Numerisk Lineær Algebra F2021
Notesæt 26

Andrew Swann

7. maj 2021

Sidst ændret: 7. maj 2021.
Versionskode: dd8a21f.

Indhold

Figurer

26 Egendekomponering via QR-metoden

Vi vil nu indføre QR-metoden for beregningen af samtlige egenværdier og egenvektorer af symmetriske reelle matricer.

26.1 QR-metoden: første version

Vi begynder med en først idé til hvordan QR-faktorisering kan bruges i forbindelse med beregning af egenværdier. Ideen er meget enkelt, men det er ikke oplagt hvorfor det skulle virke.

FØRSTE QR-METODE(A)
1 $A^{(0)} = A$
2 **for** $k \in \{1, 2, \ldots\}$:
3 Beregn QR-faktorisering $A^{(k-1)} = Q^{(k)}R^{(k)}$
4 Sæt $A^{(k)} = R^{(k)}Q^{(k)}$

Her har vi skrevet $A^{(k)}$ osv. for matricerne dannet ved den k'te iteration. Under visse antagelser konvergerer $A^{(k)}$ til en diagonalmatrix hvis $A \in \mathbb{R}^{n\times n}$ er reel symmetrisk. (Den kan også bruges til at producere en Schurform for $A \in \mathbb{R}^{n\times n}$ eller for $A \in \mathbb{C}^{n\times n}$.) I det reelle tilfælde vil vi se hvorfor og hvornår dette virker, ved at relatere metoden til en potensmetode.

Lad os først bemærke at

$$RQ = Q^T QRQ = Q^T AQ,$$

så matricerne $A = QR$ og $A^{(1)} = RQ$ har i hver fald de samme egenværdier. Det der er ikke oplaget er hvorfor $A^{(1)}$ er tættere på diagonalform end A.

Vi kan dog kikke på et eksempel. Betragt matricen

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

Denne matrix er symmetrisk

```
print(np.all(a.T == a))
```

```
True
```

Lad os prøve et første skridt af QR-metoden ovenfor

```
q, r = householder_qr(a)
a1 = r @ q
print(a1)
```

```
[[ 3.66666667 -1.40705294  0.86164044]
 [-1.40705294 -0.57575758 -0.25979437]
 [ 0.86164044 -0.25979437  1.90909091]]
```

Vi ser allerede at indgangene væk fra diagonalen er blevet mindre. Dette forstærkes med den næste iteration

```
q1, r1 = householder_qr(a1)
a2 = r1 @ q1
print(a2)
```

```
[[ 4.34020619  0.38439645  0.35812246]
 [ 0.38439645 -0.97233054  0.02577822]
 [ 0.35812246  0.02577822  1.63212435]]
```

Regner vi flere iterationer får vi

```
b = np.copy(a)
for i in range(10):
    q, r = householder_qr(b)
    b = r @ q

print(b)
```

```
[[ 4.41421356e+00  2.72594426e-06  1.01264816e-04]
 [ 2.72594427e-06 -1.00000000e+00  5.09658823e-11]
 [ 1.01264816e-04  5.09659469e-11  1.58578644e+00]]
```

som er rimelig tæt på en diagonalmatrix, så indgangene på diagonalen nærmere sig egenværdier for a. Dog er konvergens generelt ret langsomt. For eksempel, betragt denne matrix

```
a = np.array([[ 1.,  2., -1.],
              [ 2., -1.,  1.],
              [-1.,  1.,  3.]])
print(np.all(a == a.T))
```

```
True
```

```
b = np.copy(a)
for i in range(30):
    q, r = householder_qr(b)
    b = r @ q
    if i % 5 == 0:
        print('iteration:', i, ' b[0,1]:', b[0,1])

print('b:', b)
```

```
iteration: 0  b[0,1]: -2.028370211348441
iteration: 5  b[0,1]: 1.4089891237579977
iteration: 10  b[0,1]: -0.4002091153217211
iteration: 15  b[0,1]: 0.09838867659570037
iteration: 20  b[0,1]: -0.023833383264374487
iteration: 25  b[0,1]: 0.005761847932806463
b: [[ 3.42362157e+00  1.84994907e-03 -1.09531882e-05]
 [ 1.84994907e-03 -2.57699406e+00  2.11786853e-02]
 [-1.09531882e-05  2.11786853e-02  2.15337249e+00]]
```

Her tog det omtrent 10 iterationer at få `b[0,1]` reduceret med en faktor 10. Efter `30` iterationer har `b[0,1]` størrelsesorden 10^{-3} og man kan derfor kun forvente at diagonalindgangene giver egenværdierne inden for cirka 3 decimaler.

