```python
#!/usr/bin/env python3
# -*- coding: utf-8 -*-
# """
# Created on Mon Dec 11 17:44:18 2017
#
# @author: christianehelzel
# """
```

1 10. Vorlesung: Die QR Zerlegung

Ziel: Finde eine Zerlegung der Matrix A der Form

```
A = QR
```

```
mit Q orthogonale bzw. unitäre Matrix und R obere Dreiecksmatrix
```

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

A = np.array([ [ -1, 1, 4, -1], [3, 8, 1, -4], [7., 3, -1, 2], \
              [2, -4, -1, 6], [1,0,1,0] ])
```

reduzierte QR Zerlegung (default)

```python
Q,R = npl.qr(A)
```

vollständige QR Zerlegung

```python
Q,R = npl.qr(A,mode='complete')
```

Wiederholung: Wir verwenden wieder unsere Funktion SolveR

```python
# Löse R x = b
def SolveR(R,b):
    import numpy as np

    n = len(b)
    x = np.zeros(n)

    x[-1] = b[-1]/R[-1,-1]
    for k in range(n-2,-1,-1):
        x[k] = (b[k]-np.sum(R[k,k+1:]*x[k+1:]))/R[k,k]
    return x
```

1.1 Löse lineares Gleichungssystem Ax = b unter Verwendung der QR Zerlegung

Löse lineares Gleichungssystem A x = b

Beispiel:

10 x - 7 y = 7

-3x + 2y + 6z = 4

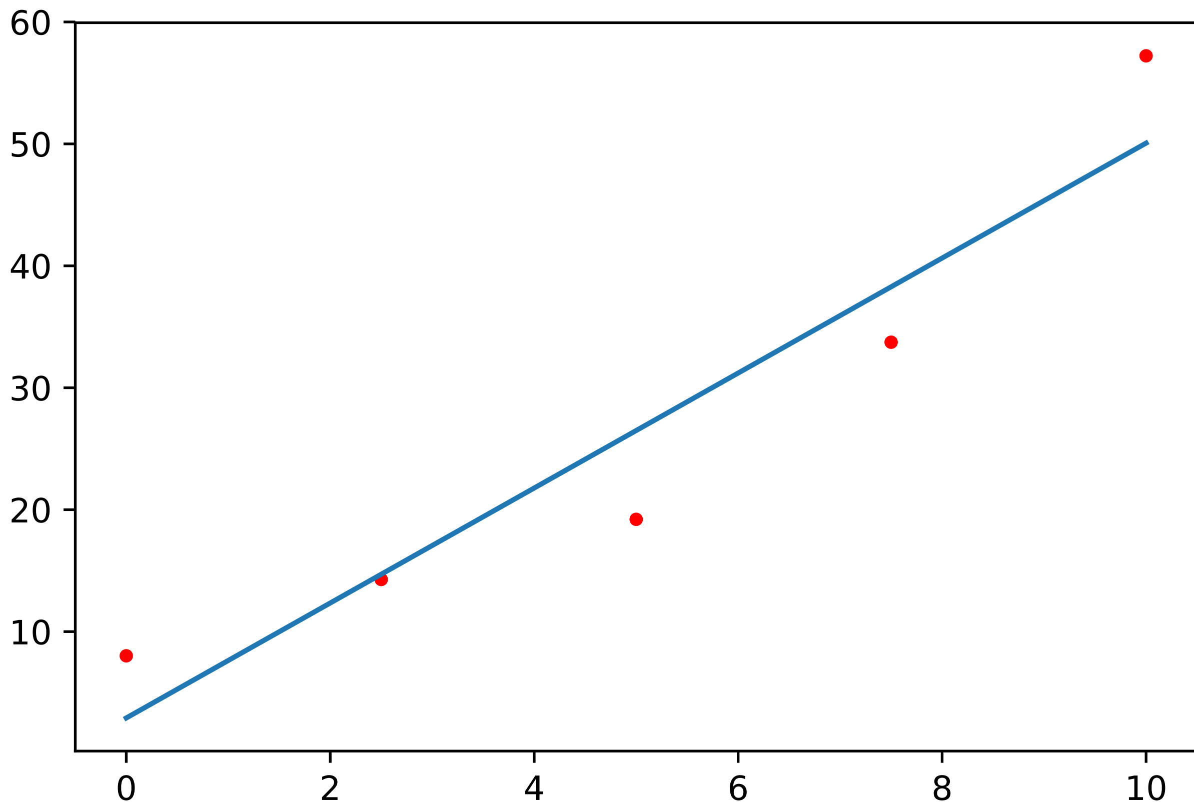
5x - y + 5z = 6

```
import numpy as np

A = np.array([[10., -7, 0], [-3, 2, 6], [5,-1,5]])
b = np.array([7., 4, 6])

