```
#!/usr/bin/env python3
# -*- coding: utf-8 -*-
#
#Created on Mon Nov 27 09:40:32 2017
#
#@author: christianehelzel
#
```

1 8. Vorlesung

1.1 Wiederholung:

```
import numpy as np

#Gauß-Elimination (Vorlesung 7)
def Gaussian_elimination5(A):

    R = A.astype('float')
    rank = 0
    m, n = np.shape(R)
    i = 0

    for j in range(n):
        p = np.argmax(abs(R[i:m,j]))
        if p > 0: # Zeilentausch
            R[[i,p+i]] = R[[p+i, i]]
        if R[i,j] != 0:
            rank += 1
            for r in range(i+1, m):
                R[r,j:] =  R[r,j:] - (R[r,j]/R[i,j])*R[i,j:]
            i += 1
        if i >= m:
            break
    return R, rank
```

Bemerkungen: Würde man in der Funktion direkt A benutzen, würde die Eingabematrix überschrieben. Das lässt sich mit R=A.copy() verhindern. Besitzt A NUR ganze Zahlen (Integer), so lassen sich auch nur Integer in der Matrix speichern, was zu falschen Rechenergebnisssen führt. A.astype('float') gibt eine Kopie von A mit 'float' (Gleitkommazahlen) zurück. Das kann man an Beispiel testen, indem man die genannte Zeile umschreibt/löscht.

Beispiel:

```
A = np.array([[10,-7,0],[-3,2,6],[5,-1,5]])
R, rank = Gaussian_elimination5(A)
```

1.2 Rundungsfehler

Beachte: Aufgrund von Rundungsfehler kann das Programm zu falschen Ergebnissen führen. Der berechnete Rang ist dann größer als der tatsächliche Rang.

Beispiel:

```
A = np.array([[1, 2.01],[2.01, 4.0401]])
R, rank = Gaussian_elimination5(A)
print(rank)
```

```
2
```

exakter Rang: 1. Aufgrund von Rundungsfehlern erhalten wir aber ein falsches Ergebnis.
Ersetze != 0 durch abs() > tol, wobei tol eine vorgegebene Toleranz ist

```
def Gaussian_elimination6(A):

    R = A.astype('float')
    rank = 0
    m, n = np.shape(R)
    tol = 1.e-15

    i = 0
    for j in range(n):
        p = np.argmax(abs(R[i:m,j]))
        if p > 0: # Zeilentausch
            R[[i,p+i]] = R[[p+i, i]]
        if abs(R[i,j]) > tol:
            rank += 1
            for r in range(i+1, m):
                R[r,j:] =  R[r,j:] - (R[r,j]/R[i,j])*R[i,j:]
            i += 1
        if i >= m:
            break
    return R, rank


A = np.array([[1, 2.01],[2.01, 4.0401]])
R,rank = Gaussian_elimination6(A)
```

Der Rundungsfehlereinfluss wird auch bei Verwendung von “numpy.linalg” Routinen sichtbar.
Informieren Sie sich über weitere numpy.linalg Routinen
help(), help> numpy.linalg

```
import numpy.linalg as npl

A = np.array([[1, 2.01],[2.01, 4.0401]])

# Determinante der Matrix A (sollte 0 sein)
print(npl.det(A))


# Eigenwerte und Vektoren der Matrix
A = np.array([[0,1],[1,0]])
E,U=npl.eig(A)
print(E)
print(U)
```

```
8.92619311799e-16
[ 1. -1.]
[[ 0.70710678 -0.70710678]
 [ 0.70710678  0.70710678]]
```

Benutze Singulärwertzerlegung zur Berechnung des Rangs einer Matrix.

Die Anzahl von 0 verschiedener Singulärwerte einer Matrix ist gleich dem Rang der Matrix

```
import numpy as np
import numpy.linalg as npl

tol = 1.e-15
A = np.array([[1, 2.01],[2.01, 4.0401]])
U, s, V = npl.svd(A) # s sind die Singulärwerte der Matrix A
rank = np.size((abs(s)>tol).nonzero())

print(rank)
```

```
1
```

Es gilt: U @ np.diag(s)@V = A

Vergleich der Laufzeit:

```
import time

tol = 1.e-15
A = np.random.uniform(0, 1, (1000, 1000))

t0 = time.clock()

U, s, V = np.linalg.svd(A)
rank = np.size((abs(s)>tol).nonzero())

t1 = time.clock()
print(t1-t0)



import time

tol = 1.e-15
A = np.random.uniform(0, 1, (1000, 1000))

t0 = time.clock()
R, rank = Gaussian_elimination6(A)
t1 = time.clock()
print(t1-t0)
```

```
1.48
3.49
```

