005921309_stats102b_hw3

Anish Deshpande

2024-05-05

Question 1:

a.)

```r
# g is a function: R^n -> R
# grad_g is the gradient of g: a function \del g: R^n -> R^n
# hess_g is the hessian of g: a function \del g: R^n -> R^(nxn)
# w0 is the initial point
# K is the number of iterations
newts_method <- function(g, grad_g, hess_g, w0, K) {
  # create initial list object:
  result <- list(index = c(), val = c())

  # set initial point:
  curr_point <- w0
  # create initial empty coordinate matrix: (ncol = n+1)
  mat <- matrix(nrow = K, ncol = length(w0) + 1)

  # iterate for K iterations:
  for (i in 1:K) {
    # calculate gradient at current step:
    grad_curr <- grad_g(curr_point)
    # calculate hessian at current step:
    hess_curr <- hess_g(curr_point)
    # calculate inverse of the hessian:
    inv_hess <- solve(hess_curr)
    # update step...
    mat[i, 1:length(w0)] <- curr_point - inv_hess %*% grad_curr
    curr_point <- mat[i, 1:length(w0)]
    # fill out function output given current point
    mat[i, length(w0) + 1] <- g(curr_point)
  }

  inputs <- mat[ , 1:length(w0)]
  outputs <- mat[ , length(w0) + 1]
  min_val <- min(outputs)
  # check if there are duplicate minimum values
  min_indices <- which(outputs == min_val)
  num_min_indices <- length(min_indices)
  if (num_min_indices > 1) {
    # if there are multiple min values, choose the first one
    min_index <- inputs[min_indices[1], ]
  } else {
    # If there is only one unique minimum value, directly extract its index
    min_index <- inputs[min_indices, ]
  }

  result$index <- min_index
  result$val <- min_val

  # return result
  result
}
```

b.)

```r
g <- function(w) {
  exp((-(w[1]^2))*(w[2])) + sin(w[2])
}

grad_g <- function(w) {
  c((-2)*w[1]*w[2]*exp((-(w[1]^2))*(w[2])), -(w[1]^2)*exp((-(w[1]^2))*(w[2])) + cos(w[2]))
}

hess_g <- function(w) {
  # create hessian matrix...
  top_left <- (2 * w[2]^2 * w[1]^2 - 2 * w[2]) * exp(-(w[1]^2) * w[2])
  top_right <- -2 * w[1] * w[2] * exp(-(w[1]^2) * w[2]) - w[1]^3 * exp(-(w[1]^2) * w[2])
  bot_left <- -2 * w[1] * w[2] * exp(-(w[1]^2) * w[2]) - w[1]^3 * exp(-(w[1]^2) * w[2])
  bot_right <- (w[1]^4 - 2 * w[1]^2 + 1) * exp(-(w[1]^2) * w[2]) - sin(w[2])
  result <- matrix(c(top_left, top_right, bot_left, bot_right), nrow = 2, byrow = TRUE)
}

newts_method(g, grad_g, hess_g, c(1, -5), 10)
```

```
## $index
## [1] 0.8273677 7.7469077
##
## $val
## [1] 0.9992495
```

- Newton's method typically converges faster that gradient descent because gradient descent only takes into account the gradient of the function at various points, while Newton's Method considers second order conditions about the function, taking into account its curvature and shape near the current point.
- A fallback of Newton's method is that it is very computationally expensive. It involves calculating the hessian and then taking the inverse of it, which does not scale well into higher dimensions.

c.)

```r
g <- function(w) {
  w[1]^8 + w[2]^8
}

grad_g <- function(w) {
  c(8*(w[1]^7), 8*(w[2]^7))
}

hess_g <- function(w) {
  matrix(c(56*(w[1]^6), 0, 0, 56*(w[2]^6)), nrow = 2, byrow = TRUE)
}

newts_method(g, grad_g, hess_g, c(1, -1), 1000)
```

```
## $index
## [1]  3.141899e-41 -3.141899e-41
##
## $val
## [1] 0
```

- I did not receive an error for running this function, possibly because of the way my function is written. Ideally, an error should occur because as the algorithm continues to step closer to the minimum value of the function, the gradient and the hessian become very small in value, since the slope of the function is very flat.
- Essentially, the error we should be seeing is a division by zero error, resulting in indeterminate output, as the value of the denominator is less than machine epsilon.
- The fix to this issue is to run a regularized Newton's Method, which adds a small value to the denominator in order to take away the division by zero error.

