1 Question 1

1.1 (a)

To begin with, the data are loaded in.

```
incomes = DataFrame( CSV.File( "data/usincmoms.csv", delim='\t',header=[:percent,:midpoint] ) )
```

From here, the bar chart is constructed, because of the strange shape required in the final bins, these must be constructed manually, then passed to the plotting library as a bar chart.

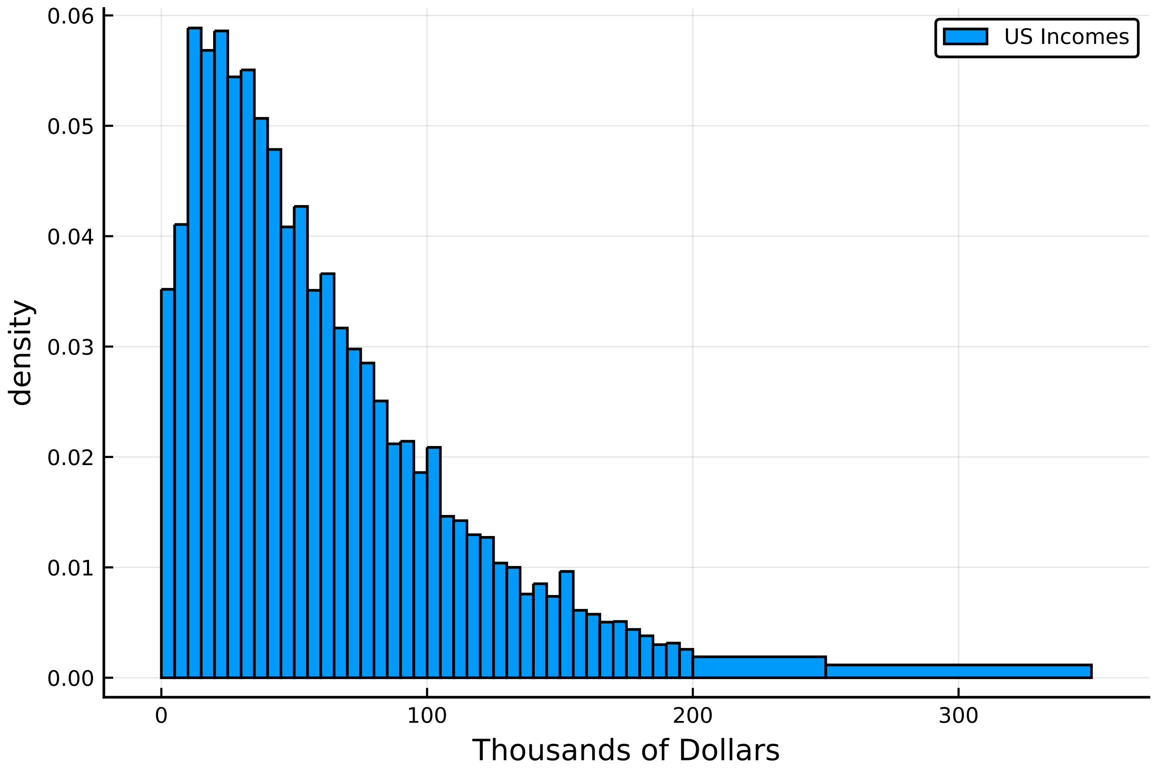
```
heights = copy(incomes[:percent])
heights[41] /= 10.0
heights[42] /= 20.0

bins = Vector{Int64}(undef,length(incomes[:midpoint])+1)
bins[1] = 0
for i in 1:length(incomes[:midpoint])
    bins[i+1] = bins[i] + ((incomes[:midpoint][i]) - bins[i])*2
end

plotBins = copy(bins)
plotBins ./= 1000.0

plot( plotBins, heights, bins = bins, seriestype=:barbins, xlabel="Thousands of Dollars", label="US Incomes",
↪  ylabel="density")

savefig("histogram.pdf")
```



We can see that there is a tail to this distribution, though not nearly as long as in the health distribution.

