PR2_sheet1

May 6, 2021

1 Exercise 1 - Eigenvalues, Eigenvectors, and Prototypes

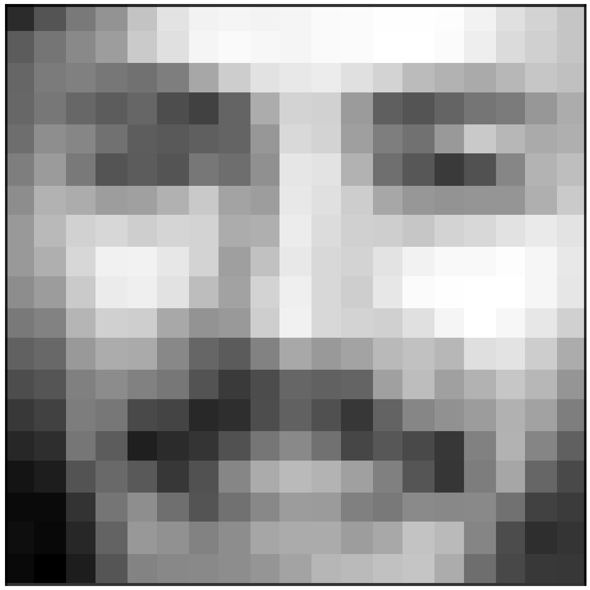
In [1]:
```
import numpy as np
import numpy.linalg as la
import matplotlib.pyplot as plt


from sklearn.preprocessing import normalize

import timeit
import functools

from scipy.spatial.distance import cdist
import numpy.random as rnd
import imageio as imageio
```

1.1 Task 1.1

In [3]:
```
### load the data as a float-vector
matX = np.load('faceMatrix.npy').astype('float')
### interpret the float-vector as a mxn matrix and print these dimensions
m, n = matX.shape
print(m,n)
```

```
361 2429
```

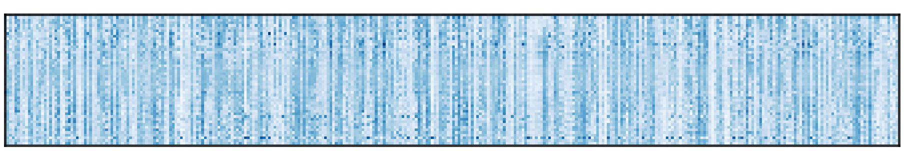
In [4]:
```
### re-interpret column 15 of this huge matrix 'matX' (lenght 361=19x19),
### as a 19x19 matrix
vecX = matX[:,14].reshape(19,19)
def plot_matrix(M):
    plt.imshow(vecX, cmap='gray')
    plt.xticks([]) ### dont plot the x/y coordinates 1..18 (instead use [])
    plt.yticks([])
    plt.show

plot_matrix(vecX)
```



1. Normalize the matrix

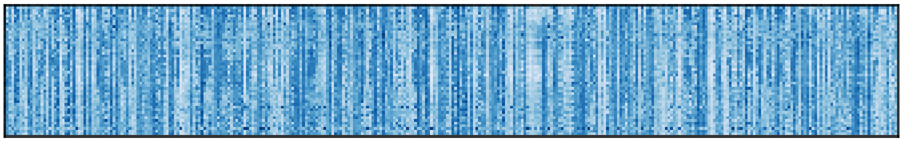
In [5]:
```
def show_mat_blue(M, n=None):
    fig, ax = plt.subplots()
    if n is None:
        ax.matshow(M, cmap=plt.cm.Blues)
    else:
        ax.matshow(M.reshape(n, n), cmap=plt.cm.Blues)
    plt.xticks([]), plt.yticks([])
    plt.show()
```

In [6]:
```
### Scale input vectors individually to unit norm
method1_normX = normalize(matX, axis=1, norm='l1')

show_mat_blue(method1_normX)
```



In [7]:
```
### Normalize data to zero mean
method2_normX = matX - np.mean(matX, axis=1).reshape(m,1)
```

```
show_mat_blue(method2_normX)
matX = method2_normX
```



```
In [8]: np.array_equal(method1_normX, method2_normX)
```

