Week-3 Graded Assignment

December 30, 2024

[1]:
```python
import numpy as np
import matplotlib.pyplot as plt
import pandas as pd
import utils

import w3_unittest
```

[2]:
```python
def T(v):
    w = np.zeros((3,1))
    w[0,0] = 3*v[0,0]
    w[2,0] = -2*v[1,0]

    return w

v = np.array([[3], [5]])
w = T(v)

print("Original vector:\n", v, "\n\n Result of the transformation:\n", w)
```

```
Original vector:
 [[3]
 [5]]

 Result of the transformation:
 [[  9.]
 [  0.]
 [-10.]]
```

[3]:
```python
u = np.array([[1], [-2]])
v = np.array([[2], [4]])

k = 7

print("T(k*v):\n", T(k*v), "\n k*T(v):\n", k*T(v), "\n\n")
print("T(u+v):\n", T(u+v), "\n\n T(u)+T(v):\n", T(u)+T(v))
```

```
T(k*v):
 [[ 42.]
```

```
 [  0.]
 [-56.]]
 k*T(v):
 [[ 42.]
 [  0.]
 [-56.]]


T(u+v):
 [[ 9.]
 [ 0.]
 [-4.]]

 T(u)+T(v):
 [[ 9.]
 [ 0.]
 [-4.]]
```

[4]:
```
def L(v):
    A = np.array([[3,0], [0,0], [0,-2]])
    print("Transformation matrix:\n", A, "\n")
    w = A @ v

    return w

v = np.array([[3], [5]])
w = L(v)

print("Original vector:\n", v, "\n\n Result of the transformation:\n", w)
```

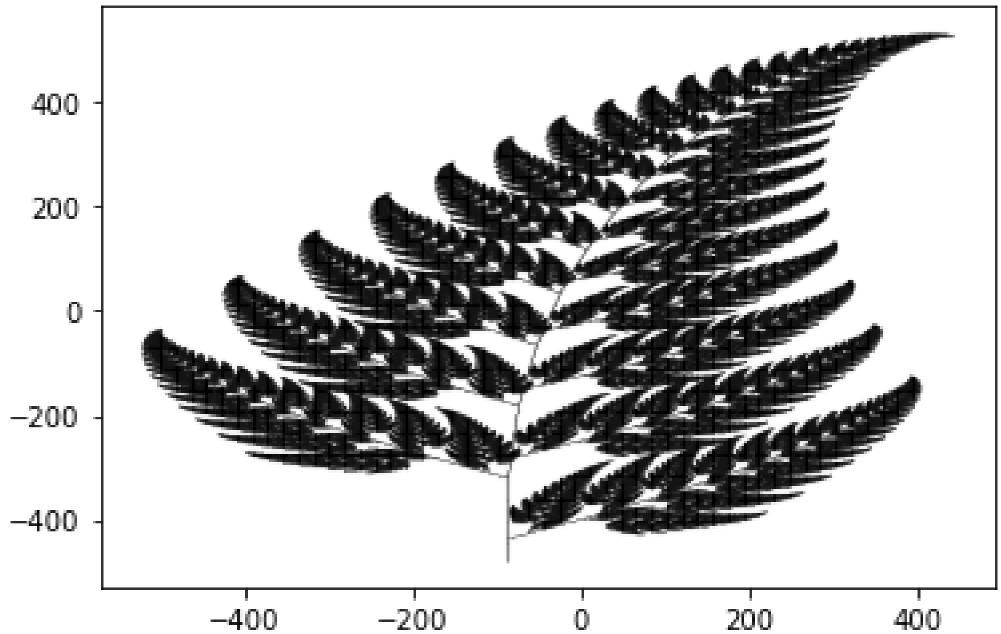
```
Transformation matrix:
 [[ 3  0]
 [ 0  0]
 [ 0 -2]]

Original vector:
 [[3]
 [5]]

 Result of the transformation:
 [[  9]
 [  0]
 [-10]]
```

[5]:
```
img = np.loadtxt('data/image.txt')
print('Shape: ',img.shape)
print(img)
```

```
Shape:  (2, 329076)
[[ 399.20891527  400.20891527  404.20891527 …  -88.79108473
   -88.79108473  -88.79108473]
 [ 534.18310664  534.18310664  534.18310664 … -476.81689336
  -477.81689336 -478.81689336]]
```

