2^2} \end{bmatrix}$$

Newton's iterative algorithm is therefore defined as:

$$\boldsymbol{\alpha}_{n+1} = \boldsymbol{\alpha}_n - \left(H_l\left(\boldsymbol{\alpha}_n\right)\right)^{-1} \nabla l\left(\boldsymbol{\alpha}_n\right), n = 0, 1, 2, \ldots$$

The R functions to implement this are as follows:

```
setwd("/Users/Ben/Library/Mobile Documents/com~apple~CloudDocs/Documents/UTSA Master'
s/Semester 4/STA 6133 Simulation & Statistical Computing/Homework/Homework 1/")
#readLines("oilspills.dat")
oil <- read.table("oilspills.dat", header = TRUE)

# Log-likelihood function for Poisson
llpois <- function(alphas, bs, N) {
  ## alphas should be 1x2, bs should be 2xn, N should be 1xn
  term <- alphas %*% bs
  return(sum(-term + N*log(term) - log(factorial(N))))
}

# Gradient of log-likelihood for Poisson
gradpois <- function(alphas, bs, N) {
  ## alphas should be 1x2, bs should be 2xn, N should be 1xn
  term <- alphas %*% bs
  drv1 <- sum(-bs[1,] + N*bs[1,]/term)
  drv2 <- sum(-bs[2,] + N*bs[2,]/term)
  return(as.matrix(c(drv1,drv2)))
}

# Hessian of log-likelihood for Poisson
hesspois <- function(alphas, bs, N) {
  ## alphas should be 1x2, bs should be 2xn, N should be 1xn
  term <- alphas %*% bs
  drv11 <- sum(-N * (bs[1,])^2 / (term^2))
  drv12 <- sum(-N * bs[1,] * bs[2,] / (term^2))
```

```
  drv22 <- sum(-N * (bs[2,])^2 / (term^2))
  return(matrix(c(drv11,drv12, drv12,drv22), nrow = 2, ncol = 2, byrow = TRUE))
}

# Newton's Method adapted for this problem
newtonmulti <- function(grad, hess, x0, data1, data2, tol = 1e-9, max.iter = 100) {
  ## Newton_Raphson algorithm for solving grad == 0.
  ## x0 is the initial guess of the roots.
  ## The algorithm terminates when the function value is within distance tol of 0,
  ## or the number of iterations exceeds max.iter, whichever happens first.

  # initialise
  x <- x0
  fx <- grad(x, data1, data2)
  iter <-  0
  # continue iterating until stopping conditions are met
  while (((abs(fx[1]) > tol) | (abs(fx[2]) > tol)) & (iter < max.iter)) {
    x <- x - t(solve(hess(x,data1,data2)) %*% grad(x,data1,data2))
    fx <- grad(x, data1, data2)
    iter <- iter+1
    cat("At iteration", iter, "value of x is:", x, "and f is:", fx, "\n")
  }

  # output depends on success of algorithm
  if ((abs(fx[1]) > tol) | (abs(fx[2]) > tol)) {
    cat("Algorithm failed to converge\n")
    return(NULL)
  } else {
    cat("Algorithm converged\n")
    return(c(root = x, f.val = fx))
  }
}
```