d.)

```r
reg_newts_method <- function(g, grad_g, hess_g, w0, K, e) {
  # create initial list object:
  result <- list(index = c(), val = c())

  # set initial point:
  curr_point <- w0
  # create initial empty coordinate matrix: (ncol = n+1)
  mat <- matrix(nrow = K, ncol = length(w0) + 1)

  # iterate for K iterations:
  for (i in 1:K) {
    # calculate gradient at current step:
    grad_curr <- grad_g(curr_point)
    # calculate hessian at current step:
    hess_curr <- hess_g(curr_point)
    # calculate hessian with error term:
    hess_adj <- hess_curr + e*diag(length(w0))
    # calculate inverse of the hessian:
    inv_hess <- solve(hess_adj)
    # update step...
    mat[i, 1:length(w0)] <- curr_point - inv_hess %*% grad_curr
    curr_point <- mat[i, 1:length(w0)]
    # fill out function output given current point
    mat[i, length(w0) + 1] <- g(curr_point)
  }

  inputs <- mat[ , 1:length(w0)]
  outputs <- mat[ , length(w0) + 1]
  min_val <- min(outputs)
  # check if there are duplicate minimum values
  min_indices <- which(outputs == min_val)
  num_min_indices <- length(min_indices)
  if (num_min_indices > 1) {
    # if there are multiple min values, choose the first one
    min_index <- inputs[min_indices[1], ]
  } else {
    # If there is only one unique minimum value, directly extract its index
    min_index <- inputs[min_indices, ]
  }

  result$index <- min_index
  result$val <- min_val

  # return result
  result
}

reg_newts_method(g, grad_g, hess_g, c(1, -1), 1000, 10^-7)
```

```
## $index
## [1]  0.01135025 -0.01135025
##
## $val
## [1] 5.508996e-16
```

e.)

```r
# vary the e parameter....
e_values <- c(1, 10^-5, 10^-20, 10^-50, 10^-90, 10^-128)
# initialize empty list to store the results
results <- list()
# iterate over each value in e_values and call reg_newts_method function
for (e_val in e_values) {
  result <- reg_newts_method(g, grad_g, hess_g, c(1, -1), 1000, e_val)
  results[[as.character(e_val)]] <- result
}
# print the results
options(digits = 10)
results
```

```
## $`1`
## $`1`$index
## [1]  0.1661072842 -0.1661072842
##
## $`1`$val
## [1] 1.159149409e-06
##
##
## $`1e-05`
## $`1e-05`$index
## [1]  0.02443261361 -0.02443261361
##
## $`1e-05`$val
## [1] 2.539740885e-13
##
##
## $`1e-20`
## $`1e-20`$index
## [1]  7.776463699e-05 -7.776463699e-05
##
## $`1e-20`$val
## [1] 2.674774585e-33
##
##
## $`1e-50`
## $`1e-50`$index
## [1]  7.884188865e-10 -7.884188865e-10
##
## $`1e-50`$val
## [1] 2.985974876e-73
##
##
## $`1e-90`
## $`1e-90`$index
## [1]  1.733446814e-16 -1.733446814e-16
##
## $`1e-90`$val
## [1] 1.630475076e-126
##
##
## $`1e-128`
## $`1e-128`$index
## [1]  8.224044119e-23 -8.224044119e-23
##
## $`1e-128`$val
## [1] 4.185173317e-177
```

