005921309_stats102b_hw2

Anish Deshpande

2024-04-27

Question 1:

a.)

Question #1 (Part A) HW#2

set S is convex if, for any two points x_1, x_2 in S, the line segment connecting x_1 and x_2 lies entirely within S.

$S = \{x \in \mathbb{R}^n : \|x\|_2 \leq r\}$ ← set consists of all points in $\mathbb{R}^n$ whose euclidean norm is less than some positive constant r

Euclidean Norm: $\|x\|_2 = \sqrt{x^T x} = \sqrt{x_1^2 + x_2^2 + \ldots + x_n^2}$, where $x_1, \ldots, x_n$ are components of x

Proof:

- consider 2 points x_1, x_2 in S. This means that $\|x_1\|_2 \leq r$ and $\|x_2\|_2 \leq r$
- we want to show that the point $\lambda x_1 + (1-\lambda)x_2$ is also in S, where λ is a scalar in $[0,1]$

Let $x = \lambda x_1 + (1-\lambda)x_2 \ldots$

$$\|x\|_2 = \sqrt{(\lambda x_1 + (1-\lambda)x_2)^T(\lambda x_1 + (1-\lambda)x_2)} = \sqrt{\lambda^2\|x_1\|_2^2 + 2\lambda(1-\lambda)x_1^T x_2 + (1-\lambda)^2\|x_2\|_2^2}$$

Since $\|x_1\|_2 < r$ and $\|x_2\|_2 < r \ldots$

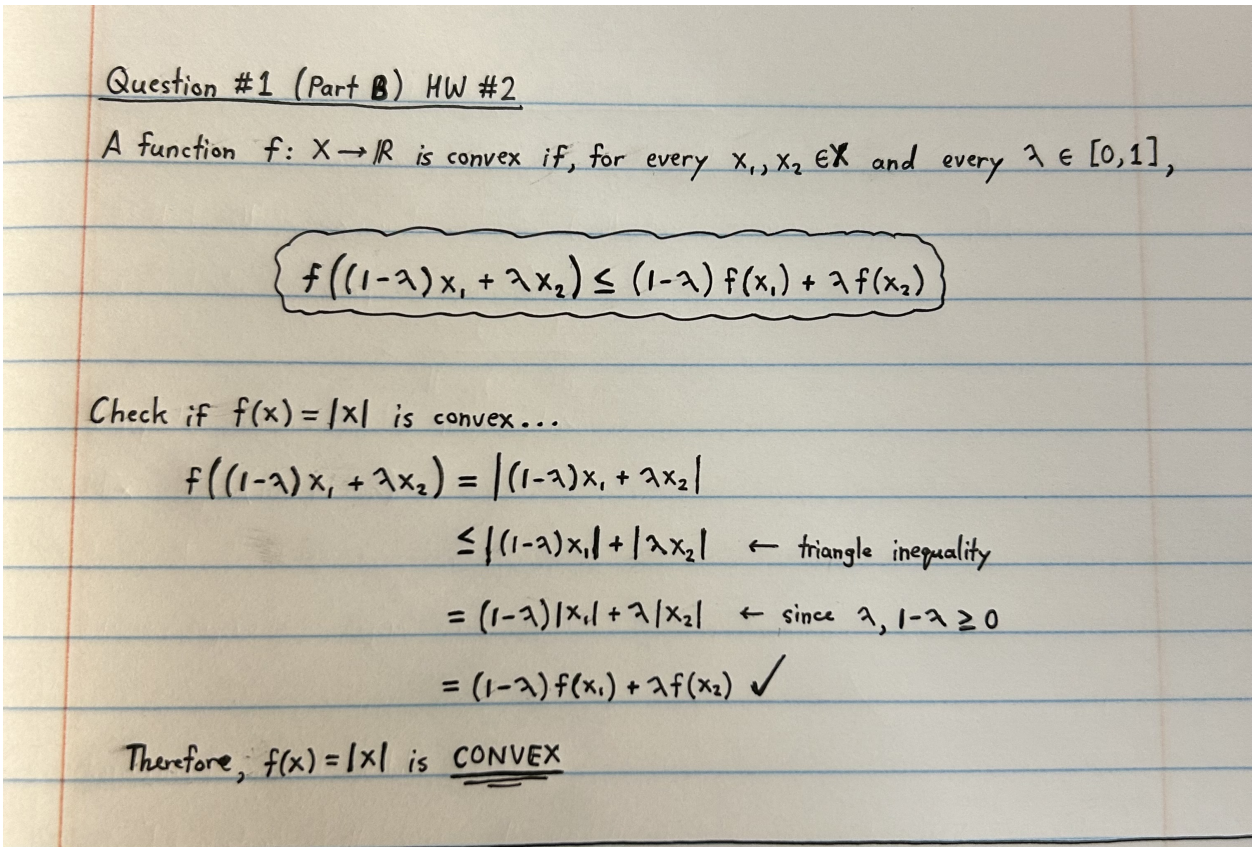
$$\|x\|_2 \leq \sqrt{\lambda^2 r^2 + 2\lambda(1-\lambda)x_1^T x_2 + (1-\lambda)^2 r^2}$$

