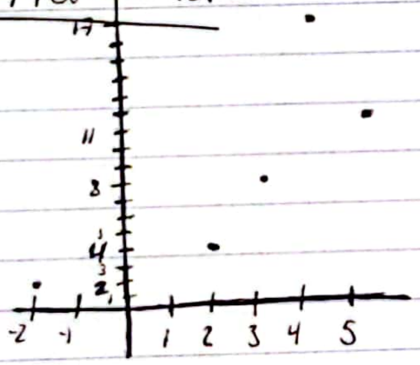
HW1 Problem 1a

Alonso Buitano



x	y	$h_\theta(x)$	error
-2	2	0.842	-1.158
2	4	7.718	3.718
3	8	9.432	1.432
5	11	12.868	1.868
4	17	11.150	-5.850

S.S.E.: 7.411

Normal equation (from class):
Using $h_\theta(x) = \theta_0 + \theta_1 x$

$$\theta_0 = \frac{\sum_{i=1}^{m} y^{(i)} - \theta_1 x^{(i)}}{m}$$

$$\theta_1 = \frac{\sum_{i=1}^{m} x^{(i)}(y^{(i)} - \theta_0)}{\sum_{i=1}^{m} (x^{(i)})^2}$$

$$\theta_0 = \frac{2 - \theta_1(-2) + 4 - 2\theta_1 + 8 - 3\theta_1 + 11 - 5\theta_1 + 17 - 4\theta_1}{5}$$

$$\theta_1 = \frac{-2(2-\theta_0) + 2(4-\theta_0) + 3(8-\theta_0) + 5(11-\theta_0) + 4(17-\theta_0)}{58}$$

$$\theta_0 = \frac{42 - 12\theta_1}{5} \qquad \theta_1 = \frac{151 - 12\theta_0}{58}$$

$$\theta_0 = \frac{42 - 12\left(\frac{151 - 12\theta_0}{58}\right)}{5} = \frac{42}{5} - \frac{12(151 - 12\theta_0)}{5(58)}$$

$$= \frac{42}{5} - \frac{12(151)}{5(58)} + \frac{144\theta_0}{5(58)}$$

$$\theta_0\left(1 - \frac{144}{5(58)}\right) = \frac{42}{5} - \frac{12(151)}{5(58)}$$

$$\theta_0 = \frac{\frac{42}{5} - \frac{12(151)}{5(58)}}{1 - \frac{144}{5(58)}} = \frac{8.4 - 6.248}{0.503} = 4.278$$

$\theta_1 = 1.718$

$$\boxed{h_\theta(x) = 4.278 + 1.718x}$$

Alonso Buitano

L~~SE derivation~~

~~$h_\theta(x) = \theta_0 + \theta_1 x$~~

MSE Loss function:

$$L = \frac{1}{2n}\sum_{i=1}^{n}\left(y^{(i)} - \left(b_0 + b_1 x^{(i)}\right)\right)^2$$

Derivation to obtain gradients:

$$\frac{\partial L}{\partial b_0} = \frac{1}{2n}\sum_{i=1}^{n} 2\left(y^{(i)} - \left(b_0 + b_1 x^{(i)}\right)\right)\cdot\left(\cancelto{0}{y^{(i)}} - \left(\cancelto{1}{b_0} + \cancelto{0}{b_1 x^{(i)}}\right)\right)$$

$$= \frac{1}{n}\sum_{i=1}^{n} -1\left(y^{(i)} - \left(b_0 + b_1 x^{(i)}\right)\right) = \boxed{\frac{1}{n}\sum_{i=1}^{n}\left(b_0 + b_1 x^{(i)} - y^{(i)}\right)}$$

$$\frac{\partial L}{\partial b_1} = \frac{1}{2n}\sum_{i=1}^{n} 2\left(y^{(i)} - \left(b_0 + b_1 x^{(i)}\right)\right)\cdot\left(\cancelto{0}{y^{(i)}} - \left(\cancelto{0}{b_0} + b_1 \cancelto{x^{(i)}}{x^{(i)}}\right)\right)$$

$$= \boxed{\frac{1}{n}\sum_{i=1}^{n} x^{(i)}\left(b_0 + b_1 x^{(i)} - y^{(i)}\right)}$$

In [1]:
```
import numpy as np
import math
```

In [2]:
```
x = np.array(input("Enter your x values separated by commas: ").split(','))
y = np.array(input("Enter your y values separated by commas (should be same

#Converting data arrays to numeric type.
x = x.astype(np.int32)
y = y.astype(np.int32)
```

```
Enter your x values separated by commas: -2,2,3,5,4
Enter your y values separated by commas (should be same size as x vector):
2,4,8,11,17
```

In [3]:
```
def normal_eq(x,y):
    # Function to find the values b0 and b1 from the normal equation seen in
    a = np.array([[x.size, np.sum(x)], [np.sum(x), np.sum(np.square(x))]])
    b = np.array([np.sum(y), np.dot(x,y)])
    [b0,b1] = np.linalg.solve(a,b)
    return b0, b1
```

In [4]:
```
[b0,b1]=normal_eq(x,y)
```

In [5]:
```
print("y-intercept is: ",b0)
print("slope is: ",b1)
```

```
y-intercept is:  4.273972602739727
slope is:  1.7191780821917806
```

