Homework 2

Jieqi Tu

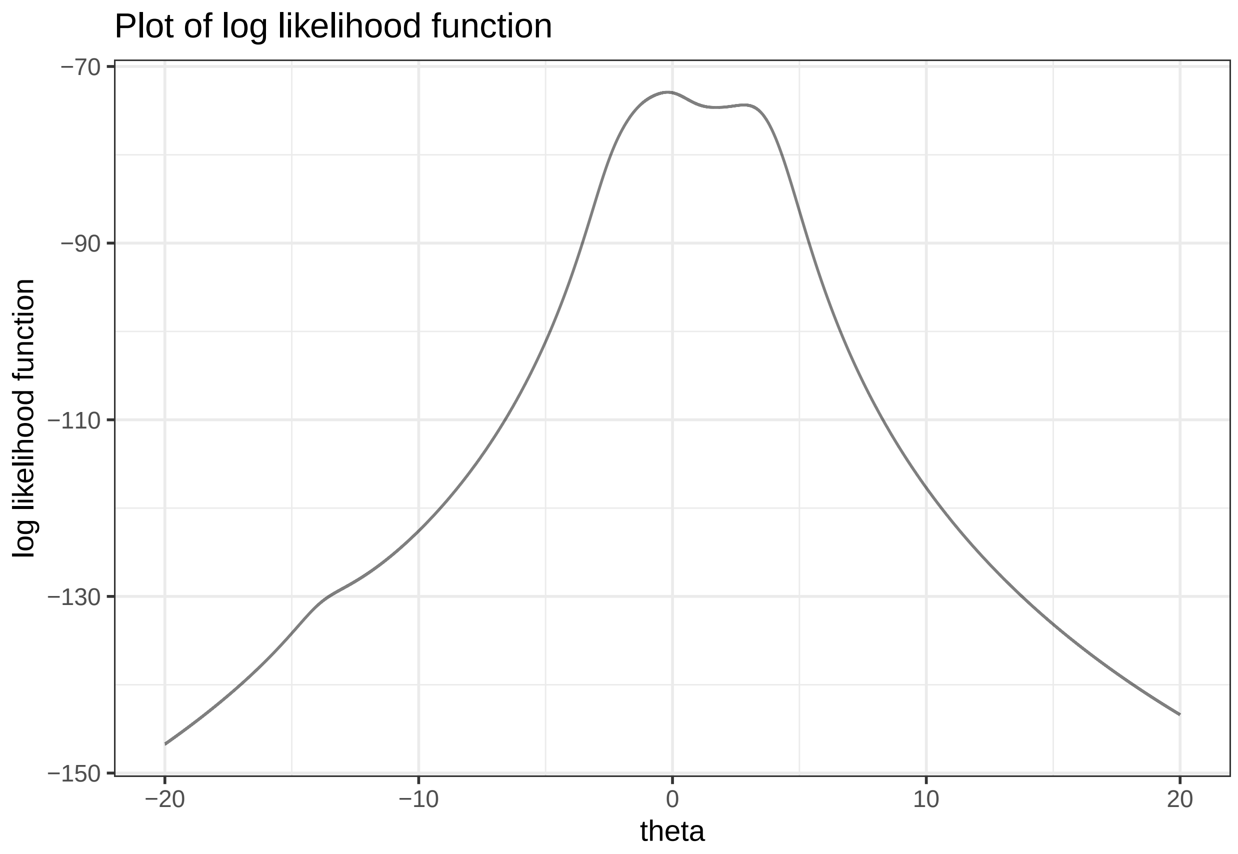
9/28/2020

Question 1

(a)

Plot the log likelihood function.

```r
# Import data
x_value = 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)

# Define the log likelihood function
log_likelihood = function(x, theta) {
  l = -20*log(pi) - sum(log(1+(x-theta)^2))
  return(l)
}

# Define the first derivative function of the log likelihood function
l_derivative_1 = function(x, theta) {
  l_1 = 2*sum((x-theta)/(1+(x-theta)^2))
  return(l_1)
}

# Define the second derivative function of the log likelihood function
l_derivative_2 = function(x, theta) {
  l_2 = 2*sum(((x-theta)^2-1)/(1+(x-theta)^2)^2)
  return(l_2)
}

# Graph the log likelihood function
theta_range = seq(-20, 20, 0.001)
n_loop = length(theta_range)
log_likelihood_value = numeric(n_loop)
for (i in 1:n_loop) {
  log_likelihood_value[i] = log_likelihood(x_value, theta_range[i])
}
result_l = cbind(theta_range, log_likelihood_value) %>% as.data.frame()
result_l %>%
  ggplot(aes(x = theta_range, y = log_likelihood_value)) + geom_line(alpha = 0.5) +
  theme_bw() + labs(
    x = "theta",
    y = "log likelihood function",
    title = "Plot of log likelihood function"
  )
```



