I have been learning the statistics for about one year. Nine months ago, I didn't believe that I can survive from statistics classes as I don't know much about statistics. For example, I don't even know that Gaussian distribution is another name of Normal distribution. I just want write something here to summarize what I have learned during this year.

统计学也陆陆续续的学了一年了。没想到自己除了土木工程还是可以学一些其他的东西。现在利用暑假总结一下自己这一年学过的统计学知识，也算是对自己统计学的辅修有一个交代。

What I am writing here is something I summarized using my own words. These "words" may not be accurate enough as I am an Engineering student, not a student majoring in statistics. I just wrote all this stuff based on my own interest. I hope this note could help someone who doesn't has much background in statistics but wants to learn some statistics.

这些所写的内容都是基于我自己的理解，里面所用到的一些词语并不是非常准确。毕竟是我只是一个工程学科的学生，不是一个统计学学生。我写下这些东西也只是基于自己的兴趣爱好，同时希望帮助一些像我一样零基础的同学学习统计学。

---------- starts from here ----------

Background (问题背景):

Assume that both X and Y are a N by 1 vector. X is called as predictor and Y is called as response.

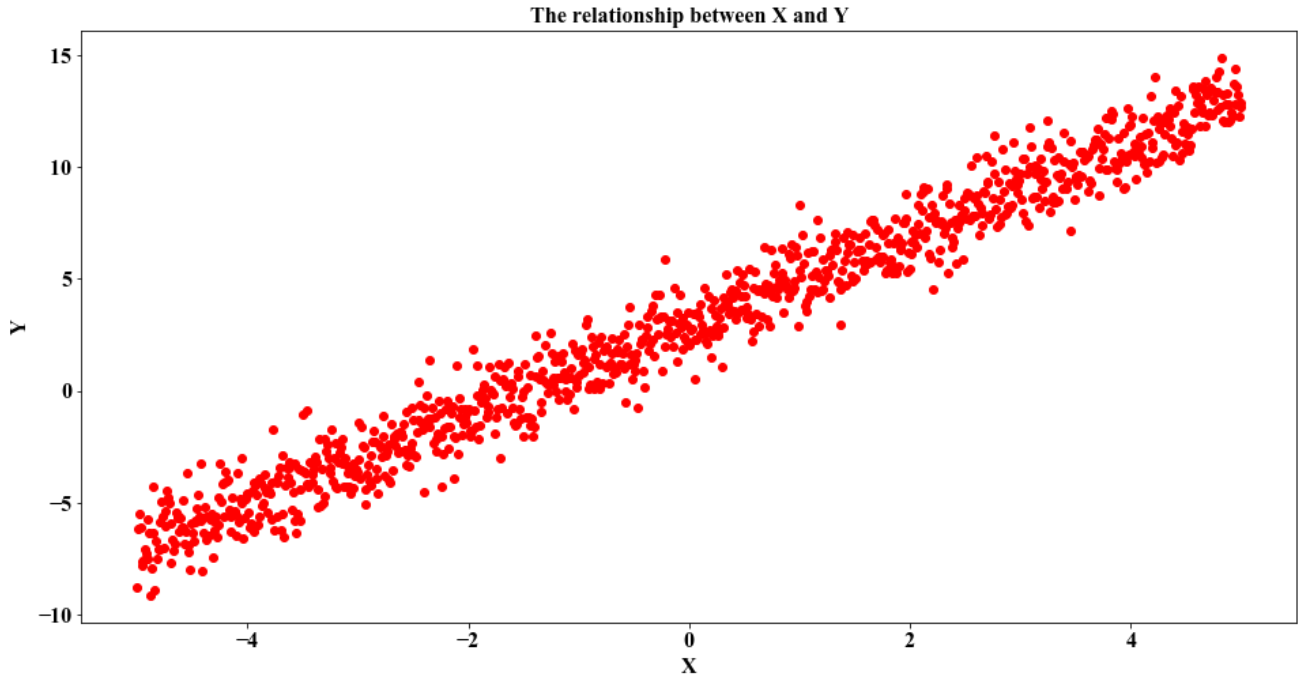
假定X和Y均是一个N行1列的向量（矩阵）。X被称作自变量，Y被称作因变量。

The relationship between X and Y is presented in the figure below (python code is provided):