Testing the function with a variety of starting points yields:

```
# Run Newton's Method
testalphas <- t(as.matrix(c(1,1)))
bs <- t(as.matrix(oil[,3:4]))
N <- oil[,2]
res <- newtonmulti(gradpois, hesspois, testalphas, bs, N)
```

```
## At iteration 1 value of x is: 1.090831 0.9426757 and f is: 0.05893873 0.01547945
## At iteration 2 value of x is: 1.097128 0.9375748 and f is: 0.0002198173 5.441683e-
05
## At iteration 3 value of x is: 1.097153 0.9375546 and f is: 3.178962e-09 7.871014e-
10
## At iteration 4 value of x is: 1.097153 0.9375546 and f is: -1.110223e-16 4.440892e
-16
## Algorithm converged
```

```
llpois(res[1:2], bs, N)
```

```
## [1] -48.02716
```

```
newtonmulti(gradpois, hesspois, t(as.matrix(c(0.5,0.5))), bs, N)
```

```
## At iteration 1 value of x is: 0.7727078 0.7356689 and f is: 9.934929 6.243195
## At iteration 2 value of x is: 1.001826 0.8965661 and f is: 2.117605 1.293061
## At iteration 3 value of x is: 1.08905 0.9364765 and f is: 0.1456226 0.08299321
## At iteration 4 value of x is: 1.097096 0.9375682 and f is: 0.0008103152 0.00039828
23
## At iteration 5 value of x is: 1.097153 0.9375546 and f is: 2.760745e-08 1.037014e-
08
## At iteration 6 value of x is: 1.097153 0.9375546 and f is: -1.110223e-16 4.440892e
-16
## Algorithm converged
```

```
##         root1         root2        f.val1        f.val2
##  1.097153e+00  9.375546e-01 -1.110223e-16  4.440892e-16
```

```
newtonmulti(gradpois, hesspois, t(as.matrix(c(0.1,0.5))), bs, N)
```

```
## At iteration 1 value of x is: 0.2055593 0.8153307 and f is: 45.0863 20.6025
## At iteration 2 value of x is: 0.4085579 1.114891 and f is: 17.60121 7.294911
## At iteration 3 value of x is: 0.7192223 1.176014 and f is: 5.309933 1.781389
## At iteration 4 value of x is: 0.9984792 1.019079 and f is: 0.9585817 0.24154
## At iteration 5 value of x is: 1.091235 0.9425422 and f is: 0.05339925 0.012897
## At iteration 6 value of x is: 1.097132 0.9375719 and f is: 0.0001887155 4.666506e-
05
## At iteration 7 value of x is: 1.097153 0.9375546 and f is: 2.344399e-09 5.804666e-
10
## At iteration 8 value of x is: 1.097153 0.9375546 and f is: 9.436896e-16 8.881784e-
16
## Algorithm converged
```

```
##        root1        root2       f.val1       f.val2
## 1.097153e+00 9.375546e-01 9.436896e-16 8.881784e-16
```

```
newtonmulti(gradpois, hesspois, t(as.matrix(c(0.8,0.5))), bs, N)
```

```
## At iteration 1 value of x is: 1.020556 0.7477629 and f is: 3.400149 2.595636
## At iteration 2 value of x is: 1.093548 0.9057989 and f is: 0.3543095 0.2991098
## At iteration 3 value of x is: 1.097232 0.9368358 and f is: 0.005220441 0.005188247
## At iteration 4 value of x is: 1.097153 0.9375543 and f is: 1.525131e-06 1.74987e-0
6
## At iteration 5 value of x is: 1.097153 0.9375546 and f is: 1.797451e-13 2.146616e-
13
## Algorithm converged
```

```
##        root1        root2       f.val1       f.val2
## 1.097153e+00 9.375546e-01 1.797451e-13 2.146616e-13
```

```
newtonmulti(gradpois, hesspois, t(as.matrix(c(2,0.5))), bs, N)
```

```
## At iteration 1 value of x is: 0.5397918 1.334004 and f is: 8.537706 2.489554
## At iteration 2 value of x is: 0.8855011 1.121394 and f is: 2.126835 0.4806032
## At iteration 3 value of x is: 1.070158 0.9606329 and f is: 0.243707 0.05690754
## At iteration 4 value of x is: 1.096722 0.9379148 and f is: 0.003902619 0.000960497
6
## At iteration 5 value of x is: 1.097152 0.9375547 and f is: 1.004578e-06 2.487083e-
07
## At iteration 6 value of x is: 1.097153 0.9375546 and f is: 6.544765e-14 1.626477e-
14
## Algorithm converged
```

```
##        root1        root2       f.val1       f.val2
## 1.097153e+00 9.375546e-01 6.544765e-14 1.626477e-14
```

```
newtonmulti(gradpois, hesspois, t(as.matrix(c(2,2))), bs, N)
```

```
## At iteration 1 value of x is: 0.3633246 -0.2292973 and f is: 19.33009 -145.1933
## At iteration 2 value of x is: 0.6813456 -0.430406 and f is: 3.548708 -76.50709
## At iteration 3 value of x is: 1.20365 -0.7617812 and f is: -3.752098 -41.64224
## At iteration 4 value of x is: 1.912676 -1.215075 and f is: -6.239723 -23.16762
## At iteration 5 value of x is: 2.584328 -1.653179 and f is: -5.39579 -12.03947
## At iteration 6 value of x is: 2.924771 -1.888807 and f is: -2.417535 -4.272862
## At iteration 7 value of x is: 2.998046 -1.947498 and f is: -0.3045471 -0.5066411
## At iteration 8 value of x is: 3.004649 -1.953398 and f is: -0.003872094 -0.0064567
01
## At iteration 9 value of x is: 3.004737 -1.953477 and f is: -6.418283e-07 -1.071488
e-06
## At iteration 10 value of x is: 3.004737 -1.953477 and f is: 4.463097e-14 6.028511e
-14
## Algorithm converged
```

```
##         root1         root2        f.val1        f.val2
##  3.004737e+00 -1.953477e+00  4.463097e-14  6.028511e-14
```