1.2 (b)

We wish to fit the log-normal distribution to the data, using the mass in each of the bins as the moment criterion. As such, this optimization can be understood as the question of:

$$\begin{aligned} \min_{\mu,\sigma} \quad & e'We \\ \text{s.t.} \quad & e_i = \int_{b_{i-1}}^{b_i} dF(\mu,\sigma) - m_i \end{aligned}$$

Where W is a given weighting matrix, b_i are the cut-off points for the bins, $b_0 = 0$, and m_i are the moments recorded in the data. This is implemented below in full distribution generality, and then applied for the specific Log-Normal Distribution.

```julia
container = Vector{Float64}(undef,42)


function GenerateMoments( container::Vector{Float64},bins::Vector{Float64},
                                distribution, params::Vector)
    cdfs = [cdf( distribution(params...), x) for x in bins]

    for i in 1:42
        container[i] = cdfs[i+1] - cdfs[i]
    end
    return container
end

W = convert( Matrix{Float64}, Diagonal(convert(Vector{Float64}, incomes[:percent])) )

function GMMCrit( W::Matrix{Float64}, dataMoments::Vector{Float64},
                  container::Vector{Float64}, bins::Vector{Float64},
                  distributions, params::Vector)
    e = GenerateMoments( container, bins, distributions, params ) - dataMoments
    return dot(e, W*e)#e'*W*e
end
```

The question remains as to what should be used as a starting point for our optimization algorithm. To this end, I simulate from the multinomial distribution defined by the bins, and calculate the Method-of-Moments estimators for the log-normal distribution from this simulated data. These estimators take the form of:

$$\widehat{\sigma}_n^2 = \log \frac{s^2}{\overline{x}_N + 1}$$
$$\widehat{\mu}_n = \log \overline{x}_N - \frac{\widehat{\sigma}_n^2}{2}$$

This is implemented below:

```julia
dataProbs = cumsum( incomes[:percent])
dataProbs[length(dataProbs)] = 1.0
N = 100000
M = length(dataProbs)
simulation = rand( Uniform(), N)
for i in 1:N
```

```
    for j in 1:M
        if( simulation[i] < dataProbs[j])
            simulation[i] = incomes[:midpoint][j]
            break
        end
    end
end

simMean = mean(simulation)
simVar = var(simulation)

sigGuess = log( simVar / (simMean*simMean) + 1)
muGuess = log(simMean) - sigGuess / 2.0
```

Using these points as the starting points for our optimization algorithm, we can then compute the minimizers for the GMM criterion function.

```
dataStuff = convert(Vector{Float64}, incomes[:percent])
cdfBins = convert(Vector{Float64}, bins)

θ = [muGuess, sqrt(sigGuess)]
fun(x::Vector) = GMMCrit( W, dataStuff, container, cdfBins, LogNormal, x  )

logResult = optimize( fun, θ, BFGS())
```

The results from the optimization are printed below:

```
Results of Optimization Algorithm
 * Algorithm: BFGS
 * Starting Point: [10.818767925970231,0.7701372401373785]
 * Minimizer: [10.862298881282367,1.022995942443407]
 * Minimum: 3.522920e-05
 * Iterations: 7
 * Convergence: true
   * |x - x'| ≤ 0.0e+00: false
     |x - x'| = 1.54e-05
   * |f(x) - f(x')| ≤ 0.0e+00 |f(x)|: false
     |f(x) - f(x')| = 1.07e-08 |f(x)|
   * |g(x)| ≤ 1.0e-08: true
     |g(x)| = 3.07e-11
   * Stopped by an increasing objective: false
   * Reached Maximum Number of Iterations: false
 * Objective Calls: 27
 * Gradient Calls: 27
```

