QR-Zerlegung

October 27, 2018

```
In [1]: %%latex
        \parindent0cm
```

1 QR-Zerlegung

Als Alternative zu LR und Cholesky betrachten wir die QR-Zerlegung. Hierbei gilt $A = QR$, wobei Q eine spaltenorthonormale Matrix ist mit

$$Q^t\, Q = I$$

und R eine rechte obere Dreiecksmatrix.

1.1 Unitäre und spaltenorthonormale Matrizen

In der Vorlesung wurden einige unitäre Matrizen vorgestellt. Wir beginnen mit Spiegelmatrizen.

1.1.1 Spiegelmatrizen (Householdermatrizen)

Wir erzeugen einen zufälligen Vektor der Länge 1 und erzeugen die zugehörige Spiegelmatrix Q. Wir testen, ob Q unitär ist.

```
In [2]: import numpy as np
        import math
        import sympy
        from sympy import Matrix
        sympy.init_printing(use_unicode=True)
```

```
In [3]: N=4
        v=np.random.uniform(-1,1,[N,1])
        v=v/np.linalg.norm(v)
        Q=np.eye(N)-2*v.dot(v.T)
        print(np.linalg.norm(Q.T.dot(Q)-np.eye(N)))
```

```
6.547610293997335e-16
```

$Qv = -v$.

```
In [4]: print(v)
        print(np.matmul(Q,v))
```

```
[[-0.59142607]
 [ 0.61675758]
 [-0.3799785 ]
 [ 0.35417741]]
[[ 0.59142607]
 [-0.61675758]
 [ 0.3799785 ]
 [-0.35417741]]
```

Für zu v orthogonale Vektoren w gilt $Qw = w$.

```
In [5]: w=np.random.uniform(-1,1,[N,1])
        w=w/np.linalg.norm(w)
        w=w-w.T.dot(v)*v
        print(w)
        print(Q.dot(w))
```

```
[[-0.62909348]
 [-0.21996962]
 [-0.06221536]
 [-0.73419379]]
[[-0.62909348]
 [-0.21996962]
 [-0.06221536]
 [-0.73419379]]
```

1.1.2 Rotationsmatrizen (Givens-Rotationen)

Rotationsmatrix zu einem zufälligen Winkel φ.

```
In [6]: N=4
        phi=np.random.uniform(0,2*math.pi)
        Q=sympy.eye(N)
        c=sympy.Symbol("c")
        s=sympy.Symbol("s")
        Q[0,0]=c
        Q[0,1]=-s
        Q[1,0]=s
        Q[1,1]=c
        Matrix(Q)
```

```
Out[6]:
```

$$\begin{bmatrix} c & -s & 0 & 0 \\ s & c & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

In [7]:
```
Matrix(Q.T*Q)
```

Out[7]:

$$\begin{bmatrix} c^2+s^2 & 0 & 0 & 0 \\ 0 & c^2+s^2 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Linksmultiplikation mit einer Rotationsmatrix erzeugt eine Linearkombination der ersten beiden Zeilen.

In [8]:
```
A=Matrix(sympy.MatrixSymbol('A',N,N))
Matrix(Q*A)
```

Out[8]:

$$\begin{bmatrix} cA_{0,0}-sA_{1,0} & cA_{0,1}-sA_{1,1} & cA_{0,2}-sA_{1,2} & cA_{0,3}-sA_{1,3} \\ cA_{1,0}+sA_{0,0} & cA_{1,1}+sA_{0,1} & cA_{1,2}+sA_{0,2} & cA_{1,3}+sA_{0,3} \\ A_{2,0} & A_{2,1} & A_{2,2} & A_{2,3} \\ A_{3,0} & A_{3,1} & A_{3,2} & A_{3,3} \end{bmatrix}$$

Rechtsmultiplikation erzeugt eine Linearkombination der ersten beiden Spalten.

