Numerisk Lineær Algebra F2021
Notesæt 25

Andrew Swann

3. maj 2021

Sidst ændret: 3. maj 2021.
Versionskode: bde54f5.

Indhold

25 Hessenberg- og tridiagonalform

Vi begynder nu arbejdet for at beskrive metoder for at bestemme alle egenværdier og en basis af tilhørende egenvektor for diagonaliserbar $A \in \mathbb{C}^{n\times n}$ og for symmetrisk $A \in \mathbb{R}^{n\times n}$. Overordnet har disse metoder, og deres udvidelse til beregning af Schurform, to trin:

(a) bring A i en særlig form, og så

(b) brug en potensmetode.

Mere konkret for generel kompleks $A \in \mathbb{C}^{n\times n}$, er disse to trin

$$\underset{A}{\begin{bmatrix} * & * & * & * & * \\ * & * & * & * & * \\ * & * & * & * & * \\ * & * & * & * & * \\ * & * & * & * & * \end{bmatrix}} \to \underset{H = \text{Hessenberg}}{\begin{bmatrix} * & * & * & * & * \\ * & * & * & * & * \\ 0 & * & * & * & * \\ 0 & 0 & * & * & * \\ 0 & 0 & 0 & * & * \end{bmatrix}} \to \underset{S = \text{Schurform}}{\begin{bmatrix} \lambda_0 & * & * & * & * \\ 0 & \lambda_1 & * & * & * \\ 0 & 0 & \lambda_2 & * & * \\ 0 & 0 & 0 & \lambda_3 & * \\ 0 & 0 & 0 & 0 & \lambda_4 \end{bmatrix}}$$

Hessenbergform er lidt svagere end øvretriangulær, da der er nogle elementer lige under diagonalen, som kan være forskellig fra nul. Mere præcist er en matrix i *Hessenbergform* $H = (h_{ij})$ har $h_{ij} = 0$ for alle element under underdiagonalen, dvs. for alle $i > j + 1$.

For generel reel $A \in \mathbb{R}^{n\times n}$, kan man også reducere til Hessenbergform, og bagefter til en reel version af Schurformen, med blokke af størrelse højst 2×2 langs diagonalen. I situationen hvor $A = A^T \in \mathbb{R}^{n\times n}$ er reel symmetrisk, eller $A = \overline{A}^T \in \mathbb{C}^{n\times n}$ kompleks hermitisk, er der garanti for at A er diagonaliserbar, og processerne kan forbedres til

$$\underset{A}{\begin{bmatrix} * & * & * & * & * \\ * & * & * & * & * \\ * & * & * & * & * \\ * & * & * & * & * \\ * & * & * & * & * \end{bmatrix}} \to \underset{T = \text{tridiagonal}}{\begin{bmatrix} * & * & 0 & 0 & 0 \\ * & * & * & 0 & 0 \\ 0 & * & * & * & 0 \\ 0 & 0 & * & * & * \\ 0 & 0 & 0 & * & * \end{bmatrix}} \to \underset{\Lambda}{\begin{bmatrix} \lambda_0 & 0 & 0 & 0 & 0 \\ 0 & \lambda_1 & 0 & 0 & 0 \\ 0 & 0 & \lambda_2 & 0 & 0 \\ 0 & 0 & 0 & \lambda_3 & 0 \\ 0 & 0 & 0 & 0 & \lambda_4 \end{bmatrix}}$$

Generelt har en *tridiagonal* matrix $T = (t_{ij})$ kun indgange, som er forskellige fra 0, på diagonalen, på overdiagonalen og på underdiagonalen, dvs. at $t_{ij} = 0$ for alle $|i - j| > 1$. I de overstående metoder opnås at T har samme egenskaber som A, så T er symmetrisk i det reelle tilfælde, og er hermitisk i det komplekse tilfælde. I dette kapitel vil vi fokusere på reelle matricer.

