ECE 570 Assignment 2 Exercises

Name:

Exercise 1 (5/100 points)

In this exercise, you will need to write a simple function that reverses and triples the values in a list. For example: input `[1,2,3]`, output `[9,6,3]`.

In [11]:
```
def reverse_triple(input:list)->list:

    # <YOUR CODE HERE>
    # print(input)
    A_reverse = []
    for i in range(len(input) - 1, -1, -1):
      # print(input[i])
      A_reverse.append(3*input[i])
    return A_reverse
    # for item in input:
      # print(item)


A = [2,5,3,8,7,9,6]
print(reverse_triple(A))
```

```
[18, 27, 21, 24, 9, 15, 6]
```

Exercise 2 (30/100 points)

In this exercise, you will need to help visualize several different distributions.

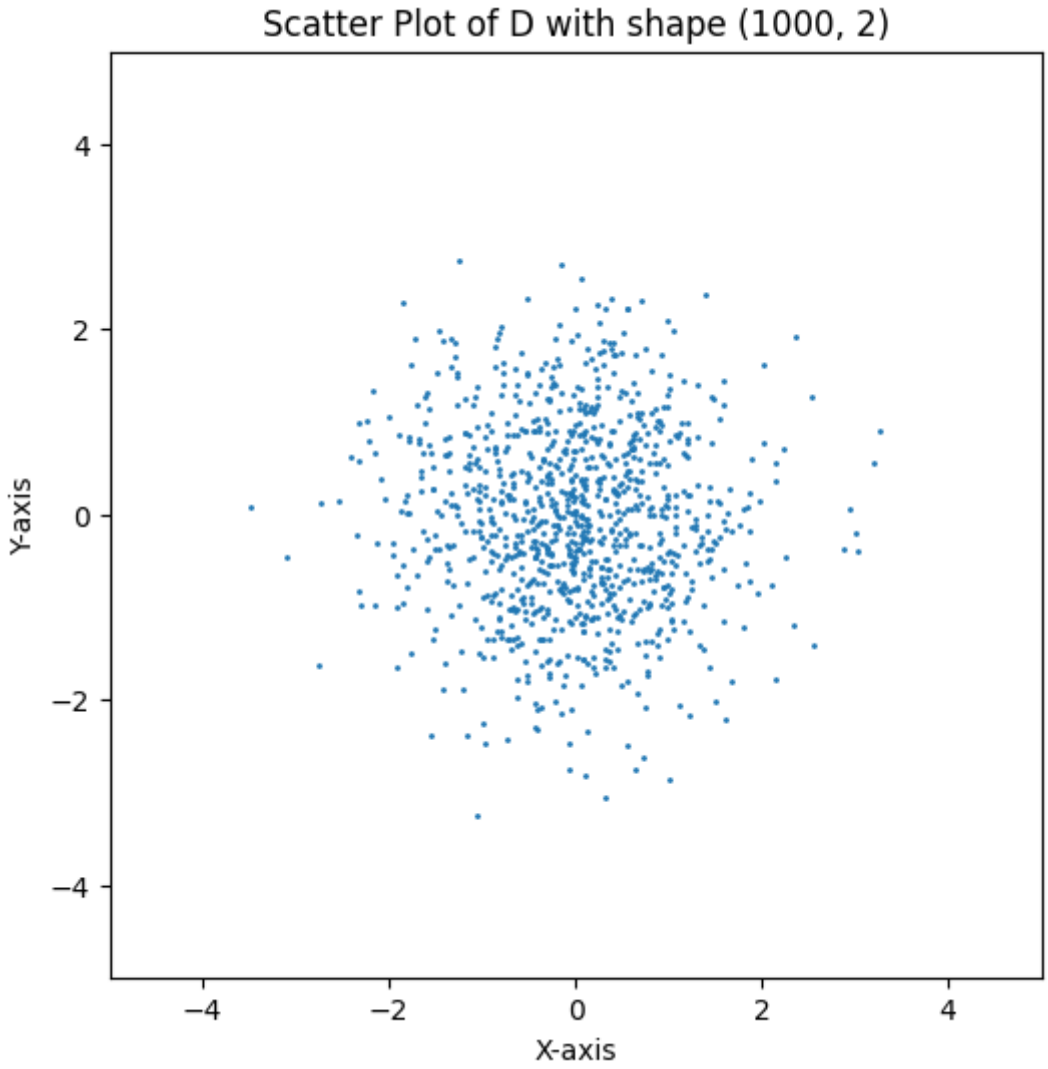
Task 1

1. Using Numpy to generate a vector **D** with the following property:
    - Each element is in a normal distribution.
    - Vector has the shape **2000x1**
2. Reshape the vector **D** into **1000x2**
3. Plot the graph in the following way:
    - Create a figure of size 6 by 6