In [9]:
```
Matrix(A*Q)
```

Out[9]:

$$\begin{bmatrix} cA_{0,0}+sA_{0,1} & cA_{0,1}-sA_{0,0} & A_{0,2} & A_{0,3} \\ cA_{1,0}+sA_{1,1} & cA_{1,1}-sA_{1,0} & A_{1,2} & A_{1,3} \\ cA_{2,0}+sA_{2,1} & cA_{2,1}-sA_{2,0} & A_{2,2} & A_{2,3} \\ cA_{3,0}+sA_{3,1} & cA_{3,1}-sA_{3,0} & A_{3,2} & A_{3,3} \end{bmatrix}$$

Konjugation:

In [10]:
```
Matrix(Q.T*A*Q)
```

Out[10]:

$$\begin{bmatrix} c\left(cA_{0,0}+sA_{1,0}\right)+s\left(cA_{0,1}+sA_{1,1}\right) & c\left(cA_{0,1}+sA_{1,1}\right)-s\left(cA_{0,0}+sA_{1,0}\right) & cA_{0,2}+sA_{1,2} & cA_{0,3}+sA_{1,3} \\ c\left(cA_{1,0}-sA_{0,0}\right)+s\left(cA_{1,1}-sA_{0,1}\right) & c\left(cA_{1,1}-sA_{0,1}\right)-s\left(cA_{1,0}-sA_{0,0}\right) & cA_{1,2}-sA_{0,2} & cA_{1,3}-sA_{0,3} \\ cA_{2,0}+sA_{2,1} & cA_{2,1}-sA_{2,0} & A_{2,2} & A_{2,3} \\ cA_{3,0}+sA_{3,1} & cA_{3,1}-sA_{3,0} & A_{3,2} & A_{3,3} \end{bmatrix}$$

Durch Permutation kann man die betroffenen Zeilen/Spalten auswählen.

```
In [11]: np.random.seed(1)
         sigma1=np.random.permutation(N)
         print(sigma1)
         P1=sympy.zeros(N)
         for i in sigma1:
             P1[i,sigma1[i]]=1
         Q1=P1.T*Q*P1
         Matrix(Q1)
```

```
[3 2 0 1]
```

Out[11]:

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & c & s \\ 0 & 0 & -s & c \end{bmatrix}$$

```
In [12]: Matrix(Q1*A)
```

Out[12]:

$$\begin{bmatrix} A_{0,0} & A_{0,1} & A_{0,2} & A_{0,3} \\ A_{1,0} & A_{1,1} & A_{1,2} & A_{1,3} \\ cA_{2,0}+sA_{3,0} & cA_{2,1}+sA_{3,1} & cA_{2,2}+sA_{3,2} & cA_{2,3}+sA_{3,3} \\ cA_{3,0}-sA_{2,0} & cA_{3,1}-sA_{2,1} & cA_{3,2}-sA_{2,2} & cA_{3,3}-sA_{2,3} \end{bmatrix}$$

1.2 Gram-Schmidtsches Orthonormalisierungsverfahren

Seien $v_1 \dots v_m \in \mathbb{R}^n$ linear unabhängig.
Nach Vorlesung liefert das Schmidtsche Orthonormalisierungsverfahren ein orthonormales System $q_1 \dots q_m$ mit

$$\text{span}(v_1 \dots v_j) \subset \text{span}(q_1 \dots q_j),\ j \leq m.$$

In der folgenden Implementation nehmen wir an, dass die Bedingung der linearen Unabhängigkeit für unsere Matrix erfüllt ist.

```
In [13]: def GramSchmidt(V):
             (N,M)=V.shape
             Q=V.copy()
             R=np.zeros([M,M])
             for i in range(0,M):
                 for j in range (0,i):
                     R[j,i]=np.dot(Q[:,i],Q[:,j])
                     Q[:,i]=Q[:,i]-R[j,i]*Q[:,j]
                 R[i,i]=np.linalg.norm(Q[:,i])
                 Q[:,i]=Q[:,i]/R[i,i]
             return (Q,R)
```

Wir testen an einer zufälligen Matrix, die mit Wahrscheinlichkeit 1 invertierbar ist.

```
In [14]: N=200
         M=100
         V=np.random.uniform(-1,1,[N,M])
         (Q,R)=GramSchmidt(V)
         print(np.linalg.norm(Q.dot(R)-V))
         print(np.linalg.norm(Q.T.dot(Q)-np.eye(M)))
```

```
2.6021156379459125e-14
5.561406560876977e-15
```

Lösen eines Gleichungssystems durch Rückwärtseinsetzen.

```
In [15]: def qrsolve(Q,R,b):
             N=A.shape[1]
             b=Q.T.dot(b)
             xn=np.zeros([N,1])
             for i in range(N-1,-1,-1):
                 for k in range(i+1,N):
                     b[i]=b[i]-R[i,k]*xn[k]
                 xn[i,0]=b[i]/R[i,i]
             return xn
```