[6]:
```
plt.scatter(img[0], img[1], s = 0.001, color = 'black')
```

[6]: <matplotlib.collections.PathCollection at 0x7cbe50b76d90>

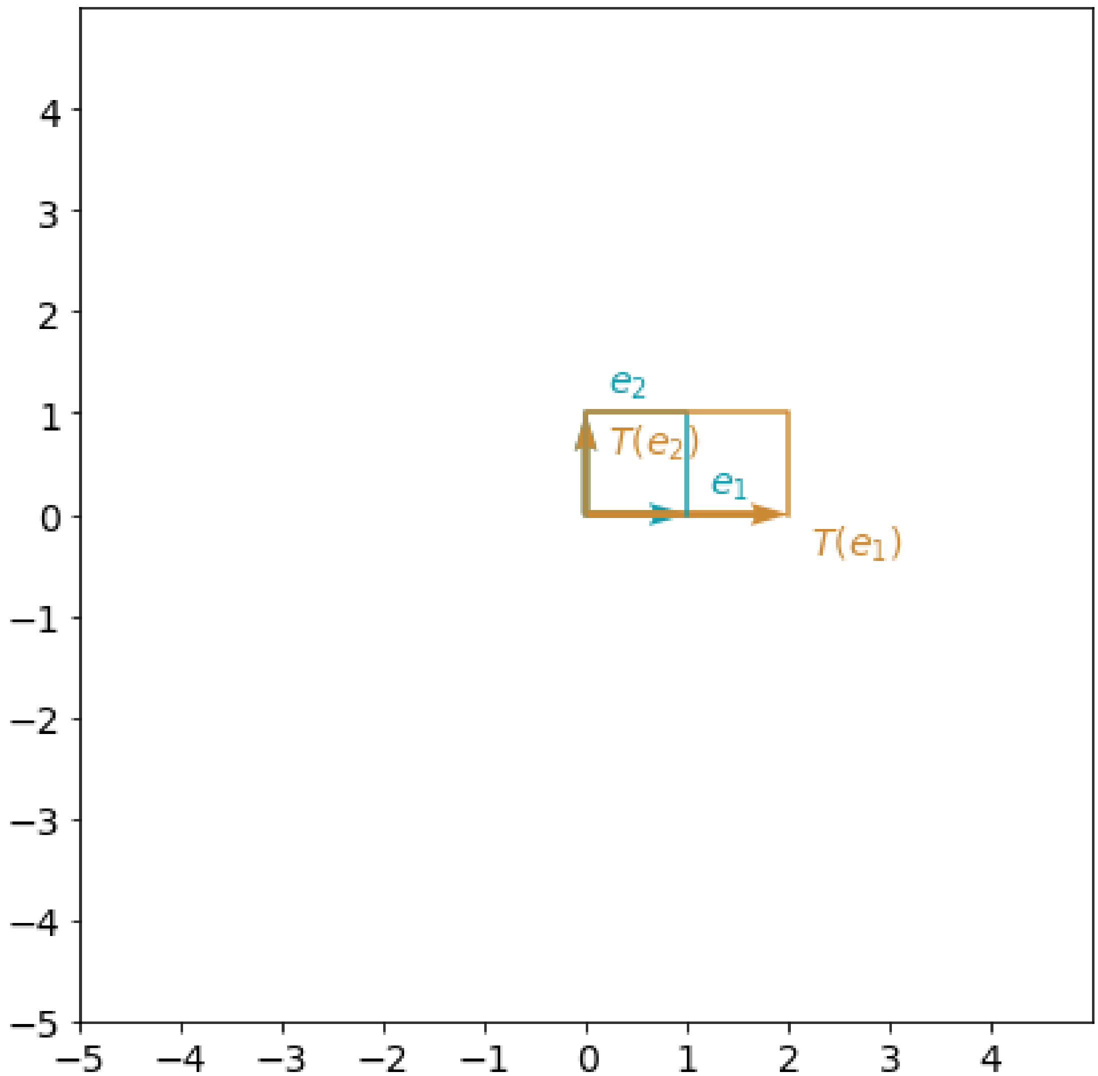


[7]:
```
def T_hscaling(v):
    A = np.array([[2,0], [0,1]])
    w = A @ v

    return w


def transform_vectors(T, v1, v2):
    V = np.hstack((v1, v2))
    W = T(V)

    return W

e1 = np.array([[1], [0]])
e2 = np.array([[0], [1]])
```

```python
transformation_result_hscaling = transform_vectors(T_hscaling, e1, e2)

print("Original vectors:\n e1= \n", e1, "\n e2=\n", e2,
      "\n\n Result of the transformation (matrix form):\n",␣
 ↪transformation_result_hscaling)
```

```
Original vectors:
 e1=
 [[1]
 [0]]
 e2=
 [[0]
 [1]]

 Result of the transformation (matrix form):
 [[2 0]
 [0 1]]
```

[8]:
```python
utils.plot_transformation(T_hscaling,e1,e2)
```



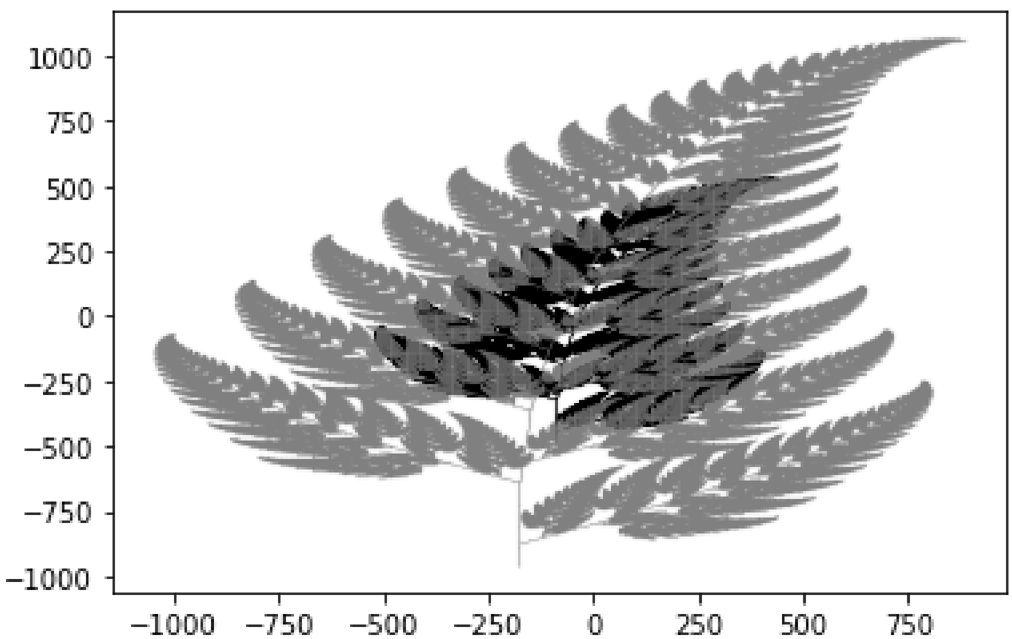
[9]:
```
# GRADED FUNCTION: T_stretch

def T_stretch(a, v):
    """
    Performs a 2D stretching transformation on a vector v using a stretching␣
↪factor a.

    Args:
        a (float): The stretching factor.
        v (numpy.array): The vector (or vectors) to be stretched.

    Returns:
        numpy.array: The stretched vector.
```

