Problem 1 (Cilia and Temporal Filtering).

Solution. (a)

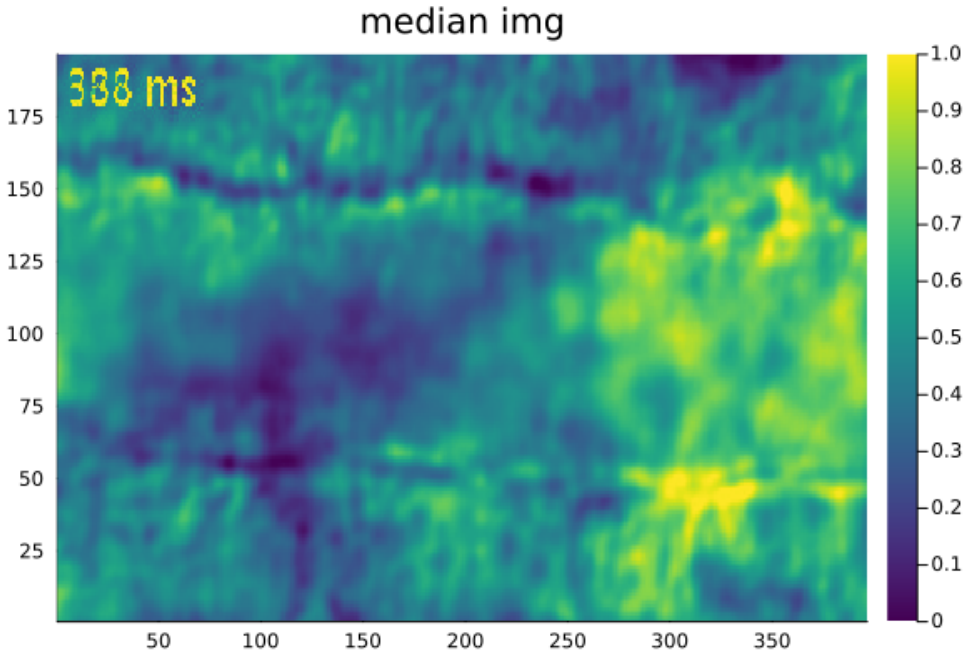


Figure 1: The median image calculated across the temporal dimension.