    - Treat the two columns of the array **D** as the **x** and **y** coordinates of 2D points. Use scatter() to visualize all the spots and set the marker size to be 1
    - Let the plot shows the range `[-5,5]x[-5,5]`
    - Give the plot a title (indicating the shape of **D**), and also label the x-axis and y-axis

Note: It is always important to include necessary information (e.g. label, legend, title) so that readers won't get confused.

In [33]:

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

#  <YOUR CODE>
mean = 0
std_dev = 1

D = np.random.normal(mean, std_dev, (2000, 1))

D = D.reshape(1000, 2)

plt.figure(figsize=(6, 6))

plt.scatter(D[:, 0], D[:, 1], s=1)

plt.xlim(-5, 5)
plt.ylim(-5, 5)
shape=D.shape
plt.title(f'Scatter Plot of D with shape {shape}')
plt.ylabel("Y-axis")
plt.xlabel("X-axis")

# Display the plot
plt.show()
```



Task 2

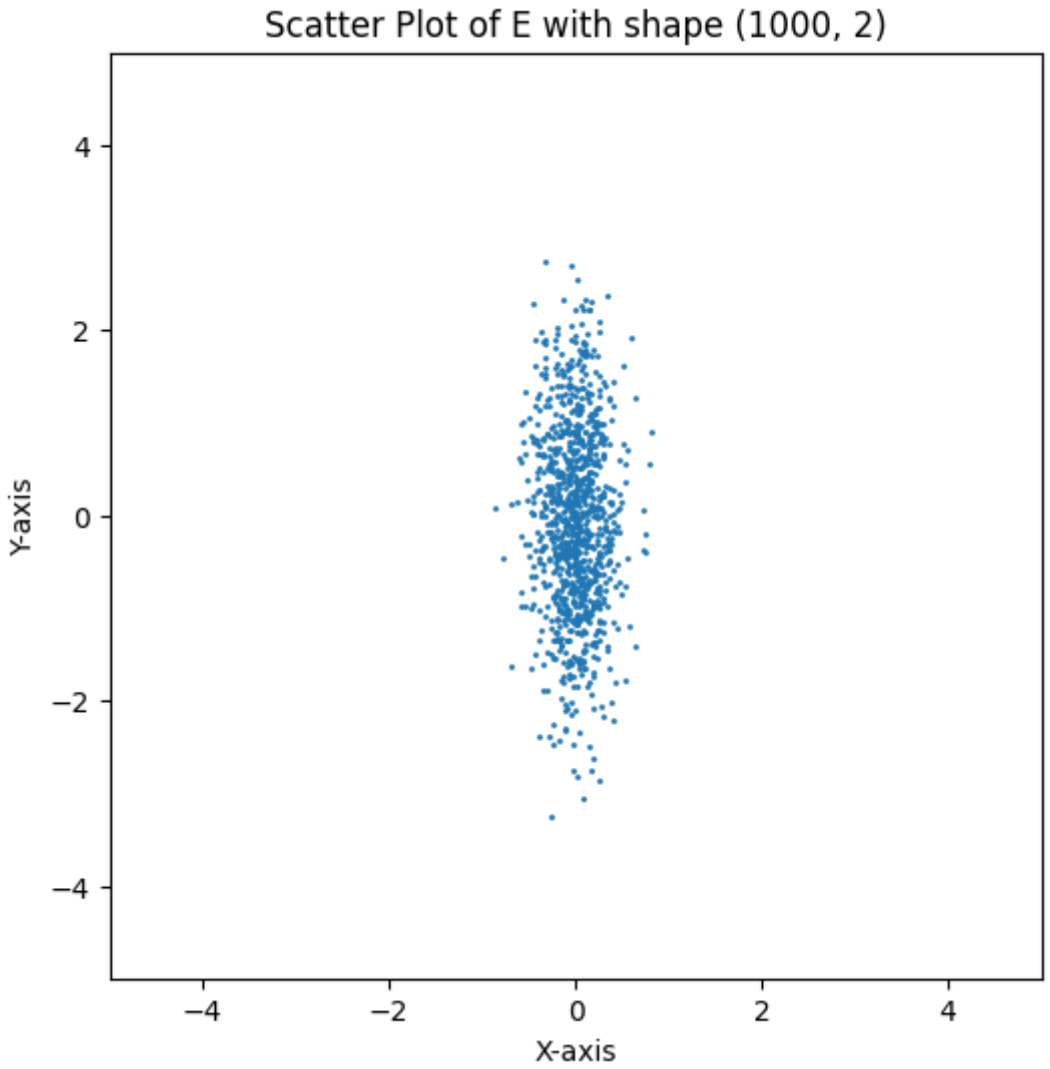
1. Create an array **R** =

$$\begin{bmatrix} 0.25 & 0 \\ 0 & 1 \end{bmatrix}$$

2. Compute **E** = **D** $\times$ **R**.
3. Repeat Step 3 above for **E**. (Title: shape of **E**)

In [39]:
```
#  <YOUR CODE>
# print(D)
R = np.array([[0.25, 0], [0, 1]])
E = np.dot(D, R)
# print(E)