Wieder Test an einem zufälligen Gleichungssystem.

```
In [16]: N=100
         A=np.random.uniform(-1,1,[N,N])
         (Q,R)=GramSchmidt(A)
         x=np.ones([N,1])
         b=A.dot(x)
         xn=qrsolve(Q,R,b)
         print(np.linalg.norm(xn-x))
```

```
6.602642500214355e-12
```

Unser Gegenbeispiel für die Stabilität in der Gausselimination ohne Pivotsuche funktioniert hier problemlos.

```
In [17]: eps=1e-20
         A=np.array([[eps,1],[1,1]])
         (Q,R)=GramSchmidt(A)
         x=np.ones([2,1])
         b=A.dot(x)
         xn=qrsolve(Q,R,b)
         print(np.linalg.norm(xn-x))
```

```
0.0
```

1.3 Householder-Verfahren

Aber wir können die Stabilität noch weiter verbessern. Hierzu nutzen wir die Householder-Matrizen. Wir zeigen das Prinzip zunächst für die erste Spalte a einer Matrix. Wir suchen eine Spiegelmatrix Q zu einem Spiegelvektor v, so dass Qa ein Vielfaches des Einheitsvektors e_1 ist. Q ist unitär, ändert also die Länge von a nicht. Wir suchen also einen Vektor v, für den gilt

$$Qa = (I - 2vv^t)a = a - 2(v^t a)\, v = \sigma||a||e_1,\ \sigma = \pm 1.$$

Also muss also gelten

$$v = \frac{1}{2(v^t a)}\left(a - \sigma||a||e_1\right).$$

Wir setzen also

$$\widetilde{v} = a - \sigma||a||e_1,\ v = \frac{\widetilde{v}}{||\widetilde{v}||}$$

. Für dieses v gilt tatsächlich

$$Qv = \sigma||a||e_1.$$

Wir implementieren und testen.

```
In [18]: N=3
         M=N
         A=np.random.uniform(-1,1,[N,M])
         a=A[:,0:1]
         e0=np.eye(N)[:,0:1]
         sigma=1
         vtilde=a-sigma*np.linalg.norm(a)*e0
         v=vtilde/np.linalg.norm(vtilde,2)
         Q=np.eye(N)-2*v.dot(v.T)
         Q.T.dot(A)

Out[18]: array([[ 8.06816028e-01, -4.90976708e-02, -1.59583554e-02],
                [ 2.22044605e-16, -1.20728630e+00, -4.15207762e-01],
                [ 1.38777878e-16, -4.57725240e-01, -1.03717120e-01]])
```

Unterhalb der Hauptdiagonalen in der ersten Spalte stehen also jetzt Nullen. Wir setzen dies für die restlichen Spalten entsprechend fort.

```
In [19]: def householder(A):
             A=A.copy()
             (N,M)=A.shape
             Q1=np.eye(N)
             for i in range(0,M-1):
                 a=A[i:N,i:i+1]
                 e=np.eye(N)[i:N,i:i+1]
                 sigma=1
                 vtilde=a-sigma*np.linalg.norm(a)*e
                 v1=vtilde/np.linalg.norm(vtilde,2)
                 v=np.zeros([N,1])
```

```
            v[i:N]=v1
            Q=np.eye(N)-2*v.dot(v.T)
            A=Q.dot(A)
            Q1=Q.dot(Q1)
        return (Q1.T,A)
```

Wir testen wieder, ob das Ergebnis eine orthogonale Matrix liefert und ob $QR = A$.

```
In [20]: N=100
         M=80
         A=np.random.uniform(-1,1,[N,M])
         (Q,R)=householder(A)
         print(np.linalg.norm(Q.dot(R)-A))
         print(np.linalg.norm(Q.T.dot(Q)-np.eye(N)))
```

```
8.926006762939312e-14
2.2067607829963666e-14
```

Wir testen das Rückwärtseinsetzen an einem Gleichungssystem.

```
In [21]: N=100
         M=100
         A=np.random.uniform(-1,1,[N,M])
         (Q,R)=householder(A)
         x=np.ones([N,1])
         b=A.dot(x)
         xn=qrsolve(Q,R,b)
         print(np.linalg.norm(xn-x))
```

```
4.447136598485395e-13
```