Q,R = npl.qr(A)
x = SolveR(R,Q.T@b)
```

Die QR Zerlegung einer mxn Matrix A mit m>n kann benutzt werden um das überbestimmte LGS Ax = b im Sinne der Methode der kleinsten Fehlerquadrate zu lösen

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

x = np.linspace(0, 10, 5)
A = np.vstack([x, np.ones(len(x))]).T
m,n = A.shape

Q,R = npl.qr(A)
# liefert mxn (Spalten) orthogonale Matrix Q
# nxn obere Dreiecksmatix R

# b ist rechte Seite des linearen Gleichungssystems
f = np.poly1d([5, 1])
b = f(x) + 6*np.random.normal(size=len(x))

# Löse  Ax=b unter Verwendung der QR-Zerlegung
# d.h. löse R x = Q.T b

coeff = SolveR(R,Q.T@b)
lsf = coeff[0]*x+coeff[1]

plt.close('all')
plt.figure()
plt.plot(x,b,'r.')
plt.plot(x, lsf)
```

```
[<matplotlib.lines.Line2D at 0x7f17bf7b6860>]
```



Beachte: dieser Ansatz ist weniger anfällig gegenüber Rundungsfehler als die Lösung von A.T A x = A.T b

Berechne QR Zerlegung unter Verwendung von Gram-Schmidt

```
def GramSchmidt(A):
    import numpy as np
    import numpy.linalg as npl

    m, n = A.shape;
    Q = np.zeros([m,n])      # Null-Array der Größe mxn
    R = np.zeros([n,n])      # Null-Array der Größe nxn
    for j in range(n):
        Q[:,j] = A[:,j];
        # Orthogonalisiere A[:,j] gegen Q[:,1], ...,Q[:,j-1]
        for i in range(j):
            R[i,j]=Q[:,i].T@A[:,j]
            Q[:,j]=Q[:,j]-R[i,j]*Q[:,i]
        # Normiere Q[:,j]
        R[j,j]=npl.norm(Q[:,j])
        Q[:,j]=Q[:,j]/R[j,j]
    return Q, R
```

Berechne QR Zerlegung unter Verwendung des modifizierten Gram-Schmidt Verfahrens

```
def ModGramSchmidt(A):
    import numpy as np
    import numpy.linalg as npl

    m, n = A.shape;
```

```
    Q = np.zeros([m,n])      # Null-Array der Größe mxn
    R = np.zeros([n,n])      # Null-Array der Größe nxn

    for j in range(n):
        Q[:,j] = A[:,j];
        # Orthogonalisiere A[:,j] gegen Q[:,1], ...,Q[:,j-1]
        for i in range(j):
            # Berechne die Koeffizienten R[i,j] mit aktuellem Q[:,j] statt A[:,j]
            R[i,j]=Q[:,i].T@Q[:,j]
            Q[:,j]=Q[:,j]-R[i,j]*Q[:,i]
        # Normiere Q[:,j]
        R[j,j]=npl.norm(Q[:,j])
        Q[:,j]=Q[:,j]/R[j,j]
    return Q, R
```

2. Genauigkeitstest

```
import numpy as np

d = 1E-8
A = np.array([[1.,1.,1.],[d,0.,0.],[0.,d,0.],[0.,0.,d]])
Q1,R1 = GramSchmidt(A)
print(Q1.T@Q1)
Q2,R2 = ModGramSchmidt(A)
print(Q2.T@Q2)
```