plt.figure(figsize=(6, 6))

plt.scatter(E[:, 0], E[:, 1], s=1)

plt.xlim(-5, 5)
plt.ylim(-5, 5)
```

```
shape=E.shape
plt.title(f'Scatter Plot of E with shape {shape}')
plt.ylabel("Y-axis")
plt.xlabel("X-axis")

# Display the plot
plt.show()
```



Task 3

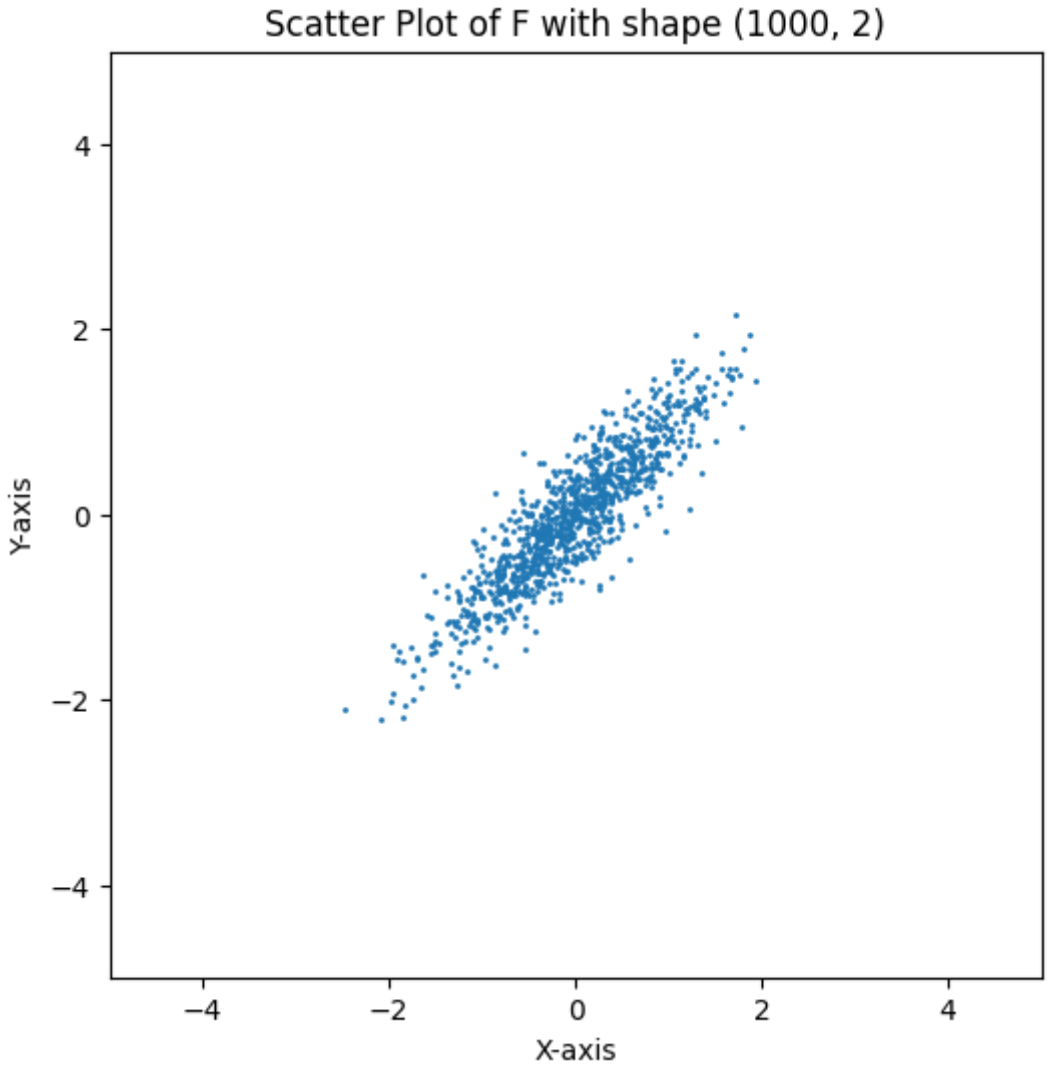
1. Create an array **R** =

$$\begin{bmatrix} \sqrt{(2)}/2 & -\sqrt{(2)}/2 \\ \sqrt{(2)}/2 & \sqrt{(2)}/2 \end{bmatrix}$$

2. Compute **F** = **E** $\times$ **R**.
3. Repeat Step 3 above for **F**. (Title: shape of **F**)

In [40]:
```
#  <YOUR CODE>

