# Problem Set 8

*Todd Faulkenberry*

*11/30/2018*

## Question 1

### Part A

For the first part of this question, we were asked to use importance sampling to estimate the mean of a truncated t-distribution centered at -4. As its name implies, importance sampling allows us to emphasize the "important" values by using a distribution that promotes choosing these values, leading to reduced variance. This chosen distribution is biased but we can make our estimates from importance sampling unbiased as long as we weight them to correct for this bias.

In this scenario, we want a distribution centered at -4 but which only considers values less than -4. One natural problem with importance sampling is with random normal generation, some numbers fall outside your intended bounds. Because we're dealing with a normal distribution, the easiest way to fix this is to take the absolute value – or in this case, the negative absolute value – so that you don't sample unnecessary numbers. To do this, we first sample 10,000 numbers from a normal distribution, then multiple the absolute value of those numbers by -1, before subtracting four. This gives us a half-normal distribution centered around -4.

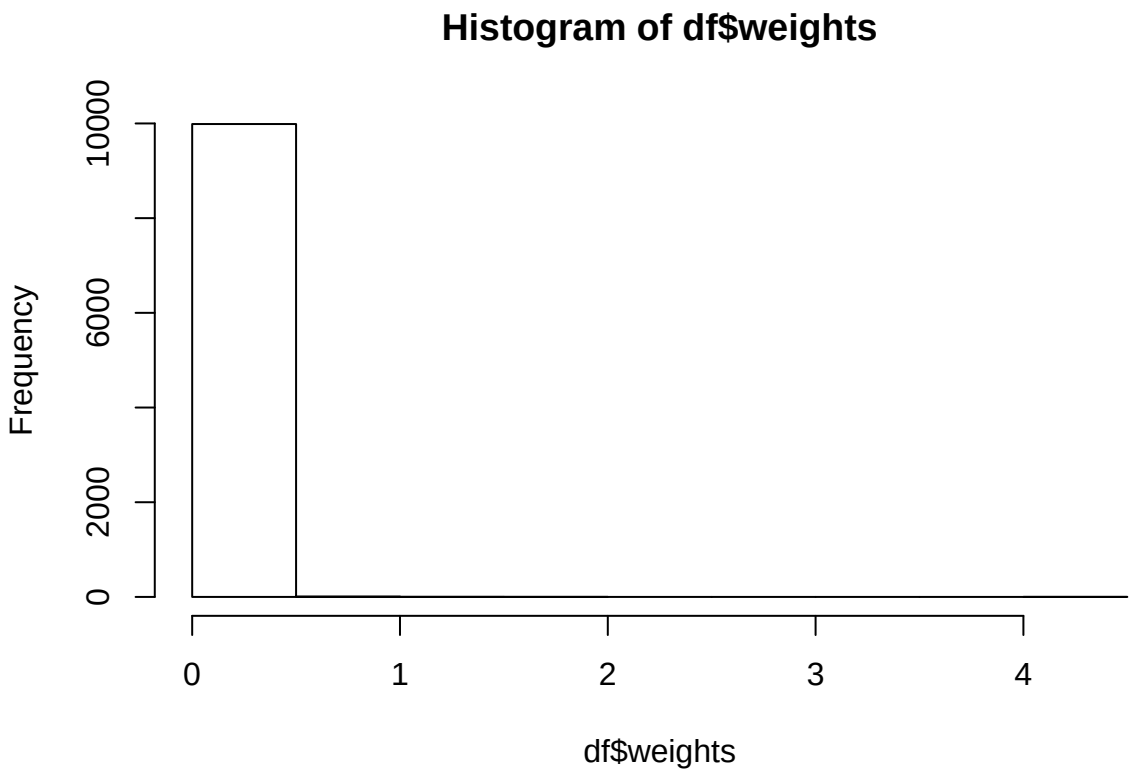
```
set.seed(1056)

## Truncated distribution centered around -4
x <- (-1 * abs(rnorm(10000))) - 4

## PDF of desired (truncated) distribution
f <- dtrunc(x, spec = 't', df = 3)

## PDF of sample distribution
g <- dnorm(x + 4)

## Likelihood ratio
likelihood <- f / g

## Dataframe of weights and weighted values
df <- data.frame(weights = likelihood, values = x * likelihood)

## Mean of weights
mean(df$values)
```

```
## [1] -0.1267118
```

```
## Variance of weights
var(df$values) / length(df$values)
```

```
## [1] 1.955076e-05
```

```
## Histogram of weights
hist(df$weights)
```




Our estimated variance here is shown above is -0.1267 - the mean of the weighted values. As seen in the above histogram of the weights themselves, we have a few weights that appear to be small outliers even though the vast majority of observations are less than one. We cannot see these outliers in the above histogram visually but the long axis tells us they are there. The maximum weight is around 4 - certainly an outlier but not having a huge amount of influence. This variance estimator appears to be performing well.