$$= \sqrt{r^2(\lambda^2 + 2\lambda(1-\lambda) + (1-\lambda)^2)} = \sqrt{r^2(\lambda^2 + 2\lambda - 2\lambda^2 + 1 + \lambda^2 - 2\lambda)} = \sqrt{r^2(1)}$$

$$= r$$

Thus $\|x\|_2 \leq r$, so S is convex ✓

b.)



Question #1 (Part B) HW #2

A function $f: X \to \mathbb{R}$ is convex if, for every $x_1, x_2 \in X$ and every $\lambda \in [0,1]$,

$$f((1-\lambda)x_1 + \lambda x_2) \leq (1-\lambda) f(x_1) + \lambda f(x_2)$$

Check if $f(x) = |x|$ is convex...

$$f((1-\lambda)x_1 + \lambda x_2) = |(1-\lambda)x_1 + \lambda x_2|$$

$$\leq |(1-\lambda)x_1| + |\lambda x_2| \quad \leftarrow \text{triangle inequality}$$

$$= (1-\lambda)|x_1| + \lambda |x_2| \quad \leftarrow \text{since } \lambda, 1-\lambda \geq 0$$

$$= (1-\lambda) f(x_1) + \lambda f(x_2) \checkmark$$

Therefore, $f(x) = |x|$ is CONVEX

c.)

Question #1 (Part C) HW#2

Check if $f(x) = x^2$ is convex...

Pick x_1, x_2 s.t. $x_1 \neq x_2$, and pick $\lambda \in (0,1)$

$$f((1-\lambda)x_1 + \lambda x_2) = ((1-\lambda)x_1 + \lambda x_2)^2$$

$$= (1-\lambda)^2 x_1^2 + \lambda^2 x_2^2 + 2(1-\lambda)\lambda x_1 x_2$$

Since $x_1 \neq x_2$, $(x_1 - x_2)^2 > 0$, so $x_1^2 + x_2^2 > 2x_1 x_2$

Thus...

$$(1-\lambda)^2 x_1^2 + \lambda^2 x_2^2 + 2(1-\lambda)\lambda x_1 x_2 < (1-\lambda)^2 x_1^2 + \lambda^2 x_2^2 + (1-\lambda)(\lambda)(x_1^2 + x_2^2)$$