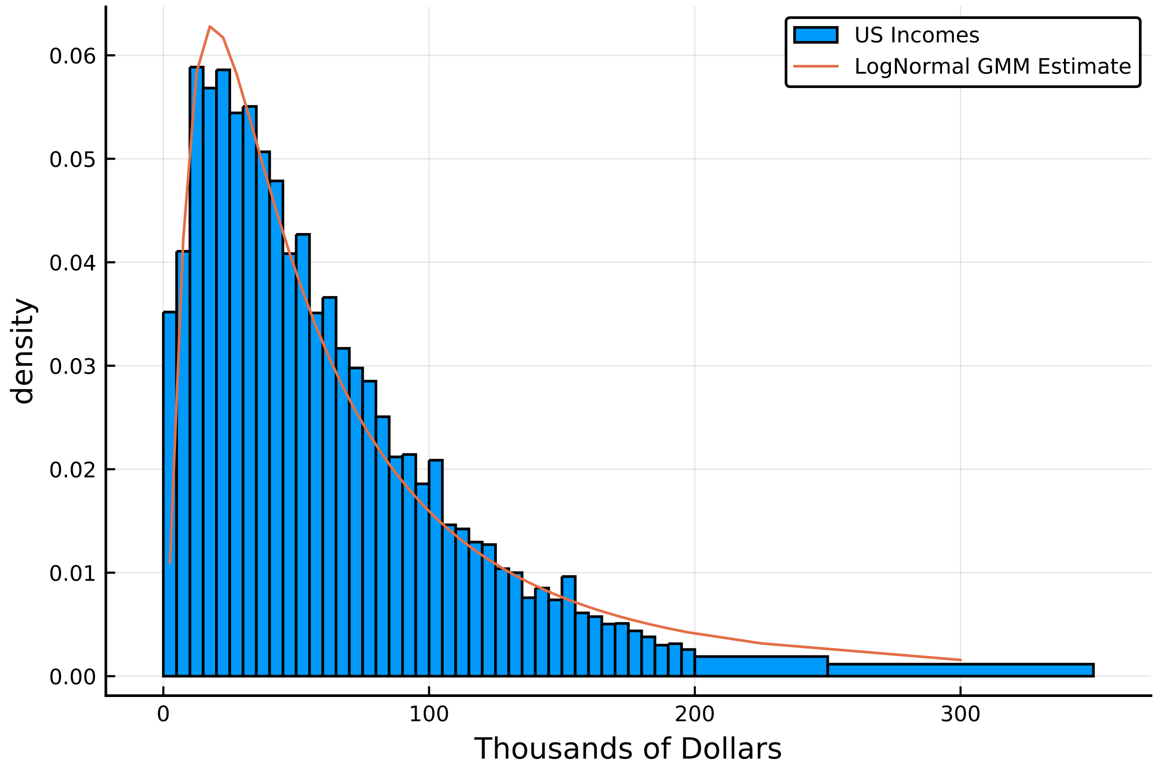
We now overlay the histogram of μ, σ on the bar chart used to display the data from part (a).

```
mu = logResult.minimizer[1]
sigma = logResult.minimizer[2]

estHeightsLoggyBoy = GenerateMoments( container, cdfBins, LogNormal, [mu, sigma] )
```

```
estHeightsLoggyBoy[41] /= 10.0
estHeightsLoggyBoy[42] /= 20.0

plot!(incomes[:midpoint], estHeightsLoggyBoy, label="LogNormal GMM Estimate")
```



1.3 (c)

For the Gamma Distribution, we are given initial values to choose, and can implement our estimation using the same function as before, with simply changing the distribution that is passed.

```
gamContainer = copy(container)
θ = [3.0, 25000.0]
betaFun(x::Vector) = GMMCrit( W, dataStuff, gamContainer, cdfBins, Gamma, x  )
gamResult = optimize( betaFun, θ, BFGS())
```

The results for our optimization algorithm are printed below:

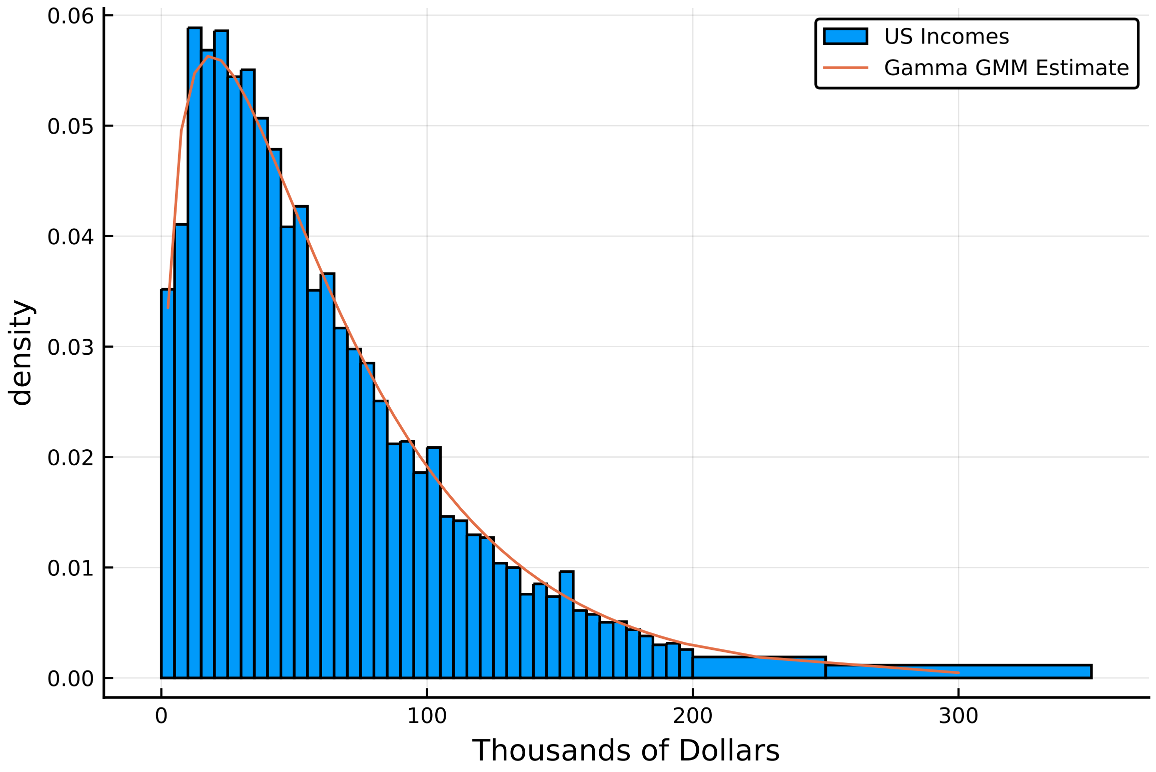
```
Results of Optimization Algorithm
 * Algorithm: BFGS
 * Starting Point: [3.0,25000.0]
 * Minimizer: [1.3984481398133792,46486.004874968334]
 * Minimum: 1.094699e-05
 * Iterations: 13
 * Convergence: true
   * |x - x'| ≤ 0.0e+00: false
```

```
    |x - x'| = 6.37e+00
  * |f(x) - f(x')| ≤ 0.0e+00 |f(x)|: false
    |f(x) - f(x')| = 7.44e-06 |f(x)|
  * |g(x)| ≤ 1.0e-08: true
    |g(x)| = 1.91e-09
  * Stopped by an increasing objective: false
  * Reached Maximum Number of Iterations: false
* Objective Calls: 52
* Gradient Calls: 52
```

The histogram is plotted in the same way as before as well:

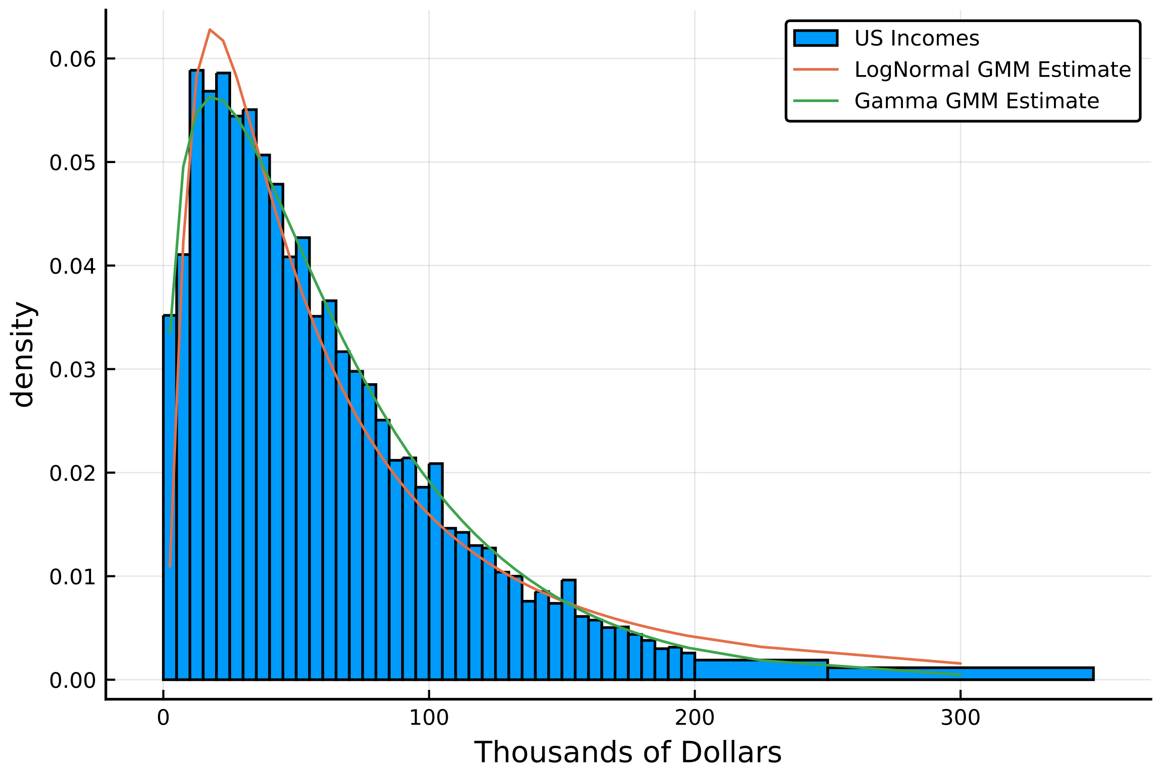
```
estHeightsGammaMan = GenerateMoments( gamContainer, cdfBins, Gamma, [gamAlpha, gamBeta] )
estHeightsGammaMan[41] /= 10.0
estHeightsGammaMan[42] /= 20.0

plot!(incomes[:midpoint] / 1000.0, estHeightsGammaMan, label="Gamma GMM Estimate")
```



1.4 (d)

The overlaid plots are shown below:



The question of the most precise way to tell which distribution fits the data the best is subjective. To define “best” we need to define a norm. The norm that these moments are minimized on is the sample frequency, and based upon this norm, which distribution fits the data best is simply which has a smaller minimum. By this norm we find that the Gamma Distribution is the better fit.

Gamma Minimum	1.09470×10^{-5}
LogNormal Minimum	3.52292×10^{-5}

However, this is not a common norm used when describing the distance between two distributions. More common notions of distance are the total-variation norm, or the kullback-leibler distance. However since the latter is not an actual norm, we will consider the total variation norm, which is commonly used to compute the “distance” between two distributions.

```
sum(abs(x - y) for (x,y) in zip( estHeightsGammaMan, dataStuff)) / 2

sum(abs(x - y) for (x,y) in zip( estHeightsLoggyBoy, dataStuff)) / 2
```

Gamma Norm	0.048159
LogNormal Norm	0.071237

Under this norm, we find that the Gamma fits the data better as well. These are both in line with the visual test, which appears to show the gamma distribution fitting the data better.