Vi ønsker at lave disse reduktioner så at A transformeres til VAV^{-1} for ortogonal (eller unitær) V. Vi har set at Householder matricer er meget nyttig og er gode eksempler på ortogonale matricer.

Vi kan altid vælger et Householder matrix $H_0 = H_0^T = H_0^{-1}$ således at H_0A

har kun nultal under diagonalen i den første søjle

$$A = \begin{bmatrix} a_{00} & * & \dots & * \\ a_{10} & * & \dots & * \\ \vdots & \vdots & \ddots & \vdots \\ a_{n-1,0} & * & \dots & * \end{bmatrix} \mapsto H_0 A = \begin{bmatrix} \pm\|a_0\|_2 & * & \dots & * \\ 0 & * & \dots & * \\ \vdots & \vdots & \ddots & \vdots \\ 0 & * & \dots & * \end{bmatrix}$$

men disse nultal bliver overskrevet når vi beregner

$$H_0 A H_0^{-1} = H_0 A H_0 = \begin{bmatrix} * & * & \dots & * \\ * & * & \dots & * \\ \vdots & \vdots & \ddots & \vdots \\ * & * & \dots & * \end{bmatrix},$$

da højre multiplikation med H_0 ændrer alle søjler.

Eksempel 25.1. Her er et eksempel i python. Vi starter med en tilfældig symmetrisk matrix $A \in \mathbb{R}^{4\times 4}$.

```
import numpy as np

rng = np.random.default_rng()

a = rng.standard_normal((4, 4))
a += a.T
print(a)
```

```
[[-1.83986249  1.13776313  3.90381649 -0.52352305]
 [ 1.13776313 -1.58453674  1.37471241 -2.22844686]
 [ 3.90381649  1.37471241 -2.68730646 -1.61066385]
 [-0.52352305 -2.22844686 -1.61066385 -0.1660438 ]]
```

Nu beregner vi Householder data for den 0'te søjle i a:

```
def house(x):
    norm_x = np.linalg.norm(x)
    if norm_x == 0:
        v = np.zeros_like(x)
        v[0] = 1
```

```
        s = 0
    else:
        u = x / np.linalg.norm(x)
        eps = -1 if u[0] >= 0 else +1
        s = 1 + np.abs(u[0])
        v = - eps * u
        v[0] += 1
        v /= s
    return v, s

v, s = house(a[:, [0]])
h0 = np.eye(4) - s * v @ v.T
print(h0 @ a)
```

```
[[ 4.49371322e+00  5.86844992e-01 -3.39717274e+00 -1.72975938e+00]
 [ 4.68375339e-17 -1.48556984e+00  2.68626164e+00 -2.01175865e+00]
 [ 3.19189120e-16  1.71428099e+00  1.81279325e+00 -8.67177783e-01]
 [ 0.00000000e+00 -2.27398486e+00 -2.21415172e+00 -2.65749326e-01]]
```

Vi ser at denne give tal tæt på 0 i den 0'te søjle underdiagonalen. Men $H_0AH_0^T = H_0AH_0$ har

```
print(h0 @ a @ h0)
```

```
[[-4.44098074  2.19187284  2.10989108 -2.46828663]
 [ 2.19187284 -1.87931773  1.33526029 -1.83058201]
 [ 2.10989108  1.33526029  0.51232287 -0.69277762]
 [-2.46828663 -1.83058201 -0.69277762 -0.46977388]]
```

som har ikke disse 0 tal. △

Vi har været for ambitiøst, med hvor mange nultal vi vil frembringe. Men ved at skrue lidt ned for ambitionerne kan vi godt bruge Householder matricer til at reducere til Hessenbergform.

