Linear_regression_julia

August 26, 2021

1 Linear Regression in Julia

File explaining linear regression as seen in class and some other things like normalization. Everything is coded in Julia and proper documentation is written in the .jl script associated with this document.

1.1 Reading of the data

[2]:
```
using DelimitedFiles
using Statistics
using LinearAlgebra
using Plots
```

[3]:
```
function data_read(str::String)
    data = readdlm(str, ',', Float64, '\n');
    return data
end
```

[4]:
```
data = data_read("data2.txt");
```

[8]:
```
X = data[:,1:end-1];
y = data[:,end];
```

1.2 Feature Normalizacion

When diferent features have very different scales of values, sometimes gradient descent can be slower so we can speed up gradient descent by utilizing values in the same range. Ideally we use values such that

$$-1 \leq x_i \leq 1$$

To do this, while mantaining the same distribution we can use feature scaling and mean normalisation. With this we can mantain the distribution of the values, but the range is only $[-1, 1]$.

$$x_i := \frac{x_i - \mu_i}{\sigma_i}$$

Where x_i is the vetor with all the values of a feature, μ_i is the mean values of the feature and σ_i is their standard deviation.

[9]:
```
function normalization(x::Vector{Float64})
    return (x.-mean(x))/std(x)
end;
```

[11]:
```
X= mapslices(normalization, X; dims = 1);
y = normalization(y);
```

1.3 Hypothesis Function

Next we define a multivariate form of the hypothesis function:

$$h_\theta(x) = \theta_0 + \theta_1 x_1 + \theta_2 x_2 + \theta_3 x_3 + \cdots + \theta_n x_n$$

Here θ_0 is the bias of the function, θ_i are weights and x_i are the different parameters to calculate an aproximate y value $\forall i = 1, 2, 3, \ldots, n$. The function can be represented using vector and matrixes as:

$$h_\theta(x) = \begin{bmatrix} \theta_0 & \theta_1 & \ldots & \theta_n \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix} = \theta^T x$$

Where we assume that $x_0^{(i)} = 1$. Then, we can calculate the hypothesis function as:

$$h_\theta(X) = X\theta$$

So, for everything to work we first need to add a column of ones as the first column of the X matrix.

[12]:
```
X = [ones(size(X)[1]) X];
```

[13]:
```
theta = rand(size(X)[2],1);
```

[14]:
```
h(theta, X) = (theta * X)[1];
```

1.4 Cost Function

We can define the cost function as:

$$E(\theta) = \frac{1}{2m} \sum_{i=1}^{m} \left(h_\theta(x^{(i)}) - y^{(i)} \right)^2$$

The vectorized verision would be

$$\begin{aligned} E(\theta) &= \frac{1}{2m}(X\theta - y)^T(X\theta - y) \\ &= \frac{1}{2m}\|X\theta - y\|_2^2 \end{aligned}$$

```
[15]: m = size(X)[1];
```

```
[16]: E(theta) = norm(X*theta - y, 2)^2/2m;
```

To be able to modify the values of θ we need to calculate the derivative of $E(\theta)$. When we calculate the derivative, we can see that

$$\begin{aligned} \frac{\partial E(\theta)}{\partial \theta_j} &= \frac{1}{m}\sum_{i=1}^{m}\left(h_\theta(x^{(i)}) - y^{(i)}\right) \cdot x_j^{(i)} \\ &= \frac{1}{m}\sum_{i=1}^{m} x_j^{(i)} \cdot \left(h_\theta(x^{(i)}) - y^{(i)}\right) \\ &= \frac{1}{m} x_j^T (X\theta - y) \\ \implies \qquad \nabla E(\theta) &= \frac{1}{m} X^T (X\theta - y) \end{aligned}$$

Using this, we can modify the values of θ using

$$\theta := \theta - \frac{\alpha}{m} X^T (X\theta - y)$$

Where α is the learning rate.

```
[17]: grad_E(theta) = 1/m * transpose(X) * (X*theta - y);
```

```
[18]: alpha = 0.01;
```

```
[19]: theta_hist = transpose(theta);
```

Using this, we can see the cost associated with θ.

```
[21]: E(theta)
```

```
[21]: 0.19250318966999655
```