1.5 (e)

There is a lot of issues surrounding the construction of this two-step weighting matrix. Ideally, with full access to this data, we would like to construct it using the outer-product of the errors where only a single measurement is supplied to each moment, and summing over all the data. However, we are not given the entire data, and merely the histogram contents.

Construction of such a two-step matrix will then have to be done via simulation. If we believe that are our estimates are actually the truth, then data simulated from these estimates should come from the distribution of the truth. Therefore we can construct a two-step weighting matrix from this simulated data.

```julia
S = 100
simulation = Vector{Vector{Float64}}(undef,S)
simMoments = Matrix{Float64}(undef, S, 42)
simMoments .= 0
for s in 1:S
    simulation[s] = rand(Gamma( gamAlpha, gamBeta), N)
    for i in 1:N
        for j in 1:42
            if( simulation[s][i] >= bins[j] && simulation[s][i] < bins[j+1]  )
                simMoments[s,j] += 1
            end
        end
    end
end

simMoments ./= N

#Theres a problem with this simulating over the upper bound of the
#bins, so we just make it sum to one
for s in 1:S
    simMoments[s,42] = 1.0 - sum(simMoments[s,1:41])
    #simMoments[s,:] = estHeightsGammaMan - simMoments[s,:]
end

simErrors = Matrix{Float64}(undef, S, 42)
for s in 1:S
    simErrors[s,:] = estHeightsGammaMan - simMoments[s,:]
end


F = cholesky(sum( simMoments[s,:]*simMoments[s,:]' for s in 1:S ))
wHat = F.U \ (F.L \ I)
```

Estimation of the Gamma distribution then follows as before:

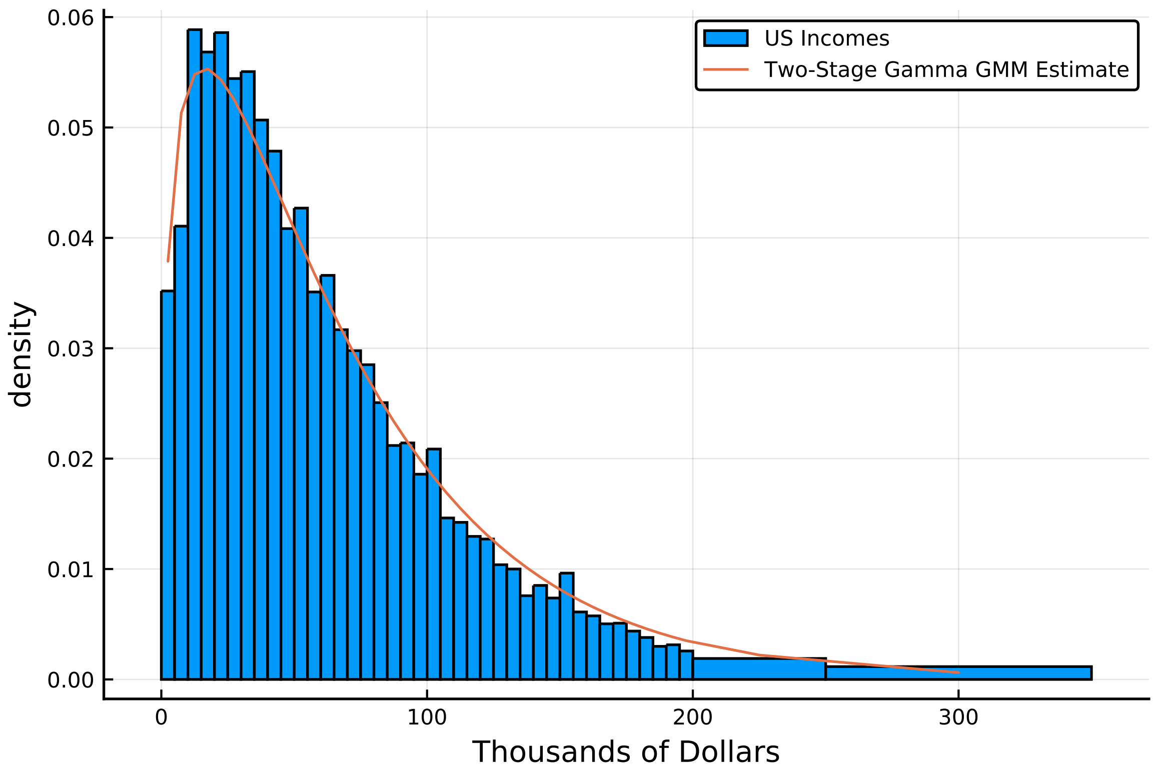
```julia
twoStageContainer = copy(container)
θ = [gamAlpha, gamBeta]
gamFun(x::Vector) = GMMCrit( wHat, dataStuff, twoStageContainer, cdfBins, Gamma, x  )
gamResult = optimize( gamFun, θ, NelderMead())

twoStageAlpha = gamResult.minimizer[1]
twoStageBeta = gamResult.minimizer[2]
```

```
Results of Optimization Algorithm
 * Algorithm: Nelder-Mead
 * Starting Point: [1.3984481398133792,46486.004874968334]
 * Minimizer: [1.3142290928311697,51131.041222344385]
 * Minimum: 3.309378e+01
 * Iterations: 40
 * Convergence: true
   *  √(Σ(yᵢ-ȳ)²)/n < 1.0e-08: true
   * Reached Maximum Number of Iterations: false
 * Objective Calls: 82
```

```
estHeightsTwoStage = GenerateMoments( twoStageContainer, cdfBins, Gamma, [twoStageAlpha, twoStageBeta] )
estHeightsTwoStage[41] /= 10.0
estHeightsTwoStage[42] /= 20.0

plot!(incomes[:midpoint] / 1000.0, estHeightsTwoStage, label="Two-Stage Gamma GMM Estimate")
```



We find that our estimates for α, β do not change by a large amount. However comparing the goodness-of-fit between the two distributions can no longer be done by comparing the minimums between the two distributions. Since the weighting matrix has been changed, the implied norm that we are minimizing with respect to has changed as well. Therefore we will return to the Total-Variation Norm to compute the goodness of fit measure.

```
sum(abs(x - y) for (x,y) in zip( estHeightsTwoStage, dataStuff)) / 2.0
```

We find that the total variation norm for the two-stage estimator to be: 0.056597. This indicates that it fits the data less well than the original Gamma distribution estimate.

2 Question 2

The empirical analogs to our estimates are obtained by the analogy-principle. From the WLLN we know that we can replace expectations with averages, and the results will converge in probability. Following this logic our moment conditions that we are minimizing with respect to are:

$$\frac{1}{T}\sum_{t=1}^{T}\left[z_{t+1} - \rho z_t - (1-\rho)\mu\right] = 0$$

$$\frac{1}{T}\sum_{t=1}^{T}\left[\left(z_{t+1} - \rho z_t - (1-\rho)\mu\right) z_t\right] = 0$$

$$\frac{1}{T}\sum_{t=1}^{T}\left[\beta\alpha\exp(z_{t+1})k_{t+1}^{\alpha-1}\frac{c_t}{c_{t+1}} - 1\right] = 0$$

$$\frac{1}{T}\sum_{t=1}^{T}\left[\left(\beta\alpha\exp(z_{t+1})k_{t+1}^{\alpha-1}\frac{c_t}{c_{t+1}} - 1\right) w_t\right] = 0$$

