exercise 2

groups, complex numbers, and flows

solutions due

until **May 09, 2021** at **23:59** via **E-Mail**

Fabrice: s6fabeau[at]uni-bonn.de
Sebastian: muellers[at]cs.uni-bonn.de

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

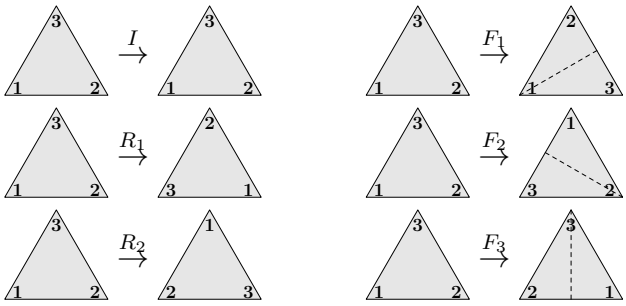
```
import numpy as np
import numpy.linalg as la
import matplotlib.pyplot as plt
```

task 2.1 [5 points]

the dihedral group D_3

NOTE: If you decide to give this task a try, then really try to solve it yourself! Those who do not like to- or cannot think for themselves may of course just go to wikipedia and look up the solution . . . but what would you gain from that?

The following are the six symmetries (three rotations and three reflections) of a unilateral triangle:



Using the notation introduced in this figure, the dihedral group $D_3 = \big(G, \circ\big)$ consists of the following elements

$$G = \big\{I, R_1, R_2, F_1, F_2, F_3\big\}$$

and the group law

$$\circ \,:\, G \times G \to G$$

is the composition, i.e. sequential execution, of any two operations in G. Given this information, complete the Cayley table for this group:

$\circ$	I	R_1	R_2	F_1	F_2	F_3
I	I	R_1	R_2	F_1	F_2	F_3
R_1	R_1	R_2	I	F_2	F_3	F_1
R_2	R_2	I	R_1	F_3	F_1	F_2
F_1	F_1	F_3	F_2	I	R_2	R_1
F_2	F_2	F_1	F_3	R_1	I	R_2
F_3	F_3	F_2	F_1	R_2	R_1	I

Does your result ($\Leftrightarrow$ the structure of the completed table) suggest that D_3 is an Abelian group or not ?

The Caylay table shows that the composition $\circ : G \times G \to G$ is not symmetric which implied that D_3 is not an Abelian group.

task 2.2 [5 points]

the quaternion group Q_8

NOTE: If you decide to give this task a try, then once again really try to solve it yourself!

Recall that any complex number $z \in \mathbb{C}$ can be written as $z = a \cdot 1 + b \cdot i$ where $1, a, b \in \mathbb{R}$ and $i = \sqrt{-1}$.

Also recall that complex numbers can be thought of as living in a 2D space called the complex plane.

In 1843, Hamilton famously extended the idea of complex numbers to 4D and introduced the *quaternions* $q \in \mathbb{H}$ where

$$q = a \cdot 1 + b \cdot i + c \cdot j + d \cdot k$$

Here, $a, b, c, d \in \mathbb{R}$ and the *unit quaternions* i, j, k are defined by their properties / multiplication rules

$$i^2 = -1 \qquad j^2 = -1 \qquad k^2 = -1 \qquad ijk = -1$$

$$ij = k \qquad jk = i \qquad ki = j$$
$$ji = -k \qquad kj = -i \qquad ik = -j$$

Given this information, complete the Cayley table for this group:

$\cdot$	$+1$	-1	$+i$	$-i$	$+j$	$-j$	$+k$	$-k$
$+1$	$+1$	-1	$+i$	$-i$	$+j$	$-j$	$+k$	$-k$
-1	-1	$+1$	$-i$	$+i$	$-j$	$+j$	$-k$	$+k$
$+i$	$+i$	$-i$	-1	$+1$	$+k$	$-k$	$-j$	$+j$
$-i$	$-i$	$+i$	$+1$	$+1$	$-k$	$+k$	$+j$	$-j$
$+j$	$+j$	$-j$	$-k$	$+k$	-1	$+1$	$+i$	$-i$
$-j$	$-j$	$+j$	$+k$	$-k$	$+1$	$+1$	$-i$	$+i$
$+k$	$+k$	$-k$	$+j$	$-j$	$-i$	$+i$	-1	$+1$
$-k$	$-k$	$+k$	$-j$	$+j$	$+i$	$-i$	$+1$	$+1$

Does your result ($\Leftrightarrow$ the structure of the completed table) suggest that Q_8 is an Abelian group or not ?

