# Assignment 1

xw-zeng

2022-10-05

Problem sets.

- `2.1 (a)(b)(c)(d)(e)`
- `2.2 (a)(d)`
- `2.5 (a)(b)(c)(d)(e)(f)(g)`

Load the R packages.

```
library(ggplot2) # For visualization.
library(Deriv) # For derivative calculation.
library(patchwork) # For plot composing.
library(RColorBrewer) # For generating beautiful colormap.
```

# README

List of common functions.

- `fx`: Probability density function of a distribution.
- `fn`: Probability mass function of a distribution.
- `logL`: Log likelihood function.
- `logL_1`: First derivative of log likelihood function.
- `logL_2`: Second derivative (namely Hessian matrix) of log likelihood function.
- `logL_2_Secant`: Using secant method to approximate `logL_2`.
- `logL_2_Fisher`: Using Fisher Information to approximate `logL_2`.
- `count_time`: Timer function, the result may be a bit different in each call.
- `compare_path`: Visualization function, to compare the iteration paths of different multivariate optimization methods.

List of optimization functions.

- `Newton_Raphson`: Newton-Raphson method (univariate & multivariate).
- `Bisection`: Bisection method (univariate).
- `Fixed_Point`: Fixed-Point iteration (univariate).
- `Secant`: Secant method (univariate).
- `Fisher_Scoring`: Fisher Scoring (multivariate).
- `Steepest_Ascent`: Steepest Ascent method (multivariate).
- `Quasi_Newton`: Quasi-Newton method (multivariate).

List of common parameters in functions.

- `start`: starting values.
- `itertime`: maximum iteration times, by default set as 100.
- `criterion`: absolute convergence criterion, by default set as $1e^{-6}$.
- `msg`: While TRUE (the default), return a message of optimization result. Otherwise, return a list containing iteration result `x`, corresponding `logL(x)`, starting value, iteration times, criterion and name of method used.
- `path`: While TRUE (default is FALSE), the returned list will be added a new object, which is an array of historical `x` in the optimization. This parameter is added to multivariate methods for visualization.

Notice that **two stopping rules** are applied in the following algorithms, **maximum iteration times** and **absolute convergence criterion** (Euclidean Norm).

## 2.1

### (a)

Import the sample points.

```
X <- 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 pdf of $Cauchy(\theta, 1)$.

$$f_X(x;\theta) = \frac{1}{\pi} * \frac{1}{(x-\theta)^2+1}$$

```
fx <- function(x, location){1 / pi * (1 / ((x - location) ^ 2 + 1))}
```

Define the log likelihood function.

$$\log L(\theta) = -n\log(\pi) - \sum_{i=1}^{n} \log[(x_i-\theta)^2+1]$$

```
logL <- function(theta){sum(log(fx(X, theta)))}
```

Graph the log likelihood curve.

```
y <- c()
for (i in seq(-50, 50, 0.1)){y <- c(y, logL(i))}
ggplot(mapping = aes(x = seq(-50, 50, 0.1), y = y)) + geom_line() +
  labs(title = 'Log Likelihood Curve', x = 'theta', y = 'LogL') + theme_light()
```

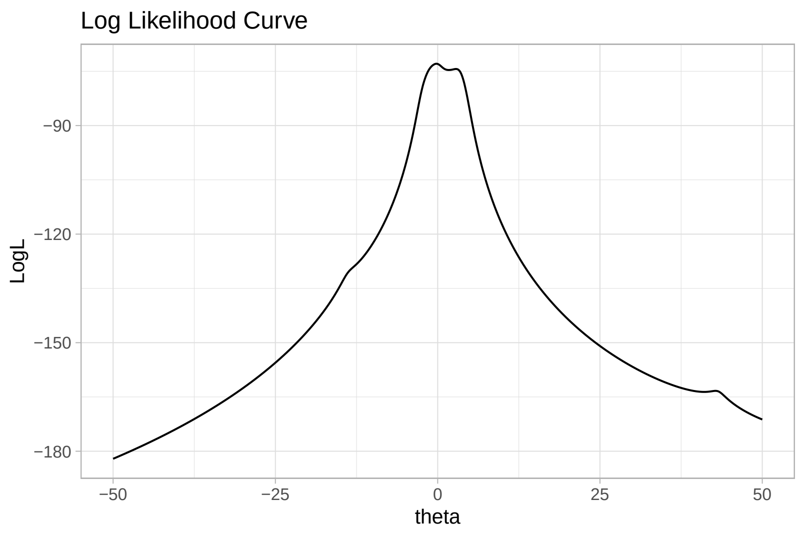

图 1: Log Likelihood Curve

Compute the derivative functions.

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

```
logL_1 <- Deriv(logL, 'theta')
logL_2 <- Deriv(logL_1, 'theta')
```

Define the **Newton-Raphson method** function (feasible for both **univariate** and **multivariate** optimization).

```
Newton_Raphson <- function(start, itertime = 100, criterion = 1e-6,
                           msg = TRUE, path = FALSE){
  tryCatch({

    ###INITIAL VALUES###
    x <- start; times <- 0; p <- start

    ###FUNCTIONS###
    if (length(start) != 1){
      delta <- function(x){solve(logL_2(x)) %*% logL_1(x)}
      y <- function(x){logL(x[1], x[2])}
    }
    else {
      delta <- function(x){logL_1(x) / logL_2(x)}
      y <- function(x){logL(x)}
    }
    epsilon <- function(x){sqrt(sum(x ^ 2))} ##L2 Norm

    ###MAIN###
    for (i in 1:itertime){ ##Stopping rule 1
      d <- delta(x)
      if (epsilon(d) > criterion){x <- x - d}
      else {break} ##Stopping rule 2
      times <- times + 1
      if (path){p <- rbind(p, t(x))}
    }

    ###OUTPUT###
    if (msg){
      print(paste0('Start: ', paste(start, collapse = ','), ', Iter_time: ',
                   times, ', Optimal: ', y(x)))}
    else{
      structure(list(x = x, xx = y(x), path = p, start = start, itertime = times,
                     criterion = criterion, method = 'Newton-Raphson Method'))}},
    error = function(e){
      if (msg){
        print(paste0('Start: ', paste(start, collapse = ','),
                     ', Hessian matrix is singular. Fail to converge.'))}
      else{
        structure(list(x = x, xx = y(x), path = p, start = start, itertime = times,
                       criterion = criterion, method = 'Newton-Raphson Method'))}
  })
}
```

Compute results.

```
startvalues <- c(-11, -1, 0, 1.5, 4, 4.7, 7, 8, 38)
for (i in 1:length(startvalues)){Newton_Raphson(startvalues[i])}
```

```
## [1] "Start: -11, Iter_time: 100, Optimal: -2868.88333620672"
## [1] "Start: -1, Iter_time: 3, Optimal: -72.9158196158461"
## [1] "Start: 0, Iter_time: 3, Optimal: -72.9158196158461"
## [1] "Start: 1.5, Iter_time: 4, Optimal: -74.6420164018362"
## [1] "Start: 4, Iter_time: 4, Optimal: -74.360461334214"
## [1] "Start: 4.7, Iter_time: 4, Optimal: -72.9158196158461"
## [1] "Start: 7, Iter_time: 8, Optimal: -163.607723832696"
## [1] "Start: 8, Iter_time: 100, Optimal: -2854.4604278752"
## [1] "Start: 38, Iter_time: 4, Optimal: -163.312886705311"
```

Use the mean of data as the starting point and compute the result.

```
Newton_Raphson(mean(startvalues))
```

```
## [1] "Start: 5.68888888888889, Iter_time: 7, Optimal: -173.334523408058"
```

The mean of the data is not a good starting point because its result falls to a minimal value of -173.33, which is far smaller than that of other starting points.

### (b)

Define the **Bisection method** function. $[a_0, b_0]$ (if $a_0 < b_0$) is the initial interval.

```r
Bisection <- function(a0, b0, itertime = 100, criterion = 1e-6, msg = TRUE){

  ###INITIAL VALUES###
  a <- a0; b <- b0; times <- 0

  if (logL_1(a) * logL_1(b) > 0){
    if (msg){print('Please select the starting point again.')}
    else{structure(list(x = NA, start = c(a0, b0), itertime = times,
                        criterion = criterion, method = 'Bisection Method'))}
  }
  else{

    ###MAIN###
    for (i in 1:itertime){
      if (abs((b - a) / 2) > criterion){
          x <- a + (b - a) / 2
          if (logL_1(a) * logL_1(x) > 0){a <- x}
          else {b <- x}
      }
      else {break}
      times <- times + 1
    }

    ###OUTPUT###
    if (msg){
      print(paste0('Start: (', a0, ', ', b0, '), Iter_time: ', times,
                   ', Optimal: ', logL(x)))}
    else{
      structure(list(x = x, xx = logL(x), start = c(a0, b0), itertime = times,
                     criterion = criterion, method = 'Bisection Method'))}
  }
}
```

Compute results.