For our initial guess, we will use the estimates that we reached during Problem Set 2.

```julia
macroData = DataFrame(load("data/MacroSeries.csv", header_exists=false, colnames=["C", "K", "W", "R"]))

function ConstructMoments( moments::Vector{Real}, α::Real, β::Real, ρ::Real,
                           μ::Real, c::Vector{Float64},k::Vector{Float64},
                           w::Vector{Float64}, r::Vector{Float64}, W::Matrix{Real})
    # $r_t - \alpha\exp(z_t)k_t^{\alpha-1} = 0$
    # $\log r_t = \log\alpha + z_t + (\alpha-1)\log k_t$
    # $z_t = \log r_t - \log\alpha - (\alpha-1)\log k_t$
    N = 100
    z = Vector{Real}(undef, N)
    for i in 1:N
        z[i] = log( r[i]) - log(α) - (α - 1.0)*log( k[i] )
    end

    moments[1] = mean( z[i+1]- ρ*z[i] - (1-ρ)*μ for i in 1:99)
    moments[2] = mean( (z[i+1]- ρ*z[i] - (1-ρ)*μ)*z[i] for i in 1:99)
    moments[3] = mean( β * α * exp(z[i+1]) * k[i+1]^(α-1.0) * (c[i]/c[i+1]) - 1 for i in 1:99)
    moments[4] = mean( (β*α*exp(z[i+1])*k[i+1]^(α-1.0) * (c[i]/c[i+1]) - 1)*w[i] for i in 1:99 )

    return sum(moments[i]*moments[i] for i in 1:4)
end

function limitedLogistic( unbounded::Real )
    return ((exp(unbounded)) / ( 1 + exp(unbounded)))*.99 + .005
end

function invertLogistic( x::Real )
    return log( (1.0-200.0*x)/ (200.0*x - 199.0))
end

W = convert(Matrix{Real}, Diagonal([1.0,1.0,1.0,1.0]))

c = convert( Vector{Float64}, macroData[:C] )
w = convert( Vector{Float64}, macroData[:W] )
```

```julia
k = convert( Vector{Float64}, macroData[:K] )
r = convert( Vector{Float64}, macroData[:R] )

mom = Vector{Real}(undef,4)

#Initialize this guy with the stuff we got the first time around
alphaStart = invertLogistic(.70216)
rhoStart = atanh(.47972)
muStart = log(5.0729)
betaStart = invertLogistic(.99)

θ = [alphaStart, betaStart, rhoStart, muStart]

f(x::Vector) = ConstructMoments( mom, limitedLogistic(x[1]), limitedLogistic(x[2]), tanh(x[3]), exp(x[4]), c, k, w, r,
↪  W )

result = optimize( f, θ, NelderMead(), autodiff = :forward, Optim.Options( g_tol = 1e-18))

alphaHat = limitedLogistic( result.minimizer[1])
betaHat = limitedLogistic( result.minimizer[2])
rhoHat = tanh( result.minimizer[3])
muHat = exp( result.minimizer[4])
```

Output from the optimization algorithm is given below:

```
Results of Optimization Algorithm
 * Algorithm: Nelder-Mead
 * Starting Point: [0.8673885548750386,5.2832037287379885, ...]
 * Minimizer: [0.8717159418980703,5.283203728737997, ...]
 * Minimum: 5.248191e-19
 * Iterations: 223
 * Convergence: true
   *  √(Σ(yᵢ-ȳ)²)/n < 1.0e-18: true
   * Reached Maximum Number of Iterations: false
 * Objective Calls: 419
```

However these minimizers are points in $\mathbb{R}^4$, so they must be translated back into the parameter space. The output from this is given below.

Criterion:	5.248191e-19
$\widehat{\alpha}_n$:	0.70305
$\widehat{\beta}_n$:	0.99000
$\widehat{\rho}_n$:	0.47873
$\widehat{\mu}_n$:	5.05862