4741 Bias and Variance
HW 4, Question 4
Aman jain (aj644@cornell.edu)

a)

In [1]:
```
using Plots
using Statistics
```

Suppose we have a sinusoid function

In [2]:
```
f(x) = 10*sin.(x)
```

Out[2]:
```
f (generic function with 1 method)
```

Our dataset D will consist of seven datapoints drawn from the following probabilistic model.

For each datapoint we randomly draw x_i uniformly in [0,6] and observe a noisy $y_i = f(x_i) + \epsilon_i$, where ϵ_i is some noise drawn from a standard normal distribution $\mathcal{N}(0, 1)$.

Generate a sample dataset from this distribution.

In [3]:
```
using Random
d = 6*rand()
d
```

Out[3]:
```
0.6742295554456206
```

In [4]:
```
h = [[1,3],[4,5]]
h[1][1]
```

Out[4]:
```
1
```

In [5]:
```
n = 7
D = zeros(n, 2)
for i = 1:n
    D[i,1] = 6*rand()
    D[i,2] = f(D[i,1])+randn() # yi = f(xi) + ϵi
end
D
```

Out[5]:
```
7×2 Array{Float64,2}:
 5.78846   -5.38629
 5.74309   -5.16613
 4.74145   -9.06159
 1.45705   10.3888
 4.32511  -11.0167
 3.87209   -7.30246
 2.31469    7.70446
```

Plot the dataset $\mathcal{D}$ and the true function $f(x)$.

In [6]:
```
"""plot function y=f(x)"""
function plotfunc_first(f;
    xmin=0,xmax=6,nsamples=1000)
    xsamples = range(xmin,stop=xmax,length=nsamples)
    plot(xsamples, [f(x) for x in xsamples], color="black")  ## only dif
end
```

Out[6]:
```
plotfunc_first
```

In [7]:
```
"""plot function y=f(x)"""
function plotfunc(f;
    xmin=0,xmax=6,nsamples=1000)
    xsamples = range(xmin,stop=xmax,length=nsamples)
    plot!(xsamples, [f(x) for x in xsamples], color="black")
end
```

Out[7]:
```
plotfunc
```

b)

Fit a linear model to $\mathcal{D}$

In [8]:
```
X = [D[:,1] ones(7)]
y = D[:,2]
w = X\y
```

Out[8]:
```
2-element Array{Float64,1}:
 -4.11675745451897
 13.775034178463931
```

In [9]:
```
function l(x)
    y_pred = [x 1]w
    return( y_pred[1])
end
```

Out[9]:
```
l (generic function with 1 method)
```

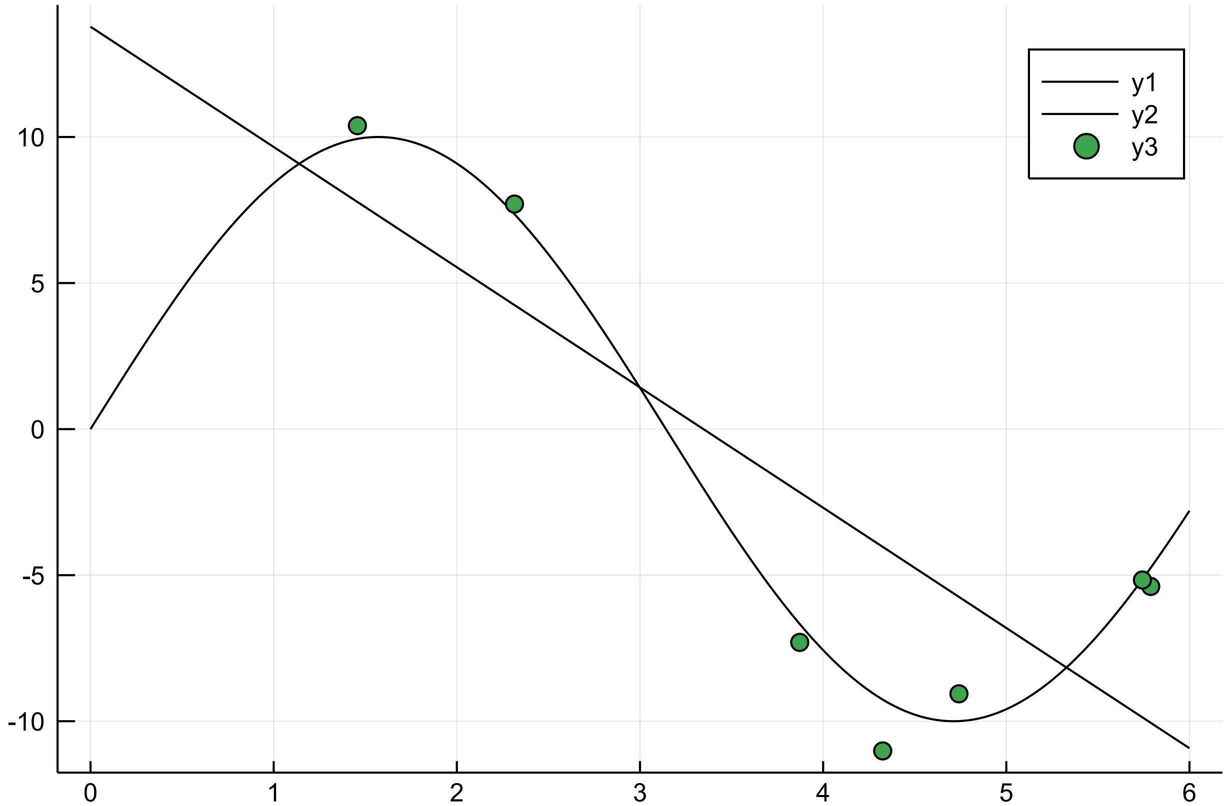
In [10]:
```
l(0.5)
```

Out[10]:
```
11.716655451204446
```

Plot the linear model $l(x)$ together with $\mathcal{D}$ and $f(x)$. Feel free to use our method plotfunc(f).

In [11]:
```
plotfunc_first(l)
plotfunc(f)
scatter!(D[:,1],D[:,2])
```

Out[11]:



c)

Fit a cubic model $c(x)$ to $\mathcal{D}$

In [12]:
```
# first, construct a Vandermonde matrix
max_order = 3
x = D[:,1]
V = zeros(n, max_order+1)
for k=0:max_order
    V[:,k+1] = x.^k
end

# solve least squares problem
w_c = V\D[:,2]
w_c
```

Out[12]:
```
4-element Array{Float64,1}:
  -4.21835236453993
  22.981019642056605
 -10.378989661685075
   1.1040788162896285
```

In [13]:
```
function c(x; order = max_order, w = w_c)
    y = 0
    for k=0:order
        y += w[k+1]*x^k
    end
    return y
end
```

