```
"""
hw5_iterative_methods.py

Python code for Homework 5, AMATH 584, Fall 2020

Author: Jacqueline Nugent
Last Modified: December 7, 2020
"""
import cv2
import math
import os

import numpy as np
import matplotlib.pyplot as plt
import matplotlib.patches as mpatches

from scipy import linalg
import scipy


##### files #####
main_dir = '/Users/jmnugent/Documents/__Year_3_2020-2021/AMATH_584-
Numerical_Linear_Algebra/Homework/python/'
crop_dir = main_dir + 'CroppedYale/'
uncrop_dir = main_dir + 'yalefaces_uncropped/yalefaces/'
save_dir = main_dir + 'amath584/hw5_iterative_methods/'


##### Define functions #####
def power_iteration(A, v0=None, tol=1e-6, num_iter=None, stop=1e6):
    """
    Performs power iteration on input matrix A (m x m) until the
    gradient of the Rayleigh quotient is less than or equal to the
    tol (default: 1e-6). If num_iter is given, will perform a set
    number of iterations (and not necessarily go until the tolerance
is reached).

    Assumes A is square. The initial guess is a randomized vector of
unit length if
    not given (v0).

    Returns the list of eigenvalue and corresponding eigenvector
iterations.
    If num_iter=None, returns only the largest eigenvalue/vector and
how many iterations it took to
    reach the tolerance.

    Stops after 'stop' iterations to prevent an infinite loop.
    """
    # functions for the Rayleigh quotient and its gradient
```

```
    r = lambda x: (x.T @ A @ x) / (x.T @ x)
    gradr = lambda x: (2 / (x.T @ x)) * ((A @ x) - (r(x)*x))

    # get size m
    m = A.shape[0]

    # iterate some number of times
    if num_iter is not None:
        if type(num_iter) != int:
            raise Exception('Invalid input for num_iter! Must be an
integer # of iterations.')
        else:
            # initialize
            lk = np.zeros(num_iter, dtype=complex)
            vk = np.zeros((num_iter, m, 1), dtype=complex)

            if v0 is None:
                v0 = np.random.randn(m, 1)
            v0 = v0 / np.linalg.norm(v0)
            vk[0, :] = v0
            lk[0] = v0.T @ A @ v0

            # iterate
            for k in range(num_iter-1):
                w = A @ vk[k, :]
                vk[k+1, :] = w / np.linalg.norm(w)
                lk[k+1] = vk[k+1, :].T @ A @ vk[k+1, :]

            # lk[-1], vk[-1, :] will give you the largest
            return [lk, vk]

    # iterate until you reach some value
    else:
        # initialize
        n = 0
        if v0 is None:
            vk = v0
        vk = np.random.randn(m, 1)
        vk = vk / np.linalg.norm(vk)
        lk = vk.T @ A @ vk

        # iterate
        while np.any(np.abs(gradr(vk)) > tol):
            if n > stop:
                print('STOPPED AFTER ' + str(int(stop)) + '
ITERATIONS')
                return [lk[0][0], vk, n]
                break
            n += 1
            w = A @ vk
```

```
            vk = w / np.linalg.norm(w)
            lk = vk.T @ A @ vk

        # lk[0][0] will give you the largest
        return [lk, vk, n]


def rq_iteration(A, v0, tol=1e-6, num_iter=None, stop=1e6):
    """
    Performs Rayleigh Quotient iteration on input matrix A (m x m)
    for some initial guess of eigenvector v0 (m x 1) until the
difference
    between lambda(k-1) and lambda(k) < tol. If num_iter is given,
will
    perform a set number of iterations (and not necessarily go until
the
    tolerance is reached).

    Assumes A is square.

    Returns the list of eigenvalue and corresponding eigenvector
iterations.
    If num_iter=None, returns only the largest eigenvalue/vector and
how many iterations it took to
    reach the tolerance.

    Stops after 'stop' iterations to prevent an infinite loop.