1.3 LR-Zerlegung

Bestimme zu einer gegebenen Matrix eine Zerlegung der Form

PA = LR

mit P Permutationsmatrix, L untere Dreiecksmatrix (L lower matrix) mit Einsen auf der Diagonale, R obere Dreicksmatrix (U upper matrix)

```
import numpy as np
import scipy.linalg

A = np.array([ [7., 3, -1, 2], [3, 8, 1, -4], [-1, 1, 4, -1], [2, -4, -1, 6] ])
P, L, U = scipy.linalg.lu(A)
print(A)
print(P)
print(L)
print(U)
# Probe
print(P@A-L@U)
```

```
[[ 7.  3. -1.  2.]
 [ 3.  8.  1. -4.]
 [-1.  1.  4. -1.]
 [ 2. -4. -1.  6.]]
[[ 1.  0.  0.  0.]
 [ 0.  1.  0.  0.]
 [ 0.  0.  1.  0.]
 [ 0.  0.  0.  1.]]
[[ 1.          0.          0.          0.        ]
 [ 0.42857143  1.          0.          0.        ]
 [-0.14285714  0.21276596  1.          0.        ]
 [ 0.28571429 -0.72340426  0.08982036  1.        ]]
[[ 7.          3.         -1.          2.        ]
 [ 0.          6.71428571  1.42857143 -4.85714286]
 [ 0.          0.          3.55319149  0.31914894]
 [ 0.          0.          0.          1.88622754]]
[[  0.00000000e+00   0.00000000e+00   0.00000000e+00   0.00000000e+00]
 [  0.00000000e+00   0.00000000e+00   0.00000000e+00  -4.44089210e-16]
 [  0.00000000e+00   0.00000000e+00   0.00000000e+00   0.00000000e+00]
 [  0.00000000e+00  -4.44089210e-16   0.00000000e+00
0.00000000e+00]]
```

Wir wollen nun selbst einen Algorithmus zur Berechnung dieser Zerlegung implementieren.

Wir betrachten zunächst eine Zerlegung der Form A = LR

1.4 Implementation der LR Zerlegung (ohne Zeilentausch)

```
# Wir kennen bereits die einfachste Form der Gauß Elimination

def Gaussian_elimination1(A):

    m, n = np.shape(A)
    R = A.astype('float')

    for j in range(n - 1):
        for i in range(j + 1, m):
            R[i,j:] = R[i,j:] - (R[i,j]/R[j,j])*R[j,j:]
    return R
```

1.4.1 Vorbetrachtungen:

Betrachte

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

Subtrahiere das

k = A[1,0]/A[0,0] fache der 1. Zeile von der 2. Zeile

Entspricht Linksmultiplikation mit Matrix

```
L = np.eye(3)
L[1,0] = -A[1,0]/A[0,0]

L@A
```

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

Subrahiere nun auch das

k=A[2,0]/A[0,0]-fache der 1. Zeile von der 3. Zeile

```
L[2,0] = -A[2,0]/A[0,0]
L@A
```

```
array([[ 1.,  2.,  3.],
       [ 0., -1., -2.],
       [ 0., -3., -8.]])
```

Allgemein: Subtraktion des k-fachen der i-ten Zeile von der j-ten Zeile entspricht Linksmultiplikation mit I - k e_j e_iˆT

Einfachste Form der LR Zerlegung (ohne Zeilentausch)

```
def LR1(A):
    import numpy as np

    n = A.shape[0]
    L = np.eye(n)
    R = A.astype('float')

    for k in range(n - 1):
        for j in range(k+1,n):
            L[j,k] = R[j,k]/R[k,k]
            for i in range(k,n):
                R[j,i] = R[j,i]-L[j,k]*R[k,i]

    return (L,R)
```

Einfachste Form der LR Zerlegung (ohne Zeilentausch) mit 2 for-Schleifen

```
def LR2(A):
    import numpy as np

    n = A.shape[0]
    L = np.eye(n)
    R = A.astype('float')

    for k in range(n - 1):
        for j in range(k+1,n):
            L[j,k] = R[j,k]/R[k,k]
            R[j,k:] = R[j,k:]-L[j,k]*R[k,k:]

    return (L,R)
```

Vektorisierte Form von LR2

```
def LR3(A):
    import numpy as np

    n = A.shape[0]
    L = np.eye(n)
    R = A.astype('float')

    for k in range(n - 1):
        L[k+1:, k] = R[k+1:, k] / R[k, k]
        R[k+1:, :]  = R[k+1:, :] - L[k+1:, [ k ]] * R[[ k ], :]
    return (L, R);
```

Vergleiche die Performance der drei Varianten

```
import time

A = np.random.uniform(0, 1, (100, 100))

t0 = time.clock()
L,R = LR1(A)
t1 = time.clock()
print(t1-t0)

print(np.max(A-L@R)) # Probe



A = np.random.uniform(0, 1, (100, 100))
t0 = time.clock()
L,R = LR2(A)
t1 = time.clock()
print(t1-t0)


print(np.max(A-L@R))  # Probe


A = np.random.uniform(0, 1, (100, 100))
t0 = time.clock()
```

```
L,R = LR3(A)
t1 = time.clock()
print(t1-t0)

print(np.max(A-L@R))   # Probe
```

```
0.1800000000000015
5.44120304369e-13
0.09999999999999964
8.74378347504e-12
0.019999999999999574
7.2497563508e-14
```