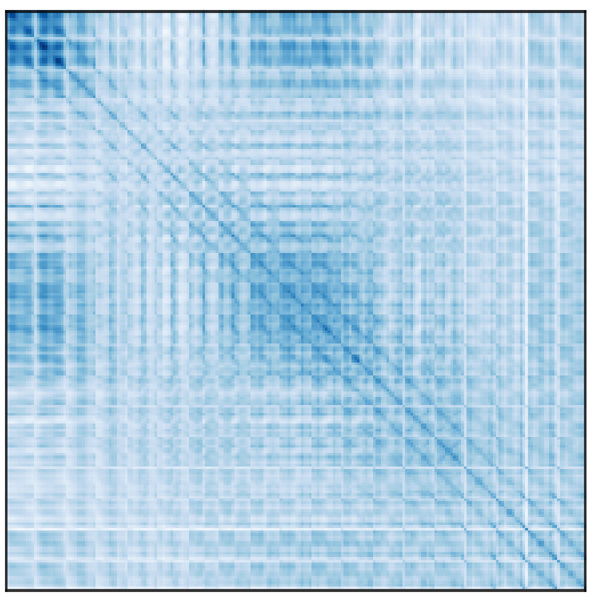
```
Out[8]: False
```

2. Compute XX^T

There are different methods for matrix multiplication. For example: - `matC_1 = np.matmul(matX, matX.transpose())` - `matC_2 = np.dot(matX, matX.T)` - `matC_3 = matX.dot(matX.T)`

But as suggested in the lecture, use the `@` operator:

```
In [9]: matC = matX @ matX.T
        show_mat_blue(matC)
```



3. Compute $C = U\Lambda U^T$ **- la.eig**

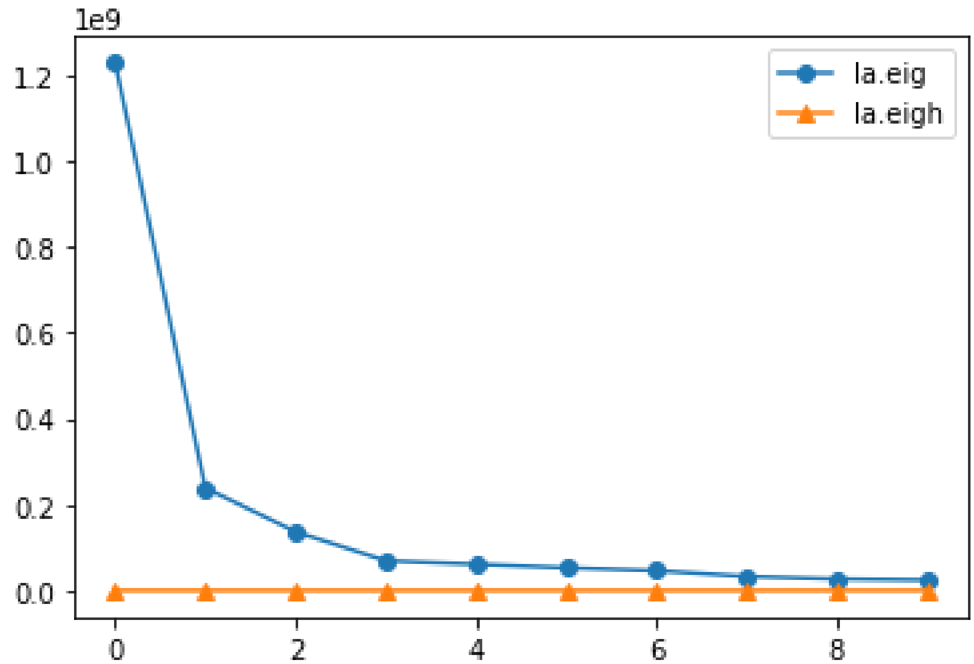
```
In [10]: eigenval_eig, eigenvec_eig = la.eig(matC)
```