The Caylay table is again not symmetric and thus the group $(H, \cdot)$ is not abelian. This can also directly be seen by the somewhat skew-symmetric relationship between unit quaternions, i.e.

$$ik = -j = (-1) \cdot j = (-1) \cdot ki = -ki$$

task 2.3 [5 points]

complex numbers as matrices

Python has an in-built data type for complex numbers. For instance, the two numbers $z_1 = 3 \cdot 1 + 4 \cdot i$ and $z_2 = 2 \cdot 1 - 2 \cdot i$ can be implemented as

```
z1 = complex(3, +4)
z2 = complex(2, -2)
```

or simply as

```
z1 = 3 + 4j
z2 = 2 - 2j
```

NOTE: For people other than electrical engineers, it may be confusing that Python refers to the imaginary unit i as `j`. But it is how it is.

Here is your *first sub-task*: using Python, compute the values of the four simple expressions $z_1 + z_2$, $z_1 \cdot z_2$, $z_1^* = 3 \cdot 1 - 4 \cdot i$, and $|z_1|$.

Now consider this: complex numbers can also be represented in terms of real valued 2×2 matrices. To see this, consider the two "basis" matrices

$$\mathbf{1} = \begin{bmatrix} +1 & 0 \\ 0 & +1 \end{bmatrix}$$

$$\boldsymbol{i} = \begin{bmatrix} 0 & -1 \\ +1 & 0 \end{bmatrix}$$

and let

$$\boldsymbol{z}_1 = 3\,\mathbf{1} + 4\,\boldsymbol{i}$$
$$\boldsymbol{z}_2 = 2\,\mathbf{1} - 2\,\boldsymbol{i}$$

Here is your *second sub-task*: implement these objects in Numpy and compute $\boldsymbol{z}_1 + \boldsymbol{z}_2$, $\boldsymbol{z}_1 \cdot \boldsymbol{z}_2$, $\boldsymbol{z}_1^\intercal$, and $\sqrt{\det \boldsymbol{z}_1}$ (where $+$ and $\cdot$ denote matrix addition and multiplication).

```
# paste your code here
```

Print the matrices $\boldsymbol{z}_1$ and $\boldsymbol{z}_2$ as well as the results of your computations. What do you observe ? How do your results relate to the results you got in the first sub-task ?

enter your discussion here . . .

bonus [10 points]

Just as the complex numbers $z \in \mathbb{C}$, the quaternions $q \in \mathbb{H}$ can be represented in terms of matrices, too. Here, we actually have to choices:

1. either, we may introduce certain 4×4 real valued matrices $\mathbf{1}, \boldsymbol{i}, \boldsymbol{j}, \boldsymbol{k}$ which represent the unit quaternions $1, i, j, k$ so that a quaternion $\boldsymbol{q} = a\,\mathbf{1} + b\,\boldsymbol{i} + c\,\boldsymbol{j} + d\,\boldsymbol{k}$ is a matrix $\boldsymbol{q} \in \mathbb{R}^{4\times4}$. Do you have an idea how the "basis" matrices $\mathbf{1}$, $\boldsymbol{i}$, $\boldsymbol{j}$, and $\boldsymbol{k}$ could look like? If so, implement them in Numpy and compute the product $\boldsymbol{ijk}$.

```
# paste your code here
```

2. or, we may introduce certain 2×2 complex valued matrices $\mathbf{1}, \boldsymbol{i}, \boldsymbol{j}, \boldsymbol{k}$ which represent the unit quaternions $1, i, j, k$ so that a quaternion $\boldsymbol{q} = a\,\mathbf{1} + b\,\boldsymbol{i} + c\,\boldsymbol{j} + d\,\boldsymbol{k}$ is a matrix $\boldsymbol{q} \in \mathbb{C}^{2\times2}$. Do you have an idea how the complex valued "basis" matrices $\mathbf{1}$, $\boldsymbol{i}$, $\boldsymbol{j}$, and $\boldsymbol{k}$ could look like? If so, implement them in Numpy and compute the product $\boldsymbol{ijk}$.

```
# paste your code here
```

Print your results and discuss what you observe.

enter your discussion here . . .