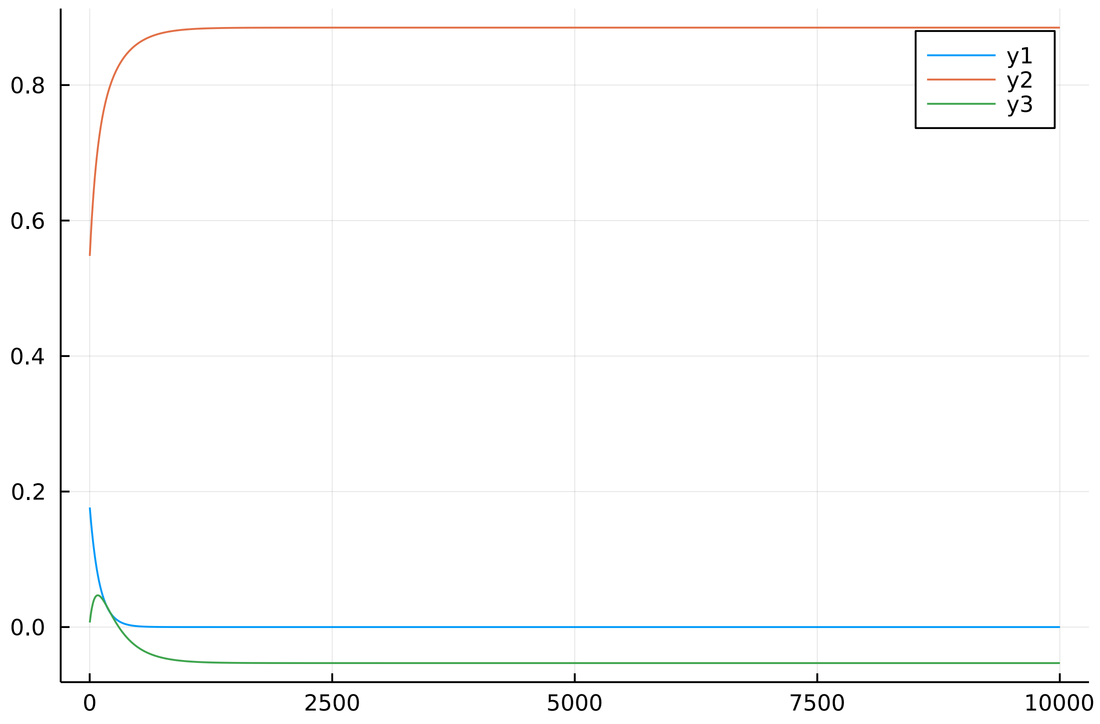
We can see how the training improves θ and reduces the cost. (Normally it should be run until convergence is seen but this here is just an example.)

[22]:
```julia
# Every time we run it, theta is improved. It should be run until convergence
for i in 1:10000
    theta -= alpha*grad_E(theta)
    theta_hist = vcat(theta_hist, transpose(theta))
end
E(theta)
```

[22]:
```
0.13068648053904197
```

[24]:
```julia
plot(theta_hist) # theta moves to certain values
```

[24]:



1.5 Putting everything together

To make it easier to solve a problem and train it, we can put everything together in a structure, so we can normalize the data automatically after reading it and save the values of the mean and std for future predictions. Also we can use this to make comparissons.

[25]:
```julia
mutable struct LR_Problem
    X::Matrix{Float64}        # Normalized data
    y::Vector{Float64}        # Normalized vector
    X_mean::Vector{Float64}   # Means of X matrix per column
    X_std::Vector{Float64}    # STD of X matrix per colum
    y_mean::Float64           # Mean of y
    y_std::Float64           # STD of y
    theta::Matrix{Float64}       # Weights and biases
    theta_hist::Matrix{Float64}
    alpha::Float64               # Learning rate
    tol::Float64             # Tolerance of error for convergence

    function LR_Problem(data::Matrix{Float64}; theta = Matrix{Float64}(undef,␣
↪0,0)::Matrix{Float64}, alpha = 0.01::Float64, tol = 1e-8::Float64)
        means = mapslices(mean, data; dims = 1)
        stds = mapslices(std, data; dims = 1)
        normalized_data = mapslices(normalization, data; dims = 1)
        X = [ones(size(data)[1]) normalized_data[:,1:end-1]]
        y = normalized_data[:,end]
        if size(theta)[2] > 1
            theta = transpose(theta)
        end
        if size(theta)[1] != size(X)[2]
            theta = rand(size(X)[2],1)
        end
        theta_hist = transpose(theta)
        new(X, y, means[1:end-1], stds[1:end-1], means[end], stds[end], theta,␣
↪theta_hist, alpha, tol)
    end
end;
```