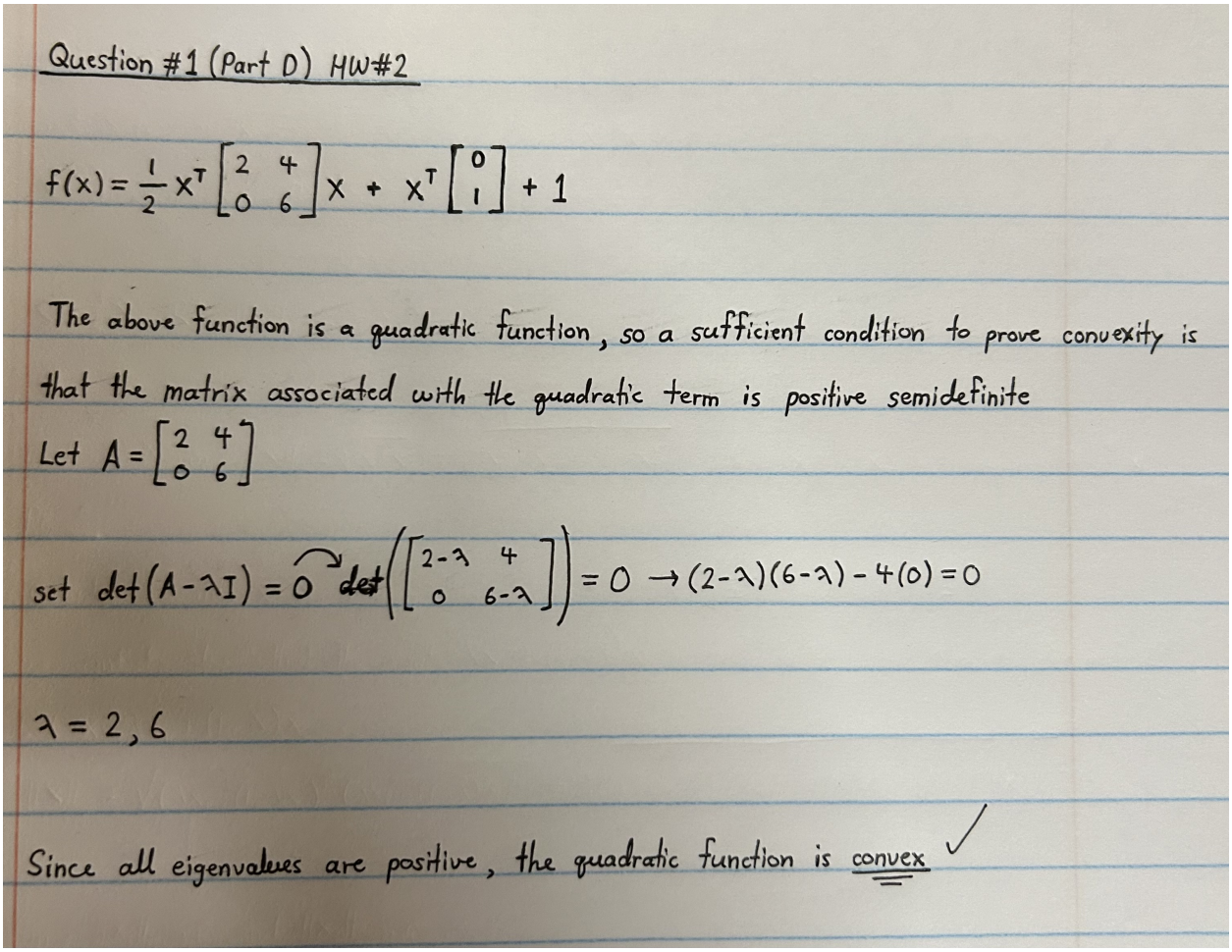
$$= (1 - 2\lambda - \lambda^2 + \lambda + \lambda^2) x_1^2 + (\lambda - \lambda^2 + \lambda^2) x_2^2$$

$$= (1-\lambda) x_1^2 + \lambda x_2^2$$

$$= (1-\lambda) f(x_1) + \lambda f(x_2) \checkmark$$

Therefore, $f(x) = x^2$ is CONVEX

d.)



Question 2:

a.)

```
# g: function from R -> R
# grad_g: gradient of g, a function \del g: R -> R
# alpha: step size
# w0: initial point
# K: number of iterations
grad_desc <- function(g, grad_g, alpha, w0, K) {
  # create initial list object
  result <- list(index = c(), val = c(), coord_matrix = c())
  # establish current point
  curr_point <- w0
  indexes <- c()
  # create initial matrix (ncol = 2, since only one dimension)
  mat <- matrix(nrow = K, ncol = 2)
```

```
  # iterate for K iterations...
  for (i in 1:K) {
    # calculate gradient at current step
    grad_curr <- grad_g(curr_point)
    # update...
    mat[i, 1] <- curr_point - alpha * grad_curr
    curr_point <- mat[i, 1]
    # add current point to index
    indexes <- append(indexes, curr_point)
  }

  eval_points <- g(indexes)
  mat[ , 2] <- eval_points
  min_val <- min(eval_points)
  min_index <- indexes[which(eval_points == min_val)]

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

  # return the result object:
  result
}
```