task 2.4 [20 points]

solving the Oja flow

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

`GaussianSample3D.csv`

which contains a (zero mean) data matrix $\boldsymbol{X} \in \mathbb{R}^{3\times 250}$ whose columns $\boldsymbol{x}_i$ represent 3D data points. To read this matrix into memory and check its size, you may use

```
X = np.loadtxt('GaussianSample3D.csv', delimiter=', ')
m, n = X.shape
print (m, n)
```

Given this data matrix, compute the sample covariance matrix

$$\boldsymbol{C} = \frac{1}{n}\boldsymbol{X}\boldsymbol{X}^\intercal$$

and its spectral decomposition

$$\boldsymbol{C} = \boldsymbol{U}\boldsymbol{\Lambda}\boldsymbol{U}^\intercal$$

and print the leading eigenvector, i.e. the eigenvector $\boldsymbol{u}_1$ belonging to the largest eigenvalues λ_1.

```
# build the sample covariance matrix
C = 1/X.shape[1] * (X @ X.T)
# compute the spectral decompostion of the covariance matrix
# remember from the last assignment sheet that la.eigh is a
# good choice here, since the covariance matrix is symmetric
# V_eig: eigenvalues of C in ascending order
# U_eig: eigenvectors of C
V_eig, U_eig = la.eigh(C)
# show the eigenvector corresponding to the largest eigenvalue
# recall the ascending order of the eigenvalues
print("u_1:", U_eig[:, -1])
```

Now recall that, in the lecture, we studied the Oja flow, i.e. the following dynamical system

$$\dot{\boldsymbol{w}} = \left(\boldsymbol{I} - \boldsymbol{w}\boldsymbol{w}^\intercal\right)\boldsymbol{C}\boldsymbol{w} \tag{1}$$

where we simply write $\dot{\boldsymbol{w}}$ and $\boldsymbol{w}$ instead $\dot{\boldsymbol{w}}(t)$ and $\boldsymbol{w}(t)$. In other words, (1) is an ordinary differential equation w.r.t. time t.

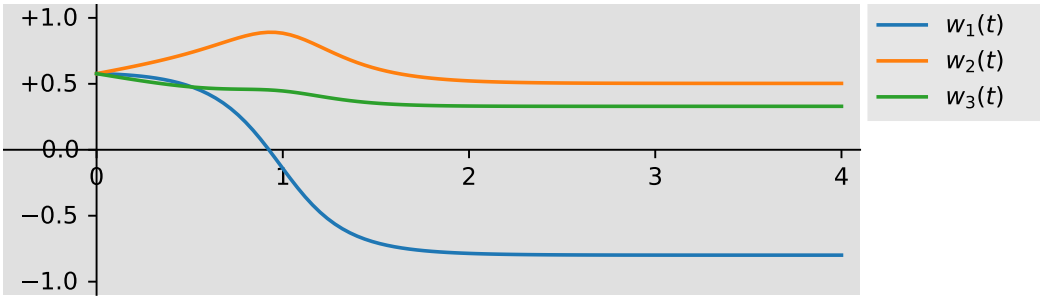
We claimed that, for initial conditions $\boldsymbol{w}(0) = \boldsymbol{w}_0$ such that $\|\boldsymbol{w}_0\| = 1$, this flow will converge to the leading eigenvector of $\boldsymbol{C}$.

In this task, you are supposed to verify this using numerical methods (rather than mathematical proof). In other words, first initialize some unit vector $\boldsymbol{w}_0$ and then use Scipy's `odeint` method in order to numerically solve the differential equation in (1).

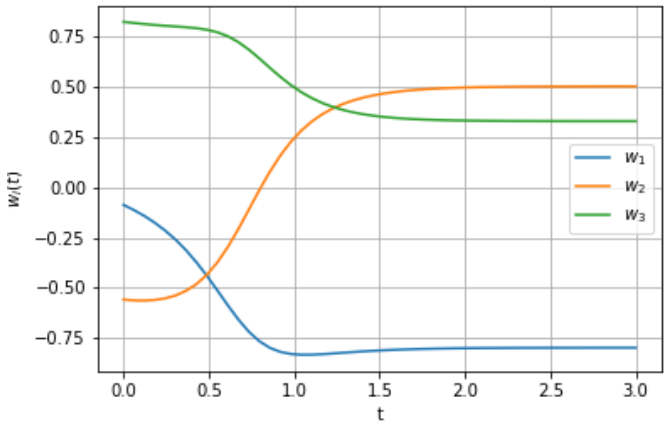
The method `odeint` is contained in the module `scipy.integrate` and fairly well documented on the Web. Try to figure out for yourself how to use it. That is, see for yourself what kind of inputs it requires and what kind of outputs it yields.

```
from scipy.integrate import odeint
# implement the differential equation
w_prime = lambda w, t: (np.eye(3) - w[:, None] @ w[None, :]) @ C @ w
# create a random unit vector as an initial guess
w_0 = np.random.uniform(-1, 1, size=(3,))
w_0 /= la.norm(w_0)
# solve the differential equation using scipy's odeint method
t = np.linspace(0, 3, 50)
w = odeint(w_prime, w_0, t)
# print approximation of u_1
print("w_t:", w[-1, :])
```