[26]:
```
function compute_cost(prob::LR_Problem)
    m = size(prob.X)[1]
    return norm(prob.X*prob.theta - prob.y, 2)^2/2m
end;
```

[27]:
```
function compute_gradients(prob::LR_Problem)
    m = size(prob.X)[1]
    return 1/m * transpose(prob.X) * (prob.X * prob.theta - prob.y)
end;
```

[28]:
```
function train!(prob::LR_Problem; max_iter  = 100000::Int64)
    iter = 0
    # Train while it has not converged or reach 10000 iterations
    while norm(compute_gradients(prob),2)>prob.tol && iter<=max_iter
        prob.theta -= prob.alpha * compute_gradients(prob)
        prob.theta_hist = vcat(prob.theta_hist, transpose(prob.theta))
    end
end;
```

[ ]:
```
function predict(prob::LR_Problem, x::Vector{Float64})
    x = (x-prob.X_mean)./prob.X_std # normalize the vector
    x = transpose(vcat(1, x)) # add first column of 1 and transpose
    y = (x * prob.theta)[1] # predict
    y = y * prob.y_std + prob.y_mean # de-normalize
    return y
end;
```

Having all the previous functions. we can use it to see how the learning rates affect the convergence of the function.

[217]:
```
# First we can create various models with different learning rates.
theta = [0 0 0]
learning_rates = [0.001 0.005 0.01 0.05 0.1 0.5 1 1.3 1.5]
problems = [LR_Problem(data, theta = theta, alpha = alpha) for alpha in␣
 ↪learning_rates];
```

[218]:
```
train!.(problems);
```

We can see that all of them converged except the learning rate of 10. This shows that if the learning rate is too large it can diverge instead.

[219]:
```
Dict(learning_rates .=> compute_cost.(problems))
```

[219]:
```
Dict{Float64, Float64} with 9 entries:
  1.3   => 0.130686
  0.01  => 0.130686
  0.5   => 0.130686
  0.005 => 0.130686
  1.5   => NaN
  0.1   => 0.130686
  0.001 => 0.130686
  0.05  => 0.130686
  1.0   => 0.130686
```

[220]:
```
function errors_learning(prob::LR_Problem)
    return [norm(prob.X * prob.theta_hist[i,:] - prob.y, 2) for i=1:size(prob.
 ↪theta_hist)[1]]
end;
```

We can see that in this particular problem, the larger learning rates converge more quickly than the smaller learning rates. But if we go a little bigger, like for $\alpha = 1.3$ it is a lot slower, and for $\alpha = 1.5$ it even diverged to it is important to find a good balance between being fast and converging.