```r
Bisection(-1, 1)
```

```
## [1] "Start: (-1, 1), Iter_time: 20, Optimal: -72.9158196158479"
```

There are three manners in which the bisection method may fail to find the global maximum. The first one is as follows: the initial interval contains more than one roots (two roots in this problem as shown in the last graph), but the algorithm can only find one of them which may not be the global maximum.

```
Bisection(-1, 5)
```

```
## [1] "Start: (-1, 5), Iter_time: 22, Optimal: -74.3604613342146"
```

The second one is as follows: the initial interval contains a root, but it is not the root of the global maximum, or even it is the root of a minimum.

```
Bisection(2, 3); Bisection(42, 43)
```

```
## [1] "Start: (2, 3), Iter_time: 19, Optimal: -74.360461334214"
```

```
## [1] "Start: (42, 43), Iter_time: 19, Optimal: -163.312886705311"
```

The third one is as follows: the first derivatives of $a_0$ and $b_0$ are both negative or positive.

```
Bisection(-2.5, 2.5)
```

```
## [1] "Please select the starting point again."
```

## (c)

Define the **Fixed-Point iteration** function. The **scaling factor** $\alpha$ is by default set as 1.

```
Fixed_Point <- function(start, alpha = 1, itertime = 100, criterion = 1e-6, msg = TRUE){

  ###INITIAL VALUES###
  x <- start; times <- 0

  ###FUNCTIONS###
  G <- function(x){alpha * logL_1(x) + x}

  ###MAIN###
  for (i in 1:itertime){
    if (is.na(logL_1(x))) {break}
    else if (abs(alpha * logL_1(x)) > criterion){x <- G(x)}
    else {break}
    times <- times + 1
  }
  ###OUTPUT###
  if (msg){
    print(paste0('Start: ', start, ', Alpha: ', alpha, ', Iter_time: ', times,
                 ', Optimal: ', logL(x)))}
  else{
    structure(list(x = x, xx = logL(x), start = start, alpha = alpha, itertime = times,
                   criterion = criterion, method = 'Fixed-Point Iteration'))}
}
```

Compute results.

```
Fixed_Point(-1, 1)
```

```
## [1] "Start: -1, Alpha: 1, Iter_time: 100, Optimal: -73.2303953802832"
```

```
Fixed_Point(-1, 0.64)
```

```
## [1] "Start: -1, Alpha: 0.64, Iter_time: 100, Optimal: -72.9159001375124"
```

```
Fixed_Point(-1, 0.25)
```

```
## [1] "Start: -1, Alpha: 0.25, Iter_time: 10, Optimal: -72.9158196158475"
```

Other choices of starting values and scaling factors.

```
Fixed_Point(0, 0.81); Fixed_Point(1, 0.49); Fixed_Point(1, 0.36); Fixed_Point(2, 0.16)
```

```
## [1] "Start: 0, Alpha: 0.81, Iter_time: 100, Optimal: -73.4160953436236"

## [1] "Start: 1, Alpha: 0.49, Iter_time: 20, Optimal: -72.9158196158466"

## [1] "Start: 1, Alpha: 0.36, Iter_time: 8, Optimal: -72.9158196158464"

## [1] "Start: 2, Alpha: 0.16, Iter_time: 39, Optimal: -74.3604613342207"
```

In this problem, it seems that a relatively small alpha is more suitable because it ensures the **Lipschitz condition** and thus enables the method to converge.

Define the visualization function of Fixed-Point iteration. The value of parameter `show` means which object to display on y axis among the return list of an optimization function.

```
fp_show <- function(starts, alphas, show){
  y <- c()
  if (length(starts) == 1) { ##Starting point fixed, different alphas
    for (alpha in alphas){y <- c(y, Fixed_Point(starts, alpha, msg = FALSE)[[show]])}
    ggplot(mapping = aes(x = alphas, y = y)) + geom_point(size = 0.001) +
      labs(subtitle = 'Fixed-Point Iteration', y = show,
              x = paste0('alpha (start = ', starts, ')')) + theme_light()}
  else { ##alpha fixed, different starting points
    for (start in starts){y <- c(y, Fixed_Point(start, alphas, msg = FALSE)[[show]])}
    ggplot(mapping = aes(x = starts, y = y)) + geom_point(size = 0.001) +
      labs(subtitle = 'Fixed-Point Iteration', y = show,
              x = paste0('start (alpha = ', alphas, ')')) + theme_light()}
}
```

For a set of fixed starting values, graph how the iteration times change with $\alpha$.

```
p1 = fp_show(0, seq(-5, 5, 0.01), 'itertime')
p2 = fp_show(0, seq(0.001, 0.7, 0.001), 'itertime')
p3 = fp_show(0, seq(0, 0.7, 0.001), 'x')
p4 = fp_show(-5, seq(0.001, 0.7, 0.001), 'itertime')
p5 = fp_show(-5, seq(0, 0.7, 0.001), 'x')
p6 = fp_show(1, seq(0.001, 0.7, 0.001), 'itertime')
p7 = fp_show(1, seq(0, 0.7, 0.001), 'x')
p8 = fp_show(5, seq(0.001, 0.7, 0.001), 'itertime')
p9 = fp_show(5, seq(0, 0.7, 0.001), 'x')
p1 + p2 + p3 + p4 + p5 + p6 + p7 + p8 + p9
```

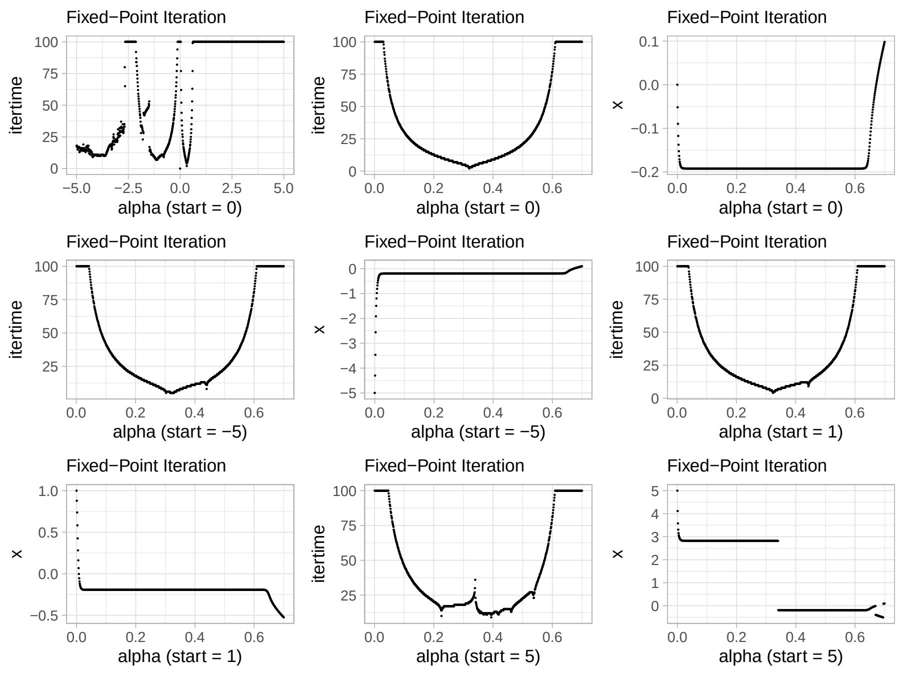

图 2: Sensitivity to Scaling Factor

It turns out that the algorithm can always converge for $\alpha$ between 0.4 and 0.6 (roughly) with iteration times less than 100, but the convergence speed varies. According to the graphs, the most suitable $\alpha$ in this problem may be about 0.33, and the convergence speed slows down with $\alpha$'s distance from 0.33.

For a set of fixed $\alpha$, graph how the iteration times change with starting values.

```
p1 = fp_show(seq(-5, 5, 0.05), 0.5, 'itertime')
p2 = fp_show(seq(-5, 5, 0.05), 0.4, 'itertime')
p3 = fp_show(seq(-5, 5, 0.05), 0.3, 'itertime')
p4 = fp_show(seq(-5, 5, 0.05), 0.5, 'x')
p5 = fp_show(seq(-5, 5, 0.05), 0.4, 'x')
p6 = fp_show(seq(-5, 5, 0.05), 0.3, 'x')
p1 + p2 + p3 + p4 + p5 + p6
```

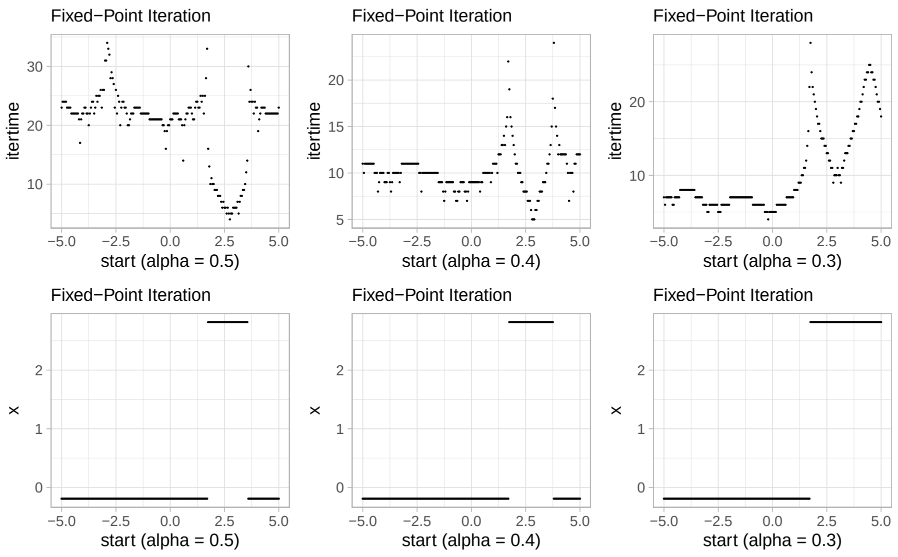

图 3: Sensitivity to Starting Values

It turns out that the algorithm can always converge for starting values less than 2 (roughly) with iteration times less than 100.

## (d)

Define the **Secant method** function. $(a_0, a_1)$ are the starting values.