25.1 Reduktion til Hessenbergform

Betragt en Householder matrix $H_0 = H_0^T = H_0^{-1} = I_n - svv^T$ af formen

$$H_0 = \begin{bmatrix} 1 & 0 & 0 & \dots & 0 \\ 0 & * & * & \dots & * \\ 0 & * & * & \dots & * \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & * & * & \dots & * \end{bmatrix},$$

dvs. med $v = (0, 1, *, \dots)$. Vælges v korrekt kan vi opnå at

$$\begin{bmatrix} a_{00} & a_{01} & a_{02} & \dots & a_{0,n-1} \\ a_{10} & * & * & \dots & * \\ a_{20} & * & * & \dots & * \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ a_{n-1,0} & * & * & \dots & * \end{bmatrix} \mapsto H_0A = \begin{bmatrix} a_{00} & a_{01} & a_{02} & \dots & a_{0,n-1} \\ * & * & * & \dots & * \\ 0 & * & * & \dots & * \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & * & * & \dots & * \end{bmatrix}.$$

Bemærk at den første række er uændret. Vektoren $v = (0, v_{[1:]})$ vælges, som i afsnit 9.4, således at $A_{[1:,0]} = (a_{10}, a_{20}, \dots, a_{n-1,0}) \in \mathbb{R}^{n-1}$ afbildes af $(H_0)_{[1:,1:]} = I_{n-1} - sv_{[1:]}v_{[1:]}^T$ til $(\pm\|A_{[1:,0]}\|_2, 0, \dots, 0)$. Når vi nu danner H_0AH_0 ved at gange H_0 fra højre på H_0A, bliver den første søjle fastholdt, og vi får

$$H_0AH_0 = \begin{bmatrix} a_{00} & * & * & \dots & * \\ * & * & * & \dots & * \\ 0 & * & * & \dots & * \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & * & * & \dots & * \end{bmatrix}.$$

Vælg nu Householdermatrix H_1 således at

$$\begin{bmatrix} a_{00} & * & * & * & * \\ * & * & * & * & * \\ 0 & * & * & * & * \\ 0 & * & * & * & * \\ 0 & * & * & * & * \end{bmatrix} \mapsto H_1H_0AH_0 = \begin{bmatrix} a_{00} & * & * & * & * \\ * & * & * & * & * \\ 0 & * & * & * & * \\ 0 & 0 & * & * & * \\ 0 & 0 & * & * & * \end{bmatrix}$$

og så

$$H_1H_0AH_0H_1 = \begin{bmatrix} a_{00} & * & * & * & * \\ * & * & * & * & * \\ 0 & * & * & * & * \\ 0 & 0 & * & * & * \\ 0 & 0 & * & * & * \end{bmatrix}.$$

Fortsættes på denne måde får vi den følgende algoritme

HESSENBERG($A \in \mathbb{R}^{n\times n}$)
1 $B = A$
2 **for** $k \in \{0, 1, \ldots, n-3\}$:
3 Beregn Householder H_k for $B_{[k+1:,k]}$
4 $B = H_kBH_k$
5 **return** B

Dette kan implementeres i python på den samme måde som for QR-dekomponering. Nemlig vi begynder med en funktion, der beregner den relevante data for en Hessenbergreduktion, men vi samler ikke Householderprodukterne, i stedet for gemmes data for Householdervektorerne under underdiagonalen og skalaerne s_k gemmes i en separat `np.array`:

```
def hessenberg_data(a):
    data = np.copy(a)
    n, _ = a.shape
    s = np.empty(n-2)
    for j in range(n-2):
        v, s[j] = house(data[j+1:, [j]])
        data[j+1:, j:] -= (s[j] * v) @ (v.T @ data[j+1:, j:])
        data[:, j+1:] -= (s[j] * (data[:, j+1:] @ v)) @ v.T
        data[j+2:, [j]] = v[1:]
    return data, s
```

Selve Hessenbergmatricen er nu blot delen af matricen `data`, der ligger på underdiagonalen og derover

```
def hessenberg(a):
    data, s = hessenberg_data(a)
    return np.triu(data, -1)
```