All but one of these starting points converge to the same root, $\hat{\boldsymbol{\alpha}}_{\boldsymbol{MLE}} = \mathbf{(1.0972, 0.9376)}$. (The final starting point sees α_2 converge to a negative number, which seems inappropriate given the Poisson distribution.) Plotting the log-likelihood for a range of αs (not shown) to check that we are converging to a maximum reveals that the maximum does appear to be where both αs are between 0 and 2, with the highest-valued sample point being $(1.1, 0.9)$, so our root appears to be correct!

(b)

Derive the Fisher scoring iterative algorithm for finding the MLE of $\boldsymbol{\alpha}$, and write an R function to implement this algorithm. Then run this iterative algorithm to compute the MLE of $\boldsymbol{\alpha}$ based on the above dataset.

The terms of the Fisher Information matrix are:

$$(I(\boldsymbol{\alpha}))_{ij} = -E_{\boldsymbol{\alpha}}\left[\frac{\partial^2}{\partial\alpha_i\partial\alpha_j} l(\boldsymbol{\alpha}, \mathbf{N})\right]$$

The first term can therefore be calculated to be:

$$(I(\boldsymbol{\alpha}))_{11} = E\left[\frac{\partial^2 l}{\partial\alpha_1^2}\right] = E\left[\sum_{i=1}^{n} \frac{N_i b_{i1}^2}{(\alpha_1 b_{i1} + \alpha_2 b_{i2})^2}\right] = \sum_{i=1}^{n} \frac{{b_{i1}}^2 E\left[N_i\right]}{(\alpha_1 b_{i1} + \alpha_2 b_{i2})^2}$$

Because $E[N_i]$ is the Poisson parameter, we get:

$$= \sum_{i=1}^{n} \frac{b_{i1}^2\,(\alpha_1 b_{i1} + \alpha_2 b_{i2})}{(\alpha_1 b_{i1} + \alpha_2 b_{i2})^2} = \sum_{i=1}^{n} \frac{b_{i1}^2}{\alpha_1 b_{i1} + \alpha_2 b_{i2}}$$

Similarly:

$$(I(\boldsymbol{\alpha}))_{12} = (I(\boldsymbol{\alpha}))_{21} = \sum_{i=1}^{n} \frac{b_{i1} b_{i2}}{\alpha_1 b_{i1} + \alpha_2 b_{i2}}$$

$$(I(\boldsymbol{\alpha}))_{22} = \sum_{i=1}^{n} \frac{b_{i2}^2}{\alpha_1 b_{i1} + \alpha_2 b_{i2}}$$