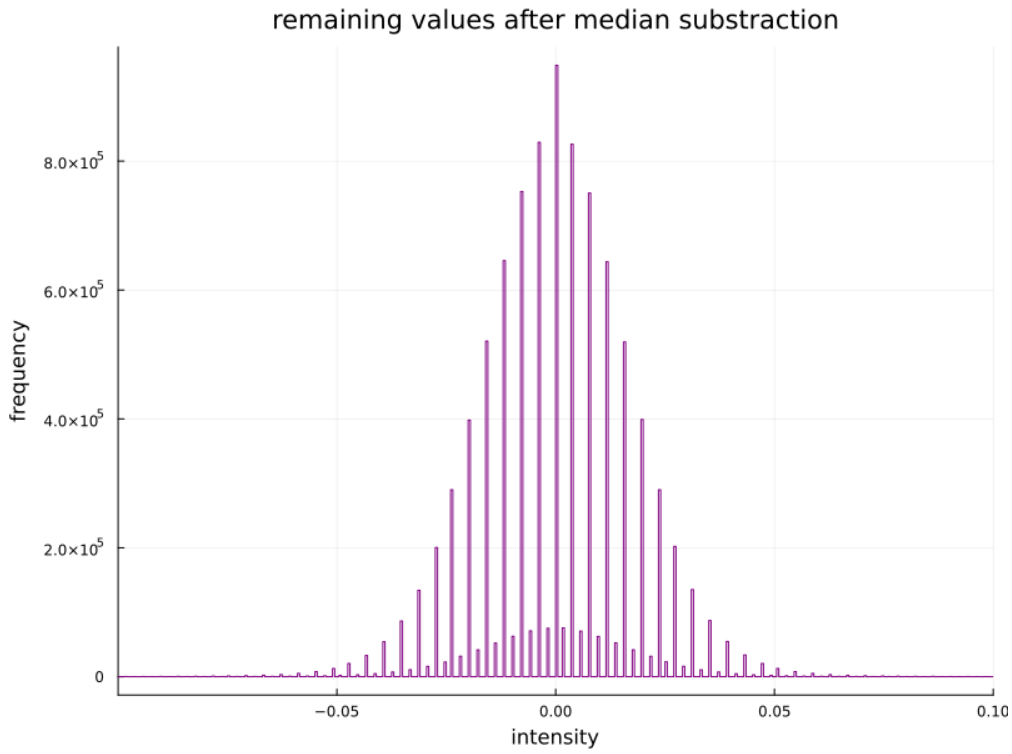
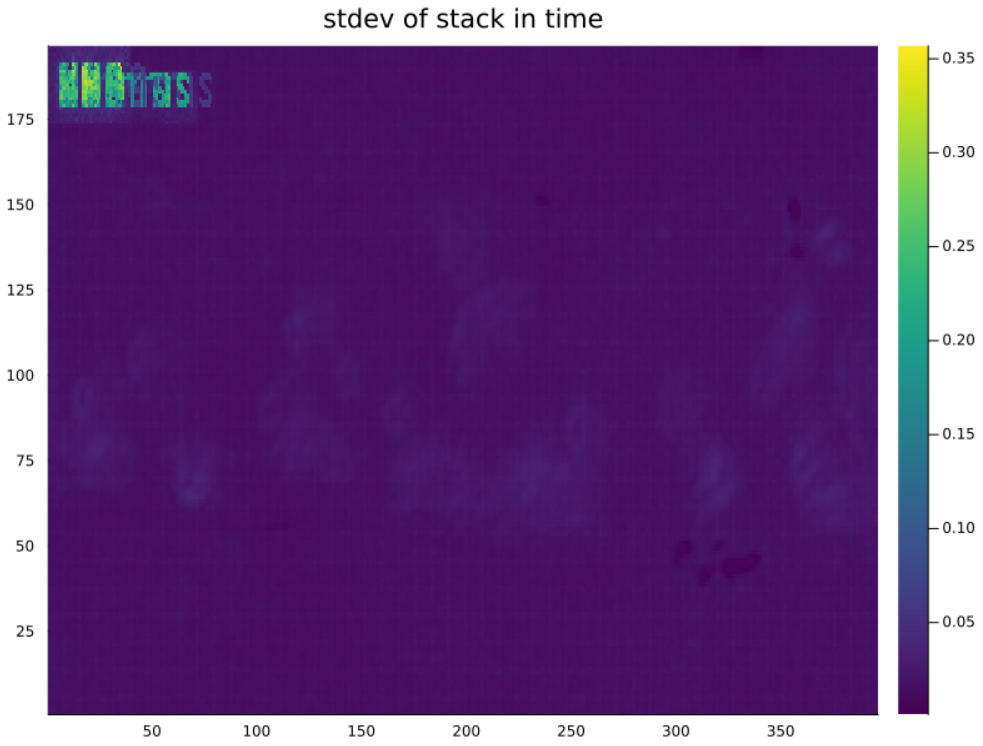


Figure 2: Histogram of pixel values after median subtraction. As expected, they crowd around zero.

(b) I chose the range $[-0.05, 0.05]$. Even though some intensities do fall outside this range, they're minuscule compared to the intensity of the other values. The tiff is attached with the assignment on Canvas.

(c)



(d) I used a Gaussian convolution ($\sigma = 2$) to try and smooth out the noise and possibly see the cilia better. Tiff submitted along with assignment as `cilia_sub_gauss_Bret.tif`.

(e) The cilia do not appear to move back and forth. This wouldn't make sense as anything they're pushing around would just get pushed back and forth. Instead, they have a motion that pushes things along, then sort of 'curls under', reverts to its original position, then repeats.

Problem 2 (MLE Practice).