Ønsker man også en samlet matrix Q således at $A = QHQ^T$, kan den dannes ved at gange Householdermatricerne sammen

```
def hessenberg_qh(a):
    data, s = hessenberg_data(a)
    n, _ = a.shape
    h = np.triu(data, -1)
    q = np.eye(n)
    for j in reversed(range(n-2)):
        x = data[j+2:, [j]]
        v = np.vstack([[1], x])
        q[j+1:, j+1:] -= (s[j] * v) @ (v.T @ q[j+1:, j+1:])
    return q, h
```

Vi tester dette på en matrix:

```
a = np.array(np.arange(25),dtype=float).reshape(5,5)
print(a)
```

```
[[ 0.  1.  2.  3.  4.]
 [ 5.  6.  7.  8.  9.]
 [10. 11. 12. 13. 14.]
 [15. 16. 17. 18. 19.]
 [20. 21. 22. 23. 24.]]
```

```
q, h = hessenberg_qh(a)

print('Tjek q ortogonal:',
      np.allclose(q.T @ q, np.eye(5),
                  atol = np.finfo(float).eps))

print('Tjek dekomponering af a:',
      q @ h @ q.T)

print('Hessenbergmatrix')
print(h)
```

```
Tjek q ortogonal: True
Tjek dekomponering af a: [[ 0.  1.  2.  3.  4.]
 [ 5.  6.  7.  8.  9.]
 [10. 11. 12. 13. 14.]
 [15. 16. 17. 18. 19.]
 [20. 21. 22. 23. 24.]]
Hessenbergmatrix
[[ 0.00000000e+00 -5.47722558e+00  1.07698418e-15
  -7.82567726e-16 -5.12822540e-17]
 [-2.73861279e+01  6.00000000e+01  2.23606798e+01
  -1.37628386e-14  7.93849888e-15]
 [ 0.00000000e+00  4.47213595e+00 -4.73619917e-16
   1.76311711e-15  3.49592482e-15]
 [ 0.00000000e+00  0.00000000e+00 -2.44814982e-17
  -9.83087960e-16 -2.44879131e-15]
 [ 0.00000000e+00  0.00000000e+00  0.00000000e+00
  -8.52299065e-17 -2.08626660e-16]]
```

Reduktion til Hessenbergform uden beregning af Q bruger $\sim 10n^3/3$ flops.

25.2 Reduktion til tridiagonalform

Lad os gå videre og reducere fra Hessenbergform til tridiagonalform. Her begynder vi med en reel symmetrisk matrix $A = A^T \in \mathbb{R}^{n\times n}$. I princippet bruger vi den samme procedure, som for Hessenbergreduktion, men der er en væsentlig detalje når Householdermultiplikation skal implementeres.

Først vælger vi en Householdermatrix $H_0 = H_0^T = H_0^{-1}$ således at

$$\begin{bmatrix} a_{00} & a_{01} & a_{02} & \dots & a_{0,n-1} \\ a_{10} & * & * & \dots & * \\ a_{20} & * & * & \dots & * \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ a_{n-1,0} & * & * & \dots & * \end{bmatrix} \mapsto H_0A = \begin{bmatrix} a_{00} & a_{01} & a_{02} & \dots & a_{0,n-1} \\ * & * & * & \dots & * \\ 0 & * & * & \dots & * \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & * & * & \dots & * \end{bmatrix}$$

Denne gang når vi beregne $H_0AH_0 = H_0AH_0^T$, med A symmetrisk, får vi igen

en symmetrisk matrix. Så

$$H_0AH_0 = \begin{bmatrix} a_{00} & * & 0 & \dots & 0 \\ * & * & * & \dots & * \\ 0 & * & * & \dots & * \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & * & * & \dots & * \end{bmatrix}$$

og dette begynder at ligne den tridiagonalform vi ønsker.