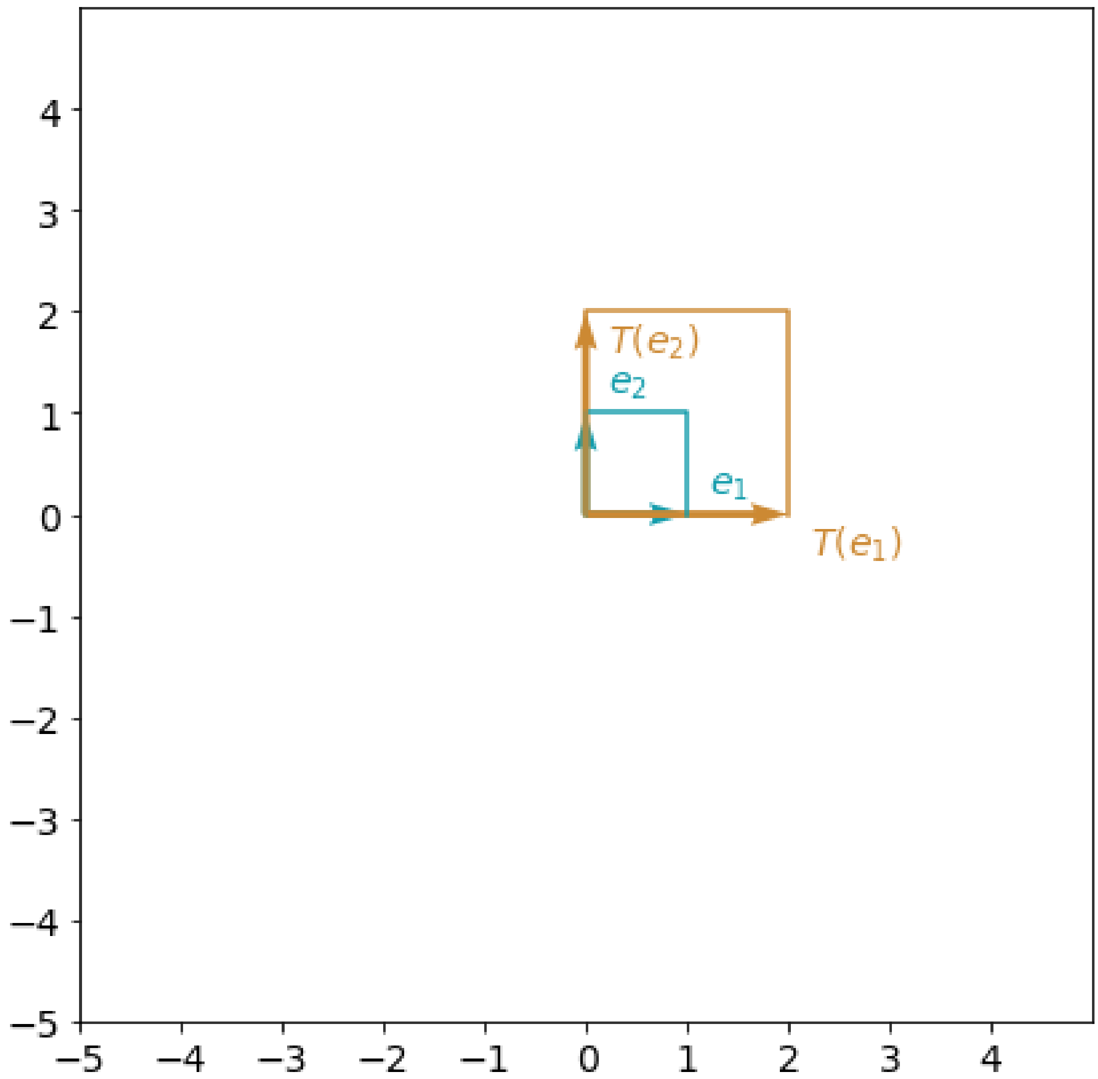
```
    """

    ### START CODE HERE ###
    # Define the transformation matrix
    T = np.array([[a,0], [0,a]])

    # Compute the transformation
    w = T @ v
    ### END CODE HERE ###

    return w
```

[10]:

```
w3_unittest.test_T_stretch(T_stretch)
plt.scatter(img[0], img[1], s = 0.001, color = 'black')
plt.scatter(T_stretch(2,img)[0], T_stretch(2,img)[1], s = 0.001, color = 'grey')
utils.plot_transformation(lambda v: T_stretch(2, v), e1,e2)
```

```
All tests passed
```





[11]:

```
# GRADED FUNCTION: T_hshear

def T_hshear(m, v):
    """
    Performs a 2D horizontal shearing transformation on an array v using a␣
 ↪shearing factor m.

    Args:
        m (float): The shearing factor.
        v (np.array): The array to be sheared.

    Returns:
        np.array: The sheared array.
```