Find the MLE for theta using Newton-Raphson method.

```
n_iteration = 1000
theta_iteration = numeric(n_iteration)
theta_iteration[1] = mean(x_value)
# Mean of the data as the starting point
for (i in 2:n_iteration) {
  theta_iteration[i] = theta_iteration[i-1] - l_derivative_1(x_value, theta_iteration[i-1])/l_derivative
}
tail(theta_iteration)
```

```
## [1] 54.87662 54.87662 54.87662 54.87662 54.87662 54.87662
```

```
# Starting point = -11
theta_iteration[1] = -11
for (i in 2:n_iteration) {
  theta_iteration[i] = theta_iteration[i-1] - l_derivative_1(x_value, theta_iteration[i-1])/l_derivative
}
tail(theta_iteration)
```

```
## [1] NaN NaN NaN NaN NaN NaN
```

```r
# Starting point = -1
theta_iteration[1] = -1
for (i in 2:n_iteration) {
  theta_iteration[i] = theta_iteration[i-1] - l_derivative_1(x_value, theta_iteration[i-1])/l_derivativ
}
tail(theta_iteration)
```

```
## [1] -0.1922866 -0.1922866 -0.1922866 -0.1922866 -0.1922866 -0.1922866
```

```r
# Starting point = 0
theta_iteration[1] = 0
for (i in 2:n_iteration) {
  theta_iteration[i] = theta_iteration[i-1] - l_derivative_1(x_value, theta_iteration[i-1])/l_derivativ
}
tail(theta_iteration)
```

```
## [1] -0.1922866 -0.1922866 -0.1922866 -0.1922866 -0.1922866 -0.1922866
```

```r
# Starting point = 1.5
theta_iteration[1] = 1.5
for (i in 2:n_iteration) {
  theta_iteration[i] = theta_iteration[i-1] - l_derivative_1(x_value, theta_iteration[i-1])/l_derivativ
}
tail(theta_iteration)
```

```
## [1] 1.713587 1.713587 1.713587 1.713587 1.713587 1.713587
```

```r
# Starting point = 4
theta_iteration[1] = 4
for (i in 2:n_iteration) {
  theta_iteration[i] = theta_iteration[i-1] - l_derivative_1(x_value, theta_iteration[i-1])/l_derivativ
}
tail(theta_iteration)
```

```
## [1] 2.817472 2.817472 2.817472 2.817472 2.817472 2.817472
```

```r
# Starting point = 4.7
theta_iteration[1] = 4.7
for (i in 2:n_iteration) {
  theta_iteration[i] = theta_iteration[i-1] - l_derivative_1(x_value, theta_iteration[i-1])/l_derivativ
}
tail(theta_iteration)
```

```
## [1] -0.1922866 -0.1922866 -0.1922866 -0.1922866 -0.1922866 -0.1922866
```

```r
# Starting point = 7
theta_iteration[1] = 7
for (i in 2:n_iteration) {
  theta_iteration[i] = theta_iteration[i-1] - l_derivative_1(x_value, theta_iteration[i-1])/l_derivativ
}
tail(theta_iteration)
```

```
## [1] 41.04085 41.04085 41.04085 41.04085 41.04085 41.04085
```

```
# Starting point = 8
theta_iteration[1] = 8
for (i in 2:n_iteration) {
  theta_iteration[i] = theta_iteration[i-1] - l_derivative_1(x_value, theta_iteration[i-1])/l_derivativ
}
tail(theta_iteration)
```

```
## [1] NaN NaN NaN NaN NaN NaN
```

```
# Starting point = 38
theta_iteration[1] = 38
for (i in 2:n_iteration) {
  theta_iteration[i] = theta_iteration[i-1] - l_derivative_1(x_value, theta_iteration[i-1])/l_derivativ
}
tail(theta_iteration)
```

```
## [1] 42.79538 42.79538 42.79538 42.79538 42.79538 42.79538
```

Discussion: the true theta would be around 0 and a little bit less than 0 (from the plot). So from our results, starting point -1 and 0 did a great job. Newton-Raphson method is very sensitive to the starting points. The mean of data is not a good starting point. We should set the starting point close to what we guess from the plot.

(b)

Use Bisection method with starting points -1 and 1.

```
# Starting points = -1 and 1
L = -1
U = 1
n_iteration = 100000
theta_iteration = numeric(n_iteration)
for (i in 1:n_iteration) {
  theta_iteration[i] = (L + U)/2
  eval_1 = l_derivative_1(x_value, theta_iteration[i])
  eval_2 = l_derivative_1(x_value, L)
  eval_3 = eval_1 * eval_2
  if(eval_3 < 0) {
    U = theta_iteration[i]
  } else {L = theta_iteration[i]}
}

tail(theta_iteration)
```

```
## [1] -0.1922866 -0.1922866 -0.1922866 -0.1922866 -0.1922866 -0.1922866
```

```
# Starting points = 1 and 2
L = 1
U = 2
```

```r
n_iteration = 100000
theta_iteration = numeric(n_iteration)
for (i in 1:n_iteration) {
  theta_iteration[i] = (L + U)/2
  eval_1 = l_derivative_1(x_value, theta_iteration[i])
  eval_2 = l_derivative_1(x_value, L)
  eval_3 = eval_1 * eval_2
  if(eval_3 < 0) {
    U = theta_iteration[i]
  } else {L = theta_iteration[i]}
}

tail(theta_iteration)
```

```
## [1] 1.713587 1.713587 1.713587 1.713587 1.713587 1.713587
```

```r
# Starting points = 50 and 52
L = 50
U = 55
n_iteration = 100000
theta_iteration = numeric(n_iteration)
for (i in 1:n_iteration) {
  theta_iteration[i] = (L + U)/2
  eval_1 = l_derivative_1(x_value, theta_iteration[i])
  eval_2 = l_derivative_1(x_value, L)
  eval_3 = eval_1 * eval_2
  if(eval_3 < 0) {
    U = theta_iteration[i]
  } else {L = theta_iteration[i]}
}

tail(theta_iteration)
```

```
## [1] 54.87662 54.87662 54.87662 54.87662 54.87662 54.87662
```

Bisection method sometimes still fails to find the globlal maximum. Since it would be easier to get the local maximum near the starting points, sometimes it is possible to converge to a local maximum. Moreover, Bisection method might be slower than Newton-Raphson method.

(c)

Apply fixed-point iterations, starting from -1, with and without scaling.

```r
# Define the g function
g_function = function(theta, alpha) {
  g_value = theta + alpha*l_derivative_1(x_value, theta)
  return(g_value)
}
# Without scaling (alpha = 1)
n_iteration = 2000000
theta_iteration = numeric(n_iteration)
theta_iteration[1] = -1
```

```
for (i in 2:n_iteration) {
  theta_iteration[i] = g_function(theta_iteration[i-1], 1)
}

tail(theta_iteration)
```

```
## [1] -0.8210256  0.6863265 -0.8584719  0.7297522 -0.7781577  0.6392481
```

```
# Alpha = 0.64
n_iteration = 100000
theta_iteration = numeric(n_iteration)
theta_iteration[1] = -1
for (i in 2:n_iteration) {
  theta_iteration[i] = g_function(theta_iteration[i-1], 0.64)
}

tail(theta_iteration)
```

```
## [1] -0.1922866 -0.1922866 -0.1922866 -0.1922866 -0.1922866 -0.1922866
```

```
# Alpha = 0.25
n_iteration = 100
theta_iteration = numeric(n_iteration)
theta_iteration[1] = -1
for (i in 2:n_iteration) {
  theta_iteration[i] = g_function(theta_iteration[i-1], 0.25)
}

tail(theta_iteration)
```

```
## [1] -0.1922866 -0.1922866 -0.1922866 -0.1922866 -0.1922866 -0.1922866
```

```
head(theta_iteration)
```

```
## [1] -1.0000000 -0.5196461 -0.2999846 -0.2198341 -0.1987656 -0.1937823
```