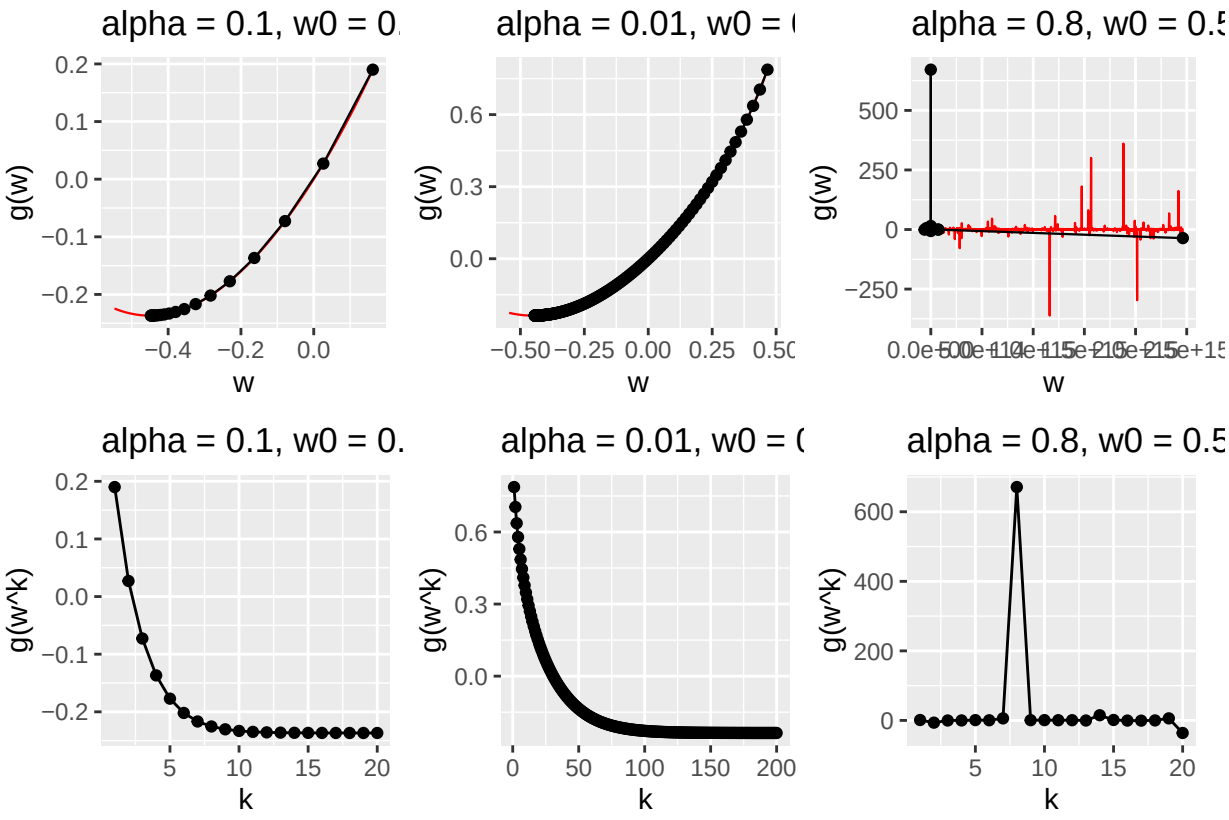
b.)

```
sec <- function(x) {
  1 / cos(x)
}

g <- function(w) {
  tan(w^2 + sin(w))
}

grad_g <- function(w) {
  (2*w + cos(w)) * (sec(w^2 + sin(w)))^2
}

output_1 <- grad_desc(g, grad_g, 0.1, 0.5, 20)
output_2 <- grad_desc(g, grad_g, 0.01, 0.5, 200)
output_3 <- grad_desc(g, grad_g, 0.8, 0.5, 20)
```