4. Compute $C = U\Lambda U^T$ - la.eigh

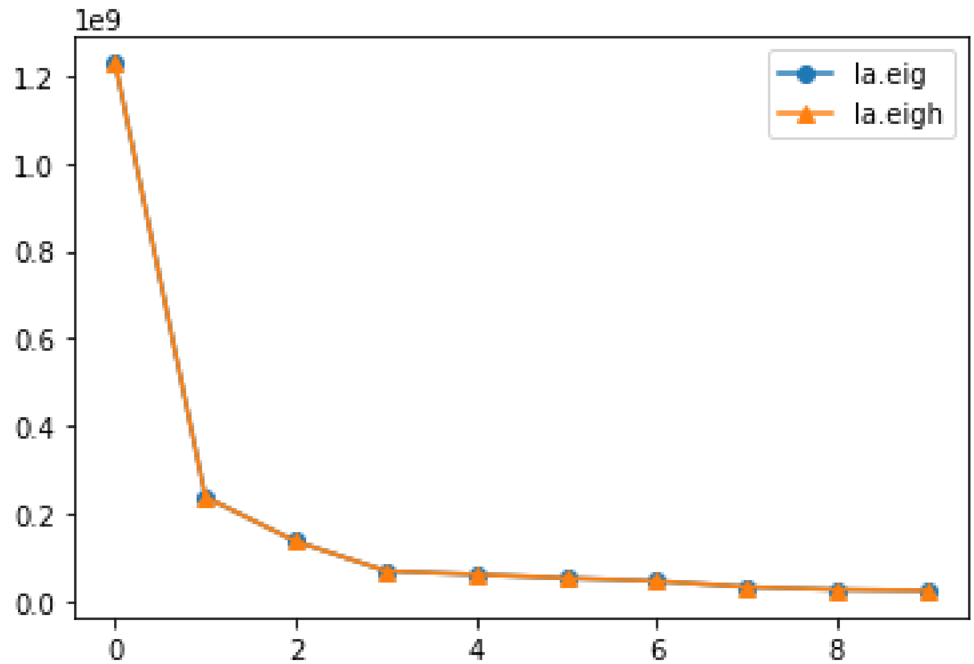
```
In [11]: eigenval_eigh, eigenvec_eigh = la.eigh(matC)
```

5. Compare la.eig and la.eigh

```
In [12]: plt.plot(eigenval_eig[:10], marker='o', label='la.eig')
         plt.plot(eigenval_eigh[:10], marker='^', label='la.eigh')
         plt.legend()
         plt.show()
```




```
In [13]: ### reverse the order of eigenvalues computed using la.eigh
         eigenval_eigh = eigenval_eigh[::-1]

         plt.plot(eigenval_eig[:10], marker='o', label='la.eig')
         plt.plot(eigenval_eigh[:10], marker='^', label='la.eigh')
         plt.legend()
         plt.show()
```



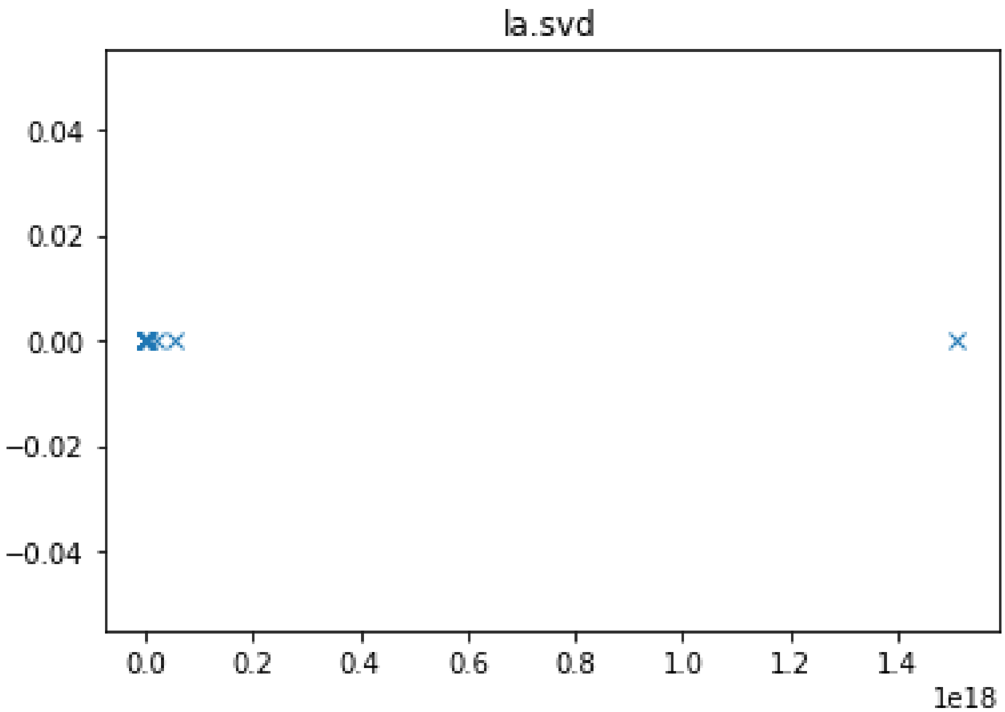
- `la.eig` does not give any guarantees on the order of the eigenvalues it returns
  - From documentation: "Compute the eigenvalues and right eigenvectors of *a square array*."
- `la.eigh`
  - From documentation: "Return the eigenvalues and eigenvectors of *a complex Hermitian (conjugate symmetric) or a real symmetric matrix*"
  - returns eigenvalues in *ascending* order
- both methods return normalized eigenvectors