With the same starting point -1, the more shrinkage we have on the original function, the faster we can reach the convergence of theta.

```
# Alpha = 0.25, starting point = 0
n_iteration = 100
theta_iteration = numeric(n_iteration)
theta_iteration[1] = 0
for (i in 2:n_iteration) {
  theta_iteration[i] = g_function(theta_iteration[i-1], 0.25)
}

tail(theta_iteration)
```

```
## [1] -0.1922866 -0.1922866 -0.1922866 -0.1922866 -0.1922866 -0.1922866
```

```
theta_iteration
```

```
##   [1]  0.0000000 -0.1502570 -0.1829210 -0.1901521 -0.1917973 -0.1921743
##   [7] -0.1922608 -0.1922807 -0.1922853 -0.1922863 -0.1922865 -0.1922866
##  [13] -0.1922866 -0.1922866 -0.1922866 -0.1922866 -0.1922866 -0.1922866
##  [19] -0.1922866 -0.1922866 -0.1922866 -0.1922866 -0.1922866 -0.1922866
##  [25] -0.1922866 -0.1922866 -0.1922866 -0.1922866 -0.1922866 -0.1922866
##  [31] -0.1922866 -0.1922866 -0.1922866 -0.1922866 -0.1922866 -0.1922866
##  [37] -0.1922866 -0.1922866 -0.1922866 -0.1922866 -0.1922866 -0.1922866
##  [43] -0.1922866 -0.1922866 -0.1922866 -0.1922866 -0.1922866 -0.1922866
##  [49] -0.1922866 -0.1922866 -0.1922866 -0.1922866 -0.1922866 -0.1922866
##  [55] -0.1922866 -0.1922866 -0.1922866 -0.1922866 -0.1922866 -0.1922866
##  [61] -0.1922866 -0.1922866 -0.1922866 -0.1922866 -0.1922866 -0.1922866
##  [67] -0.1922866 -0.1922866 -0.1922866 -0.1922866 -0.1922866 -0.1922866
##  [73] -0.1922866 -0.1922866 -0.1922866 -0.1922866 -0.1922866 -0.1922866
##  [79] -0.1922866 -0.1922866 -0.1922866 -0.1922866 -0.1922866 -0.1922866
##  [85] -0.1922866 -0.1922866 -0.1922866 -0.1922866 -0.1922866 -0.1922866
##  [91] -0.1922866 -0.1922866 -0.1922866 -0.1922866 -0.1922866 -0.1922866
##  [97] -0.1922866 -0.1922866 -0.1922866 -0.1922866
```

```
# Alpha = 0.25, starting point = 2
n_iteration = 100
theta_iteration = numeric(n_iteration)
theta_iteration[1] = 2
for (i in 2:n_iteration) {
  theta_iteration[i] = g_function(theta_iteration[i-1], 0.25)
}

tail(theta_iteration)
```

```
## [1] 2.817472 2.817472 2.817472 2.817472 2.817472 2.817472
```

```
theta_iteration
```

```
##   [1] 2.000000 2.053837 2.116650 2.189510 2.273055 2.366643 2.466962 2.566709
##   [9] 2.655287 2.723247 2.767730 2.792994 2.805933 2.812156 2.815051 2.816375
##  [17] 2.816976 2.817248 2.817371 2.817427 2.817452 2.817463 2.817468 2.817470
##  [25] 2.817471 2.817472 2.817472 2.817472 2.817472 2.817472 2.817472 2.817472
##  [33] 2.817472 2.817472 2.817472 2.817472 2.817472 2.817472 2.817472 2.817472
##  [41] 2.817472 2.817472 2.817472 2.817472 2.817472 2.817472 2.817472 2.817472
##  [49] 2.817472 2.817472 2.817472 2.817472 2.817472 2.817472 2.817472 2.817472
##  [57] 2.817472 2.817472 2.817472 2.817472 2.817472 2.817472 2.817472 2.817472
##  [65] 2.817472 2.817472 2.817472 2.817472 2.817472 2.817472 2.817472 2.817472
##  [73] 2.817472 2.817472 2.817472 2.817472 2.817472 2.817472 2.817472 2.817472
##  [81] 2.817472 2.817472 2.817472 2.817472 2.817472 2.817472 2.817472 2.817472
##  [89] 2.817472 2.817472 2.817472 2.817472 2.817472 2.817472 2.817472 2.817472
##  [97] 2.817472 2.817472 2.817472 2.817472
```

Having the same scaling, convergence could be totally different with different starting point. So the choice of starting point is very important, and should be close to the global maximum.

(d)

Apply the secant method to estimate theta.

```r
# Define the secant function
secant_function = function(theta1, theta2) {
  f_xi = l_derivative_1(x_value, theta2)
  f_xi_1 = l_derivative_1(x_value, theta1)
  theta_new = theta2 - f_xi*(theta2-theta1)/(f_xi-f_xi_1)
  return(theta_new)
}

# Starting points = -2 and -1
n_iteration = 10
theta_iteration = numeric(n_iteration)
theta_iteration[1] = -2
theta_iteration[2] = -1
for (i in 3:n_iteration) {
  theta_iteration[i] = secant_function(theta_iteration[i-2], theta_iteration[i-1])
}
tail(theta_iteration)
```

```
## [1] -0.1984022 -0.1923655 -0.1922865 -0.1922866 -0.1922866 -0.1922866
```

```r
# Starting points = -3 and 3
n_iteration = 10
theta_iteration = numeric(n_iteration)
theta_iteration[1] = -3
theta_iteration[2] = 3
for (i in 3:n_iteration) {
  theta_iteration[i] = secant_function(theta_iteration[i-2], theta_iteration[i-1])
}
tail(theta_iteration)
```

```
## [1] 2.826083 2.817013 2.817466 2.817472 2.817472 2.817472
```

```r
# Starting points = 50 and 55
n_iteration = 10
theta_iteration = numeric(n_iteration)
theta_iteration[1] = 50
theta_iteration[2] = 55
for (i in 3:n_iteration) {
  theta_iteration[i] = secant_function(theta_iteration[i-2], theta_iteration[i-1])
}
tail(theta_iteration)
```

```
## [1] 54.87662 54.87662 54.87662 54.87662 54.87662      NaN
```

We get to the true convergence when starting points = -2 and -1. When the starting points are -3 and 3, or 50 and 55, we got the local maximum. So the starting points should be chosen carefully.

(e)

Compare the speed and stability of these methods.

- Newton-Raphson method is usually the fatest method than others, but it is also very sensitive to the starting points.
- Fixed point method is the most stable one but it is slower, the scaling can ajust the speed to convergence.
- Bisection method is also stable and slower.
- Secant is also fast. However, after reaching convergence, the theta might becomes NaN, due to the zero in denominator.