Out[13]:
```
c (generic function with 1 method)
```

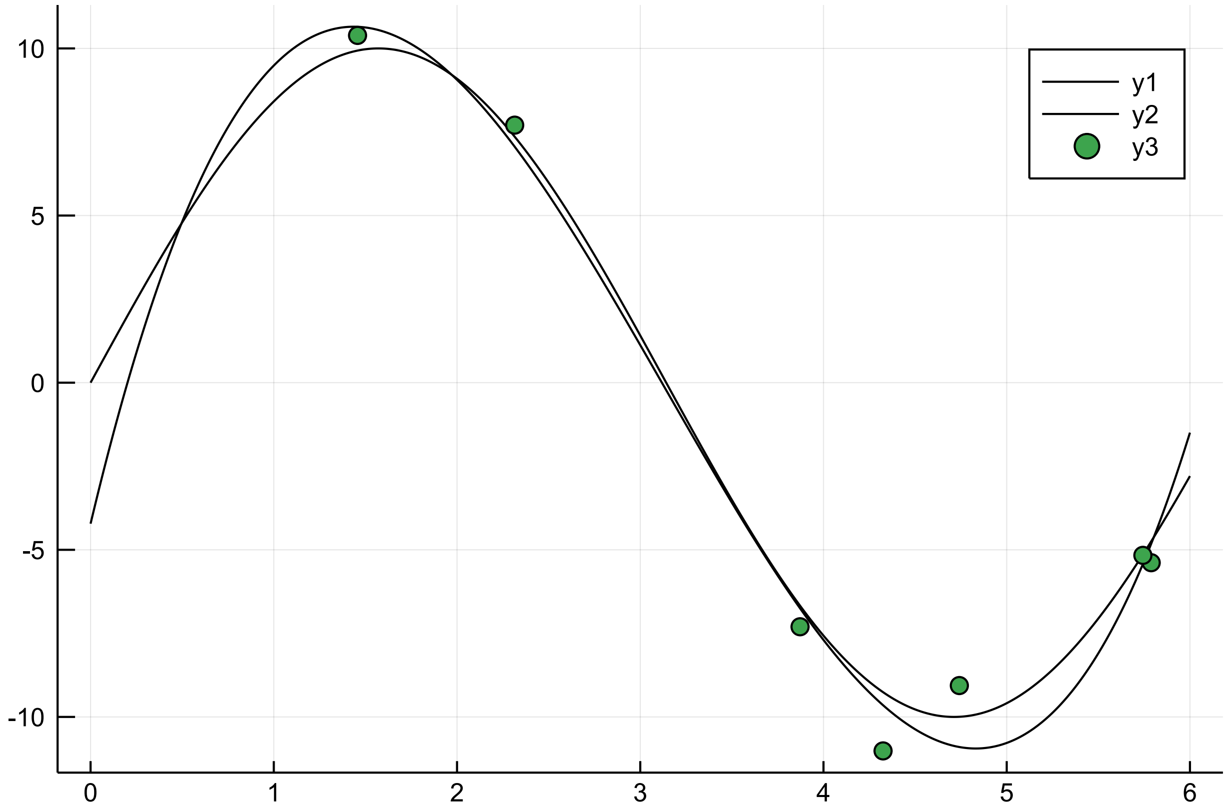
In [14]:
```
c(0.5)
```

Out[14]:
```
4.815419893103307
```

Plot the cubic model with $\mathcal{D}$ and $f(x)$

In [15]:
```
plotfunc_first(c)
plotfunc(f)
scatter!(D[:,1],D[:,2])
```

Out[15]:



d)

Repat the parts (b) and (c) for 1000 different sets $\mathcal{D}$. Compute $\bar{l}$ and $\bar{c}$, the average linear and average cubic models. (Please name $\bar{l}$ as l_avg(x) and $\bar{c}$ as c_avg(x) in following codes)

In [16]:
```
max_iter = 1000
w_l_bar_all_iter = zeros((2, max_iter))
for j = 1:max_iter
    D_l_bar = zeros((n, 2))
    for i = 1:n
        D_l_bar[i,1] = 6*rand()
        D_l_bar[i,2] = f(D_l_bar[i,1])+randn() # yi = f(xi) + εi
    end
    X_l_bar = [D_l_bar[:,1] ones(7)]
    y_l_bar = D_l_bar[:,2]
    w_l_iter = X_l_bar\y_l_bar

    w_l_bar_all_iter[:,j] = w_l_iter
end

w_l_bar_all_iter
```

Out[16]:
```
2×1000 Array{Float64,2}:
 -5.54828  -4.94422  -1.68622  -5.32331  …  -4.4278  -4.01315  -4.7377
8
 18.2672   16.1442    2.00356  19.207       13.8299  11.1007   14.052
```

In [17]:
```
w_l_avg = mean(w_l_bar_all_iter, dims=2)
```

Out[17]:
```
2×1 Array{Float64,2}:
 -3.6639131625395844
 11.162068365196964
```

In [18]:
```
function l_avg(x)
    y_pred = [x 1]w_l_avg
    return( y_pred[1])
end
```

Out[18]:
```
l_avg (generic function with 1 method)
```

In [19]:
```
l_avg(0.5)
```

Out[19]:
```
9.330111783927173
```

In [20]:
```
max_iter = 1000
max_order = 3
w_c_bar_all_iter = zeros((4, max_iter))
n = 7

for j = 1:max_iter
    D_c_bar = zeros((n, 2))
    for i = 1:n
        D_c_bar[i,1] = 6*rand()
        D_c_bar[i,2] = f(D_c_bar[i,1])+randn() # yi = f(xi) + ϵi
    end

    x = D_c_bar[:,1]
    #print(x)
    V = zeros(n, max_order+1)
    for k=0:max_order
        V[:,k+1] = x.^k
    end

    # solve least squares problem
    w_c_iter = V\D_c_bar[:,2]
    #print(w_c_iter)
    #w_c_iter

    w_c_bar_all_iter[:,j] =  w_c_iter
end

w_c_bar_all_iter
```

Out[20]:
```
4×1000 Array{Float64,2}:
 -4.08815    -2.87338   -3.32336  …  -2.56673    4.58324    -6.82033
 22.5895     22.4843    20.152       20.3015    11.9309     26.6007
 -9.6757    -10.2223    -9.36472     -9.237     -7.0215    -11.9851
  0.993027    1.07941    1.0218       0.985538   0.813287    1.33425
```

In [21]:
```
w_c_avg = mean(w_c_bar_all_iter, dims=2)
```

Out[21]:
```
4×1 Array{Float64,2}:
 -2.9469356018885002
 20.817732508578587
 -9.43436502256204З
  0.9902613312517625
```

In [22]:
```
c_avg(x) = c(x; order = max_order, w = w_c_avg)
```

Out[22]:
```
c_avg (generic function with 1 method)
```

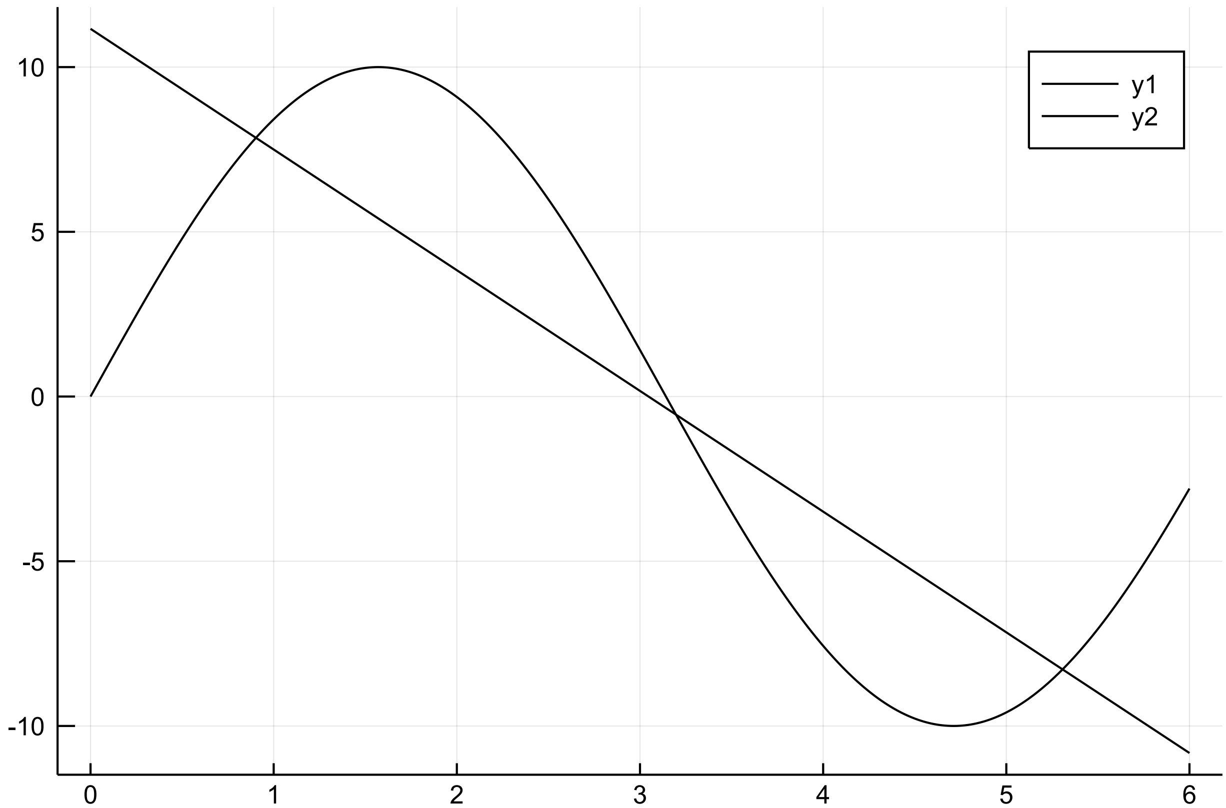
In [23]:
```
c_avg(0.5)
```

Out[23]: 5.227122063166754

Plot $\bar{l}$ together with $f(x)$.

