# Assingment 04

# EE-527 Machine Learning Laboratory

## Group Members


```
RAJENDRA KUJUR (214161008)
ROHIT RAJ SINGH CHAUHAN (214161009)
```

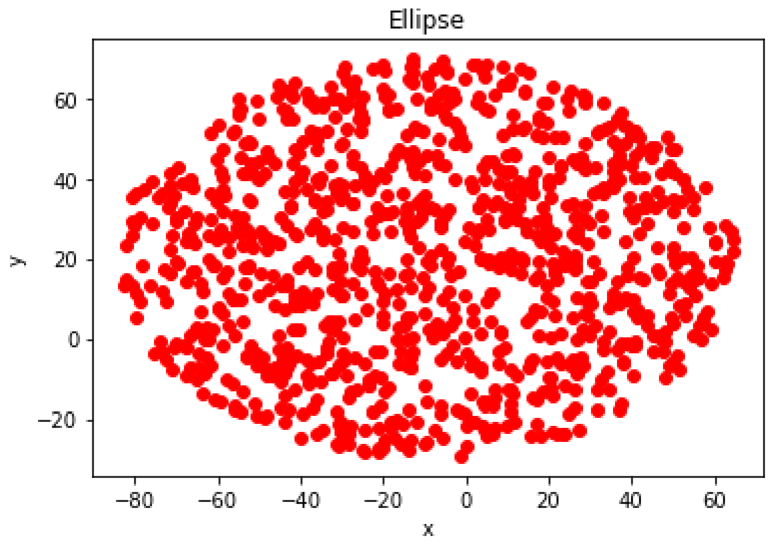

In [1]:

```
# Importing all necessary libraries
import numpy as np
import matplotlib.pyplot as plt
import random
import math
```

### Problem 1: Generation of Points within an Ellipse

In [2]:

```python
import random
import matplotlib.pyplot as plt
import numpy as np

# Ellipse Class
class ellipse:
# Initializes the ellipse class
  def __init__(self, origin_x, origin_y, major_axis, minor_axis):
      self.origin_x = origin_x
      self.origin_y = origin_y
      self.major_axis = major_axis
      self.minor_axis = minor_axis

# Checks whether the point lies inside the ellipse or not
  def isInsideEllipse(self, x_value, y_value):
      if ((x_value - self.origin_x)**2)/((self.major_axis/2)**2) + ((y_value - self
          return True
      else:
          return False


# Execution begins here
# define an ellipse with given constraints
e = ellipse(origin_x= -10, origin_y = 20, major_axis = 150, minor_axis = 100)

points= []
# Loop continues till we get 1000 valid points inside the ellipse
while len(points) < 1000:
    lower_limit_x = e.origin_x - (e.major_axis/2)
    upper_limit_x = e.origin_x + (e.major_axis/2)
    lower_limit_y = e.origin_y - (e.minor_axis/2)
    upper_limit_y = e.origin_y + (e.minor_axis/2)

    new_point_x = random.uniform(lower_limit_x, upper_limit_x)
    new_point_y = random.uniform(lower_limit_y, upper_limit_y)

    # if generated point lies inside ellipse then append point
    if e.isInsideEllipse(new_point_x, new_point_y) is True:
        points.append([new_point_x, new_point_y])

x = [points[i][0] for i in range(len(points))]
y = [points[i][1] for i in range(len(points))]

plt.scatter(x,y, color ='red')
plt.xlabel('x')
plt.ylabel('y')
plt.title('Ellipse')
plt.show()
```




## Problem 2: Generation of Points within 10 dimensional hypersphere

**Execution takes more than 30 seconds since it is in 10 dimensions (have patience)**

In [3]:

```
# Returns true if the generated point lies inside the hypersphere
def isInsideHyperSphere(radius, center, new_point):

    # calculate the respective dimenstion difference and square and store into an a
    # let's say we are storing into differ (for 3 dimension) = [(x1-x2)^2, (y1-y2)^
    # hence for 10 dimesnsion it can be written in shorthand
    differ = (center - new_point)**2

    # if the following equation is less than zero means point is inside hypersphere
    # if greater outside the hypersphere
    if sum(differ) - (radius**2) < 0:
        return True
    else:
        return False

# Generates the points inside hypersphere
def generateInsideHypersphere(radius, center, total_points):

    # To store the generated valid points
    S = []
    valid_points = 0

    # Iterate until we get required number of points
    while valid_points < total_points:
        new_point = np.array([center[i] + np.random.uniform(-1*radius, radius) for

        if isInsideHyperSphere(radius, center, new_point) is True:
            S.append(new_point)
            valid_points += 1

    # points in 3 dimension look like [x1,y1,z1], [x2,y2,z2]
    # can be generalized in 10 dimension
    S = np.array(S)
    return S


# Execution begins here
# Given fixed values
radius = 100
# the number of elements in the center is deciding factor to know how many dimensio
center = np.array([-1, -2, -1, 0, 0, 0, 3, 4, 9, 0])
total_points = 1000

# Points will be finally store into points variable
points = generateInsideHypersphere(radius, center, total_points)

# Points can be fetched as follows, along each dimension
# x = np.array([points[i][0] for i in range(len(points))])
# y = np.array([points[i][1] for i in range(len(points))])
```