### Part B

Next, we perform the same computations as above, but instead of sampling from a normal half-distribution, we will sample from a t-distribution with one degree of freedom, centered at -4, and truncated.

```
set.seed(243)

## Truncated distribution centered around -4
x <- (-1 * abs(rnorm(10000))) - 4

## PDF of desired (truncated) distribution
f <- dtrunc(x, spec = 't', df = 3)

## PDF of sample distribution
g <- dt(x + 4, df = 1)

## Likelihood ratio
likelihood <- f / g

## Dataframe of weights and weighted values
df <- data.frame(weights = likelihood, values = x * likelihood)

## Mean of weights
```

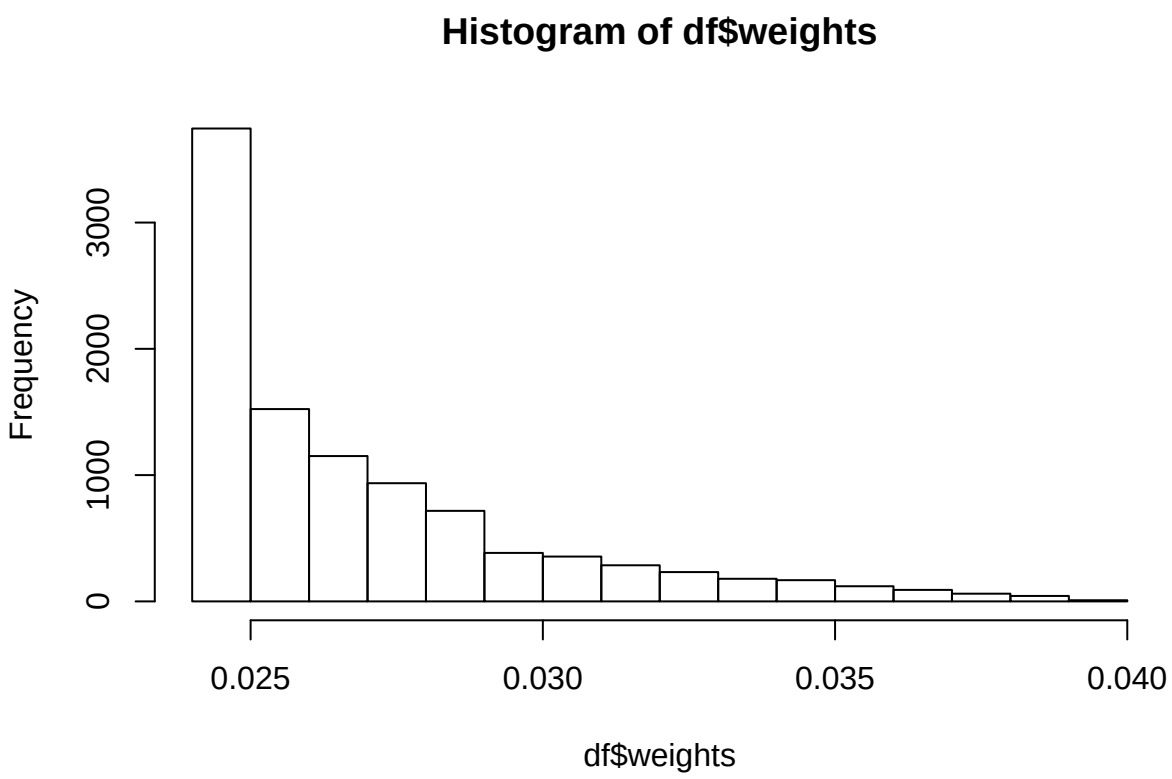
```r
mean(df$values)
```

```
## [1] -0.1307436
```

```r
## Variance of weights
var(df$values) / length(df$values)
```

```
## [1] 1.06756e-07
```

```r
## Histogram of weights
hist(df$weights)
```