In [24]:
```
plotfunc_first(l_avg)
plotfunc(f)
```

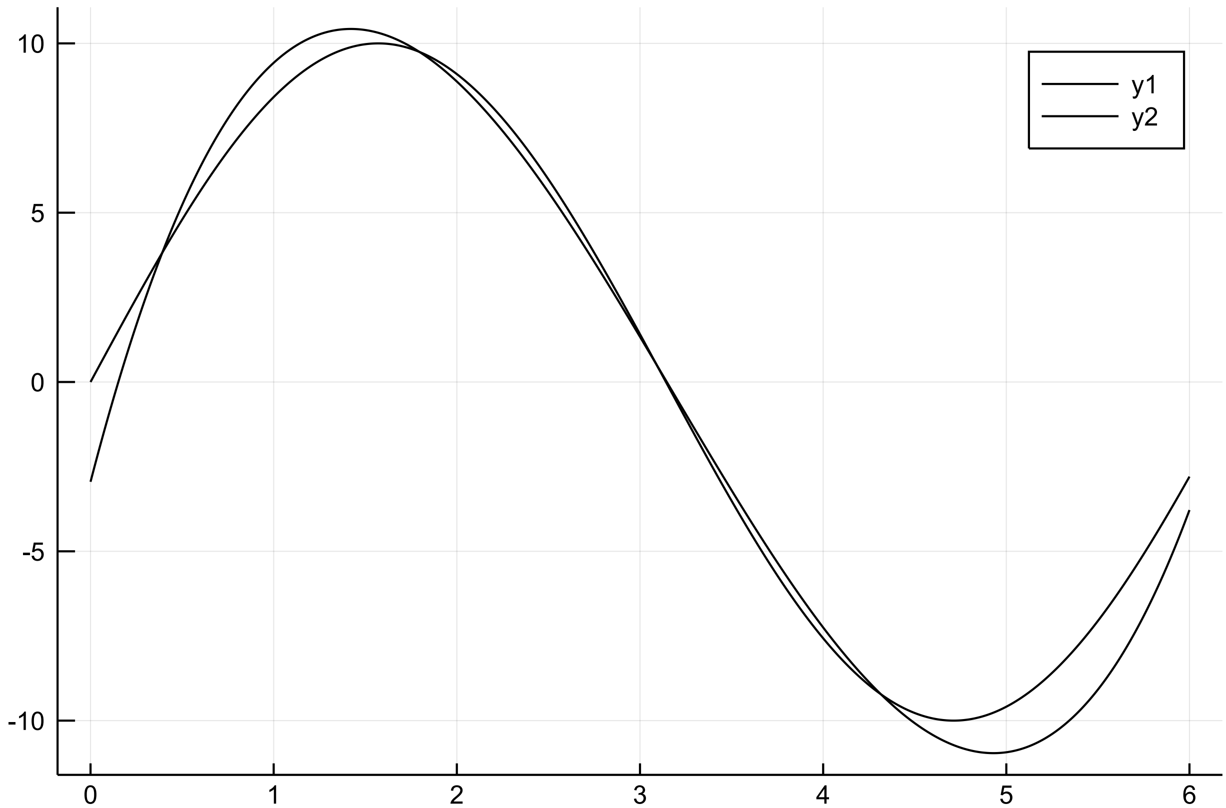
Out[24]:



Plot $\bar{c}(x)$ together with $f(x)$.

In [25]:
```
plotfunc_first(c_avg)
plotfunc(f)
```

Out[25]:



e)

Compute the bias of $\bar{l}$. You can use our integrate function.

In [26]:
```
function integrate(f, a, b)
    n = 1000
    delta = (b - a)/n;              ## nothing to change below here
    xs = a*ones(n) + [0:1:n-1;] * delta;          ## n, right is 1:n * d
    fx = map(f, xs);
    return sum(fx) * delta
end
```

Out[26]: integrate (generic function with 1 method)

In [27]:
```
g_l(x) = (f(x) - l_avg(x))^2
```

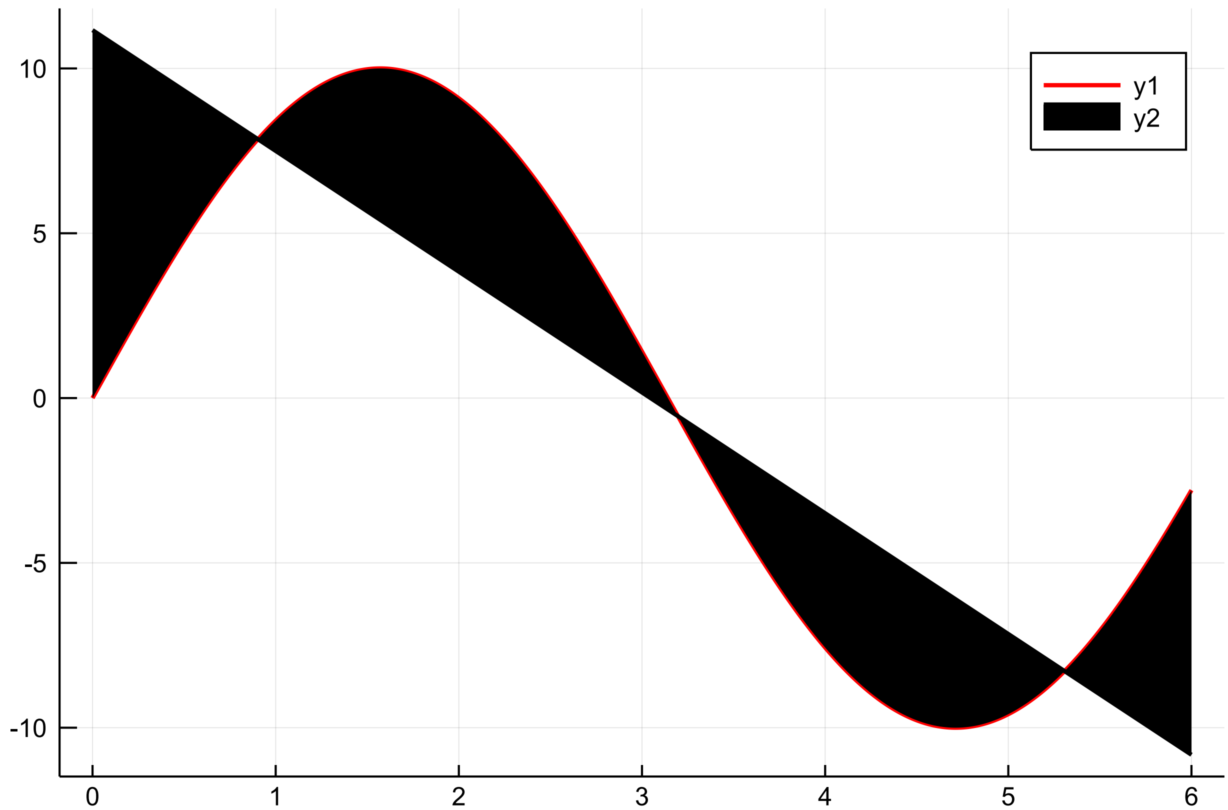
Out[27]: g_l (generic function with 1 method)

In [28]:
```
bias_l_bar = integrate(g_l,0,6)
```

Out[28]: 103.88269898743333

In [29]:
```
x_range = range(0,stop=6,length=1000)
x = [x for x in x_range]
y = f(x)
y_2 = [l_avg(x) for x in x_range]
plot(x, y, color="red", linewidth=2.0)
plot!(x, y_2, fillrange=[y y_2], color="black", linewidth=2.0)
```

Out[29]:



Compute the bias of the cubic model $\bar{c}(x)$.

In [30]:
```
g_c(x) = (f(x) - c_avg(x))^2
```

Out[30]: g_c (generic function with 1 method)

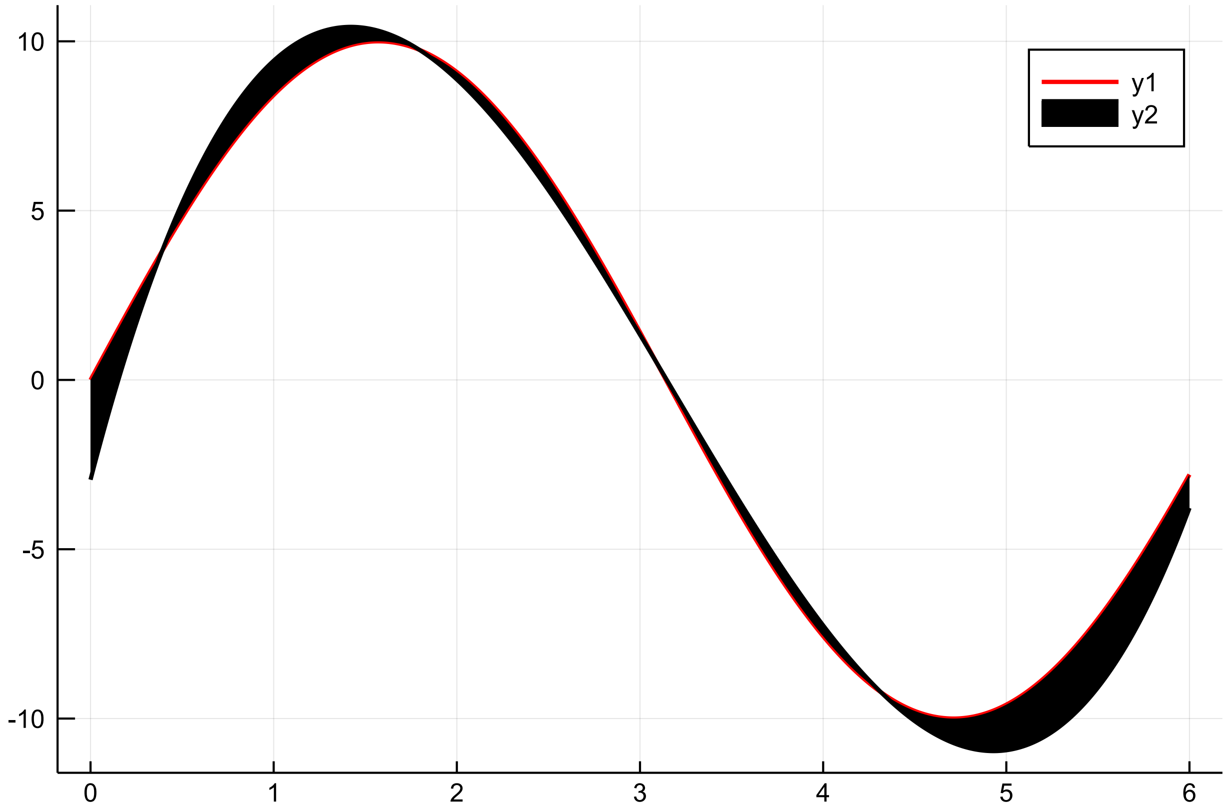
In [31]:
```
bias_c_bar = integrate(g_c,0,6)
```