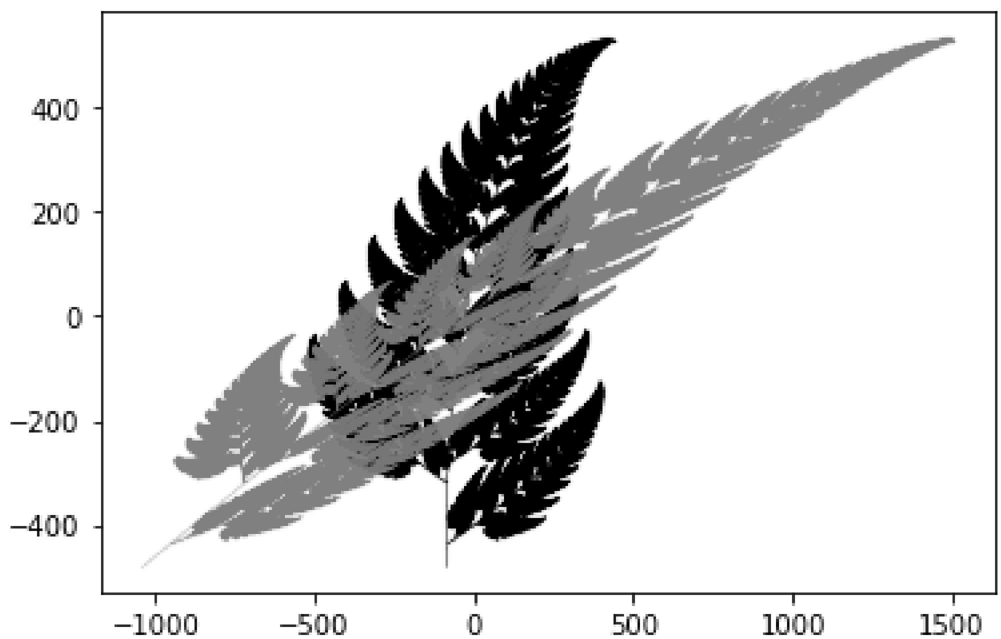
```
    """

    ### START CODE HERE ###
    # Define the transformation matrix
    T = np.array([[1,m], [0,1]])

    # Compute the transformation
    w = T @ v

    ### END CODE HERE ###

    return w
```

[12]:
```
w3_unittest.test_T_hshear(T_hshear)
```

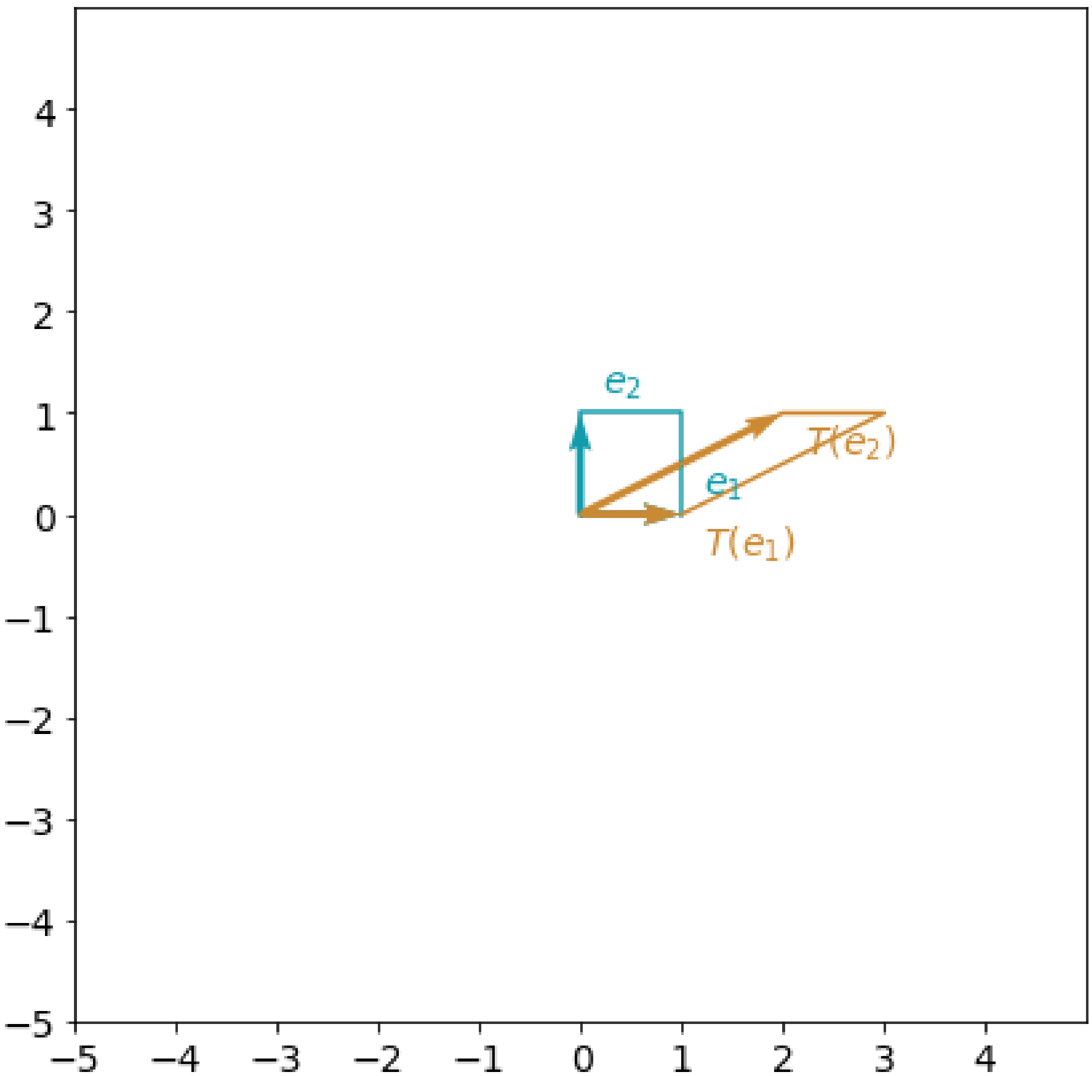
```
All tests passed
```

[13]:
```
plt.scatter(img[0], img[1], s = 0.001, color = 'black')
plt.scatter(T_hshear(2,img)[0], T_hshear(2,img)[1], s = 0.001, color = 'grey')
```

