Optimization - Practical 3

November 4, 2024

Theodoros lambrou

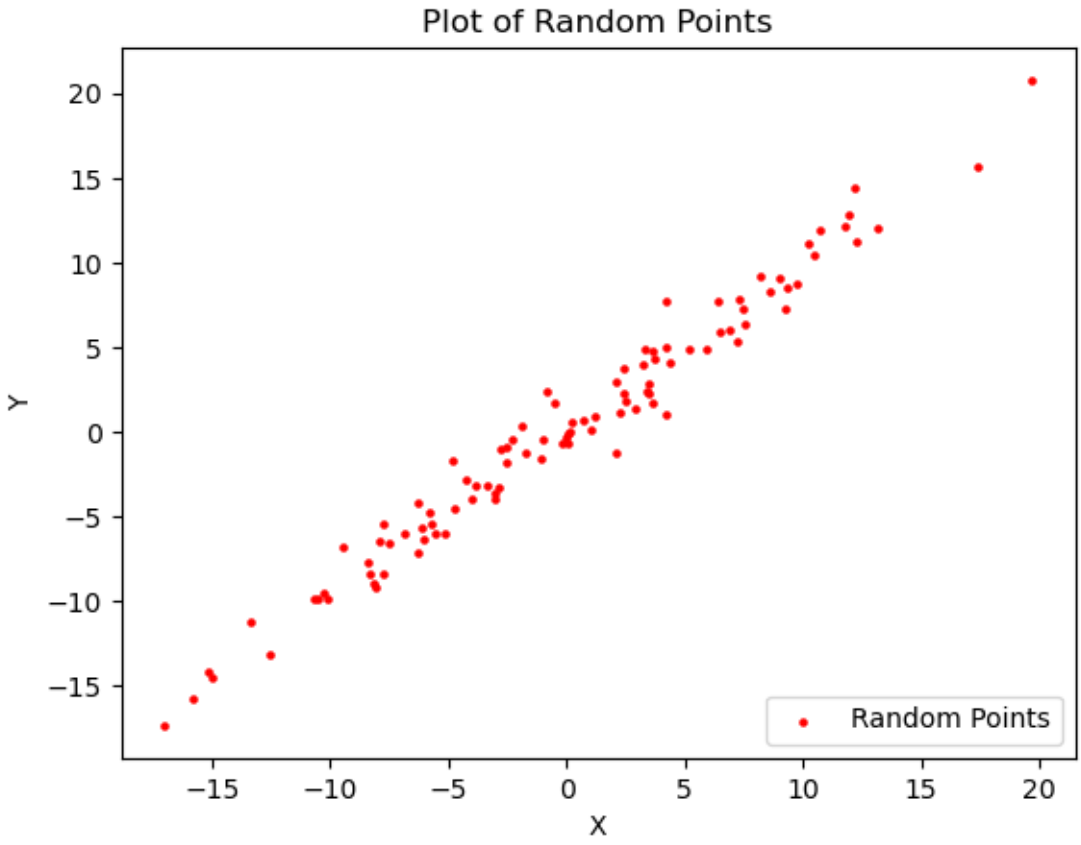
1 Robust Linear Regression

1.1 Introduction

1.1.1 Implement the method proposed with gradient descent and backtracking (or a small constant value). You may check your method with a randomly generated set of points

[1]:
```
import numpy as np
import matplotlib.pyplot as plt
import math

np.random.seed(5)

def generate_random_point(n):
  m = np.array([0., 0.]) # defining the mean vector
  angle = 45 * math.pi / 180
  rot = np.array([[math.cos(angle), -math.sin(angle)], [math.sin(angle), math.
 ↪cos(angle)]])
  lamb = np.array([[n, 0], [0, 1]]) # defininining the covariance matrix
  s = np.matmul(rot, np.matmul(lamb, rot.transpose()))
  points = np.random.multivariate_normal(m, s, n)   # 100 random points

  return points


def plot_points(points):
  # Extraction of x and y co-ordinates
  x = points[:, 0]
  y = points[:, 1]

  # scatter plot of points
  plt.scatter(x, y, marker='o',c='red', s=5, label='Random Points')

  plt.xlabel('X')
  plt.ylabel('Y')
```

```
  plt.title('Plot of Random Points')
  plt.legend(loc=4)
  plt.show()


n = 100 # defining no of points
points = generate_random_point(n)

plot_points(points)
```



Linear regression model: $\hat{y} = w_0 + w_1 x$

Calculating the error using LMS:

$Q = \frac{1}{2} \sum_{i=1}^{n} (\hat{y}_i - y_i)^2$

$Q = \frac{1}{2} \sum_{i=1}^{m} (w_0 x_i + w_1 - y_i)^2$

This is the function to be minimized.

Either the gradient descent or Newton method can be used to find the minimum.

The gradient descent would iteratively apply:

$w_{k+1} = w_k - \alpha_k \nabla Q(w)$

$\nabla Q(w) = (\frac{\partial Q}{\partial w0}, \frac{\partial Q}{\partial w1})$

Where:

$\frac{\partial Q}{\partial w_0} = \sum_{i=1}^{m} (w_0 x_i + w_1 - y_i) x_i$

$\frac{\partial Q}{\partial w_1} = \sum_{i=1}^{m} (w_0 x_i + w_1 - y_i)$

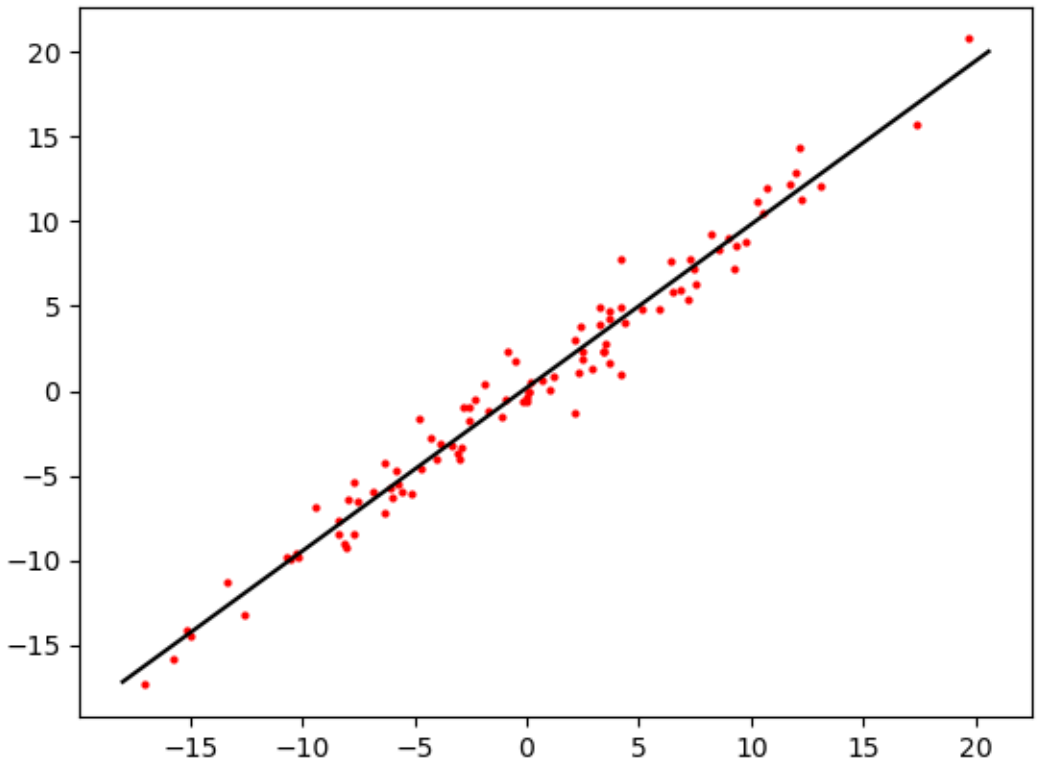
Defining the function to be optimized and its gradient.

[2]:
```
def function(x,y,w0,w1):
    return 1/2*(((w0*x+w1-y)**2).sum())

def grad_func(x,y,w0,w1):
    return np.array([((w0*x+w1-y)*x).sum(), (w0*x+w1-y).sum()]) #gradient␣
 ↪calculation

#gradient descent optimization
def grad_descent(f,x,y,w0 = 0, w1 = 0, max_iters = 100, threshold = (10**-3)):
    coeffs = np.array([w0,w1])
    steps = np.matrix(coeffs) #the optimization steps will be saved in a matrix
    steps = np.vstack([steps, coeffs])

    #doing a max. of max_iters iterations
    for i in range(1, max_iters):
        alpha = 1 #step size
        coeffs_2 = coeffs - alpha * grad_func(x,y,coeffs[0],coeffs[1])

        while(f(x, y, coeffs_2[0], coeffs_2[1]) > f(x,y,coeffs[0],coeffs[1])):
            alpha= alpha / 2 #alpha is updated as long as not decreaseing
            coeffs_2 = coeffs - alpha * grad_func(x,y,coeffs[0],coeffs[1])

        if (f(x,y,coeffs[0],coeffs[1]) - f(x,y,coeffs_2[0],coeffs_2[1])) <␣
 ↪threshold:
            i = i-1
            break #ending the loop if the change is smaller than treshold

        coeffs = coeffs_2.copy()
        steps = np.vstack([steps, coeffs])
        f_value = f(x,y,coeffs[0],coeffs[1])

    return coeffs, steps, f_value, i+1
```

[3]:
```
coeffs, steps, f, iters = grad_descent(f = function,x = points[:, 0],y =␣
 ↪points[:, 1], w0 = 0, w1 = 0, max_iters = 1000, threshold = 10**-6)
```

[4]:
```
plt.scatter(points[:,0],points[:,1], c='red',s=4)
reg = np.arange(points[:, 0].min() - 1, points[:, 0].max() + 1,0.1)
plt.plot(reg, reg * coeffs[0] + coeffs[1],color = 'black')
print("Coefficients:", coeffs[0], coeffs[1])
```

```
Coefficients: 0.9644792569601512 0.18229370412293203
```



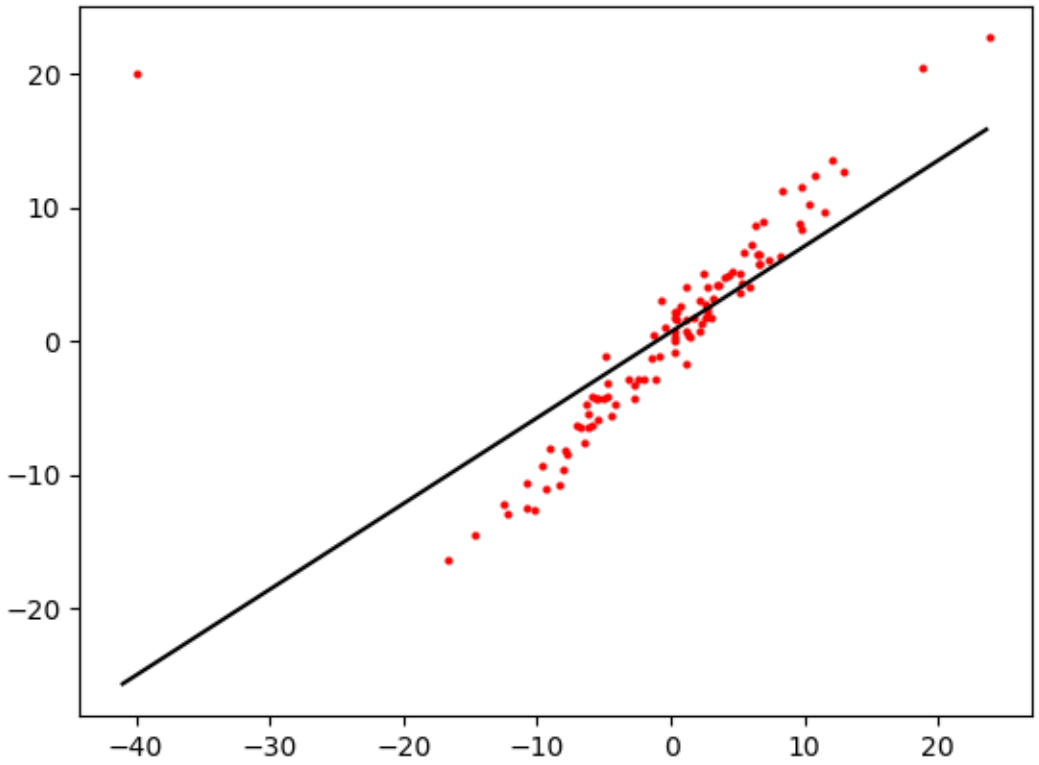
The regressor looks to be accurate since the data is scattered well.