```
Secant <- function(a0, a1, itertime = 100, criterion = 1e-6, msg = TRUE){

  ###INITIAL VALUES###
  x0 <- a0; x1 <- a1; times <- 0
```

```r
  ###FUNCTIONS###
  logL_2_Secant <- function(x1, x0){(logL_1(x1) - logL_1(x0)) / (x1 - x0)}

  ###MAIN###
  for (i in 1:itertime){
    if (abs(logL_1(x1) / logL_2_Secant(x1, x0)) > criterion){
      x <- x1 - logL_1(x1) / logL_2_Secant(x1, x0); x0 <- x1; x1 <- x}
    else {break}
    times <- times + 1
  }

  ###OUTPUT###
  if (msg){
    print(paste0('Start: (', a0, ', ', a1, '), Iter_time: ', times, ', Optimal: ', logL(x)))}
  else{
    structure(list(x = x, xx = logL(x), start = c(a0, a1), itertime = times,
                   criterion = criterion, method = 'Secant Method'))}
}
```

Compute results with starting values of (-2,-1).

```r
Secant(-2, -1)
```

```
## [1] "Start: (-2, -1), Iter_time: 5, Optimal: -72.9158196158461"
```

The result of Secant method is the same as that of Newton-Raphson method though this method contains an approximation to Hessian matrix.

Compute results with starting values of (-3,3).

```r
Secant(-3, 3)
```

```
## [1] "Start: (-3, 3), Iter_time: 6, Optimal: -74.3604613342139"
```

Compute results with other choices of starting values.

```r
Secant(-2, 1); Secant(-5, 5); Secant(1, 5)
```

```
## [1] "Start: (-2, 1), Iter_time: 8, Optimal: -74.6420164018362"

## [1] "Start: (-5, 5), Iter_time: 6, Optimal: -72.9158196158461"

## [1] "Start: (1, 5), Iter_time: 6, Optimal: -72.9158196158461"
```

But sometimes the result may fall to local optimum instead of global maximum.

### (e)

Define the timer function.

```r
count_time <- function(fun, msg = TRUE){
  start <- Sys.time(); fun; end <- Sys.time()
  if (msg){print(paste0(fun$method, ': ', end - start, 2, ' seconds'))}
  else{return (end - start)}
}
```

Compute the computing time with similar starting values and the same stopping rules.

```r
count_time(Newton_Raphson(0, msg = F))
count_time(Secant(-1, 0, msg = F))
count_time(Bisection(-1, 0, msg = F))
count_time(Fixed_Point(0, 0.33, msg = F))
count_time(Fixed_Point(0, 0.44, msg = F))
count_time(Fixed_Point(0, 0.55, msg = F))
```

```
## [1] "Newton-Raphson Method: 4.69684600830078e-052 seconds"
## [1] "Secant Method: 6.00814819335938e-052 seconds"
## [1] "Bisection Method: 7.51018524169922e-052 seconds"
## [1] "Fixed-Point Iteration: 3.50475311279297e-052 seconds"
## [1] "Fixed-Point Iteration: 7.51018524169922e-052 seconds"
## [1] "Fixed-Point Iteration: 0.0001730918884277342 seconds"
```

**Speed:** Newton_Raphson method > Secant method > Bisection method.

The convergence speed of Fixed-Point iteration is greatly dependent on the value of $\alpha$, which has been shown in question (c).

**Stability:** Use a set of starting values to test the stability.

Define the visualization function of Newton-Raphson method.

```r
nr_show <- function(starts, show){
  y <- c()
  for (start in starts){y <- c(y, Newton_Raphson(start, msg = FALSE)[[show]])}
  ggplot(mapping = aes(x = starts, y = y)) + geom_point(size = 0.001) +
    labs(subtitle = 'Newton-Raphson method', x = 'start', y = show) + theme_light()
}
```

Define the visualization function of Bisection method. Keep one starting value fixed, and change the other.

```
bs_show <- function(fix_start, starts, show){
  y <- c()
  for (start in starts){y <- c(y, Bisection(fix_start, start, msg = FALSE)[[show]])}
  ggplot(mapping = aes(x = starts, y = y)) + geom_point(size = 0.001) +
    labs(subtitle = 'Bisection method', x = 'start', y = show) + theme_light()
}
```

Define the visualization function of Secant method. Keep one starting value fixed, and change the other.

```
sc_show <- function(fix_start, starts, show){
  y <- c()
  for (start in starts){y <- c(y, Secant(fix_start, start, msg = FALSE)[[show]])}
  ggplot(mapping = aes(x = starts, y = y)) + geom_point(size = 0.001) +
    labs(subtitle = 'Secant method', x = 'start', y = show) + theme_light()
}
```

Compare the stability of four methods.

```
p1 <- nr_show(seq(-10, 10, 0.01), 'x')
p2 <- nr_show(seq(-10, 10, 0.01), 'x') + ylim(-0.5, 3)
p3 <- bs_show(-10, seq(-10, 10, 0.01), 'x') + ylim(-0.5, 3)
p4 <- sc_show(-10, seq(-10+0.01, 10, 0.01), 'x')
p5 <- sc_show(-10, seq(-10+0.01, 10, 0.01), 'x') + ylim(-0.5, 3)
p6 <- fp_show(seq(-10, 10, 0.01), 0.33, 'x') + ylim(-0.5, 3)
p1 + p2 + p3 + p4 + p5 + p6
```

If only one starting point is changeable and the other parameters are fixed, then Fixed-Point iteration and Bisection method are the most stable (as long as the former is given a great $\alpha$ and the latter is given great starting values whose product is negative). Newton-Raphson method and Secant method are less stable because the ending iteration points are greatly fluctuating.

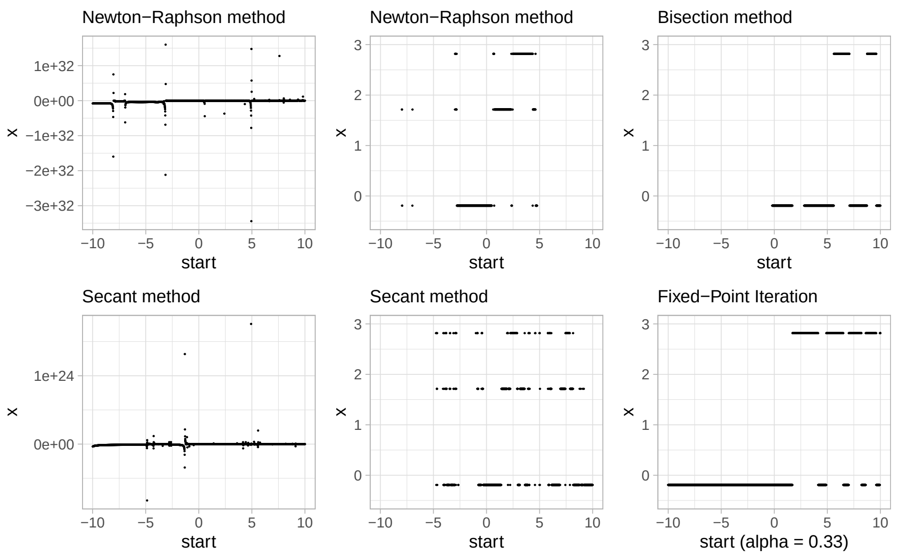

图 4: Stability of Different Methods

**Apply the methods to a random sample of size 20 from a $N(\theta, 1)$ distribution.**

Generate random sample.

```r
set.seed(2022)
X <- rnorm(20, runif(1), 1)
```

Define the pdf of $N(\theta, 1)$.

$$f_X(x;\theta) = \frac{1}{\sqrt{2\pi}}\exp(-\frac{(x-\theta)^2}{2})$$

```r
fx <- function(x, mu){1 / sqrt(2 * pi) * exp(- (x - mu) ^ 2 / 2)}
```

Define the log likelihood function.

$$\log L(\theta) = -\sum_{i=1}^{n}\frac{(x_i-\theta)^2}{2} - \frac{n}{2}\log(2\pi)$$

```r
logL <- function(theta){sum(log(fx(X, theta)))}
```

Graph the log likelihood curve.