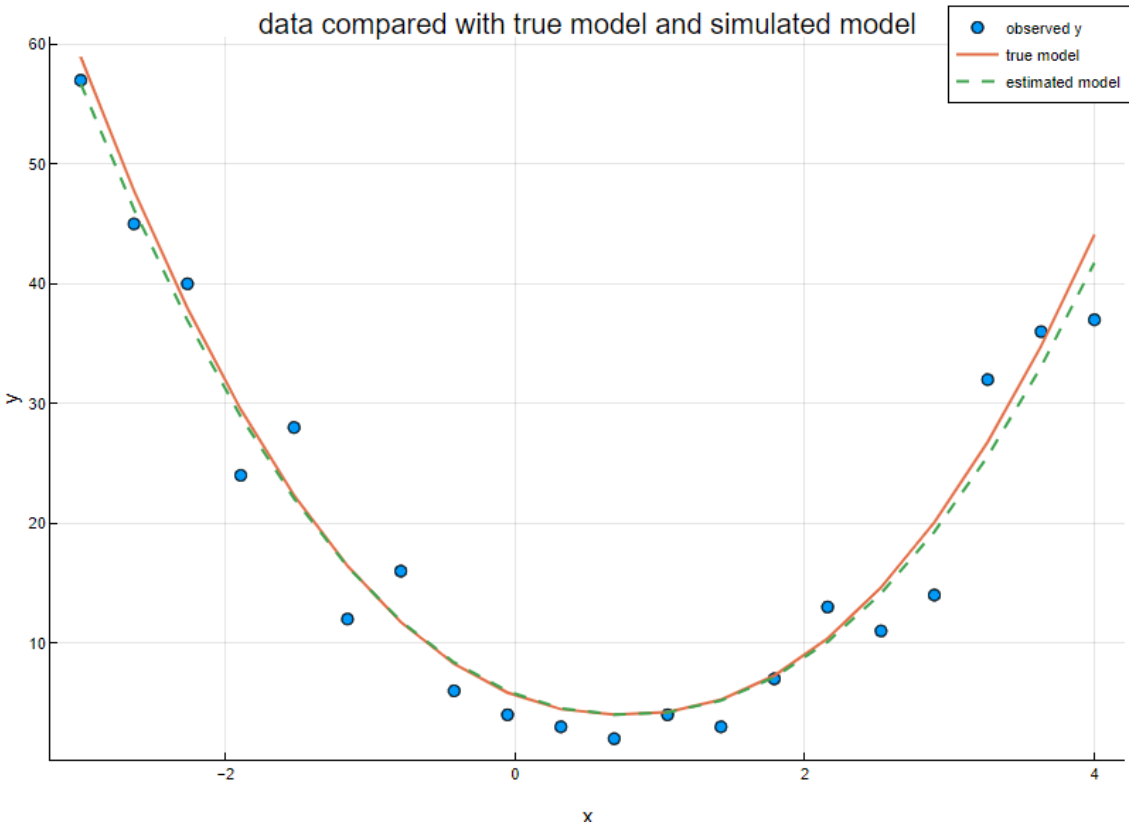
Solution.

```
# We want parameters xₒ and b to maximize likelihood of observing y
function mle_obj(params, x, y)
    xₒ, b = params
    A = A_calc(x, xₒ, b)

    # fine negative log likelihood
    log_like = sum(A .- y .* log.(A))

    return log_like
end; ✓

function A_calc(x, xₒ, b)
    A = 3 * abs.(x .- xₒ).^b .+ 4

    return A
end  A_calc (generic function with 1 method)

# Define the true values
xₒ, b = 0.75, 2.2; ✓

# Set up parameters
N = 20; ✓
x = range(-3, 4, length=N); ✓

A = A_calc(x, xₒ, b); # true values A ✓

y = []; # Need an array for y values based on A ✓
for i = 1:1:N
    push!(y, rand(Poisson(A[i])))
end ✓

############################
# A lot of the above could probably be refactored
# Now to actually try and optimize

guess = [0.5, 2.0]; ✓
result = optimize(p -> mle_obj(p, x, y), guess, BFGS())  * Status: success
# BFGS = Broyden-Fletcher-Goldfarb-Shanno
# This algorithm uses gradient descents (I may code this by hand later)

# Run the optimizer from Optim package
xₒ_est, b_est = Optim.minimizer(result)  2-element Vector{Float64}:
```