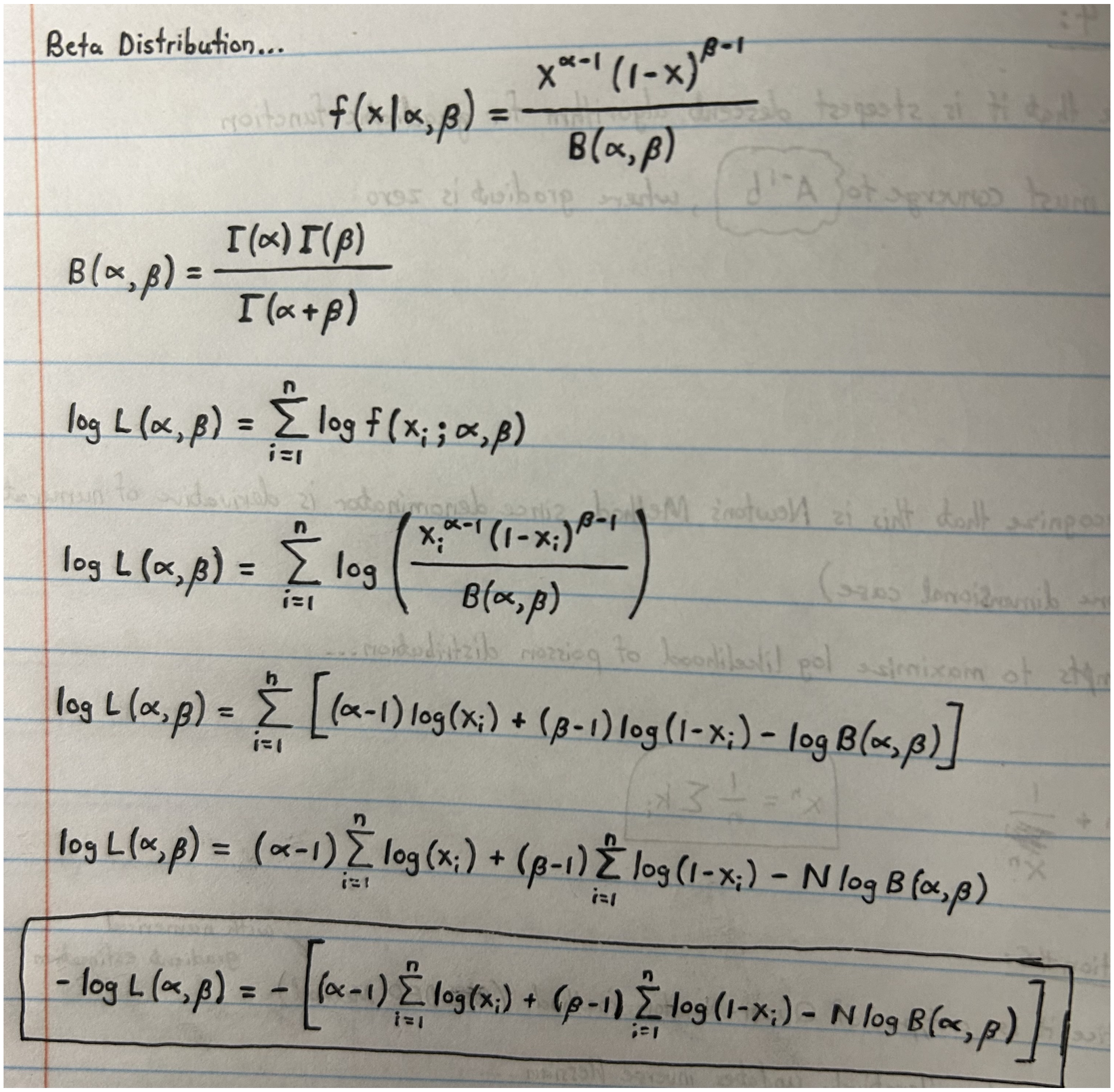
- We see that with a smaller "e" parameter, the algorithm provides a minimum value closer and closer to the true minimum of 0 after 1000 iterations.

Question 2:

a.)

```
# g: a function R^n -> R
# h: a small step size in the x direction
# k: a small step size in the y direction
# w0: initial point
# K: number of iterations
newts_method_num <- function(g, h, k, w0, K) {
  # create initial list object:
  result <- list(index = c(), val = c())

  # set initial point:
  curr_point <- w0
  # create initial empty coordinate matrix: (ncol = n+1)
  mat <- matrix(nrow = K, ncol = length(w0) + 1)

  # creating a gradient function:
  grad_g <- function(x) {
    result_vec <- c()
    for (i in 1:length(x)) {
      # create standard basis vector
      sbv <- rep(0, length(w0))
      sbv[i] <- 1
      current_result <- (g(x + h*sbv) - g(x - h*sbv)) / (2*h)
      result_vec <- append(result_vec, current_result)
    }
    result_vec
  }

  # creating a hessian function:
  hess_g <- function(x) {
    h_vec <- c(1, 0)
    k_vec <- c(0, 1)
    hk_vec <- c(h, k)
    fxx <- (g(x + h*h_vec) - 2*g(x) + g(x - h*h_vec)) / h^2
    fyy <- (g(x + k*k_vec) - 2*g(x) + g(x - k*k_vec)) / k^2
    fxy <- (g(x + hk_vec*c(1, 1)) - g(x + hk_vec*c(1, -1)) -
              g(x + hk_vec*c(-1, 1)) + g(x + hk_vec*c(-1, -1))) / 4*h*k
    result <- matrix(c(fxx, fxy, fxy, fyy), nrow = 2, byrow = TRUE)
    result
  }

  # iterate for K iterations:
  for (i in 1:K) {
    # calculate gradient at current step:
    grad_curr <- grad_g(curr_point)
    # calculate hessian at current step:
    hess_curr <- hess_g(curr_point)
    # calculate inverse of the hessian:
    inv_hess <- solve(hess_curr)
    # update step...
    mat[i, 1:length(w0)] <- curr_point - inv_hess %*% grad_curr
    curr_point <- mat[i, 1:length(w0)]
    # fill out function output given current point
    mat[i, length(w0) + 1] <- g(curr_point)
  }

  inputs <- mat[ , 1:length(w0)]
  outputs <- mat[ , length(w0) + 1]
  min_val <- min(outputs)
  # check if there are duplicate minimum values
  min_indices <- which(outputs == min_val)
  num_min_indices <- length(min_indices)
  if (num_min_indices > 1) {
    # if there are multiple min values, choose the first one
    min_index <- inputs[min_indices[1], ]
  } else {
    # If there is only one unique minimum value, directly extract its index
    min_index <- inputs[min_indices, ]
  }

  result$index <- min_index
  result$val <- min_val

  # return result
  result
}
```