```r
y <- c()
for (i in seq(-20, 20, 0.1)){y <- c(y, logL(i))}
ggplot(mapping = aes(x = seq(-20, 20, 0.1), y = y)) + geom_line() +
  labs(title = 'Log Likelihood Curve', x = 'theta', y = 'LogL') + theme_light()
```

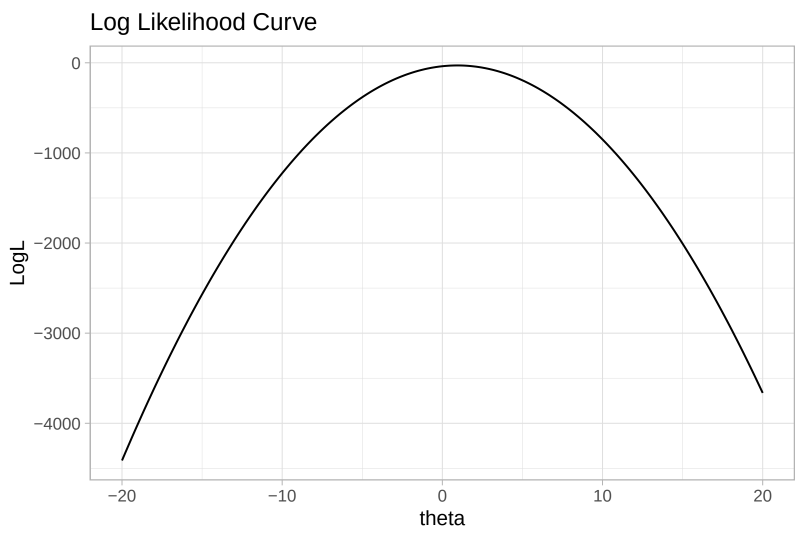

图 5: Log Likelihood Curve

Compute the derivative functions.

$$\log L'(\theta) = \sum_{i=1}^{n}(x_i - \theta) \log L''(\theta) = -n$$

```
logL_1 <- Deriv(logL, 'theta') # first derivative
logL_2 <- Deriv(logL_1, 'theta') # second derivative
```

### Newton-Raphson method

```
Newton_Raphson(-10); Newton_Raphson(0)
```

```
## [1] "Start: -10, Iter_time: 1, Optimal: -29.3317106554455"

## [1] "Start: 0, Iter_time: 1, Optimal: -29.3317106554455"
```

### Bisection method

```
Bisection(-1, 1); Bisection(-5, 5)
```

```
## [1] "Start: (-1, 1), Iter_time: 20, Optimal: -29.3317106554813"

## [1] "Start: (-5, 5), Iter_time: 23, Optimal: -29.3317106554594"
```

### Fixed-Point iteration

```
Fixed_Point(0, 1); Fixed_Point(0, 0.06)
```

```
## [1] "Start: 0, Alpha: 1, Iter_time: 2, Optimal: -Inf"

## [1] "Start: 0, Alpha: 0.06, Iter_time: 9, Optimal: -29.3317106554478"
```

The convergence of this method is dependent on $\alpha$. Graph the iteration curve with different $\alpha$.

```r
p1 <- fp_show(-10, seq(-5, 5, 0.01), 'x')
p2 <- fp_show(-10, seq(-0.3, 0.3, 0.001), 'x')
p3 <- fp_show(0, seq(-5, 5, 0.01), 'x')
p4 <- fp_show(0, seq(-0.3, 0.3, 0.001), 'x')
p1 + p2 + p3 + p4
```

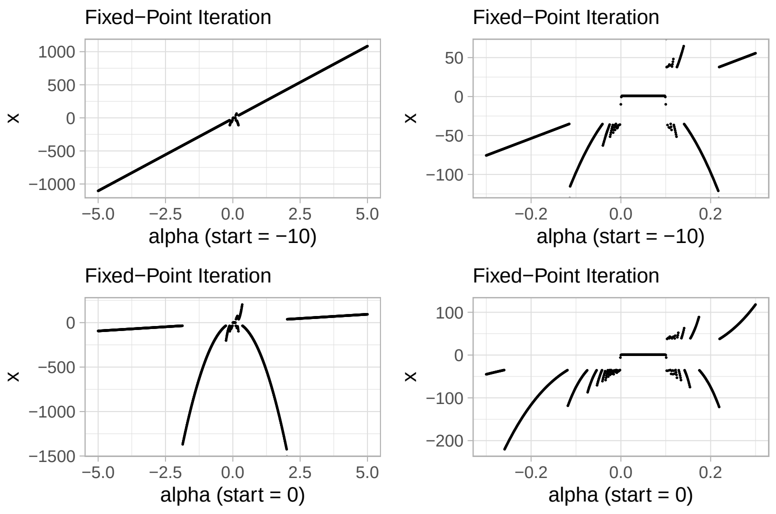

图 6: Fixed-Point Iteration Curve

In this problem, only when $\alpha \in (0, 0.1)$ (roughly) will Fixed-Point iteration converges to the global maximum.

### Secant method

```r
Secant(-1, 1); Secant(-5, 5)
```

```
## [1] "Start: (-1, 1), Iter_time: 1, Optimal: -29.3317106554455"

## [1] "Start: (-5, 5), Iter_time: 1, Optimal: -29.3317106554455"
```

### Speed

```r
count_time(Secant(0, 1, msg = F))
count_time(Newton_Raphson(0, msg = F))
count_time(Bisection(0, 1, msg = F))
count_time(Fixed_Point(0, 0.03, msg = F))
count_time(Fixed_Point(0, 0.05, msg = F))
```

```
## [1] "Secant Method: 5.00679016113281e-052 seconds"
## [1] "Newton-Raphson Method: 4.60147857666016e-052 seconds"
## [1] "Bisection Method: 0.000138044357299805 seconds"
## [1] "Fixed-Point Iteration: 0.00016093254089355 seconds"
## [1] "Fixed-Point Iteration: 3.31401824951172e-05 seconds"
```

The convergence speed is a little different from the previous conclusion. Secant method is the fastest, Newton-Raphson method is the second, and then the Bisection method. The convergence speed of Fixed-Point iteration still greatly depends on the value of $\alpha$.

### Stability

```
p1 <- nr_show(seq(-10, 10, 0.01), 'x') + ylim(0.93,0.94)
p2 <- fp_show(seq(-10, 10, 0.01), 0.05, 'x') + ylim(0.93,0.94)
p4 <- fp_show(seq(-10, 10, 0.01), 0.03, 'x') + ylim(0.93,0.94)
p3 <- bs_show(-10, seq(-10, 10, 0.01), 'x')
p6 <- sc_show(-10, seq(-10+0.01, 10, 0.01), 'x')
p5 <- sc_show(-10, seq(-10+0.01, 10, 0.01), 'x') + ylim(0.93,0.94)
p1 + p2 + p3 + p4 + p5 + p6
```

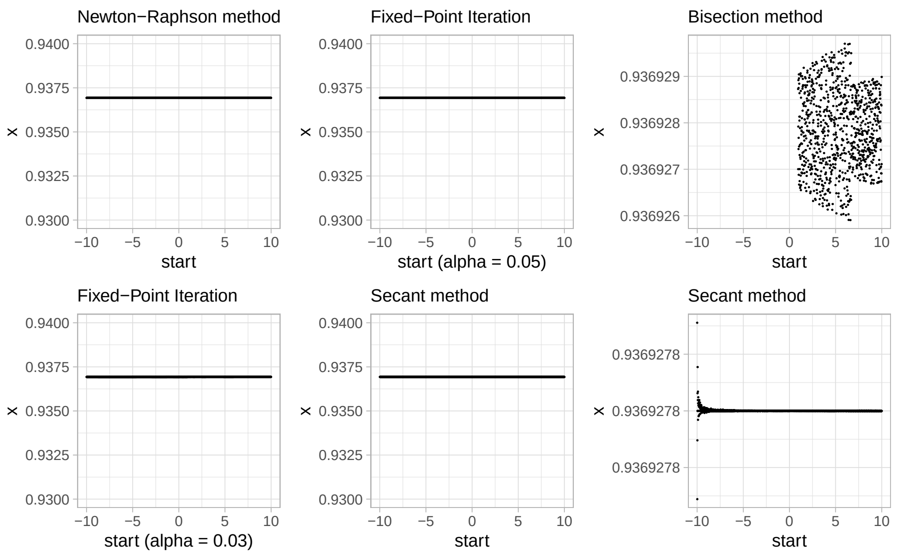

图 7: Stability of Different Methods

The stability is a little different from the previous conclusion. If only one starting point is changeable and the other parameters are fixed, Newton-Raphson method and Fixed-Point iteration are the most stable (if the latter is given a great $\alpha$). Secant method is also stable if the changeable starting value larger than -7.5 (roughly in this example), more stable than Bisection method because its precision is higher (concluded from the y axis in graphs).

## 2.2

### (a)

Import the sample points.