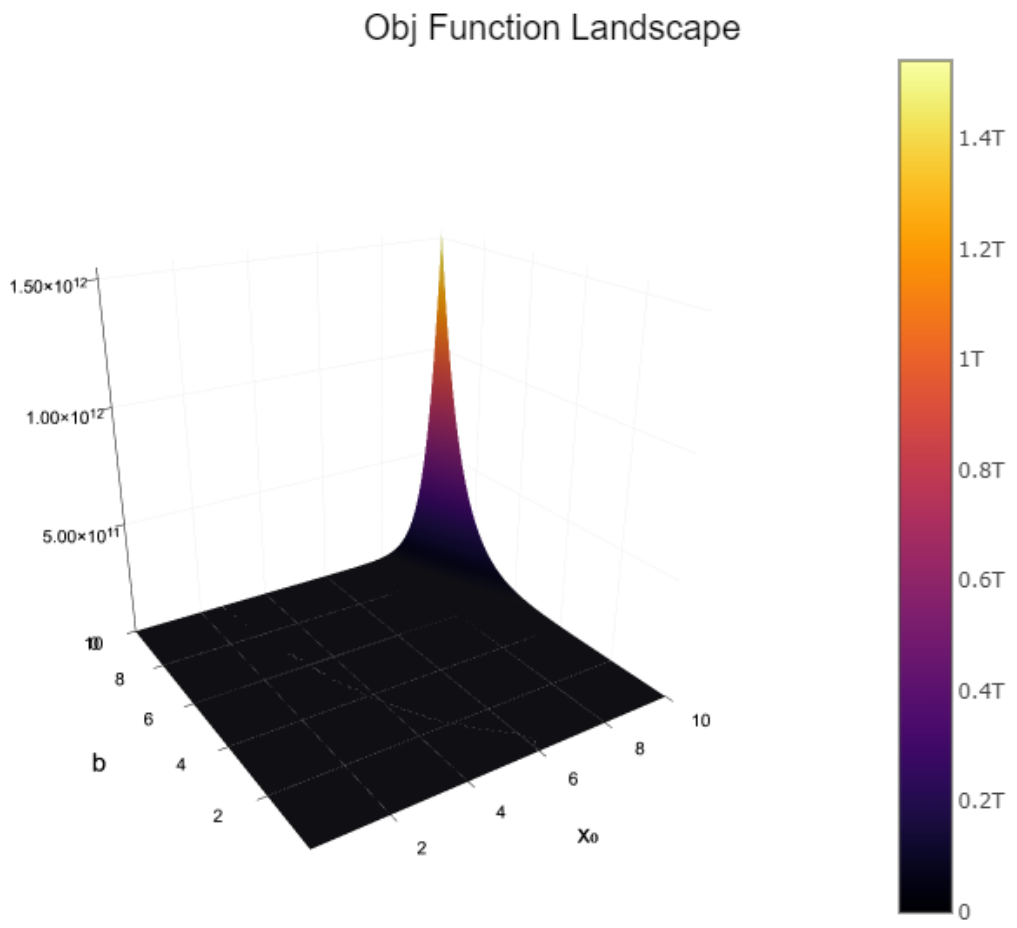
(a) For initial values, I chose 0.5 and 2.0 for x_0 and b respectively. I asked about this in class, but realistically we should know *something* about our true image so we can make a reasonable guess as to what the real values are. Since we know the true values, guessing around them should be fine.

(b) My estimates came out to ~ 0.77 and ~ 2.16 which are pretty close, but close is relative to what we're trying to calculate and could mean anything. The fit in part (c) does better at showing how close we are.

(c)



(d)


Obj Function Landscape
1.50×10^12
1.00×10^12
5.00×10^11
b
2
4
6
8
10
X0
2
4
6
8
10
1.4T
1.2T
1T
0.8T
0.6T
0.4T
0.2T
0

Problem 3 (Centroid localization timing).

Solution. On my work computer, this took roughly 12ms for all images or roughly 0.12ms each. On my laptop (unplugged), this took roughly 0.0065s each.

Problem 4 (Gaussian MLE for particle localization!).