1.1.2 Let us now check the sensitivity of the method to outliers, that is, points that do not follow the model. Using a set of points generate with an angle of 45 degrees, change the value of one point to a value "far away" from the set of points, for instance points[1] = [-40,20]. Draw the line that approximates the set of points and observe that one point may have a large influence in the obtained solution. Change now other points of the original dataset to assess the influence of outliers.

[5]:
```
points = generate_random_point(n=100)
points[1] = [-40, 20]
#points[2] = [-20,10]

x1 = points[:,0]
x2 = points[:,1]
```

```
coeffs,steps,f,iters = grad_descent(f = function, x = x1, y = x2, w0 = 0, w1 =␣
↪0, max_iters = 1000, threshold = 10**-6)

plt.scatter(points[:,0],points[:,1], c='red',s=4)
x_aux = np.arange(x1.min() - 1, x2.max() + 1, 0.1)
plt.plot(x_aux, x_aux*coeffs[0] + coeffs[1], color = 'black')

print("Coefficients:", coeffs[0], coeffs[1])
```

```
Coefficients: 0.6422607426507068 0.6645742240425668
```



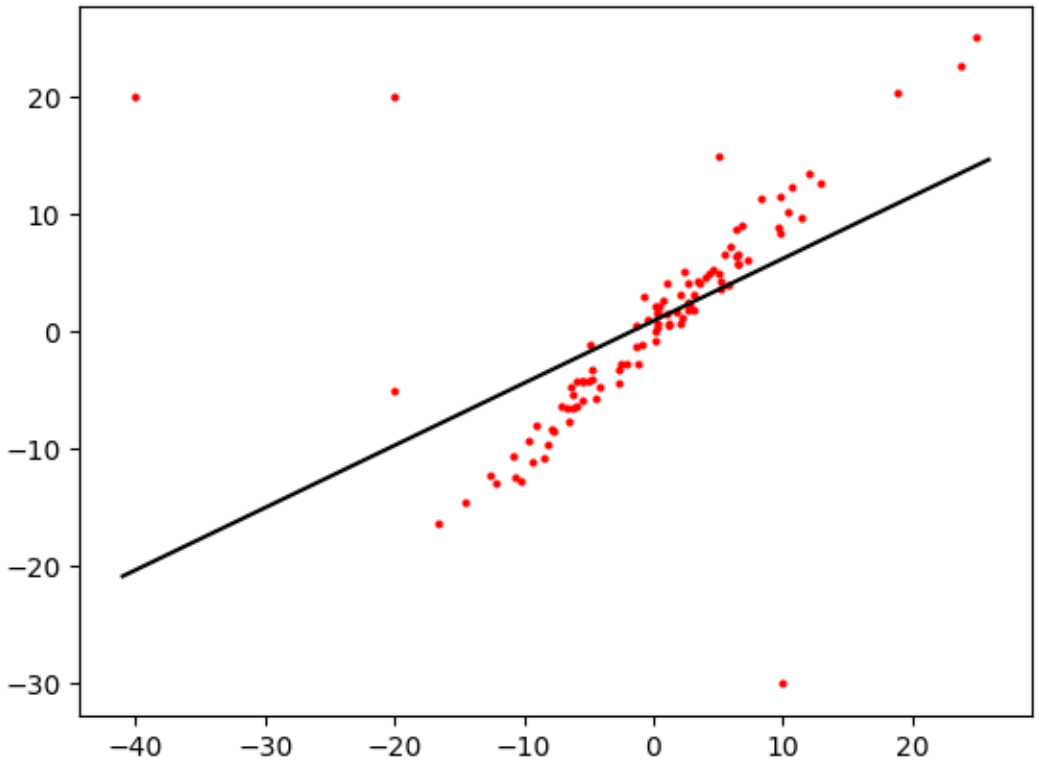
It can be seen that one outlier has indeed a large influence in the obtained solution meaning the model has high sensitivity to outliers.

Repeating with more outliers:

[6]:
```
points[5] = [5,15]
points[10] = [-20,20]
points[20] = [10,-30]
points[40] = [25,25]
points[55] = [-20,-5]
```

```
x1 = points[:,0]
x2 = points[:,1]

coeffs, steps, f, iters = grad_descent(f = function, x = x1, y = x2, w0 = 0, w1␣
 ↪= 0, max_iters = 1000, threshold = 10**-6)
print("Coefficients:", coeffs[0], coeffs[1])

plt.scatter(points[:,0],points[:,1], c= 'red',s=4)
x_aux = np.arange(x1.min() - 1, x2.max() + 1, 0.1)
plt.plot(x_aux, x_aux*coeffs[0] + coeffs[1], color = 'black')
print("Coefficients:", coeffs[0], coeffs[1])
```

```
Coefficients: 0.5302609555448059 0.9046940891673342
Coefficients: 0.5302609555448059 0.9046940891673342
```