Abschließend noch die Frage: Warum könnte das stabiler sein als Gram-Schmidt?
Wir betrachten die Zeile, in der $\widetilde{v}$ berechnet wird. Dabei ändert sich a nur in der ersten Zeile. Hier könnte Auslöschung entstehen. Wir verhindern sie, indem wir $\sigma = -\mathrm{sgn}(a_1)$ setzen. Die endgültige Version ist also:

```
In [22]: def householder_stabil(A):
             A=A.copy()
             (N,M)=A.shape
             Q1=np.eye(N)
             for i in range(0,M-1):
                 a=A[i:N,i:i+1]
                 e=np.eye(N)[i:N,i:i+1]
                 sigma=-np.sign(a[1])
                 vtilde=a-sigma*np.linalg.norm(a)*e
                 v1=vtilde/np.linalg.norm(vtilde,2)
                 v=np.zeros([N,1])
```

```
            v[i:N]=v1
            Q=np.eye(N)-2*v.dot(v.T)
            A=Q.dot(A)
            Q1=Q.dot(Q1)
        return (Q1.T,A)
```

Wir testen, ob für unser Beispiel das einen Unterschied macht.

```
In [23]: N=256
         M=256
         np.random.seed(1003)
         A=np.random.uniform(-1,1,[N,M])
         (Q,R)=householder(A)
         (Q1,R1)=householder_stabil(A)
         (Q2,R2)=GramSchmidt(A)
         x=np.ones([N,1])
         b=A.dot(x)
         xn=qrsolve(Q,R,b)
         xn1=qrsolve(Q1,R1,b)
         xn2=qrsolve(Q2,R2,b)
         print(np.linalg.norm(xn-x))
         print(np.linalg.norm(xn1-x))
         print(np.linalg.norm(xn2-x))
```

```
5.342467475329372e-12
1.7650773521011535e-12
1.519794651291005e-11
```

Immerhin bekommt man hier mal einen Faktor 3. Tatsächlich sind aber zufällige Matrizen für eine Demonstration dieses Effekts ungeeignet.
Hinweis: Diese Implementation ist maximal ineffizient und dient ausschließlich der Veranschaulichung.

1.4 Orthogonalisierung durch Givens–Rotationen

Wir deuten hier nur kurz an, dass man auch mit Givens–Rotationen eine QR-Zerlegung berechnen kann. Wir erzeugen wieder eine zufällige Matrix A und wählen eine Rotation Q so, dass QA an der Stelle $(2,1)$ eine 0 erhält.
Nach Gleichung (7) muss also gelten:

$$s\,A_{0,0} + c\,A_{1,0} = 0,\; c^2 + s^2 = 1.$$

Hieraus folgt sofort

$$c^2 = \frac{A_{0,0}^2}{A_{0,0}^2 + A_{1,0}^2},\; s^2 = \frac{A_{1,0}^2}{A_{0,0}^2 + A_{1,0}^2}$$

Wir wählen noch die Vorzeichen geeignet, und die Gleichungen sind erfüllt.

```
In [24]: N=3
         M=N
         A=np.random.seed(1)
         A=np.random.uniform(-1,1,[N,M])
         l=math.sqrt(A[0,0]*A[0,0]+A[1,0]*A[1,0])
         c=A[0,0]/l
         s=-A[1,0]/l
         S=np.eye(N)
         S[0,0]=c
         S[0,1]=-s
         S[1,0]=s
         S[1,1]=c
         print(S.dot(A))
```

```
[[ 0.42875522  0.48085963  1.13874661]
 [ 0.          0.67975815 -0.60625902]
 [-0.62747958 -0.30887855 -0.20646505]]
```

Wir halten fest: Durch Givens-Rotationen können in einer Matrix an vorgegebenen Stellen einfach Nullen erzeugt werden. Durch Vorgehen wie bei Householder können nach und nach unterhalb der Hauptdiagonalen Nullen erzeugt werden, und es entsteht wie dort eine QR-Zerlegung.

1.5 Hessenbergform

Wir werden mit Hilfe der QR-Zerlegung Eigenwerte von Matrizen berechnen. Ein Problem dabei ist, dass sich die Eigenwerte ändern, wenn man von A z.B. zu $R = Q^t A$ übergeht. Es reicht deshalb nicht, einfach nur die Eigenwerte von R zu berechnen.
Wir würden gern so etwas setzen wie $R = Q^t AQ$. In diesem Fall wären R und A ähnlich und hätten dieselben Eigenwerte, wir müssten nur noch die Eigenwerte von R berechnen. Wir untersuchen zunächst mal, was Householder in diesem Fall tut. Wir kopieren den Code von oben.

```
In [25]: N=3
         M=N
         A=np.random.uniform(-1,1,[N,M])
         a=A[:,0:1]
         e0=np.eye(N)[:,0:1]
         sigma=1
         vtilde=a-sigma*np.linalg.norm(a)*e0
         v=vtilde/np.linalg.norm(vtilde,2)
         Q=np.eye(N)-2*v.dot(v.T)
         Q.T.dot(A)
```

```
Out[25]: array([[ 6.86773277e-01, -7.51253501e-01,  9.13686445e-01],
                [-5.55111512e-17,  1.84059092e-01, -4.18069774e-01],
                [ 5.55111512e-17,  1.64632015e-01, -1.86675464e-01]])
```