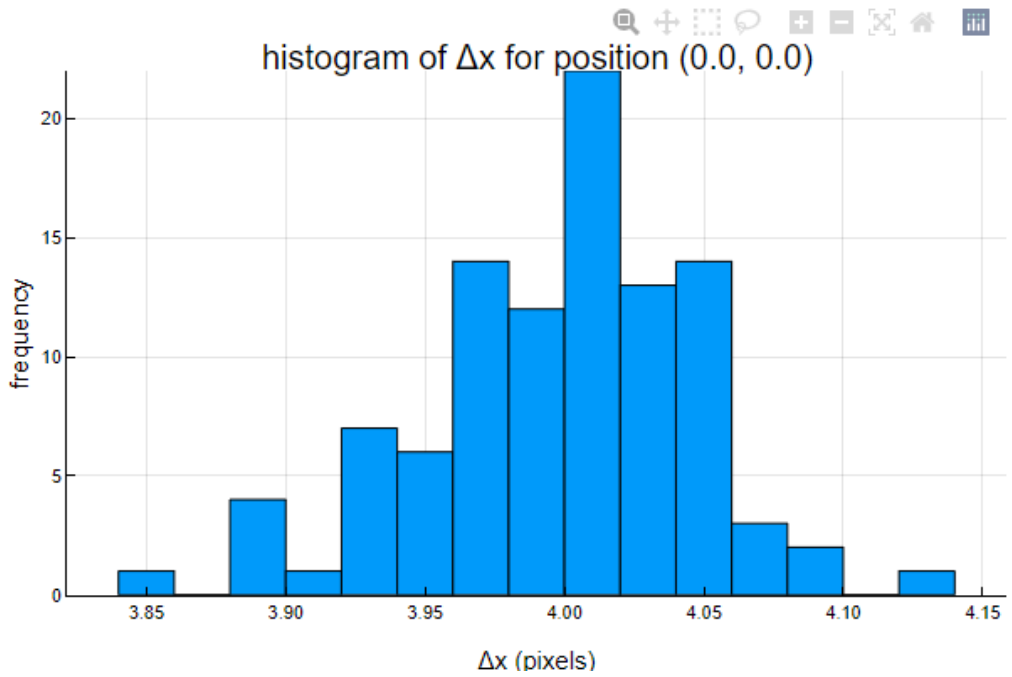
Solution. (a)

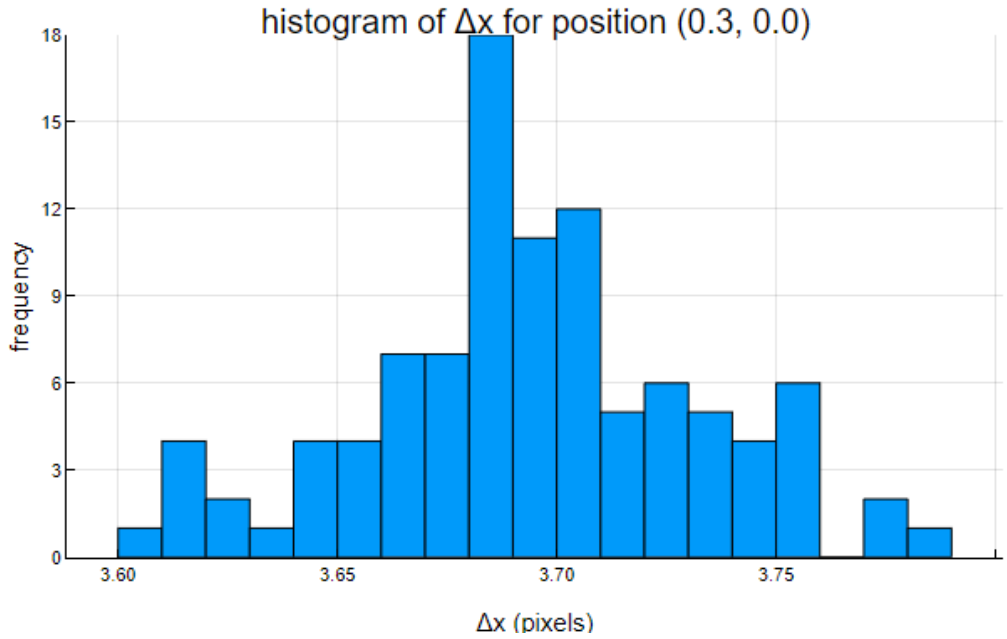
```
function gauss_psf(x, y, A₀, xc, yc, σ, B)
    A = A₀ * exp(-((x - xc)^2 + (y - yc)^2) / (2 * σ^2)) + B

    return A
end; ✓

function neg_log(params, img, x_grid, y_grid)
    A₀, xc, yc, σ, B = params

    # calculate expected intensities for each pixel
    exp_Is = []
    exp_Is = gauss_psf.(x_grid, y_grid, A₀, xc, yc, σ, B)
    # for i = 1:1:length(x_grid), j = 1:1:length(y_grid)
    #     push!(exp_Is, gauss_psf(x_grid[i], y_grid[j], A₀, xc, yc, σ, B))
    # end

    # negative log-likelihood assuming Poisson noise
    log_likelihood = sum(exp_Is .- img .* log.(exp_Is))

    return log_likelihood
end; ✓
```

