ECE 473 Assignment 2 Exercises

Name: Tina Xu

In [2]:
```
random_seed = 473 # seed to control randomness
```

Exercise 1 (5/100 points)

In this exercise, you will need to write a simple function that reverses and doubles the values in a list. For example: input `[1,2,3]`, output `[6,4,2]`.

In [3]:
```
def reverse_double(input:list)->list:

    # <YOUR CODE>
    input.reverse()
    return [x * 2 for x in input]


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

```
[12, 18, 14, 16, 6, 10, 4, 6]
```

Exercise 2 (30/100 points)

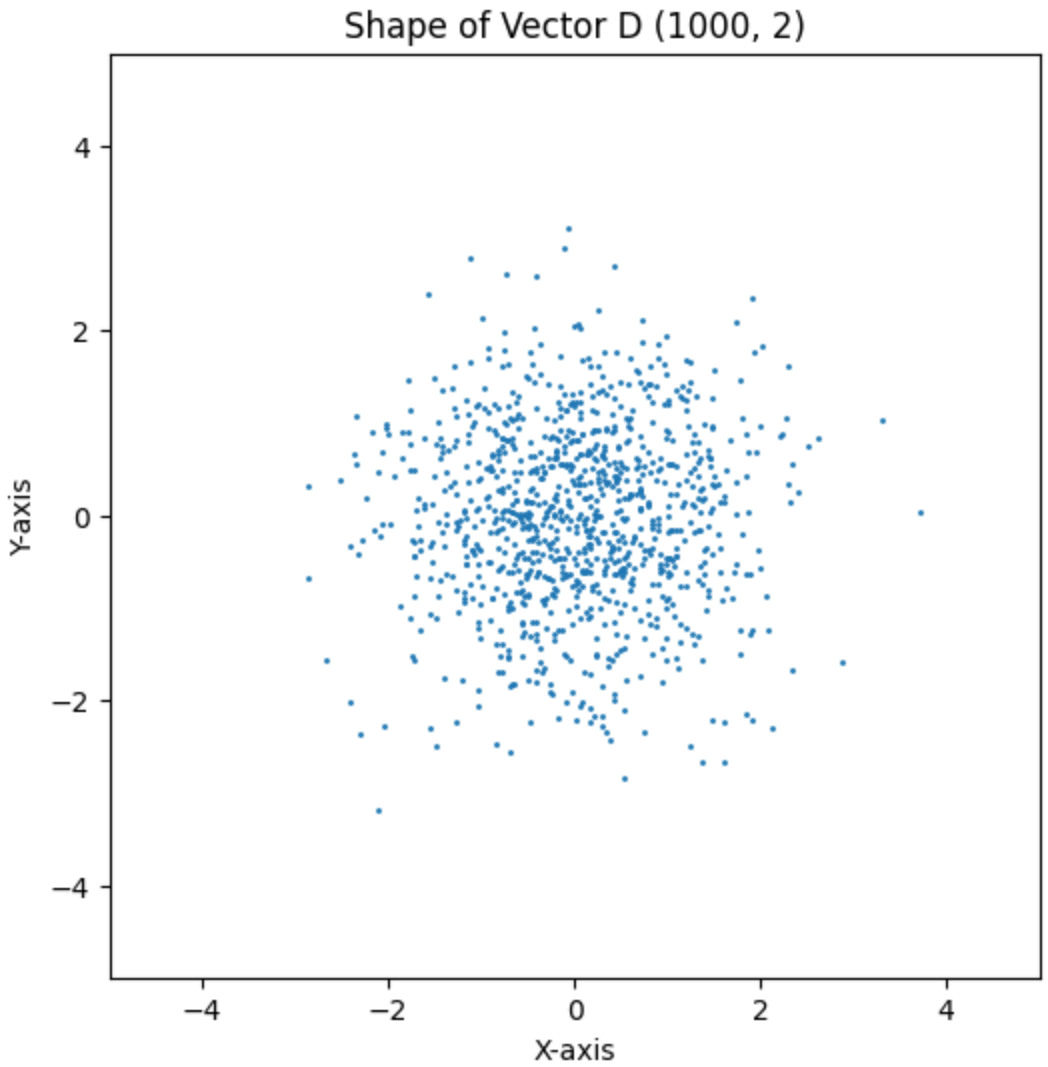
In this exercise, you will need to help visualize several different distrubtions.

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 [4]:
```
import numpy as np
import matplotlib.pyplot as plt
np.random.seed(random_seed) # to fix randomness
D = np.random.normal(0, 1, size=(2000, 1))
D = D.reshape(1000, 2)