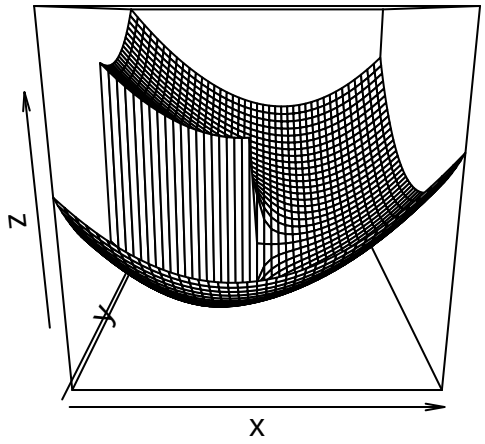

When sampling from this distribution, we calculate a mean of -0.1307. As the histogram of the weights here shows, however, we have no outlier. In fact, our max value is only around 0.4. This shows that, when sampling from the t-distribution, we have no particularly significant weights on the mean estimator.

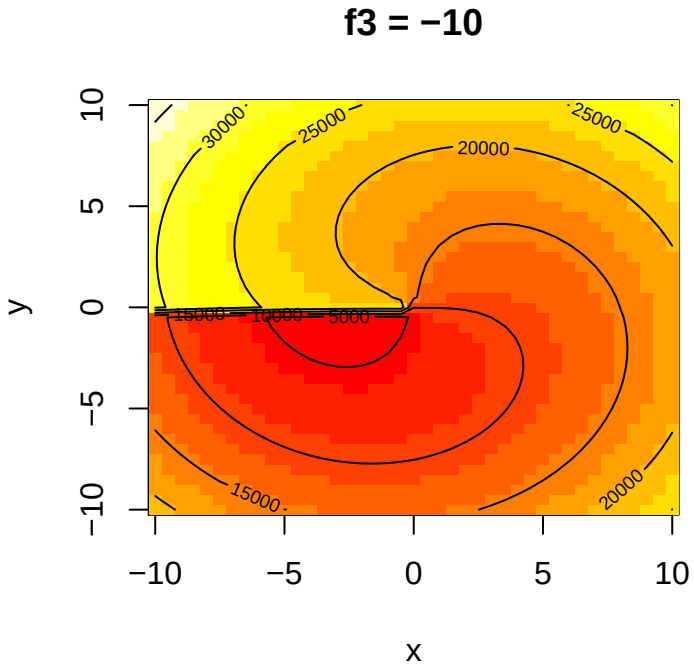
## Question 2

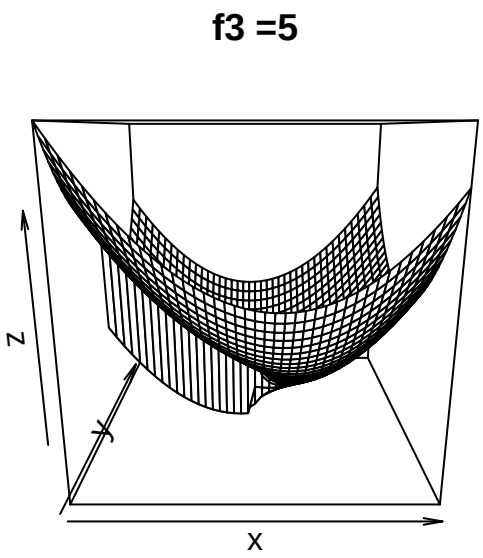
This question asks us to consider the "helical valley" function, which has been recreated below:

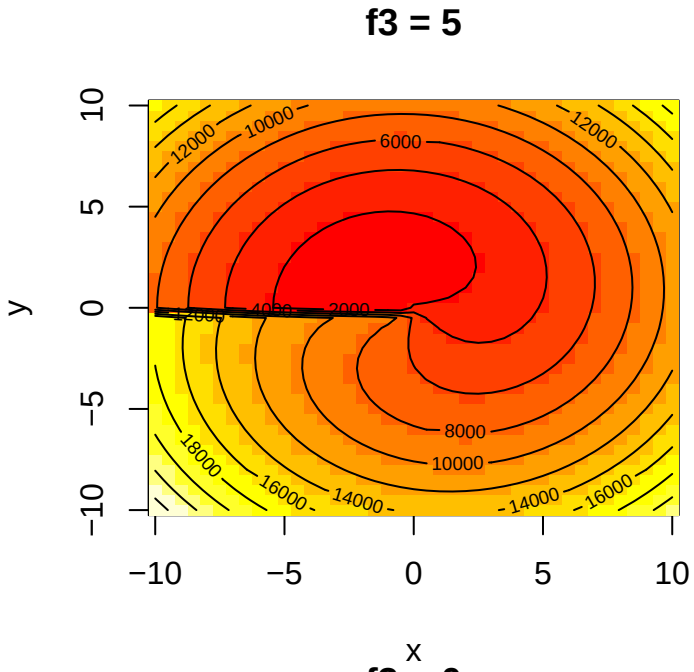
```r
## Code provided by Chris
theta <- function(x1,x2) atan2(x2, x1)/(2*pi)

f <- function(x) {
  f1 <- 10*(x[3] - 10*theta(x[1],x[2]))
  f2 <- 10*(sqrt(x[1]^2 + x[2]^2) - 1)
  f3 <- x[3]
  return(f1^2 + f2^2 + f3^2)
}

## Grabbing dimensions of plot
```

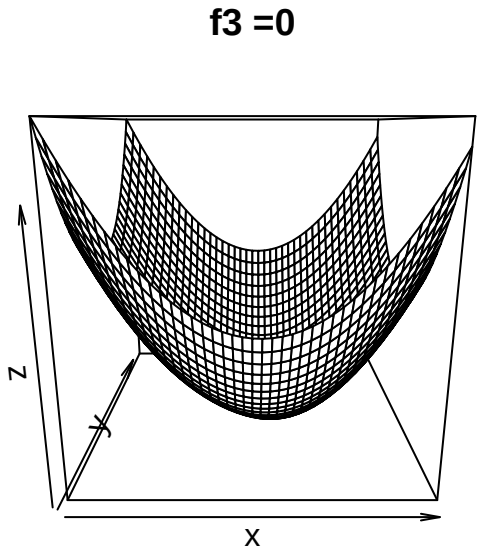
```r
x <- y <- seq(-10, 10, 0.5)
Dim <- length(x)

# Grabbing z values on which to plot function
z_vals <- c(-10, 5, 0, 5, 10)


for (i in 1:length(z_vals)) {

  #using paste0 to assign f1, f2, f3 with an apply
  assign(paste0('f', i), apply(as.matrix(expand.grid(x, y)), 1, function(x) f(c(x, z_vals[i]))))
  # Grab z values
  z <- matrix(get(paste0('f', i)), Dim, Dim)
  #plot the slices in 3D
  persp(x, y, z, main=paste0('f3 =', z_vals[i]))
  #plot the slices in 2D
  image(x, y, z, main=paste0('f3 = ', z_vals[i]))
  #add contour
  contour(x, y, z, add=TRUE)
}
```