Once you have solved (1) for appropriate initial conditions, plot your result. In principle your plot should look something like this (not necessarily as fancy though):



NOTE: This result was obtained using $\boldsymbol{w}_0 = \frac{1}{\sqrt{3}}\mathbf{1}$. When you enter your result below, please use some random vector $\boldsymbol{w}_0$ with $\|\boldsymbol{w}_0\| = 1$.



Compare the stable point the flow converges to to the leading eigenvector $\boldsymbol{u}_1$ you computed above. What do you observe? Also, experiment with different initial conditions / starting point for the flow. What do you observe?

The two code fragments output the following:

u_1: [0.7988784 , -0.50319912, -0.32952077]
w_t: [0.79885363, -0.50324345, -0.32951316]

So one can see that w_t is in fact an approximation of the leading eigenvector u_1 of the sample covariance matrix C. Depending on the initial condition for the flow it might happen that the the two vectors are of different sign, i.e. $w_t \approx -u_1$. This is represented by the cosine similarity converging to -1 instead of +1. However $-u_1$ is still a leading eigenvector of C.

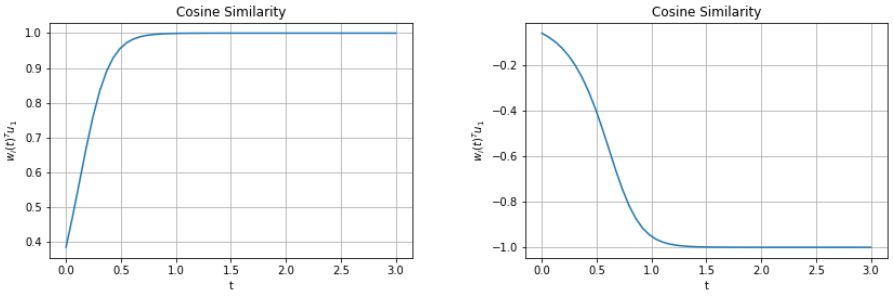


Figure 1: Cosine similaritity between $w(t)$ and u_1 for two different setups. Note that the two setups differ in the initial condition w_0, i.e. once $w_0^T u_1 > 0$ (left) and once $w_0^T u_1 < 0$ (right).

To get a bit more precise one can observe that the cosine similarity between the iterative approximations w_i and the target u_1 never cross the x-axes (see figure 1). What does this mean? Consider the plane containing the origin and for which u_1 is a normal vector. Then all the iterative approximations w_t will be on the same side of the plane. This includes not just the randomly chosen initial condition w_0 but also the vector of convergence. Thus we see the following condition:

$$w_t \approx sgn(w_0^T u_1) u_1$$

task 2.5 [5 points]

the Oja flow is isometric

Prove the following claim: If the Oja flow

$$\dot{\boldsymbol{w}} = \big(\boldsymbol{I} - \boldsymbol{w}\boldsymbol{w}^\intercal\big)\boldsymbol{C}\boldsymbol{w} \tag{2}$$

starts with a unit vector $\boldsymbol{w}(0) = \boldsymbol{w}_0$ such that $\|\boldsymbol{w}_0\|^2 = \boldsymbol{w}_0^\intercal \boldsymbol{w}_0 = 1$, then it holds that

$$\frac{d}{dt}\|\boldsymbol{w}(t)\|^2 = 0.$$

Observe that this is to say that, for unit vectors, the flow in (2) is *norm preserving* or *isometric*.

enter your proof here

$$\frac{d}{dt}\|\boldsymbol{w}\|^2 = \dots$$

task 2.6 [10 points]

the QR algorithm can sort!