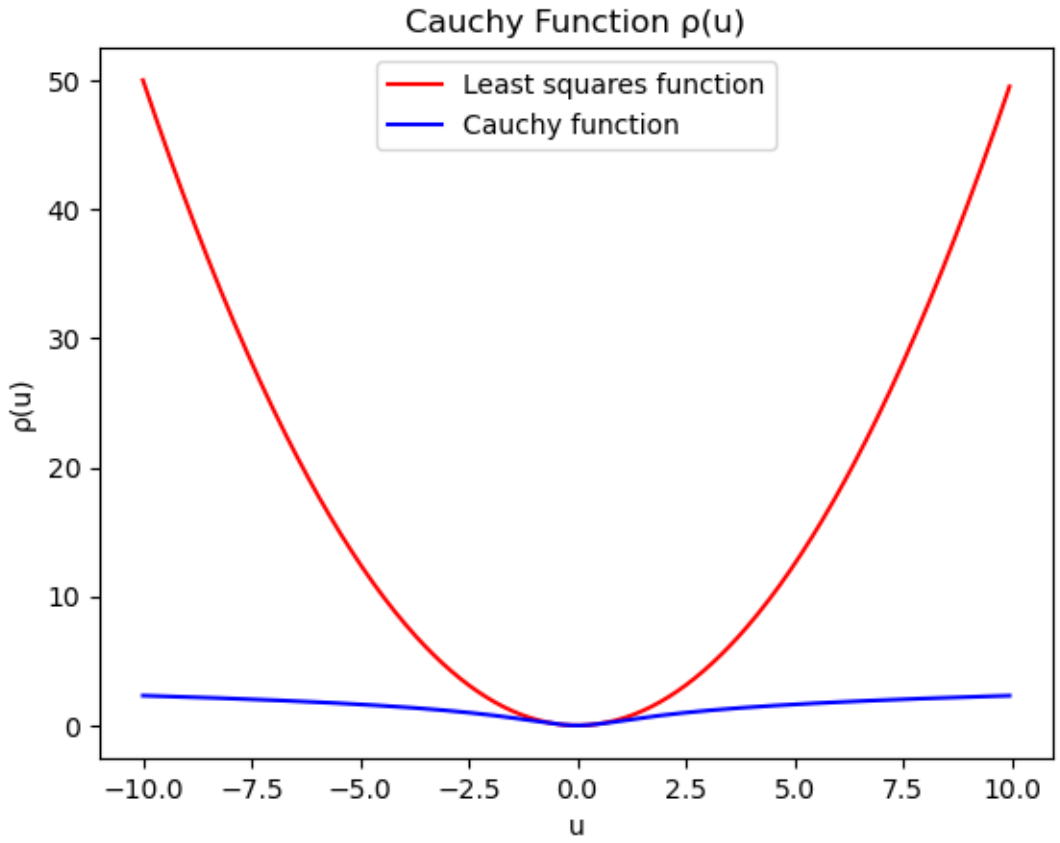
It can be seen that with the addition of more outliers, the sensivitity increases further confirming the influence of the outliers.

1.2 Robust functions

1.2.1 Plot the least squares function $\rho(u) = \frac{1}{2}u^2$, and compare it with the Cauchy function, Eq.(2), in order to see the "importance" given to each prediction error u, you may, for instance, plot the function (u) for |u| 10.

[7]:
```
u_function = np.arange(-10, 10, 0.05)
c = 1

plt.plot(u_function, 1/2*(u_function**2), color='red', label="Least squares␣
↪function")
plt.plot(u_function, (c**2/2) * np.log(1 + (u_function/c)**2), color='blue',␣
↪label="Cauchy function")
plt.title('Cauchy Function  (u)')
plt.xlabel('u')
plt.ylabel(' (u)')
plt.legend()
```

[7]: <matplotlib.legend.Legend at 0x115d4c890>



We can see that the Cauchy's function curve is more smooth while the LS function curve is more

sensitive to the variability.

1.2.2 Implement the algorithm that allows to compute the parameters w0 and w1 using the Cauchy function. For that issue you can use the backtracking gradient descent method (there is no need to use the Newton method).

[8]:
```
def cauchy(x, y, w0 = 0, w1 = 0, c = 1):
    return ((c**2)/2*np.log(1 + (((w0*x)+w1-y)/c)**2)).sum()

def grad_cauchy(x, y, w0, w1, c):
    return np.array([((c**2)/2*1/(1+(((w0*x)+w1-y)/c)**2)*2*(w0*x+w1-y)/
 ↪(c**2)*x).sum(), (c**2/2*1/(1+((w0*x+w1-y)/c)**2)*2*(w0*x+w1-y)/(c**2)).
 ↪sum()])

def grad_descent_cauchy(f, x, y, w0 = 0, w1 = 0, max_iters = 100, threshold =␣
 ↪10**-3, c = 1):
    coeffs = np.array([w0, w1])
    steps = np.matrix(coeffs)
    steps = np.vstack([steps,coeffs])
    for i in range(1, max_iters):
        alpha = 1
        coeffs_2 = coeffs - alpha * grad_cauchy(x, y, coeffs[0], coeffs[1], c =␣
 ↪c)
        while(f(x, y, coeffs_2[0], coeffs_2[1], c = c) > f(x, y, coeffs[0],␣
 ↪coeffs[1], c = c)):
            alpha = alpha / 2
            coeffs_2 = coeffs - alpha * grad_cauchy(x, y, coeffs[0], coeffs[1],␣
 ↪c = c)
        if (f(x, y, coeffs[0], coeffs[1], c = c) - f(x, y, coeffs_2[0],␣
 ↪coeffs_2[1], c = c)) < threshold:
            i -= 1
            break
        coeffs = coeffs_2.copy()
        steps = np.vstack([steps,coeffs])
    f_value = f(x, y, coeffs[0], coeffs[1], c = c)

    return coeffs, steps, f_value, i+1
```

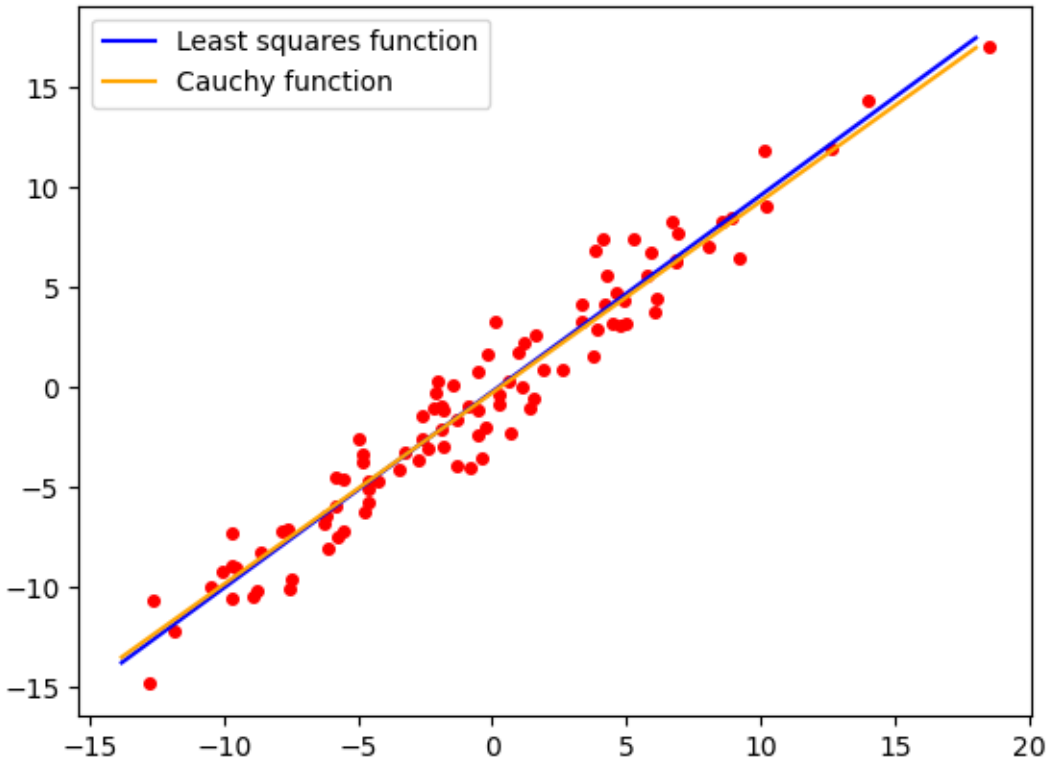
1.2.3 Compare the results obtained with the least squares function and with the Cauchy function, assuming that there are no outliers in the dataset.