Out[31]: 5.355579462582125

We can interpret the bias as how far off our averaged model is from the true function. One way to visually see this is by plotting $\bar{l}(x)$ with $f(x)$ and color in their difference. Try out the plotting function below.

In [32]:
```
x_range = range(0,stop=6,length=1000)
x = [x for x in x_range]
y = f(x)
y_2 = [c_avg(x) for x in x_range]
plot(x, y, color="red", linewidth=2.0)
plot!(x, y_2, fillrange=[y y_2], color="black", linewidth=2.0)
#plot!([x y_2], fillrange=[y y_2], fillalpha=0.3, c=:orange)
```

Out[32]:



Which model has smaller bias?

Polynomial model has a smaller bias (of 4.6) as compared to Linear model (which has a bias of 104).

f)

Next compute the variance of the linear model.

In [33]:
```
x_range = range(0,stop=6,length=1000)
x = [x for x in x_range]
Var_l = 0

for i = 1:1000
    #print(x[i])
    X_1 = [x[i] 1]
    #print(i, X_1)
    #print(i,w_l_bar_all_iter[:,i])
    y_jth_model_pred = X_1*w_l_bar_all_iter[:,i]
    #print(y_jth_model_pred)
    #print(w_l_avg)
    y_l_avg_pred = X_1*w_l_avg
    Var_l = Var_l + (y_jth_model_pred[1] - y_l_avg_pred[1])^2
end

variance_linear = Var_l*6/1000  #integration factor (b-a/n)
```

Out[33]:
```
50.37938888279812
```

Compute the variance of the cubic model.

In [34]:
```
x_range = range(0,stop=6,length=1000)
x = [x for x in x_range]
Var_c = 0

for i = 1:1000
    #print(x[i])
    X_1 = x[i]
    #print(i, X_1)
    #print(i,w_l_bar_all_iter[:,i])
    y_jth_model_pred = c(X_1, w=w_c_bar_all_iter[:,i])
    #print(y_jth_model_pred)
    #print(w_l_avg)
    y_c_avg_pred = c(X_1, w=w_c_avg)
    Var_c = Var_c + (y_jth_model_pred[1] - y_c_avg_pred[1])^2
end

variance_cubic = Var_c*6/1000
```

Out[34]:
```
426.55981691979196
```

In [35]:
```
›tal_error_linear = variance_linear + bias_l_bar  ## I am doing square whi
›tal_error_cubic = variance_cubic + bias_c_bar
:int("Total linear error: ",total_error_linear,"Total cubic error: ",total
```

```
Total linear error: 154.26208787023145Total cubic error: 431.915396382
37407
```

Which model had higher variance? How do you interpret this? Which model has smaller overall error?

Cubic model has higher variance (426) as compared to linear model (50).
Total error is high for cubic model (431) as compared to linear models (154).

This indicetes typical bias-variance tradeoff. Models with high values of bias (linear model in this case), undermine their training dataset (underfit) leading to not learn the trends in training dataset. consquently these models are simpler also. These models do not vary a great deal with change in training dataset.

High-variance are more complex and represent their training set well (at times overfit). However, these models are not able to handle random variation in test dataset and hence do not generalize well.

Ideally, one wants to choose a model with training error being low and generalize on out of set data as well - Alas! not easy to get there.

g)

How do you think your results would depend on the number of points in the data set $\mathcal{D}$? Feel free to perform an experiment to check. How many points would you need before the opposite model has smaller overall error?

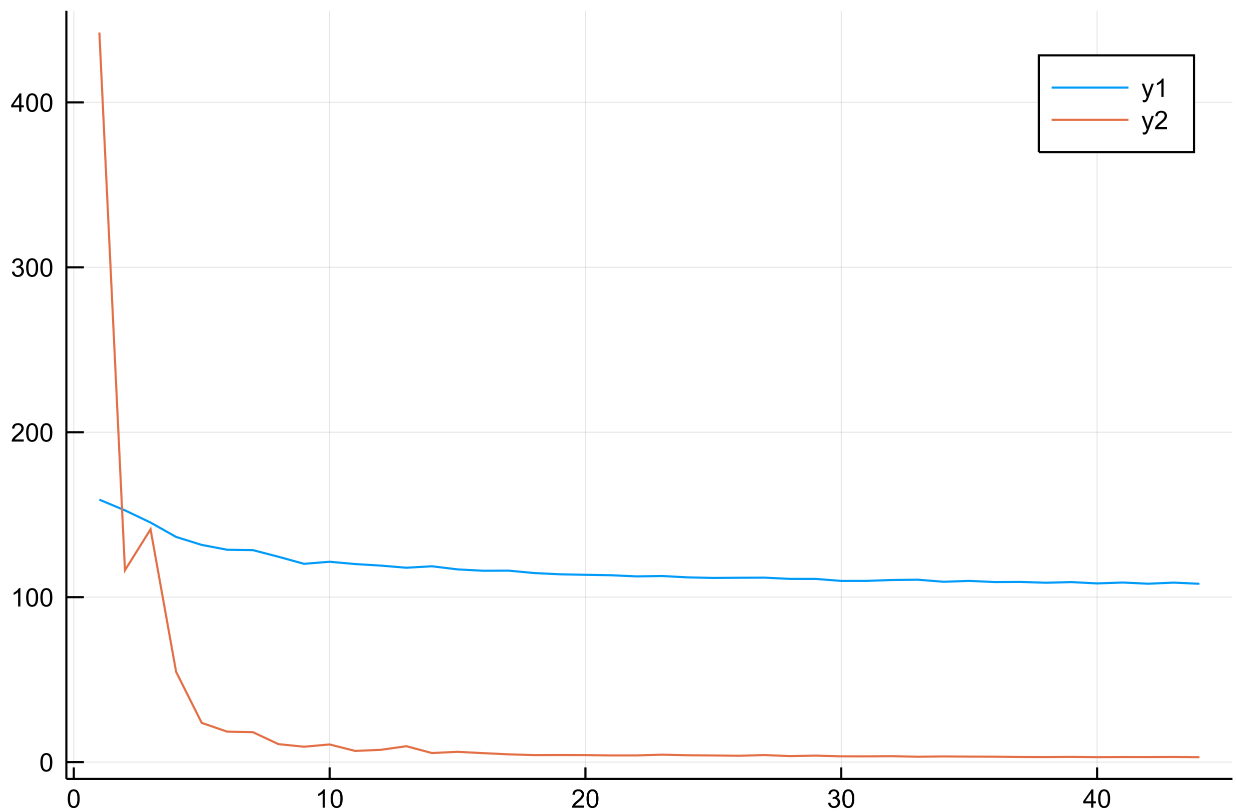
In [36]:
```
n_min = 7 # min data points
n_max = 50 #max data points
max_iter = 1000
x_range = range(0,stop=6,length=max_iter)
x = [x for x in x_range]
n_iter = n_max-n_min+1
max_order = 3
```

```
Variance_matrix = zeros((2,n_iter))
Bias_matrix = zeros((2,n_iter))
total_error_matrix = zeros((2,n_iter))

for n = n_min:n_max

    ## Linear model calculations start here
    w_l_bar_all_iter = zeros((2, max_iter))
    w_c_bar_all_iter = zeros((4, max_iter))

    for j = 1:max_iter
        D_bar = zeros((n, 2))
        for i = 1:n
            D_bar[i,1] = 6*rand()
            D_bar[i,2] = f(D_bar[i,1])+randn() # yi = f(xi) + ϵi
        end
        X_l_bar = [D_bar[:,1] ones(n)]
        y_l_bar = D_bar[:,2]
        w_l_iter = X_l_bar\y_l_bar
        w_l_bar_all_iter[:,j] = w_l_iter


        b = D_bar[:,1]
        V = zeros(n, max_order+1)
        for k=0:max_order
            V[:,k+1] = b.^k
        end
        w_c_iter = V\D_bar[:,2]
        w_c_bar_all_iter[:,j] =  w_c_iter

    end

    w_l_avg = mean(w_l_bar_all_iter, dims=2)
    Var_l = 0
    Bias_l = 0

    w_c_avg = mean(w_c_bar_all_iter, dims=2)
    Var_c = 0
    Bias_c = 0

    for k = 1:max_iter
        X_l = [x[k] 1]
        y_jth_model_pred = X_l*w_l_bar_all_iter[:,k]
        y_l_avg_pred = X_l*w_l_avg
        Var_l = Var_l + (y_jth_model_pred[1] - y_l_avg_pred[1])^2
        Bias_l = Bias_l + (f(x[k][1])-y_l_avg_pred[1])^2

        X_c = x[k]
        y_jth_model_pred = c(X_c, w=w_c_bar_all_iter[:,k])
        y_c_avg_pred = c(X_c, w=w_c_avg)
```

```
        Var_c = Var_c + (y_jth_model_pred[1] - y_c_avg_pred[1])^2
        Bias_c = Bias_c + (f(x[k][1])-y_c_avg_pred[1])^2

    end
    Variance_matrix[1,n-n_min+1] = Var_l*(6/max_iter)   ##integration fa
    Bias_matrix[1,n-n_min+1] = Bias_l*(6/max_iter)      ##integration fa

    Variance_matrix[2,n-n_min+1] = Var_c*(6/max_iter)
    Bias_matrix[2,n-n_min+1] = Bias_c*(6/max_iter)

end

total_error_matrix = Variance_matrix + Bias_matrix

#print(Variance_matrix)
#print(Bias_matrix)

print(total_error_matrix[:,1:5])
plot(total_error_matrix[1,:])
plot!(total_error_matrix[2,:])
```

```
[159.11809730300524 152.6357284653181 145.24229994630056 136.516953270
8011 131.68946169239317; 442.33623019159825 116.32572754368468 141.147
0892034337 54.70764400947264 23.785039529958958]
```