- With input values alpha = 0.1, w0 = 0.5, and K = 20, the local minimum is predicted to occur at -0.4474417
- With input values alpha = 0.01, w0 = 0.5, and K = 200, the local minimum is predicted to occur at -0.4439198
- With input values alpha = 0.8, w0 = 0.5, and K = 20, the local minimum is predicted to occur at 2.4621739×10^{15}

```r
# for output #1...
# generate sequence of x-values for plotting function g(w)
x_values <- seq(min(output_1$coord_matrix[, 1]) - 0.1, max(output_1$coord_matrix[, 1]),
                length.out = 1000)

# calculate corresponding g(w) values
g_values <- g(x_values)
# Create a data frame for plotting
df <- data.frame(x = x_values, g = g_values, k = 1:nrow(output_1$coord_matrix))

# plot it:
p1 <- ggplot(data = df, aes(x = x, y = g)) +
  geom_line(color = "red", linewidth = 0.4) +
  geom_line(data = as.data.frame(output_1$coord_matrix),
            aes(x = output_1$coord_matrix[,1], y = output_1$coord_matrix[,2]),
            linewidth = 0.4) +
  geom_point(data = as.data.frame(output_1$coord_matrix),
            aes(x = output_1$coord_matrix[,1], y = output_1$coord_matrix[,2])) +
  xlab("w") + ylab("g(w)") +
  ggtitle("alpha = 0.1, w0 = 0.5, K = 20")

# for output #1...
df <- data.frame(x = output_1$coord_matrix[ ,1],
                 g = output_1$coord_matrix[ ,2], k = 1:nrow(output_1$coord_matrix))
p4 <- ggplot(data = df, aes(x = k, y = g)) +
  geom_line() + geom_point() +
  ylab("g(w^k)") + ggtitle("alpha = 0.1, w0 = 0.5, K = 20")


# I have similar code to get the next plots, however since it will be a lot of code,
# I am hiding it. (echo = FALSE)
```

```r
layout <- (p1 | p2 | p3) /
         (p4 | p5 | p6)
layout
```