In task 1.3, you already implemented the QR algorithm and used it to compute the eigenvalues of a sample covariance matrix. Yet, amazingly (?), the QR algorithm can do much more . . .

To see an example, implement the following procedure in Numpy / Scipy

1. Assume you were given a vector

   $$\boldsymbol{x} = [x_1, x_2, x_3, \ldots, x_n]^\intercal$$

   consisting of n elements $x_i \in \mathbb{R}$

```
# get the dimension of the input vector
n = x.shape[0]
```

2. Create an $n \times n$ tri-diagonal matrix $\boldsymbol{X}$ whose main diagonal contains the elements of $\boldsymbol{x}$ and whose off diagonals contain a fixed small number ϵ. In other words, create a matrix like this

   $$\boldsymbol{X} = \begin{bmatrix} x_1 & \epsilon & & & \\ \epsilon & x_2 & \epsilon & & \\ & \epsilon & x_3 & \epsilon & \\ & & & \ddots & \epsilon \\ & & & \epsilon & x_n \end{bmatrix}$$

```
# build the tri-diagonal matrix with the vector x on the main
# diagonal and eps on both off-diagonals
off_diag = [eps] * (n-1)
X = np.diag(x) + np.diag(off_diag, k=1) + np.diag(off_diag, k=-1)
```

3. compute

   $$\boldsymbol{X}' = \exp\big(\boldsymbol{X}\big)$$

```
# compute the matrix exponential
X_prime = la.expm(X)
```

4. feed $\boldsymbol{X}'$ into the QR algorithm to obtain a matrix

   $$\boldsymbol{Y}' = \mathrm{qrAlgorithm}\big(\boldsymbol{X}'\big)$$

```
# apply the qr-algorithm
Y_prime = qr_alg(X_prime, **kwargs)
```

5. compute

$$\boldsymbol{Y} = \log\left(\boldsymbol{Y}'\right)$$

```
# compute the matrix logarithm
Y = la.logm(Y_prime)
```

NOTE: In this task, we write $\exp(\cdot)$ and $\log(\cdot)$ to denote the matrix exponential- and logarithm functions. Scipy provides them as *la.expm* and *la.logm*.

In order to work with a specific example, run you code with the following input and parameter

$$\boldsymbol{x} = [4, -3, 2, 7, 12, 1]^\intercal$$
$$\epsilon = 0.0001$$

Run your code four times using $t_{\text{max}} \in \{1, 5, 10, 50\}$ iterations for the QR algorithm in step 4. After each run, print the diagonal entries of the matrix $\boldsymbol{Y}$ you obtain in step 5. What do you observe ?

Just for completeness we insert the full code here:

```
def qr_alg(A:np.ndarray, k:int =10) -> np.ndarray:
    # iterate for k iterations
    for _ in range(k):
        # decompose and build next matrix
        Q, R = la.qr(A)
        A = R @ Q
    # return last
    return A
def qr_sort(x:np.ndarray, eps:float =1e-4, **kwargs) -> np.ndarray:
    # get the dimension of the input vector
    n = x.shape[0]
    # build the tri-diagonal matrix with the vector x on the main
    # diagonal and eps on both off-diagonals
    off_diag = [eps] * (n-1)
    X = np.diag(x) + np.diag(off_diag, k=1) + np.diag(off_diag, k=-1)
    # compute the matrix exponential
    X_prime = la.expm(X)
    # apply the qr-algorithm
    Y_prime = qr_alg(X_prime, **kwargs)
    # compute the matrix logarithm
    Y = la.logm(Y_prime)
    # return the diagonal entries
    return np.diag(Y)
# create some sample input
x = np.asarray([4, -3, 2, 7, 12, 1], dtype=np.float32)
# apply for different iterations of the qr algorithm
for k in [1, 5, 10, 50]:
    sorted_x = qr_sort(x, k=k)
    sorted_x = np.rint(sorted_x)
    print("k=%i:\t" % k, sorted_x)
```

The output looks as follows:

$k = k : [4, -3, 2, 7, 12, 1]$
$k = 5 : [4, 12, 7, 2, -3, 1]$
$k = 10 : [12, 4, 7, 2, -3, 1]$
$k = 50 : [12, 7, 4, 2, 1, -3]$

One can observe that with increasing number of iterations of the QR algorithm, the output is being sorted better and better.