# print(E)
R=np.array([[(np.sqrt(2)/2), (-np.sqrt(2)/2)], [(np.sqrt(2)/2), (np.sqrt(2)/2)]
```

```
F = np.dot(E, R)
# print(F)

plt.figure(figsize=(6, 6))

plt.scatter(F[:, 0], F[:, 1], s=1)

plt.xlim(-5, 5)
plt.ylim(-5, 5)
shape=F.shape
plt.title(f'Scatter Plot of F with shape {shape}')
plt.ylabel("Y-axis")
plt.xlabel("X-axis")

# Display the plot
plt.show()
```

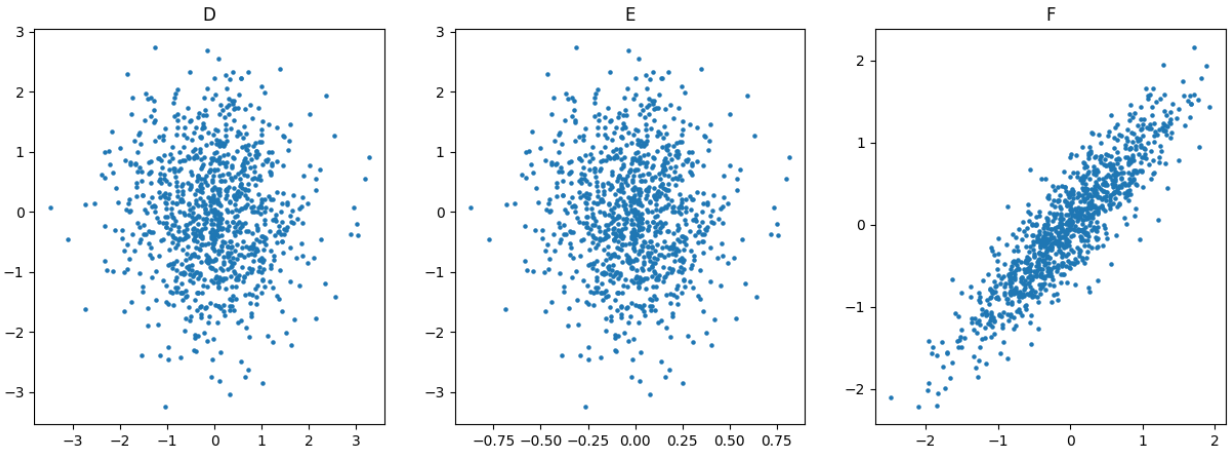


Task 4

Plot the above three graphs (D, E and F) in one figure using subplot.

In [42]:
```
# plot the above three figures in one figure
###### ###### Your code ###### ######
fig, axs = plt.subplots(1, 3, figsize=(15, 5))
```