6. Compute $X = U\Sigma V^T$ - la.svd

```
In [14]: np.seterr(all="print")
         unitaries_U_svd, singularvals_svd, unitaries_Vt_svd = la.svd(matC)
         eigenval_svd = singularvals_svd**2
```

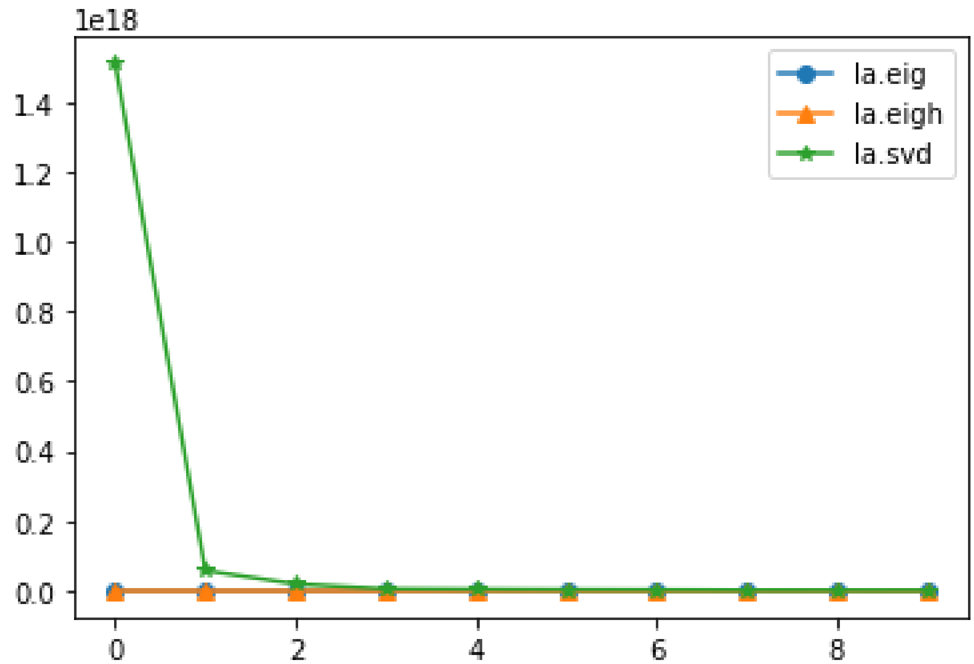
7. Square Eigenvalues

```
In [15]: fig, ax = plt.subplots()
         # ax.matshow(eigenval_svd.reshape(19,19), cmap=plt.cm.Blues)
         # plt.xticks([]), plt.yticks([])
         plt.title('la.svd')
         plt.plot(eigenval_svd, np.zeros_like(eigenval_svd), 'x')

         plt.show()
```