    """
    # functions for the Rayleigh quotient and its gradient
    r = lambda x: (x.T @ A @ x) / (x.T @ x)
    gradr = lambda x: (2 / (x.T @ x)) * ((A @ x) - (r(x)*x))

    # get size m
    m = A.shape[0]

    # initialize arrays
    eigs = np.zeros(m, dtype=complex)
    vecs = np.zeros((m, m), dtype=complex)

    # iterate some number of times
    if num_iter is not None:
        if type(num_iter) != int:
            raise Exception('Invalid input for num_iter! Must be an
integer # of iterations.')
        else:
            # initialize
            lk = np.zeros(num_iter, dtype=complex)
            vk = np.zeros((num_iter, m, 1), dtype=complex)
            vk[0, :] = v0
            lk[0] = v0.T @ A @ v0
```

```
            # iterate
            for k in range(num_iter-1):
                try:
                    w = np.linalg.solve((A - lk[k]*np.eye(m)),
vk[k, :])
                except:
                    return [lk[:k], vk[:k]]
                vk[k+1, :] = w / np.linalg.norm(w)
                lk[k+1] = vk[k+1, :].T @ A @ vk[k+1, :]

            # lk[-1], vk[-1, :] will give you the largest
            return [lk, vk]

    # iterate until the difference is very small
    else:
        # initialize
        l0 = v0.T @ A @ v0

        # first iteration
        n = 1
        w = np.linalg.solve((A - l0*np.eye(m)), v0)
        vk = w / np.linalg.norm(w)
        lk = vk.T @ A @ vk

        # iterate until lambdas are equal within the tolerance
        lk_prev = l0
        while np.abs(lk - lk_prev) > tol:
            n += 1
            if n > stop:
                print('STOPPED AFTER ' + str(int(stop)) + '
ITERATIONS')
                return [lk[0][0], vk, n]
                break
            lk_prev = lk
            try:
                w = np.linalg.solve((A - lk*np.eye(m)), vk)
            except:
                return [lk[0][0], vk, n]
                break
            vk = w / np.linalg.norm(w)
            lk = vk.T @ A @ vk

        return [lk[0][0], vk, n]


def rand_unitvec(m, c=False):
    """Generate a random mx1 vector of unit length. Complex if c=True.
    """
    if c:
```

```
        v = np.random.randn(m, 1) + np.random.randn(m, 1)*1j
    else:
        v = np.random.randn(m, 1)
    v1 = v / np.linalg.norm(v)

    return v1


########## problem 1 ##########

#### 1. (a) ####
# build the matrix
m = 10
A_nonsymm = np.random.randn(m, m)
A = np.tril(A_nonsymm) + np.tril(A_nonsymm, -1).T

# check symmetric
print(np.all(A == A.T))

# ground truth
eigvals, eigvecs = scipy.linalg.eig(A)
print(eigvals)

#### 1. (b) ####
# function to find gradient of the rayleigh quotient
r = lambda x: (x.T @ A @ x) / (x.T @ x)
gradr = lambda x: (2 / (x.T @ x)) * ((A @ x) - (r(x)*x))

# to check for accuracy, this should be true
tol = 1e-6
is_close = lambda x: np.all(np.abs(gradr(x)) < tol)


## make plots ##
# try 300 iterations
k = 300
l_list, v_list = power_iteration(A, num_iter=k)
print(l_list[-1])
print(is_close(v_list[-1, :]))
print(k, 'iterations')

# for comparison, see how many iterations it takes to converge
l_conv, v_conv, n_conv = power_iteration(A, tol=tol)
print('\n', l_conv)
print(is_close(v_conv))
print(n_conv, 'iterations')

plt.plot(np.arange(k), [x-eigvals[0] for x in l_list])
plt.title('Accuracy vs. Iterations')
plt.ylabel('Estimate - Largest Eigenvalue')
```

```
plt.xlabel('Iteration ({} total)'.format(k))
plt.xlim((0, k))
plt.savefig(save_dir + '1b_accuracy_vs_iterations.png', dpi=300,
bbox_inches='tight')
plt.show()


#### 1. (c) ####
def plot_accuracy(l_list, k, n, eigvals=eigvals, save_dir=save_dir,
save=False):
    """Make the plot for each eigenvector
    """
    truth = eigvals[np.abs(l_list[-1]-eigvals).argmin()]

    plt.plot(np.arange(len(l_list)), [x-truth for x in l_list])

    plt.title('Accuracy vs. Iterations\n(eigenvalue =