In [1]:
```
import numpy as np
import math
```

In [2]:
```
f = np.genfromtxt("p1_data.csv", delimiter = ',', skip_header=1)
x = f[:,0]
y = f[:,1]
#x = np.array(input("Enter your x values separated by commas: ").split(','))
#y = np.array(input("Enter your y values separated by commas (should be same

#Converting data arrays to numeric type.
#x = x.astype(np.int32)
#y = y.astype(np.int32)
```

In [3]:
```
def normal_eq(x,y):
    # Function to find the values b0 and b1 from the normal equation seen in
    a = np.array([[x.size, np.sum(x)], [np.sum(x), np.sum(np.square(x))]])
    b = np.array([np.sum(y), np.dot(x,y)])
    [b0,b1] = np.linalg.solve(a,b)
    return b0, b1
```

In [4]:
```
[b0,b1]=normal_eq(x,y)
```

In [5]:
```
print("y-intercept: ",b0)
print("slope: ",b1)
```

```
y-intercept:  4.080657141894917
slope:  -0.44236913850430776
```

In [1]:
```python
def includes(item, val, start_ind=None):
    if type(item) == dict:
        key = list(item.keys())
        for i in range(len(key)):
            if item[key[i]] == val:
                return True
    else:
        if start_ind == None:
            if val in item:
                return True
        else:
            for i in range(start_ind, len(item)):
                if item[i] == val:
                    return True
    return False
```

In [2]:
```python
includes([2, 3, 4], 2, 0) # True
```

Out[2]: True

In [3]:
```python
includes([2, 3, 4], 2, 1) # False
```

Out[3]: False

In [4]:
```python
includes([2, 3, 4], 4, 1) # True
```

Out[4]: True

In [5]:
```python
includes({'a':1,'b':2}, 1) # True
```

Out[5]: True

In [6]:
```python
includes({'a':1,'b':2}, 'a') # False
```

Out[6]: False

In [7]:
```python
includes('abcd', 'b') # True
```

Out[7]: True

In [1]:
```python
def moving_average():
    numbers = []
    def average(x):
        numbers.append(x)
        return round(sum(numbers)/len(numbers),1)
    return average
```

In [2]:
```python
mAvg = moving_average()
print(mAvg(10)) #10.0
print(mAvg(11)) #10.5
print(mAvg(12)) #11.0
```

```
10.0
10.5
11.0
```

```python
In [1]: def same_frequency(num1, num2):
            item1, item2 = str(num1), str(num2)
            return sorted(item1) == sorted(item2)
```

```python
In [2]: same_frequency(551122,221515) # True
```

```
Out[2]: True
```

```python
In [3]: same_frequency(12345,31354) # False
```

```
Out[3]: False
```

```python
In [4]: same_frequency(321142,3212215) # False
```

```
Out[4]: False
```

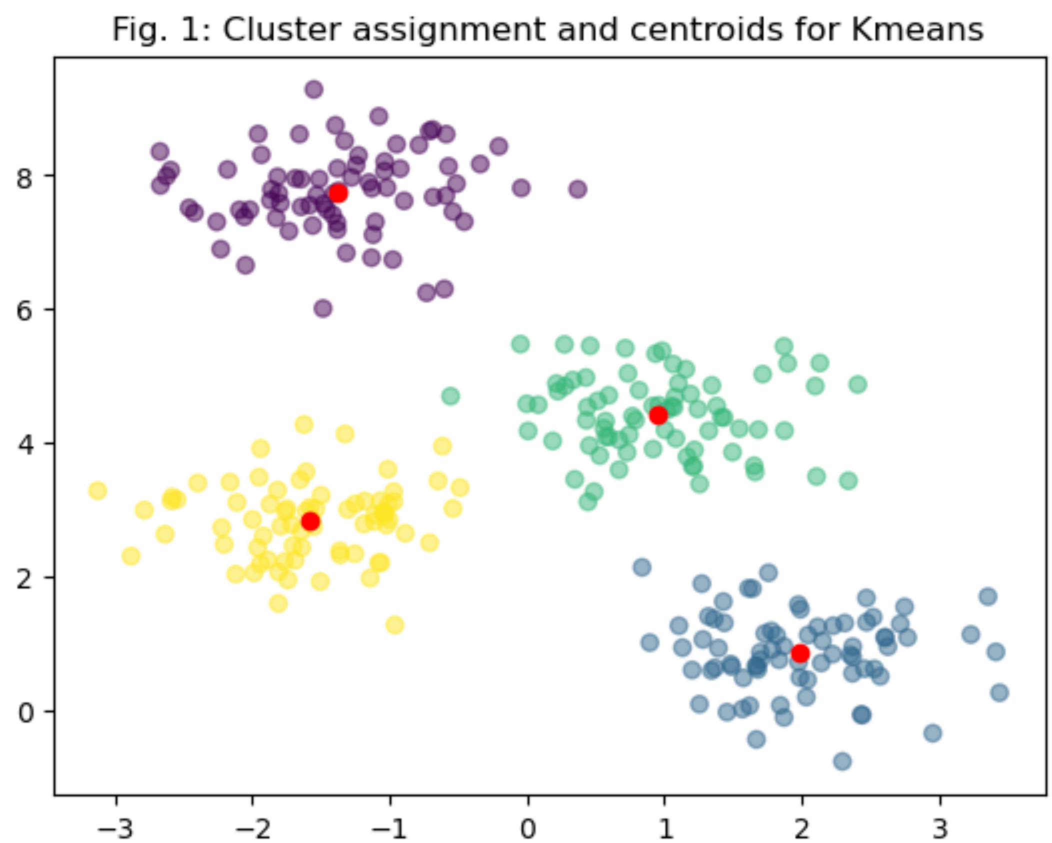
```python
In [5]: same_frequency(1212, 2211) # True
```

```
Out[5]: True
```