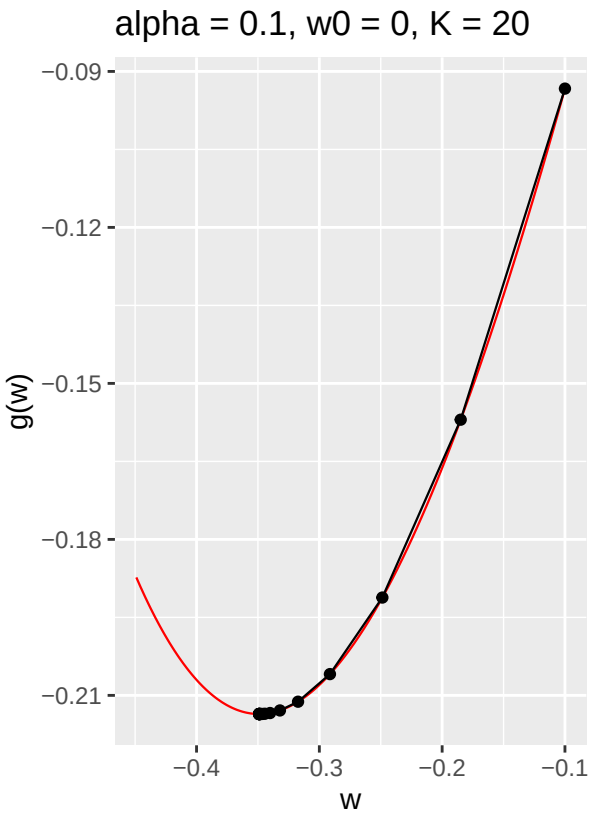


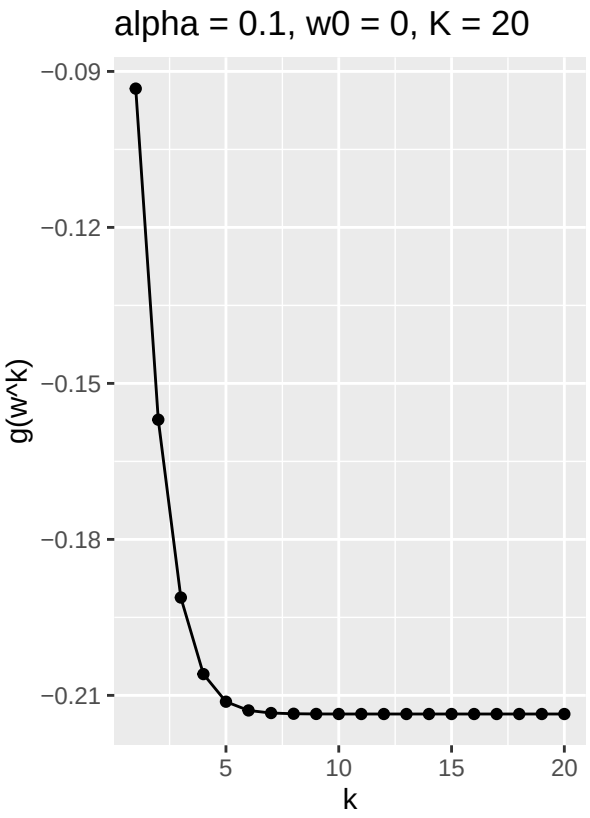
c.)

```
g <- function(w) {
  (1/4)*(w^4) - (3/2)*(w^3) + (1/2)*(w^2) + sin(w)
}

grad_g <- function(w) {
  (w^3) - (9/2)*(w^2) + w + cos(w)
}

output_4 <- grad_desc(g, grad_g, alpha = 0.1, w0 = 0, K = 20)
```

- The local minimum of the function g given the above gradient descent parameters is predicted to occur at -0.3490194, and the value of the function at that minimum is predicted to be -0.2135861.

```
layout_2 <- (p7 | p8)
layout_2
```





Question 3:

a.)

```
# g is a function: R^n -> R
# grad_g is the gradient of g: \del g: R^n -> R^n
# alpha is the step size
# w0 is the initial point
# K is the number of iterations
grad_desc_n <- function(g, grad_g, alpha, 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)
    # update...
```

```
    mat[i, 1:length(w0)] <- curr_point - alpha * grad_curr
    curr_point <- mat[i, 1:length(w0)]
  }

  # fill out the last column of the matrix:
  mat[, length(w0) + 1] <- apply(mat[, 1:length(w0)], 1, function(row) g(row))

  inputs <- mat[, 1:length(w0)]
  outputs <- mat[, length(w0) + 1]
  min_val <- min(outputs)
  min_index <- inputs[which(outputs == min_val), ]

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

  # return the result object:
  result
}
```

b.)

- The local minimum of the function $g(w_1, w_2) = (w_1 - 2)^2 + (w_2 - 4)^2 - 1$ is relatively easy to find out.
- By inspection, we see that both the squared terms can never be negative, so to minimize the function, both the squared terms should equal zero. This is only possible if $w_1 = 2$ and if $w_2 = 4$. Thus, the global minimum of this function is $(2, 4)$.

```
g <- function(w) {
  (w[1] - 2)^2 + (w[2] - 4)^2 - 1
}

grad_g <- function(w) {
  c(2*(w[1] - 2), 2*(w[2] - 4))
}

output_5 <- grad_desc_n(g, grad_g, 0.2, c(10, 11), 20)
output_5$index
```

```
## [1] 2.000292 4.000256
```

```
output_5$val
```

```
## [1] -0.9999998
```

- we see that the gradient descent algorithm comes very close to the true minimum value after 20 iterations and a step size of 0.2.

c.)

```
g <- function(w) {
  sin(w[1] + 2) + cos(w[2] + 4) + (w[3])^2
}

grad_g <- function(w) {
  c(cos(w[1] + 2), -sin(w[2] + 4), 2 * w[3])
}

alpha = 0.4
w0 = c(0, 25, 6)
K = 30
output_6 <- grad_desc_n(g, grad_g, alpha, w0, K)
```

- My gradient descent algorithm gives me a minimum value of -2 regardless of my starting w0, given a step size of 0.4 and 30 iterations.
- By inspection, we can confirm this is correct, since the minimum value of sin(x) is -1 and the minimum value of cos(x) is also -1. Adding these two terms together gives us a minimum of -2.
- Since this function involves terms of sine and cosine, it will oscillate and have numerous local minimums.
- The formula for all local minima is

$$(w_1 = \frac{3\pi}{2} - 2 + 2n\pi, w_2 = \pi + 4 + 2n\pi, w_3 = 0)$$

  where $n \in \mathbb{Z}$.