#  <YOUR CODE>
plt.figure(figsize=(6,6))
plt.scatter(D[:, 0], D[:, 1], s=1)
plt.xlim([-5, 5])
plt.ylim([-5, 5])

plt.title(f"Shape of Vector D {D.shape}")
plt.xlabel("X-axis")
plt.ylabel("Y-axis")
plt.show()
```



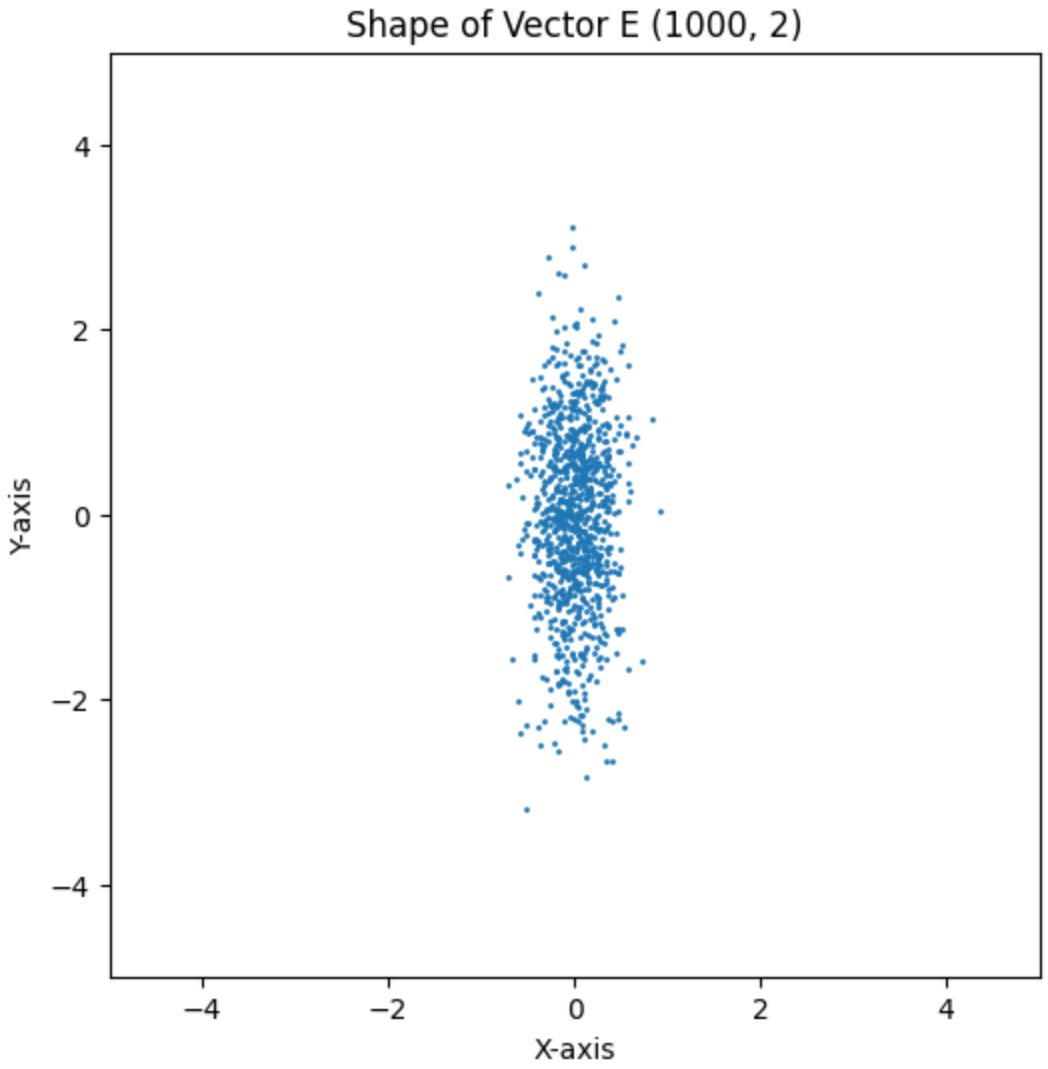
Task 2

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 [5]:
```
#  <YOUR CODE>
R = np.array([[0.25, 0], [0, 1]])
E = D@R
plt.figure(figsize=(6,6))
plt.scatter(E[:, 0], E[:, 1], s=1)
plt.xlim([-5, 5])
plt.ylim([-5, 5])

