exercise 1

eigenvalues, eigenvectors, and prototypes

solutions due

until **April 25, 2021** at **23:59** via **ecampus**

students handing in this solution set

last name	first name	student ID	enrolled with
Nikolskyy	Oleksander	TODO	TODO
Schier	Marie	TODO	TODO
Doll	Niclas	3075509	Uni Bonn
Safavi	Arash	TODO	TODO
Wani	Mohamad Saalim	TODO	TODO
Bonani	Mayara Everlim	TODO	TODO

general remarks

As you know, your instructor is an avid proponent of open science and education. Therefore, **MATLAB implementations will not be accepted** in this course.

The goal of this exercise is to get used to scientific Python. There are numerous resources on the web related to Python programming. Numpy and Scipy are well documented and Matplotlib, too, comes with numerous tutorials. Play with the code that is provided. Most of the above tasks are trivial to solve, just look around for ideas as to how it can be done.

Remember that you have to achieve at least 50% of the points of the exercises to be eligible to the written exam at the end of the semester. Your grades (and credits) for this course will be decided based on the exam only, but –once again– you have to succeed in the exercises to get there.

Your solutions have to be *satisfactory* to count as a success. Your code and results will be checked and need to be convincing.

If your solutions meets the above requirements and you can demonstrate that they work in practice, it is a *satisfactory* solution.

A *very good* solution (one that is rewarded full points) requires additional efforts especially w.r.t. to readability of your code. If your code is neither commented nor well structured, your solution is not good! The same holds for your discussion of your results: these should be concise and convincing and demonstrate that you understood what the respective task was all about. Striving for very good solutions should always be your goal!

practical advice

The problem specifications you'll find below assume that you use python / numpy / scipy for your implementations. They also assume that you have imported the following

```
import numpy as np
import numpy.linalg as la
import matplotlib.pyplot as plt
```

task 1.1 [10 points]

computing spectral- and singular value decompositions

In the `Data` folder for this exercise, you will find the file

`faceMatrix.npy`

which contains a data matrix $\boldsymbol{X} \in \mathbb{R}^{361 \times 2429}$ whose columns $\boldsymbol{x}_i$ represent tiny face images of size 19×19 pixels. To read this matrix into memory and check its size, you may use

```
matX = np.load('faceMatrix.npy').astype('float')
m, n = matX.shape
print (m, n)
```

To have a look at one of the images it contains, say $\boldsymbol{x}_{15}$, you may use this

```
vecX = matX[:,14].reshape(19,19)
plt.imshow(vecX, cmap='gray')
plt.xticks([])
plt.yticks([])
plt.show()
```

Having read matrix $\boldsymbol{X}$ into memory, here is what you are supposed to do

1. normalize $\boldsymbol{X}$ such that its column mean is $\boldsymbol{0}$

   ```
   # normalize the data matrix such that the column mean is zero
   # by simply subtracting the mean of each column
   X -= X.mean(axis=0, keepdims=True)
   ```

2. compute $\boldsymbol{C} = \boldsymbol{X}\boldsymbol{X}^\intercal$
   NOTE: we deliberately omit the scaling factor of $\frac{1}{n}$

3. compute the spectral decomposition $\boldsymbol{C} = \boldsymbol{U}\boldsymbol{\Lambda}\boldsymbol{U}^\intercal$ using *la.eig*

4. compute the spectral decomposition $\boldsymbol{C} = \boldsymbol{U}\boldsymbol{\Lambda}\boldsymbol{U}^\intercal$ using *la.eigh*

5. compare the eigenvalues you obtain from these two approaches; what do you observe ?

   Up to permutation the two spectra are exaclty the same (see figure 1). The reason for the different permutations is simply that *la.eigh*

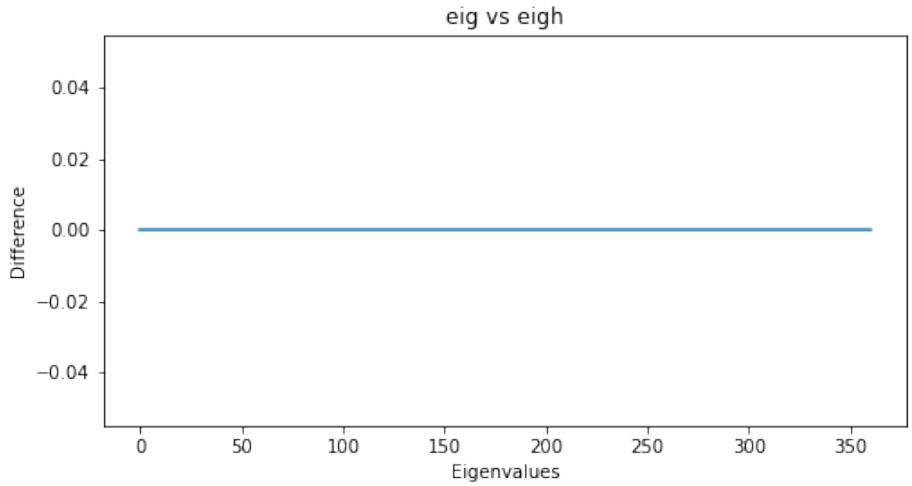


Figure 1: Difference of the sorted eigenvalues computed by the two methods. The x-axis lists all the eigenvalues of the matrix C and the y-axis shows the difference of the corresponding values. Really not much to see here.

guarantees that the eigenvalues are sorted whereas `la.eig` doesn't. Also it might happen that some of the elements have flipped signs but we didn't observe this here.