8. Plot spectra

```
In [16]: ### plot the eigenvalues
         plt.plot(eigenval_eig[:10], marker='o', label='la.eig')
         plt.plot(eigenval_eigh[:10], marker='^', label='la.eigh')
         plt.plot(eigenval_svd[:10], marker='*', label='la.svd')
         plt.legend()
         plt.show()

         ### Note the scale of the y-axis: 1e18 = 10**18
         ### Above it was: 1e9 = 10**9
```



1.2 Task 1.2

In [17]:
```
### Define functions for time measurements
### For fairness: include the necessary computation of C
def PCA_eig(X):
    C = X @ X.T
    l, U = la.eig(C)

def PCA_eigh(X):
    C = X @ X.T
    l, U = la.eigh(C)

def SVD(X):
    U, s, Vt = la.svd(X)
```

In [18]:
```
matX = np.load('Exercise1/Data/faceMatrix.npy').astype('float')
m, n = matX.shape
matX = matX - np.mean(matX, axis=1).reshape(m,1)
```

```
---------------------------------------------------------------------------

FileNotFoundError                         Traceback (most recent call last)
```

```
    <ipython-input-18-32451c31b09d> in <module>
----> 1 matX = np.load('Exercise1/Data/faceMatrix.npy').astype('float')
      2 m, n = matX.shape
      3 matX = matX - np.mean(matX, axis=1).reshape(m,1)


    ~/anaconda3/lib/python3.7/site-packages/numpy/lib/npyio.py in load(file, mmap_mode, al
    414             own_fid = False
    415         else:
--> 416             fid = stack.enter_context(open(os_fspath(file), "rb"))
    417             own_fid = True
    418


    FileNotFoundError: [Errno 2] No such file or directory: 'Exercise1/Data/faceMatrix.npy
```

```
In [19]: # according to: https://www.researchgate.net/publication/329449786_NumPy_SciPy_Recipe
         ts = timeit.Timer(functools.partial(PCA_eig, matX)).repeat(3, 100)
         print (min(ts) / 100)

         ts = timeit.Timer(functools.partial(PCA_eigh, matX)).repeat(3, 100)
         print (min(ts) / 100)

         ts = timeit.Timer(functools.partial(SVD, matX)).repeat(3, 100)
         print (min(ts) / 100)
```

```
0.07110150369000622
0.015912863520206884
0.9599347343301633
```

The timeit-number (here '100') is the number of executions of the main statement. = Return time it takes to execute the fct 100 times [sec, float] The repeat-number (here '3') is the number of timeit-executions. From python.org (https://docs.python.org/3/library/timeit.html): Do not calculate mean and std.dev. Use the lowest value = lower bound for how fast your machine can run the given code snippet Higher values are typically not caused by variability in Python's speed, but by other processes interfering with your timing accuracy.

Here are runtimes (in seconds) measured on an Intel i5 (2.9 GHz) (outside a notebook)

- la.eig : 0.03231685656995978
- la.eigh : 0.00624352695012930
- la.svd : 0.14175234847993123

Conclusion: la.eigh is by far the fastest!

1.3 Task 1.3

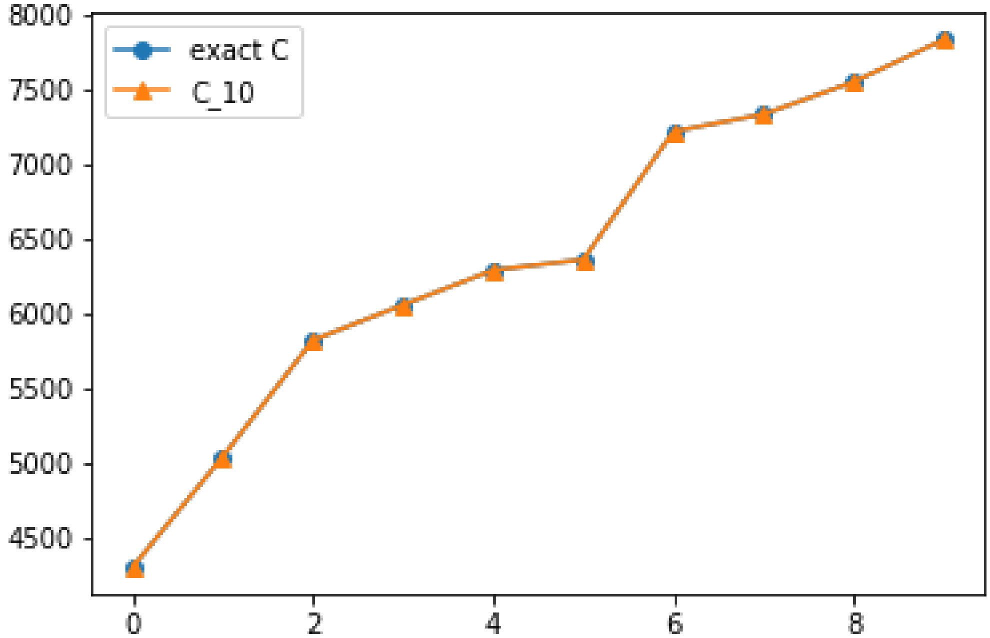
In [20]:
```
def qrAlgorithm(X, tmax=10):
    C = X @ X.T

    for _ in range(tmax):
        Q, R = la.qr(C)
        C = R @ Q

    return C
```