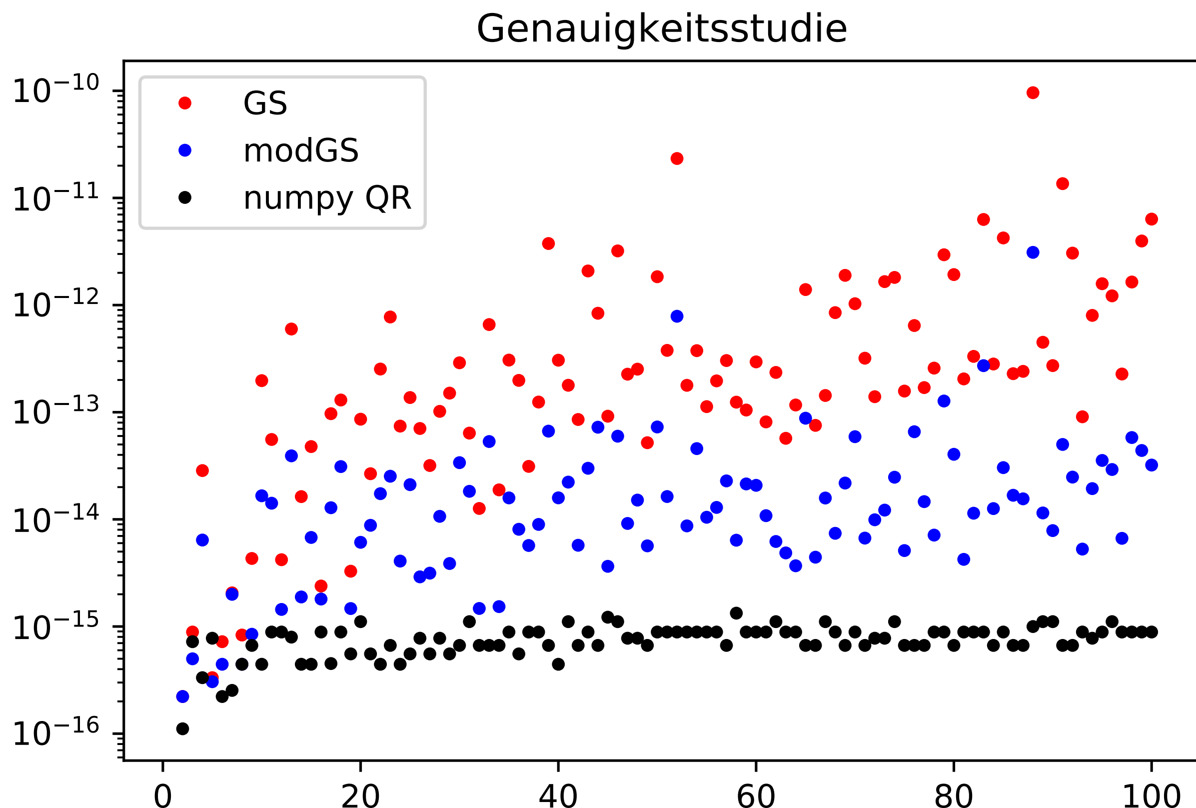
```
[[  1.00000000e+00  -7.07106781e-09  -7.07106781e-09]
 [ -7.07106781e-09   1.00000000e+00   5.00000000e-01]
 [ -7.07106781e-09   5.00000000e-01   1.00000000e+00]]
[[  1.00000000e+00  -7.07106781e-09  -4.08248290e-09]
 [ -7.07106781e-09   1.00000000e+00   1.11022302e-16]
 [ -4.08248290e-09   1.11022302e-16   1.00000000e+00]]
```

2. Genauigkeitstest

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

n = 100
errGS = np.zeros([n])
errMGS = np.zeros([n])
errQR = np.zeros([n])
dim = np.zeros([n])

for i in range(100):
    A = np.random.uniform(0, 1, (i+1, i+1))
    Q1,R1 = GramSchmidt(A)
    errGS[i] = np.max(np.abs(Q1.T@Q1-np.eye(i+1)))
    Q2,R2 = ModGramSchmidt(A)
    errMGS[i] = np.max(np.abs(Q2.T@Q2-np.eye(i+1)))
    Q3,R3 = npl.qr(A)
    errQR[i] = np.max(np.abs(Q3.T@Q3-np.eye(i+1)))
    dim[i] = i+1

plt.close('all')
```

```
plt.figure()

plt.semilogy(dim,errGS,'r.')
plt.semilogy(dim,errMGS,'b.')
plt.semilogy(dim,errQR,'k.')
plt.legend(['GS', 'modGS','numpy QR'])
plt.title('Genauigkeitsstudie')
```

```
Text(0.5,1,'Genauigkeitsstudie')
```



Weitere Informationen zur QR-Zerlegung und zu Gram-Schmidt finden Sie in den Kapiteln 7 und 8 des Buches "Numerical Linear Algebra" (Trefethen und Bau) (siehe Vorlesungsseite)

1.2 QR Algorithm: iterativer Algorithmus zur Berechnung der Eigenwerte von A

Für symmetrische Matrizen konvergiert die iterativ berechnete Matrix (Anew) gegen eine Diagonalmatrix. Da Anew und A selbstähnlich sind, liefern die
Diagonalelemente von Anew eine Approximation an die Eigenwerte von A

```
A = np.array([[1.,3.,4.],[3.,1.,2.],[4.,2.,1.]])
nit = 100

Anew = A.copy()
for i in range(nit):
    Q,R = npl.qr(Anew)
    Anew = R@Q
```

für symmetrische Matrizen konvergiert der Prozess gegen eine Diagonalmatrix

```
eval_estimate = np.diag(Anew)
```

Zum Vergleich: Berechne EWe mit npl.eig

```
eval_ref,evec = npl.eig(A)
```

Weitere Informationen zu diesem Algorithmus finden Sie in Kapitel 28 des Buches "Numerical Linear Algebra" (Trefethen und Bau) (siehe Vorlesungsseite)