X与Y之间的关系见下图所示 (python源代码也附在下面)：

```
# Add this command in order to show the plot in Anaconda
# Otherwise the plot may not show up
% matplotlib inline

# Import necessary packages
import matplotlib.pyplot as plt  # Package used for plotting the figure
import numpy as np  # Package for matrix operation

# Define the number of observation
# This number is also the length of X or Y vector
number_observation = 1000

# Generate X using linear space, i.e., X increase from -5 to 5. The interval is a constant such
that there will be 1000 X.
X = np.linspace(-5, 5, number_observation)

# Generate Y
# Assume Y is generated using the equation: Y = 2X + 3 + e
# e refers to the noise, which may be caused by measurement error.
# e is assumed to follow normal distribution with a mean of 0 and a standard deviation of 1
e = np.random.normal(0, 1, number_observation)
Y = 2*X + 3 + e
```

```
# Plot the X and Y
plt.figure(figsize=(16,8))
plt.scatter(X, Y, color='r')
font = {'family':'Times New Roman','weight' : 'normal','size': 16,}
plt.xlabel('X', font)
plt.ylabel('Y', font)
plt.xticks(fontsize=16, fontname='Times New Roman')
plt.yticks(fontsize=16, fontname='Times New Roman')
plt.title('The relationship between X and Y', font)
plt.show()
```



Now we roughly know the relationship between X and Y. It seems that Y is linearly dependent on X. Therefore we try to use a linear function to fit these data points.

现在我们大致了解X与Y之间的关系。通过图可以看出，Y似乎和X呈现出一种线性关系。因此我们需要用线性函数来拟合这些离散的数据点。

The linear function is presented below:

线性函数的形式如下所示：

$h(x_i) = \beta_1 X_i + \beta_0$

Where $h(x_i)$ is the predicted response when x_i is given.

函数中，$h(x_i)$是当给定X_i值时，预测的Y值。

Now, we only need to determine the values for β_1 and β_0 such that $h(x_i)$ is close to Y_i as much as possible.

现在，我们需要确定β_1和β_0的值，使得我们预测出的每一个 $h(x_i)$都尽可能的接近真实值y_i.

Therefore, we need to create a indicator to quantify how close the predicted response is close to actual response. The mean squared error loss function is used here as the indicator:

因此我们需要建立一个量化指标来衡量到底预测值$h(x_i)$与真实值y_i有多接近。我们将使用平方和损失函数（目标函数）作为这个量化指标：

$L(\beta_0, \beta_1) = \frac{1}{N} \sum_{i=1}^{N} (\beta_0 + \beta_1 x_i - y_i)^2$

Based on this loss function, we try to find optimized values for β_0 and β_1 such that the loss is minimized.

借助于这个损失函数，我们试图寻找β_0和β_1的最优解，在这个最优解下，损失函数的值最小。

If this problem is solved manually, we can just take the derivative of the loss function with respect to β_1 or β_0. In this case, we can obtain the optimized values directly.

如果我们试图用手算来解决这个问题，我们可以直接将损失函数对β_0和β_1求导数，即可获得β_0和β_1的解。

Here, we would like to solve this problem using computer programming. Therefore, optimization algorith gradient descent is involved.