b.)



Beta Distribution...

$$f(x|\alpha,\beta) = \frac{x^{\alpha-1}(1-x)^{\beta-1}}{B(\alpha,\beta)}$$

$$B(\alpha,\beta) = \frac{\Gamma(\alpha)\Gamma(\beta)}{\Gamma(\alpha+\beta)}$$

$$\log L(\alpha,\beta) = \sum_{i=1}^{n} \log f(x_i;\alpha,\beta)$$

$$\log L(\alpha,\beta) = \sum_{i=1}^{n} \log\left(\frac{x_i^{\alpha-1}(1-x_i)^{\beta-1}}{B(\alpha,\beta)}\right)$$

$$\log L(\alpha,\beta) = \sum_{i=1}^{n}\left[(\alpha-1)\log(x_i) + (\beta-1)\log(1-x_i) - \log B(\alpha,\beta)\right]$$

$$\log L(\alpha,\beta) = (\alpha-1)\sum_{i=1}^{n}\log(x_i) + (\beta-1)\sum_{i=1}^{n}\log(1-x_i) - N\log B(\alpha,\beta)$$

$$\boxed{-\log L(\alpha,\beta) = -\left[(\alpha-1)\sum_{i=1}^{n}\log(x_i) + (\beta-1)\sum_{i=1}^{n}\log(1-x_i) - N\log B(\alpha,\beta)\right]}$$

