```
#!/usr/bin/env python3
# -*- coding: utf-8 -*-
"""
Created on Mon Nov  6 13:31:07 2017

@author: christianehelzel
"""

# 5. Vorlesung
```

```
'Created on Mon Nov  6 13:31:07 2017 @author: christianehelzel'
```

1 Wiederholung: Definition von Arrays

```
import numpy as np
# Definiere Matrix
A = np.array([[1,2],[3,4]]);
# Definiere Vektor
v = np.array([1,2]);
```

1.1 Einfache Matrizen / Vektoren

Matrix mit Einsen

```
np.ones((3,2))
```

```
array([[ 1.,  1.],
       [ 1.,  1.],
       [ 1.,  1.]])
```

Nullmatrix

```
np.zeros((3,2))
```

```
array([[ 0.,  0.],
       [ 0.,  0.],
       [ 0.,  0.]])
```

Einheitsmatrix

```
np.eye(3)
```

```
array([[ 1.,  0.,  0.],
       [ 0.,  1.,  0.],
       [ 0.,  0.,  1.]])
```

1.2 Einfache Operationen für Arrays

```
A = np.array([[1,2],[3,4]]);
v = np.array([1,2]);
```

transponierte Matrix

```
print(A)
print(A.T)
```

```
[[1 2]
 [3 4]]
[[1 3]
 [2 4]]
```

komplexwertige Matrix

```
C = np.array([[1.+2.j,2.-1.j],[2.+1.j,3.-5.j]]);
print(C)
```

```
[[ 1.+2.j  2.-1.j]
 [ 2.+1.j  3.-5.j]]
```

transponiert-konjugierte Matrix

```
C.conj().T
```

```
array([[ 1.-2.j,  2.-1.j],
       [ 2.+1.j,  3.+5.j]])
```

1.3 Achtung: +, -, * , % wirken komponentenweise

Die Standardoperationen wirken komponentenweise auf die Einträge der Arrays. A * A ist also KEINE Matrix-Matrix-Multiplikation!

```
A = np.array([[1,2],[3,4]]);
```

Komponentenweises +

```
A+A
```

```
array([[2, 4],
       [6, 8]])
```

Komponentenweises *

```
A*A
```

```
array([[ 1,  4],
       [ 9, 16]])
```

Komponentenweises % (auch das geht bei Arrays)

```
A%A
```

```
array([[0, 0],
       [0, 0]])
```

1.4 Matrix-Multiplikation

mit np.dot

```
A = np.array([[1,2],[3,4]]);
np.dot(A,A)
```

```
array([[ 7, 10],
       [15, 22]])
```

verkürzte Notation: @ Statt des "dot"-Befehls kann man auch das @-Zeichen verwenden.

```
A@A
```

```
array([[ 7, 10],
       [15, 22]])
```

Matrix-Vektor Produkt funktioniert genau so

```
v = np.array([1,2])
np.dot(A,v)
```

```
array([ 5, 11])
```

Auch hier kann @ verwendet werden

```
A@v
```

```
array([ 5, 11])
```

Skalarprodukt zweier Vektoren

```
np.dot(v,v)
```

```
5
```

Beachte: gleiche Resultate

```
np.dot(v,v.T)
```

```
5
```

v@v

```
np.dot(v.T,v)
```

```
5
```

Wieso ist das so? 'v' ist KEIN Zeilen/Spaltenvektor (hat nur eine Dimension), also bewirkt das Transponieren nichts.

Kreuzprodukt / Vektorprodukt mit np.cross

```
x = np.array([1,0,0])
y = np.array([0,1,0])
np.cross(x,y)
```

```
array([0, 0, 1])
```

Dyadisches Produkt / tensorielles Produkt (Multiplikation eines Spaltenvektor mit einem Zeilenvektor liefert Matrix)

```
np.outer(v,v)
```

```
array([[1, 2],
       [2, 4]])
```

Mehr Informationen zu Operationen auf Arrays finden Sie unter

https://www.python-kurs.eu/numpy_numerische_operationen_auf_arrays.php

1.5 Übersichtlicher: Definiere Vektoren als Zeilen- bzw. Spaltenvektor

@ funktionert dann immer so wie man es aus der linearen Algebra erwartet, d.h. Zeilenvektor @ Spaltenvektor liefert skalare Größe, Spaltenvektor @ Zeilenvektor liefert Matrix

```
v = np.array([1,0]).reshape(2,1)  # Spaltenvektor
u = np.array([0,1]).reshape(1,2)  # Zeilenvektor
A = np.array([[1,2],[3,4]]);
```

