PR2_sheet2

May 6, 2021

1 Exercise 2 - Groups, Complex Numbers, and Flows

In [90]:
```
import numpy as np
import scipy.linalg as la
import matplotlib.pyplot as plt

from scipy.integrate import odeint
```

1.1 Task 2.1

We compute the Cayley table for the dihedral group D_3. We first apply the operator in the row, then the operator in the column, i.e. $R_1 \circ F_1 = F_2$. This leads to the following table:

For a group to be an Abelian Group the group law $\circ$ needs to be commutative, i.e., $\forall a, b \in G : a \circ b = b \circ a$. This would lead to a symmetric Cayley table and as one can clearly see the Cayley table for the dihedral group D_3 is not symmetric.

From the table we can also see that each operation is represented exactly once in each row and column.

1.2 Task 2.2

We can see that Q_8 is not abelian as well.

1.3 Task 2.3

In [102]:
```
# we have two ways to initialize complex variables in Python
# calling the complex type directly:
z1 = complex(3, +4)
z2 = complex(2, -2)

# or using syntactic sugar:
z1 = 3 + 4j
z2 = 2 - 2j

print ('computing with complex numbers')
print ('z1 + z2 = ', z1+z2)
print ('z1 * z2 = ', z1*z2)
print ('con(z1) = ', z1.conjugate())
print ('abs(z1) = ', abs(z1))
```

```python
print ()



mat1 = np.array([[+1, 0], [0, +1]])
mati = np.array([[0, -1], [+1, 0]])

# define z1, z2 in terms of the matrices:
matz1 = 3*mat1 + 4*mati
matz2 = 2*mat1 - 2*mati

print ('computing with matrix representations of complex numbers')
print ('Z1 + Z2 = \n', matz1 + matz2, '\n')
print ('Z1 @ Z2 = \n', matz1 @ matz2, '\n')
print ('Z1.T = \n', matz1.T, '\n')
print ('sqrt(det(Z1)) = ', np.sqrt(la.det(matz1)))
print ('\n\n')
```

```
computing with complex numbers
z1 + z2 =  (5+2j)
z1 * z2 =  (14+2j)
con(z1) =  (3-4j)
abs(z1) =  5.0

computing with matrix representations of complex numbers
Z1 + Z2 =
 [[ 5 -2]
 [ 2  5]]

Z1 @ Z2 =
 [[14 -2]
 [ 2 14]]

Z1.T =
 [[ 3  4]
 [-4  3]]

sqrt(det(Z1)) =  5.0
```

2 TODO

We notice that:

Bonus task

In [103]:
```
### unit quaternions as 4 x 4 real matrices
mat1 = np.eye(4)

mati = np.array([[0, -1, 0, 0],
                 [+1, 0, 0, 0],
                 [0, 0, 0, -1],
                 [0, 0, +1, 0]])
matj = np.array([[0, 0, -1, 0],
                 [0, 0, 0, +1],
                 [+1, 0, 0, 0],
                 [0, -1, 0, 0]])
matk = np.array([[0, 0, 0, -1],
                 [0, 0, -1, 0],
                 [0, +1, 0, 0],
                 [+1, 0, 0, 0]])

print ('i @ j @ k = \n', mati @ matj @ matk)
print ()
```

```
i @ j @ k =
 [[-1  0  0  0]
 [ 0 -1  0  0]
 [ 0  0 -1  0]
 [ 0  0  0 -1]]
```

In [104]:
```
### unit quaternions as 2 x 2 complex matrices
mat1 = np.eye(2)

###
### NOTE: these matrices work and are a valid solution
###
# mati = np.array([[+1j, 0],
#                  [0, -1j]])
# matj = np.array([[0, +1],
#                  [-1, 0]])
# matk = np.array([[0, +1j],
#                  [+1j, 0]])

###
### however, if we multiply the following matrices by i,
### we obtain the Pauli matrices sx = i*matK, sy = i*matj, sz = i*mati
### in this sense, these matrices constitute a more interesting solution
###
mati = np.array([[+1j, 0],
                 [0, -1j]])
matj = np.array([[0, -1],
```

```
                  [+1, 0]])
matk = np.array([[0, -1j],
                 [-1j, 0]])

print ('i @ j @ k = \n', mati @ matj @ matk)
```

```
i @ j @ k =
 [[-1.+0.j  0.+0.j]
 [ 0.+0.j -1.+0.j]]
```

3 TODO

3.1 Task 2.4

```
In [92]: matX = np.loadtxt('exercise2/GaussianSample3D.csv', delimiter=', ')
         m, n = matX.shape
         print (m, n)

         C = 1/n * matX @ matX.T
         print(C.shape)
         vals, vecs = la.eigh(C)
         print(f"eigen values:\n {vals} \neigen vecs:\n {vecs}")

         _w_prime = lambda w: (np.eye(w.shape[0]) - np.outer(w,w)) @ C @ w

         def sample_w0(dim):
             w = np.random.rand(dim)
         #      w = w/np.linalg.norm(w)
         #      assert np.isclose(np.sqrt(np.sum(w**2)), 1.)
             return w/np.linalg.norm(w)

         sample_w0(3)
```

```
3 250
(3, 3)
eigen values:
 [0.7 1.4 5.1]
eigen vecs:
 [[ 0.4 -0.5  0.8]
 [-0.  -0.9 -0.5]
 [ 0.9  0.1 -0.3]]
```

```
Out[92]: array([0.6, 0.6, 0.6])
```