plt.title(f"Shape of Vector E {E.shape}")
plt.xlabel("X-axis")
plt.ylabel("Y-axis")
plt.show()
```



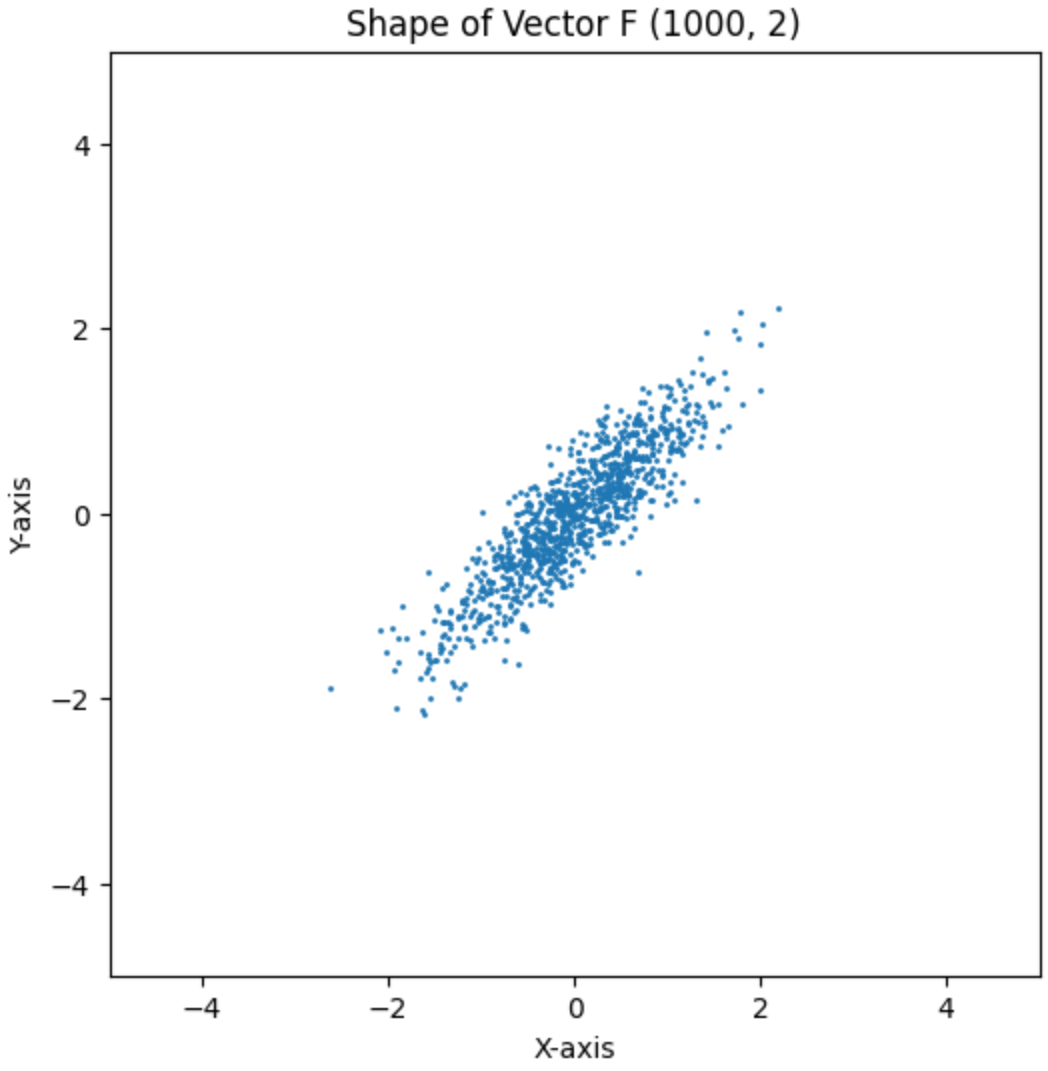
Task 3

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** × **R**.
3. Repeat Step 3 above for **F**. (Title: shape of **F**)

In [6]:
```
#  <YOUR CODE>
R = np.array([[np.sqrt(2)/2, -np.sqrt(2)/2], [np.sqrt(2)/2, np.sqrt(2)/2]])
F = E@R
plt.figure(figsize=(6,6))
plt.scatter(F[:, 0], F[:, 1], s=1)
plt.xlim([-5, 5])
plt.ylim([-5, 5])