Out[36]:



As number of datapoints in D increases, bias comes down significantly. As n > 9, opposite model (in this case Cubic model) has lower totoal error than part (f). Errors for both models flattens out after 30 odd data points.

h)

Instead of sampling new data to compute the bias and variance of our model, we could use a bootstrap estimator to get more use out of the few data points we have. Try this for a few different data set sizes and report on your results. How big a data set is needed for the bootstrap to give a reliable estimate of the bias and variance?

Plan:
1. In order to understand how many data points are needed for a reliable measure of true error (which in trun will give reliable measures for bias and variance) , I am going to take n =7 and plot mean error square ((y_pred-y_true)^2) for different values of K (bootstrap sample size). At some point of K, this error should flatten out - which will give us a threshold for K.
2. Try exercise 1 for couple of different values of n.
3. Understanding the affect of n and K on Confidence intervals for w.

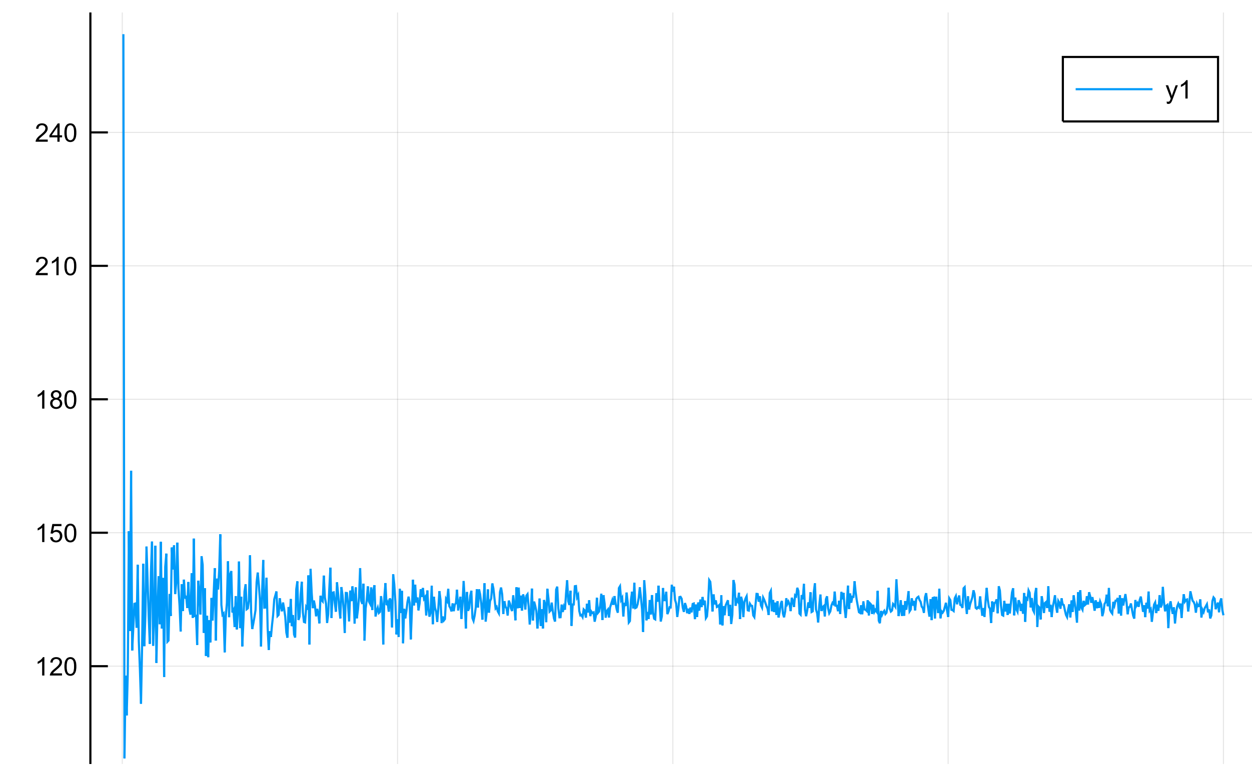
Note: I will be restricting this exercise of bootstrapping to linear models and look at variation of first cofficent of w in those models. Conclusions from this should hold for other models and other values of w as well.

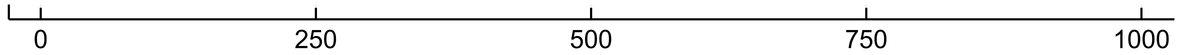
<u>Part 1</u>
Mean error vs K (for n = 7)

In [107]:
```
n_h_1 = 7

## Generate a sample size of 500 points from which we will be drawing sa
Sample_size = 500
Samples = zeros((Sample_size, 2))
for i = 1:Sample_size
    Samples[i,1] = 6*rand()
    Samples[i,2] = f(Samples[i,1])+randn() # yi = f(xi) + ϵi
end

Boot_Strap_Sample_Size_max = 1000
Avg_total_Bootstrap_error = zeros(Boot_Strap_Sample_Size_max)
```

```
Avg_total_Bootstrap_error = zeros(Boot_Strap_Sample_Size_max)
for m = 1:Boot_Strap_Sample_Size_max
    loop_error = 0
    for k = 1:m
        error = 0
        D = zeros((n_h_1,2))
        for j = 1:n_h_1
            u = rand(1:Sample_size)
            D_random_point = Samples[u,:]
            #print(D_random_point)
            D[j,:] = D_random_point
        end
        #print(D)
        X_l_bar = [D[:,1] ones(n_h_1)]
        y_l_bar = D[:,2]
        w_l_iter = X_l_bar\y_l_bar
        y_pred = X_l_bar*w_l_avg
        #print(y_pred)
        error = sum((y_pred-y_l_bar).^2)
        loop_error = loop_error + error
    end
    Avg_total_Bootstrap_error[m] = loop_error/m
end

plot(Avg_total_Bootstrap_error)
#true_error_plot = true_error*ones(size(boot_error))
#plot!(true_error_plot)
#size(true_error)
#print(true_error)
```

Out[107]:





As we see above for about K =250, errors reduce in a constant range.