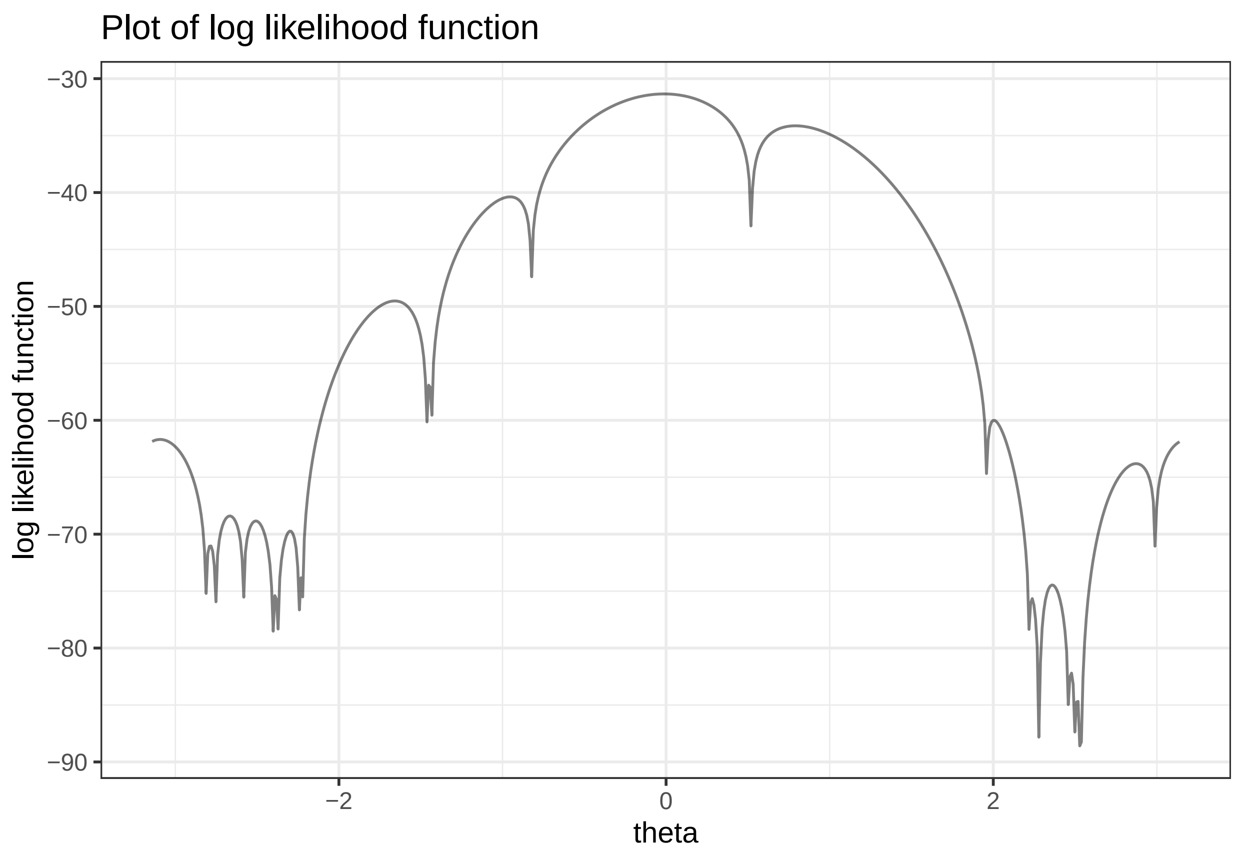
Question 2

```r
# Import data
x_observed = c(3.91, 4.85, 2.28, 4.06, 3.70, 4.04, 5.46, 3.53, 2.28, 1.96, 2.53, 3.88, 2.22, 3.47, 4.82
2.54, 0.52, 2.50)

# Define the log likelihood function
l_function = function(x, theta) {
  l_value = -20*log(2*pi) + sum(log(1-cos(x-theta)))
  return(l_value)
}

# Define the 1st derivative of the log likelihood function
l_derivative_1 = function(x, theta) {
  l_der_value = -sum(sin(x-theta)/(1-cos(x-theta)))
  return(l_der_value)
}

# Define the 2nd derivative of the log likelihood function
l_derivative_2 = function(x, theta) {
  l_second_der = -sum(1/(1-cos(x-theta)))
  return(l_second_der)
}
```

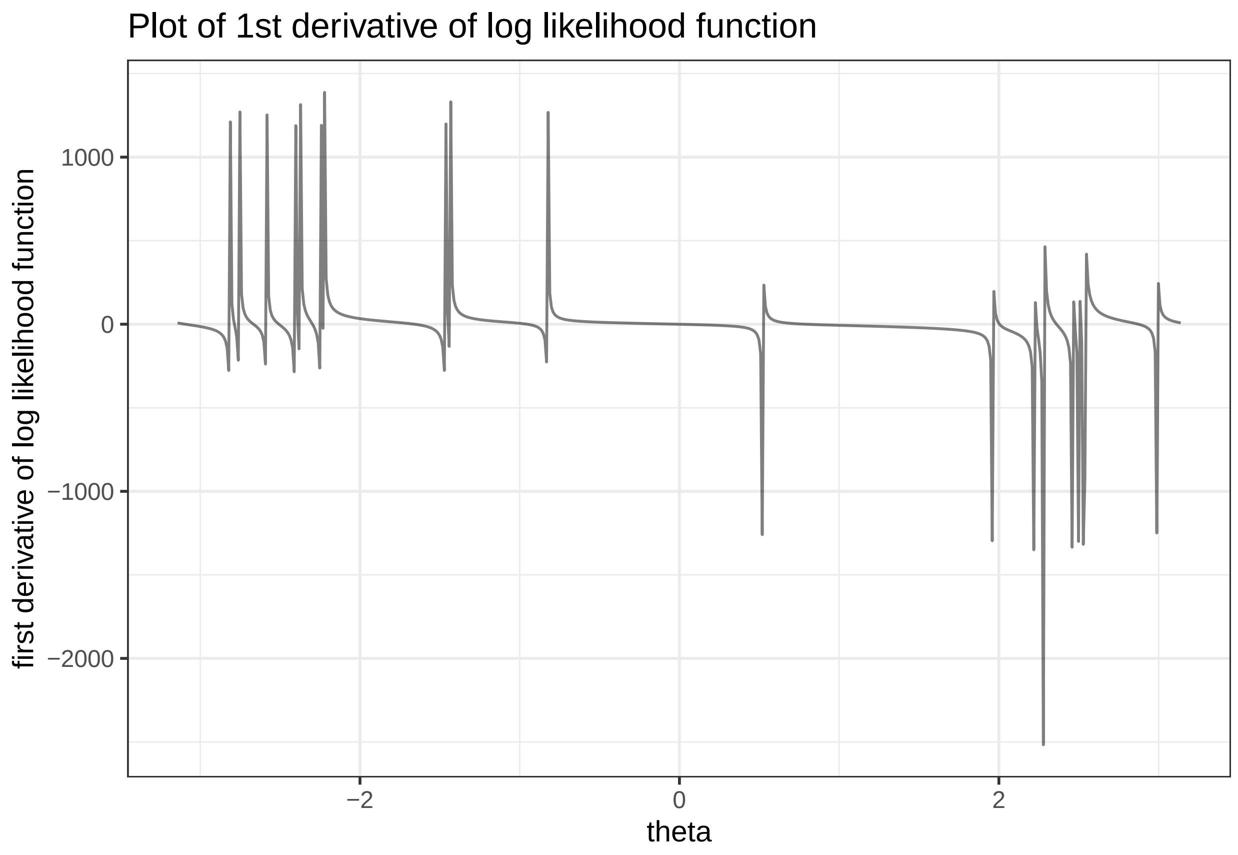
(a)