plt.title(f"Shape of Vector F {F.shape}")
plt.xlabel("X-axis")
plt.ylabel("Y-axis")
plt.show()
```



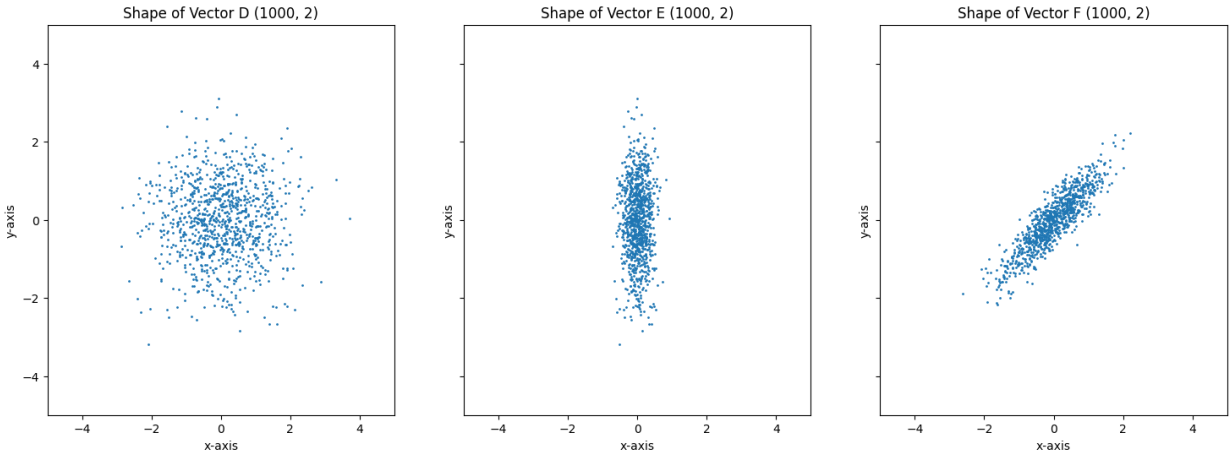
Task 4

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

In [7]:
```
# plot the above three figures in one figure
###### ###### Your code ###### ######
```

```
plt.figure(figsize=(18,6))         # Create a figure with size 15 by 6
fig, axes = plt.subplots(nrows=1, ncols=3, figsize=(18,6), sharey=True)  # Create a fi
# Plot on first subfigure
ax = axes[0]
ax.scatter(D[:, 0], D[:, 1], s=1)               # Plot Y w.r.t X
ax.set_title(f"Shape of Vector D {D.shape}")            # Give the current subplot a
ax.set_xlim([-5, 5])
ax.set_ylim([-5, 5])
ax.set_xlabel('x-axis')            # give x label a name
ax.set_ylabel('y-axis')            # give y label a name

ax = axes[1]
ax.scatter(E[:, 0], E[:, 1], s=1)
ax.set_title(f"Shape of Vector E {E.shape}")
ax.set_xlim([-5, 5])
ax.set_ylim([-5, 5])
ax.set_xlabel('x-axis')            # give x label a name
ax.set_ylabel('y-axis')            # give y label a name

ax = axes[2]
ax.scatter(F[:, 0], F[:, 1], s=1)
ax.set_title(f"Shape of Vector F {F.shape}")
ax.set_xlim([-5, 5])
ax.set_ylim([-5, 5])
ax.set_xlabel('x-axis')            # give x label a name
ax.set_ylabel('y-axis')            # give y label a name

plt.show()


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

```
<Figure size 1800x600 with 0 Axes>
```



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**.