Question 4:

a.)

Question #4 (Part A) HW#2

pdf $\rightarrow f(x|\alpha,\beta) = \frac{\beta^{\alpha}}{\Gamma(\alpha)} x^{\alpha-1} e^{-\beta x}$, where $\Gamma(\alpha) = \int_0^{\infty} x^{\alpha-1} e^{-x}\, dx$ is the gamma function

First write likelihood function...

$$L(\alpha,\beta) = \prod_{i=1}^{n} f(x_i|\alpha,\beta)$$

$$L(\alpha,\beta) = \prod_{i=1}^{n} \frac{\beta^{\alpha}}{\Gamma(\alpha)} x_i^{\alpha-1} e^{-\beta x_i}$$

Now take negative logarithm of likelihood function...

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

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

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

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

$$= \boxed{-\left[n\left(\alpha\log\beta - \log\Gamma(\alpha)\right) + (\alpha-1)\sum_{i=1}^{n}\log(x_i) - \beta\sum_{i=1}^{n} x_i\right]}$$ ✓

b.)

```
# reading in RDS file
gamma_values <- readRDS("gamma_values.rds")

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

l <- function(params) {
  -(n * (params[1] * log(params[2] - log(gamma(params[1])))) +
      (params[1] - 1) * sum(log(gamma_values)) - params[2] * sum(gamma_values))
}

grad_desc_h <- function(g, h, alpha, 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)

  # create gradient approximation function:
  grad_g <- function(x) {
    result_vec <- c()
    for (i in 1:length(x)) {
      current_result <- (g(x + h) - g(x - h)) / (2*h)
      result_vec <- append(result_vec, current_result)
    }
    result_vec
  }

  # iterate for K iterations...
  for (i in 1:K) {
    # calculate gradient at current step
    grad_curr <- grad_g(curr_point)
    # update...
    mat[i, 1:length(w0)] <- curr_point - alpha * grad_curr
    curr_point <- mat[i, 1:length(w0)]
  }

  # fill out the last column of the matrix:
  mat[, length(w0) + 1] <- apply(mat[, 1:length(w0)], 1, function(row) g(row))

  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 minimum 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
}

output_7 <- grad_desc_h(l, 0.0001, 0.005, c(1.5, 1), 200)
output_7$index
```

```
## [1] 1.758879 1.258879
```

- The output from above shows that with the given 100 data points, the parameters of gamma distribution which would make the observed data most likely are alpha = 1.7588789, and beta = 1.2588789.