26.2 Ortogonal iteration

Vi kan udvide potensmetoden fra afsnit 24.1 til at beregne flere egenværdier og egenvektorer. Givet en symmetrisk matrix $A \in \mathbb{R}^{n\times n}$, lad os sige at vi ønsker at beregne de »første« m egenværdier og egenvektorer, hvor $m \leqslant n$.

Metoden begynder med en vilkårlig matrix

$$W = \left[w_0 \;\middle|\; w_1 \;\middle|\; \ldots \;\middle|\; w_{m-1} \right] \in \mathbb{R}^{n\times m}$$

med ortonormal søjler. I princippet vil vi gerne bare beregne $A^k W$ og håber at søjle konvergerer mod m egenvektorer. Men vi ved at dette kan ikke forventes, da fra potensmetoden ved vi at hver søjle $A^k w_i$ forventes $(A^k w_i)/\|A^k w_i\|_2$ at nærmere sig retningen for egenvektoren hørende til egenværdien, som er numerisk størst.

Vi kan ændre på denne forventning ved, at vælge en ortonormal basis for søjlerummet af $A^k W$. Dette kan gøres ved at beregne QR-dekomponeringen $A^k W = Q^{[k]} R^{[k]}$. Så udgør søjlerne af $Q^{[k]}$ denne ortonormal basis. (Vi bruger $Q^{[k]}$ her for at skelne mellem disse matricer og dem, der optræder i den faktiske ortoganal iteration nedenfor.)

Søjlerne af $Q^{[k]}$ fås ved at anvende en version af Gram-Schmidt processen på $A^k w_0, A^k w_1, \ldots, A^k w_{m-1}$. Så den første søjle af $Q^{[k]}$ er netop vektorerne vi få via potensmetoden på w_0 Den næste søjle af $Q^{[k]}$ er nødvendigvis vinkel ret på w_0, så må konvergerer mod en anden egenvektor.

Lad $v_0, v_1, \ldots, v_{n-1}$ være en ortonormal basis for A med egenværdier $\lambda_0, \lambda_1, \ldots, \lambda_{n-1}$. Hvis

$$|\lambda_0| > |\lambda_1| > \cdots > |\lambda_{m-1}| > |\lambda_m| \geqslant \cdots \geqslant |\lambda_{n-1}| \tag{26.1}$$

så kan analysen af potensmetoden udvides til at vise at for generisk w_j vil søjlerne $q_j^{[k]}$ af $Q^{[k]}$ konvergerer mod $\pm v_j$, for $j = 0, 1, \ldots, m-1$.

Processen kan gøres lidt mere stabil ved at vælge at arbejde med en ortonormal basis for søjlerummet af $A^k W$ i hvert trin. Dette giver følgende algoritme

ORTOGONAL ITERATION(A, m)
1 $W^{(0)} \in \mathbb{R}^{n\times m}$ tilfældig med ortonormale søjler
2 **for** $k \in \{1, 2, \ldots\}$:
3 $\quad V^{(k)} = AW^{(k-1)}$
4 $\quad Q^{(k)} R^{(k)} = V^{(k)}$
5 $\quad W^{(k)} = Q^{(k)}$

Formelt findes der $\varepsilon_i^{(k)} \in \{\pm 1\}$ således at søjlerne $q_i^{(k)}$ af $Q^{(k)}$ afviger fra $\varepsilon_i^{(k)} v_i$ til

$$O\left(\left(\max_{i=0,1,\ldots,m-1}\left|\frac{\lambda_{i+1}}{\lambda_i}\right|\right)^k\right)$$

når $k \to \infty$. Dette sker hvis egenværdierne opfylder (26.1) og alle delmatricer

$$\left((W^{(0)})^T\left[v_0 \mid v_1 \mid \ldots \mid v_{m-1}\right]\right)_{[:i,:i]}, \qquad \text{for } i = 1, \ldots, m$$

er invertibel. Vi har så lineært konvergens til egenvektorerne.