```
# Plot D in the first subplot
axs[0].scatter(D[:, 0], D[:, 1], s=5)
axs[0].set_title('D')

# Plot E in the second subplot
axs[1].scatter(E[:, 0], E[:, 1], s=5)
axs[1].set_title('E')

# Plot F in the third subplot
axs[2].scatter(F[:, 0], F[:, 1], s=5)
axs[2].set_title('F')

# Adjust spacing between subplots
# plt.tight_layout()

# Show the plot
plt.show()




###### ###### ###### ###### ###### ######
```



Exercise 3 (65/100 points)

Task 1: Generate a sparse matrix

1. Generate a matrix **X** with size 100x50 with each element randomly picked from a uniform distribution **U**[0,1].
2. Use logical(boolean) indexing to set the elements in **X** to **0** whenever the value of the element is smaller than 0.90 (In this way, you should get the matrix to have roughly 90% of its elements zero's).
3. Use the function csr_matrix() to convert the matrix **X** into sparse matrix and call it **X_sparse**.
4. Hint: Check the shape and its meaning for the numpy array / matrix you are using to make sure it is what you want.

In [101…
```
import numpy as np
from scipy.sparse import csr_matrix

#  <YOUR CODE>
# 1
X = np.random.rand(100, 50)
#2
X[X < 0.90] = 0
# 3
X_sparse = csr_matrix(X)
shape_x = X.shape
print(shape_x, 'shape')
print(X_sparse.shape)

print(f'X has type {type(X)} and has {100-np.sum(X!=0)/50}% of zeros')
print(f'X_sparse has type {type(X_sparse)} and has {100-np.sum(X_sparse!=0)/50}
```

```
(100, 50) shape
(100, 50)
X has type <class 'numpy.ndarray'> and has 90.14% of zeros
X_sparse has type <class 'scipy.sparse._csr.csr_matrix'> and has 90.14% of zer
os
```

Task 2: Construct the power iteration function

Following the algorithm in the instructions notebook, write a function that takes a sparse matrix **X** and number of iterations as input and returns the top right singular vector of the centered matrix as output. We have provided some starter code and you need to fill in the rest.

In [103…
```
def power_iter(X, num_iter:int):

      v = np.random.randn(X.shape[1])  # Initialize with a random vector with s
      one_vec = np.ones(X.shape[0])  # All ones vector with shape (n,)

      mu_row_matrix = np.mean(X, axis=0)  # Returns a 1-row matrix with shape (
      mu = np.array(mu_row_matrix).squeeze()  # Convert from a sparse column ma

      # for _ in range(num_iter):
      #     # Center the data by subtracting the mean
      #     X_centered = X - mu

      #     # Compute the matrix-vector product (X_centered^T * X_centered) * v
      #     # Xv = X_centered.T.dot(X_centered.dot(v))
      #     Xv1 = np.dot(X_centered, v)
      #     print(Xv1.shape, 'xv1')
      #     Xv2 = np.transpose(X_centered)
      #     print(Xv2.shape, 'xv2')
      #     Xv = np.dot(Xv2, Xv1)
      #     print(Xv, "Xv")
      #     # Compute the norm of the resulting vector
      #     norm = np.linalg.norm(Xv)

      #     # Update the singular vector approximation
      #     v = Xv / norm
      for c in range(num_iter):
        v = X.T.dot(X.dot(v)) - mu.dot(one_vec.T.dot(X.dot(v))) - X.T.dot(one_v
```