Running Newton's algorithm for the same set of starting points yields:

```
# Fisher Information of log-likelihood for Poisson
ipois <- function(alphas, bs, N) {
  ## alphas should be 1x2, bs should be 2xn, N should be 1xn
  term <- alphas %*% bs
  drv11 <- -sum((bs[1,])^2 / term)
  drv12 <- -sum(bs[1,] * bs[2,] / term)
  drv22 <- -sum((bs[2,])^2 / term)
  return(matrix(c(drv11,drv12, drv12,drv22), nrow = 2, ncol = 2, byrow = TRUE))
}

# Run Newton's Method
res2 <- newtonmulti(gradpois, ipois, testalphas, bs, N)
```

```
## At iteration 1 value of x is: 1.117785 0.9062845 and f is: -0.05712725 0.07045913
## At iteration 2 value of x is: 1.090702 0.9473314 and f is: 0.01871967 -0.02155272
## At iteration 3 value of x is: 1.099195 0.9344591 and f is: -0.005841277 0.00687103
7
## At iteration 4 value of x is: 1.096508 0.9385308 and f is: 0.001850667 -0.00216218
## At iteration 5 value of x is: 1.097356 0.9372463 and f is: -0.0005835163 0.0006831
982
## At iteration 6 value of x is: 1.097088 0.9376519 and f is: 0.000184263 -0.00021559
48
## At iteration 7 value of x is: 1.097173 0.9375239 and f is: -5.815872e-05 6.806245e
-05
## At iteration 8 value of x is: 1.097146 0.9375643 and f is: 1.835935e-05 -2.148428e
-05
## At iteration 9 value of x is: 1.097155 0.9375515 and f is: -5.795342e-06 6.781905e
-06
## At iteration 10 value of x is: 1.097152 0.9375556 and f is: 1.829394e-06 -2.140805
e-06
## At iteration 11 value of x is: 1.097153 0.9375543 and f is: -5.774754e-07 6.757782
e-07
## At iteration 12 value of x is: 1.097152 0.9375547 and f is: 1.822889e-07 -2.133196
e-07
## At iteration 13 value of x is: 1.097153 0.9375546 and f is: -5.754227e-08 6.733757
e-08
## At iteration 14 value of x is: 1.097153 0.9375546 and f is: 1.816409e-08 -2.125613
e-08
## At iteration 15 value of x is: 1.097153 0.9375546 and f is: -5.733772e-09 6.709819
e-09
## At iteration 16 value of x is: 1.097153 0.9375546 and f is: 1.809952e-09 -2.118057
e-09
## At iteration 17 value of x is: 1.097153 0.9375546 and f is: -5.713394e-10 6.685968
e-10
## Algorithm converged
```

```
llpois(res2[1:2], bs, N)
```

```
## [1] -48.02716
```

```
newtonmulti(gradpois, ipois, t(as.matrix(c(0.5,0.5))), bs, N)
```

```
## At iteration 1 value of x is: 1.117785 0.9062845 and f is: -0.05712725 0.07045913
## At iteration 2 value of x is: 1.090702 0.9473314 and f is: 0.01871967 -0.02155272
## At iteration 3 value of x is: 1.099195 0.9344591 and f is: -0.005841277 0.00687103
7
## At iteration 4 value of x is: 1.096508 0.9385308 and f is: 0.001850667 -0.00216218
## At iteration 5 value of x is: 1.097356 0.9372463 and f is: -0.0005835163 0.0006831
982
## At iteration 6 value of x is: 1.097088 0.9376519 and f is: 0.000184263 -0.00021559
48
## At iteration 7 value of x is: 1.097173 0.9375239 and f is: -5.815872e-05 6.806245e
-05
## At iteration 8 value of x is: 1.097146 0.9375643 and f is: 1.835935e-05 -2.148428e
-05
## At iteration 9 value of x is: 1.097155 0.9375515 and f is: -5.795342e-06 6.781905e
-06
## At iteration 10 value of x is: 1.097152 0.9375556 and f is: 1.829394e-06 -2.140805
e-06
## At iteration 11 value of x is: 1.097153 0.9375543 and f is: -5.774754e-07 6.757782
e-07
## At iteration 12 value of x is: 1.097152 0.9375547 and f is: 1.822889e-07 -2.133196
e-07
## At iteration 13 value of x is: 1.097153 0.9375546 and f is: -5.754227e-08 6.733757
e-08
## At iteration 14 value of x is: 1.097153 0.9375546 and f is: 1.816409e-08 -2.125613
e-08
## At iteration 15 value of x is: 1.097153 0.9375546 and f is: -5.733772e-09 6.709819
e-09
## At iteration 16 value of x is: 1.097153 0.9375546 and f is: 1.809952e-09 -2.118057
e-09
## At iteration 17 value of x is: 1.097153 0.9375546 and f is: -5.713394e-10 6.685968
e-10
## Algorithm converged
```

```