```
X <- 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, 2.46, 2.99, 2.54, 0.52, 2.50)
```

Define the pdf.

$$f_X(x;\theta) = \frac{1-\cos(x-\theta)}{2\pi}$$

```
fx <- function(x, theta){(1 - cos(x - theta)) / (2 * pi)}
```

Define the log likelihood function.

$$\log L(\theta) = \sum_{i=1}^{n} \log(1-\cos(x_i-\theta)) - n\log(2\pi)$$

```
logL <- function(theta){sum(log(fx(X, theta)))}
```

Graph the log likelihood curve.

```
y <- c()
for (i in seq(-pi, pi, 0.01)){y <- c(y, logL(i))}
ggplot(mapping = aes(x = seq(-pi, pi, 0.01), y = y)) + geom_line() +
  labs(title = 'Log Likelihood Curve', x = 'theta', y = 'LogL') + theme_light()
```

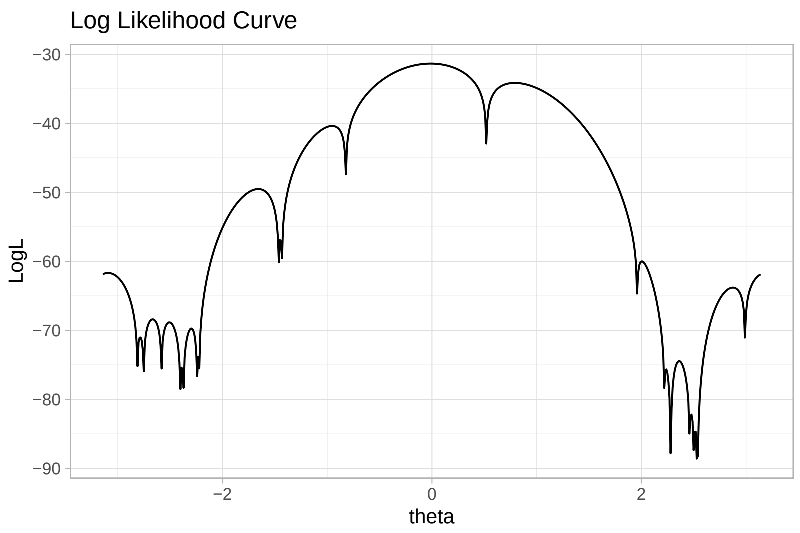

图 8: Log Likelihood Curve

Compute the derivative functions.

$$\log L'(\theta) = -\sum_{i=1}^{n} \frac{\sin(x_i - \theta)}{1 - \cos(x_i - \theta)} \log L''(\theta) = -\sum_{i=1}^{n} (\frac{\sin(x_i - \theta)^2}{(1 - \cos(x_i - \theta))^2} - \frac{\cos(x_i - \theta)}{1 - \cos(x_i - \theta)})$$

```r
logL_1 <- Deriv(logL, 'theta')
logL_2 <- Deriv(logL_1, 'theta')
```

## (d)

Based on the last log likelihood curve, add a scatter plot of logL (after Newton-Raphson method optimization) with 200 equally spaced starting values between $-\pi$ and $\pi$.

```r
nr_show(seq(-pi, pi, pi / 100), 'xx') +
  geom_line(mapping = aes(x = seq(-pi, pi, 0.01), y = y, colour = 'red')) +
  guides(colour = 'none') + labs(x = 'theta', y = 'LogL')
```

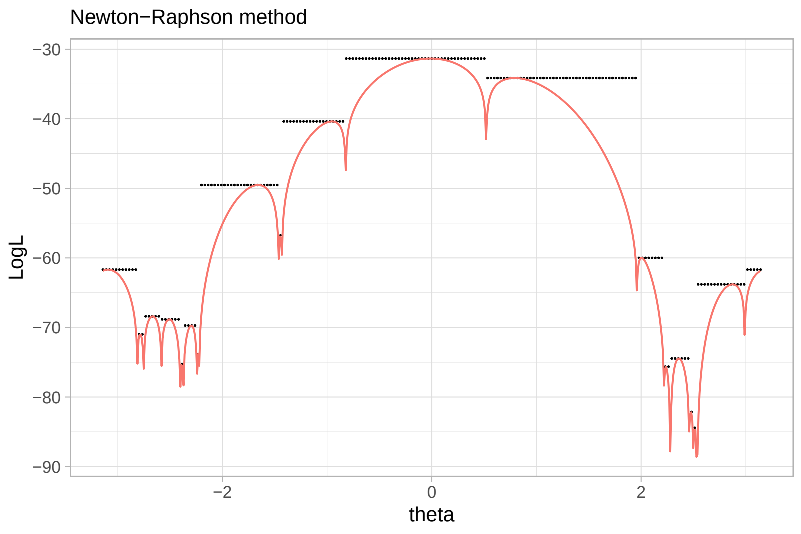

图 9: Log Likelihood Curve

As shown in the graph above, if the starting values are divided into several separate groups, then each group corresponds to the xaxis range (namely range of $\theta$) of a complete hill in the log likelihood function curve, and the optimization outcome of each group corresponds to the peak of that hill.

## 2.5

### Preparation

Load the data `oilspills.dat`.

```r
data <- read.table('oilspills.dat', header = TRUE)
```

Define $N_i, b_1 = (b_{11}, b_{21}, ..., b_{i1}), b_2 = (b_{12}, b_{22}, ..., b_{i2})$.

```r
n <- data$spills
b1 <- data$importexport
b2 <- data$domestic
```

Define the pmf of $Poisson(\lambda_i)$ distribution. $\lambda_i = \alpha_1 b_{i1} + \alpha_2 b_{i2}$.

$$P(X = n_i; \alpha_1, \alpha_2) = \frac{(\alpha_1 b_{i1} + \alpha_2 b_{i2})^{n_i}}{n_i!} \exp\{-(\alpha_1 b_{i1} + \alpha_2 b_{i2})\}$$

```r
fn <- function(n, b1, b2, alpha1, alpha2){
  lambda <- alpha1 * b1 + alpha2 * b2
  exp(- lambda) * lambda ^ n / factorial(n)
}
```

Define the log likelihood function.

$$\log L(\alpha_1, \alpha_2) = \sum_{i=1}^{n} n_i \log(\alpha_1 b_{i1} + \alpha_2 b_{i2}) - \sum_{i=1}^{n} (\log(n_i!) + \alpha_1 b_{i1} + \alpha_2 b_{i2})$$

```r
logL <- function(alpha1, alpha2){sum(log(fn(n, b1, b2, alpha1, alpha2)))}
```

### (a)

Compute the first derivative function.

$$\frac{d \log L}{d\alpha_1} = \sum_{i=1}^{n} b_{i1}(\frac{n_i}{\alpha_1 b_{i1} + \alpha_2 b_{i2}} - 1)$$
$$\frac{d \log L}{d\alpha_2} = \sum_{i=1}^{n} b_{i2}(\frac{n_i}{\alpha_1 b_{i1} + \alpha_2 b_{i2}} - 1)$$
$$\log L'(\alpha_1, \alpha_2) = (\frac{d \log L}{d\alpha_1}, \frac{d \log L}{d\alpha_2})^\top$$

```r
dyda1 <- Deriv(logL, 'alpha1')
dyda2 <- Deriv(logL, 'alpha2')
logL_1 <- function(alpha){matrix(c(dyda1(alpha[1], alpha[2]),
                                   dyda2(alpha[1], alpha[2])), ncol = 1)}
```

Compute the Hessian matrix function.

$$
\begin{aligned}
\frac{d^2 \log L}{d{\alpha_1}^2} &= -\sum_{i=1}^{n} {b_{i1}}^2 \frac{n_i}{(\alpha_1 b_{i1} + \alpha_2 b_{i2})^2} \\
\frac{d^2 \log L}{d\alpha_1 d\alpha_2} &= -\sum_{i=1}^{n} b_{i1} b_{i2} \frac{n_i}{(\alpha_1 b_{i1} + \alpha_2 b_{i2})^2} \\
\frac{d^2 \log L}{d{\alpha_2}^2} &= -\sum_{i=1}^{n} {b_{i2}}^2 \frac{n_i}{(\alpha_1 b_{i1} + \alpha_2 b_{i2})^2} \\
\log L''(\alpha_1, \alpha_2) &= \begin{bmatrix} \frac{d^2 \log L}{d{\alpha_1}^2} & \frac{d^2 \log L}{d\alpha_1 d\alpha_2} \\ \frac{d^2 \log L}{d\alpha_1 d\alpha_2} & \frac{d^2 \log L}{d{\alpha_2}^2} \end{bmatrix}
\end{aligned}
$$