这里，我们想写计算机代码来解决这个问题，因此我们要用到叫做“梯度下降”的优化算法。

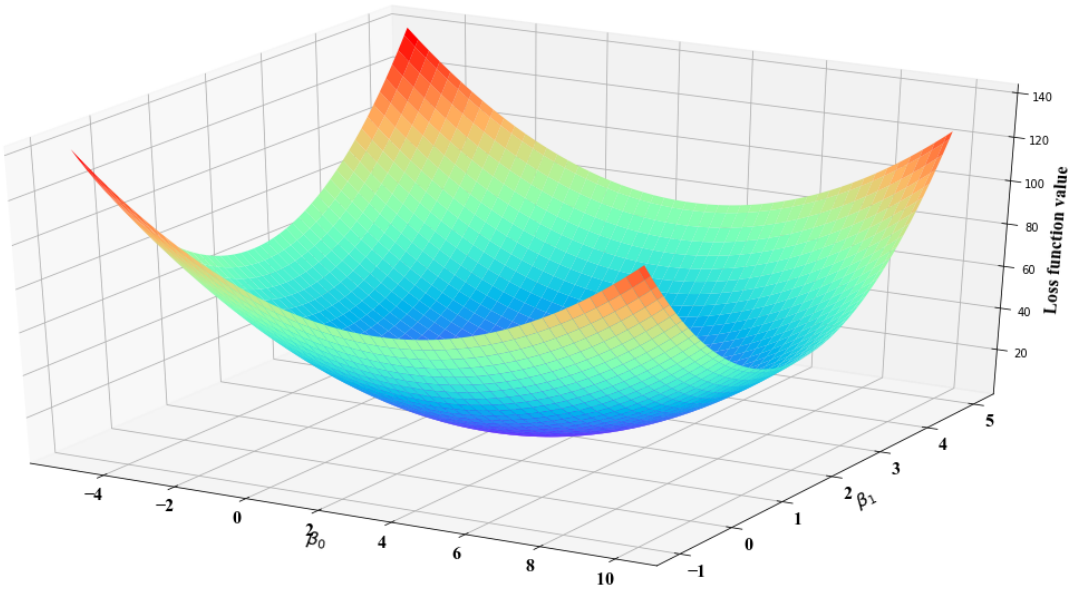
Graident descent will be introduced later in this session.

梯度下降的算法将会在稍后进行介绍。

The plot for the loss function is presented below (python code is also provided):

平方和损失函数如下图所示（python源代码一并提供）：

```
# Import the package for three-dimensional plot
from mpl_toolkits.mplot3d import Axes3D

# Generate values for beta_0 and beta_1
number_beta = 100
beta_0 = np.linspace(-5, 10, number_beta)
beta_1 = np.linspace(-1, 5, number_beta)

# Create a grid to evaluate loss function values on the grid
# You can either use command in line #11 or comand in line #18, 19, 23, 24 to create grid
# grid_beta_0, grid_beta_1 = np.meshgrid(beta_0, beta_1)

# Calculate the loss function values under different beta_0 and beta_1
# Initialize a matrix to store loss function values
loss = np.zeros([number_beta, number_beta])

# Initialize a matrix to store grid values
grid_beta_0 = np.zeros([number_beta, number_beta])
grid_beta_1 = np.zeros([number_beta, number_beta])

for i in range(number_beta):
    for j in range(number_beta):
        grid_beta_0[i,j] = beta_0[i]
        grid_beta_1[i,j] = beta_1[j]
        predict_response = beta_0[i] + beta_1[j] * X
        loss[i,j] = np.dot((predict_response - Y).T, (predict_response-Y))/number_observation

# Plot the loss function values over the beta_0 and beta_1 values
fig = plt.figure(figsize=(16,8))
ax = Axes3D(fig)
ax.set_xlabel(r"$\beta_0$", font)
ax.set_ylabel(r"$\beta_1$", font)
ax.set_zlabel("Loss function value", font)
plt.xticks(fontsize=16, fontname='Times New Roman')
plt.yticks(fontsize=16, fontname='Times New Roman')
ax.plot_surface(grid_beta_0, grid_beta_1, loss, cmap='rainbow')
plt.show()
```



The figure above shows that a global minimum exists for this loss function. This global minimum is the case that we want to achieve because the loss is smallest. In other words, the predicted response is closest to the actual response.