##         root1         root2        f.val1        f.val2
##  1.097153e+00  9.375546e-01 -5.713394e-10  6.685968e-10
```

```
newtonmulti(gradpois, ipois, t(as.matrix(c(0.1,0.5))), bs, N)
```

```
## At iteration 1 value of x is: 1.417701 0.451749 and f is: -0.4419727 1.38702
## At iteration 2 value of x is: 1.009561 1.070303 and f is: 0.2901992 -0.2737298
## At iteration 3 value of x is: 1.125967 0.8938844 and f is: -0.07863297 0.09904877
## At iteration 4 value of x is: 1.088177 0.9511568 and f is: 0.02615637 -0.02992437
## At iteration 5 value of x is: 1.099998 0.9332428 and f is: -0.008125141 0.00957696
6
## At iteration 6 value of x is: 1.096256 0.9389139 and f is: 0.002577974 -0.00300998
7
## At iteration 7 value of x is: 1.097436 0.9371253 and f is: -0.0008124677 0.0009514
534
## At iteration 8 value of x is: 1.097063 0.9376901 and f is: 0.0002565982 -0.0003002
105
## At iteration 9 value of x is: 1.097181 0.9375118 and f is: -8.098612e-05 9.477898e
-05
## At iteration 10 value of x is: 1.097144 0.9375681 and f is: 2.556579e-05 -2.991713
e-05
## At iteration 11 value of x is: 1.097155 0.9375503 and f is: -8.070102e-06 9.443926
e-06
## At iteration 12 value of x is: 1.097152 0.9375559 and f is: 2.547463e-06 -2.981106
e-06
## At iteration 13 value of x is: 1.097153 0.9375542 and f is: -8.041441e-07 9.410326
e-07
## At iteration 14 value of x is: 1.097152 0.9375547 and f is: 2.538404e-07 -2.970511
e-07
## At iteration 15 value of x is: 1.097153 0.9375545 and f is: -8.012857e-08 9.37687e
-08
## At iteration 16 value of x is: 1.097153 0.9375546 and f is: 2.52938e-08 -2.959951e
-08
## At iteration 17 value of x is: 1.097153 0.9375546 and f is: -7.984371e-09 9.343536
e-09
## At iteration 18 value of x is: 1.097153 0.9375546 and f is: 2.520388e-09 -2.949429
e-09
## At iteration 19 value of x is: 1.097153 0.9375546 and f is: -7.95598e-10 9.310322e
-10
## Algorithm converged
```

```
##         root1         root2        f.val1        f.val2
##  1.097153e+00  9.375546e-01 -7.955980e-10  9.310322e-10
```

```
newtonmulti(gradpois, ipois, t(as.matrix(c(0.8,0.5))), bs, N)
```

```
## At iteration 1 value of x is: 1.061965 0.9908827 and f is: 0.1071518 -0.1148385
## At iteration 2 value of x is: 1.108444 0.9204415 and f is: -0.03177974 0.03827084
## At iteration 3 value of x is: 1.093607 0.9429283 and f is: 0.01023817 -0.01187422
## At iteration 4 value of x is: 1.098274 0.9358555 and f is: -0.003211356 0.00376868
8
## At iteration 5 value of x is: 1.096799 0.9380907 and f is: 0.001015757 -0.00118760
5
## At iteration 6 value of x is: 1.097264 0.9373853 and f is: -0.0003204355 0.0003750
884
## At iteration 7 value of x is: 1.097117 0.937608 and f is: 0.0001011706 -0.00011838
21
## At iteration 8 value of x is: 1.097164 0.9375377 and f is: -3.193402e-05 3.737113e
-05
## At iteration 9 value of x is: 1.097149 0.9375599 and f is: 1.008066e-05 -1.179656e
-05
## At iteration 10 value of x is: 1.097154 0.9375529 and f is: -3.182093e-06 3.723784
e-06
## At iteration 11 value of x is: 1.097152 0.9375551 and f is: 1.004478e-06 -1.175467
e-06
## At iteration 12 value of x is: 1.097153 0.9375544 and f is: -3.170785e-07 3.710542
e-07
## At iteration 13 value of x is: 1.097152 0.9375546 and f is: 1.000907e-07 -1.171289
e-07
## At iteration 14 value of x is: 1.097153 0.9375546 and f is: -3.159514e-08 3.697351
e-08
## At iteration 15 value of x is: 1.097153 0.9375546 and f is: 9.973485e-09 -1.167125
e-08
## At iteration 16 value of x is: 1.097153 0.9375546 and f is: -3.148283e-09 3.684207
e-09
## At iteration 17 value of x is: 1.097153 0.9375546 and f is: 9.938049e-10 -1.162975
e-09
## At iteration 18 value of x is: 1.097153 0.9375546 and f is: -3.137112e-10 3.671102
e-10
## Algorithm converged
```

```