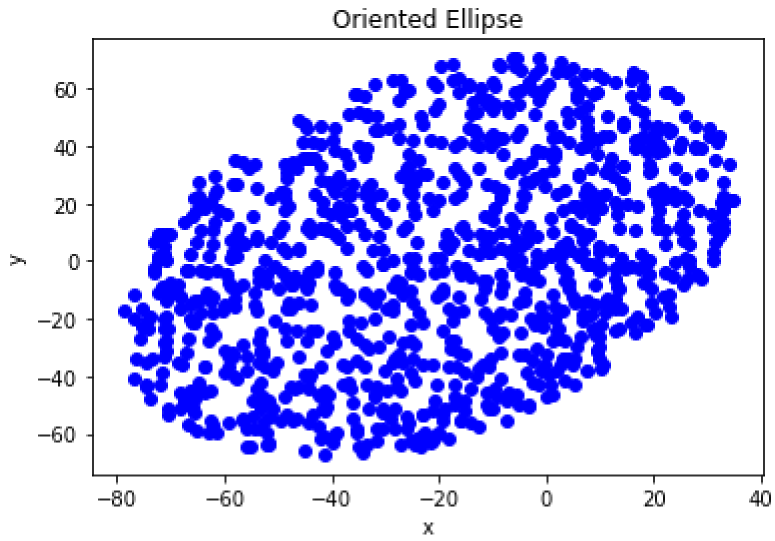
### Problem 3: Generation of points within oriented ellips

Randomly generate 2D ellipse of axes major axis makes an angle of 2D points , inside an oriented $eo$ and centered at $c$ with the horizontal axis

In [4]:

```
# Ellipse Class
class ellipse:
# Initializes the ellipse class
  def __init__(self, origin_x, origin_y, major_axis, minor_axis):
      self.origin_x = origin_x
      self.origin_y = origin_y
      self.major_axis = major_axis
      self.minor_axis = minor_axis

# Checks whether the point lies inside the ellipse or not
  def isInsideEllipse(self, x_value, y_value):
      if ((x_value - self.origin_x)**2)/((self.major_axis/2)**2) + ((y_value - self
          return True
      else:
          return False


# Execution begins here
# define an ellipse with given constraints
origin_x, origin_y = -10, 20
major_axis, minor_axis = 150, 100
e = ellipse(origin_x, origin_y, major_axis, minor_axis)

points= []

lower_limit_x = e.origin_x - (e.major_axis/2)
upper_limit_x = e.origin_x + (e.major_axis/2)
lower_limit_y = e.origin_y - (e.minor_axis/2)
upper_limit_y = e.origin_y + (e.minor_axis/2)
# Loop continues till we get 1000 valid points inside the ellipse
while len(points) < 1000:
    lower_limit_x = e.origin_x - (e.major_axis/2)
    upper_limit_x = e.origin_x + (e.major_axis/2)

    lower_limit_y = e.origin_y - (e.minor_axis/2)
    upper_limit_y = e.origin_y + (e.minor_axis/2)

    new_point_x = round(random.uniform(lower_limit_x, upper_limit_x),5)
    new_point_y = round(random.uniform(lower_limit_y, upper_limit_y),5)

    # if generated point lies inside ellipse then append point
    if e.isInsideEllipse(new_point_x, new_point_y) is True:
        points.append([new_point_x, new_point_y])

x = np.array([points[i][0] for i in range(len(points))])
y = np.array([points[i][1] for i in range(len(points))])
point_matrix = []
point_matrix.append(x)
point_matrix.append(y)

theta = math.pi/3
rotation_matrix = np.array([[np.cos(theta), -1*np.sin(theta)], [np.sin(theta), np.c
point_matrix = np.array(point_matrix)

result = rotation_matrix.dot(point_matrix)

x_new = []
y_new = []
```

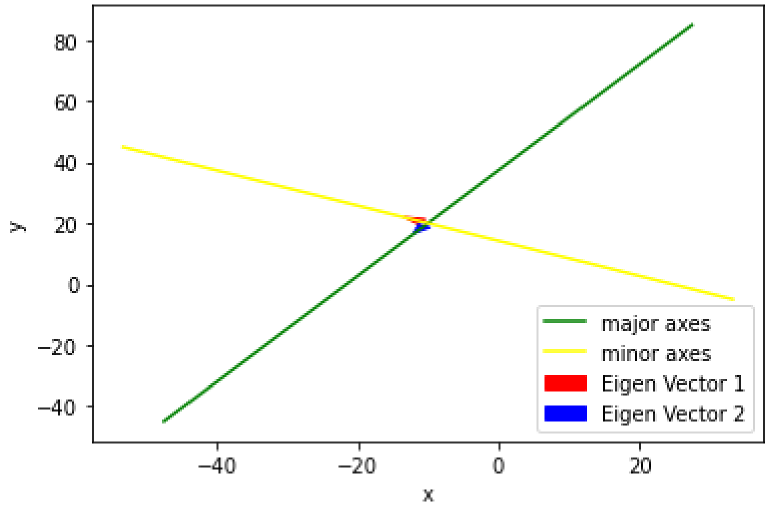
```python
x_new = np.array([result[0][i] for i in range(len(result[0]))])
y_new = np.array([result[1][i] for i in range(len(result[0]))])

plt.scatter(x_new, y_new, color ='blue')
plt.xlabel('x')
plt.ylabel('y')
plt.title('Oriented Ellipse')
plt.show()
```




## Problem 4: Covariance Matrix Computation

In [5]:

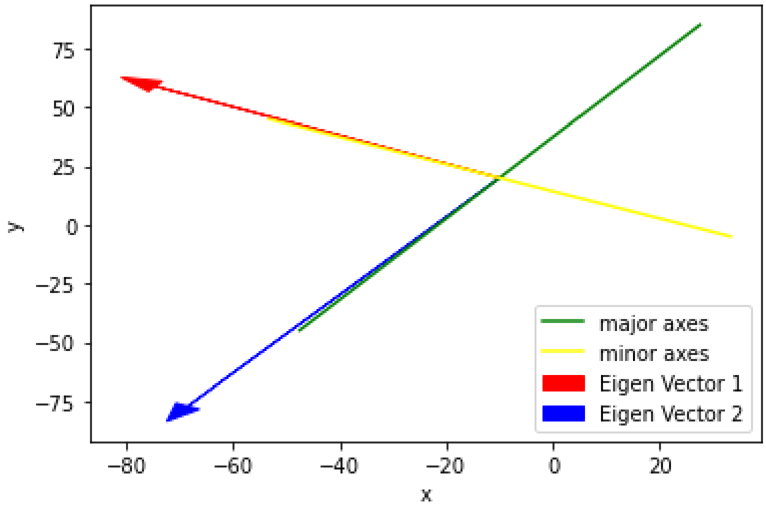
```
cov_mat = np.cov(x_new, y_new)

# previous major axes
major_axes_x = np.array([origin_x - major_axis/2, origin_x + major_axis/2])
major_axes_y = np.array([origin_y, origin_y])

# previous minor axes
minor_axes_x = np.array([origin_x, origin_x])
minor_axes_y = np.array([origin_y - minor_axis/2, origin_y + minor_axis/2])

major_axes_x_new, major_axes_y_new = [], []
minor_axes_x_new, minor_axes_y_new = [], []

# update major and minor axes due to rotation of an angle = pi/3
for i in range(2):
    major_axes_x_new.append((major_axes_x[i]-origin_x)*np.cos(math.pi/3) - (major_a
    major_axes_y_new.append((major_axes_x[i]-origin_x)*np.sin(math.pi/3) + (major_a
    minor_axes_x_new.append((minor_axes_x[i]-origin_x)*np.cos(math.pi/3) - (minor_a
    minor_axes_y_new.append((minor_axes_x[i]-origin_x)*np.sin(math.pi/3) + (minor_a


eigen_values, eigen_vectors = np.linalg.eig(cov_mat)
vec_1 = eigen_vectors[:, 0]
vec_2 = eigen_vectors[:, 1]

plt.arrow(origin_x, origin_y, vec_1[0], vec_1[1], color='red', label = 'Eigen Vecto
plt.arrow(origin_x, origin_y, vec_2[0], vec_2[1], color='blue', label = 'Eigen Vect
plt.plot(major_axes_x_new, major_axes_y_new, color = 'green',label = 'major axes')
plt.plot(minor_axes_x_new, minor_axes_y_new, color = 'yellow', label = 'minor axes'
plt.xlabel('x')
plt.ylabel('y')
plt.legend(loc = 4)
plt.show()

k = [3, 4, 5]
```




**For k = 3**

In [6]:

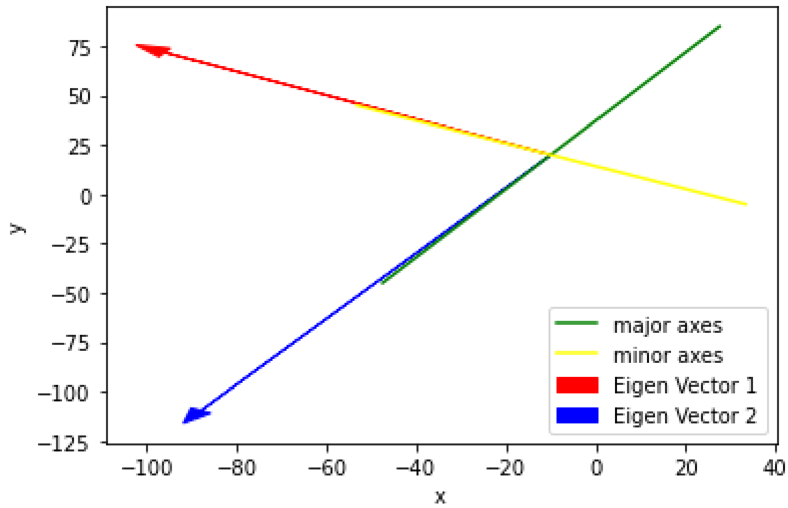
```
length_1 = k[0]*(eigen_values[0]**0.5)
length_2 = k[0]*(eigen_values[1]**0.5)

v_1 = vec_1*length_1
v_2 = vec_2*length_2

plt.arrow(origin_x, origin_y, v_1[0], v_1[1], color='red', label = 'Eigen Vector 1'
plt.arrow(origin_x, origin_y, v_2[0], v_2[1], color='blue', label = 'Eigen Vector 2
plt.plot(major_axes_x_new, major_axes_y_new, color = 'green',label = 'major axes')
plt.plot(minor_axes_x_new, minor_axes_y_new, color = 'yellow', label = 'minor axes'
plt.xlabel('x')
plt.ylabel('y')
plt.legend(loc = 4)
plt.show()
```




**For k = 4**

In [7]:

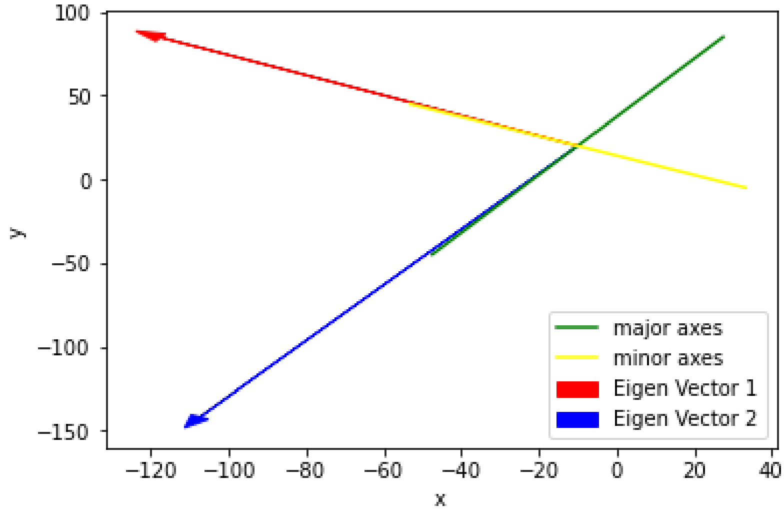
```
length_1 = k[1]*(eigen_values[0]**0.5)
length_2 = k[1]*(eigen_values[1]**0.5)

v_1 = vec_1*length_1
v_2 = vec_2*length_2

plt.arrow(origin_x, origin_y, v_1[0], v_1[1], color='red', label = 'Eigen Vector 1'
plt.arrow(origin_x, origin_y, v_2[0], v_2[1], color='blue', label = 'Eigen Vector 2
plt.plot(major_axes_x_new, major_axes_y_new, color = 'green',label = 'major axes')
plt.plot(minor_axes_x_new, minor_axes_y_new, color = 'yellow', label = 'minor axes'
plt.xlabel('x')
plt.ylabel('y')
plt.legend(loc = 4)
plt.show()
```




**For k = 5**

In [8]:

```
length_1 = k[2]*(eigen_values[0]**0.5)
length_2 = k[2]*(eigen_values[1]**0.5)

v_1 = vec_1*length_1
v_2 = vec_2*length_2

plt.arrow(origin_x, origin_y, v_1[0], v_1[1], color='red', label = 'Eigen Vector 1'
plt.arrow(origin_x, origin_y, v_2[0], v_2[1], color='blue', label = 'Eigen Vector 2
plt.plot(major_axes_x_new, major_axes_y_new, color = 'green',label = 'major axes')
plt.plot(minor_axes_x_new, minor_axes_y_new, color = 'yellow', label = 'minor axes'
plt.xlabel('x')
plt.ylabel('y')
plt.legend(loc = 4)
plt.show()
```




## Problem 4: Distribution estimation

Necessary user defined function that will be required for calculations

In [9]:

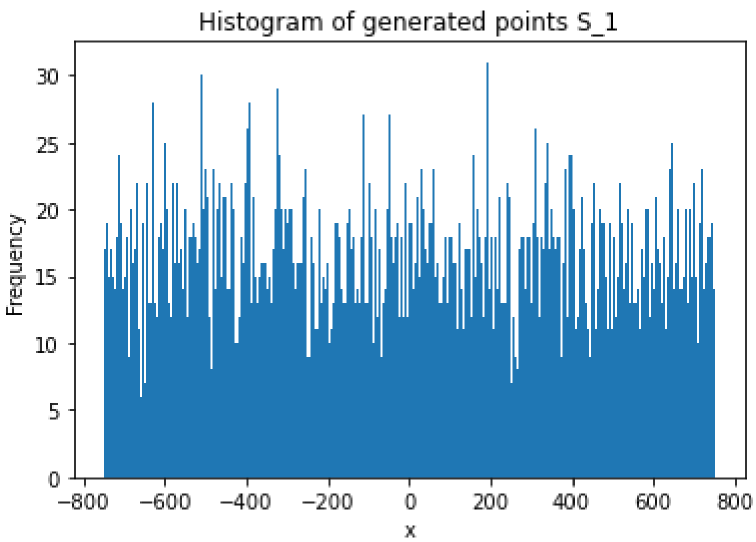
```
# Generate points for S_1
def generateNumbers(x_min, x_max,number_of_points):
    points = np.random.randint(x_min, x_max, number_of_points)
    return points

# Calculating Probability distribution function
# Will store x values in 0th index and cumulative value till that point in 1th inde
def generateCDF(pmf):
    cdf = []
    cum_sum = 0
    for index in range(len(pmf)):

        # make a fresh list to store cdf at a particular point
        cdf_at_x = []

        # first append the x value
        cdf_at_x.append(pmf[index][0])

        cum_sum += pmf[index][1]
        # then append the corresponding cdf value
        cdf_at_x.append(cum_sum)

        # finally append it to the cdf function
        cdf.append(cdf_at_x)

    cdf = np.array(cdf)
    return cdf
```

**Generating S_1**

```
at x_min, x_max = -750, 750
number_of_points = 5000
bin_size = 5
```

In [10]:

```python
# Generates points within given range and returns
def generatePoints(x_min, x_max,number_of_points):

    points = np.random.randint(x_min, x_max, number_of_points)
    return points

# Execution begins here
x_min, x_max = -750, 750
number_of_points_P = 5000
bin_size = 5

# Store the generated points into S_1
S_1 = generatePoints(x_min, x_max, number_of_points_P)

# plot the histogram for the generated points
plt.hist(S_1, bins = int((x_max-x_min)/bin_size))
plt.xlabel('x')
plt.ylabel('Frequency')
plt.title('Histogram of generated points S_1')
plt.show()

# Calculate and store x_values and bin_frequncies
hist, bin_edges = np.histogram(S_1, bins = int((x_max-x_min)/bin_size))

# probability mass function stores as (xi, Pi) where xi is event and Pi is probabil
pmf = [[(int(bin_edges[i]+bin_edges[i+1]))/2, hist[i]/number_of_points_P] for i in

# probability mass function of P
pmf_P = np.array(pmf)

# cumulative distribution of P
cdf_P = generateCDF(pmf_P)
# separate out the values and store into individual arrays x_values and probability
# x_values = np.array([pmf[i][0] for i in range(len(pmf))])
# prob_values = np.array([pmf[i][1] for i in range(len(pmf))])
```




**Problem 5: Data Generation and Distribution estimation**

In [11]:

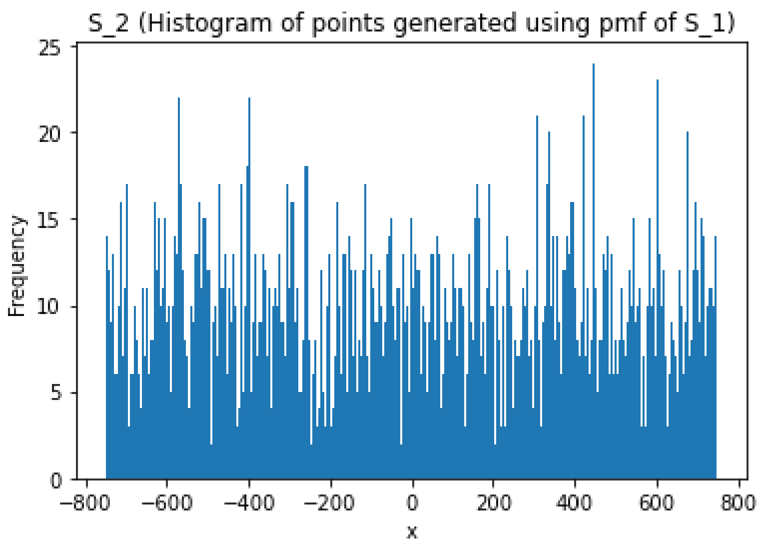
```
# Generates random number uniform distribution
def generateFromP(x_min, x_max, pmf, cdf, number_of_points):

    points = []

    # loop untill we generate required number of points
    while len(points) < number_of_points:

        # First generate a random value uniformly i.e. p~U(0,1)
        random_value = np.random.uniform(0,1)

        index = 0
        for index in range(len(pmf)):
            # if we hit cdf valeu greater than random value then generate x from th

            if cdf[index][1] > random_value:

                # if we get the random_value is even less than the first cdf_value
                if index == 0:
                    x = ((random_value - 0)*(cdf[index][0] - x_min)/pmf[index][1])
                else:
                    # since we are at (r+1) i.e. index so our calculation will be i
                    x = ((random_value - cdf[index-1][1])*(cdf[index][0] - cdf[inde

                # append the obtained x into points and break from loop to generate
                points.append(x)

                break

    points = np.array(points)

    return points
```

**Generating Q (will be stored in S_2) from the distribution of P (which having points in S_1)**

```
at x_min, x_max = -750, 750
   number_of_points = 3000
   bin_size = 10
```

In [12]:

```
x_min, x_max = -750, 750
number_of_points_Q = 3000
bin_size = 5
# Store the generated points into S_2
S_2 = generateFromP(x_min, x_max, pmf_P, cdf_P, number_of_points_Q)

# plot the histogram for the generated points
plt.hist(S_2, bins = int((x_max-x_min)/bin_size))
plt.xlabel('x')
plt.ylabel('Frequency')
plt.title('S_2 (Histogram of points generated using pmf of S_1)')
plt.show()

# Calculate and store x_values and bin_frequncies
hist, bin_edges = np.histogram(S_2, bins = int((x_max-x_min)/bin_size))

# probability mass function stores as (xi, Pi) where xi is event and Pi is probabil
pmf = [[(int(bin_edges[i]+bin_edges[i+1]))/2, hist[i]/number_of_points_Q] for i in

# probability mass function of Q
pmf_Q = np.array(pmf)

# cumulative distribution of PQ
cdf_Q = generateCDF(pmf_Q)
```




### Problem 6: Comparing the Distributions

In [13]:

```
# Calculate number of bins
number_of_bins = int((x_max-x_min)/bin_size)

# store the multiple and sqaure root into BC_sequence for final summation
BC_sequence = np.array([(pmf_P[i][1]*pmf_Q[i][1])**0.50 for i in range(number_of_bi

# sum all the elements obtained in BC_sequence
BC_of_PQ = sum(BC_sequence)

# Print result
print('Result')
print(f'\nx_min : {x_min} \tx_max : {x_max} \tbin size : {bin_size}')
print(f'\nFor P\nn : {number_of_points_P} \n')
print(f"\nFor Q\nn' : {number_of_points_Q} \n")
print(f'Bhattacharya Coefficient : {BC_of_PQ}')
```

```
Result

x_min : -750    x_max : 750     bin size : 5

For P
n : 5000


For Q
n' : 3000

Bhattacharya Coefficient : 0.9830786882176388
```

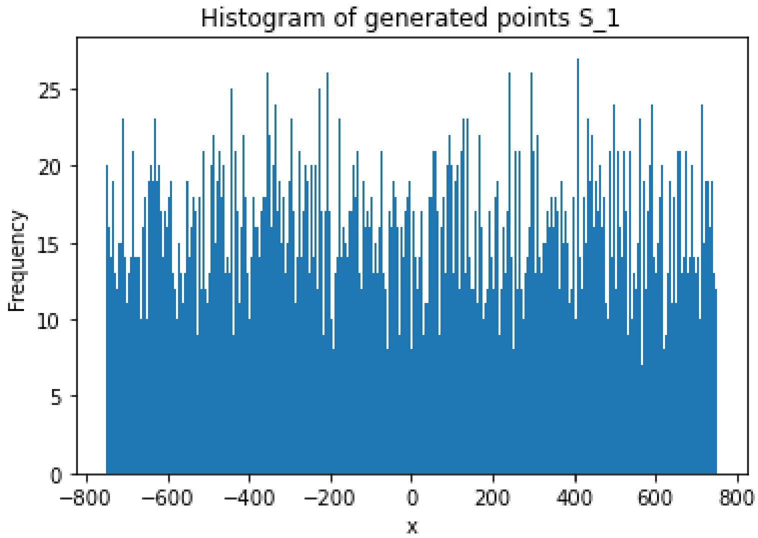
## Comparing Distribution #2:

#### Generating S_1

```
at x_min, x_max = -750, 750
number_of_points = 6000
bin_size = 4
```

In [14]:

```
# Execution begins here
x_min, x_max = -750, 750
number_of_points_P = 6000
bin_size = 4

# Store the generated points into S_1
S_1 = generatePoints(x_min, x_max, number_of_points_P)

# plot the histogram for the generated points
plt.hist(S_1, bins = int((x_max-x_min)/bin_size))
plt.xlabel('x')
plt.ylabel('Frequency')
plt.title('Histogram of generated points S_1')
plt.show()

# Calculate and store x_values and bin_frequncies
hist, bin_edges = np.histogram(S_1, bins = int((x_max-x_min)/bin_size))

# probability mass function stores as (xi, Pi) where xi is event and Pi is probabil
pmf = [[(int(bin_edges[i]+bin_edges[i+1]))/2, round(hist[i]/number_of_points_P, 5)]

# probability mass function of P
pmf_P = np.array(pmf)

# cumulative distribution of P
cdf_P = generateCDF(pmf_P)
```

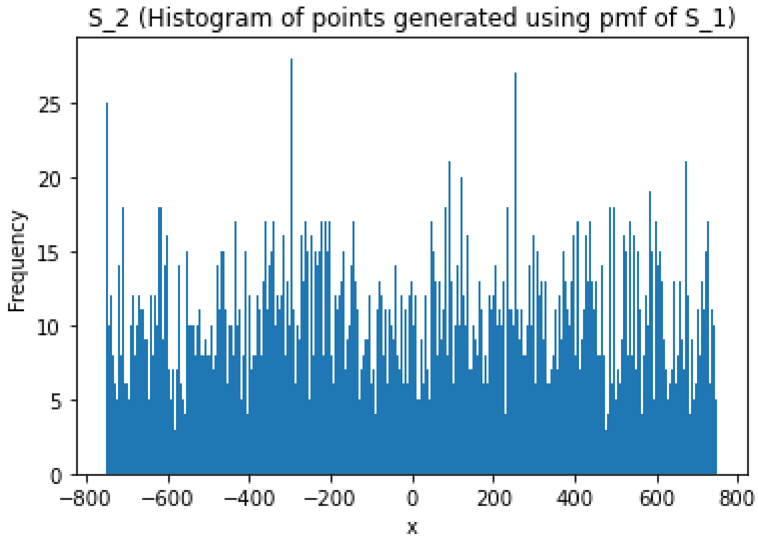



**Generating Q (will be stored in S_2) from the distribution of P (which having points in S_1)**

```
at x_min, x_max = -750, 750
   number_of_points = 4000
   bin_size = 4
```

In [15]:

```
x_min, x_max = -750, 750
number_of_points_Q = 4000
bin_size = 4
# Store the generated points into S_2
S_2 = generateFromP(x_min, x_max, pmf_P, cdf_P, number_of_points_Q)

# plot the histogram for the generated points
plt.hist(S_2, bins = int((x_max-x_min)/bin_size))
plt.xlabel('x')
plt.ylabel('Frequency')
plt.title('S_2 (Histogram of points generated using pmf of S_1)')
plt.show()

# Calculate and store x_values and bin_frequncies
hist, bin_edges = np.histogram(S_2, bins = int((x_max-x_min)/bin_size))

# probability mass function stores as (xi, Pi) where xi is event and Pi is probabil
pmf = [[(int(bin_edges[i]+bin_edges[i+1]))/2, round(hist[i]/number_of_points_Q, 5)]

# probability mass function of Q
pmf_Q = np.array(pmf)

# cumulative distribution of Q
cdf_Q = generateCDF(pmf_Q)
```




**Calculating Bhattacharya Coefficient and Results**

In [16]:

```
# Calculate number of bins
number_of_bins = int((x_max-x_min)/bin_size)

# store the multiple and sqaure root into BC_sequence for final summation
BC_sequence = np.array([(pmf_P[i][1]*pmf_Q[i][1])**0.50 for i in range(number_of_bi

# sum all the elements obtained in BC_sequence
BC_of_PQ = sum(BC_sequence)

# Print result
print('Result')
print(f'\nx_min : {x_min} \tx_max : {x_max} \tbin size : {bin_size}')
print(f'\nFor P\nn : {number_of_points_P} \n')
print(f"\nFor Q\nn' : {number_of_points_Q} \n")
print(f'Bhattacharya Coefficient : {BC_of_PQ}')
```

```
Result

x_min : -750    x_max : 750     bin size : 4

For P
n : 6000


For Q
n' : 4000

Bhattacharya Coefficient : 0.9851296908710356
```

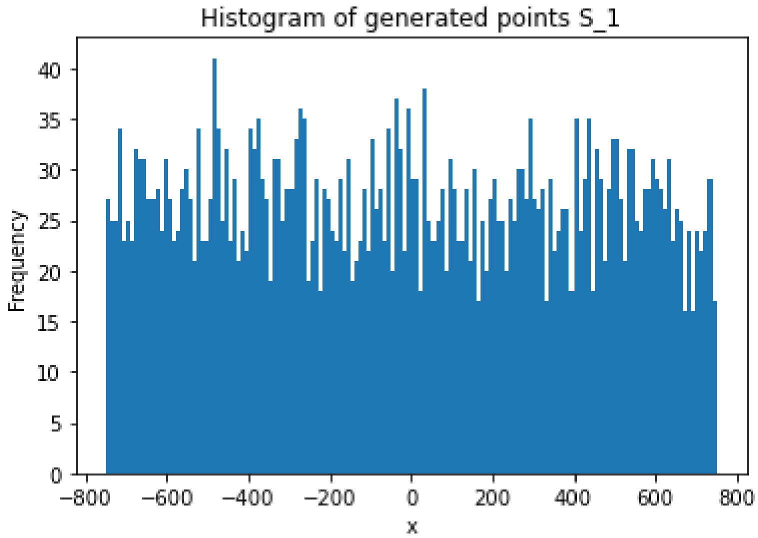
## Comparing Distribution #3:

#### Generating S_1

```
at x_min, x_max = -750, 750
number_of_points = 4000
bin_size = 10
```

In [17]:

```
# Execution begins here
x_min, x_max = -750, 750
number_of_points_P = 4000
bin_size = 10

# Store the generated points into S_1
S_1 = generatePoints(x_min, x_max, number_of_points_P)

# plot the histogram for the generated points
plt.hist(S_1, bins = int((x_max-x_min)/bin_size))
plt.xlabel('x')
plt.ylabel('Frequency')
plt.title('Histogram of generated points S_1')
plt.show()

# Calculate and store x_values and bin_frequncies
hist, bin_edges = np.histogram(S_1, bins = int((x_max-x_min)/bin_size))

# probability mass function stores as (xi, Pi) where xi is event and Pi is probabil
pmf = [[(int(bin_edges[i]+bin_edges[i+1]))/2, round(hist[i]/number_of_points_P, 5)]

# probability mass function of P
pmf_P = np.array(pmf)

# cumulative distribution of P
cdf_P = generateCDF(pmf_P)
```

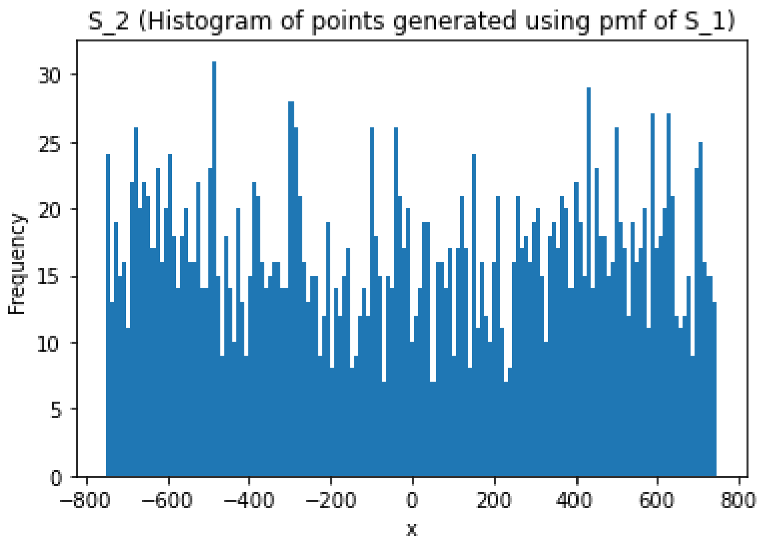



**Generating Q (will be stored in S_2) from the distribution of P (which having points in S_1)**

```
at x_min, x_max = -750, 750
   number_of_points = 2500
   bin_size = 10
```

In [18]:

```
x_min, x_max = -750, 750
number_of_points_Q = 2500
bin_size = 10
# Store the generated points into S_2
S_2 = generateFromP(x_min, x_max, pmf_P, cdf_P, number_of_points_Q)

# plot the histogram for the generated points
plt.hist(S_2, bins = int((x_max-x_min)/bin_size))
plt.xlabel('x')
plt.ylabel('Frequency')
plt.title('S_2 (Histogram of points generated using pmf of S_1)')
plt.show()

# Calculate and store x_values and bin_frequncies
hist, bin_edges = np.histogram(S_2, bins = int((x_max-x_min)/bin_size))

# probability mass function stores as (xi, Pi) where xi is event and Pi is probabil
pmf = [[(int(bin_edges[i]+bin_edges[i+1]))/2, round(hist[i]/number_of_points_Q, 5)]

# probability mass function of Q
pmf_Q = np.array(pmf)

# cumulative distribution of Q
cdf_Q = generateCDF(pmf_Q)
```




**Calculating Bhattacharya Coefficient and Results**

In [19]:

```python
# Calculate number of bins
number_of_bins = int((x_max-x_min)/bin_size)

# store the multiple and sqaure root into BC_sequence for final summation
BC_sequence = np.array([(pmf_P[i][1]*pmf_Q[i][1])**0.50 for i in range(number_of_bi

# sum all the elements obtained in BC_sequence
BC_of_PQ = sum(BC_sequence)

# Print result
print('Result')
print(f'\nx_min : {x_min} \tx_max : {x_max} \tbin size : {bin_size}')
print(f'\nFor P\nn : {number_of_points_P} \n')
print(f"\nFor Q\nn' : {number_of_points_Q} \n")
print(f'Bhattacharya Coefficient : {BC_of_PQ}')
```

```
Result

x_min : -750    x_max : 750     bin size : 10

For P
n : 4000


For Q
n' : 2500

Bhattacharya Coefficient : 0.991368718937178
```