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

In [2]:
```python
file = np.load('kmeans.npz')
data = file['data']
pred = file['pred']
centers = file['centers']
```

In [3]:
```python
plt.scatter(data[:,0], data[:,1], c=pred, alpha=0.5)
plt.scatter(centers[:,0], centers[:,1], c='r')
plt.title("Fig. 1: Cluster assignment and centroids for Kmeans")
plt.show()
```



In [1]:
```python
import numpy as np
```

In [2]:
```python
def NUMPY_outer(X,Y):
    outer=np.zeros((len(X),len(Y)), dtype=int)
    for i in range(len(X)):
        for j in range(len(Y)):
            outer[i][j] = X[i]*Y[j]
    return outer
```

In [3]:
```python
np.random.seed(24787)
X = np.random.randint(-1000, 1000, size=3000)
Y = np.random.randint(-1000, 1000, size=3000)
```

In [4]:
```python
np.outer(X,Y)
```

Out[4]:
```
array([[ 288116,  433466,  322354, ...,  234498,  459306,  323646],
       [ 214972,  323422,  240518, ...,  174966,  342702,  241482],
       [-312200, -469700, -349300, ..., -254100, -497700, -350700],
       ...,
       [ 180184,  271084,  201596, ...,  146652,  287244,  202404],
       [ -66454,  -99979,  -74351, ...,  -54087, -105939,  -74649],
       [ 203376,  305976,  227544, ...,  165528,  324216,  228456]])
```

In [5]:
```python
NUMPY_outer(X,Y)
```

Out[5]:
```
array([[ 288116,  433466,  322354, ...,  234498,  459306,  323646],
       [ 214972,  323422,  240518, ...,  174966,  342702,  241482],
       [-312200, -469700, -349300, ..., -254100, -497700, -350700],
       ...,
       [ 180184,  271084,  201596, ...,  146652,  287244,  202404],
       [ -66454,  -99979,  -74351, ...,  -54087, -105939,  -74649],
       [ 203376,  305976,  227544, ...,  165528,  324216,  228456]])
```

In [6]:
```python
NUMPY_outer(X,Y)==np.outer(X,Y)
```

Out[6]:
```
array([[ True,  True,  True, ...,  True,  True,  True],
       [ True,  True,  True, ...,  True,  True,  True],
       [ True,  True,  True, ...,  True,  True,  True],
       ...,
       [ True,  True,  True, ...,  True,  True,  True],
       [ True,  True,  True, ...,  True,  True,  True],
       [ True,  True,  True, ...,  True,  True,  True]])
```

The built-in implementation by numpy is much faster because of how numpy manages its arrays, with homogeneous data types stored together in memory. Also due to how it integrates C/C++ code, which are faster than normal python. It also breaks down tasks into fragments and executes in parallel, so each row would be calculated in parallel in this excercise.

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

In [2]:
```python
def MSE(x, y, b0, b1):
    # Function to obtain the loss and the gradients using Mean Squared Error
    pred = np.multiply(b1, x) + b0
    loss = 0.5 * np.mean(np.power((y - pred), 2))
    gradient0 = np.mean(pred - y)
    gradient1 = np.mean(x * (pred - y))
    return loss, gradient0, gradient1
```

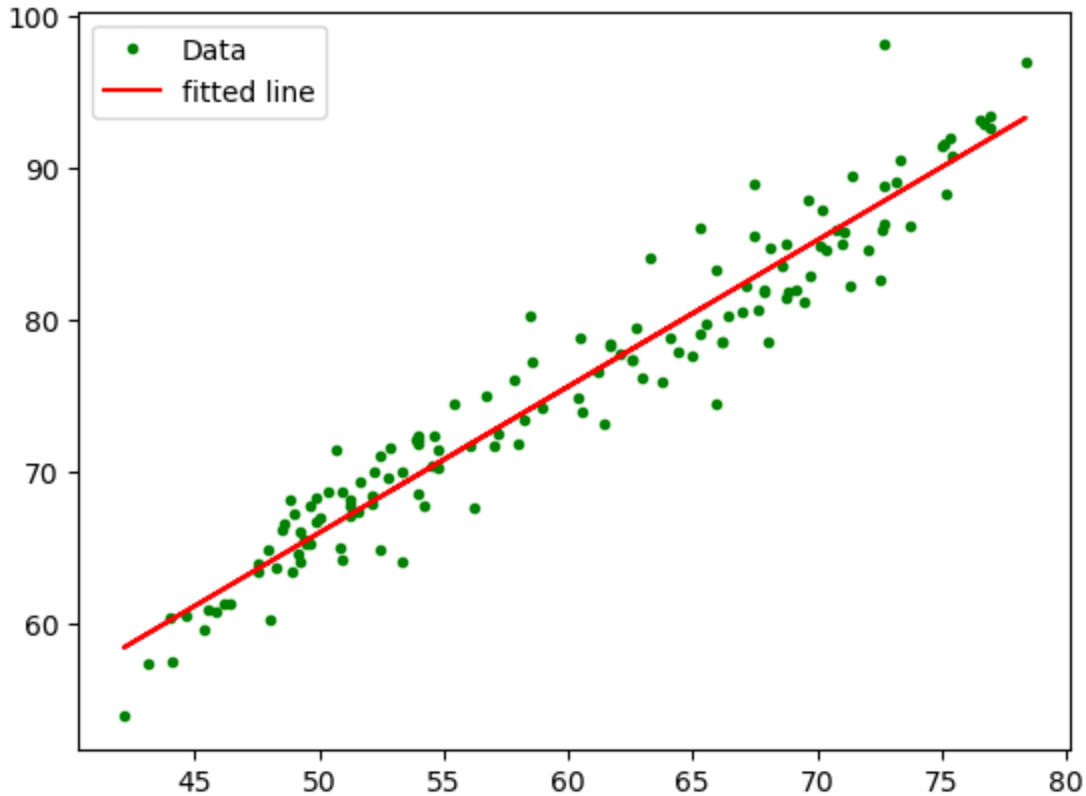
In [3]:
```python
def gradient_descent(data, init_b0, init_b1, lr, num_epochs):
    x = data[:,0]
    y = data[:,1]
    b0 = init_b0 # Starting with initial guess for b0
    b1 = init_b1 # Starting with initial guess for b1
    b_list = []
    loss_list = []

    for i in range(num_epochs):
        b_list.append((b0,b1)) # Adds current (b0, b1) to the list
        # Compute loss and gradients
        loss, grad0, grad1 = MSE(x, y, b0, b1)
        loss_list.append(loss) # Adds current loss to the list
        # Update model parameters
        b0 -= (lr * grad0)
        b1 -= (lr * grad1)

    #Add final (b0, b1) and loss values to their list.
    b_list.append((b0,b1))
    loss, grad0, grad1 = MSE(x, y, b0, b1)
    loss_list.append(loss)

    return b_list, loss_list
```