Das hatten wir oben schon gesehen: Bei dieser Wahl ist $Q^t A$ eine Matrix mit Nullen unterhalb der Hauptdiagonalen in der ersten Spalte. Wir wenden nun noch von rechts Q an.

```
In [26]: Q.T.dot(A).dot(Q)
```

```
Out[26]: array([[ 1.17780771, -0.27476479,  0.6388549 ],
                [-0.36595939, -0.17105963, -0.21324262],
                [-0.23436766, -0.06279307, -0.05550009]])
```

Das war zu erwarten. Die Rechtsmultiplikation wirkt auf die Spalten von A, und damit wird unsere einfache Form wieder zerstört.
Wir wählen $A_{0,0} = 1$ und ansonsten die erste Spalte 0. Dann tritt dieses Problem nicht auf. Wir wählen für die Householder-Vektoren also $v_0 = 0$.

```
In [27]: N=3
         A=Matrix(sympy.MatrixSymbol('A',N,N))
         V1=Matrix(sympy.MatrixSymbol('v',N,1))
         V=sympy.zeros(N,1)
         for i in range(1,N):
             V[i]=V1[i]
         Q=sympy.eye(N)-2*V*V.T
         Matrix(Q)
```

Out[27]:

$$\begin{bmatrix} 1 & 0 & 0 \\ 0 & -2v_{1,0}^2 + 1 & -2v_{1,0}v_{2,0} \\ 0 & -2v_{1,0}v_{2,0} & -2v_{2,0}^2 + 1 \end{bmatrix}$$

```
In [28]: Matrix(A*Q)
```

Out[28]:

$$\begin{bmatrix} A_{0,0} & \left(-2v_{1,0}^2 + 1\right) A_{0,1} - 2A_{0,2}v_{1,0}v_{2,0} & \left(-2v_{2,0}^2 + 1\right) A_{0,2} - 2A_{0,1}v_{1,0}v_{2,0} \\ A_{1,0} & \left(-2v_{1,0}^2 + 1\right) A_{1,1} - 2A_{1,2}v_{1,0}v_{2,0} & \left(-2v_{2,0}^2 + 1\right) A_{1,2} - 2A_{1,1}v_{1,0}v_{2,0} \\ A_{2,0} & \left(-2v_{1,0}^2 + 1\right) A_{2,1} - 2A_{2,2}v_{1,0}v_{2,0} & \left(-2v_{2,0}^2 + 1\right) A_{2,2} - 2A_{2,1}v_{1,0}v_{2,0} \end{bmatrix}$$

Ok, Rechtsmultiplikation lässt die erste Spalte unverändert. Wir schauen, was wir dann noch erreichen können.

```
In [29]: Matrix(Q.T*A)
```

Out[29]:

$$\begin{bmatrix} A_{0,0} & A_{0,1} & A_{0,2} \\ \left(-2v_{1,0}^2 + 1\right) A_{1,0} - 2A_{2,0}v_{1,0}v_{2,0} & \left(-2v_{1,0}^2 + 1\right) A_{1,1} - 2A_{2,1}v_{1,0}v_{2,0} & \left(-2v_{1,0}^2 + 1\right) A_{1,2} - 2A_{2,2}v_{1,0}v_{2,0} \\ \left(-2v_{2,0}^2 + 1\right) A_{2,0} - 2A_{1,0}v_{1,0}v_{2,0} & \left(-2v_{2,0}^2 + 1\right) A_{2,1} - 2A_{1,1}v_{1,0}v_{2,0} & \left(-2v_{2,0}^2 + 1\right) A_{2,2} - 2A_{1,2}v_{1,0}v_{2,0} \end{bmatrix}$$