6. compute the singular value decomposition $\boldsymbol{X} = \boldsymbol{U}\boldsymbol{\Sigma}\boldsymbol{V}^\intercal$ using `la.svd`

7. square the resulting singular values and compare them to the eigenvalues you found above; what do you observe ?

   As the theory suggests we observe that the squared singular values equal the eigenvalues of the matrix C (again up to permutation). Nevertheless we see distortions between the two sets (see figure **??**). This is probably due to numerical error, since `la.svd` doesn't directly compute the eigenvalues but their square root. The magnitude of the eigenvalues that show differences also support our hypothesis. Still the differences are negligible.

8. create a plot of the three spectra you just computed; do you get any warnings or error messages ?

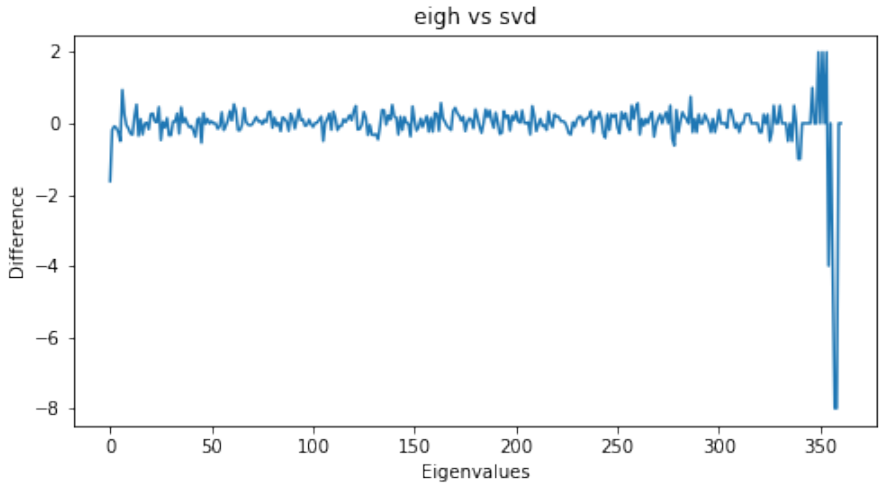
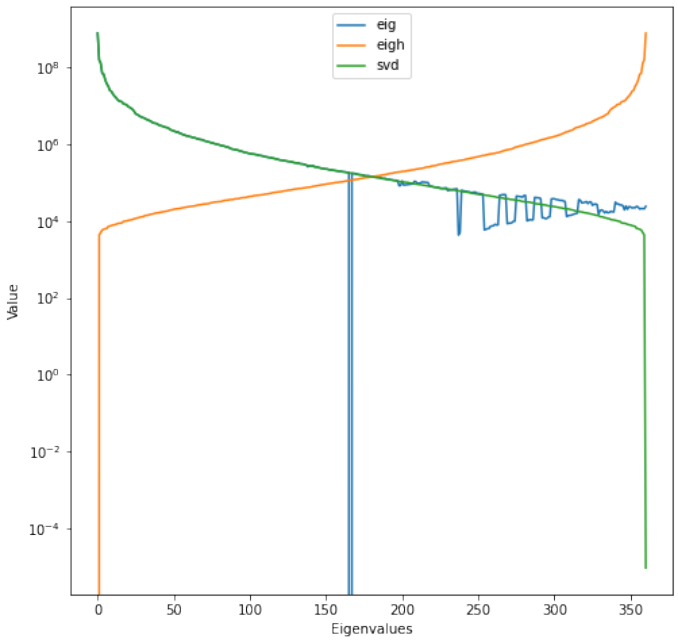


Figure 2: Difference between the eigenvalues computed by `la.eigh` and `la.svd`. Other than before there actually are differences between the two spectra. But w.r.t to the magnitude of the values, the error is negligible. Note that the eigenvalues are sorted in ascending order.



task 1.2 [10 points]

timing spectral- and singular value decompositions

Proceed just as above, but now measure the (average) run-time of the methods `la.eig`, `la.eigh`, and `la.svd`.