[13]: <matplotlib.collections.PathCollection at 0x7cbe586cebe0>



[14]:
```
utils.plot_transformation(lambda v: T_hshear(2, v), e1,e2)
```



[17]:
```
# GRADED FUNCTION: T_rotation
def T_rotation(theta, v):
    """
    Performs a 2D rotation transformation on an array v using a rotation angle␣
 ↪theta.

    Args:
        theta (float): The rotation angle in radians.
        v (np.array): The array to be rotated.

    Returns:
        np.array: The rotated array.
    """
```

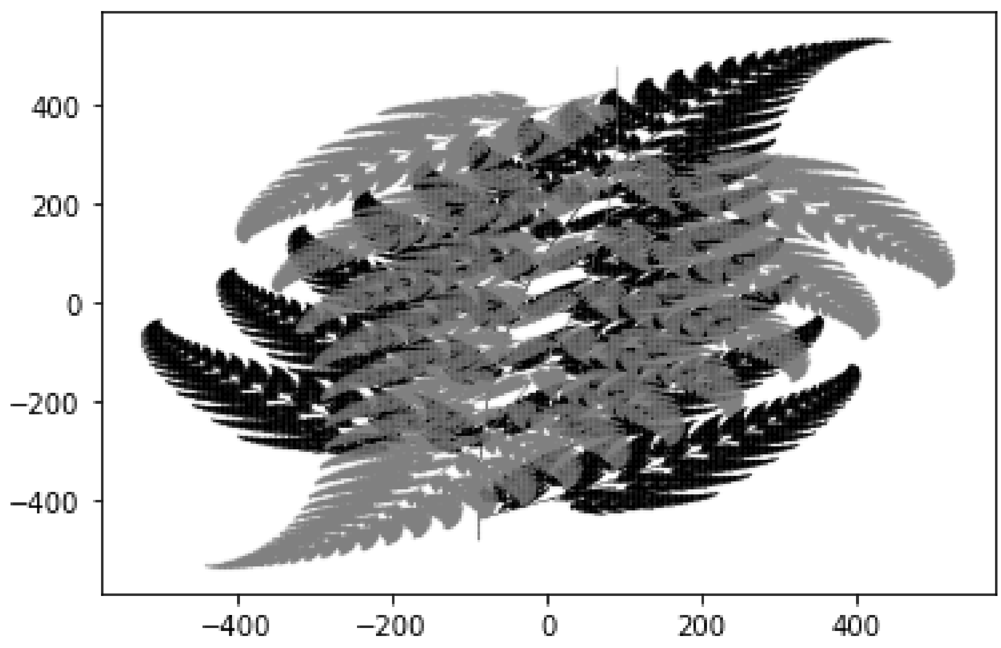
```
    ### START CODE HERE ###
    # Define the transformation matrix
    T = np.array([[np.cos(theta),-np.sin(theta)], [np.sin(theta),np.
↪cos(theta)]])

    # Compute the transformation
    w = T @ v

    ### END CODE HERE ###

    return w
```