[9]:
```
points1 = generate_random_point(100) #no outliers
x11 = points1[:, 0]
x22 = points1[:, 1]
coeffs, steps, f, iters = grad_descent(function, x11, x22, 0, 0, 100, 10**-3)
coeffs_2, steps_2, f_2, iters_2 = grad_descent_cauchy(cauchy, x11, x22, 0, 0,␣
 ↪100, (10**-3))
```

```
plt.plot(x11, x22, 'bo', markersize=4, c='red')
plt.plot()
reg_2 = np.arange(x11.min() - 1, x22.max() + 1, 0.1)
plt.plot(reg_2, reg_2*coeffs[0] + coeffs[1], c='blue', label="Least squares␣
 ↪function")
plt.plot(reg_2, reg_2*coeffs_2[0] + coeffs_2[1], c='orange', label="Cauchy␣
 ↪function")
plt.legend()
```

```
/var/folders/ng/1q_6q8_n1fx35q947ctj0w1m0000gn/T/ipykernel_81338/1478202535.py:7
: UserWarning: color is redundantly defined by the 'color' keyword argument and
the fmt string "bo" (-> color='b'). The keyword argument will take precedence.
  plt.plot(x11, x22, 'bo', markersize=4, c='red')
```

[9]: `<matplotlib.legend.Legend at 0x1161ecb00>`



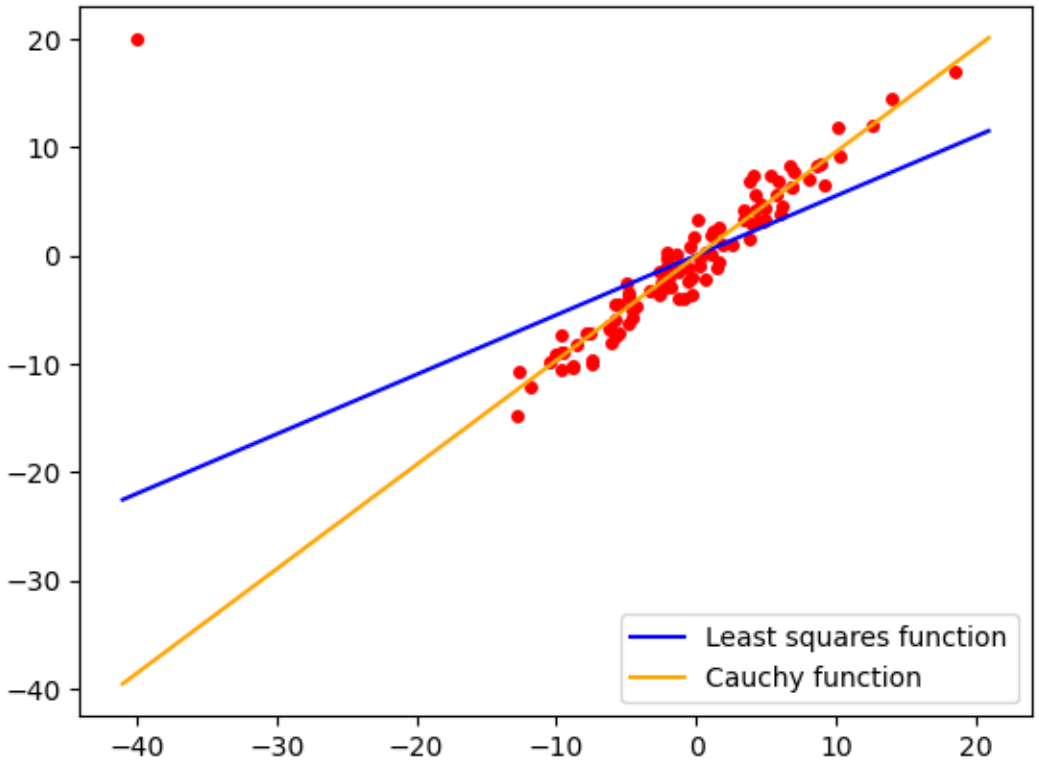
No significant differences spotted between the 2 methods

1.2.4 Compare now the results with only one outlier. You may proceed as previously proposed. The Cauchy function should be more robust than the quadratic function.

[10]:
```
points1[1] = [-40,20] #sample outlier
x11 = points1[:, 0]
x22 = points1[:, 1]
coeffs, steps, f, iters = grad_descent(function, x11, x22, 0, 0, 100, 10**-3)
coeffs_2, steps_2, f_2, iters_2 = grad_descent_cauchy(cauchy, x11, x22, 0, 0,␣
 ↪100, 10**-3, c = 1)

#Lets plot the functions
plt.plot(x11, x22, 'bo', markersize=4, c='red')
plt.plot()
reg_2 = np.arange(x11.min() - 1, x22.max() + 1, 0.1)
plt.plot(reg_2, reg_2 * coeffs[0] + coeffs[1], c = 'blue', label = "Least␣
 ↪squares function")
plt.plot(reg_2, reg_2 * coeffs_2[0] + coeffs_2[1], c = 'orange', label =␣
 ↪"Cauchy function")
plt.legend()
```

```
/var/folders/ng/1q_6q8_n1fx35q947ctj0w1m0000gn/T/ipykernel_81338/660842997.py:8:
UserWarning: color is redundantly defined by the 'color' keyword argument and
the fmt string "bo" (-> color='b'). The keyword argument will take precedence.
  plt.plot(x11, x22, 'bo', markersize=4, c='red')
```