{})'.format(truth.real))
    plt.ylabel('Estimate - Actual')
    plt.xlabel('Iteration ({} total)'.format(k))
    plt.xlim((0, k))

    if save:
        plt.savefig(save_dir +
'1b_accuracy_vs_iterations_eigval_{}.png'.format(n), dpi=300,
bbox_inches='tight')

    plt.show()


k_rq = 10

# initialize list of eigenvalues with an array because it
# gets returned as a float
eigs_rq = np.zeros(1)

# perform one iteration to find an eigenvalue/eigenvector pair
eigs_rq0, vecs_rq = rq_iteration(A, rand_unitvec(m), num_iter=k_rq)
plot_accuracy(eigs_rq0, k=k_rq, n=0, save=True)
eigs_rq[0] = eigs_rq0[-1]

# repeat the iteration until you find all 10 unique ones;
# check for uniqueness to 6 decimal points (because tol=1e-6)
while len(eigs_rq) < 10:
    eig, vecs = rq_iteration(A, rand_unitvec(m), num_iter=k_rq)
    if np.round(eig, 6) not in np.round(eigs_rq, 6):
        plot_accuracy(eig, k=k_rq, n=len(eigs_rq), save=True)
        eigs_rq = np.append(eigs_rq, eig[-1])

# check that you got all 10
```

```
print('From Rayleigh Quotient iteration:')
# print(sorted(eigs_rq, key=abs)[::-1])
for x in eigs_rq: print(x)

print('\nGround truth:')
for x in eigvals: print(x)


#### 1. (d) ####
# build the matrix
m = 10

# first guess
A_ns = np.random.randn(m, m)
eigvals_ns, eigvecs_ns = scipy.linalg.eig(A_ns)

# if the convergence ratio is very large, try again
while np.abs(eigvals_ns[1]/eigvals_ns[0]) > 0.98:
    A_ns = np.random.randn(m, m)
    eigvals_ns, eigvecs_ns = scipy.linalg.eig(A_ns)

# ground truth
print(eigvals_ns[0])
print(eigvals_ns)

### (b), nonsymmetric ###
# function to find gradient of the rayleigh quotient
r = lambda x: (x.T @ A_ns @ x) / (x.T @ x)
gradr = lambda x: (2 / (x.T @ x)) * ((A_ns @ x) - (r(x)*x))

# to check for accuracy, this should be true
tol = 1e-6
is_close = lambda x: np.all(np.abs(gradr(x)) < tol)

# initial complex
v0c = rand_unitvec(m, c=True)

# k2 iterations
k2 = 150
lpi_ns, vpi_ns = power_iteration(A_ns, v0=v0c, num_iter=k2)
print(k2, 'iterations')
print(lpi_ns[-1])
print(is_close(vpi_ns[-1, :]), '\n')

# unit it converges
lpi_ns2, vpi_ns2, npi_ns2 = power_iteration(A_ns, v0=v0c)
print(npi_ns2, 'iterations')
print(lpi_ns2)
```

```
plt.plot(np.arange(k2), [np.abs(x)-np.abs(eigvals_ns[0]) for x in
lpi_ns])

plt.xscale('log')
plt.title('Accuracy vs. Iterations')
plt.ylabel('|Esimate| - |Largest Eigenvalue|')
plt.xlabel('Iteration ({} total)'.format(k2))
plt.xlim((1, k2))
plt.savefig(save_dir + '1d_b_accuracy_vs_iterations.png', dpi=300,
bbox_inches='tight')
plt.show()

### (c), nonsymmetric ###
def plot_accuracy_ns(l_list, k, n, eigvals=eigvals_ns,
save_dir=save_dir, save=False):
    """Make the plot for each eigenvector
    """
    truth = eigvals[np.abs(np.abs(l_list[-1])-[np.abs(x) for x in
eigvals]).argmin()]

    plt.plot(np.arange(len(l_list)), [np.abs(x)-np.abs(truth) for x in
l_list])

    plt.title('Accuracy vs. Iterations')
    plt.ylabel('|Estimate| - |Actual|')
    plt.xlabel('Iteration ({} total)'.format(k))
    plt.xscale('log')
    plt.xlim((1, k))

    if save:
        plt.savefig(save_dir +
'1dc_accuracy_vs_iterations_eigval_{}.png'.format(n), dpi=300,
bbox_inches='tight')

    plt.show()


k_rq_ns = 150

# initialize list of eigenvalues
eigs_rq_ns = np.zeros(1, dtype=complex)

# perform one iteration to find an eigenvalue/eigenvector pair
eigs_rq_ns0, vecs_rq = rq_iteration(A_ns, rand_unitvec(m, c=True),
num_iter=k_rq_ns)
plot_accuracy_ns(eigs_rq_ns0, k=k_rq_ns, n=0, save=False)
eigs_rq_ns[0] = eigs_rq_ns0[-1]

# repeat the iteration until you find all 10 unique ones;
# check for uniqueness to 6 decimal points (becA_nsuse tol=1e-6)
```

```
while len(eigs_rq_ns) < 10:
    eig, vecs = rq_iteration(A_ns, rand_unitvec(m, c=True),
num_iter=k_rq_ns)
    eig_rounded = np.round(eig[-1].real, 6) + np.round(eig[-1].imag,
6)*1j