```r
d2yda1a1 <- Deriv(dyda1, 'alpha1')
d2yda1a2 <- Deriv(dyda1, 'alpha2')
d2yda2a2 <- Deriv(dyda2, 'alpha2')
logL_2 <- function(alpha){matrix(c(d2yda1a1(alpha[1], alpha[2]),
                                   d2yda1a2(alpha[1], alpha[2]),
                                   d2yda1a2(alpha[1], alpha[2]),
                                   d2yda2a2(alpha[1], alpha[2])), ncol = 2)}
```

The Newton-Raphson update for finding the MLEs of $\alpha_1$ and $\alpha_2$ is as follows.

$$
\alpha = (\alpha_1, \alpha_2) \alpha^{(t+1)} = \alpha^{(t)} - [\log L''(\alpha)]^{-1} \log L'(\alpha)
$$

## (b)

Calculate Fisher Information.

$$
\begin{aligned}
I(\alpha) =& E\{\log L'(\alpha)(\log L'(\alpha))^T\} \\
=& E\{(\frac{d \log L}{d\alpha_1}, \frac{d \log L}{d\alpha_2})^T(\frac{d \log L}{d\alpha_1}, \frac{d \log L}{d\alpha_2})\} \\
=& E \begin{bmatrix} (\frac{d \log L}{d\alpha_1})^2 & (\frac{d \log L}{d\alpha_1})(\frac{d \log L}{d\alpha_2}) \\ (\frac{d \log L}{d\alpha_2})(\frac{d \log L}{d\alpha_1}) & (\frac{d \log L}{d\alpha_2})^2 \end{bmatrix} \\
=& \begin{bmatrix} E\{(\frac{d \log L}{d\alpha_1})^2\} & E\{(\frac{d \log L}{d\alpha_1})(\frac{d \log L}{d\alpha_2})\} \\ E\{(\frac{d \log L}{d\alpha_2})(\frac{d \log L}{d\alpha_1})\} & E\{(\frac{d \log L}{d\alpha_2})^2\} \end{bmatrix}
\end{aligned}
$$

Compute the expectation respectively (Using the independence of $N_i$ and the properties of $Poisson(\lambda_i)$ distribution).

$$\begin{aligned}
E\{(\frac{d\log L}{d\alpha_1})^2\} =& E\{(\sum_{i=1}^{n} b_{i1}(\frac{N_i}{\lambda_i}-1))^2\} \\
& (Independence \quad of \quad \{N_i\}) \\
=& \sum_{i=1}^{n} E\{(b_{i1}(\frac{N_i}{\lambda_i}-1))^2\} \\
=& \sum_{i=1}^{n} {b_{i1}}^2 E\{\frac{{N_i}^2}{{\lambda_i}^2}-\frac{2N_i}{\lambda_i}+1\} \\
& (E(N_i)=\lambda_i, E({N_i}^2)=Var(N_i)+E^2(N_i)=\lambda_i+{\lambda_i}^2) \\
=& \sum_{i=1}^{n} {b_{i1}}^2 (\frac{\lambda_i+{\lambda_i}^2}{{\lambda_i}^2}-\frac{2\lambda_i}{\lambda_i}+1) \\
=& \sum_{i=1}^{n} \frac{{b_{i1}}^2}{\lambda_i}
\end{aligned}$$

Similarly, we can calculate $E\{(\frac{d\log L}{d\alpha_1})(\frac{d\log L}{d\alpha_2})\}$ and $E\{(\frac{d\log L}{d\alpha_2})^2\}$.

$$E\{(\frac{d\log L}{d\alpha_1})(\frac{d\log L}{d\alpha_2})\} = \sum_{i=1}^{n} \frac{b_{i1}b_{i2}}{\lambda_i} E\{(\frac{d\log L}{d\alpha_2})^2\} = \sum_{i=1}^{n} \frac{{b_{i2}}^2}{\lambda_i}$$

So the expression of Fisher Information is as follows.

$$I(\alpha)=I(\alpha_1,\alpha_2)=\begin{bmatrix} \sum_{i=1}^{n} \frac{{b_{i1}}^2}{\alpha_1 b_{i1}+\alpha_2 b_{i2}} & \sum_{i=1}^{n} \frac{b_{i1}b_{i2}}{\alpha_1 b_{i1}+\alpha_2 b_{i2}} \\ \sum_{i=1}^{n} \frac{b_{i2}b_{i1}}{\alpha_1 b_{i1}+\alpha_2 b_{i2}} & \sum_{i=1}^{n} \frac{{b_{i2}}^2}{\alpha_1 b_{i1}+\alpha_2 b_{i2}} \end{bmatrix}$$

Define the **Fisher Information** function.

```
logL_2_Fisher <- function(alpha){
  lambda <- alpha[1] * b1 + alpha[2] * b2
  fisher11 <- sum(b1 ^ 2 / lambda)
  fisher12 <- sum(b1 * b2 / lambda)
  fisher22 <- sum(b2 ^ 2 / lambda)
  out <- matrix(c(fisher11, fisher12, fisher12, fisher22), ncol = 2)
  return (out)
}
```

The Fisher Scoring update for finding the MLEs of $\alpha_1$ and $\alpha_2$ is as follows.

$$\alpha=(\alpha_1,\alpha_2)\alpha^{(t+1)}=\alpha^{(t)}+[I(\alpha)]^{-1}\log L'(\alpha)$$

### (c)

Define the **Fisher Scoring** function.

```
Fisher_Scoring <- function(start, itertime = 100, criterion = 1e-6,
                           msg = TRUE, path = FALSE){

  ###INITIAL VALUES###
  x <- start; times <- 0; p <- start

  ###FUNCTIONS###
  epsilon <- function(x){sqrt(sum(x ^ 2))}

  ###MAIN###
  for (i in 1:itertime){
    delta <- solve(- logL_2_Fisher(x)) %*% logL_1(x)
    if (epsilon(delta) > criterion){x <- x - delta}
    else {break}
    times <- times + 1
    if (path){p <- rbind(p, t(x))}
  }

  ###OUTPUT###
  if (msg){
    print(paste0('Start: (', paste(start, collapse = ','), '), Iter_time: ', times,
                 ', Optimal: ', logL(x[1], x[2])))}
  else{
    structure(list(x = x, xx = logL(x[1], x[2]), path = p, start = start, itertime = times,
                   criterion = criterion, method = 'Fisher Scoring Method'))}
}
```

Compute results.

```
Newton_Raphson(c(1, 1)); Newton_Raphson(c(10, 10))
```

```
## [1] "Start: 1,1, Iter_time: 3, Optimal: -48.0271622547274"
```

```
## [1] "Start: 10,10, Hessian matrix is singular. Fail to converge."
```

```
Fisher_Scoring(c(1, 1)); Fisher_Scoring(c(10, 10))
```

```
## [1] "Start: (1,1), Iter_time: 11, Optimal: -48.0271622547275"
## [1] "Start: (10,10), Iter_time: 11, Optimal: -48.0271622547275"
```

**Speed**

```r
count_time(Newton_Raphson(c(1, 1), msg = FALSE))
count_time(Fisher_Scoring(c(1, 1), msg = FALSE))
```

```
## [1] "Newton-Raphson Method: 0.000322103500366211 seconds"
## [1] "Fisher Scoring Method: 0.000365972518920898 seconds"
```

Compare the results of two methods. The convergence speed of Newton-Raphson method is faster than that of Fisher Scoring, and also the precision of Newton-Raphson method is higher since Fisher Scoring uses Fisher Information to estimate Hessian matrix.

**Stability**

However, it is sometimes hard for Newton-Raphson method to be implemented because it will fail to converge if the Hessian matrix turns singular, and thus Fisher Scoring is more stable (concluded from the results above with starting values of (10, 10)).

## (d)

There are two common ways to estimate standard errors for the MLEs of $\alpha_1$ and $\alpha_2$. The first one is to use $-[logL''(\hat{\alpha})]^{-1}$, in which $\hat{\alpha}$ is obtained by Newton-Raphson method.

```r
mle_nr <- Newton_Raphson(c(1, 1), msg = FALSE)$x
cov_nr <- -solve(logL_2(mle_nr))
print(paste0('(', sqrt(cov_nr[1,1]), ', ', sqrt(cov_nr[2,2]), ')'))
```

```
## [1] "(0.389612939541137, 0.554675998145071)"
```

The second one is to use $[I(\hat{\alpha})]^{-1}$, in which $\hat{\alpha}$ is obtained by Fisher Scoring.

```r
mle_fs <- Fisher_Scoring(c(1, 1), msg = FALSE)$x
cov_fs <- solve(logL_2_Fisher(mle_fs))
print(paste('(', sqrt(cov_fs[1,1]), ', ', sqrt(cov_fs[2,2]), ')', sep=''))
```

```
## [1] "(0.437555982636431, 0.631468697181668)"
```

The standard errors estimated by Fisher Information are larger than Hessian matrix because the former is an approximation to the latter.

## (e)

Define the **Steepest Ascent method** function. The starting value of the step length parameter `step` is 1.

