Assignment 02

First, set the coefficient matrix A and the variable vector x. You may use an off-the-shelf random number generator function to create A and x matrices.

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

In [2]:
```
A = np.random.randint(10, size=(3, 3))
A
```

Out[2]:
```
array([[3, 1, 3],
       [9, 0, 7],
       [7, 4, 5]])
```

In [3]:
```
x = np.random.randint(10, size=3)
x
```

Out[3]:
```
array([9, 5, 1])
```

Using these two, obtain the outcome vector b such that Ax = b.

In [4]:
```
b = np.dot(A, x)
b
```

Out[4]:
```
array([35, 88, 88])
```

a. Obtain the closed-form solution of Ax =b. Print the estimated solution, x. Test with both regular and pseudo-inverse of A.

Closed-form solution

In [5]:
```
x_close_sol = np.dot(np.linalg.inv(A), b)

x_close_sol
```

Out[5]:
```
array([9., 5., 1.])
```

Pseudo-inverse solution

In [6]:
```
A_T = np.transpose(A)

A_T_times_A = np.dot(A_T, A)
A_T_times_b = np.dot(A_T, b)

x_pseudo_sol = np.dot(np.linalg.inv(A_T_times_A), A_T_times_b)
x_pseudo_sol
```

Out[6]:
```
array([9., 5., 1.])
```

b. Obtain the sum of squared error (goodness of the estimation) of this closed-form solution as follows. $E = \sum_n (x_i - \hat{x}_i)^2$, where n is the total number of elements in the solution vector, x

In [7]:
```
closed_sol_sq_err = sum(x - x_close_sol) ** 2
closed_sol_sq_err
```

Out[7]:
```
2.8398992587956425e-29
```

In [8]:
```
pseudo_sol_sq_err = sum(x - x_pseudo_sol) ** 2
pseudo_sol_sq_err
```

Out[8]:
```
4.245964936261574e-26
```

c. Implement the gradient descent optimization algorithm to solve for x such that it minimizes $\| Ax - b \|_2$, where $\|.\|_2$ is the two norm. You need to update the solution for x iteratively taking the slope of the error into consideration.

[482]:

```python
n = 3

def gradient_descent(A,b,x0):
    step_size = 0.00001

    new_x = np.array([1, 1, 1])

    iteration = 0
    max_iters = 1000
    alpha = 0.002

    e_str = []
    it_str = [0]

    while iteration <= max_iters:


        old_x = new_x
        e = (np.dot(A, old_x) - np.transpose(b))
        new_x = old_x - alpha * 2 * np.dot(e, A)

        e_str.append(e)

        step = new_x - old_x


        if (abs(step) <= step_size).all():
            break

        iteration += 1

        it_str.append(iteration)

    print(iteration)
    return new_x, e_str, it_str
```

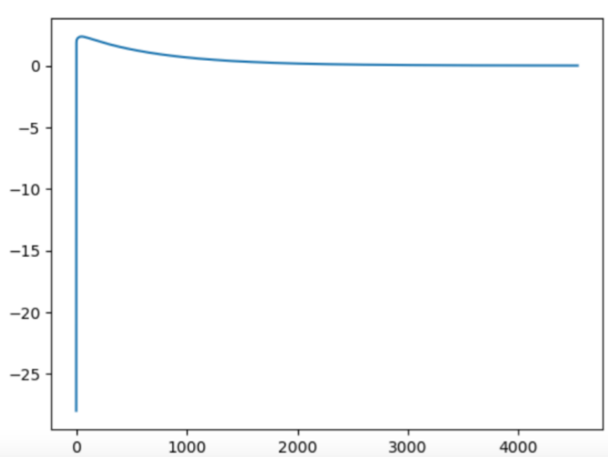
d. Plot a figure showing the sum of squared error (this is an unsupervised error, (e = Ax - b) versus the iteration number. Based on this figure state the number of iterations required to reach the minimum error point. Print the estimated solution, x.

In [14]:

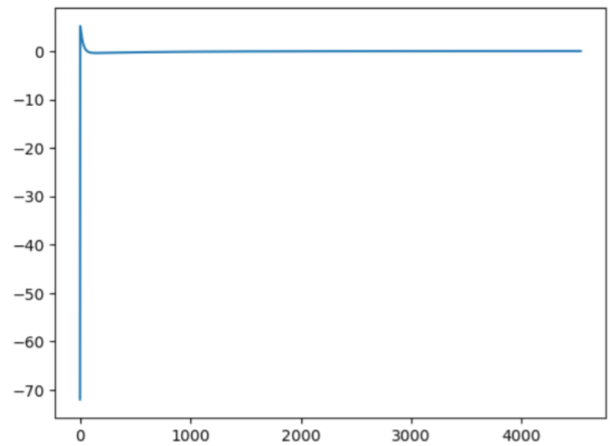
```python
import matplotlib.pyplot as plt
e_str = np.array(e_str)

plt.plot(itr, e_str[:,0])
```

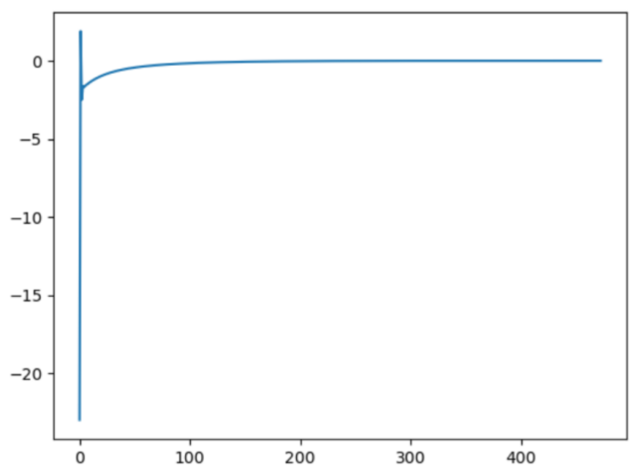
Out[14]: [<matplotlib.lines.Line2D at 0x7f8fd3727d30>]



```
In [15]: plt.plot(itr, e_str[:,1])
Out[15]: [<matplotlib.lines.Line2D at 0x7f8fd528aaf0>]
```




```
In [487]: plt.plot(itr, e_str[:,2])
Out[487]: [<matplotlib.lines.Line2D at 0x7fa635374f40>]
```




```
In [16]: x_final
Out[16]: array([8.99422051, 5.0006911 , 1.00728922])
```

(actual x: [9, 5, 1])

e. Show the sum squared error (goodness of the estimation as shown in part b), E for the final estimated x using the gradient descent algorithm.

In [17]:
```python
grad_desc_err = sum(e_str[-1])

grad_desc_err
```

Out[17]: 0.002987591897813502

f. Compare the E values obtained using 1) the closed-form solution and 2) gradient descent algorithm.

In [18]:
```python

print("Closed-form error: ", closed_sol_sq_err)
print("Gradient-descent error: ", grad_desc_err)
```

```
Closed-form error:  2.8398992587956425e-29
Gradient-descent error:  0.002987591897813502
```