[222]:
```
Dict(learning_rates .=> sizeof.(errors_learning.(problems)))
```

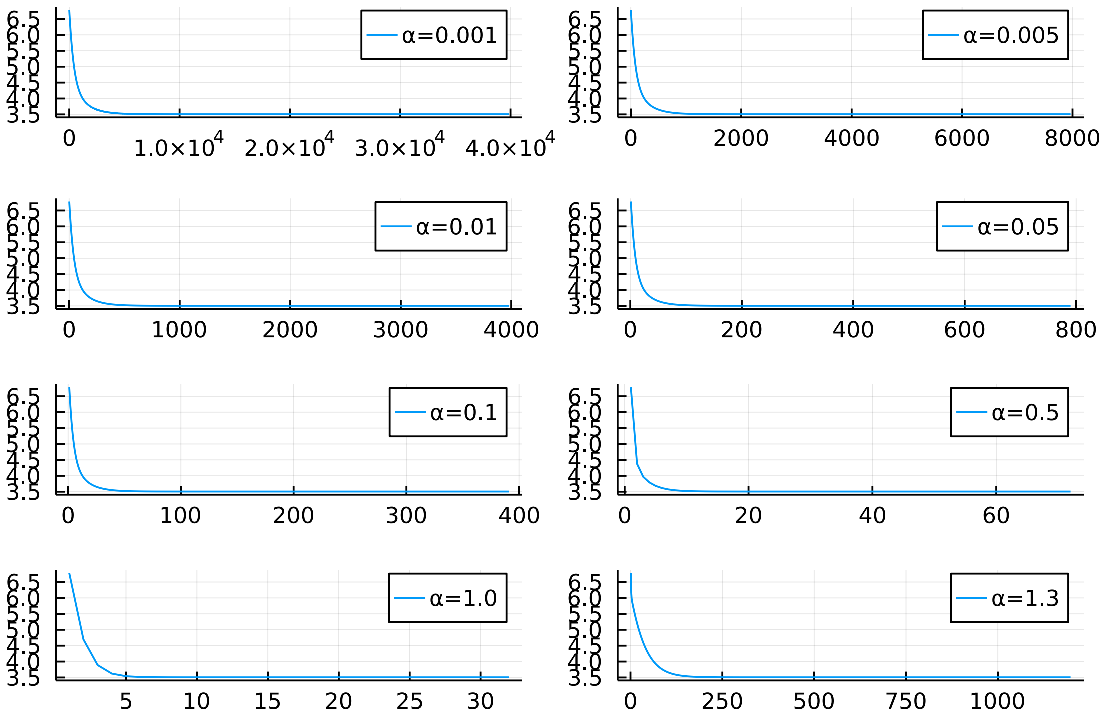
[222]:
```
Dict{Float64, Int64} with 9 entries:
  1.3   => 9584
  0.01  => 31832
  0.5   => 576
  0.005 => 63720
```

```
1.5   => 22280
0.1   => 3128
0.001 => 318840
0.05  => 6320
1.0   => 256
```

We can see that the fastest convergence is with $\alpha = 1$ and it slows down again in $\alpha = 1.3$.

[223]:
```
labels = ["alpha=0.001" "alpha=0.005" "alpha=0.01" "alpha=0.05" "alpha=0.1"␣
 ↪"alpha=0.5" "alpha=1.0" "alpha=1.3"]
plot(errors_learning.(problems[:1:end-1]), layout = (4,2), label = labels)
```

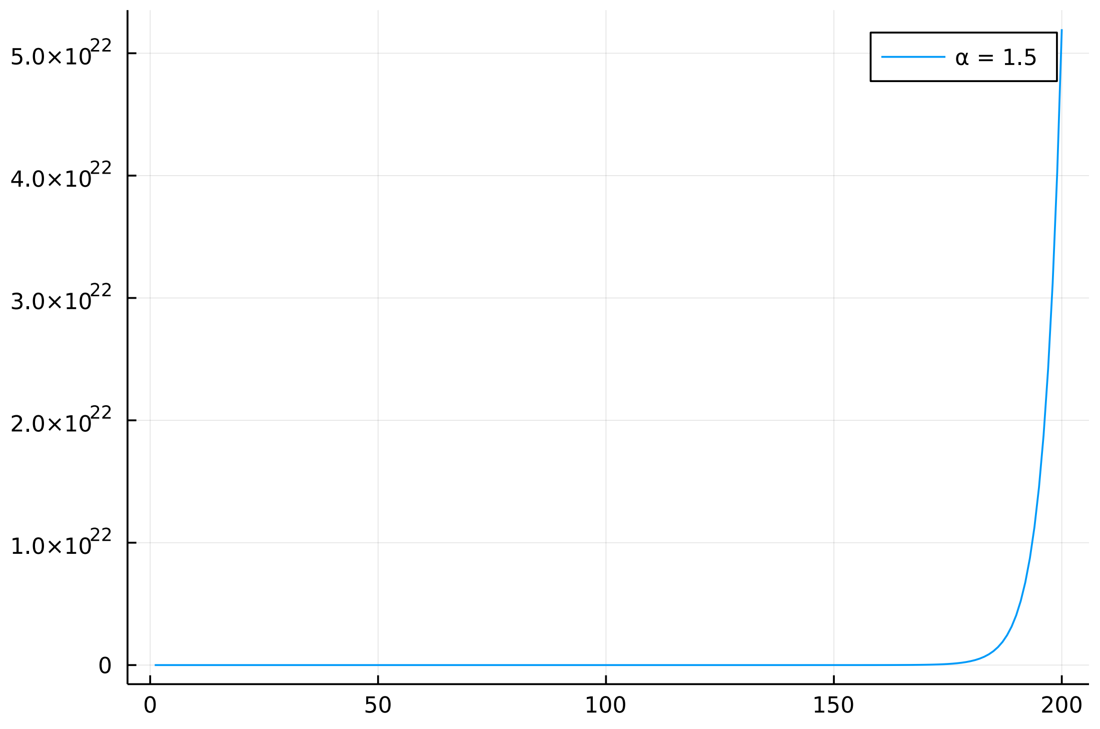
[223]:



But when we see $\alpha = 1.5$ we can see how the error grows exponentially.