In [4]:
```python
data = np.load('p3_data.npy')
```

In [5]:
```python
B, loss = gradient_descent(data, 18, 5, 0.0001, 1000)
```

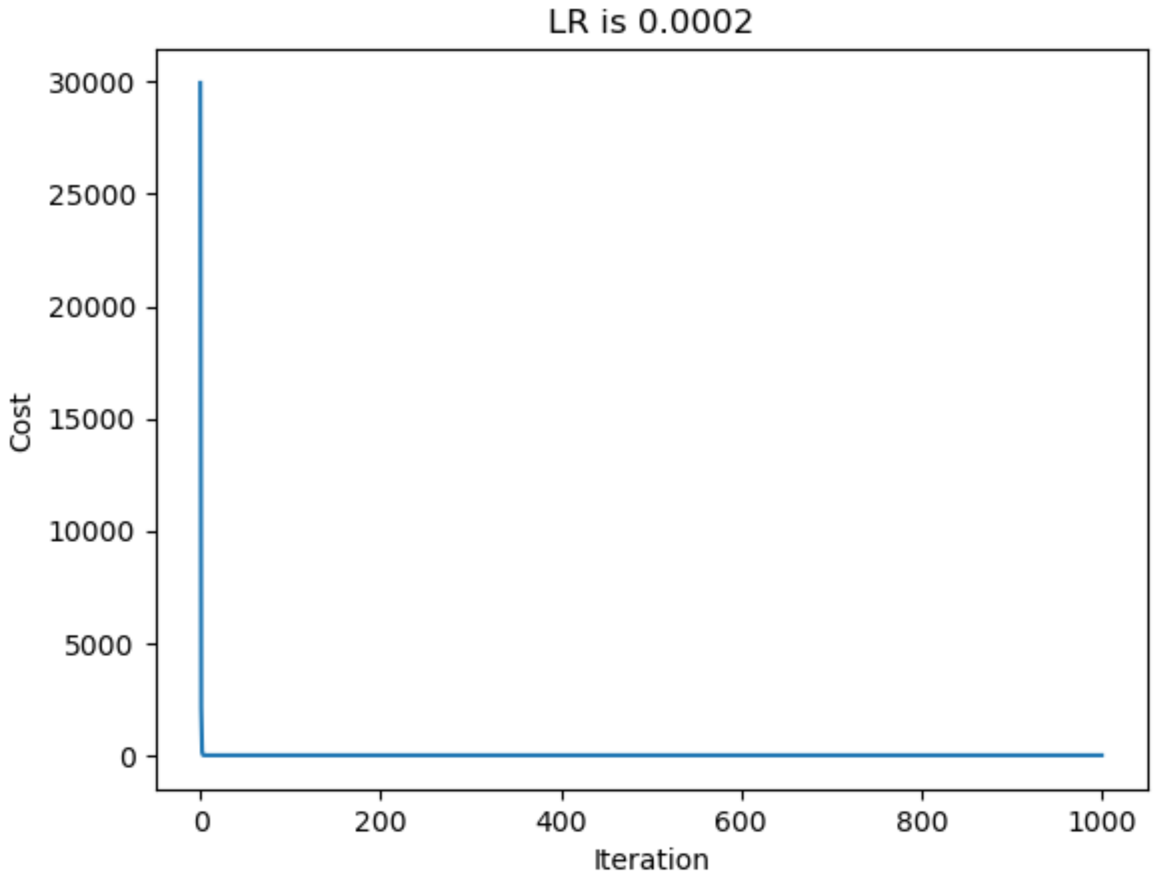
In [6]:
```python
print("Final b0 and b1 values are: ", B[-1])
print("Final loss value is: ", loss[-1])

plt.plot(data[:,0], data[:,1], '.', c='g', label = "Data")
plt.plot(data[:,0], B[-1][0] + data[:,0]*B[-1][1], c='r', label = "fitted li
plt.legend()
plt.show()
```

```
Final b0 and b1 values are:  (17.93432667747319, 0.9618342679700583)
Final loss value is:  3.097854325598297
```