    eigs_rq_rounded = [np.round(x.real, 6) + np.round(x.imag, 6)*1j
for x in eigs_rq_ns]
    if eig_rounded not in eigs_rq_rounded:
        plot_accuracy_ns(eig, k=k_rq_ns, n=len(eigs_rq_ns),
save=False)
        eigs_rq_ns = np.append(eigs_rq_ns, eig[-1])

# check that you got all 10:
print('From Rayleigh Quotient iteration:')
for x in eigs_rq_ns:
    print(x)

print('\nGround truth:')
for x in eigvals_ns:
    print(x)


########## problem 2 ##########

#### read in data ####
# get a list of paths to each subfolder in CroppedYale
paths = [crop_dir + dirname for dirname in os.listdir(crop_dir)
         if os.path.isdir(os.path.join(crop_dir, dirname))]

# initialize list to hold the averaged data matrices for each image
n_img = len(paths)
cropped_pics = [[]]*n_img
cropped_avgs = [[]]*n_img

for i in range(n_img):
    # get the list of file names within the subfolder for that image
    subfolder = paths[i] + '/'
    imagenames = [subfolder + f for f in os.listdir(subfolder)
                  if os.path.isfile(os.path.join(subfolder, f))]

    # make one list containing the data matrices for each (grayscale)
image
    cropped_pics[i] = [cv2.cvtColor(cv2.imread(x), cv2.COLOR_BGR2GRAY)
for x in imagenames]

    # averaged the data matrix for this image and add to the list
    cropped_avgs[i] = np.mean(cropped_pics[i], axis=0)

# stack so each image is one column in the data matrix
all_pics_c = [cropped_pics[i][j].flatten() for i in
```

```
range(len(cropped_pics))
              for j in range(len(cropped_pics[i]))]
A_yf = np.transpose(np.asarray(all_pics_c))

print(A_yf.shape)

# get the square correlation matrix
C_yf = A_yf.T @ A_yf # is this right? gives 2432 x 2432
print(C_yf.shape)


#### 2 (a) ####
# iterate on the correlation matrix
l_data, v_data, n_data = power_iteration(C_yf)
print('took', n_data, 'iterations')

# function to find gradient of the rayleigh quotient
r = lambda x: (x.T @ C_yf @ x) / (x.T @ x)
gradr = lambda x: (2 / (x.T @ x)) * ((C_yf @ x) - (r(x)*x))

# to check for accuracy, this should be true
tol = 1e-6
is_close = lambda x: np.all(np.abs(gradr(x)) < tol)
print(l_data)
print(gradr(v_data))
print(is_close(v_data))

# perform (economy) SVD
[Uh, Sh, VTh] = np.linalg.svd(C_yf, full_matrices=False)

# check that the dimensions are correct
print(C_yf.shape, Uh.shape, Sh.shape, VTh.shape)

# compare values
print(Sh[0], l_data)
print(Sh[0] - l_data)

# plot eig spectrum
n_eigs = 20
plt.scatter(1, l_data, color='C1', marker='*', s=150, label='Power
Iteration (largest)')
plt.plot(np.arange(1, n_eigs+1), Sh[:n_eigs], marker='.',color='C0',
label='SVD (first {})'.format(n_eigs))
plt.title('Eigenvalue Spectrum')
plt.legend()
plt.yscale('log')
xmax = n_eigs + 0.5
plt.xlim((0.5, xmax))
plt.ylim((5e3, 1e6))
plt.savefig(save_dir + '2a_eigenvalue_spectra.png', dpi=300,
```

```
bbox_inches='tight')
plt.show()

# reshape first to match the shape of Uh[:, 0]
v_data_rs = np.reshape(v_data, (len(v_data)))

# This is true if the power iteration vector & SVD vector
# have opposite signs
print(np.all(np.sign(v_data_rs) == -np.sign(Uh[:, 0])))

# find the difference in magnitude at each row
vdiff = np.abs(v_data_rs) - np.abs(Uh[:, 0])

# plot the difference
fig, ax = plt.subplots(figsize=(10, 4))
ax.bar(np.arange(1, len(vdiff)+1), vdiff)
ax.set_title('Difference in largest eigenvector magnitude\n(Power
Iteration - SVD)')
ax.set_ylim(-.5e-12, .5e-12)
ax.set_xlim(-25, len(vdiff)+25)
ax.set_xlabel('row index')
plt.savefig(save_dir + '2a_diff_in_eigenvector_mag.png', dpi=300,
bbox_inches='tight')
plt.show()


#### 2 (b) ####
def randomized_sampling(A, k, check_sizes=False):
    """
    Perform randomized sampling on matrix A. Input
    number of random projections (k)

    Returns Uapprox, Sapprox, and VTapprox from the
    SVD on the randomly sampled matrix.