上图展示了我们损失函数有一个全局最小值，而这个最小值正是我们试图寻找的。该最小值表征了我们预测的Y值与真实Y值是最接近的。

As mentioned above, we are using gradient descent to find the global minimum. For those who are not familiar with gradient descent, please refer to the following link for more information:

https://en.wikipedia.org/wiki/Gradient_descent

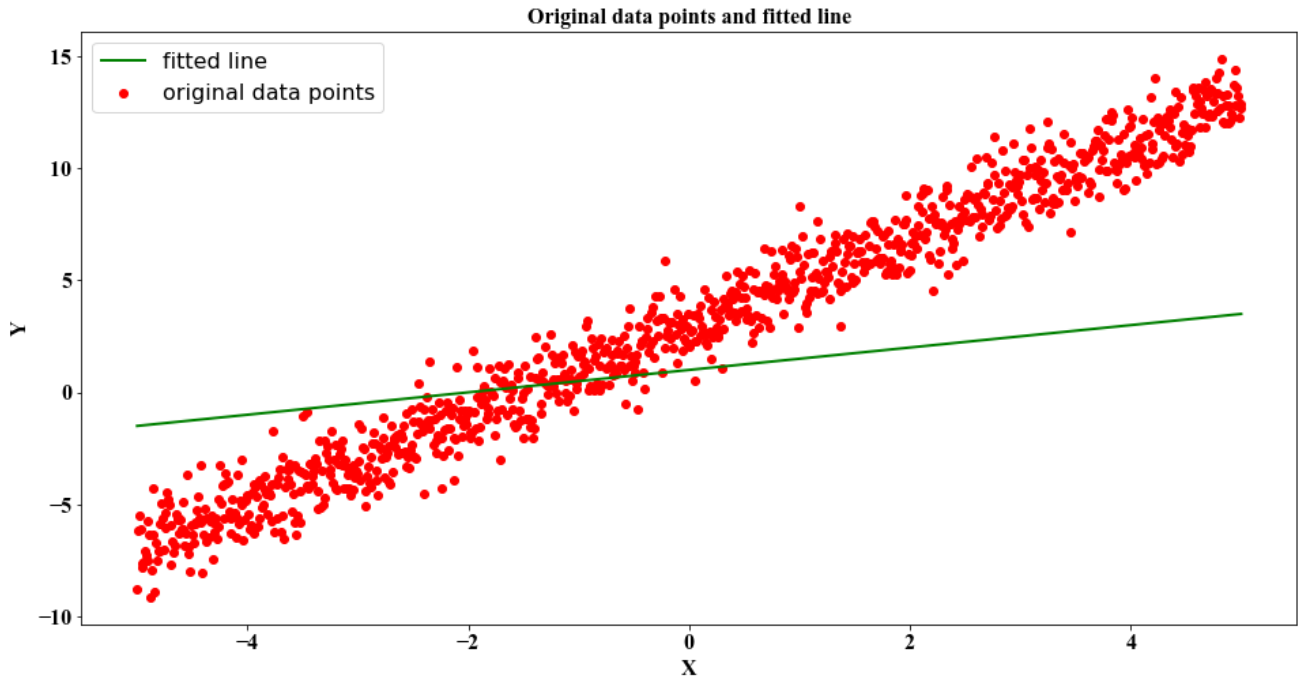
如上文提到的，我们将使用梯度下降方法来寻找全局最小值。对于还不是很熟悉梯度下降概念的同学，可以点击下面链接来了解梯度下降方法：

https://baike.baidu.com/item/梯度下降/4864937?fr=aladdin

For gradient descent algorithm, the first step is to give arbitary values for β_0 and β_1.