In [8]:
```
import numpy as np
from scipy.sparse import csr_matrix
np.random.seed(random_seed) # to fix randomness

######  <YOUR CODE>  ###### ######
X = np.random.uniform(0, 1, size=(100, 50))
X[X < 0.90] = 0
X_sparse = csr_matrix(X)
###### ###### ###### ###### ######

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}% of ze
```

```
X has type <class 'numpy.ndarray'> and has 89.5% of zeros
X_sparse has type <class 'scipy.sparse._csr.csr_matrix'> and has 89.5% of zeros
```

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 [9]:
```
def power_iter(X, num_iter:int, rng = np.random.RandomState(random_seed)):

  v = rng.randn(X.shape[1])  # Initialize with random vector with shape (d,)
  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 (d, 1) since
  mu = np.array(mu_row_matrix).squeeze()  # Convert from a sparse column matrix to a d

  ######  <YOUR CODE>  ###### ######
  for _ in range(num_iter):
        v = X.T.dot(X.dot(v)) - mu.dot(one_vec.dot(X.dot(v))) - X.T.dot(one_vec).dot(m
        norm_v = np.linalg.norm(v)
        if norm_v != 0:
            v /= norm_v
  ###### ###### ###### ###### ######
  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.

In [10]:
```
def verify_v1(X):
  # Compute the top right singular vector using other methods
  #  <YOUR CODE>
  mean_column = np.mean(X, axis=0)
  X_centered = X - mean_column
  _, _, v1_svd = np.linalg.svd(X_centered, full_matrices=False)
  return v1_svd[0,:]

# 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]) * v1_sim
mae_corrected = np.mean(np.abs(diff_sign_corrected))
print(f'The average absolute difference of the two function output is {mae_corrected}'
```

```
The average absolute difference of the two function output is 1.4988444513308608e-16
```

Task 4: Comparing runtimes of `power_iter`

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,20)` and `power_iter(X,20)`. Use `time.time()` to capture the start and end times (subtracting them gets you the time in seconds).

What do you observe?

In [11]:
```
import time
np.random.seed(random_seed) # to fix randomness
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.uniform(0, 1, (dim, dim))
    X[X < threshold] = 0
    X_sparse = csr_matrix(X)

    start1 = time.time()
    power_iter(X_sparse,20)
    end1 = time.time()
    sparse_time = end1 - start1

    start2 = time.time()
    power_iter(X,20)
```

```
    end2 = time.time()
    normal_time = end2 - start2

###### ###### ######### ###### ######
    ratio = normal_time/sparse_time
    print(f"Size: {dim}x{dim} - Sparse method is {ratio if ratio > 1 else 1/ratio:.3f}
```

```
Nonzero 10.00%
Size: 100x100 - Sparse method is 3.138 times slower
Size: 1000x1000 - Sparse method is 2.420 times faster
Size: 10000x10000 - Sparse method is 2.057 times faster
Nonzero 1.00%
Size: 100x100 - Sparse method is 5.802 times slower
Size: 1000x1000 - Sparse method is 3.484 times faster
Size: 10000x10000 - Sparse method is 33.752 times faster
Nonzero 0.10%
Size: 100x100 - Sparse method is 6.338 times slower
Size: 1000x1000 - Sparse method is 8.492 times faster
Size: 10000x10000 - Sparse method is 295.772 times faster
Nonzero 0.01%
Size: 100x100 - Sparse method is 1.138 times slower
Size: 1000x1000 - Sparse method is 5.125 times faster
Size: 10000x10000 - Sparse method is 446.797 times faster
```

(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?