Zahl

```
u@v
```

```
array([[0]])
```

Matrix

```
v@u
```

```
array([[0, 1],
       [0, 0]])
```

Matrix-Vektor-Produkt

```
A@v
```

```
array([[1],
       [3]])
```

Das klappt auch von links

```
u@A
```

```
array([[3, 4]])
```

Achtung, hier stimmen die Dimensionen nicht mehr!

```
A@u
```

```
---------------------------------------------------------------------------ValueError
call last)<ipython-input-1-fd6556ebfa37> in
<module>()
----> 1A@u
ValueError: shapes (2,2) and (1,2) not aligned:
2 (dim 1) != 1 (dim 0)
```

Weiter Informationen zu reshape: https://www.python-kurs.eu/numpy_dimensionen_anpassen.php

2 Weitere Arrayoperationen

np.amax(A) oder A.max() liefert das maximale Matrixelement:

```
np.amax(A)
```

```
4
```

Maximum der 1. Zeile

```
np.max(A[0,:])
```

```
2
```

“flattens” A in ein 1-dim. Array

```
A.flatten()
```

```
array([1, 2, 3, 4])
```

np.argmin(A) / np.argmax(A) liefert Index bezüglich “flattened array” des kleinsten / größten Matrixelements Falls das größte / kleinste Element mehrfach vorkommt, dann erhält man denersten Index

```
A = np.array([[1,2],[2,1]])
np.argmax(A)
```

```
1
```

liefert das (erste) maximale Element

```
A.flatten()[np.argmax(A)]
```

```
2
```

2.1 Lösung linearer Gleichungssysteme

```
# Zuerst muss das "Lineare Algebra"-Modul von numpy importiert werden
import numpy.linalg as npl
A = np.array([[1,2,3],[0,3,7],[1,1,1]])
b = np.array([14,28,6])
# löse LGS Ax=b nach x
x = npl.solve(A,b)
```

Man kann auch die Inverse von A berechnen und “x” so berechnen (wird NICHT empfohlen)

```
Ainv=npl.inv(A)
x2=Ainv@b
x-x2
```

```
array([ -1.06581410e-14,   1.42108547e-14,  -3.55271368e-15])
```

Wie man sieht, sind die Ergebnisse (fast) gleich.

2.2 Weitere Möglichkeiten zur Definition von Arrays

A.reshape(i,j) transformiert A in ein array mit i Zeilen und j Spalten

```
A = np.arange(12)
```

A ist ein 1D array

```
A.shape
```

```
(12,)
```

B hat zwei Dimensionen

```
B = A.reshape(3,4)
B.shape
```

```
(3, 4)
```

Die Anzahl der Elemente ändert sich natürlich nicht

```
A.size-B.size
```

```
0
```

Wichtige Erinnerung: A und B zeigen noch immer auf die selben Einträge im Speicher!

```
A[3]=-42
B
```

```
array([[  0,   1,   2, -42],
       [  4,   5,   6,   7],
       [  8,   9,  10,  11]])
```

2.3 np.diag, np.triu, np.tril

```
A = np.arange(16).reshape(4,4)
```

Diagonale von A

```
np.diag(A)
```

```
array([ 0,  5, 10, 15])
```

Untere Dreiecksmatrix

```
np.triu(A)
```

```
array([[ 0,  1,  2,  3],
       [ 0,  5,  6,  7],
       [ 0,  0, 10, 11],
       [ 0,  0,  0, 15]])
```

Obere Dreiecksmatrix

```
np.tril(A)
```

```
array([[ 0,  0,  0,  0],
       [ 4,  5,  0,  0],
       [ 8,  9, 10,  0],
       [12, 13, 14, 15]])
```

np.diag kann auch Diagonalmatrizen erzeugen

np.diag für Nebendiagonalen

```
UU = np.diag(A,k=1)  # obere Nebendiagonale
LL = np.diag(A,k=-1) # untere Nebendiagonale
```

Benutze np.diag zur Definition von Matrizen

```
x = [1,2,3]
np.diag(x,k=-1)+np.diag(x,k=1)
```

```
array([[0, 1, 0, 0],
       [1, 0, 2, 0],
       [0, 2, 0, 3],
       [0, 0, 3, 0]])
```

2.4 Index Arrays

Wir können Integer-Arrays verwenden um andere Arrays zu definieren

1. Beispiel zur Verwendung von Index-Arrays

```
# Array mit den ersten 8 Zweierpotenzen
a = 2**np.arange(8);
# Array mit geraden Zahlen von 0 bis 7
i = 2*np.arange(4); # Index-Array
# Jedes zweite Element von a in b speichern
b = a[i]
```