梯度下降方法中，我们先随机赋予β_0和β_1两个数值。

```
# These two values are given arbitrarily
# You can give any values as you like
beta_0 = 1
beta_1 = 0.5
predict_response = beta_0 + beta_1 * X

# Visualize the fitted line
plt.figure(figsize=(16,8))
plt.scatter(X, Y, color='r', label='original data points')
plt.plot(X, predict_response, color='g', linewidth=2.0, label='fitted line')
font = {'family':'Times New Roman','weight' : 'normal','size': 16,}
plt.xlabel('X', font)
plt.ylabel('Y', font)
plt.xticks(fontsize=16, fontname='Times New Roman')
plt.yticks(fontsize=16, fontname='Times New Roman')
plt.title('Original data points and fitted line', font)
plt.legend(fontsize=16)
plt.show()
```



As shown in the figure above, the initial fitted line does not match the data points. This is not surprised as the values of β_0 and β_1 are given arbritarily.

上图展示了拟合的直线与离散的数据点吻合并不是很好。这并不出乎意料，因为我们只是随意的给β_0和β_1赋值。

Don't worry, we will use gradient descent to "train" β_0 and β_1. Eventually, a good fitted line will be obtained.

我们会用梯度下降来“训练”β_0和β_1，使得最终获得的较好的拟合效果。

We need to find the gradient of the loss function, i.e., the derivative with respect to β_0 and β_1:

首先，我们需要寻找函数的“梯度”，即关于β_0和β_1的导数：

$$\frac{\partial L(\beta_0,\beta_1)}{\partial \beta_0} = \frac{2}{N}\sum_{i=1}^{N}(\beta_0 + \beta_1 x_i - y_i)$$

$$\frac{\partial L(\beta_0,\beta_1)}{\partial \beta_1} = \frac{2}{N}\sum_{i=1}^{N}(\beta_0 + \beta_1 x_i - y_i)x_i$$

Then we just need to use the negative gradient direction to update β_0 and $beta_1$

然后，我们沿着梯度的负方向来迭代更新β_0和β_1两个参数。

$$\beta_0^{n+1} = \beta_0^n - s\frac{\partial L(\beta_0^n,\beta_1^n)}{\partial \beta_0^n}$$

$$\beta_1^{n+1} = \beta_1^n - s\frac{\partial L(\beta_0^n,\beta_1^n)}{\partial \beta_1^n}$$

where s is the step size. Step size cannot be neither too large nor too small. If it is too large, then we may miss the global minimum value. If it is too small, this update algorithm would be time-consuming.

表达式中的s是代表步长。步长不能太长，也不能太短。步长如果太长，会使得我们错过了全局最小值。如果步长太短，会使得算法花很长的时间来收敛。

Then the question is how to determine the value for step size. Honestly, I don't know the answer. Evertime I did this. I always given an arbitrary value for the step size. If it doesn't work well, then I just change the value for step size, and try again. It's like a trial and error process.