```
Steepest_Ascent <- function(start, itertime = 100, criterion = 1e-6,
                            msg = TRUE, path = FALSE){

  ###INITIAL VALUES###
  x <- start; M <- diag(-1, 2, 2); step <- 1; times <- 0; p <- start

  ###FUNCTIONS###
  epsilon <- function(x){sqrt(sum(x ^ 2))}

  ###MAIN###
  for (i in 1:itertime){
    delta <- step * solve(M) %*% logL_1(x)

    ##BACKTRACKING
    temp <- logL((x - delta)[1], (x - delta)[2]) - logL(x[1], x[2])
    while (temp < 0 | is.na(temp)){
      step <- step / 2
      delta <- step * solve(M) %*% logL_1(x)
      temp <- logL((x - delta)[1], (x - delta)[2]) - logL(x[1], x[2])
    }

    if (epsilon(delta) > criterion){x <- x - delta}
    else {break}
    times <- times + 1
    if (path){p <- rbind(p, t(x))}
  }

  ###OUTPUT###
  if (msg){
    print(paste0('Start: (', paste(start, collapse = ','), '), Iter_time: ', times,
                 ', Optimal: ', logL(x[1], x[2])))}
  else{
    structure(list(x = x, xx = logL(x[1], x[2]), path = p, start = start, itertime = times,
                   criterion = criterion, method = 'Steepest Ascent Method'))}
}
```

Compute results.

```
Steepest_Ascent(c(1, 1)); Steepest_Ascent(c(10, 10))
```

```
## [1] "Start: (1,1), Iter_time: 60, Optimal: -48.0271622547676"
```

```
## [1] "Start: (10,10), Iter_time: 83, Optimal: -48.0271622547758"
```

The Steepest Ascent method takes more steps to converge.

### (f)

Define the **Quasi-Newton method** function. `M0` is the starting matrix. `half` is a logical indicating whether step halving is applied. The starting value of the step length parameter `step` is 1.

```
Quasi_Newton <- function(start, M0, half = TRUE, itertime = 100, criterion = 1e-6,
                         msg = TRUE, path = FALSE){

  ###INITIAL VALUES###
  x <- start; M <- M0; step <- 1; times <- 0; p <- start

  ###FUNCTIONS###
  delta <- function(x, M){step * solve(M) %*% logL_1(x)}
  epsilon <- function(x){sqrt(sum(x ^ 2))}
  generate_M <- function(x, M){
    z <- - delta(x, M)
    y <- logL_1(x + z) - logL_1(x)
    v <- y - M %*% z
    a <- t(v) %*% z; c <- 1 / a
    if (abs(a) < 1e-6 | is.na(a)){out <- M}
    else {out <- M + c[1] * v %*% t(v)}
    return (out)
  }

  ###MAIN###
  for (i in 1:itertime){
    M <- generate_M(x, M)
    d <- delta(x, M)

    ##BACKTRACKING
    if (half){
      temp <- logL((x - d)[1], (x - d)[2]) - logL(x[1], x[2])
      while (temp < 0 | is.na(temp)){
        step <- step / 2
        d <- delta(x, M)
        temp <- logL((x - d)[1], (x - d)[2]) - logL(x[1], x[2])
      }
    }

    if (epsilon(d) > criterion & is.na(sum(d)) == FALSE){x <- x - d}
    else {break}
    times <- times + 1
    if (path){p <- rbind(p, t(x))}
  }

  ###OUTPUT###
  if (msg){
    print(paste0('Start: (', paste(start, collapse = ','), ')',
                 ifelse(half, ', Step halving', ''),
                 ', Iter_time: ', times, ', Optimal: ', logL(x[1], x[2])))}
  else{
    structure(list(x = x, xx = logL(x[1], x[2]), path = p, start = start,
                   itertime = times, criterion = criterion, method = paste0(
                     ifelse(half, '', 'Non-'), 'Step-Halving Quasi-Newton Method')))}
}
```

Compute results given $M_0 = -I$.

```
Quasi_Newton(c(1, 1), diag(-1, 2, 2))
Quasi_Newton(c(1, 1), diag(-1, 2, 2), half = FALSE)
Quasi_Newton(c(0.8, 1), diag(-1, 2, 2))
Quasi_Newton(c(0.8, 1), diag(-1, 2, 2), half = FALSE)
Quasi_Newton(c(0.5, 1), diag(-1, 2, 2))
Quasi_Newton(c(0.5, 1), diag(-1, 2, 2), half = FALSE)
```

```
## [1] "Start: (1,1), Step halving, Iter_time: 16, Optimal: -48.0271622547289"
## [1] "Start: (1,1), Iter_time: 4, Optimal: -48.0271622547274"
## [1] "Start: (0.8,1), Step halving, Iter_time: 35, Optimal: -48.0271622548219"
## [1] "Start: (0.8,1), Iter_time: 4, Optimal: NaN"
## [1] "Start: (0.5,1), Step halving, Iter_time: 78, Optimal: -48.0271622552977"
## [1] "Start: (0.5,1), Iter_time: 4, Optimal: NaN"
```

When both methods (with or without step halving) converge, the convergence speed without step halving is much faster than that with step halving. However, the method without step halving is not stable because it cannot adjust the step length to ensure ascent. If the starting value changes slightly, it may not converge. By contrast, the method with step halving is very stable.

Compute results given $M_0 = -I(\theta_0)$.

```
Quasi_Newton(c(1, 1), -logL_2_Fisher(c(1, 1)))
Quasi_Newton(c(1, 1), -logL_2_Fisher(c(1, 1)), half = FALSE)
Quasi_Newton(c(0.5, 1), -logL_2_Fisher(c(0.5, 1)))
Quasi_Newton(c(0.5, 1), -logL_2_Fisher(c(0.5, 1)), half = FALSE)
Quasi_Newton(c(10, 1), -logL_2_Fisher(c(10, 1)))
Quasi_Newton(c(10, 1), -logL_2_Fisher(c(10, 1)), half = FALSE)
```

```
## [1] "Start: (1,1), Step halving, Iter_time: 4, Optimal: -48.0271622547275"
## [1] "Start: (1,1), Iter_time: 4, Optimal: -48.0271622547275"
## [1] "Start: (0.5,1), Step halving, Iter_time: 4, Optimal: -48.0271622547274"
## [1] "Start: (0.5,1), Iter_time: 4, Optimal: -48.0271622547274"
## [1] "Start: (10,1), Step halving, Iter_time: 79, Optimal: -48.0271622553112"
## [1] "Start: (10,1), Iter_time: 6, Optimal: NaN"
```

$M_0 = -I(\theta_0)$ is a very good starting matrix! The method with step halving converges faster than before. Besides, the one without step halving becomes more stable. Nevertheless, it is still less stable than the one with step halving.

Certainly we can also change the starting value of step length parameter `step` to make the non step halving method more stable, but the conclusion that it is less stable than step halving method won't be changed. That's the advantage of backtracking!

### (g)

Define the visualization function of multivariate optimization paths.

```
###BASE###
show_map <- function(min, max, bins){
  step <- (max - min) / 100
  rg <- seq(min + step, max, step)
  a1 <- c(); a2 <- c(); l <- c()
  for (i in 1:length(rg)){
    for (j in 1:length(rg)){
      a1 <- c(a1, rg[i]); a2 <- c(a2, rg[j]); l <- c(l, logL(rg[i], rg[j]))}}
  ggplot() + geom_tile(aes(x = a1, y = a2, fill = l)) +
    scale_fill_gradientn(colours = colorRampPalette(rev(brewer.pal(9, 'Blues')))(256)) +
    geom_contour(aes(x = a1, y = a2, z = l), bins = bins, color = 'black', size = 0.2) +
    labs(title = 'Contour Map', x='alpha1', y='alpha2', fill='LogL') +
    guides(fill = 'none') + theme_light()
}

###ADDITION###
##Add paths
show_line <- function(fun){
  geom_path(aes(x = fun$path[,1], y = fun$path[,2]), color = 'black')}
##Add points
show_point <- function(fun){
  geom_point(aes(x = fun$path[,1], y = fun$path[,2]), color = 'black', size = 1.1)}
##Add starting and ending points
show_sted <- function(fun){
  geom_point(aes(x = c(fun$start[1], fun$x[1]), y = c(fun$start[2], fun$x[2])),
             color = 'red')}
```

Define the visualization function of comparing different optimization methods.

```
compare_path <- function(start, min, max, bins){
  p0 <- show_map(min, max, bins); p <- list(); out <- p0
  fun <- list(Newton_Raphson(start, msg = F, path = T),
              Fisher_Scoring(start, msg = F, path = T),
              Steepest_Ascent(start, msg = F, path = T),
              Quasi_Newton(start, -logL_2_Fisher(c(1, 1)), msg = F, path = T),
              Quasi_Newton(start, -logL_2_Fisher(c(1, 1)), half = F, msg = F, path = T))
  for (i in 1:5){
    p[[i]] <- p0 + show_line(fun[[i]]) + show_point(fun[[i]]) +
      show_sted(fun[[i]]) + labs(title = fun[[i]]$method)
    out <- out + p[[i]]
    ggsave('path.png', p[[i]])
  }
  return (out)
}
```

Graph the paths with the following given starting values.