```
        normal_form = np.linalg.norm(v)
        v = v/normal_form

      return v

v1_yours = power_iter(X_sparse,1000).squeeze()
print(v1_yours.shape)
```

```
(50,)
```

Task 3: Verifying your top singular vector

Using any method you like to verify the vector that is computed by your function is indeed the top right singular vector of the **centered** data matrix. First write another function that outputs the top right singular vector for sure (you can use the function svd(), note that it returns V^T instead of V). Then, the provided code will compute the mean absolute error (MAE) between the two functions you wrote. (Note: The provided evaluation code will correct for the fact that the two vectors can be the negative of each other singular value decomposition is only unique up to signs). The MAE should be close to machine precision (i.e., it should be less than about `1e-15`).

Note: This is for testing the correctness of your algorithm. It is often a very good idea to write simple checks of your code as you write it to avoid bugs early on in your development process. Do not worry about efficiency for this exercise.

Hint: The singular vectors are the columns of U and V, which is the rows of U^T and V^T.

In [104…

```
def verify_v1(X):
  # Compute the top right singular vector using other methods
  #  <YOUR CODE>
 U, S, VT = np.linalg.svd(X)
 svd_calc = VT[0,:]
 print(svd_calc.shape)
 return svd_calc

# Note here we just pass in the dense 2D array `X`
#  which represents the same matrix as `X_sparse`
v1_simple = verify_v1(X).squeeze()
# Compute a sign corrected difference between the vectors
#  (accounting for the fact that SVD is only unique up to signs)
diff_sign_corrected = np.sign(v1_yours[0]) * v1_yours - np.sign(v1_simple[0]) *
mae_corrected = np.mean(np.abs(diff_sign_corrected))
print(f'The average absolute difference of the two function output is {mae_corr
```

```
(50,)
The average absolute difference of the two function output is 0.14904364443235
518
```

Task 4: Timing the power iteration function

Below, try `power_iter` method with larger sparse and dense X matrices (100x100, 1000 x 1000, 10000x10000 with 10%, 1%, 0.1%, 0.01% nonzeros, i.e. very sparse) and time the difference. That is, compare the time time taken by `power_iter(X_sparse,10)` and

`power_iter(X,10)`. Use `time.time()` to capture the start and end times (subtracting them gets you the time in seconds).

What do you observe?

In [105…

```
import time
for threshold in [0.9, 0.99, 0.999, 0.9999]:
    print(f"Nonzero {(1 - threshold) * 100:.2f}%")
    for dim in [100, 1000, 10000]:
        ###### ###### Your code ###### ######
        X = np.random.rand(dim, dim)

        X[X < threshold] = 0

        X_sparse = csr_matrix(X)

        start_sparse = time.time()
        power_iter(X, 10)
        end_sparse= time.time()

        start_normal = time.time()
        power_iter(X, 10)
        end_normal = time.time()

        normal_time = end_normal - start_normal
        sparse_time = end_sparse - start_sparse

        ###### ###### ######### ###### ######
        ratio = normal_time/sparse_time
        print(f"Size: {dim}x{dim} - Sparse method is {ratio if ratio > 1 else 1
```

```
Nonzero 10.00%
Size: 100x100 - Sparse method is 1.619 times slower
Size: 1000x1000 - Sparse method is 1.032 times slower
Size: 10000x10000 - Sparse method is 1.007 times slower
Nonzero 1.00%
Size: 100x100 - Sparse method is 1.141 times slower
Size: 1000x1000 - Sparse method is 1.141 times slower
Size: 10000x10000 - Sparse method is 1.013 times faster
Nonzero 0.10%
Size: 100x100 - Sparse method is 1.152 times slower
Size: 1000x1000 - Sparse method is 1.239 times slower
Size: 10000x10000 - Sparse method is 1.003 times faster
Nonzero 0.01%
Size: 100x100 - Sparse method is 1.070 times slower
Size: 1000x1000 - Sparse method is 1.147 times slower
Size: 10000x10000 - Sparse method is 1.030 times slower
```

(Optional and ungraded, 0 points) Task 5: Going beyond

- In what scenarios we might find the power iteration method useful?
  - Google's original ranking algorithm called "PageRank" uses a variant of this power iteration on very sparse graphs that represent connections between websites. See PageRank.
- Can you optimize your algorithm further by avoiding reusing computations?