Lad os kikke på et eksempel. Betragt matricen

```
a = np.array([[1., 2., 1., 1., 0.],
              [2., 2., 1., 2., 1.],
              [1., 1., 3., 1., 0.],
              [1., 2., 1., 4., 1.],
              [0., 1., 0., 1., 5.]])
```

som er symmetrisk af størrelse 5×5:

```
print(np.all(a.T == a), a.shape)
```

```
True (5, 5)
```

Vi vil gerne finde de første 3 egenværdier.

Vi begynder med en tilfældig matrix med ortonormal søjler, fundet ved at beregne QR-dekomponering af en tilfældig (5×3)-matrix

```
rng = np.random.default_rng()
q, r = householder_qr(rng.random((a.shape[0], 3)))
w = q
print(w)
```

```
[[-0.25614492  0.22043255 -0.26763774]
 [-0.48074936 -0.30890802 -0.68157562]
 [-0.02108575  0.85658238 -0.35504922]
 [-0.6548477  -0.14231221  0.17298681]
 [-0.52344982  0.31937311  0.55483361]]
```

Så laver vi ortogonal iteration

```
for i in range(10):
    v = a @ w
    q, r = householder_qr(v)
    w = q

print(w)
```

```
[[-0.2961658  -0.2535628  -0.07545805]
 [-0.48144405 -0.16370616  0.04376184]
 [-0.31673777 -0.39346607 -0.7677259 ]
 [-0.60575686 -0.18667951  0.58727646]
 [-0.46177187  0.84808101 -0.24102754]]
```

Vi kan beregne de tilsvarende Rayleighkvotienter, som diagonal indgangerne i `q.T @ (a @ q)`

```
lambdaer = np.diag(q.T @ (a @ q))
print(lambdaer)
```

```
[7.36230784 4.58185123 2.29148033]
```

Vi kan sammenligne Aq_i med $\lambda_i q_i$

```
print((a @ q) - q * lambdaer)
```

```
[[-0.00108472  0.0006663   0.00452683]
 [-0.00070375 -0.00320472  0.00212766]
 [-0.00165906  0.01845632  0.01163135]
 [-0.00082251 -0.01774048  0.00668558]
 [ 0.00364639  0.00423835 -0.02178954]]
```

og ser at de stemmer overens indenfor to eller tre decimaler. Igen er konvergens ikke specielt hurtig.

26.3 Ækvivalens af metoderne

Vi påstår at QR-metoden ovenfor og ortogonal iteration med $m = n$ og $W^{(0)} = I_n$, producerer denne samme oplysning, nemlig en QR-faktorisering af A^k, når A er invertibel.

Da vi ved at ortogonal iteration konvergerer mod en diagonalisering af A følger det at det samme gælder for QR-metoden ovenfor.

For at se ækvivalensen, først lad os betragte QR-metoden. Her har vi $A^{(k-1)} = Q^{(k)}R^{(k)}$ og $A^{(k)} = R^{(k)}Q^{(k)}$. Disse to relationer kan bruges til at give

$$R^{(k)}Q^{(k)} = Q^{(k+1)}R^{(k+1)},$$

som viser hvordan vi kan bytte et RQ-produkt ud med et QR-produkt, ved at hæve indekserne. Vi har nu

$$A = Q^{(1)}R^{(1)}$$

og dermed

$$A^2 = Q^{(1)}R^{(1)}Q^{(1)}R^{(1)} = Q^{(1)}Q^{(2)}R^{(2)}R^{(1)}.$$

Dette kan bruges til at regne A^3, som

$$\begin{aligned} A^3 &= AA^2 = Q^{(1)}R^{(1)}Q^{(1)}Q^{(2)}R^{(2)}R^{(1)} \\ &= Q^{(1)}Q^{(2)}R^{(2)}Q^{(2)}R^{(2)}R^{(1)} = Q^{(1)}Q^{(2)}Q^{(3)}R^{(3)}R^{(2)}R^{(1)}. \end{aligned}$$

Generelt får vi på samme måde

$$A^k = Q^{(1)}Q^{(2)}\dots Q^{(k)}R^{(k)}\dots R^{(2)}R^{(1)}. \tag{26.2}$$

Dette er en QR-dekomponering $A = QR$ med

$$Q = Q^{(1)}Q^{(2)}\dots Q^{(k)} \quad \text{og} \quad R = R^{(k)}\dots R^{(2)}R^{(1)},$$

da produkter af ortogonale matricer er ortogonal, og produkter af øvre triangulære matricer er øvre triangulær.