[10]:
```
<matplotlib.legend.Legend at 0x1162a96d0>
```

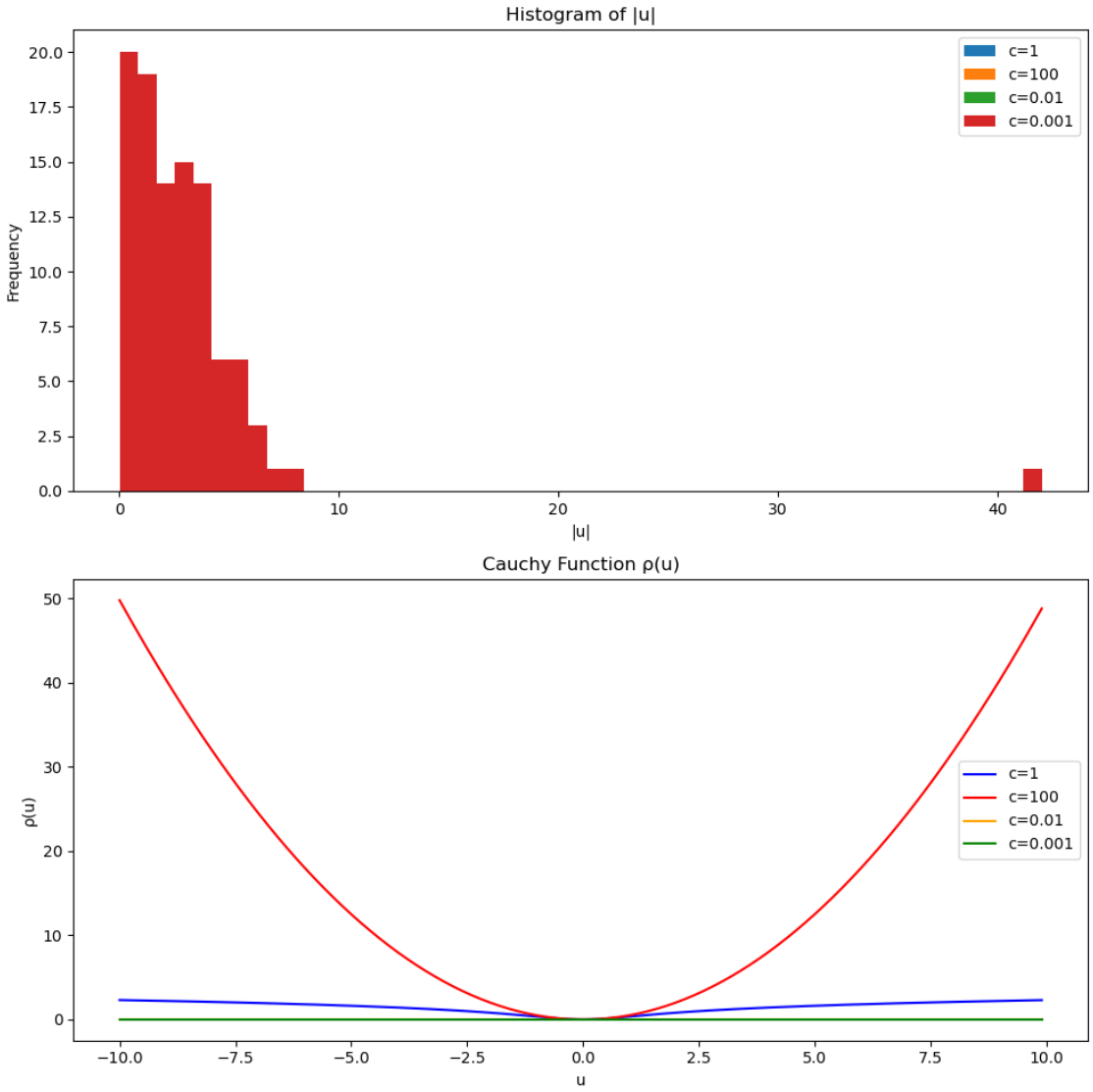


It can be observed that the Cauchy function is better than the least squares function when there are outliers.

1.2.5 Test the influence of the parameter c in the parameters obtained. You may, for instance, check the results obtained with $c = 1$, $c = 100$, $c = \frac{1}{100}$ and $c = \frac{1}{1000}$. Can you reason why of these results? To this end, you are recommended to plot the histogram of the error function $|u|$ and to compare with the shape corresponding $\rho(u)$ functions. Consider performing a "zoom" of the Cauchy function to see the interval at which the function behaves as a quadratic function. Which values are considered as "inliers" / "outliers"?

[11]:
```
cauchy_c = [1, 100, 1/100, 1/1000]
colors = ['blue', 'red', 'orange', 'green']

error_functions = []
# calculating error function |u| for each c
for c in cauchy_c:
    error = np.abs(x22-coeffs[0] * x11-coeffs[1])
    error_functions.append(error)

# Generate a range of values for the Cauchy function
u_values = np.arange(-10, 10, 0.1)
```

```
cauchy_functions = [c**2 / 2 * np.log(1 + (u_values / c)**2) for c in cauchy_c]

# Create a figure with two rows and one column for the subplots
fig, axs = plt.subplots(2, 1, figsize=(10,10))

for i in range(len(cauchy_c)):
    #histogram of |u|
    axs[0].hist(error_functions[i], bins=50,label=f'c={cauchy_c[i]}')
    axs[0].set_title('Histogram of |u|')
    axs[0].set_xlabel('|u|')
    axs[0].set_ylabel('Frequency')
    axs[0].legend()

    #Cauchy function  (u)
    axs[1].plot(u_values, cauchy_functions[i], color=colors[i],␣
 ↪label=f'c={cauchy_c[i]}')
    axs[1].set_title('Cauchy Function  (u)')
    axs[1].set_xlabel('u')
    axs[1].set_ylabel(' (u)')
    axs[1].legend()

plt.tight_layout()
plt.show()
```



It can be clearly observed that bigger values of the parameter c have bigger penalization.