Plot the log likelihood function and the 1st derivative of the log likelihood function between $-\pi$ and π.

```r
x_range = seq(-pi, pi, 0.01)
l_value = numeric(length(x_range))
l_derivative = numeric(length(x_range))
for (i in 1:length(x_range)) {
  l_value[i] = l_function(x_observed, x_range[i])
  l_derivative[i] = l_derivative_1(x_observed, x_range[i])
}
result = cbind(x_range, l_value, l_derivative) %>% as.data.frame()

# Plot the log likelihood function
result %>%
  ggplot(aes(x = x_range, y = l_value)) + geom_line(alpha= 0.5) +
  theme_bw() + labs(
    x = "theta",
    y = "log likelihood function",
    title = "Plot of log likelihood function"
  )
```



```r
# Plot the derivative of log likelihood function
result %>%
  ggplot(aes(x = x_range, y = l_derivative)) + geom_line(alpha= 0.5) +
  theme_bw() + labs(
    x = "theta",
    y = "first derivative of log likelihood function",
    title = "Plot of 1st derivative of log likelihood function"
  )
```



(b)

Find the method-of-moments estimator of θ. The first theoretical moment about the origin is: $E(X) = \int_{-\infty}^{\infty} xf(x)dx = \int_0^{2\pi} x(1 - cos(x - \theta))/2\pi dx = \frac{1}{2\pi}(sin(\theta) + 2\pi^2)$. Therefore, we have $sin\theta = E(X) - \pi$. So the method-of-moments estimator would be $\hat{\theta} = \arcsin((E(X) - \pi))$.

```
# MME of theta
theta_mme = asin((mean(x_observed)-pi));theta_mme
```

```
## [1] 0.05844061
```

(c)

Find the MLE for θ using the Newton-Raphson method.

```
n_iteration = 100
theta_iteration = numeric(n_iteration)
theta_iteration[1] = theta_mme
# MME of theta as the starting point
for (i in 2:n_iteration) {
  theta_iteration[i] = theta_iteration[i-1] - l_derivative_1(x_observed, theta_iteration[i-1])/l_deriva
}
tail(theta_iteration)
```

```
## [1] -0.011972 -0.011972 -0.011972 -0.011972 -0.011972 -0.011972
```

```r
# Starting point = -2.7
theta_iteration = numeric(n_iteration)
theta_iteration[1] = -2.7
for (i in 2:n_iteration) {
  theta_iteration[i] = theta_iteration[i-1] - l_derivative_1(x_observed, theta_iteration[i-1])/l_deriva
}
tail(theta_iteration)
```

```
## [1] -2.6667 -2.6667 -2.6667 -2.6667 -2.6667 -2.6667
```

```r
# Starting point = 2.7
theta_iteration = numeric(n_iteration)
theta_iteration[1] = 2.7
for (i in 2:n_iteration) {
  theta_iteration[i] = theta_iteration[i-1] - l_derivative_1(x_observed, theta_iteration[i-1])/l_deriva
}
tail(theta_iteration)
```

```
## [1] 2.873095 2.873095 2.873095 2.873095 2.873095 2.873095
```

When the starting value is MME, it converges to -0.011972. When the starting value is -2.7, it converges to -2.6667. When the starting value is 2.7, it converges to 2.873095.

(d)

Repeat part (c) using 200 equally spaced starting values between $-\pi$ and π.

```r
# Create starting values
start_values = seq(-pi, pi, length = 200)

# Newton-Raphson method for each starting values
convergence = numeric(200)
n_iteration = 100
theta_iteration = numeric(n_iteration)
for (j in 1:200) {
  # Set starting value
  theta_iteration[1] = start_values[j]
  # Find the convergence theta value
  for (i in 2:n_iteration) {
    theta_iteration[i] = theta_iteration[i-1] - l_derivative_1(x_observed, theta_iteration[i-1])/l_deri
  }
  convergence[j] = theta_iteration[100]
}

result_list = cbind(start_values, convergence)
result_list %>% knitr::kable()
```

start_values	convergence
-3.1415927	-3.0930917
-3.1100189	-3.0930917
-3.0784451	-3.0930917

start_values	convergence
-3.0468713	-3.0930917
-3.0152975	-3.0930917
-2.9837237	-3.0930917
-2.9521499	-3.0930917
-2.9205761	-3.0930917
-2.8890023	-3.0930917
-2.8574285	-3.0930917
-2.8258547	-3.0930917
-2.7942809	-2.7861668
-2.7627071	-2.7861668
-2.7311333	-2.6666999
-2.6995595	-2.6666999
-2.6679857	-2.6666999
-2.6364119	-2.6666999
-2.6048381	-2.6666999
-2.5732643	-2.5076132
-2.5416905	-2.5076132
-2.5101167	-2.5076132
-2.4785429	-2.5076132
-2.4469692	-2.5076132
-2.4153954	-2.5076132
-2.3838216	-2.3882005
-2.3522478	-2.2972562
-2.3206740	-2.2972562
-2.2891002	-2.2972562
-2.2575264	-2.2972562
-2.2259526	-2.2321673
-2.1943788	-1.6582832
-2.1628050	-1.6582832
-2.1312312	-1.6582832
-2.0996574	-1.6582832
-2.0680836	-1.6582832
-2.0365098	-1.6582832
-2.0049360	-1.6582832
-1.9733622	-1.6582832
-1.9417884	-1.6582832
-1.9102146	-1.6582832
-1.8786408	-1.6582832
-1.8470670	-1.6582832
-1.8154932	-1.6582832
-1.7839194	-1.6582832
-1.7523457	-1.6582832
-1.7207719	-1.6582832
-1.6891981	-1.6582832
-1.6576243	-1.6582832
-1.6260505	-1.6582832
-1.5944767	-1.6582832
-1.5629029	-1.6582832
-1.5313291	-1.6582832
-1.4997553	-1.6582832
-1.4681815	-1.6582832
-1.4366077	-1.4474788