    """
    # --- stage A ---
    m, n = A.shape

    # random projections
    omega = np.random.randn(n, k)
    Y = A @ omega

    # QR decomp
    Q, R = np.linalg.qr(Y, mode='reduced')

    # --- stage B ---
    B = Q.T @ A_yf
    Ut, St, VTt = np.linalg.svd(B, full_matrices=False)

    # project back out
```

```
    Uapprox = Q @ Ut

    # --- check sizes if requested ---
    if check_sizes:
        print('m, n, k:', m, n, k)
        print('Y:', Y.shape)
        print('Q:', Q.shape)
        print('B:', B.shape)
        print('U:', Uapprox.shape)
        print('S:', St.shape)

    return [Uapprox, St, VTt]


# ground truth for yale faces
U, S, VT = np.linalg.svd(A_yf, full_matrices=False)


#### 2 (c) ####
# pick some ks
k_list = [5, 10, 50, 100, 1000]
Ua = [[]]*len(k_list)
Sa = [[]]*len(k_list)
VTa= [[]]*len(k_list)

# loop through each number of iterations
for i in range(len(k_list)):
    Ua[i], Sa[i], VTa[i] = randomized_sampling(A_yf, k=k_list[i])
    print('k = {} done'.format(k_list[i]))


## plot: eigenvalue spectra ##
fig = plt.figure(figsize=(8, 8))

ax1 = fig.add_subplot(211)
ax2 = fig.add_subplot(212)

# truth
ax1.scatter(np.arange(1, len(S)+1), S, color='k', s=10, label='ground
truth')
ax1.plot(np.arange(1, len(S)+1), S, color='k')
ax2.scatter(np.arange(1, len(S)+1), S/np.sum(S), color='k', s=10,
label='ground truth')
ax2.plot(np.arange(1, len(S)+1), S/np.sum(S), color='k')

# approximations
for i in range(len(k_list)):
    ax1.scatter(np.arange(1, len(Sa[i])+1), Sa[i], s=10, label='k =
{}'.format(k_list[i]))
    ax1.plot(np.arange(1, len(Sa[i])+1), Sa[i], alpha=0.6)
```

```
    ax2.scatter(np.arange(1, len(Sa[i])+1), Sa[i]/np.sum(Sa[i]), s=10,
label='k = {}'.format(k_list[i]))
    ax2.plot(np.arange(1, len(Sa[i])+1), Sa[i]/np.sum(Sa[i]),
alpha=0.6)

ax1.set_yscale('log')
ax1.legend()
ax1.set_xlim((-.5, 50))
ax1.set_title('Eigenvalue Spectra\n(First 50 Eigenvalues)')
ax1.set_ylim((5e3, 1e6))

ax2.legend()
ax2.set_xlim((-.5, 50))
ax2.set_title('% Variance')

# plt.savefig(save_dir + '2c_eigenvalue_spectra--no_line.png',
dpi=300, bbox_inches='tight')
plt.savefig(save_dir + '2c_eigenvalue_spectra--line.png', dpi=300,
bbox_inches='tight')

plt.show()


## reconstructions ##
# from the random sampling
A_rs_list = [[]]*len(k_list)
for i in range(len(k_list)):
    A_rs_list[i] = Ua[i] @ np.diag(Sa[i]) @ VTa[i]

# from the "true" SVD
A_rs_svd = U @ np.diag(S) @ VT

x, y = cropped_pics[0][0].shape

mode0_svd = np.reshape(A_rs_svd[:, 0], (x, y))
mode1_svd = np.reshape(A_rs_svd[:, 1], (x, y))
mode2_svd = np.reshape(A_rs_svd[:, 2], (x, y))

mode0_approx_list = [[]]*len(k_list)
mode1_approx_list = [[]]*len(k_list)
mode2_approx_list = [[]]*len(k_list)