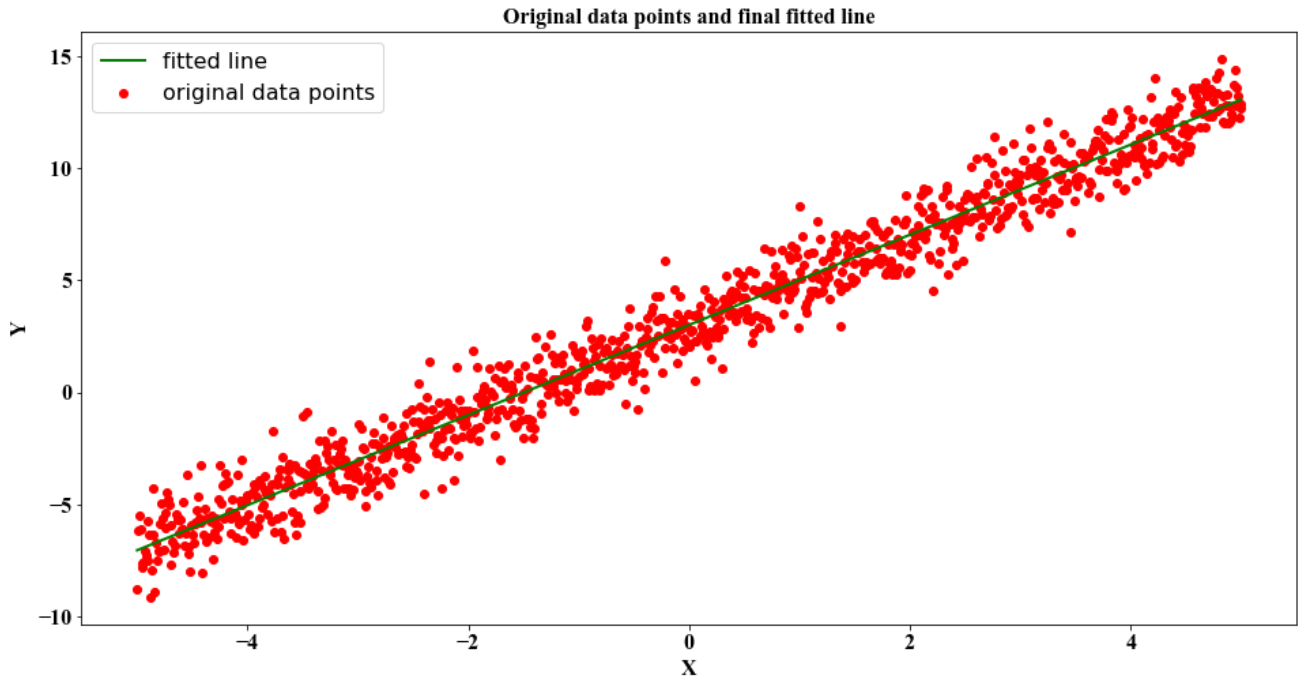
那么如何确定步长的大小。这是一个“玄学”。至少我本人不是很了解如何确定。我通常的做法是给定一个步长，跑整个程序。如果结果不好，我再修改步长，再运行，直到结果比较满意为止。

The following section gives the code on how we perform gradience descent algorithm. The final results are also shown below.