[227]:
```
plot(errors_learning(problems[end])[1:200], label = "alpha = 1.5")
```

[227]:

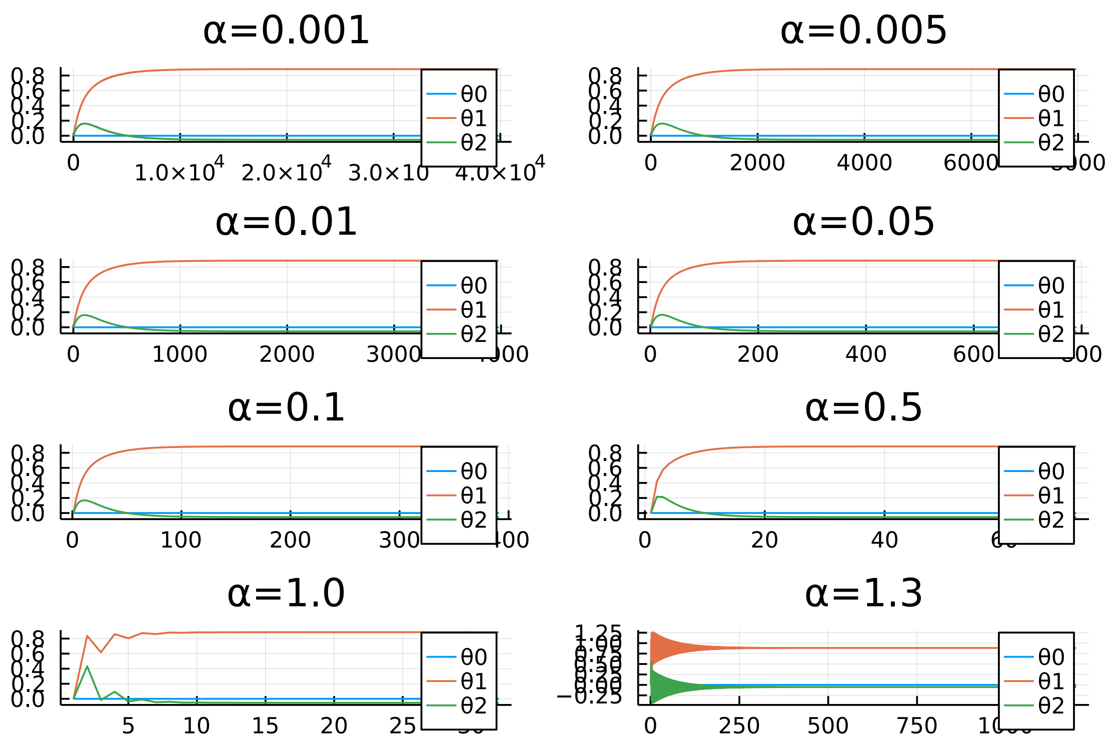


Now we can see how the variables change depending on the learning rate.

We can see that up until $\alpha = 0.5$ it goes smoothly. But in $\alpha = 1$ the variables are getting smaller and bigger until it converges, so it would be very dangerous to have such a high learning rate because it is not as stable. This is further seen in $\alpha = 1.3$.

[228]:
```
p1 = plot(problems[1].theta_hist, title = labels[1], labels = ["theta0" "theta1"␣
  ↪"theta2"])
p2 = plot(problems[2].theta_hist, title = labels[2], labels = ["theta0" "theta1"␣
  ↪"theta2"])
p3 = plot(problems[3].theta_hist, title = labels[3], labels = ["theta0" "theta1"␣
  ↪"theta2"])
p4 = plot(problems[4].theta_hist, title = labels[4], labels = ["theta0" "theta1"␣
  ↪"theta2"])
p5 = plot(problems[5].theta_hist, title = labels[5], labels = ["theta0" "theta1"␣
  ↪"theta2"])
p6 = plot(problems[6].theta_hist, title = labels[6], labels = ["theta0" "theta1"␣
  ↪"theta2"])
p7 = plot(problems[7].theta_hist, title = labels[7], labels = ["theta0" "theta1"␣
  ↪"theta2"])
p8 = plot(problems[8].theta_hist, title = labels[8], labels = ["theta0" "theta1"␣
  ↪"theta2"])
plot(p1,p2,p3,p4,p5,p6,p7,p8, layout =(4,2))
```