[18]:
```
w3_unittest.test_T_rotation(T_rotation)
```

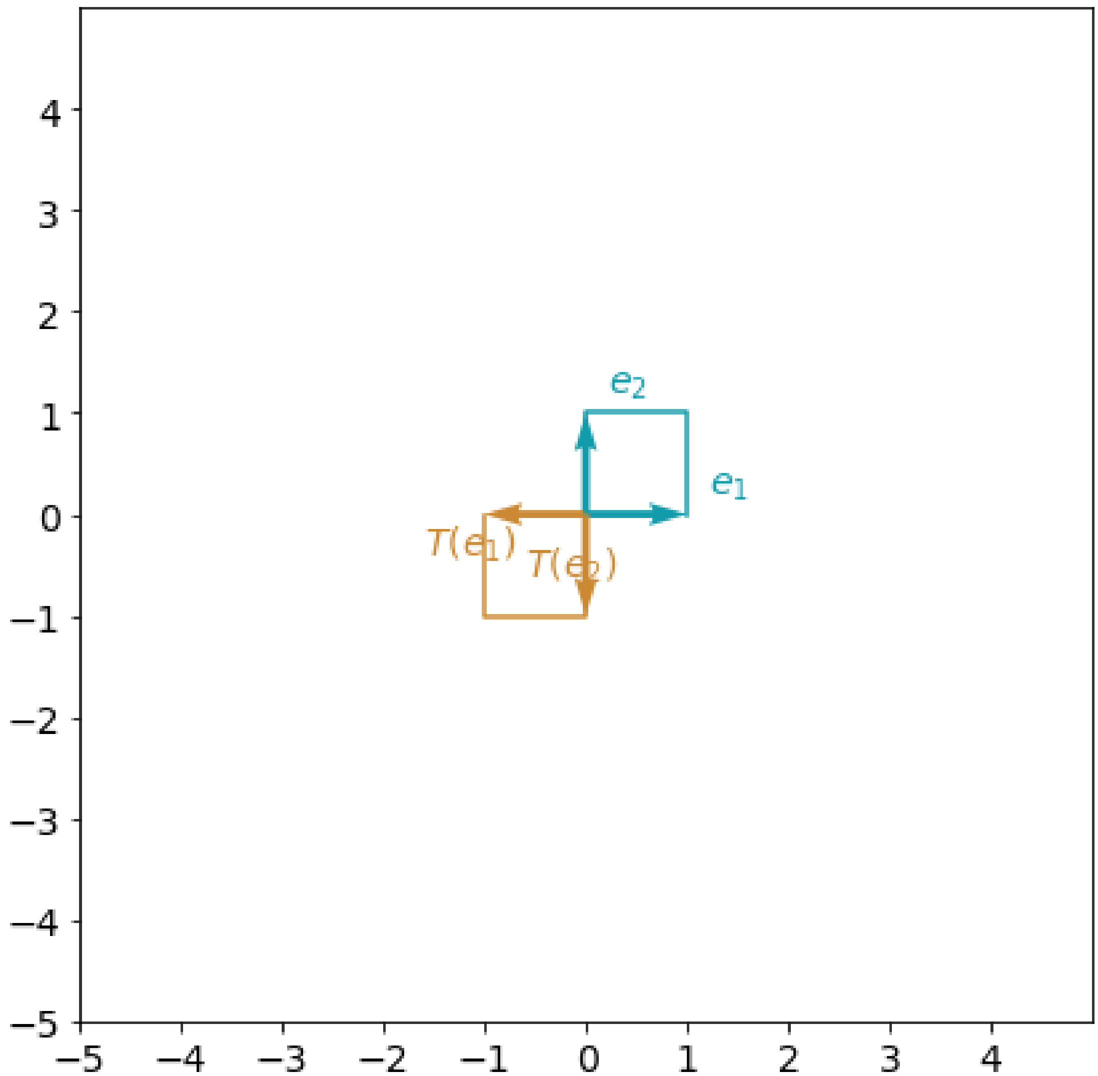
```
All tests passed
```

[19]:
```
plt.scatter(img[0], img[1], s = 0.001, color = 'black')
plt.scatter(T_rotation(np.pi,img)[0], T_rotation(np.pi,img)[1], s = 0.001,␣
↪color = 'grey')
```

[19]: <matplotlib.collections.PathCollection at 0x7cbe583776a0>



[20]:
```
utils.plot_transformation(lambda v: T_rotation(np.pi, v), e1,e2)
```



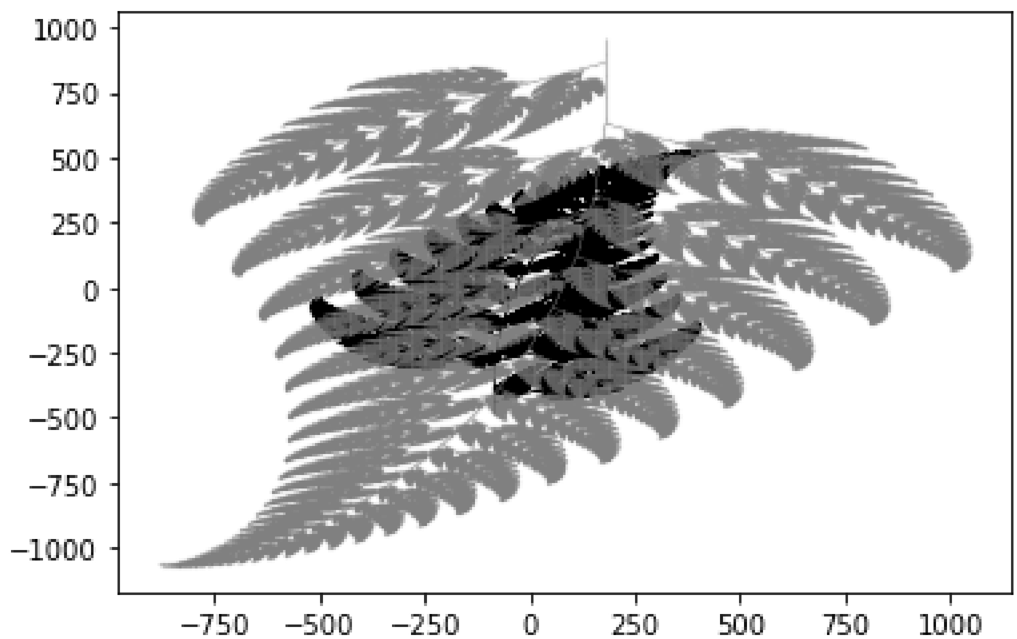
[25]:
```
def T_rotation_and_stretch(theta, a, v):
    """
    Performs a combined 2D rotation and stretching transformation on an array v␣
↪using a rotation angle theta and a stretching factor a.

    Args:
        theta (float): The rotation angle in radians.
        a (float): The stretching factor.
        v (np.array): The array to be transformed.

    Returns:
        np.array: The transformed array.
    """
```