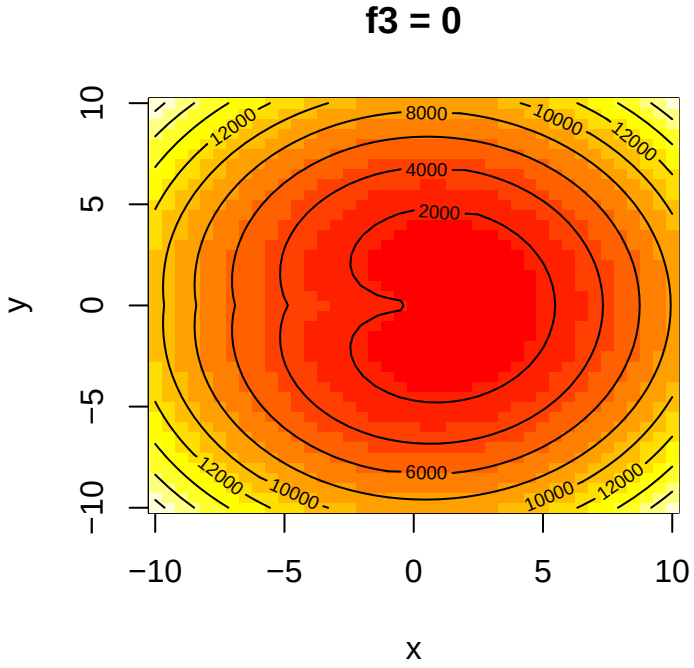


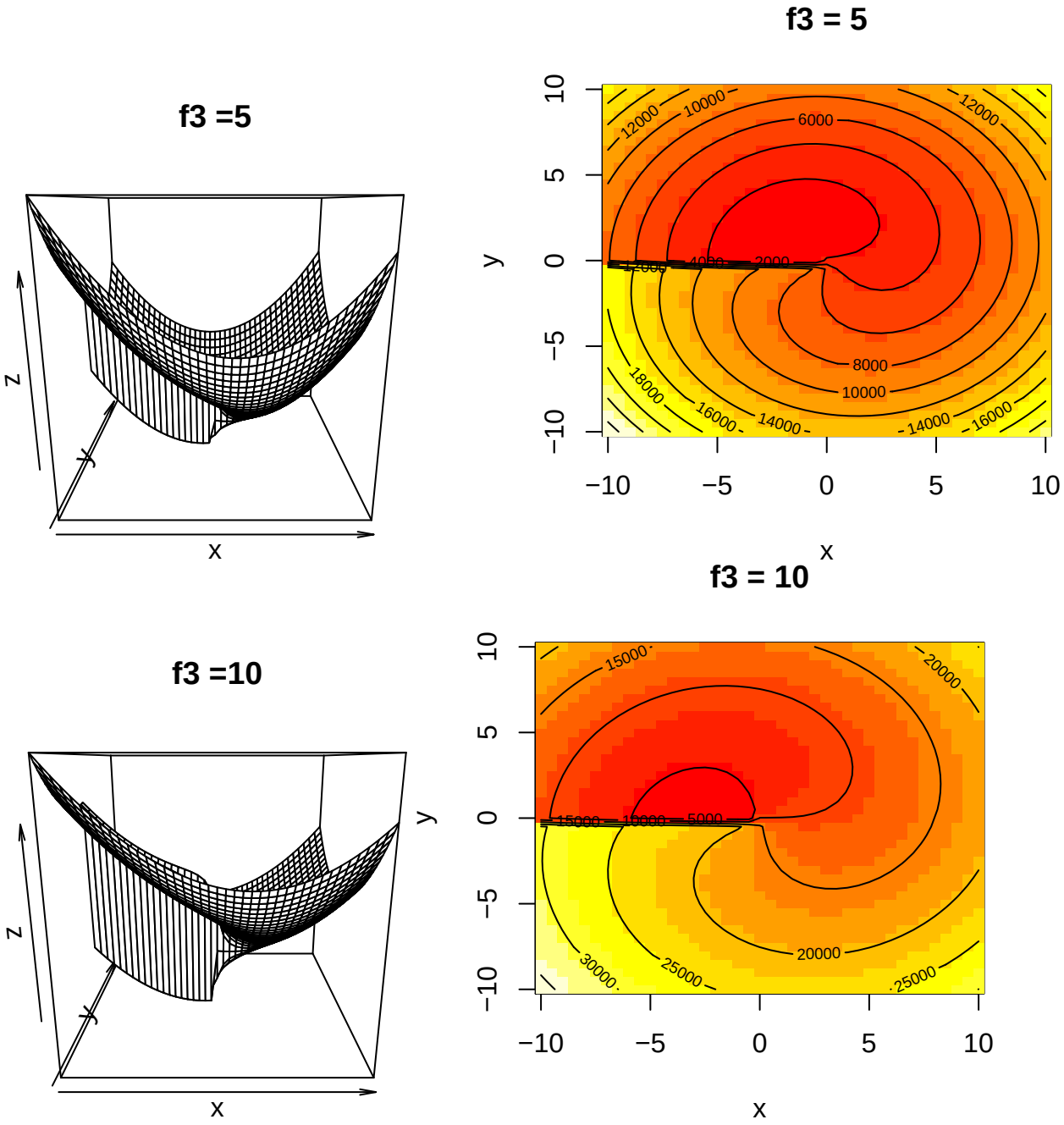




f3 =5
z
y
x


f3 = 5
y
x
10
5
0
−5
−10
−10
−5
0
5
10
12000
10000
6000
2000
8000
18000
16000
14000


f3 =0
z
y
x


f3 = 0
y
x
10
5
0
−5
−10
−10
−5
0
5
10
8000
12000
4000
2000
6000
10000




From these images, we can see why it's called the "helical valley" function. At more extreme but constant values for z, the graphs above show that the function produces an image that looks like a helix except for the valley that arises on one side of the values. The 2D graphs at the more extreme values highlight that the jump in values occurs close to (0, 0, 0).