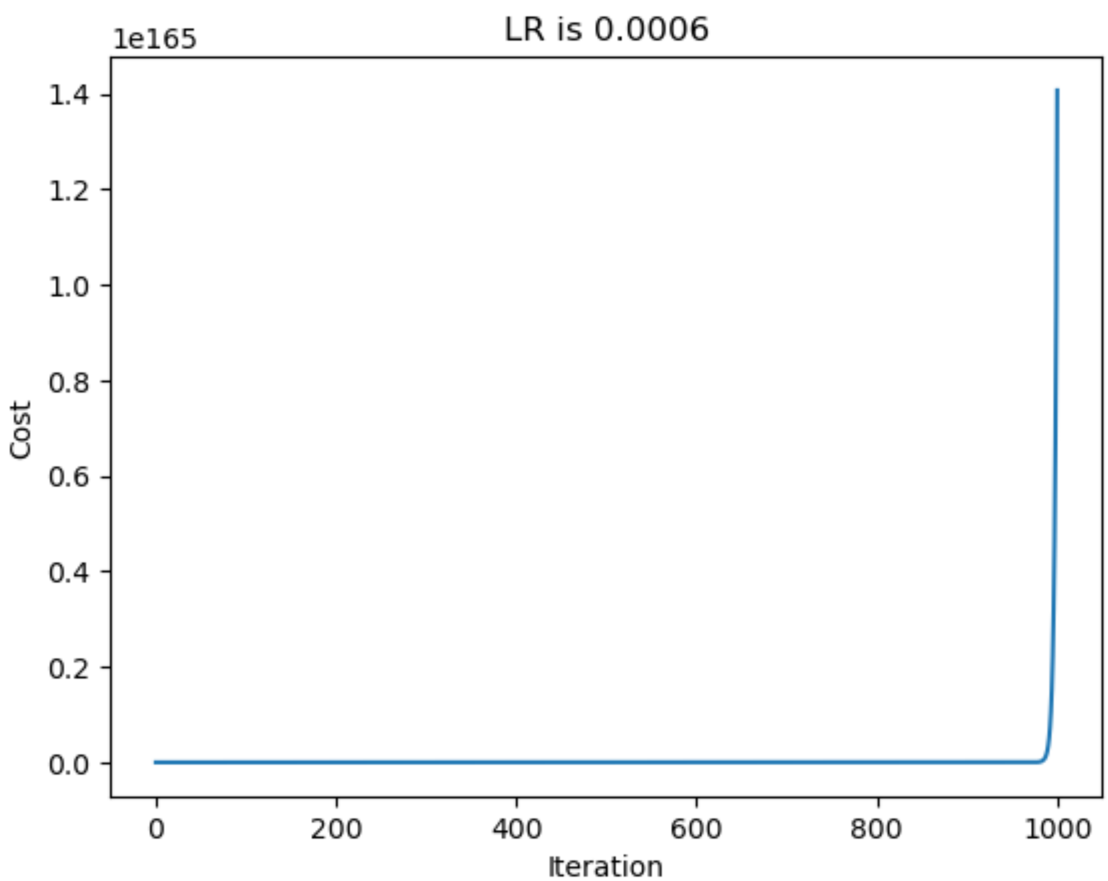


In [7]:
```
for lr in [0.0002, 0.0006, 0.01, 10]:
    B, loss = gradient_descent(data, 18, 5, lr, 1000)
    print("Final loss and B values are: ", loss[-1], ", ", B[-1])

    plt.plot(loss)
    plt.xlabel("Iteration")
    plt.ylabel("Cost")
    plt.title(f"LR is {lr}")
    plt.show()
```

```
Final loss and B values are:  3.0978541866329126 ,  (17.934444545603046, 0.
96183234761207)
```