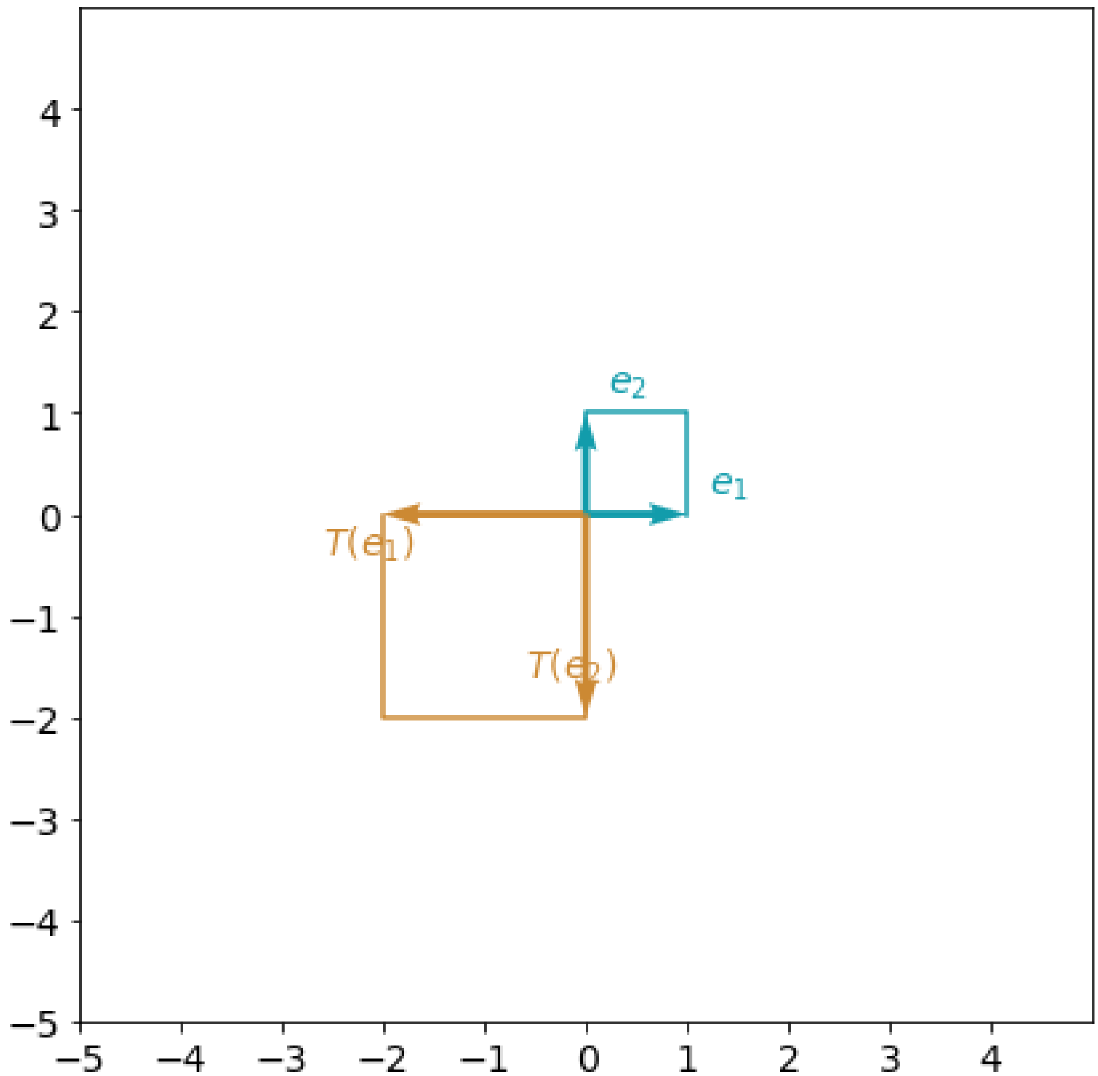
```
    ### START CODE HERE ###

    rotation_T = np.array([
        [np.cos(theta), -np.sin(theta)],
        [np.sin(theta), np.cos(theta)]
    ])
    stretch_T = np.array([
        [a, 0],
        [0, a]
    ])

    w = rotation_T @ (stretch_T @ v)

    ### END CODE HERE ###

    return w
```

[26]:
```
w3_unittest.test_T_rotation_and_stretch(T_rotation_and_stretch)
```

```
All tests passed
```

[27]:
```
plt.scatter(img[0], img[1], s = 0.001, color = 'black')
plt.scatter(T_rotation_and_stretch(np.pi,2,img)[0], T_rotation_and_stretch(np.
 ↪pi,2,img)[1], s = 0.001, color = 'grey')
utils.plot_transformation(lambda v: T_rotation_and_stretch(np.pi, 2, v), e1,e2)
```





[29]:
```
parameters = utils.initialize_parameters(2)
print(parameters)
```

```
{'W': array([[-0.00359009,  0.00983772]]), 'b': array([[0.]])}
```

[44]:
```
# GRADED FUNCTION: forward_propagation

def forward_propagation(X, parameters):
    """
    Argument:
    X -- input data of size (n_x, m), where n_x is the dimension input (in our␣
↪example is 2) and m is the number of training samples
```

```
    parameters -- python dictionary containing your parameters (output of␣
↪initialization function)

    Returns:
    Y_hat -- The output of size (1, m)
    """
    # Retrieve each parameter from the dictionary "parameters".
    W = parameters["W"]
    b = parameters["b"]

    # Implement Forward Propagation to calculate Z.
    ### START CODE HERE ### (~ 2 lines of code)
    Z = (W @ X)
    Y_hat = Z
    ### END CODE HERE ###


    return Y_hat
```

[45]:
```
w3_unittest.test_forward_propagation(forward_propagation)
```

```
All tests passed
```

[46]:
```
def compute_cost(Y_hat, Y):
    """
    Computes the cost function as a sum of squares

    Arguments:
    Y_hat -- The output of the neural network of shape (n_y, number of examples)
    Y -- "true" labels vector of shape (n_y, number of examples)

    Returns:
    cost -- sum of squares scaled by 1/(2*number of examples)

    """
    # Number of examples.
    m = Y.shape[1]

    # Compute the cost function.
    cost = np.sum((Y_hat - Y)**2)/(2*m)

    return cost
```