For ortogonal iteration gælder generelt at

$$AW^{(k-1)} = W^{(k)}R^{(k)}.$$

I tilfældet hvor $m = n$ og $W^{(0)} = I_n$ har vi at hvert $W^{(k)}$ er en ortogonal matrix. Vi kan regne

$$A = AI_n = AW^{(0)} = W^{(1)}R^{(1)}$$

og

$$A^2 = AW^{(1)}R^{(1)} = W^{(2)}R^{(2)}R^{(1)}.$$

Tilsvarende har vi

$$A^3 = AW^{(2)}R^{(2)}R^{(1)} = W^{(3)}R^{(3)}R^{(2)}R^{(1)}$$

og generelt

$$A^k = W^{(k)}R^{(k)}\dots R^{(2)}R^{(1)}. \tag{26.3}$$

Dette er igen en QR-dekomponering af A^k, da $W^{(k)}$ er ortogonal og faktoren $R^{(k)}\dots R^{(2)}R^{(1)}$ er øvre triangulær. For A invertibel, er A^k også invertibel, så søjlerne af A^k er lineært uafhængig. Proposition 14.6 giver så at QR-dekomponeringen af A^k er entydigt, så (26.2) og (26.3) giver den samme QR-dekomponering af A^k, og vores påstand er vist.

Bemærk at vi har specielt at

$$W^{(k)} = Q = Q^{(1)}Q^{(2)}\dots Q^{(k)}$$

og at

$$R = R^{(k)}\dots R^{(2)}R^{(1)}.$$

Desuden har vi

$$A^{(k)} = Q^TAQ$$

da

$$\begin{aligned}
A^{(k)} &= R^{(k)}Q^{(k)} = (Q^{(k)})^TQ^{(k)}R^{(k)}Q^{(k)} \\
&= (Q^{(k)})^TR^{(k-1)}Q^{(k-1)}Q^{(k)} = (Q^{(k)})^T(Q^{(k-1)})^TQ^{(k-1)}R^{(k-1)}Q^{(k-1)}Q^{(k)} \\
&= \dots = (Q^{(k)})^T\dots(Q^{(2)})^T(Q^{(1)})^TQ^{(1)}R^{(1)}Q^{(1)}Q^{(2)}\dots Q^{(k)} \\
&= (Q^{(k)})^T\dots(Q^{(2)})^T(Q^{(1)})^TAQ^{(1)}Q^{(2)}\dots Q^{(k)} \\
&= Q^TAQ.
\end{aligned}$$

Det følger at for $A^{(k)}$ i QR-metoden har vi at

$$A \text{ symmetrisk} \implies A^{(k)} \text{ symmetrisk}$$

og at $A^{(k)}$ og A har de samme egenværdier.

Konvergens resultater for ortogonal iteration giver nu:

Proposition 26.1. *Hvis er $A \in \mathbb{R}^{n\times n}$ er symmetrisk med egenværdier der opfylder*

$$|\lambda_0| > |\lambda_1| > \cdots > |\lambda_{n-1}| > 0$$

og alle delmatricer $A_{[:i,:i]}$, $i = 1,\ldots,n$ er invertible, så konvergerer $A^{(k)}$ mod en diagonalmatrix D og Q mod en basis af egenvektorer med rate

$$O\left(\max_{j=0,\ldots,n-2}\left|\frac{\lambda_{j+1}}{\lambda_j}\right|\right).$$

Igen er dette kun lineær konvergens.

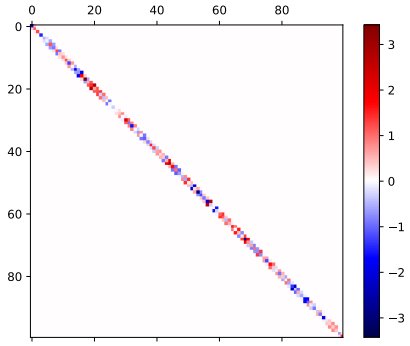
26.4 *QR*-dekomponering af tridiagonale matricer

Som noteret i kapitel 25, er den første trin i en typisk beregning af egenværdier for en symmetrisk matrix, at vi reducerer til tridiagonal form. Det viser sig at *QR*-dekomponering af tridiagonale matricer har pæne egenskaber, som er relevant når vi skal anvende *QR*-metoden på dem.