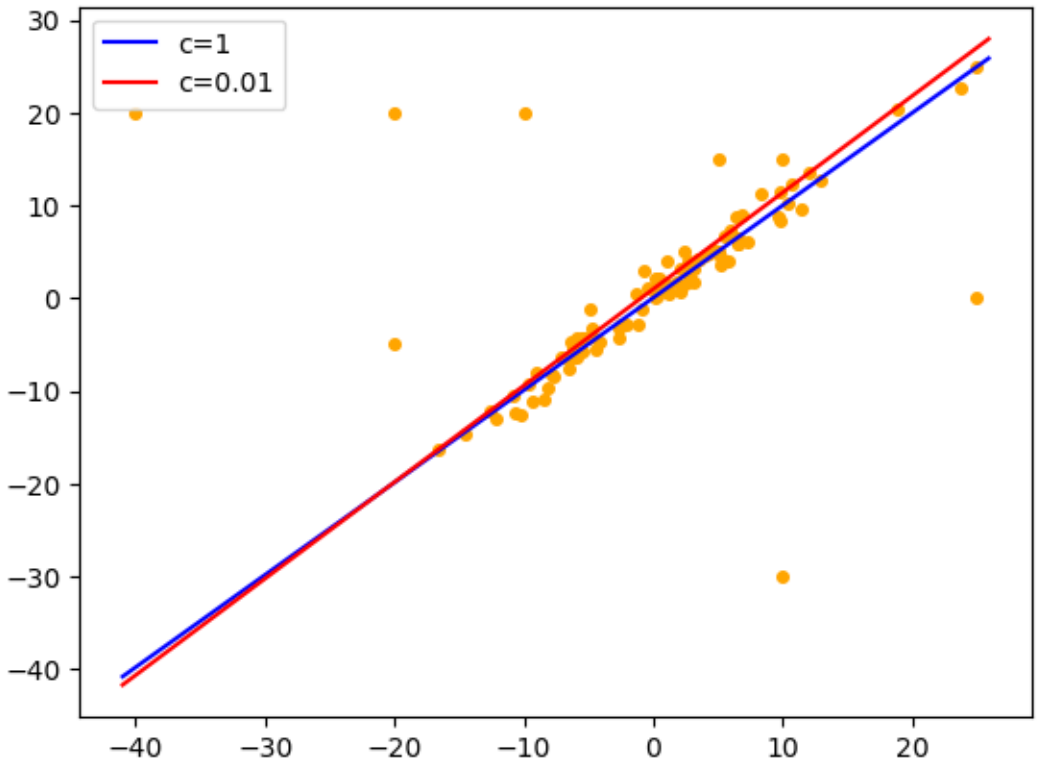
1.2.6 The Cauchy function is not "perfect", and it is not robust for any number of outliers. Using $c = 1$ and $c = 1/100$ you may, as before, gradually introduce more number of outliers into the dataset. You should see that for a certain number of outliers, the the Cauchy function will be sensitive to the "high" number of outliers. Can you comment on the experiments you have performed?

[12]:
```
points[2] = [25,0]
points[11] = [10,15]
points[21] = [-10,20]

cauchy_c = [1,1/100]
```

```
x12 = points[:,0]
x22 = points[:,1]

plt.plot(x12, x22, 'bo', markersize=4, c='orange')

reg_4 = np.arange(x11.min() - 1, x22.max() + 1, 0.1)

for i in range(len(cauchy_c)):
    coeffs_2, steps_2, f_2, iters_2 = grad_descent_cauchy(cauchy, x12, x22, 1,␣
 ↪1, 1000, 10**-6, c=cauchy_c[i])
    reg4 = np.arange(x12.min() - 1, x22.max() + 1, 0.1)
    plt.plot(reg_4, reg_4 * coeffs_2[0] + coeffs_2[1], color=colors[i],␣
 ↪label=f'c={cauchy_c[i]}')

plt.legend()
```

```
/var/folders/ng/1q_6q8_n1fx35q947ctj0w1m0000gn/T/ipykernel_81338/3133936685.py:9
: UserWarning: color is redundantly defined by the 'color' keyword argument and
the fmt string "bo" (-> color='b'). The keyword argument will take precedence.
  plt.plot(x12, x22, 'bo', markersize=4, c='orange')
```

[12]: <matplotlib.legend.Legend at 0x11621c4a0>



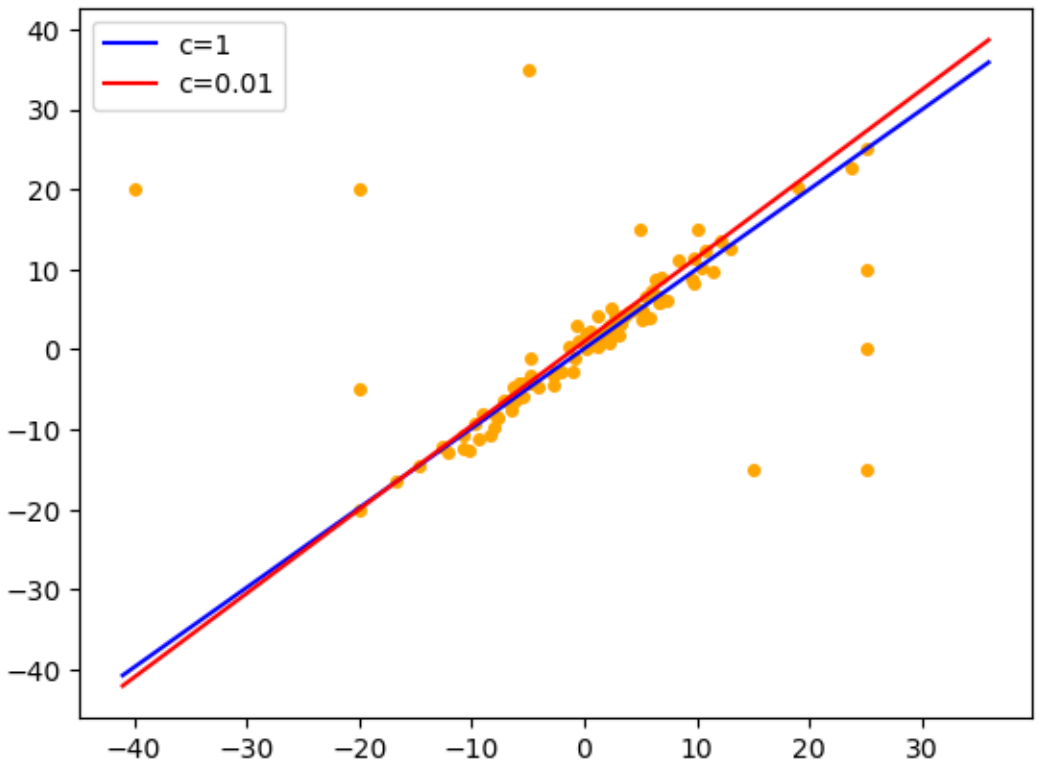
When adding 3 outliers, the Cauchy function still seems to be ok, albeit a little variance between c=1 and c=0.01