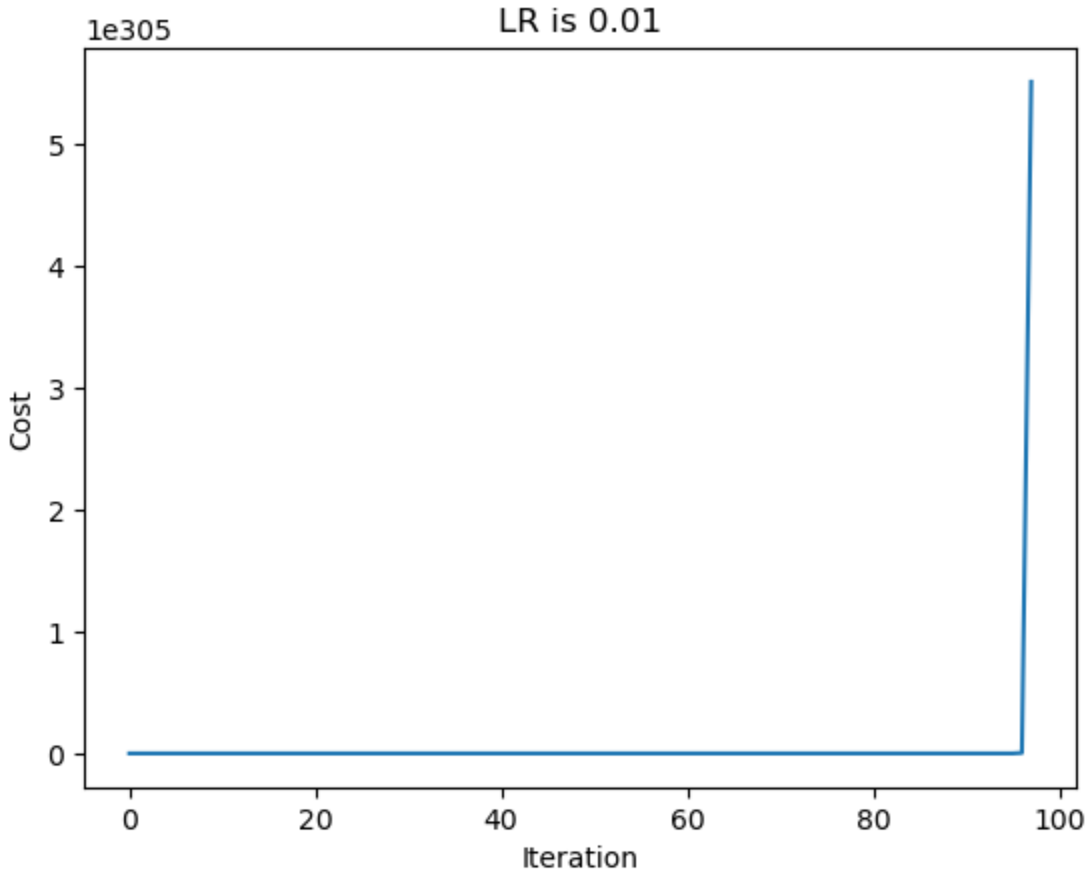



```
Final loss and B values are:  1.4073988952349527e+165 ,  (1.426267297867182
e+79, 8.75417292526297e+80)
```



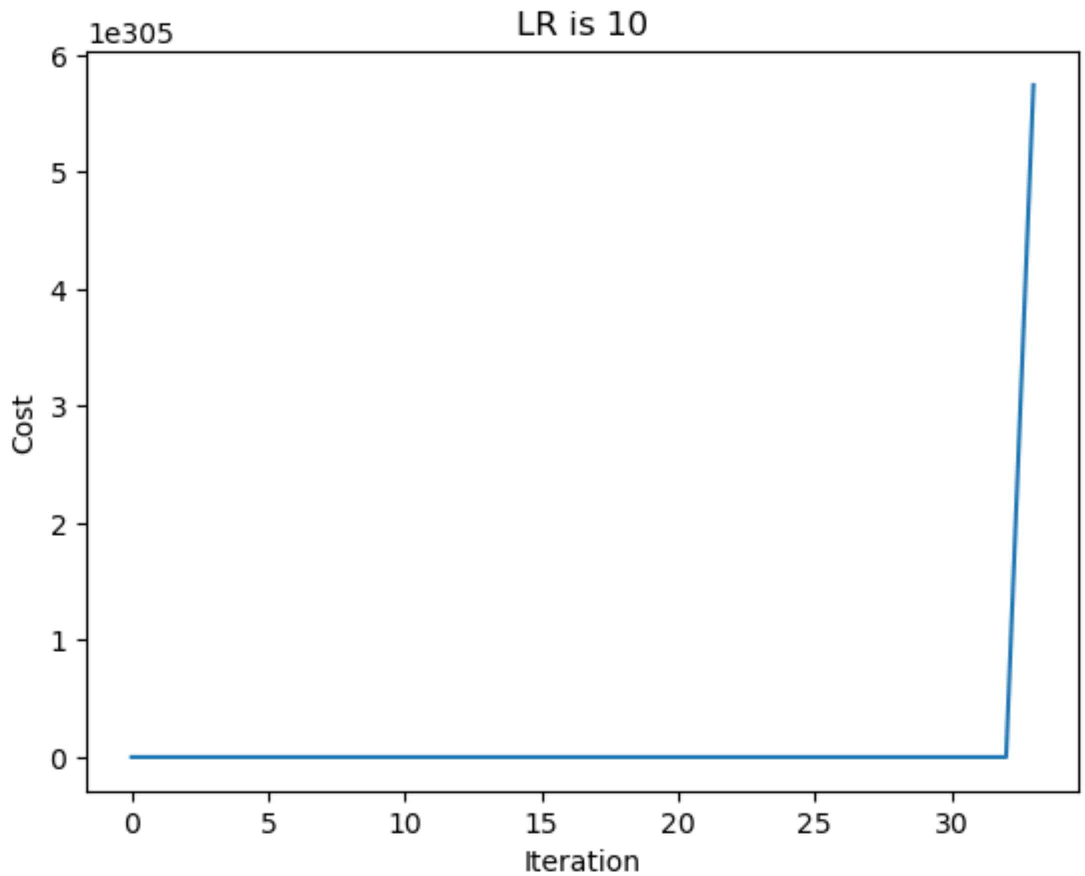
```
Final loss and B values are:  nan ,  (nan, nan)
```

```
/var/folders/9_/zlyt_0jn1cj4plrbb7kf80d80000gn/T/ipykernel_36279/400944365
6.py:4: RuntimeWarning: overflow encountered in power
  loss = 0.5 * np.mean(np.power((y - pred), 2))
/Users/abuitano/miniforge3/lib/python3.10/site-packages/numpy/core/_method
s.py:179: RuntimeWarning: overflow encountered in reduce
  ret = umr_sum(arr, axis, dtype, out, keepdims, where=where)
/var/folders/9_/zlyt_0jn1cj4plrbb7kf80d80000gn/T/ipykernel_36279/272358059
7.py:16: RuntimeWarning: invalid value encountered in double_scalars
  b1 -= (lr * grad1)
```




```
Final loss and B values are:  nan ,  (nan, nan)
```

```
/var/folders/9_/zlyt_0jn1cj4plrbb7kf80d80000gn/T/ipykernel_36279/400944365
6.py:3: RuntimeWarning: overflow encountered in multiply
  pred = np.multiply(b1, x) + b0
/var/folders/9_/zlyt_0jn1cj4plrbb7kf80d80000gn/T/ipykernel_36279/400944365
6.py:6: RuntimeWarning: overflow encountered in multiply
  gradient1 = np.mean(x * (pred - y))
/var/folders/9_/zlyt_0jn1cj4plrbb7kf80d80000gn/T/ipykernel_36279/272358059
7.py:15: RuntimeWarning: invalid value encountered in double_scalars
  b0 -= (lr * grad0)
```