start_values	convergence
-1.4050339	-0.9533363
-1.3734601	-0.9533363
-1.3418863	-0.9533363
-1.3103125	-0.9533363
-1.2787387	-0.9533363
-1.2471649	-0.9533363
-1.2155911	-0.9533363
-1.1840173	-0.9533363
-1.1524435	-0.9533363
-1.1208697	-0.9533363
-1.0892959	-0.9533363
-1.0577221	-0.9533363
-1.0261484	-0.9533363
-0.9945746	-0.9533363
-0.9630008	-0.9533363
-0.9314270	-0.9533363
-0.8998532	-0.9533363
-0.8682794	-0.9533363
-0.8367056	-0.9533363
-0.8051318	-0.0119720
-0.7735580	-0.0119720
-0.7419842	-0.0119720
-0.7104104	-0.0119720
-0.6788366	-0.0119720
-0.6472628	-0.0119720
-0.6156890	-0.0119720
-0.5841152	-0.0119720
-0.5525414	-0.0119720
-0.5209676	-0.0119720
-0.4893938	-0.0119720
-0.4578200	-0.0119720
-0.4262462	-0.0119720
-0.3946724	-0.0119720
-0.3630986	-0.0119720
-0.3315249	-0.0119720
-0.2999511	-0.0119720
-0.2683773	-0.0119720
-0.2368035	-0.0119720
-0.2052297	-0.0119720
-0.1736559	-0.0119720
-0.1420821	-0.0119720
-0.1105083	-0.0119720
-0.0789345	-0.0119720
-0.0473607	-0.0119720
-0.0157869	-0.0119720
0.0157869	-0.0119720
0.0473607	-0.0119720
0.0789345	-0.0119720
0.1105083	-0.0119720
0.1420821	-0.0119720
0.1736559	-0.0119720
0.2052297	-0.0119720

start_values	convergence
0.2368035	-0.0119720
0.2683773	-0.0119720
0.2999511	-0.0119720
0.3315249	-0.0119720
0.3630986	-0.0119720
0.3946724	-0.0119720
0.4262462	-0.0119720
0.4578200	-0.0119720
0.4893938	-0.0119720
0.5209676	0.7906013
0.5525414	0.7906013
0.5841152	0.7906013
0.6156890	0.7906013
0.6472628	0.7906013
0.6788366	0.7906013
0.7104104	0.7906013
0.7419842	0.7906013
0.7735580	0.7906013
0.8051318	0.7906013
0.8367056	0.7906013
0.8682794	0.7906013
0.8998532	0.7906013
0.9314270	0.7906013
0.9630008	0.7906013
0.9945746	0.7906013
1.0261484	0.7906013
1.0577221	0.7906013
1.0892959	0.7906013
1.1208697	0.7906013
1.1524435	0.7906013
1.1840173	0.7906013
1.2155911	0.7906013
1.2471649	0.7906013
1.2787387	0.7906013
1.3103125	0.7906013
1.3418863	0.7906013
1.3734601	0.7906013
1.4050339	0.7906013
1.4366077	0.7906013
1.4681815	0.7906013
1.4997553	0.7906013
1.5313291	0.7906013
1.5629029	0.7906013
1.5944767	0.7906013
1.6260505	0.7906013
1.6576243	0.7906013
1.6891981	0.7906013
1.7207719	0.7906013
1.7523457	0.7906013
1.7839194	0.7906013
1.8154932	0.7906013
1.8470670	0.7906013

start_values	convergence
1.8786408	0.7906013
1.9102146	0.7906013
1.9417884	0.7906013
1.9733622	2.0036449
2.0049360	2.0036449
2.0365098	2.0036449
2.0680836	2.0036449
2.0996574	2.0036449
2.1312312	2.0036449
2.1628050	2.0036449
2.1943788	2.0036449
2.2259526	2.2362194
2.2575264	2.2362194
2.2891002	2.3607182
2.3206740	2.3607182
2.3522478	2.3607182
2.3838216	2.3607182
2.4153954	2.3607182
2.4469692	2.3607182
2.4785429	2.4753736
2.5101167	2.5135932
2.5416905	2.8730945
2.5732643	2.8730945
2.6048381	2.8730945
2.6364119	2.8730945
2.6679857	2.8730945
2.6995595	2.8730945
2.7311333	2.8730945
2.7627071	2.8730945
2.7942809	2.8730945
2.8258547	2.8730945
2.8574285	2.8730945
2.8890023	2.8730945
2.9205761	2.8730945
2.9521499	2.8730945
2.9837237	2.8730945
3.0152975	3.1900936
3.0468713	3.1900936
3.0784451	3.1900936
3.1100189	3.1900936
3.1415927	3.1900936

Discussion: Different starting values have different convergence of theta. However, only starting values from -0.8051318 to 0.4893938 lead to the true maximum.

(e)