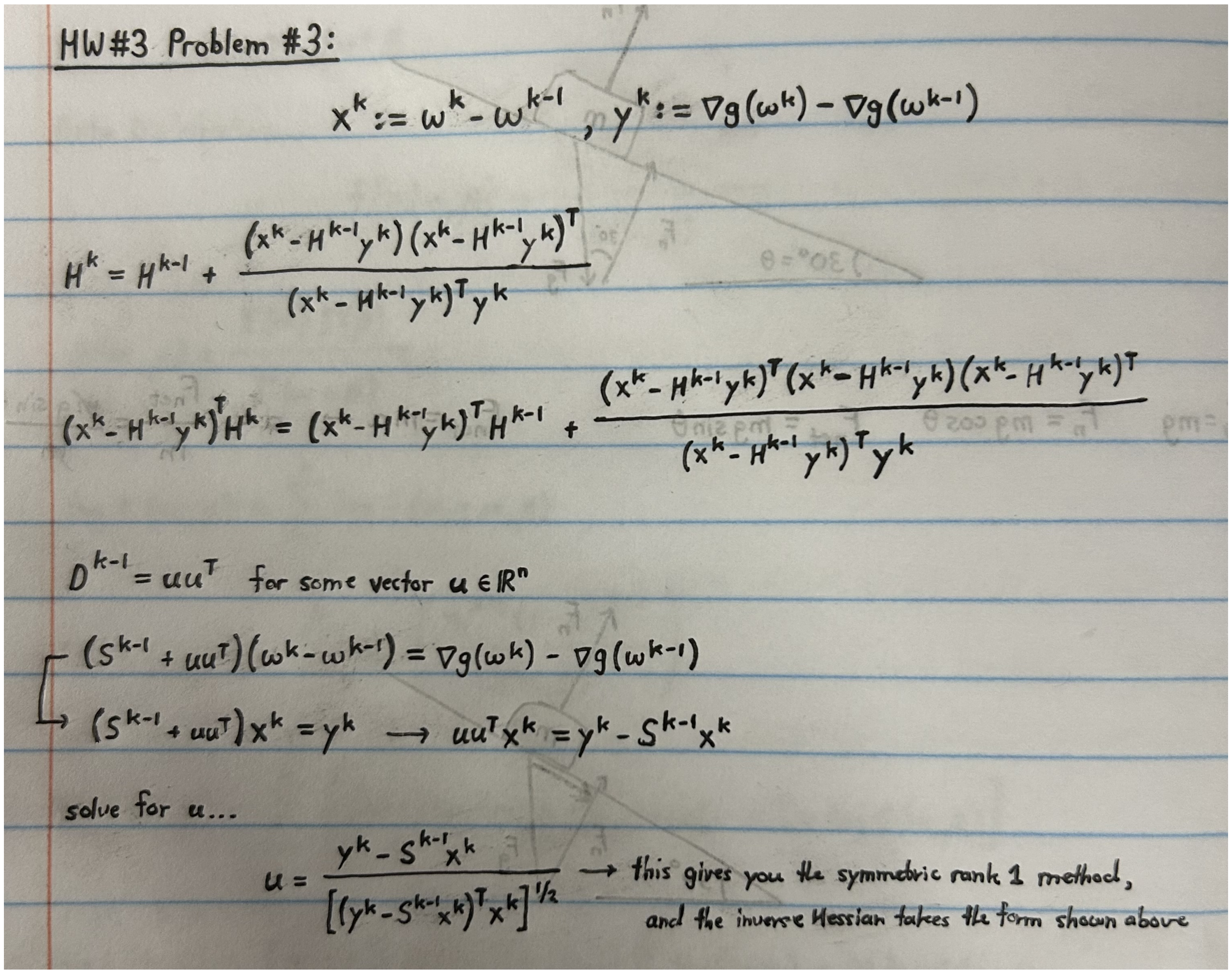
c.)

```r
# reading in RDS file of beta data values...
beta_values <- readRDS("beta_values.rds")

# creating negative log likelihood function:
n <- length(beta_values)

l <- function(params) {
  -((params[1] - 1)*sum(log(beta_values)) + (params[2] - 1)*sum(log(1 - beta_values)) -
      n*log(beta(params[1], params[2])))
}

newts_method_num(l, 0.0001, 0.0001, c(1, 1), 100)
```

```
## $index
## [1] 2.464958847 6.045783042
##
## $val
## [1] -548.9442631
```

- The results from above show that with the given beta values, the parameters of the beta distribution that would make the observed data most likely is alpha = 2.4649, and beta = 6.0457

Question 3:



HW#3 Problem #3:

$$x^k := w^k - w^{k-1}, \quad y^k := \nabla g(w^k) - \nabla g(w^{k-1})$$

$$H^k = H^{k-1} + \frac{(x^k - H^{k-1}y^k)(x^k - H^{k-1}y^k)^T}{(x^k - H^{k-1}y^k)^T y^k}$$

$$(x^k - H^{k-1}y^k)^T H^k = (x^k - H^{k-1}y^k)^T H^{k-1} + \frac{(x^k - H^{k-1}y^k)^T (x^k - H^{k-1}y^k)(x^k - H^{k-1}y^k)^T}{(x^k - H^{k-1}y^k)^T y^k}$$

$D^{k-1} = uu^T$ for some vector $u \in \mathbb{R}^n$

$$(S^{k-1} + uu^T)(w^k - w^{k-1}) = \nabla g(w^k) - \nabla g(w^{k-1})$$

$$\rightarrow (S^{k-1} + uu^T)x^k = y^k \longrightarrow uu^T x^k = y^k - S^{k-1}x^k$$

solve for u...

$$u = \frac{y^k - S^{k-1}x^k}{\left[(y^k - S^{k-1}x^k)^T x^k\right]^{1/2}} \longrightarrow$$

this gives you the symmetric rank 1 method, and the inverse Hessian takes the form shown above