[47]:
```
# GRADED FUNCTION: nn_model

def nn_model(X, Y, num_iterations=1000, print_cost=False):
    """
```

```
    Arguments:
    X -- dataset of shape (n_x, number of examples)
    Y -- labels of shape (1, number of examples)
    num_iterations -- number of iterations in the loop
    print_cost -- if True, print the cost every iteration

    Returns:
    parameters -- parameters learnt by the model. They can then be used to make␣
↪predictions.
    """

    n_x = X.shape[0]

    # Initialize parameters
    parameters = utils.initialize_parameters(n_x)

    # Loop
    for i in range(0, num_iterations):

        ### START CODE HERE ### (~ 2 lines of code)
        # Forward propagation. Inputs: "X, parameters". Outputs: "Y_hat".
        Y_hat = forward_propagation(X, parameters)

        # Cost function. Inputs: "Y_hat, Y". Outputs: "cost".
        cost = compute_cost(Y_hat, Y)
        ### END CODE HERE ###


        # Parameters update.
        parameters = utils.train_nn(parameters, Y_hat, X, Y, learning_rate = 0.
↪001)

        # Print the cost every iteration.
        if print_cost:
            if i%100 == 0:
                print ("Cost after iteration %i: %f" %(i, cost))

    return parameters
```

[48]:
```
w3_unittest.test_nn_model(nn_model)
```

```
All tests passed
```

[49]:
```
df = pd.read_csv("data/toy_dataset.csv")
df.head()
X = np.array(df[['x1','x2']]).T
Y = np.array(df['y']).reshape(1,-1)
```

[50]:
```
parameters = nn_model(X,Y, num_iterations = 5000, print_cost= True)
```

```
Cost after iteration 0: 0.498391
Cost after iteration 100: 0.411451
Cost after iteration 200: 0.339714
Cost after iteration 300: 0.280522
Cost after iteration 400: 0.231681
Cost after iteration 500: 0.191380
Cost after iteration 600: 0.158127
Cost after iteration 700: 0.130689
Cost after iteration 800: 0.108049
Cost after iteration 900: 0.089367
Cost after iteration 1000: 0.073953
Cost after iteration 1100: 0.061234
Cost after iteration 1200: 0.050739
Cost after iteration 1300: 0.042080
Cost after iteration 1400: 0.034934
Cost after iteration 1500: 0.029038
Cost after iteration 1600: 0.024174
Cost after iteration 1700: 0.020159
Cost after iteration 1800: 0.016847
Cost after iteration 1900: 0.014114
Cost after iteration 2000: 0.011859
Cost after iteration 2100: 0.009998
Cost after iteration 2200: 0.008463
Cost after iteration 2300: 0.007196
Cost after iteration 2400: 0.006150
Cost after iteration 2500: 0.005288
Cost after iteration 2600: 0.004576
Cost after iteration 2700: 0.003989
Cost after iteration 2800: 0.003504
Cost after iteration 2900: 0.003104
Cost after iteration 3000: 0.002775
Cost after iteration 3100: 0.002502
Cost after iteration 3200: 0.002278
Cost after iteration 3300: 0.002092
Cost after iteration 3400: 0.001939
Cost after iteration 3500: 0.001813
Cost after iteration 3600: 0.001709
Cost after iteration 3700: 0.001623
Cost after iteration 3800: 0.001552
Cost after iteration 3900: 0.001494
Cost after iteration 4000: 0.001445
Cost after iteration 4100: 0.001406
Cost after iteration 4200: 0.001373
Cost after iteration 4300: 0.001346
Cost after iteration 4400: 0.001323
```

```
Cost after iteration 4500: 0.001305
Cost after iteration 4600: 0.001290
Cost after iteration 4700: 0.001277
Cost after iteration 4800: 0.001267
Cost after iteration 4900: 0.001258
```

[51]:
```python
# GRADED FUNCTION: predict

def predict(X, parameters):

    W = parameters['W']
    b = parameters['b']

    Z = np.dot(W, X) + b

    return Z
```

[52]:
```python
y_hat = predict(X,parameters)
```

[53]:
```python
df['y_hat'] = y_hat[0]
```

[54]:
```python
for i in range(10):
    print(f"(x1,x2) = ({df.loc[i,'x1']:.2f}, {df.loc[i,'x2']:.2f}): Actual␣
 ↪value: {df.loc[i,'y']:.2f}. Predicted value: {df.loc[i,'y_hat']:.2f}")
```

```
(x1,x2) = (-0.82, -0.52): Actual value: -0.94. Predicted value: -0.95
(x1,x2) = (-0.67, 1.53): Actual value: 0.75. Predicted value: 0.68
(x1,x2) = (-0.16, 0.27): Actual value: 0.03. Predicted value: 0.09
(x1,x2) = (2.14, 0.43): Actual value: 1.80. Predicted value: 1.78
(x1,x2) = (0.16, 0.26): Actual value: 0.31. Predicted value: 0.30
(x1,x2) = (-0.80, 2.07): Actual value: 1.12. Predicted value: 0.99
(x1,x2) = (-2.64, -1.36): Actual value: -2.85. Predicted value: -2.81
(x1,x2) = (1.04, -0.38): Actual value: 0.45. Predicted value: 0.43
(x1,x2) = (0.15, 0.19): Actual value: 0.25. Predicted value: 0.25
(x1,x2) = (-1.57, -0.34): Actual value: -1.31. Predicted value: -1.33
```

[ ]:
```python
```