Lad os lave et eksperiment i python. Først definerer vi en funktion, som giver et tydeligt billede af en matrix; en heatmap med farveskala `seismic`, som er hvid i midten af skalaen.

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

def heat_map(a):
    fig, ax = plt.subplots()
    ax.set_aspect('equal')
    mx = a.max()
    mi = a.min()
    r = np.max([mx, -mi])
    if r < 20 * np.finfo(float).eps:
        r = 1
    im = ax.matshow(a, cmap='seismic', clim = (-r, r))
    fig.colorbar(im)
```

Lad os kikke på en tilfældig symmetrisk tridiagonal matrix `t`



Figur 26.1: En tridiagonal matrix.

```
n = 100
d = rng.standard_normal(n)
u = rng.standard_normal(n-1)
t = np.diag(d) + np.diag(u, 1) + np.diag(u, -1)

heat_map(t)
```

Billedet vises i figur 26.1. Vi kan nu beregne en QR-dekomponering af `t` og plotte faktorerne på samme måde

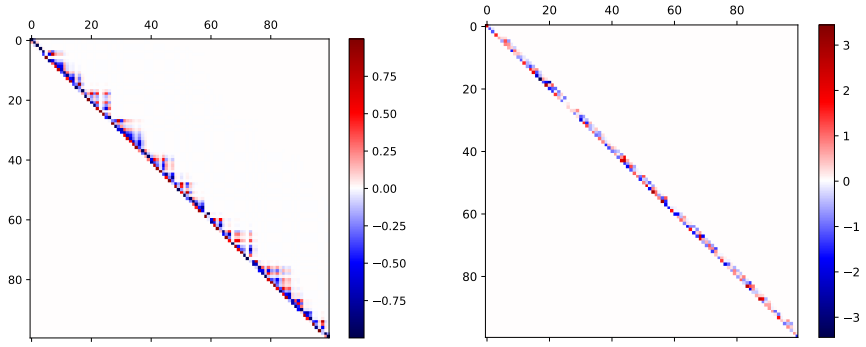
```
q, r = householder_qr(t)

heat_map(q)
```

```
heat_map(r)
```

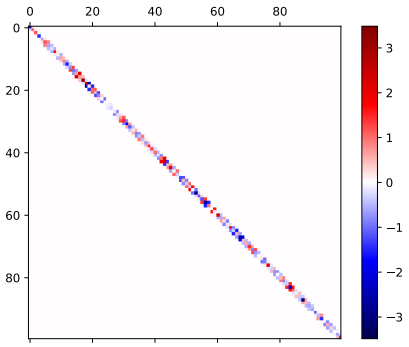
I figur 26.2 ser vi at begge faktorer har næsten en øvretriangulærform: R er naturligvis øvre triangulær, og Q har nultal under underdiagonalen. Vi kan bekræfte denne form for Q, ved

```
print(np.all(np.tril(q, -2) == 0))
```

```
True
```



Figur 26.2: Q og R faktorer for en tridiagonal matrix.



Figur 26.3: Resultat af et trin af QR-metoden på en symmetrisk tridiagonal matrix.

Dette betyder at når vi danne RQ, som trin i QR-metoden, får vi en matrix som igen har nultal i alle indgangene under underdiagonalen. Men for `t`