[228]:



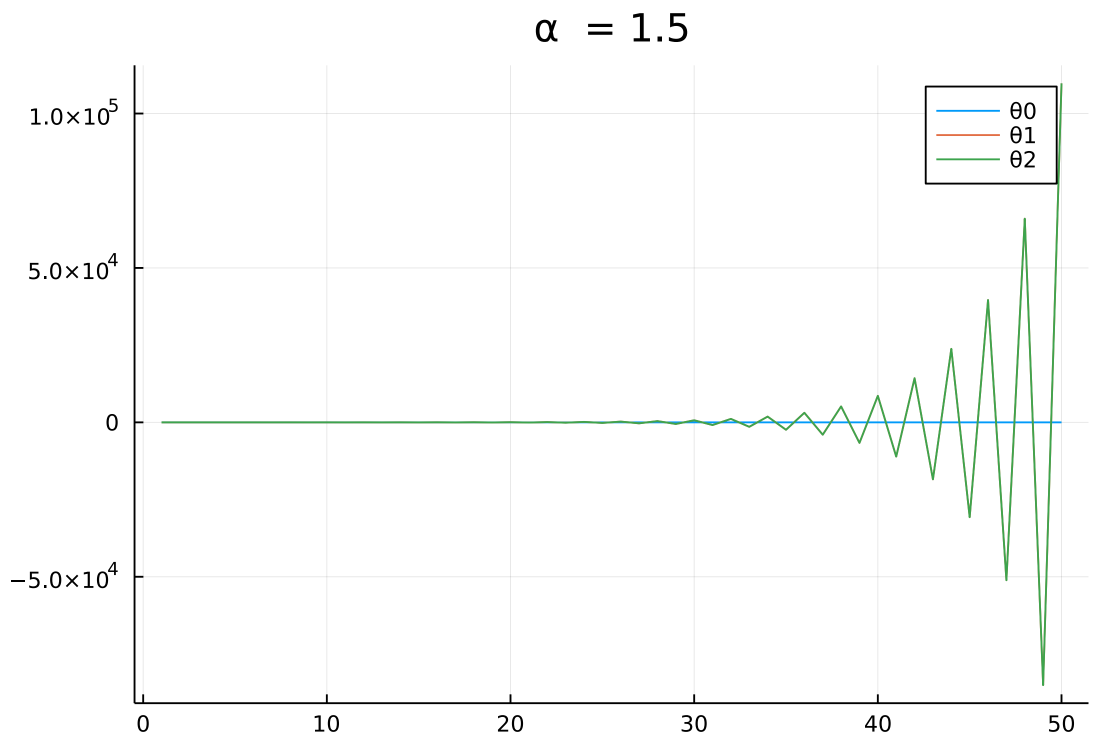
[229]:
```
problems[9].theta_hist[1:10,:]
```

[229]:
```
10×3 Matrix{Float64}:
  0.0           0.0        0.0
 -1.25493e-16   1.25519    0.649277
 -3.02123e-16   0.133899  -0.68651
  2.51378e-16   1.75688    0.860546
 -6.17391e-16  -0.274614  -1.19783
  6.13722e-16   2.36845    1.43572
 -1.05197e-15  -1.03372   -1.96981
  1.19136e-15   3.35841    2.42111
 -1.82194e-15  -2.30717   -3.24489
  2.11495e-15   5.0027     4.06484
```

Here it can be seen how the variables jump back and forth with each jump getting bigger. This is why it diverges, because each iteration the absolute value of the variable will grow exponentially and it will approach infinity.

[238]:
```
plot(problems[9].theta_hist[1:50,:], title = "alpha  = 1.5", labels = ["theta0"␣
 ↪"theta1" "theta2"])
```

[238]:


α = 1.5
θ0
θ1
θ2
1.0×10^5
5.0×10^4
0
−5.0×10^4
0
10
20
30
40
50