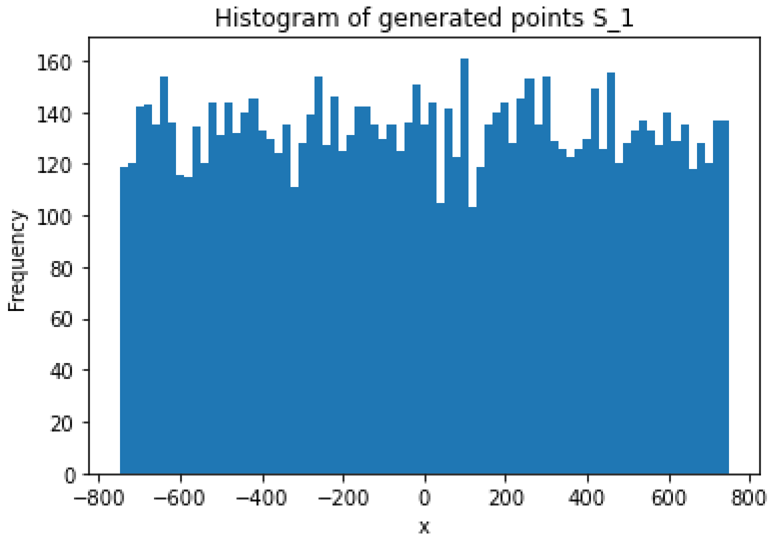
## Comparing Distribution #4:

#### Generating S_1

```
at x_min, x_max = -750, 750
number_of_points = 10000
bin_size = 20
```

In [20]:

```python
# Execution begins here
x_min, x_max = -750, 750
number_of_points_P = 10000
bin_size = 20

# Store the generated points into S_1
S_1 = generatePoints(x_min, x_max, number_of_points_P)

# plot the histogram for the generated points
plt.hist(S_1, bins = int((x_max-x_min)/bin_size))
plt.xlabel('x')
plt.ylabel('Frequency')
plt.title('Histogram of generated points S_1')
plt.show()

# Calculate and store x_values and bin_frequncies
hist, bin_edges = np.histogram(S_1, bins = int((x_max-x_min)/bin_size))

# probability mass function stores as (xi, Pi) where xi is event and Pi is probabil
pmf = [[(int(bin_edges[i]+bin_edges[i+1]))/2, round(hist[i]/number_of_points_P, 5)]

# probability mass function of P
pmf_P = np.array(pmf)

# cumulative distribution of P
cdf_P = generateCDF(pmf_P)
```

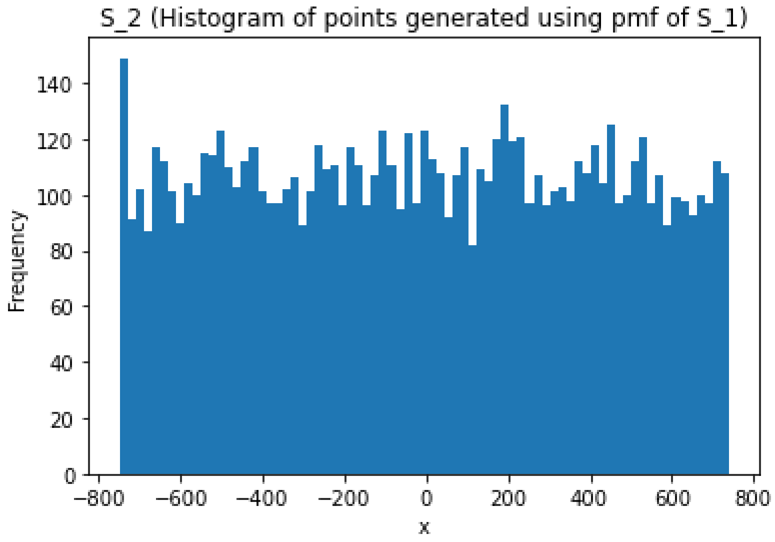



**Generating Q (will be stored in S_2) from the distribution of P (which having points in S_1)**

```
at x_min, x_max = -750, 750
   number_of_points = 8000
   bin_size = 20
```

In [21]:

```
x_min, x_max = -750, 750
number_of_points_Q = 8000
bin_size = 20
# Store the generated points into S_2
S_2 = generateFromP(x_min, x_max, pmf_P, cdf_P, number_of_points_Q)

# plot the histogram for the generated points
plt.hist(S_2, bins = int((x_max-x_min)/bin_size))
plt.xlabel('x')
plt.ylabel('Frequency')
plt.title('S_2 (Histogram of points generated using pmf of S_1)')
plt.show()

# Calculate and store x_values and bin_frequncies
hist, bin_edges = np.histogram(S_2, bins = int((x_max-x_min)/bin_size))

# probability mass function stores as (xi, Pi) where xi is event and Pi is probabil
pmf = [[(int(bin_edges[i]+bin_edges[i+1]))/2, round(hist[i]/number_of_points_Q, 5)]

# probability mass function of Q
pmf_Q = np.array(pmf)

# cumulative distribution of Q
cdf_Q = generateCDF(pmf_Q)
```




**Calculating Bhattacharya Coefficient and Results**

In [22]:

```python
# Calculate number of bins
number_of_bins = int((x_max-x_min)/bin_size)

# store the multiple and sqaure root into BC_sequence for final summation
BC_sequence = np.array([(pmf_P[i][1]*pmf_Q[i][1])**0.50 for i in range(number_of_bi

# sum all the elements obtained in BC_sequence
BC_of_PQ = sum(BC_sequence)

# Print result
print('Result')
print(f'\nx_min : {x_min} \tx_max : {x_max} \tbin size : {bin_size}')
print(f'\nFor P\nn : {number_of_points_P} \n')
print(f"\nFor Q\nn' : {number_of_points_Q} \n")
print(f'Bhattacharya Coefficient : {BC_of_PQ}')
```

```
Result

x_min : -750    x_max : 750     bin size : 20

For P
n : 10000


For Q
n' : 8000

Bhattacharya Coefficient : 0.9982280778965426
```