```
compare_path(c(0.1, 0.1), 0, 4.5, 25)
```

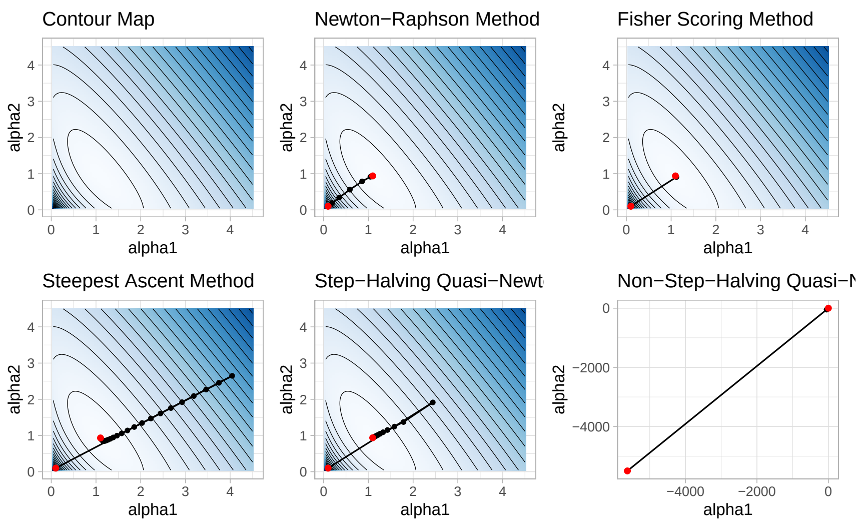

图 10: Iteration Path Starting with (0.1, 0.1)

```
compare_path(c(0.5, 0.5), 0, 1.5, 25)
```

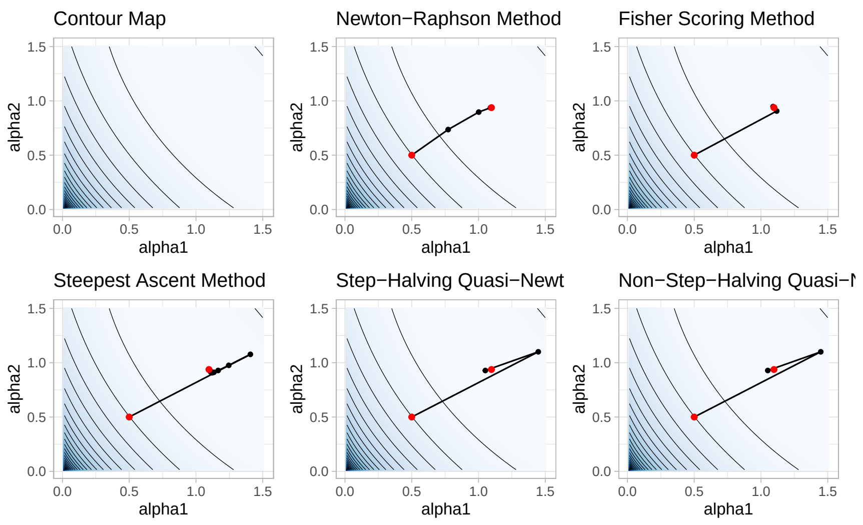

图 11: Iteration Path Starting with (0.5, 0.5)

```r
compare_path(c(1, 1), 0.85, 1.15, 25)
```

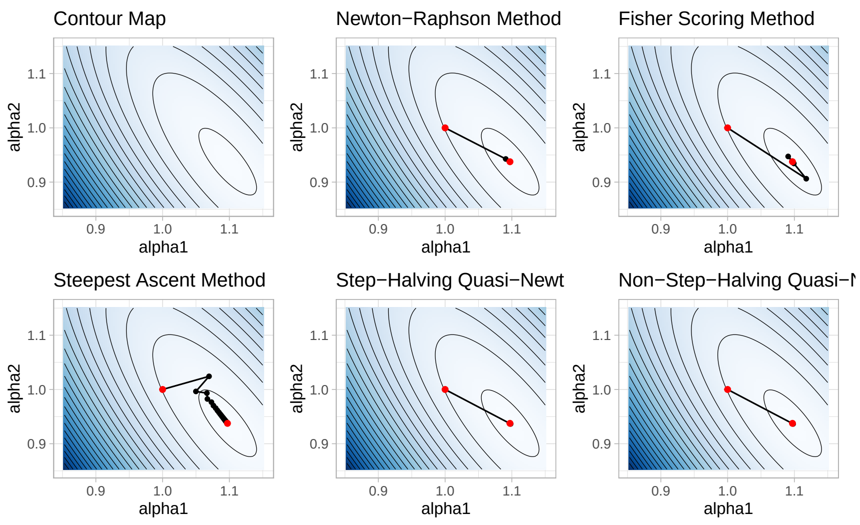

图 12: Iteration Path Starting with (1, 1)

```r
compare_path(c(5, 5), 0, 5, 25)
```

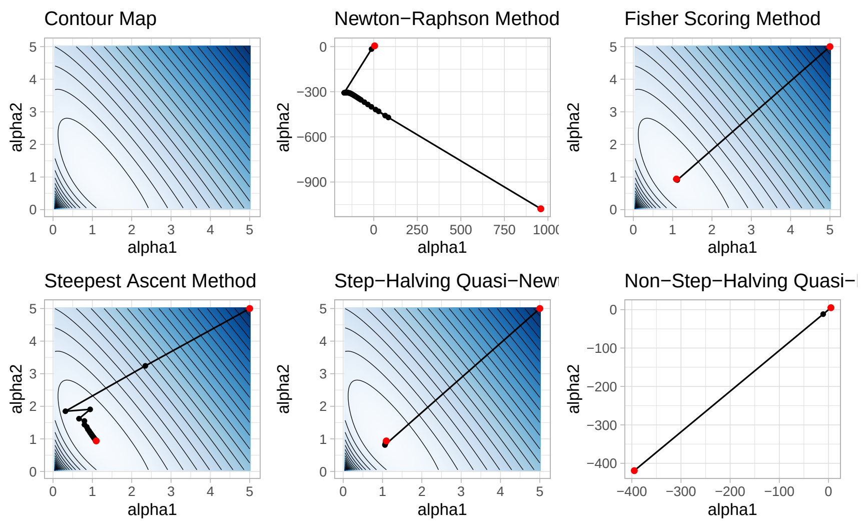

图 13: Iteration Path Starting with (5, 5)

```
compare_path(c(10, 10), 0, 10, 25)
```

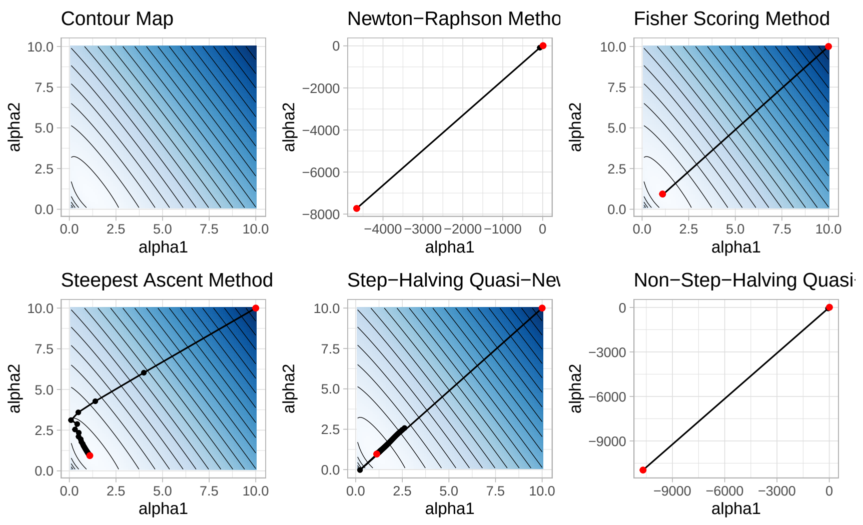

图 14: Iteration Path Starting with (10, 10)

From the graphs above, we can come to summarize that:

- **Newton-Raphson method**: converges fast; not stable because it will fails to converge when the Hessian matrix is singular.
- **Fisher Scoring**: converges fast at first; very stable since it can always converge (if the starting point is not too far away from true point); precision may be a bit lower due to approximation.
- **Steepest Ascent method**: needs many steps to converge; very stable since it can always converge (backtracking).
- **Step Halving Quasi Newton method**: converge not so fast; very stable since it can always converge (backtracking).
- **Non Step Halving Quasi Newton method**: converges fast if given good starting values; very unstable (always cannot converge).

which is consistent with the conclusion in the previous problems (a)-(f).

## THE END. THANKS! ^_^