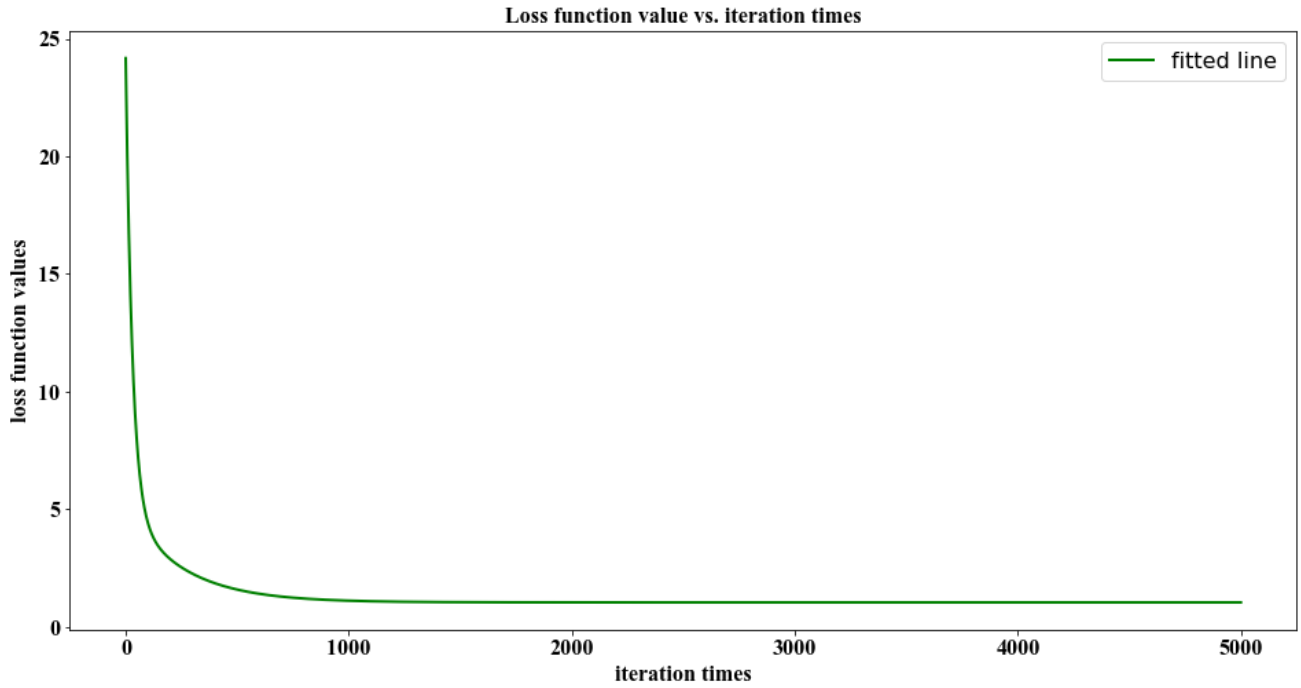
接下来的代码展示了我们如何使用梯度下降方法。最终结果也一并画图呈现。

```
# Assume we are going to iterate 5000 times
max_iteration = 5000

# Assume step size is 0.001
step_size = 0.001

# Initialize beta_0, beta_1, and loss to track the update algorithm
beta_0_update = np.zeros([max_iteration+1, 1])
beta_1_update = np.zeros([max_iteration+1, 1])
loss_update = np.zeros([max_iteration+1, 1])

# Initial value for beta_0 and beta_1
beta_0_update[0,0] = beta_0
beta_1_update[0,0] = beta_1

# Iterate many times in order to get the best values for beta_0 and beta_1
for iter in range(max_iteration):
    predict_response = beta_0_update[iter,0] + beta_1_update[iter,0] * X
    difference = predict_response - Y
    loss_update[iter,0] = np.dot(difference.T, difference)/number_observation
    gradient_beta_0 = 2 * np.sum(difference)/number_observation
    gradient_beta_1 = 2 * np.dot(difference, X)/number_observation
    beta_0_update[iter+1] = beta_0_update[iter] - step_size * gradient_beta_0
    beta_1_update[iter+1] = beta_1_update[iter] - step_size * gradient_beta_1
```

```
Y_predict = beta_0_update[-1,0]+ beta_1_update[-1,0] * X

plt.figure(figsize=(16,8))
plt.scatter(X, Y, color='r', label='original data points')
plt.plot(X, Y_predict, color='g', linewidth=2.0, label='fitted line')
font = {'family':'Times New Roman','weight' : 'normal','size': 16,}
plt.xlabel('X', font)
plt.ylabel('Y', font)
plt.xticks(fontsize=16, fontname='Times New Roman')
plt.yticks(fontsize=16, fontname='Times New Roman')
plt.title('Original data points and final fitted line', font)
plt.legend(fontsize=16)
plt.show()
```




```
plt.figure(figsize=(16,8))
plt.plot(range(max_iteration), loss_update[0:max_iteration,0], color='g', linewidth=2.0,
label='fitted line')
font = {'family':'Times New Roman','weight' : 'normal','size': 16,}
plt.xlabel('iteration times', font)
plt.ylabel('loss function values', font)
plt.xticks(fontsize=16, fontname='Times New Roman')
plt.yticks(fontsize=16, fontname='Times New Roman')
plt.title('Loss function value vs. iteration times', font)
plt.legend(fontsize=16)
plt.show()
```



As observed from the plot of loss function value vs. iteration times, the loss already reaches convergence. This means that even we increase the iteration times, the loss would not further decrease. The final fitted line and data points are shown in the figure above.

从损失函数与迭代次数的关系图象可以看出，损失函数早已收敛。这意味着，即便我们增加迭代次数，损失函数的值不会进一步减小。最终拟合曲线与离散点在上文的图也已展示。

OK. We are done. Feel free to leave any comments.

到此结束。