<u>Part 2</u>

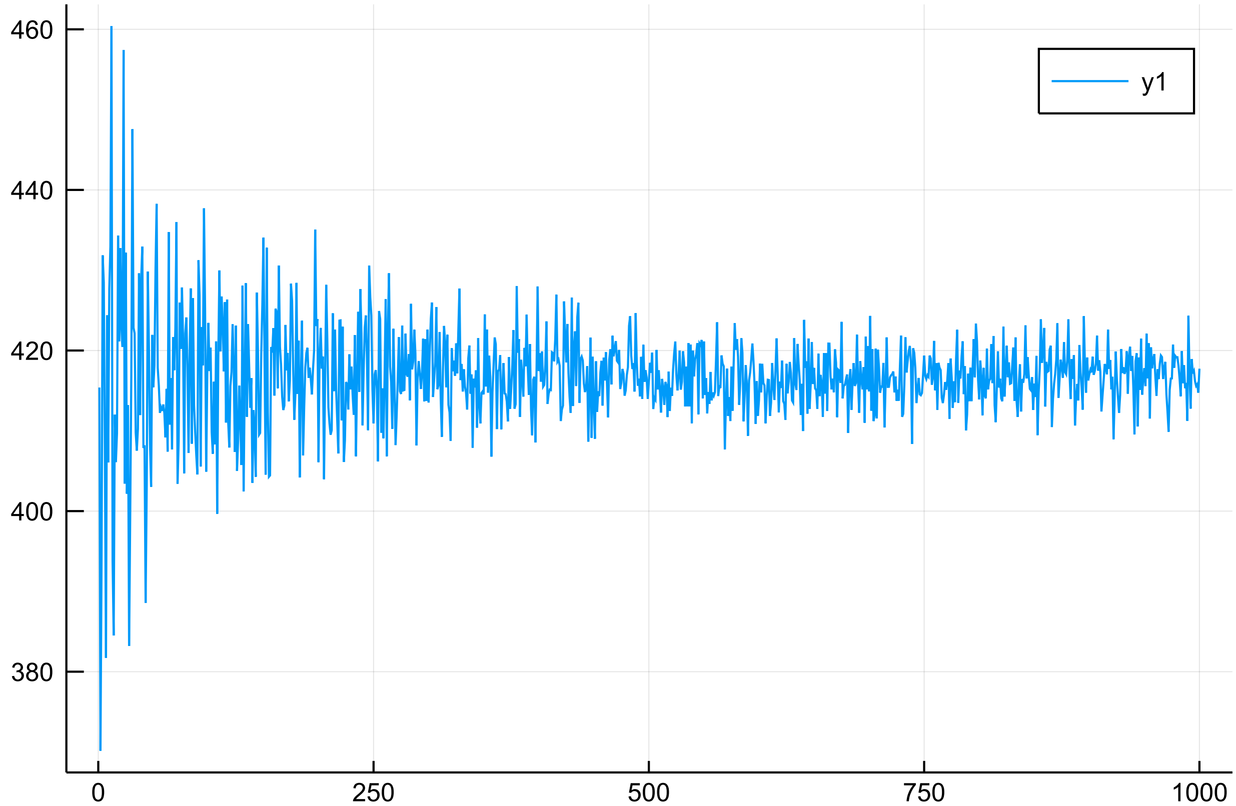
Part 1 for n = 25,45, 60.

In [109]:

```
n_h_1 = 25

## Generate a sample size of 500 points from which we will be drawing sa
Sample_size = 500
Samples = zeros((Sample_size, 2))
for i = 1:Sample_size
    Samples[i,1] = 6*rand()
    Samples[i,2] = f(Samples[i,1])+randn() # yi = f(xi) + εi
end

Boot_Strap_Sample_Size_max = 1000
Avg_total_Bootstrap_error = zeros(Boot_Strap_Sample_Size_max)
for m = 1:Boot_Strap_Sample_Size_max
    loop_error = 0
    for k = 1:m
        error = 0
        D = zeros((n_h_1,2))
        for j = 1:n_h_1
            u = rand(1:Sample_size)
            D_random_point = Samples[u,:]
            #print(D_random_point)
            D[j,:] = D_random_point
        end
        #print(D)
        X_l_bar = [D[:,1] ones(n_h_1)]
        y_l_bar = D[:,2]
        w_l_iter = X_l_bar\y_l_bar
        y_pred = X_l_bar*w_l_avg
        #print(y_pred)
        error = sum((y_pred-y_l_bar).^2)
        loop_error = loop_error + error
    end
    Avg_total_Bootstrap_error[m] = loop_error/m
end

plot(Avg_total_Bootstrap_error)
#true_error_plot = true_error*ones(size(boot_error))
#plot!(true_error_plot)
#size(true_error)
```

```
#print(true_error)
```

Out[109]:

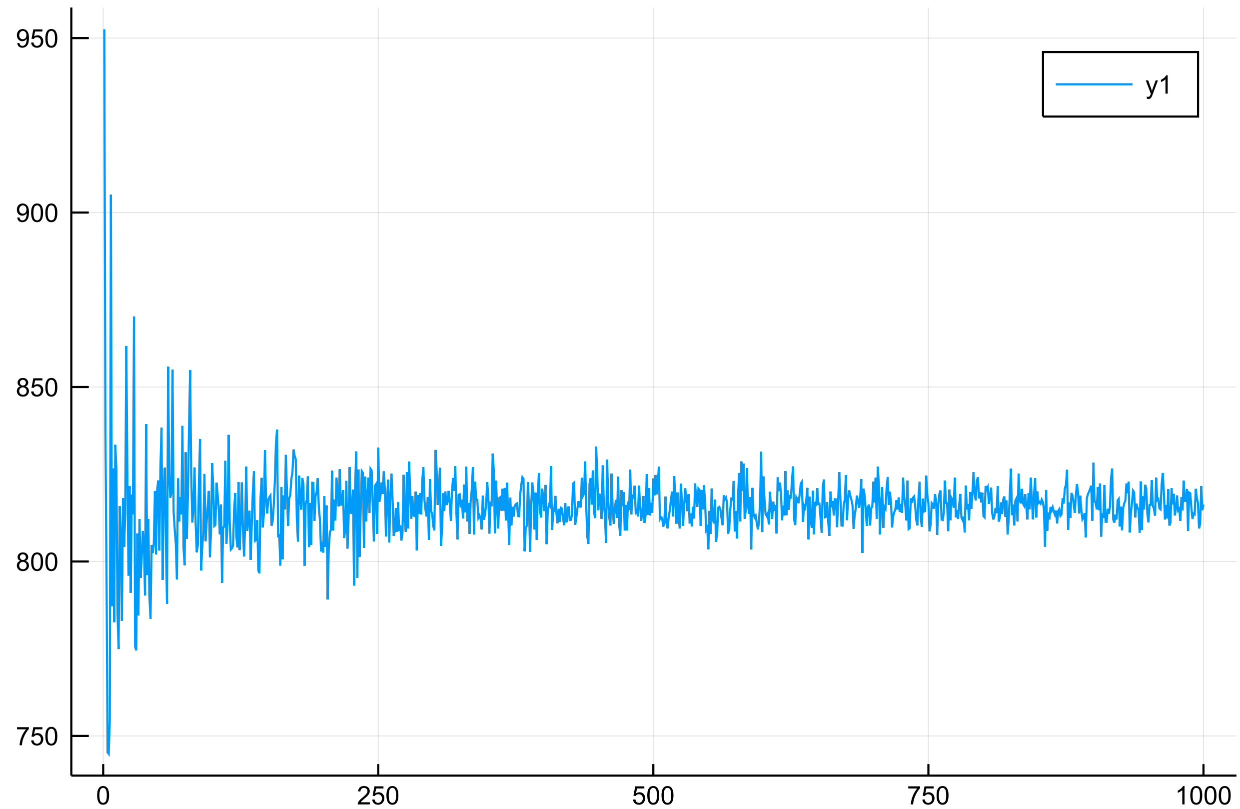


In [111]:

```
n_h_1 = 45

## Generate a sample size of 500 points from which we will be drawing sa
Sample_size = 500
Samples = zeros((Sample_size, 2))
for i = 1:Sample_size
    Samples[i,1] = 6*rand()
    Samples[i,2] = f(Samples[i,1])+randn() # yi = f(xi) + ϵi
end

Boot_Strap_Sample_Size_max = 1000
Avg_total_Bootstrap_error = zeros(Boot_Strap_Sample_Size_max)
for m = 1:Boot_Strap_Sample_Size_max
    loop_error = 0
    for k = 1:m
        error = 0
        D = zeros((n_h_1,2))
        for j = 1:n_h_1
            u = rand(1:Sample_size)
            D_random_point = Samples[u,:]
            #print(D_random_point)
            D[j,:] = D_random_point
        end
        #print(D)
```

```
        X_l_bar = [D[:,1] ones(n_h_1)]
        y_l_bar = D[:,2]
        w_l_iter = X_l_bar\y_l_bar
        y_pred = X_l_bar*w_l_avg
        #print(y_pred)
        error = sum((y_pred-y_l_bar).^2)
        loop_error = loop_error + error
    end
    Avg_total_Bootstrap_error[m] = loop_error/m
end

plot(Avg_total_Bootstrap_error)
#true_error_plot = true_error*ones(size(boot_error))
#plot!(true_error_plot)
#size(true_error)
#print(true_error)
```