Vi fortsætter ved at vælge en Householdermatrix H_1 så at

$$\begin{bmatrix} a_{00} & * & 0 & 0 & 0 \\ * & * & * & * & * \\ 0 & * & * & * & * \\ 0 & * & * & * & * \\ 0 & * & * & * & * \end{bmatrix} \mapsto H_1H_0AH_0 = \begin{bmatrix} a_{00} & * & 0 & 0 & 0 \\ * & * & * & * & * \\ 0 & * & * & * & * \\ 0 & 0 & * & * & * \\ 0 & 0 & * & * & * \end{bmatrix}$$

og får dermed

$$H_1H_0AH_0H_1 = \begin{bmatrix} a_{00} & * & 0 & 0 & 0 \\ * & * & * & 0 & 0 \\ 0 & * & * & * & * \\ 0 & 0 & * & * & * \\ 0 & 0 & * & * & * \end{bmatrix},$$

osv.

Det er væsentlig at bemærke at for en symmetrisk matrix A og en Householdermatrix $H = I_n - svv^T$ har vi

$$\begin{aligned} HAH &= (I_n - svv^T)A(I_n - svv^T) \\ &= A - svv^TA - sAvv^T + s^2vv^TAvv^T \\ &= A - svv^TA - sAvv^T + s^2(v^TAv)vv^T \\ &= A - v(sv^TA^T) - (sAv)v^T \\ &\qquad + \frac{s}{2}v(v^T(sAv))v^T + \frac{s}{2}((sAv)^Tv)vv^T \\ &= A - vw^T - wv^T, \end{aligned}$$

hvor

$$w = (sAv) - \frac{s}{2}\big((sAv)^Tv\big)v.$$

Vi kan derfor forkorte beregningen af HAH.

```
import numpy as np

def house_plus(x):
    norm_x = np.linalg.norm(x)
    if norm_x == 0:
        v = np.zeros_like(x)
        v[0] = 1
        s = 0
        eps = 1
    else:
        u = x / np.linalg.norm(x)
        eps = -1 if u[0] >= 0 else +1
        s = 1 + np.abs(u[0])
        v = - eps * u
        v[0] += 1
        v /= s
    return v, s, eps, norm_x

def tridiagonal_data(a):
    data = np.copy(a)
    if not np.allclose(a, a.T):
        raise np.linalg.LinAlgError(
            'In tridiagonal_data() input must ' +
            'be a symmetric matrix')
    n, _ = a.shape
    s = np.empty(n - 2)
    for j in range(n - 2):
        v, s[j], eps, norm = house_plus(data[j+1:, [j]])
        u = s[j] * (data[j+1:, j+1:] @ v)
        w = u - ((s[j]/2) * (u.T @ v)) * v
        v_wT = v @ w.T
        data[j+1, j] = eps * norm
        data[j, j+1] = data[j+1, j]
        data[j+1:, j+1:] -= v_wT + v_wT.T
        data[j+2:, [j]] = v[1:]
    return data, s
```

```
def tridiagonal_qt(a):
    data, s = tridiagonal_data(a)
    n, _ = a.shape
    t = np.tril(np.triu(data, -1), 1)
    q = np.eye(n)
    for j in reversed(range(n-2)):
        x = data[j+2:, [j]]
        v = np.vstack([[1], x])
        q[j+1:, j+1:] -= s[j] * v @ (v.T @ q[j+1:, j+1:])
    return q, t
```

Vi afprøver dette på en tilfældig symmetrisk matrix af størrelse (30×30):

```
rng = np.random.default_rng()

n = 30
a = rng.normal(0.0, 5.0, (n, n))
a = (a + a.T)/2
q, t = tridiagonal_qt(a)

print('Tjek q ortogonal:',
      np.allclose(q.T @ q, np.eye(n),
                  atol=2*np.finfo(float).eps))

print('Tjek dekomponering af a:',
      np.allclose(q @ t @ q.T, a,
                  atol=np.finfo(float).eps))
```

```
Tjek q ortogonal: True
Tjek dekomponering af a: True
```

Tridiagonalisering uden beregning af Q bruger $\sim 4n^3/3$ flops.

Python indeks

Indeks