Das Element links oben in der Ecke bleibt also unverändert.
Da durch die Multiplikation mit einer unitären Matrix sich die Länge nicht ändert, finden wir kein Q, so dass die letzten zwei Zeilen der ersten Spalte 0 werden. Wir wählen v so, dass zumindest ab der übernächsten Zeile alles 0 wird, also hier nur das Element in der linken unteren Ecke.
Wir modifizieren unser Householder-Programm minimal und testen wieder.

```
In [30]: def hessenberg(A):
             A=A.copy()
             B=A.copy()
             (N,M)=A.shape
             for i in range(0,M-2):
                 a=A[i+1:N,i:i+1]
                 e=np.eye(N)[i+1:N,i+1:i+2]
                 sigma=-np.sign(a[1])
                 vtilde=a-sigma*np.linalg.norm(a)*e
                 v1=vtilde/np.linalg.norm(vtilde,2)
                 v=np.zeros([N,1])
                 v[i+1:N]=v1
                 Q=np.eye(N)-2*v.dot(v.T)
                 A=Q.dot(A).dot(Q.T)
             return A

In [31]: N=4
         M=4
         A=np.random.uniform(-1,1,[N,M])
         H=hessenberg(A)
         print(H)
```

```
[[-7.19226123e-01 -8.45439777e-01 -6.47977466e-01  6.84684179e-01]
 [ 1.21379423e+00  6.06242168e-01  5.65198057e-01 -6.98498275e-01]
 [-5.40539703e-17 -1.18673968e+00  6.27113602e-02 -7.62645663e-01]
 [-1.26355138e-17 -5.55111512e-17 -3.50852748e-01 -8.78316887e-01]]
```

Wir haben nicht ganz geschafft, was wir wollten. Dies ist keine rechte obere Dreiecksmatrix. auf der unteren Nebendiagonalen stehen noch Elemente. Aber immerhin: Die Eigenwerte haben sich nicht geändert (aber natürlich die Eigenvektoren).

```
In [32]: print(np.linalg.eig(A))
         print(np.linalg.eig(H))
```

```
(array([-0.10986447+1.17541891j, -0.10986447-1.17541891j,
        0.44847481+0.j        , -1.15733535+0.j        ]), array([[ 0.57649201+0.j        ,  0.5
         0.49612493+0.j        , -0.03798286+0.j        ],
       [ 0.28031364-0.35020786j,  0.28031364+0.35020786j,
        -0.82461738+0.j        ,  0.08266526+0.j        ],
       [ 0.20120434+0.4466333j ,  0.20120434-0.4466333j ,
         0.25977377+0.j        , -0.80409407+0.j        ],
       [ 0.42660178+0.21090784j,  0.42660178-0.21090784j,
        -0.07989881+0.j        ,  0.58750019+0.j        ]]))
(array([-0.10986447+1.17541891j, -0.10986447-1.17541891j,
        0.44847481+0.j        , -1.15733535+0.j        ]), array([[-0.47970868+0.31972272j, -0.4
        -0.49612493+0.j        , -0.03798286+0.j        ],
       [ 0.60818012+0.j        ,  0.60818012-0.j        ,
        -0.12877083+0.j        ,  0.13555648+0.j        ],
```

```
       [ 0.10750532+0.51838388j,  0.10750532-0.51838388j,
         0.83011569+0.j        ,  0.61623076+0.j        ],
       [-0.12309839-0.04838847j, -0.12309839+0.04838847j,
        -0.21951326+0.j        ,  0.77488154+0.j        ]]))
```

Wir halten fest: Bei der Berechnung von Eigenwerten einer Matrix A gehen wir zunächst über zu einer Hessenbergmatrix H, die dieselben Eigenwerte hat, aber eine sehr einfache Form (fast rechte obere Dreiecksmatrix, nur eine weitere Nebendiagonale unten ist besetzt).
Und noch eine wichtige Eigenschaft: Symmetrie geht in diesem Algorithmus nicht verloren. Sei also A symmetrisch, dann ist auch H symmetrisch, und besteht damit aus der Hauptdiagonalen mit zwei besetzten (gleichen) Nebendiagonalen.

```
In [33]: N=4
         M=4
         A=np.random.uniform(-1,1,[N,M])
         A=A.T.dot(A)
         H=hessenberg(A)
         print(H)
```

```
[[ 1.93607255e+00  1.64938414e+00 -2.12605034e-16  1.16361346e-16]
 [ 1.64938414e+00  1.94655225e+00  9.34713432e-01 -2.77555756e-16]
 [-2.12605034e-16  9.34713432e-01  1.71730494e+00  1.59308018e-01]
 [ 1.16361346e-16  5.55111512e-17  1.59308018e-01  3.85244621e-01]]
```