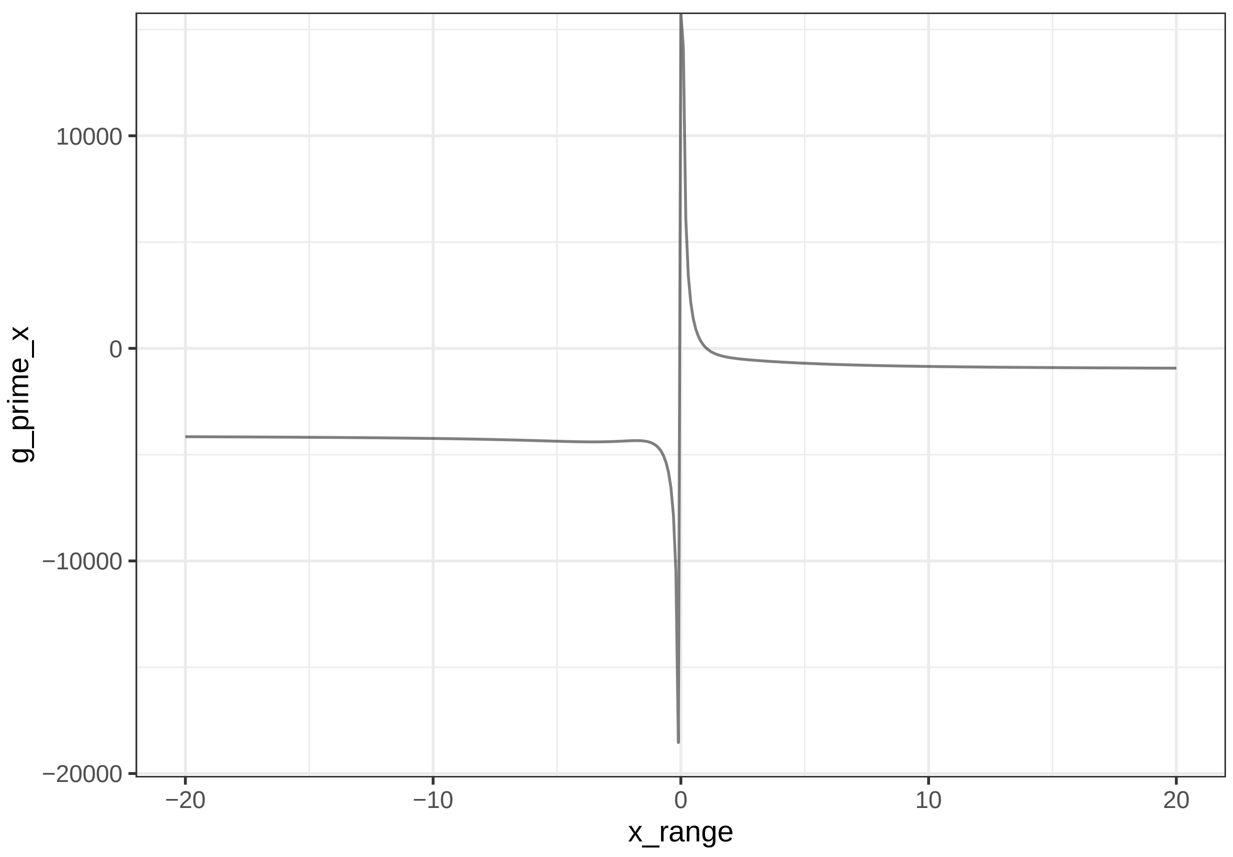
Find two starting values, as nearly equal as you can, for which the Newton–Raphson method converges to two different solutions. From part (d), we could know that, we can split the range of $[-\pi, \pi]$ as much as possible. We could assign the space as small as possible.

Question 3

(a)

Use fixed point iteration with $G_1(x)$.

```r
# Define g'(x)
g_x_derivative = function(epsilon, x) {
  g_x_value = 1628/x - 1013 - 3062*(1-epsilon)*exp(-x)/(epsilon+(1-epsilon)*exp(-x))
  return(g_x_value)
}

# Define G1(x)
G1_x = function(epsilon, x) {
  G1_value = 1628*(epsilon+(1-epsilon)*exp(-x))/(3062*(1-epsilon)*exp(-x)+1013*(epsilon+(1-epsilon)*exp
  return(G1_value)
}

# Plot the g'(x)
x_range = seq(-20, 20, 0.1)
g_prime_x = numeric(length(x_range))
for (i in 1:length(x_range)) {
  g_prime_x[i] = g_x_derivative(0.61489, x_range[i])
}
result = cbind(x_range, g_prime_x) %>% as.data.frame()
result %>%
  ggplot(aes(x = x_range, y = g_prime_x)) + geom_line(alpha = 0.5) +
  theme_bw()
```



```r
# Fixed point iteration with G1(x)
n_iteration = 20
x_list = numeric(n_iteration)
x_list = 1 # Starting value is 1
for (i in 2:n_iteration) {
  x_list[i] = G1_x(0.61489, x_list[i-1])
}
g_prime_x=g_x_derivative(0.61489, x_list)
result = cbind(x_list, g_prime_x) %>% as.data.frame()
result %>% knitr::kable()
```

x_list	g_prime_x
1.000000	41.6104171
1.026230	12.1233052
1.034132	3.6135223
1.036512	1.0843050
1.037228	0.3260186
1.037443	0.0980832
1.037508	0.0295138
1.037528	0.0088814
1.037534	0.0026726
1.037535	0.0008043
1.037536	0.0002420
1.037536	0.0000728
1.037536	0.0000219

x_list	g_prime_x
1.037536	0.0000066
1.037536	0.0000020
1.037536	0.0000006
1.037536	0.0000002
1.037536	0.0000001
1.037536	0.0000000
1.037536	0.0000000

Using fixed point iteration, $G_1(x)$ converges to 1.037536 with starting value 1.

(b)

Demonstrate that $G_2(x) = x + g'(x)$ fails to converge.

```r
# Define G2(x)
G2_x = function(epsilon, x) {
  G2_value = x + 1628/x - 1013 - 3062*(1-epsilon)*exp(-x)/(epsilon+(1-epsilon)*exp(-x))
  return(G2_value)
}

# Fixed point iteration with G2(x)
n_iteration = 5
x_list = numeric(n_iteration)
x_list[1] = 1 # Starting value = 1
for (i in 2:n_iteration) {
  x_list[i] = G2_x(0.61489, x_list[i-1])
}
g_prime_x=g_x_derivative(0.61489, x_list)
result = cbind(x_list, g_prime_x) %>% as.data.frame()
result %>% knitr::kable()
```

x_list	g_prime_x
1.00000	41.61042
42.61042	-974.79338
-932.18296	NaN
NaN	NaN
NaN	NaN

The x values become NaN after a few iterations. So it is not convergent to some value with starting point 1.

(c)

Try $G_3(x) = x + \alpha g'(x), \alpha = 1/1000$.

```r
# Define G3(x)
G3_x = function(epsilon, x, alpha) {
  G3_value = x + alpha*(1628/x - 1013 - 3062*(1-epsilon)*exp(-x)/(epsilon+(1-epsilon)*exp(-x)))
  return(G3_value)
}
```

```r
# Fixed point iteration with scaling
n_iteration = 10
x_list = numeric(n_iteration)
x_list[1] = 1
for (i in 2:n_iteration) {
  x_list[i] = G3_x(0.61489, x_list[i-1], 0.001)
}
g_prime_x=g_x_derivative(0.61489, x_list)
result = cbind(x_list, g_prime_x) %>% as.data.frame()
result %>% knitr::kable()
```

x_list	g_prime_x
1.000000	41.6104171
1.041610	-4.2858584
1.037325	0.2236357
1.037548	-0.0128569
1.037535	0.0007356
1.037536	-0.0000421
1.037536	0.0000024
1.037536	-0.0000001
1.037536	0.0000000
1.037536	0.0000000

After scaling with $\alpha = 0.001$, it converges to 1.037536.