Question 4:

a.)

```r
# g: function R^n -> R
# grad_g: gradient of g, a function \del g: R^n -> R^b
# w0: initial point
# K: number of iterations
bfgs <- function(g, grad_g, w0, K) {
  # create initial list object:
  result <- list(index = c(), val = c())
  # set initial point:
  curr_point <- w0
  # create initial empty coordinate matrix: (ncol = n+1)
  mat <- matrix(0, nrow = K, ncol = length(w0) + 1)
  # set initial hessian approximation to be the identity matrix
  curr_hess <- diag(length(w0))
  # set initial gradient to be the gradient at the current point
  curr_grad <- grad_g(curr_point)

  # iterate for K iterations or until convergence:
  for (i in 1:K) {
    prev_point <- curr_point
    prev_hess <- curr_hess
    prev_grad <- curr_grad

    # update the current values...
    curr_point <- prev_point - solve(prev_hess, prev_grad)
    curr_grad <- grad_g(curr_point)
    # introduce x_k and y_k
    x_k <- curr_point - prev_point
    y_k <- curr_grad - prev_grad

    # update the current hessian approximation
    skTyk <- as.numeric(t(x_k) %*% y_k)
    if (skTyk == 0) {
      curr_hess <- prev_hess
    } else {
      curr_hess <- prev_hess + (y_k %*% t(y_k)) / skTyk -
        (prev_hess %*% x_k) %*% t(prev_hess %*% x_k) / (as.numeric(t(x_k) %*% prev_hess %*% x_k))
    }

    # add info to output matrix
    mat[i, 1:length(w0)] <- curr_point
    # fill out function output given current point
    mat[i, length(w0) + 1] <- g(curr_point)

    # check for convergence
    if (norm(as.matrix(curr_grad)) < 1e-6 || norm(as.matrix(curr_point) - as.matrix(prev_point)) < 1e-6
      if (i > 2) {
        break
      }
    }
  }

  # Remove extra rows from mat (if convergence happened before K iterations)
  mat <- mat[1:i, ]

  inputs <- mat[ , 1:length(w0)]
  outputs <- mat[ , length(w0) + 1]
  min_val <- min(outputs)
  # check if there are duplicate minimum values
  min_indices <- which(outputs == min_val)
  num_min_indices <- length(min_indices)
  if (num_min_indices > 1) {
    # if there are multiple min values, choose the first one
    min_index <- inputs[min_indices[1], ]
  } else {
    # If there is only one unique minimum value, directly extract its index
    min_index <- inputs[min_indices, ]
  }

  result$index <- min_index
  result$val <- min_val
  # return result
  result
}
```

b.)

```r
g <- function(w) {
  (1 - w[1])^2 + 100*(w[2] - w[1]^2)^2
}

grad_g <- function(w) {
  c(-2*(1-w[1]) + 200*(w[2]-w[1]^2)*(-2*w[1]), 200*(w[2] - w[1]^2))
}

# try different values of K and different starting points
bfgs(g, grad_g, c(2, 5), 10)
```

```
## $index
## [1] 2.008294097 4.033187516
##
## $val
## [1] 1.016657319
```

```r
bfgs(g, grad_g, c(2, 5), 100)
```

```
## $index
## [1] 1.000000003 1.000000006
##
## $val
## [1] 1.869234806e-17
```

```r
bfgs(g, grad_g, c(1, 1), 10)
```

```
## $index
## [1] 1 1
##
## $val
## [1] 0
```

```r
bfgs(g, grad_g, c(20, 12), 5)
```

```
## $index
## [1]  19.99993748 399.99749937
##
## $val
## [1] 360.9976243
```

- we see that my BFGS algorithm works well with initial points near the true global minimum and with enough number of iterations, however, given a w0 relatively far from the true minimum, the algorithm may begin to experience erratic and unstable behavior, and might not converge appropriately.

c.)

```r
f <- function(x) {
  ((1/2)*(x[1]^2 + x[2]^2 + x[3]^2)) - ((1/2)*(x[1] + x[2]) + (1/3)*(x[2] + x[3]))
}

grad_f <- function(x) {
  c(x[1] - (1/2), x[2] - (1/2) - (1/3), x[3] - (1/3))
}

bfgs(f, grad_f, c(50, 20, 54), 20)
```

```
## $index
## [1] 0.5000000000 0.8333333333 0.3333333333
##
## $val
## [1] -0.5277777778
```

d.)

```r
optim(par = c(5, 5, 5), fn = f, method = "BFGS")
```

```
## $par
## [1] 0.5000000000 0.8333333333 0.3333333333
##
## $value
## [1] -0.5277777778
```

```
##
## $counts
## function gradient
##        9        2
##
## $convergence
## [1] 0
##
## $message
## NULL
```