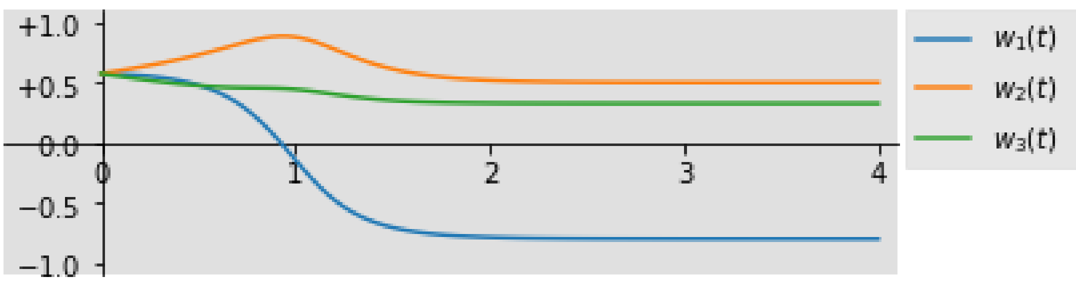
```
In [93]: def deriv(w, t, C, I):
             return (I - np.outer(w, w)) @ C @ w
```

In [108]:
```
### load data matrix
matX = np.loadtxt('exercise2/GaussianSample3D.csv', delimiter=', ')
m, n = matX.shape
print (m,n)

### compute sample covariance matrix
matC = np.cov(matX)

### compute spectral decomposition and print leading eigenvector
vecL, matU = la.eigh(matC)
print (matU[:,-1])

### prepare ingredients for solving Oja's flow
# identity matrix
matI = np.eye(m)

# initial unit vector w(0)
vecW = np.ones(m) / np.sqrt(m)

# time steps for ODE solver
stps = np.linspace(0, 4, 101)

### use odeint to solve Oja flow
matW = odeint(deriv, vecW, stps, (matC,matI))
flow = matW.T

### print stable point (pint the flow converges to)
print (matW[-1])
```

```
3 250
[ 0.8 -0.5 -0.3]
[-0.8  0.5  0.3]
```

In [111]:
```
### plot the evolution over time
# initialize figure and axes
fig = plt.figure()
fig.patch.set_facecolor('w')
axs = fig.add_subplot(211, facecolor='#e0e0e0')

# nicer way of showing coordinate axes
for pos in ['left','bottom']:
    axs.spines[pos].set_position('zero')
    axs.spines[pos].set_zorder(1)
for pos in ['right','top']:
    axs.spines[pos].set_visible(False)
axs.xaxis.set_ticks_position('bottom')
axs.yaxis.set_ticks_position('left')
```

```
axs.tick_params(direction='out')

for i, wt in enumerate(flow):
    axs.plot(stps, wt, '-', label=r'$w_{%i}(t)$' % (i+1))

leg = axs.legend(bbox_to_anchor=(1.01,0.0,0.2,1), loc="upper left",
                 mode="expand",
                 borderaxespad=0.,
                 facecolor='#e0e0e0', edgecolor='#e0e0e0',
                 fancybox=False)

axs.set_xlim(-0.5, np.max(stps+0.1))
axs.set_ylim(-1.1, 1.1)
ytics = np.linspace(-1, +1, 5)
ylabs = ['${0:{1}}$'.format(t, '+' if t else '') for t in ytics]
axs.set_yticks(ytics)
_ = axs.set_yticklabels(ylabs)
```



4 TODO

discussion after figure

4.1 Task 2.5

Oja flow is isometric # TODO

4.2 Task 2.6

```
In [58]: def qrAlgorithm(X, tmax=10):
             C = X @ X.T

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

             return C
```

```
In [113]: ### a vector x whose entries are supposed to be sorted
          vecX = np.array([4, -3, 2, 7, 12, 1])
          print ('elements of vector x:', vecX)




          ### create a tridiagonal matrix with x on the diagonal
          ### and rather small entries on the two subdiagonals
          n   = vecX.size
          eps = 0.0001

          matX = np.diag(vecX) \
                 + eps * np.diag(np.ones(n-1), +1) \
                 + eps * np.diag(np.ones(n-1), -1)



          ### just for the fun of it, look at the eigenvalues of X
          vecL, matU = la.eigh(matX)
          print ('eigenvalues of matrix X:', vecL)



          # use the QR algorithm to solve Toda flow which, for a
          # tridiagonal matrix X, is equivalent to Brockett flow
          matY = la.logm(qrAlgorithm(la.expm(matX), tmax=5))
          print ('diagonal of matrix Y:', np.diag(matY))
          print ()


          ### Solution to the actual task:
          for tmax in [1, 5, 10, 50]:
              matY = la.logm(qrAlgorithm(la.expm(matX), tmax=tmax))
              print ('diagonal of matrix Y:', np.diag(matY))
```

```
elements of vector x: [ 4 -3  2  7 12  1]
eigenvalues of matrix X: [-3.  1.  2.  4.  7. 12.]
diagonal of matrix Y: [24.  8. 14.  4. -6.  2.]

diagonal of matrix Y: [ 8.  -4.2  4.2 13.8 22.2  2. ]
diagonal of matrix Y: [24.  8. 14.  4. -6.  2.]
diagonal of matrix Y: [24. 14.  8.  4.  2. -6.]
diagonal of matrix Y: [24. 14.  8.  4.  2. -6.]
```

5 TODO