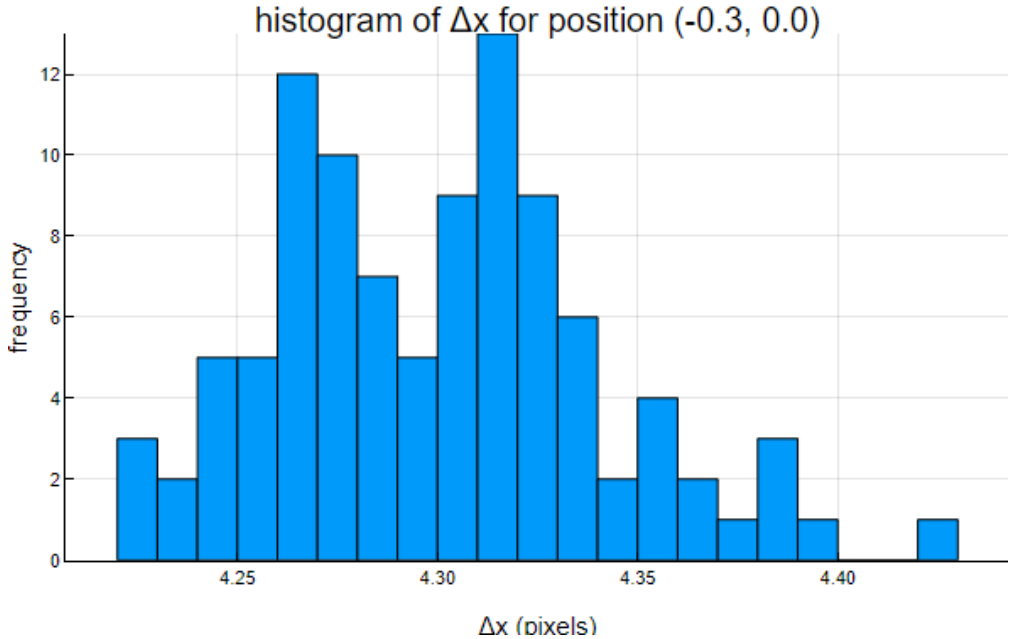
```julia
function mle_localization(img)
    # initial guess
    init_params = [maximum(img), size(img, 1) / 2, size(img, 2) / 2, 1.0, minimum(img)]
    #=  maximum(img) - guess for Aₒ is maximum of the image
        sizes - reasonable to guess center
        1.0 - suppose that the standard deviation could be 1.0
        minimum(img) - guess for B which is background noise =#

    # create dang grids
    rows, cols = size(img)
    x_grid = repeat(collect(1:cols)', rows, 1)
    y_grid = repeat(collect(1:rows), 1, cols)

    # run optimization using Optim packagee
    result = optimize(params -> neg_log(params, img, x_grid, y_grid), init_params)

    # extract the optimized parameters
    Aₒ, xc, yc, σ, B = Optim.minimizer(result)

    return (xc, yc, Aₒ, σ, B)
end; ✓

# we already have simulated images so lets try the mle_localization on
# that array so that the comparison is logical
mle_results = [] | Any[]
@time begin
    for i = 1:1:length(sim_imgs)
        push!(mle_results, mle_localization(sim_imgs[i]))
    end
end ✓
```

(b) This took roughly 1.83s for all 100 or about 0.0183s each.

(c) The MLE estimate looks *somewhat* unbiased but not as much as before. Maybe with 100000 photons we would see this more but with only 1000 it's hard to say for sure, especially with the $(-0.3, 0.0)$. I tried this again with different bin sizes and that made it look more biased but that could also be due to grouping rather than interpreting actual data. Here I have 20 bins.




histogram of Δx for position (0.3, 0.0)
frequency
Δx (pixels)
0
3
6
9
12
15
18
3.60
3.65
3.70
3.75


histogram of Δx for position (-0.3, 0.0)
frequency
Δx (pixels)
0
2
4
6
8
10
12
4.25
4.30
4.35
4.40

(d) It looks like as the position increases, the mean error also increases, however there's a somewhat linear sweet spot which makes me think there are values for p which can keep the error under control. I struggled with this one a lot programmatically because out of nowhere I started getting domain errors and almost called it. The domain errors were related to the log function (if that wasn't obvious) and I eventually fixed it but that being said, I'm not sure if the plot is what we were supposed to see.