Beachte: Diesen Vektor können wir auch folgendermaßen erzeuge:

```
a[::2]
```

```
array([ 1,  4, 16, 64])
```

Die Konstruktion unter Verwendung von Index-Arrays ist aber im Allgemeinen flexibler

2. Beispiel zur Verwendung von Index-Arrays

```
# Erzeuge einen Vektor, der die Gegendiagonale einer Matrix enthält
A = np.arange(16)
A = A.reshape(4,4)
i = np.arange(4)
j = i[::-1]

A[i,j]
```

```
array([ 3,  6,  9, 12])
```

2.5 Anwendung: Berechne die Primzahlen von 2 bis 100 unter Verwendung eines Index-Arrays

```
import numpy as np

maxnumber = 100
is_prime = np.ones(maxnumber)

# Cross out 0 and 1 which are not primes:
is_prime[:2] = 0

# cross out its higher multiples:
nmax = int(np.sqrt(maxnumber))
for i in range(2, nmax+1):
    is_prime[2*i::i] = 0

print(np.nonzero(is_prime))
# np.nonzero(a) returns the indices of the elements of a that are non-zero
```

```
(array([ 2,  3,  5,  7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47,
53, 59,
       61, 67, 71, 73, 79, 83, 89, 97]),)
```

Inspiriert durch https://www.python-kurs.eu/list_comprehension.php

2.6 Verwende boolsche Ausdrücke bei der Definition von Matrizen

Bsp: Setze alle negativen Matrixeinträge auf Null

```
A = np.arange(16)
A = A.reshape(4,4)
B = A-A.T
```

```
print(B)
B[B<0] = 0
print(B)
```

```
[[ 0 -3 -6 -9]
 [ 3  0 -3 -6]
 [ 6  3  0 -3]
 [ 9  6  3  0]]
[[0 0 0 0]
 [3 0 0 0]
 [6 3 0 0]
 [9 6 3 0]]
```

Punktweises logisches “oder” und “und”

```
B = A-A.T
print(B)
```

```
[[ 0 -3 -6 -9]
 [ 3  0 -3 -6]
 [ 6  3  0 -3]
 [ 9  6  3  0]]
```

Welche Elemente von B sind kleiner als -5 ODER größer als 6?

```
np.logical_or(B<-5,B>6)
```

```
array([[False, False,  True,  True],
       [False, False, False,  True],
       [False, False, False, False],
       [ True, False, False, False]], dtype=bool)
```

Welche Elemente von B sind kleiner als 2 UND größer als 0?

```
np.logical_and(B<2,B>0)
```

```
array([[False, False, False, False],
       [False, False, False, False],
       [False, False, False, False],
       [False, False, False, False]], dtype=bool)
```

Unter https://www.python-kurs.eu/python_numpy_maskierung.php finden Sie weitere Informationen zur boolschen Indzizierung von Arrays

2.7 Kronecker Produkt

Erzeugt ein zusammengesetztes Array aus Blöcken des 2. Arrays unter Verwendung der durch das 1. Array beschriebenen Skalierung

einmal so rum

```
np.kron([1,10,100], [5,6,7])
```

```
array([  5,   6,   7,  50,  60,  70, 500, 600, 700])
```

anders rum kommt was anderes raus

```
np.kron([5,6,7],[1,10,100])
```

```
array([  5,  50, 500,   6,  60, 600,   7,  70, 700])
```

noch ein Beispiel

```
np.kron(np.eye(2), np.ones((2,2)))
```

```
array([[ 1.,  1.,  0.,  0.],
       [ 1.,  1.,  0.,  0.],
       [ 0.,  0.,  1.,  1.],
       [ 0.,  0.,  1.,  1.]])
```

2.8 vstack und hstack

Füge eine Zeile oder Spalte b zu einer 3x3 Matrix A hinzu

Zeile anhängen mit np.vstack:

```
A = np.ones((3,3))
b = np.array((2,2,2))
np.vstack([A,b])
```

```
array([[ 1.,  1.,  1.],
       [ 1.,  1.,  1.],
       [ 1.,  1.,  1.],
       [ 2.,  2.,  2.]])
```

Spalte anhängen mit np.hstack

```
A = np.ones((3,3))
b = np.array((2,2,2)).reshape(3,1)  # erzeugt Spaltenvektor
np.hstack([A,b])
```

```
array([[ 1.,  1.,  1.,  2.],
       [ 1.,  1.,  1.,  2.],
       [ 1.,  1.,  1.,  2.]])
```

Alternativ kann man auch die (mächtigere) Funktion “np.concatenate” verwenden (siehe Hilfe)