(d)

Newton-Raphson method using starting value 3.

```r
# Define the second derivative of g(x)
g_x_derivative_2 = function(epsilon, x) {
  second_deri_value = 3062*(1-epsilon)*epsilon*exp(-x)/(epsilon+(1-epsilon)*exp(-x))^2-1628/x^2
  return(second_deri_value)
}

# Newton-Raphson iteration with starting value 3
n_iteration = 5
x_list = numeric(n_iteration)
x_list[1] = 3
for (i in 2:n_iteration) {
  x_list[i] = x_list[i-1] - g_x_derivative(0.61489, x_list[i-1]/g_x_derivative_2(0.61489, x_list[i-1]))
}
g_prime_x=g_x_derivative(0.61489, x_list)
result = cbind(x_list, g_prime_x) %>% as.data.frame()
result %>% knitr::kable()
```

x_list	g_prime_x
3.00	-562.9254
51654.33	-1012.9685
NaN	NaN

x_list	g_prime_x
NaN	NaN
NaN	NaN

(e)

Use Newton-Raphson method with starting value = 1.5.

```r
n_iteration = 20
x_list = numeric(n_iteration)
x_list[1] = 1.5 # Starting value = 1.5
for (i in 2:n_iteration) {
  x_list[i] = x_list[i-1] - g_x_derivative(0.61489, x_list[i-1])/g_x_derivative_2(0.61489, x_list[i-1])
}
g_prime_x=g_x_derivative(0.61489, x_list)
result = cbind(x_list, g_prime_x) %>% as.data.frame()
result %>% knitr::kable()
```

x_list	g_prime_x
1.5000000	-303.1079151
0.7309801	504.7674436
0.9327450	126.9932930
1.0244014	14.1159457
1.0373228	0.2254716
1.0375360	0.0000597
1.0375360	0.0000000
1.0375360	0.0000000
1.0375360	0.0000000
1.0375360	0.0000000
1.0375360	0.0000000
1.0375360	0.0000000
1.0375360	0.0000000
1.0375360	0.0000000
1.0375360	0.0000000
1.0375360	0.0000000
1.0375360	0.0000000
1.0375360	0.0000000
1.0375360	0.0000000
1.0375360	0.0000000

(f)

Use secant method with starting value 1.5 and 1.49.

```r
# Define the secant function
secant_function = function(x_1, x_2) {
  f_xi = g_x_derivative(0.61489, x_2)
  f_xi_1 = g_x_derivative(0.61489, x_1)
  x_new = x_2 - f_xi*(x_2-x_1)/(f_xi-f_xi_1)
  return(x_new)
}
```

```r
# Secant method with starting value = 1.5 and 1.49
n_iteration = 10
x_list = numeric(n_iteration)
x_list[1] = 1.5
x_list[2] = 1.49
for (i in 3:n_iteration) {
  x_list[i] = secant_function(x_list[i-2], x_list[i-1])
}
g_prime_x=g_x_derivative(0.61489, x_list)
result = cbind(x_list, g_prime_x) %>% as.data.frame()
result %>% knitr::kable()
```

x_list	g_prime_x
1.5000000	-303.1079151
1.4900000	-299.1303221
0.7379615	487.4956992
1.2040223	-145.1702726
1.0970809	-58.5768213
1.0247396	13.7467313
1.0384897	-1.0070132
1.0375512	-0.0159851
1.0375360	0.0000189
1.0375360	0.0000000

(g)

Use Muller's method. Choose your third starting value as 1.48.

```r
# Initialization
n_iteration = 10
x_list = numeric(n_iteration)
x_list[1] = 1.5
x_list[2] = 1.49
x_list[3] = 1.48

# Define Muller's function
muller_function = function(x1,x2,x3) {
  f_i_2 = g_x_derivative(0.61489, x1)
  f_i_1 = g_x_derivative(0.61489, x2)
  f_i = g_x_derivative(0.61489, x3)
  dd = (f_i - 2*f_i_1 + f_i_2)/(x3 - x1)
  s = f_i - f_i_1 + (x3 - x1)/dd
  q = (x3-x2)/(x2-x1)
  A = q*f_i - q*(1+q)*f_i_1 + q^2*f_i_2
  B = (2*q+1)*f_i-(1+q)^2*f_i_1+q^2*f_i_2
  C = (1+q)*f_i
  x4 = x3-(x3-x2)*2*C/(B+sign(s)*sqrt(B^2-4*A*C))
  return(x4)
}

# Iterations
```

```r
for (i in 4:n_iteration) {
  x_list[i] = muller_function(x_list[i-3], x_list[i-2], x_list[i-1])
}
g_prime_x=g_x_derivative(0.61489, x_list)
result = cbind(x_list, g_prime_x) %>% as.data.frame()
result %>% knitr::kable()
```

x_list	g_prime_x
1.5000000	-303.1079151
1.4900000	-299.1303221
1.4800000	-295.0792741
0.9814546	63.6802061
1.0459032	-8.7548905
1.0377325	-0.2076829
1.0375362	-0.0001350
1.0375360	0.0000000
1.0375360	0.0000000
1.0375360	0.0000000

(h)

Use bisection method. The starting values would be 1 and 2.

```r
L = 1
U = 2
n_iteration = 25
x_list = numeric(n_iteration)
for (i in 1:n_iteration) {
  x_list[i] = (L + U)/2
  eval_1 = g_x_derivative(0.61489, x_list[i])
  eval_2 = g_x_derivative(0.61489, L)
  eval_3 = eval_1 * eval_2
  if(eval_3 < 0) {
    U = x_list[i]
  } else {L = x_list[i]}
}

g_prime_x=g_x_derivative(0.61489, x_list)
result = cbind(x_list, g_prime_x) %>% as.data.frame()
result %>% knitr::kable()
```

x_list	g_prime_x
1.500000	-303.1079151
1.250000	-176.4526784
1.125000	-83.2880692
1.062500	-25.5957503
1.031250	6.6979962
1.046875	-9.7600619
1.039062	-1.6107709
1.035156	2.5234289
1.037109	0.4513145

x_list	g_prime_x
1.038086	-0.5809779
1.037598	-0.0651446
1.037354	0.1930067
1.037476	0.0639115
1.037537	-0.0006214
1.037506	0.0316438
1.037521	0.0155109
1.037529	0.0074446
1.037533	0.0034116
1.037535	0.0013951
1.037536	0.0003868
1.037536	-0.0001173
1.037536	0.0001347
1.037536	0.0000087
1.037536	-0.0000543
1.037536	-0.0000228

(i)

Use secant-bracket method. The starting values are 1 and 2.

```r
n_iteration = 20
x_list = numeric(n_iteration)
L = 1 #
U = 2
```