```
t1 = r @ q
heat_map(t1)
```

viser i figur 26.3 at RQ er faktisk symmetrisk og tridiagonal. Dette gælder da

$$RQ = Q^T QRQ = Q^T TQ$$

er symmetrisk.

26.5 Praktisk QR-metode

Konvergens QR-metoden kan forbedres med tre teknikker. Først reducerer vi A til triangulærform $T^{(0)}$. Derefter indføres nogle estimeringer $\mu^{(k)}$ for den mindste egenværdi af $T^{(k-1)}$, på en måde der ligner inverspotensmetoden.

Det sidste forbedring er at dele problemet i underproblemer, når indgang på underdiagonalen af $T^{(k)}$ bliver 0.

Samlet giver disse tiltag følgende algoritme

PRAKTISK QR-METODE($A \in \mathbb{R}^{n\times n}$ symmetrisk)
1 Beregn tridiagonalform $A = (Q^{(0)})^T T^{(0)} Q^{(0)}$
2 **for** $k \in \{1, 2, \ldots\}$:
3 Vælg $\mu^{(k)} \in \mathbb{R}$
4 $Q^{(k)} R^{(k)} = T^{(k-1)} - \mu^{(k)} I_n$
5 $T^{(k)} = R^{(k)} Q^{(k)} + \mu^{(k)} I_n$
6 Hvis der findes j med $(T^{(k)})_{j,j-1} = 0$
7 Sæt $T_0 = (T^{(k)})_{[:j,:j]}$ og $T_1 = (T^{(k)})_{[j:,j:]}$
8 Anvend QR-metoden på T_0 og på T_1

Kernen af algoritmen er QR-faktorisering af $T^{(k-1)} - \mu^{(k)} I_n$ til $Q^{(k)} R^{(k)}$. Matricen $T^{(k)} = R^{(k)} Q^{(k)} + \mu^{(k)} I_n$ har igen samme egenværdier som $T^{(k-1)}$ da

$$\begin{aligned} T^{(k)} &= R^{(k)} Q^{(k)} + \mu^{(k)} I_n = (Q^{(k)})^T Q^{(k)} R^{(k)} Q^{(k)} + \mu^{(k)} (Q^{(k)})^T Q^{(k)} \\ &= (Q^{(k)})^T \big(Q^{(k)} R^{(k)} + \mu^{(k)} I_n\big) Q^{(k)} = (Q^{(k)})^T T^{(k-1)} Q^{(k)}. \end{aligned}$$

I trin 3 er et simpelt valg $\mu^{(k)} = (T^{(k-1)})_{n-1,n-1}$, som er Rayleighs kvotient for den sidste søjle af Q.

En første python version af dette algoritme uden trin 6 er dermed

```
def semi_praktisk_qr_metode(a):
    n, _ = a.shape
    q, t = tridiagonal_qt(a)
    i_max = 20
    for i in range(i_max):
        mu_eye = t[-1,-1] * np.eye(n)
        q, r = householder_qr(t - mu_eye)
        t = r @ q + mu_eye
        if np.allclose(np.diag(t, -1), np.zeros(n-1),
                       atol = 10 * np.finfo(float).eps):
```

```
            break
    return t, i+1
```

Til orientering har vi valgt at denne funktion giver os den resultaterne tridiagonal matrix samt antallet af iterationer der er kørt. Vi stopper tidligt hvis vi har stort set fået en diagonalmatrix. Ellers bruger vi `i_max = 20` iterationer. Lad os prøve dette på en tilfældig matrix

```
n = 10
a = rng.normal(0.0, 5.0, (n, n))
a = (a + a.T) / 2

t, i = semi_praktisk_qr_metode(a)

print(np.diag(t))
print('iterationer:', i)
```

```
[ 15.71689333   9.22607015   9.83616821   4.40635964
 -16.07051729   0.94104428 -14.45726336  -0.9670911
 -10.62649682  -8.09774905]
iterationer: 20
```

Diagonalværdierne fra `t` kan sammenlignes med egenværdier beregnet af NumPys indbygget funktion

```
numpy_eig = np.linalg.eig(a)[0]
print(numpy_eig)
```

```
[ 15.71692285  10.16801159   8.894413   -16.57091967
 -14.47134622   4.40799549   1.44141913  -0.95483231
 -10.62649682  -8.09774905]
```

Nogle egenværdier er tæt på det rigtige, andre er langt fra.