for i in range(len(k_list)):
    mode0_approx_list[i] = np.reshape(A_rs_list[i][:, 0], (x, y))
    mode1_approx_list[i] = np.reshape(A_rs_list[i][:, 1], (x, y))
    mode2_approx_list[i] = np.reshape(A_rs_list[i][:, 2], (x, y))

## PLOT (note: lots of hardcoding here ##
fig, axes = plt.subplots(3, 6, figsize=(15, 9))
plt.subplots_adjust(hspace=0.3)
```

```
for ax in axes.flatten():
    ax.axis('off')

# first mode
axes[0, 0].pcolormesh(np.flip(mode0_approx_list[0]), cmap='gray')
axes[0, 0].set_title('Randomized, k=5\n(first mode)')

axes[0, 1].pcolormesh(np.flip(mode0_approx_list[1]), cmap='gray')
axes[0, 1].set_title('Randomized, k=10\n(first mode)')

axes[0, 2].pcolormesh(np.flip(mode0_approx_list[2]), cmap='gray')
axes[0, 2].set_title('Randomized, k=50\n(first mode)')

axes[0, 3].pcolormesh(np.flip(mode0_approx_list[3]), cmap='gray')
axes[0, 3].set_title('Randomized, k=100\n(first mode)')

axes[0, 4].pcolormesh(np.flip(mode0_approx_list[4]), cmap='gray')
axes[0, 4].set_title('Randomized, k=1000\n(first mode)')

axes[0, 5].pcolormesh(np.flip(mode0_svd), cmap='gray')
axes[0, 5].set_title('Truth\n(first mode)')

# second mode
axes[1, 0].pcolormesh(np.flip(mode1_approx_list[0]), cmap='gray')
axes[1, 0].set_title('Randomized, k=5\n(second mode)')

axes[1, 1].pcolormesh(np.flip(mode1_approx_list[1]), cmap='gray')
axes[1, 1].set_title('Randomized, k=10\n(second mode)')

axes[1, 2].pcolormesh(np.flip(mode1_approx_list[2]), cmap='gray')
axes[1, 2].set_title('Randomized, k=50\n(second mode)')

axes[1, 3].pcolormesh(np.flip(mode1_approx_list[3]), cmap='gray')
axes[1, 3].set_title('Randomized, k=100\n(second mode)')

axes[1, 4].pcolormesh(np.flip(mode1_approx_list[4]), cmap='gray')
axes[1, 4].set_title('Randomized, k=1000\n(second mode)')

axes[1, 5].pcolormesh(np.flip(mode1_svd), cmap='gray')
axes[1, 5].set_title('Truth\n(second mode)')

# third mode
axes[2, 0].pcolormesh(np.flip(mode2_approx_list[0]), cmap='gray')
axes[2, 0].set_title('Randomized, k=5\n(third mode)')

axes[2, 1].pcolormesh(np.flip(mode2_approx_list[1]), cmap='gray')
axes[2, 1].set_title('Randomized, k=10\n(third mode)')

axes[2, 2].pcolormesh(np.flip(mode2_approx_list[2]), cmap='gray')
```

```
axes[2, 2].set_title('Randomized, k=50\n(third mode)')

axes[2, 3].pcolormesh(np.flip(mode2_approx_list[3]), cmap='gray')
axes[2, 3].set_title('Randomized, k=100\n(third mode)')

axes[2, 4].pcolormesh(np.flip(mode2_approx_list[4]), cmap='gray')
axes[2, 4].set_title('Randomized, k=1000\n(third mode)')

axes[2, 5].pcolormesh(np.flip(mode2_svd), cmap='gray')
axes[2, 5].set_title('Truth\n(third mode)')

plt.savefig(save_dir + '2c_mode_comparison.png', dpi=300,
bbox_inches='tight')

plt.show()
```