Out[111]:

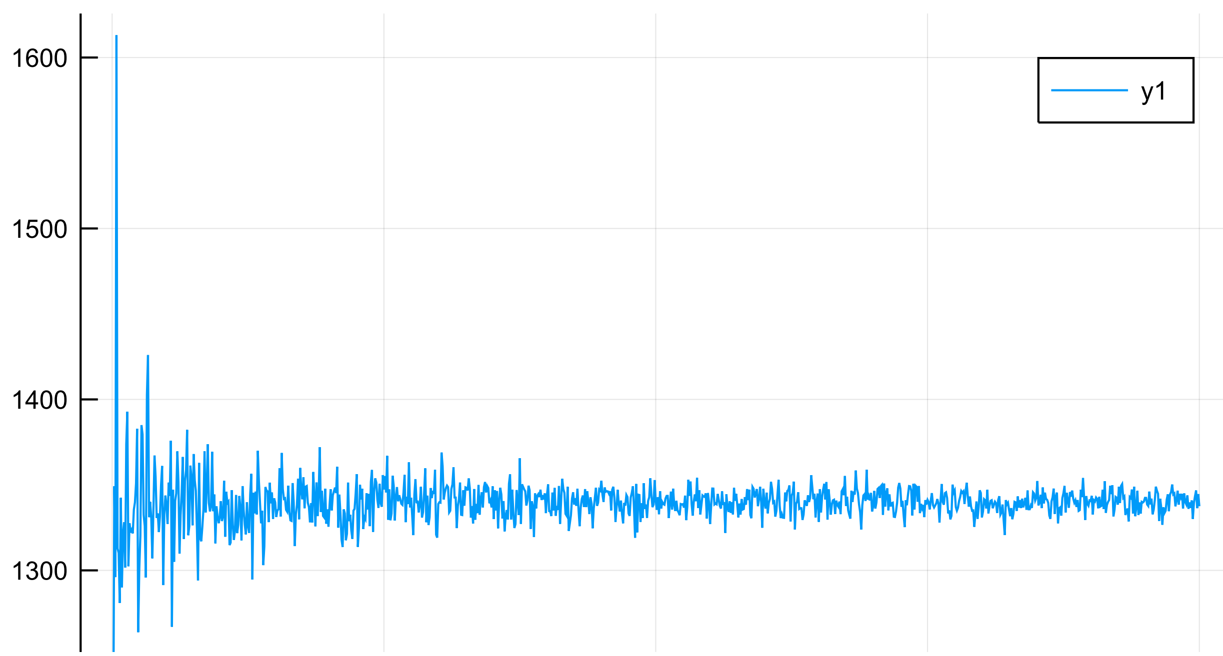


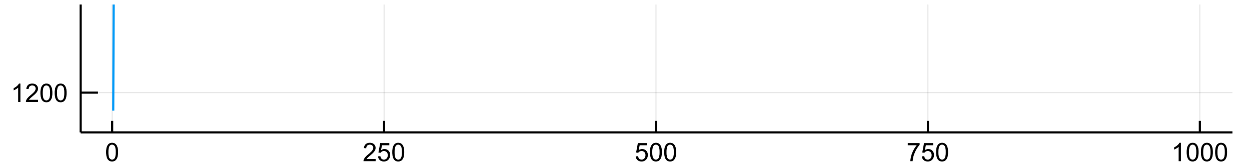
In [112]:

```
n_h_1 = 70

## Generate a sample size of 500 points from which we will be drawing sa
Sample_size = 500
Samples = zeros((Sample_size, 2))
for i = 1:Sample_size
    Samples[i,1] = 6*rand()
    Samples[i,2] = f(Samples[i,1])+randn() # yi = f(xi) + εi
end
```

```
end

Boot_Strap_Sample_Size_max = 1000
Avg_total_Bootstrap_error = zeros(Boot_Strap_Sample_Size_max)
for m = 1:Boot_Strap_Sample_Size_max
    loop_error = 0
    for k = 1:m
        error = 0
        D = zeros((n_h_1,2))
        for j = 1:n_h_1
            u = rand(1:Sample_size)
            D_random_point = Samples[u,:]
            #print(D_random_point)
            D[j,:] = D_random_point
        end
        #print(D)
        X_l_bar = [D[:,1] ones(n_h_1)]
        y_l_bar = D[:,2]
        w_l_iter = X_l_bar\y_l_bar
        y_pred = X_l_bar*w_l_avg
        #print(y_pred)
        error = sum((y_pred-y_l_bar).^2)
        loop_error = loop_error + error
    end
    Avg_total_Bootstrap_error[m] = loop_error/m
end

plot(Avg_total_Bootstrap_error)
#true_error_plot = true_error*ones(size(boot_error))
#plot!(true_error_plot)
#size(true_error)
#print(true_error)
```

Out[112]:





In [ ]:

Part 3

We can borrow some learnings from part g, where we saw affect of size of dataset (n) on total error. Total errors flattens after a certain point (somewhere around n= 27).

We can further study impact of n on bootstrapping by taking three values of n. n = 7, 15, 25.
Also I will be considering number of datapoints taken, k = 100.
Below we will be looking at the Confidence interval for each value of n. Half width for 95% Confidence interval should keep reducing as n increases.

In [37]:

```
n_h = 35
K = 1000
max_iter = K
w_l_bar_all_iter = zeros((2, max_iter))
for j = 1:max_iter
    D_l_bar = zeros((n_h, 2))
    for i = 1:n_h
        D_l_bar[i,1] = 6*rand()
        D_l_bar[i,2] = f(D_l_bar[i,1])+randn() # yi = f(xi) + ϵi
    end
    X_l_bar = [D_l_bar[:,1] ones(n_h)]
    y_l_bar = D_l_bar[:,2]
    w_l_iter = X_l_bar\y_l_bar

    w_l_bar_all_iter[:,j] = w_l_iter
end

z = w_l_bar_all_iter[1,:]   #first cofficent of w
UB = mean(z) + 1.96*sqrt(var(z)/K) # 95% confidence interval
LB = mean(z) - 1.96*sqrt(var(z)/K)
print("n = ", n_h, "; k = ", K, "; Confidence interval = ", LB,",", UB,
```

```
n = 35; k = 1000; Confidence interval = -3.518638495232864,-3.45718432
75311056; Half width = 0.0307270838508793
```

n = 7; k = 100; Confidence interval = -3.8342240783025403,-3.301831389430105; Half width = 0.26619634443621765
n = 15; k = 100; Confidence interval = -3.807226401341788,-3.4520754000707012; Half width = 0.17757550063554328
n = 25; k = 100; Confidence interval = -3.6357753693738997,-3.4047754358186473; Half width = 0.1154999667776262
n = 35; k = 100; Confidence interval = -3.5271356160037763,-3.3509984924327982; Half width = 0.08806856178548905
n = 100; k = 100; Confidence interval = -3.524302390585087,-3.4115248830183442; Half width = 0.05638875378337138

n = 35; k = 10; Confidence interval = -3.7530071105002247,-3.0721762268779265; Half width = 0.34041544181114913
n = 35; k = 50; Confidence interval = -3.588653833079747,-3.2770852642801134; Half width = 0.15578428439981673
n = 35; k = 100; Confidence interval = -3.6330291669220243,-3.4462648808811265; Half width = 0.0933821430204489
n = 35; k = 1000; Confidence interval = -3.5192832189851346,-3.457504226763966; Half width = 0.030889496110584336

As we can see from above data points:

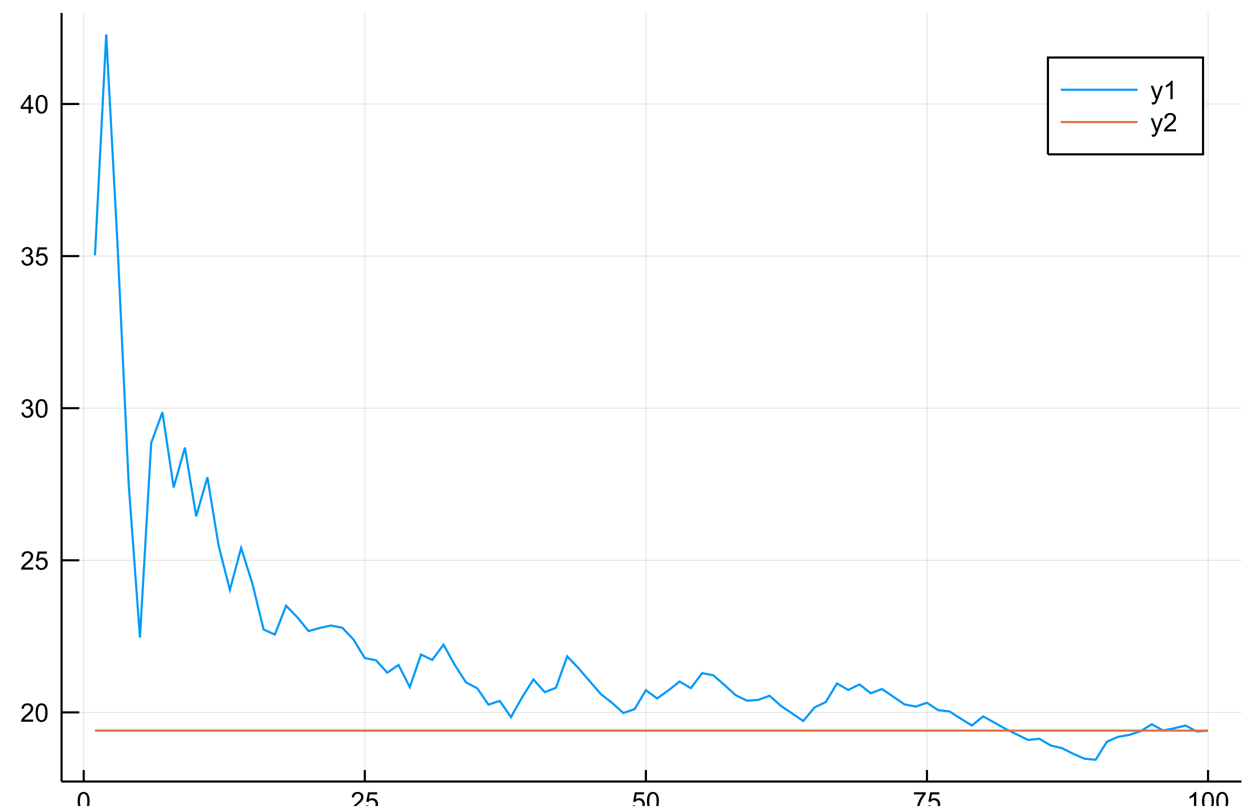
1. As n and k increases half width reduces, which means with more confidence we can locate true value of w[1].
2. this makes intuitive sense, as higher the K (number of samples withdrawn), more close the estimates will be to actual.
3. For our use case n = 25-35 and K~100 should give 95% confidence interval of half width within 1.5% of the mean can be achieved.

Rough Work

In [76]:
```
n_h_1 = 100
true_error = zeros(n_h_1)

D_l_bar = zeros(n_h_1, 2)

for i = 1:n_h_1
    D_l_bar[i,1] = 6*rand()
    D_l_bar[i,2] = f(D_l_bar[i,1])+randn() # yi = f(xi) + ϵi
```

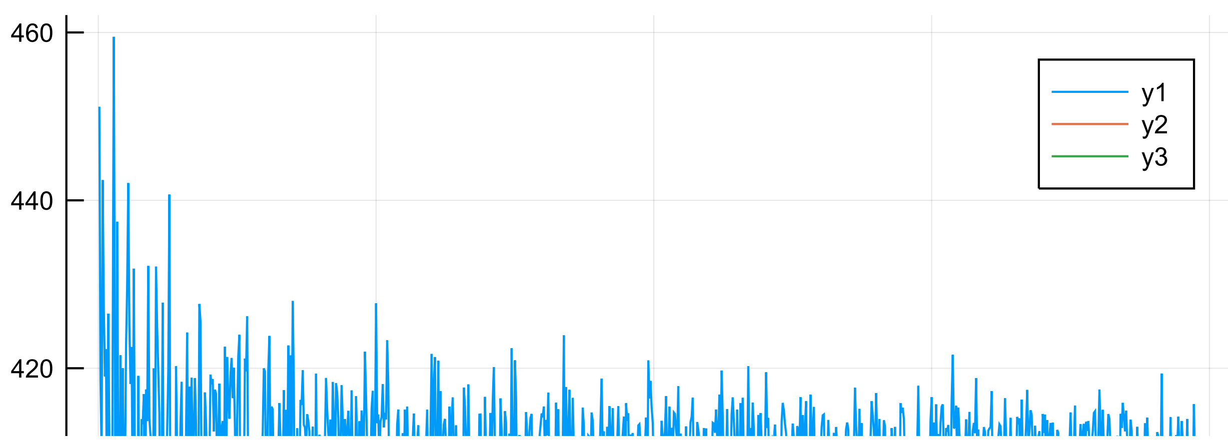
```
end
X_l_bar = [D_l_bar[:,1] ones(n_h_1)]
y_l_bar = D_l_bar[:,2]
w_l_iter = X_l_bar\y_l_bar
y_pred = X_l_bar*w_l_iter
error = (y_pred-y_l_bar).^2
true_error = sum((y_pred-y_l_bar).^2)/n_h_1

##Boot strapping
K_min = 1
K_max = n_h_1
boot_error = zeros(n_h_1)

for i = K_min:K_max
    error_i = 0
    for j = 1:i
        u = rand(1:n_h_1)
        error_i = error_i + error[j]
    end
    boot_error[i] = error_i/i
end

plot(boot_error)
true_error_plot = true_error*ones(size(boot_error))
plot!(true_error_plot)
#size(true_error)
#print(true_error)
```

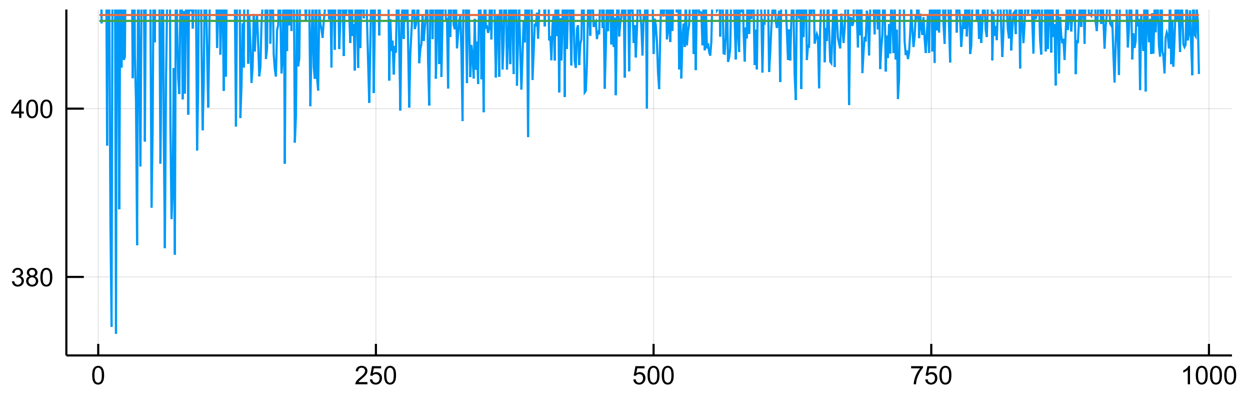
Out[76]:



In [71]:
```
n_h_1 = 25
K_min = 10
K_max = 1000
true_error = zeros(0)

for j = K_min:K_max
    max_iter = j
    w_l_bar_all_iter = zeros((2, max_iter))
    error = 0
    for j = 1:max_iter
        D_l_bar = zeros((n_h_1, 2))
        for i = 1:n_h_1
            D_l_bar[i,1] = 6*rand()
            D_l_bar[i,2] = f(D_l_bar[i,1])+randn() # yi = f(xi) + ϵi
        end
        X_l_bar = [D_l_bar[:,1] ones(n_h_1)]
        y_l_bar = D_l_bar[:,2]
        w_l_iter = X_l_bar\y_l_bar
        y_pred = X_l_bar*w_l_iter
        error = error + sum((y_pred-y_l_bar).^2)
        #print(error)
    end
    append!(true_error, error/j)
end

plot(true_error)
UB = mean(true_error) + 1.67*sqrt(var(true_error)/(K_max-K_min)) # 95% c
LB = mean(true_error) - 1.67*sqrt(var(true_error)/(K_max-K_min))
UB_plot = UB*ones(size(true_error))
LB_plot = LB*ones(size(true_error))
plot!(UB_plot)
plot!(LB_plot)
#print("n = ", n_h, "; k = ", K, "; Confidence interval = ", LB,",", UB,
#size(true_error)
#print(true_error)
```

Out[71]:




400
380
0
250
500
750
1000