I think the best learning rate out of the ones tested is 0.0002, as this will still converge and in less iterations than 0.0001. The other larger learning rates start diverging, so they are not good for this model.

Normal equation derivation is in handwritten notes

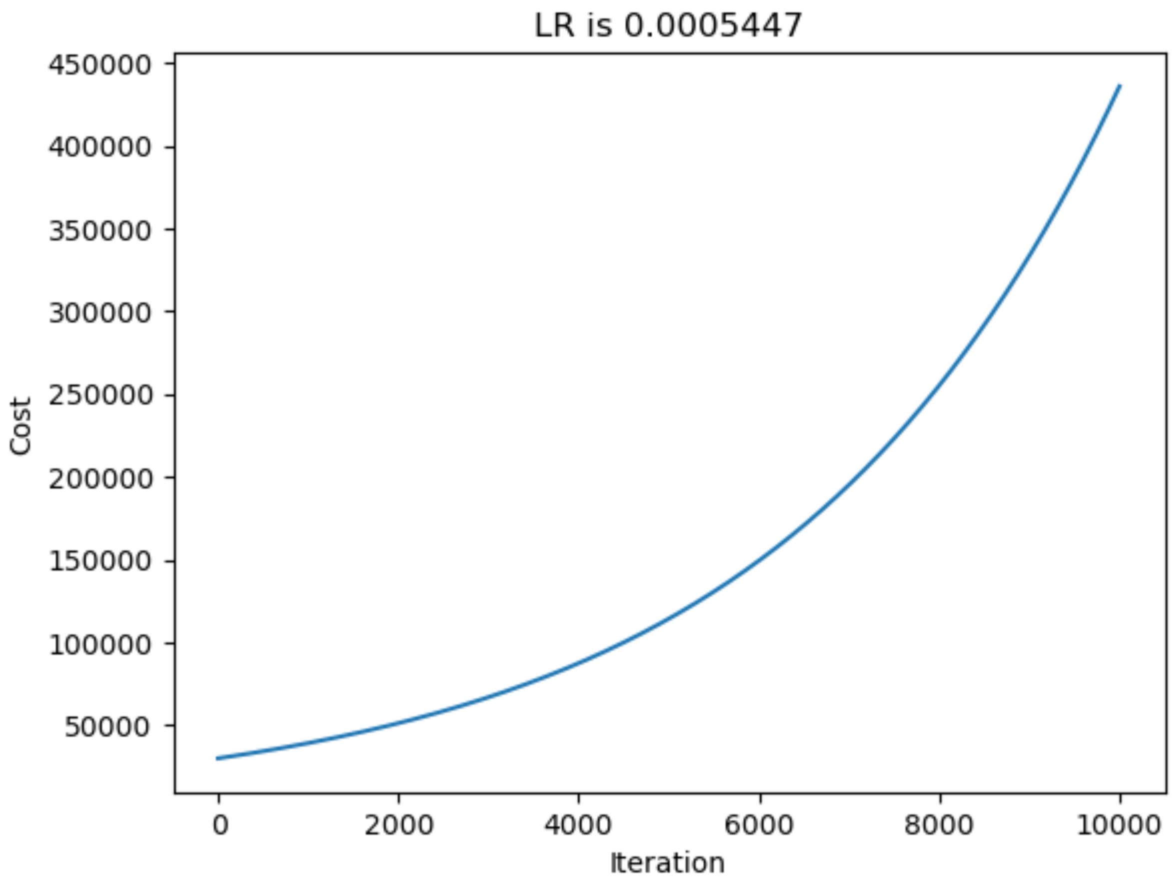
Gradient descent is an iterative method to optimize a function, or to find a local minimum by moving in the direction of steepest descent, using the gradient. We use it to find the parameters (b0, b1,..., bn) of our hypothesis function in order to minimize the loss we get vs the ground truths.

Three types of gradient descent:

- Batch gradient descent: in this type of GD, we process all of the data with the current parameters before updating.
- Stochastic gradient descent: here, we update the parameters for every sample, at every step (slow)
- Mini-batch gradient descent: A trade-off of the last two types of GD, where we update after a specified subset of the data (a batch) is processed, at each step.