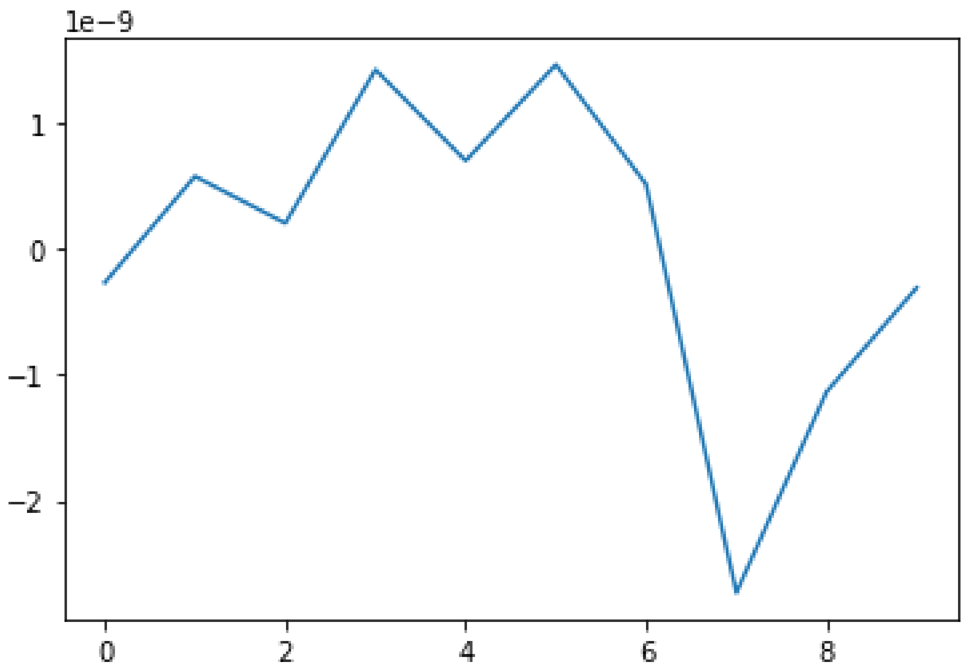
Run the QR algorithm and print diagonal of resulting matrix *C*

In [23]:
```
matX = np.load('faceMatrix.npy').astype('float')
m, n = matX.shape
matX = matX - np.mean(matX, axis=1).reshape(m,1)
```

In [24]:
```
matC = qrAlgorithm(matX)
# print (np.diag(matC))
```

In [25]:
```
C = matX @ matX.T
eigenvals, U = la.eigh(C)
eigenvals_10, U_10 = la.eigh(matC)

difference = eigenvals - eigenvals_10

plt.plot(eigenvals[:10], marker='o', label='exact C')
plt.plot(eigenvals_10[:10], marker='^', label='C_10')
plt.legend()
plt.show()
plt.plot(difference[:10])
plt.show()
```





In [26]:
```
### Compating the diagonal entries
plt.plot(np.diag(C)[:100], marker='o', label='exact C')
plt.plot(np.diag(matC)[:100], marker='^', label='C_10')
```