[13]:
```
points[20] = [-5,35]
points[21] = [25,10]
points[22] = [15,-15]
points[30] = [-20,-20]
points[31] = [25,-15]
```

[14]:
```
cauchy_c = [1,1/100]  # Values for c
x12 = points[:,0]
x22 = points[:,1]

plt.plot(x12, x22, 'bo', markersize=4, c='orange')

for i in range(len(cauchy_c)):
    coeffs_2, steps_2, f_2, iters_2 = grad_descent_cauchy(cauchy, x12, x22, 1,␣
 ↪1, 1000, 10**-6, c=cauchy_c[i])
    reg_4 = np.arange(x12.min() - 1, x22.max() + 1, 0.1)
    plt.plot(reg_4, reg_4 * coeffs_2[0] + coeffs_2[1], color=colors[i],␣
 ↪label=f'c={cauchy_c[i]}')

plt.legend()
```

```
/var/folders/ng/1q_6q8_n1fx35q947ctj0w1m0000gn/T/ipykernel_81338/820069969.py:5:
UserWarning: color is redundantly defined by the 'color' keyword argument and
the fmt string "bo" (-> color='b'). The keyword argument will take precedence.
  plt.plot(x12, x22, 'bo', markersize=4, c='orange')
```

[14]:
```
<matplotlib.legend.Legend at 0x116644a40>
```



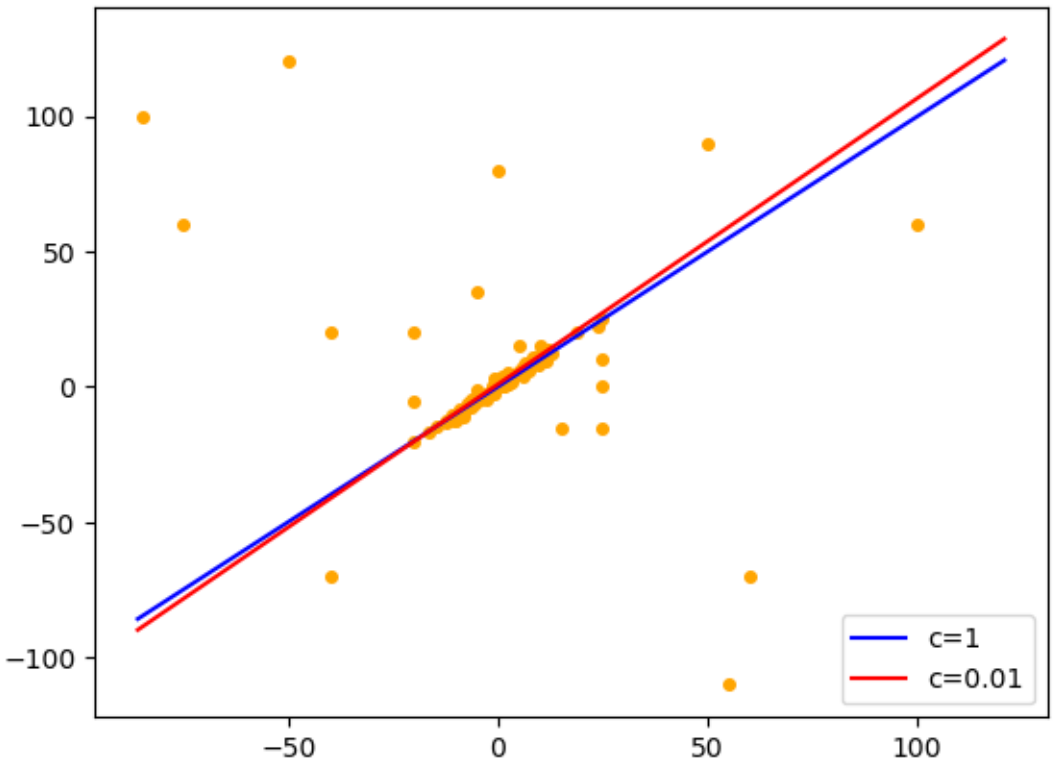
By adding 5 more outliers, both functions still seem to be ok.

[15]:
```
points[6] = [0, 80]
points[7] = [50, 90]
points[8] = [55, -110]
points[9] = [-75, 60]
points[12] = [60, -70]
points[13] = [-85, 100]
points[14] = [100, 60]
points[33] = [-50, 120]
points[34] = [-40, -70]

cauchy_c = [1,1/100]
x12 = points[:,0]
x22 = points[:,1]

plt.plot(x12, x22, 'bo',markersize=4, c='orange')

for i in range(len(cauchy_c)):
    coeffs_2, steps_2, f_2, iters_2 = grad_descent_cauchy(cauchy, x12, x22, 1,␣
 ↪1, 1000, 10**-6, c=cauchy_c[i])
    reg_4 = np.arange(x12.min() - 1, x22.max() + 1, 0.1)
```

```
    plt.plot(reg_4, reg_4 * coeffs_2[0] + coeffs_2[1], color=colors[i],␣
 ↪label=f'c={cauchy_c[i]}')

plt.legend()
```

```
/var/folders/ng/1q_6q8_n1fx35q947ctj0w1m0000gn/T/ipykernel_81338/1494042006.py:1
5: UserWarning: color is redundantly defined by the 'color' keyword argument and
the fmt string "bo" (-> color='b'). The keyword argument will take precedence.
  plt.plot(x12, x22, 'bo',markersize=4, c='orange')
```

[15]: <matplotlib.legend.Legend at 0x1166c44a0>



It can be observed that by adding even more outliers, and of bigger values, both functions still appear to be functioning relatively fine.