In [8]:
```
lr = 0.0005447
B, loss = gradient_descent(data, 18, 5, lr, 10000)
```

In [9]:
```
plt.plot(loss)
plt.xlabel("Iteration")
plt.ylabel("Cost")
plt.title(f"LR is {lr}")
plt.show()
```

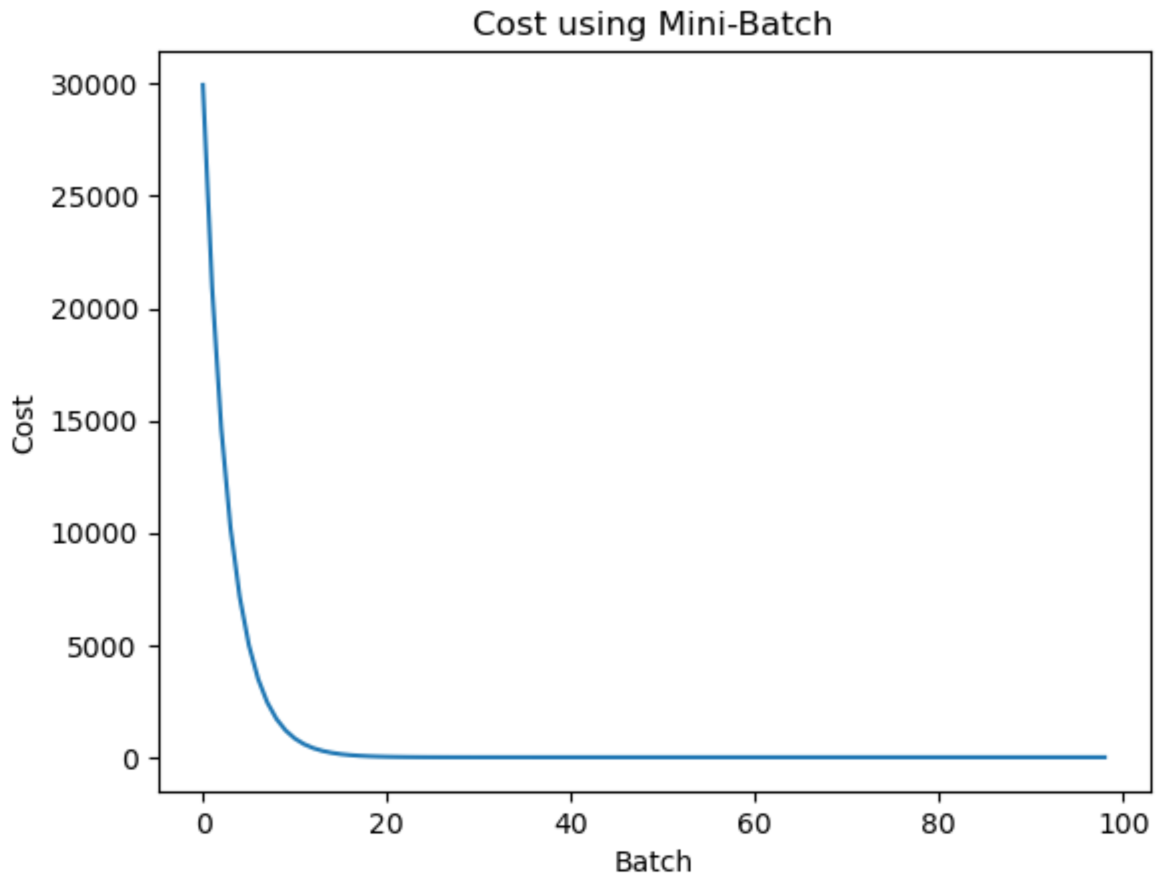


The upper bound of learning rate for this model seems to be just under 0.000545. Divergence occurs starting at a learning rate of 0.0005447.

In [10]:
```
def mini_GD(data, init_b0, init_b1, lr, num_iterations, batch_size):
    # Function for mini-batch gradient descent
    x = data[:,0]
    y = data[:,1]
    b0 = init_b0
    b1 = init_b1
    b_list = []
    loss_list = []
    num_batches = len(x) // batch_size
    num_epochs = num_iterations // num_batches

    for epoch in range(num_epochs):
        for b in range(num_batches):
            b_list.append((b0,b1)) # Adds current (b0, b1) to the list
            # Compute loss and gradients

            pred = np.multiply(b1, x[b*batch_size: (b+1)*batch_size]) + b0
            loss = 0.5 * np.mean(np.power((y[b*batch_size: (b+1)*batch_size]
            gradient0 = np.mean(pred - y[b*batch_size: (b+1)*batch_size])
            gradient1 = np.mean(x[b*batch_size: (b+1)*batch_size] * (pred -

            loss, grad0, grad1 = MSE(x, y, b0, b1)
            loss_list.append(loss) # Adds current loss to the list
            # Update model parameters
            b0 -= (lr * grad0)
            b1 -= (lr * grad1)

    #Add final (b0, b1) and loss values to their list.
    b_list.append((b0,b1))
    loss, grad0, grad1 = MSE(x, y, b0, b1)
    loss_list.append(loss)

    return b_list, loss_list
```

In [11]:
```
B, loss = mini_GD(data, 18, 5, 0.0005, 100, 20)
```

In [12]:
```
plt.plot(loss)
plt.xlabel("Batch")
plt.ylabel("Cost")
plt.title("Cost using Mini-Batch")
plt.show()
```



Using mini-batch GD, the cost decreases much faster. It converges with less iterations than the batch GD done previously.