##         root1         root2        f.val1        f.val2
##  1.097153e+00  9.375546e-01 -3.137112e-10  3.671102e-10
```

```
newtonmulti(gradpois, ipois, t(as.matrix(c(2,0.5))), bs, N)
```

```
## At iteration 1 value of x is: 0.9969554 1.089408 and f is: 0.33856 -0.3098281
## At iteration 2 value of x is: 1.130314 0.8872973 and f is: -0.0897943 0.1143875
## At iteration 3 value of x is: 1.086844 0.9531772 and f is: 0.03010956 -0.03433193
## At iteration 4 value of x is: 1.100422 0.9325993 and f is: -0.009330961 0.01101008
## At iteration 5 value of x is: 1.096122 0.9391164 and f is: 0.002962824 -0.00345816
2
## At iteration 6 value of x is: 1.097478 0.9370613 and f is: -0.0009335322 0.0010933
45
## At iteration 7 value of x is: 1.09705 0.9377103 and f is: 0.0002948559 -0.00034495
9
## At iteration 8 value of x is: 1.097185 0.9375054 and f is: -9.305857e-05 0.0001089
087
## At iteration 9 value of x is: 1.097142 0.9375701 and f is: 2.937706e-05 -3.437697e
-05
## At iteration 10 value of x is: 1.097156 0.9375497 and f is: -9.273145e-06 1.085178
e-05
## At iteration 11 value of x is: 1.097152 0.9375561 and f is: 2.927226e-06 -3.425513
e-06
## At iteration 12 value of x is: 1.097153 0.9375541 and f is: -9.240217e-07 1.081317
e-06
## At iteration 13 value of x is: 1.097152 0.9375547 and f is: 2.916816e-07 -3.413339
e-07
## At iteration 14 value of x is: 1.097153 0.9375545 and f is: -9.207372e-08 1.077472
e-07
## At iteration 15 value of x is: 1.097153 0.9375546 and f is: 2.906447e-08 -3.401205
e-08
## At iteration 16 value of x is: 1.097153 0.9375546 and f is: -9.174642e-09 1.073642
e-08
## At iteration 17 value of x is: 1.097153 0.9375546 and f is: 2.896115e-09 -3.389114
e-09
## At iteration 18 value of x is: 1.097153 0.9375546 and f is: -9.142008e-10 1.069826
e-09
## At iteration 19 value of x is: 1.097153 0.9375546 and f is: 2.885815e-10 -3.377061
e-10
## Algorithm converged
```

```
##         root1         root2        f.val1        f.val2
##  1.097153e+00  9.375546e-01  2.885815e-10 -3.377061e-10
```

```
newtonmulti(gradpois, ipois, t(as.matrix(c(2,2))), bs, N)
```

```
## At iteration 1 value of x is: 1.117785 0.9062845 and f is: -0.05712725 0.07045913
## At iteration 2 value of x is: 1.090702 0.9473314 and f is: 0.01871967 -0.02155272
## At iteration 3 value of x is: 1.099195 0.9344591 and f is: -0.005841277 0.00687103
7
## At iteration 4 value of x is: 1.096508 0.9385308 and f is: 0.001850667 -0.00216218
## At iteration 5 value of x is: 1.097356 0.9372463 and f is: -0.0005835163 0.0006831
982
## At iteration 6 value of x is: 1.097088 0.9376519 and f is: 0.000184263 -0.00021559
48
## At iteration 7 value of x is: 1.097173 0.9375239 and f is: -5.815872e-05 6.806245e
-05
## At iteration 8 value of x is: 1.097146 0.9375643 and f is: 1.835935e-05 -2.148428e
-05
## At iteration 9 value of x is: 1.097155 0.9375515 and f is: -5.795342e-06 6.781905e
-06
## At iteration 10 value of x is: 1.097152 0.9375556 and f is: 1.829394e-06 -2.140805
e-06
## At iteration 11 value of x is: 1.097153 0.9375543 and f is: -5.774754e-07 6.757782
e-07
## At iteration 12 value of x is: 1.097152 0.9375547 and f is: 1.822889e-07 -2.133196
e-07
## At iteration 13 value of x is: 1.097153 0.9375546 and f is: -5.754227e-08 6.733757
e-08
## At iteration 14 value of x is: 1.097153 0.9375546 and f is: 1.816409e-08 -2.125613
e-08
## At iteration 15 value of x is: 1.097153 0.9375546 and f is: -5.733772e-09 6.709819
e-09
## At iteration 16 value of x is: 1.097153 0.9375546 and f is: 1.809952e-09 -2.118057
e-09
## At iteration 17 value of x is: 1.097153 0.9375546 and f is: -5.713394e-10 6.685968
e-10
## Algorithm converged
```

```
##         root1         root2        f.val1        f.val2
##  1.097153e+00  9.375546e-01 -5.713394e-10  6.685968e-10
```

