```
# Kenneth Meyer
# CSE 382M hw2
# 2/11/23

from sys import path

import numpy as np
import matplotlib.pyplot as plt
from scipy.stats import ortho_group
from math import comb
from PIL import Image

## answers to questions will be coded as functions so I don't need to run things
repeatedly ##

def orthotest(A):
    '''
        Checks the orthogonality of columns of a matrix A, m x n
    '''
    n = A.shape[1]
    combos = comb(n,2)
    errors = np.zeros(combos)

    count = 0
    for i in range(0,n-1):
        for j in range(i+1,n):
            errors[count] =
(180/np.pi)*np.arccos(np.dot(np.abs(A[:,i]),np.abs(A[:,j]))
/(np.linalg.norm(A[:,i])*np.linalg.norm(A[:,j])))
            #errors[count] = np.abs(np.dot(A[:,i],A[:,j])) # dot product between
vectors to examine orthogonality
            count = count + 1

    # potentially do a bit more analysis (mean, standard deviation) to compare the
errors

    return errors

def plot_ortho_errors(errors,filename):
    '''
        Saves histogram of errors to "/figs" in cwd
        PARAMETERS
        ----------
        errors : 1-d np.array of angles between vectors
        filename : str, name of plot to save (.png)
        OUTPUT
        ------
        none
    '''
    filepath = "./figs/" + filename

    counts, bins = np.histogram(errors)
    plt.stairs(counts, bins)
    plt.savefig(filepath)
    plt.close()
```

```
# Q1: construction of "almost orthogonal" vectors in 100-d space
def q1():
    '''
        No inputs, just executes question 1a.
    '''

    # 1000 orthonormal 1000-d vectors
    #A = np.random.rand((1000,1000))
    A_nonOrtho = np.random.rand(1000,1000)
    A, s, vh = np.linalg.svd(A_nonOrtho) # are there faster ways?
    print(np.linalg.norm(A[:,0])) # check for normality
    phi = np.random.rand(100,1000) # vectors to use to project
    A_r = phi @ A
    #A_r = A_r.T

    # 1000 100-d random standard gaussian vectors
    B = np.random.rand(100,1000)

    # examine orthonormality of A and B
    A_err = orthotest(A_r)
    B_err = orthotest(B)

    # plot the results on separate histograms
    plot_ortho_errors(A_err,"Q_1a_histo.png")
    plot_ortho_errors(B_err,"Q_1b_histo.png")

# q6_a - normal power method (dumb one or good one? idrk)
# q6_b - modified power method

# NOTES : A is square and symmetric, so we don't need to compute B = A.T @ A
def power_method(A,k,tol=1e-6):
    '''
        Power method for computing the first left singular vector of a matrix
        PARAMETERS
        ----------
        A : np.array (m x n)
        k : int, number of iterations of the power method to run
        OUTPUTS
        -------
        u1 : first left singular vector of A
    '''

    # consider implementing a tolerance threshold to stop the power iteration at if
the results look good.

    dim = A.shape[1]
    x = np.random.rand(dim)
    #x = x/np.linalg.norm(x)
    #B = A # test using this if desired; A is symmetric
    #B = A.T @ A

    v_k = A @ x # initial iteration, should be a vector
    for i in range(0,k-1):
```

```
        v_k_1 = A @ (A.T @ v_k)
        v_k_1 = v_k_1/np.linalg.norm(v_k_1) # forgot to normalize v_k_1 earlier
        err = np.linalg.norm(np.abs(v_k_1) - np.abs(v_k))
        #print(err)
        v_k = v_k_1
        if err < tol:
            print("power method terminated at iteration " + str(i))
            break

    # line below is unnecessary as I never actually save the result.
    #v_k = B_k[:,0] # first column of B
    v_1 = v_k/np.linalg.norm(v_k) # normalize v_1

    return v_1,err # they ask for the first left singular vector...might need to
compute Av1 = s1u1 ???

def modified_power_method(A,k,tol=1e-6):
    '''
        Power method modification that compute first four left singular vectors of a
matrix
        PARAMETERS
        ----------
    '''
    # randomly select 4 vectors
    dim = A.shape[1]
    x_i = np.random.rand(dim,4)

    # initial orthonormal basis
    B_1 = A @ (A.T @ x_i)
    x = np.linalg.qr(B_1)[0]

    # subsequent iterations
    for i in range(0,k-1):
        Bx = A @ (A.T @ x)
        x_k = np.linalg.qr(Bx)[0]
        # compute the average error
        err = 0
        for j in range(0,4):
            err += np.linalg.norm(np.abs(x[:,j]) - np.abs(x_k[:,j]))
        err = np.linalg.norm(np.abs(x) - np.abs(x_k))
        x = x_k
        # not sure about this ...
        if err < 4*tol:
            print("modified power iteration terminated after " + str(i) + "
iterations")
            break

    v_4k = x
    return v_4k,err


# q7 - greyscale image question

def frobenius_norm_percent(A,A_k):
    return np.linalg.norm(A_k,ord='fro')/np.linalg.norm(A,ord='fro')
```

```python
def image_svd(im,k,case='cat'):
    '''
        Computes reduced svd of an image for low rank approximations k
    '''

    m,n = im.shape
    U,S,Vt = np.linalg.svd(im) # im is a 256 x 256 greyscale image!