```
print(np.sort(np.diag(t)) - np.sort(numpy_eig))
```

```
[ 5.00402374e-01  1.40828606e-02  3.55271368e-15
  0.00000000e+00 -1.22587995e-02 -5.00374843e-01
 -1.63584698e-03  3.31657153e-01 -3.31843382e-01
 -2.95161951e-05]
```

Egenværdierne, der beregnes bedst, er dem, der har relativt stort afstand fra nabo egenværdier. Ofte er dette den sidste, takket være $\mu^{(k)}$.

Valget $\mu^{(k)} = (T^{(k-1)})_{n-1,n-1}$ har dog problemer hvis $T^{(k-1)}$ har to sidste egenværdier med $|\lambda_{n-2}| = |\lambda_{n-1}|$. Så generelt foretrækkes det at bruge *Wilkinsons shift*, som er bestemt af den nederste (2×2)-blok af $T^{(k-1)}$ til at være

$$\mu^{(k)} = T^{(k-1)}_{[-1,-1]} - \frac{\varepsilon T^{(k-1)}_{[-1,-2]}}{|\delta| + \sqrt{\delta^2 + \left(T^{(k-1)}_{[-1,-2]}\right)^2}},$$

hvor

$$\delta = \frac{1}{2}\left(T^{(k-1)}_{[-2,-2]} - T^{(k-1)}_{[-1,-1]}\right) \quad \text{og} \quad \varepsilon = \begin{cases} 1 & \text{hvis } \delta \geqslant 0, \\ -1 & \text{ellers.} \end{cases}$$

Trin 6 tillader os at dele problemet i to på den følgende måde. Når der er et j så at indgangen $(T^{(k)})_{j,j-1} = 0$, har vi at

$$T^k = \begin{bmatrix} T_0 & 0 \\ 0 & T_1 \end{bmatrix},$$

hvor T_i er som givet i algoritmen, og vi kan regne videre med T_0 og T_1 hver for sig. I praksis er det nok at finde en indgang $(T^{(k)})_{j,j-1}$, som er tilstrækkelig tæt på 0.

Sættes begge disse betragtninger i spil, fås noget i retning af det følgende

```python
def praktisk_qr_metode(a):
    n, _ = a.shape
    if n == 1:
        t = a
        i = 0
        return t, i
    q, t = tridiagonal_qt(a)
    for i in range(20):
        delta = (t[-2, -2] - t[-1, -1]) / 2
        eps = 1 if delta >=0 else -1
```

```
        mu = t[-1,-1] - eps * (t[-1, -2]
             / (np.abs(delta)
                + np.sqrt(delta**2 + t[-1,-2]**2)))
        mu_eye = mu * np.eye(n)
        q, r = householder_qr(t - mu_eye)
        t = r @ q + mu_eye
        underdiag_abs = np.abs(np.diag(t, -1))
        zz = np.argwhere(underdiag_abs
                         < np.finfo(float).eps)
        if zz.shape[0] == underdiag_abs.shape[0]:
            break
        if zz.shape[0] != 0:
            zj = zz[0, 0] + 1
            t[:zj, :zj], j = praktisk_qr_metode(t[:zj, :zj])
            t[zj:, zj:], k = praktisk_qr_metode(t[zj:, zj:])
            i += j + k
            break
    return t, i
```

På vores eksempel giver dette

```
t, i = praktisk_qr_metode(a)

print(np.diag(t))
print('samlet iterationer:', i)

print('afvigelser')
print(np.sort(np.diag(t)) - np.sort(numpy_eig))
```

```
[ 15.71692285  10.16801159   8.894413   -16.57091967
 -14.47134622   4.40799549   1.44141913  -0.95483231
 -10.62649682  -8.09774905]
samlet iterationer: 24
afvigelser
[ 1.42108547e-14 -1.42108547e-14  0.00000000e+00
  0.00000000e+00 -4.32986980e-15 -3.77475828e-15
  1.06581410e-14  8.88178420e-15 -1.06581410e-14
  7.10542736e-15]
```

med alle afvigelser indenfor en rimelig faktor ganger machine epsilon.

Generelt kan det vises at den praktiske QR-metode har de følgende egenskaber:

- konvergens er tredje ordens,
- egenværdier beregnes indenfor $\|A\|_2 \epsilon_{\text{machine}} = \sigma_0 \epsilon_{\text{machine}}$,
- kan implementeres med $\sim 4n^3/3$ flops

Specielt er den præcist og hurtigt. Korrekt implementeret ligger det største arbejde i tridiagonalisering af $\mathtt{a}$, da QR-metoden på en tridiagonal matrix kan omskrives til $O(n^2)$.

Python indeks

Indeks