All converge to the root found in (a), $\hat{\boldsymbol{\alpha}}_{MLE} = (\mathbf{1.0972, 0.9376})$, even the (2,2) starting point which converged to a negative α_2 previously.

(c)

Compare the implementation ease and performance of the two methods for this dataset.

The Fisher approach results in a matrix with slightly fewer terms in each position than the Hessian, but the derivation and implementation were fairly similar. We can look at runtime as a measure of performance, however:

```
tot_time_H <- numeric(100)
tot_time_F <- numeric(100)
for (i in 1:100) {
  # Run time to generate Hessian matrix
  start_time_H <- Sys.time()
  hesspois(testalphas, bs, N)
  end_time_H <- Sys.time()
  tot_time_H[i] <- as.numeric(end_time_H - start_time_H)
  #print(paste0("Hessian method takes ", tot_time_H))

  # Run time to generate Fisher Information matrix
  start_time_F <- Sys.time()
  ipois(testalphas, bs, N)
  end_time_F <- Sys.time()
  tot_time_F[i] <- as.numeric(end_time_F - start_time_F)
  #print(paste0("Fisher Information method takes ", tot_time_F))
}
print(paste0("Hessian method takes an average of ", mean(tot_time_H)," over 100 attem
pts on the same data."))
```

```
## [1] "Hessian method takes an average of 1.49726867675781e-05 over 100 attempts on
the same data."
```

```
print(paste0("Fisher Information method takes an average of ", mean(tot_time_F)," ove
r 100 attempts on the same data."))
```

```
## [1] "Fisher Information method takes an average of 1.29485130310059e-05 over 100 a
ttempts on the same data."
```

```
print(paste0("Fisher Information method is, on average, ",round((mean(tot_time_H) - m
ean(tot_time_F))/mean(tot_time_H)*100, digits = 3),"% faster than the Hessian method
on this data."))
```

```
## [1] "Fisher Information method is, on average, 13.519% faster than the Hessian met
hod on this data."
```

We can see that calculating the Fisher Information matrix is slightly faster than calculating the Hessian.

(d)

Estimate the standard errors of the MLEs of α_1 and α_2.

We can estimate the standard errors by taking the square root of the diagonal of the inverse Hessian (or its approximation):

```
sqrt(diag(solve(-hesspois(res[1:2], bs, N))))   # Estimated standard errors of MLEs i
s square root of diagonal of inverse Hessian matrix
```

```
## [1] 0.3896129 0.5546760
```

```
sqrt(diag(solve(-ipois(res[1:2], bs, N))))   # Estimated standard errors of MLEs is s
quare root of diagonal of inverse Fisher Information matrix
```

```
## [1] 0.4375560 0.6314687
```

We see that the standard errors when using the actual Hessian matrix are smaller than when using the Fisher Information matrix in its place. In both cases, the standard error for α_1 is smaller than that of α_2.