c.)

```
# implementing momentum based gradient descent (beta = decay rate):
grad_desc_mom <- function(g, h, alpha, w0, K, beta) {
  # 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)

  # create gradient approximation function:
  grad_g <- function(x) {
    result_vec <- c()
    for (i in 1:length(x)) {
      current_result <- (g(x + h) - g(x - h)) / (2*h)
      result_vec <- append(result_vec, current_result)
    }
    result_vec
  }

  # iterate for K iterations...
  for (i in 1:K) {
    # calculate gradient at current step
    grad_curr <- grad_g(curr_point)
    # update descent direction using momentum term
    if (i == 1) {
      d <- -grad_curr
    } else {
      d <- beta * d + (1 - beta) * (-grad_curr)
```

```
    }
    # update the current point
    curr_point <- curr_point + alpha * d
    mat[i, 1:length(w0)] <- curr_point
    mat[i, length(w0) + 1] <- g(curr_point)
  }

  min_index <- which.min(mat[, length(w0) + 1])
  if (length(min_index > 1)) {
    min_index <- min_index[1]
  }
  min_input <- mat[min_index, 1:length(w0)]
  min_output <- mat[min_index, length(w0) + 1]

  result$index <- min_input
  result$val <- min_output

  # return result:
  result
}

# looking at how results change with different beta values:
# Define the parameter values
h <- 0.0001
alpha <- 0.01
w0 <- c(1.5, 1)
K <- 50
beta_values <- c(0.1, 0.3, 0.5, 0.7, 0.9, 0.99)
# Initialize an empty list to store the results
results <- list()
# Iterate over each value of beta and call grad_desc_mom function
for (beta in beta_values) {
  result <- grad_desc_mom(l, h, alpha, w0, K, beta)
  results[[as.character(beta)]] <- result
}
# Print the results
options(digits = 10)
results
```

```
## $`0.1`
## $`0.1`$index
## [1] 1.75887892 1.25887892
##
## $`0.1`$val
## [1] 95.57000536
##
##
## $`0.3`
## $`0.3`$index
## [1] 1.758878936 1.258878936
##
## $`0.3`$val
## [1] 95.57000536
```

```
##
##
## $`0.5`
## $`0.5`$index
## [1] 1.758878931 1.258878931
##
## $`0.5`$val
## [1] 95.57000536
##
##
## $`0.7`
## $`0.7`$index
## [1] 1.75888532 1.25888532
##
## $`0.7`$val
## [1] 95.57000536
##
##
## $`0.9`
## $`0.9`$index
## [1] 1.758490778 1.258490778
##
## $`0.9`$val
## [1] 95.57000858
##
##
## $`0.99`
## $`0.99`$index
## [1] 1.763193998 1.263193998
##
## $`0.99`$val
## [1] 95.57040277
```

- We see that by holding all other parameters constant and changing beta, when beta is relatively low, the index and values stay constant as they approach a local minimum. However, when beta increases to large values like 0.9 and 0.99, the index begins to change slightly, and the predicted local minimum worsens as it increases in value ever so slightly.

Question 5:

```r
# implementing normalized gradient descent:
grad_desc_norm <- function(g, h, e, alpha, 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)

  # create gradient approximation function:
```

```
  grad_g <- function(x) {
    result_vec <- c()
    for (i in 1:length(x)) {
      current_result <- (g(x + h) - g(x - h)) / (2*h)
      result_vec <- append(result_vec, current_result)
    }
    result_vec
  }

  # create gradient normalization function:
  get_grad_norm <- function(gradient) {
    result <- sqrt(sum(gradient^2))
    result
  }

  # iterate for K iterations...
  for (i in 1:K) {
    # calculate gradient at current step
    grad_curr <- grad_g(curr_point)
    # calculate the norm of the gradient:
    norm <- get_grad_norm(grad_curr)
    # update the current point:
    curr_point <- curr_point - ((alpha) / (norm + e)) * grad_curr
    mat[i, 1:length(w0)] <- curr_point
    mat[i, length(w0) + 1] <- g(curr_point)
  }

  min_index <- which.min(mat[, length(w0) + 1])
  if (length(min_index > 1)) {
    min_index <- min_index[1]
  }
  min_input <- mat[min_index, 1:length(w0)]
  min_output <- mat[min_index, length(w0) + 1]

  result$index <- min_input
  result$val <- min_output

  # return result:
  result
}

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

output_8 <- grad_desc_norm(g, 0.0001, 0.0001, 0.005, c(1.5, 1), 2000)
output_9 <- grad_desc_h(g, 0.0001, 0.005, c(1.5, 1), 2000)
output_8
```

```
## $index
## [1]  0.25 -0.25
##
## $val
```

```
## [1] 3.051757812e-05
```

```
output_9
```

```
## $index
## [1]  0.3567812762 -0.1432187238
##
## $val
## [1] 0.000262729679
```

- Since normalized gradient descent has a constant step size of 0.005, it MUST continue to move a constant amount after every iteration. Thus, given enough iterations, the algorithm will find its way towards a local minimum. In this case, it approaches the local minimum perfectly, since alpha is a multiple of the difference between w0 and the true minimum.
- For the approximated gradient descent algorithm, the predicted index and value is not great. This is because of the slow crawling nature of gradient descent. The objective function has characteristics which make the gradient very close to zero as the index gets anywhere close to the true local min. Thus, even after 2000 iterations, the algorithm is still a ways away from the true minimum.