Here, it is highly recommended you make use of Python's `timeit` module. Examples for its proper use can be found in

> C. Bauckhage, **"NumPy / SciPy Recipes for Image Processing: Avoiding for Loops over Pixel Coordinates"**, technical report, 2018

Make sure that your run-time comparisons are fair! In other words, pay attention to the fact that `la.eig` and `la.eigh` require to compute matrix $\boldsymbol{C}$ from $\boldsymbol{X}$ whereas `la.svd` works directly on $\boldsymbol{X}$.

```
from timeit import timeit
n = 100
eig_time = timeit(lambda: la.eig(X @ X.T), number=n)
eigh_time = timeit(lambda: la.eigh(X @ X.T), number=n)
svd_time = timeit(lambda: la.svd(X)[1] ** 2, number=n)
# print the average time spend for each method
print("la.eig:  %.3fs" % eig_time / n)
print("la.eigh: %.3fs" % eigh_time / n)
print("la.svd:  %.3fs" % svd_time / n)
```

Run your code. What do you observe ? Do your result so far suggest any preference for any of the methods considered up to this point ?

The code outputs the following:

la.eig: 0.090s
la.eigh: 0.024s
la.svd: 0.909s

We see that singular value decomposition is by far the slowest method. This observation can be explained by the dimensions of the matrix X (361×2429) in comparison to C (361×361).

The fastest method is `la.eigh`. Reason for this is that `la.eigh` assumes the input matrix to be symmetric/hermitian and thus can use a faster algorithm than `la.eig`. On the other hand `la.eig` doesn't assume anything on the input matrix other than it being square.

Together with the observations from exercise 1, the prefered method for this specific problem instance is `la.eigh` for the already mentioned reasons.

task 1.3 [10 points]

implementing the QR algorithm

Given the matrix $\boldsymbol{C}$ as computed in task 1.1, implement the following iterative procedure

---

$\boldsymbol{C}_0 = \boldsymbol{C}$

for $t = 1, \ldots, 10$

compute the QR decomposition of $\boldsymbol{C}_{t-1}$ to obtain matrices $\boldsymbol{Q}_t$ and $\boldsymbol{R}_t$

given matrices $\boldsymbol{Q}_t$ and $\boldsymbol{R}_t$, compute

$\boldsymbol{C}_t = \boldsymbol{R}_t \boldsymbol{Q}_t$

```
# past your code here
```

Run your code and have a look at the diagonal entries of the resulting matrix $\boldsymbol{C}_{10}$. Compare them against the spectra you computed above; what do you observe ?

enter your discussion here . . .

Now, consider the following equivalencies

$$\boldsymbol{C}_t = \boldsymbol{R}_t \boldsymbol{Q}_t = \boldsymbol{Q}_t^{-1} \boldsymbol{Q}_t \boldsymbol{R}_t \boldsymbol{Q}_t = \boldsymbol{Q}_t^{-1} \boldsymbol{C}_{t-1} \boldsymbol{Q}_t = \boldsymbol{Q}_t^{\intercal} \boldsymbol{C}_{t-1} \boldsymbol{Q}_t$$

Can you use this insight to explain the result you get form running the QR algorithm ?

enter your discussion here . . .

bonus [5 points]

If you want to impress your instructors, perform run-time measurements for your implementation of the QR algorithm as well.

enter your discussion here . . .

task 1.4 [20 points]

finding maximally different images

So far, you only had to do what the problem specifications asked you to do. In this task, you need to get creative!

Reread matrix $\boldsymbol{X}$ from task 1.1 into memory but *do not normalize it to zero mean*. Now, think of / invent an algorithm that selects $k > 2$ out of the n columns of $\boldsymbol{X}$ such that the selected data vectors are as far apart as possible.

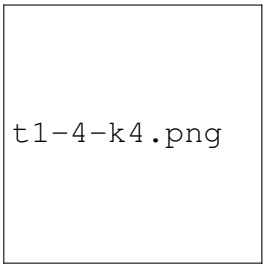
Mathematically, the problem considered in this task is a subset selection problem and we can formalize it as follows: given $X = \{\boldsymbol{x}_1, \ldots, \boldsymbol{x}_n\}$, solve

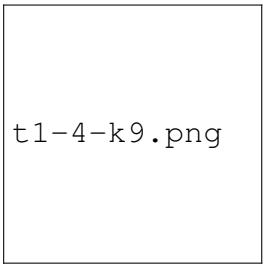
$$S^* = \underset{S \subset X}{\operatorname{argmax}} \sum_{\boldsymbol{x}_i \in S} \sum_{\boldsymbol{x}_j \in S} \left\| \boldsymbol{x}_i - \boldsymbol{x}_j \right\|^2$$
$$\text{s.t.} \quad |S| = k$$

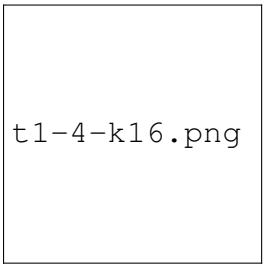
NOTE: this problem is actually way more difficult than it may appear at first sight; in fact, it is NP-complete and an *efficient* algorithm that is *guaranteed* to find the optimal S^* for very large sets X and arbitrary choices of k remains elusive to this date.

```
# past your code here
```

Test your algorithm for several choices of k, say $k \in \{25, 50, 100\}$, and visualize the extracted columns in terms of tiny images. For instance, for $k \in \{4, 9, 16\}$, your result could look like this:







Simply replace the above three images with your results for $k \in \{25, 50, 100\}$.