    # plot the singular values to see how they are different for each case
    fig = plt.figure()
    ax = fig.add_subplot()
    ax.plot(np.arange(3,S.shape[0]),S[3:])
    ax.set_title(case+"_singular_values")
    filename = "./figs/" + case + "_singular_values.png"
    plt.savefig(filename)
    plt.close()

    # generate empty numpy array to store low rank approximated images
    images = np.zeros((m,n,len(k)))
    frobenius_norms = np.zeros(len(k))

    # generate a new image for each
    image_num = 0
    for r in k:
        S_r = np.zeros((256,256))
        for i in range(0,r):
            S_r[i,i] = S[i] # populate low-rank S with first k singular values

        images[:,:,image_num] = U @ S_r @ Vt
        # compute frobeneus norm percent
        frobenius_norms[image_num] = frobenius_norm_percent(im,images[:,:,image_num])

        image_num = image_num + 1


    return images,frobenius_norms


def q7():

    FIG_NAME = "cat_256.jpg"
    FIG_PATH = "./figs/" + FIG_NAME
    img = Image.open(FIG_PATH)

    #img.show()
    #img_pix = np.reshape(np.array(list(img.getdata())), (256,256))
    img_pix = np.asarray(img)

    # reconstruct image using 1, 4, 16, and 32 singular values
    s_vals = [1,4,16,32]
    images_a,norms_a = image_svd(img_pix,s_vals)
    print("frobeneus norms captured in cat image: " + str(norms_a))
```

```
    pil_images = []
    # plot orginal and reconstructed images
    for i in range(0,len(s_vals)):
        new_im = Image.fromarray(images_a[:,:,i])
        new_im = new_im.convert("L")
        outfile = "./figs/cat_" + str(s_vals[i]) + ".jpg"
        new_im.save(outfile)

        pil_images.append(new_im) # save the image data in case I want to use it later

    # save the images so they can be displayed in a word doc


    # compute percent of frobenius norm that is captured in each case
    # (compute ||A_k||/||A||, ||*|| denotes the frobenius norm)


    # repeat a and b but with a white noise 256 x 256 image
    white_noise = np.random.rand(256,256)*256 # maybe scale this depending how it
looks, mine is just black right now.
    # save the white noise image
    new_im = Image.fromarray(white_noise)
    new_im = new_im.convert("L")
    new_im.save("./figs/whiteNoise.jpg")

    images_c,norms_c = image_svd(white_noise,s_vals,case='white_noise')
    print("frobeneus norms captured in white noise: " + str(norms_c))

    pil_images = []
    # plot orginal and reconstructed images
    for i in range(0,len(s_vals)):
        new_im = Image.fromarray(images_c[:,:,i])
        new_im = new_im.convert("L")
        outfile = "./figs/whiteNoise_" + str(s_vals[i]) + ".jpg"
        new_im.save(outfile)

        pil_images.append(new_im)


    # compare behavior of singular values for noise image and the real image


# execution of the functions
if __name__ == "__main__":
    run_q1 = True
    run_q6 = True
    run_q7 = True

    if run_q1:
        q1()

    # plots of A and B are about the same right now and I am confused lol
    # ah...could sign be an issue?....

    if run_q6:
```

```
        ## Q6 ##
        A = np.zeros((10,10))
        b = np.arange(1,11)
        n = 10
        for i in range(0,10):
            A[i,:n-i] = b[i:]

        n_its = 100 # number of iterations for power method

        ## 6A ##
        v1,err = power_method(A,n_its)
        print("v1 = " + str(v1))
        print("power iteration error: " + str(err))

        ## 6B ##
        v4,err = modified_power_method(A,n_its)
        print("first 4 singular vectors = " + str(v4))
        print("Modified power iteration error: " + str(err))

    if run_q7:
        ## Q7 ##
        q7()

    # todo:
    # Q6 : writeup
    # Q7
    # change bins and produce nice plots
    # combine pdfs of everything and submit!!
```