Next, I tried to find the minimum analytically. I explored this via the optim and nlm functions using random points. I chose a handful of random values on which to test both functions, which can be seen below. I ran optim and nlm on these points and then printed the respective minima, which can be seen below:

```
## Points to test optim() and nlm()
points <- list(c(0, 0, 0), c(5, 5, 5), c(0, 5, 10), c(10, -7, 3), c(4, 0, 2))

## Optim minima
```

```
for (i in seq(length(points))) {
  print(optim(points[[i]], f)$value)
}
```

```
## [1] 1.876851e-05
## [1] 9.451088e-05
## [1] 0.0003191456
## [1] 9.139753e-05
## [1] 9.718798e-06
```

```
## nlm minima
for (i in seq(length(points))) {
  print(nlm(f, points[[i]])$minimum)
}
```

```
## [1] 100
## [1] 4.149798e-15
## [1] 3.422816e-17
## [1] 1.700815e-08
## [1] 1.701011e-08
```

From these values given, the smallest minima we find is at (0, 0, 0). This corresponds with our cork screw graph where the value appears to jump somewhere around 0. To further investigate this, I then used the same functions to report the local minimum points at the same points used above:

```
## Optim minimum points
for (i in seq(length(points))) {
  print(optim(points[[i]], f)$par)
}
```

```
## [1] 0.999978292 0.002730698 0.004284640
## [1] 1.0004955381 0.0008258703 0.0021225637
## [1] 1.00038008 0.01040411 0.01692416
## [1] 0.999316429 0.004251985 0.006634742
## [1]  0.999862236 -0.001759335 -0.002741505
```

```
## nlm minimum points
for (i in seq(length(points))) {
  print(nlm(f, points[[i]])$estimate)
}
```

```
## [1] 0 0 0
## [1]  1.000000e+00 -3.375500e-09 -7.051085e-10
## [1]  1.000000e+00 -3.221717e-10 -3.348527e-10
## [1]  0.9999994972 -0.0000822293 -0.0001300767
## [1]  9.999995e-01 -8.223454e-05 -1.300843e-04
```

Notwithstanding (0, 0, 0), the four points fed into optim and nlm reveal that the minimum point is actually (1, 0, 0), with the slight misrepresentations occurring because of how R handles and displays numbers. Because optim and nlm at each starting point produced the same minimum, it suggests that we are not dealing with any local minima.

## Question 3

This question is above my head mathematically - I don't even know where to being. It feels so above my head that, honestly, I'm embarrassed to come into office hours to discuss it. In lieu of doing the actual math, I wrote an explanation below of EM - what it is both conceptually and (as best I can) mathematically. I know

this isn't what was asked for and I understand that it won't get me any points but I wanted to demonstrate that I took the time to understand the problem and really attempted it but didn't have the technical ability to do what was asked.

EM is the Expectation Maximization algorithm. The algorithm helps us calculate maximum likelihood estimators (MLEs) for a distribution when we have latent variables (i.e. missing data) from that distribution. Given a random variable Y, and a unknown parameter vector theta, we maximize the likelihood function p(y|theta) with respect to theta. Because the data is missing, there is usually not a solution per se to a problem that calls for EM. Instead, EM provides a numerical approximation of the MLE by maximizing local approximation of the likelihood function in an iterative manner. Over many iterations, this will lead to an optimal likelihood function.

Mathematically, EM makes use two steps: The Expectation step, and the Maximization step. In the expectation step, we approximate the likelihood function. The auxiliary function we form here is Q(theta | theta sub n), which is the log likelihood expectation given our current. The maximization step, as it name implies, maximizes the values at the current auxiliary function. Once these values are maximized at that function, we then return to the E step where we will calculate a new auxiliary function based on the new values that have been maxed out from the last auxiliary function. This process repeats until the values converge and we aren't seeing much change in the density of our variable of interest.