Numerisk Lineær Algebra F2021
Notesæt 17

Andrew Swann

6. april 2021

Sidst ændret: 6. april 2021.
Versionskode: cc54286.

Indhold

Figurer

Tabeller

17 Mindste kvadrater, konditionstal og Householder triangulering

Vi forsætter med at studere mindste kvadraters problemer og deres løsningsmetoder. Først vil vi have fokus på konditionstal forbundet til problemerne, som giver et indblik i hvor præcist løsninger kan forventes at være. Derefter vil vi se hvordan Householder matricer kan bruges til at beregne QR-dekomponeringer og løse mindste kvadraters problemer. Vi vil sammenligne de forskellige fremgangsmåder numerisk.

17.1 Konditionstal for mindste kvadraters problemer

Vi ønsker at studere mindste kvadraters løsninger $x \in \mathbb{R}^n$ til

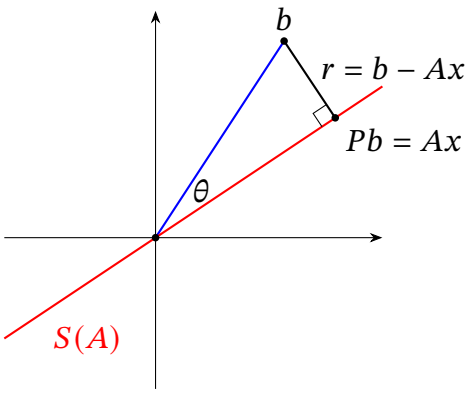
$$Ax = b$$

hvor $A \in \mathbb{R}^{m \times n}$, $m > n$, og $b \in \mathbb{R}^m$ er givet. Som vi har set, er dette ækvivalent med at løse

$$Ax = Pb \tag{17.1}$$

hvor P er projektion på søjlerummet $S(A)$.

Her er der flere delproblemer med hvert sit konditionstal. For at beskrive resultaterne er der nogle størrelser der skal indføres.



Figur 17.1: Vektorer og en vinkel for mindste kvadraters problemstilling.

I figur 17.1 har vi optegnet vektoren b og dens projektion Pb på søjlerummet $S(A)$. Restvektoren $r = b - Ax$ er ortogonal på $S(A)$. Punkterne b, Pb og origo danner en retvinklet trekant. Den spidse vinkel i origo betegnes θ. Vi har

$$\cos\theta = \frac{\|Pb\|_2}{\|b\|_2}.$$

Forbundet til A er dens konditionstal

$$\kappa(A) = \frac{\sigma_0}{\sigma_{k-1}}$$

hvor σ_i er singulærværdierne af A, og $k = n$, da $m > n$. For løsningen x til (17.1) er $\|Ax\|_2/\|x\|_2$ højst $\|A\|_2 = \max_{v\neq 0}\|Av\|_2/\|v\|_2 = \sigma_0$. Vi bruger

$$\eta = \|A\|_2 \frac{\|x\|_2}{\|Ax\|_2} = \sigma_0 \frac{\|x\|_2}{\|Ax\|_2},$$

som mål for hvor langt $\|Ax\|_2/\|x\|_2$ er fra at give den maksimale værdi $\|A\|_2$.

Lad os nu giver oplysning om konditionstal for det mindst kvadraters problem. Beviserne gennemgås ikke, men man kan finde nogle detaljer i Trefethen og Bau (1997) og referencerne derinde.

Problemstillingen afhænger af vektoren $b \in \mathbb{R}^m$ og af matricen $A \in \mathbb{R}^{m\times n}$. Hvis vi varierer b mens vi holder A fast, så er

(a) konditionstallet for beregning af Pb fra b

$$\kappa = \frac{1}{\cos\theta},$$

(b) konditionstallet for beregning af løsningen x til $Ax = Pb$, som funktion af b,

$$\kappa = \frac{\kappa(A)}{\eta \cos\theta}. \tag{17.2}$$

Disse er eksakte svar, og er ikke så besværlig at udlede. Derimod når man holder b fast og lad A varierer, er eksakte svar meget vanskelig at få fat i, og vi nøjes med at give øvre grænser:

(a) konditionstallet κ for beregning af Pb, som funktion af A, opfylder

$$\kappa \leqslant \frac{\kappa(A)}{\cos\theta};$$

(b) konditionstallet κ for beregning af løsningen x til $Ax = Pb$, som funktion af A, opfylder

$$\kappa \leqslant \kappa(A) + \frac{\kappa(A)^2 |\tan\theta|}{\eta}. \tag{17.3}$$

17.2 Et eksempel

Betragt det følgende eksempel hvor vi vil gerne beregne en tilnærmelse via polynomier til funktionen

$$f(t) = \frac{1}{c} e^{\sin(4t)}, \qquad \text{for } 0 \leqslant t \leqslant 1,$$

hvor c er en given konstant.

Vi arbejder med polynomier af grad 14. Alle funktioner erstattes af deres evaluering i 100 punkter t_i jævnt fordelt over intervallet $[0, 1]$. Vi får så at vi skal løse et mindste kvadraters problemstilling $Ax = b$, hvor A er en Vandermonde matrix med 15 søjler, og b er vektoren af værdierne $f(t_i)$, $i = 0, \ldots, 99$.

Jeg har beregnet løsningen algebraisk, og fundet værdien for konstanten c, der sikre at den korrekte mindste kvadraters løsning $x \in \mathbb{R}^{15}$ har $x_0 = 1.0$. Vi har så at $f(t)$ approksimeres af

$$p(t) = t^{14} + x_1 t^{13} + \cdots + x_{13} t + x_{14}.$$

I python kan vi opstille problemet med denne c-værdi på følgende vis

```
import numpy as np
```

```
m = 100
cols = 15

t = np.linspace(0, 1, m)

a = np.vander(t, cols)
c = 2006.787453104852
b = np.exp(np.sin(4 * t))[:, np.newaxis] / c
```

Vi vil løse problemet via forskellige metoder, og sammenligne resultaterne for x_0 med den korrekte værdi 1,0.

For at se hvor præcist en løsning vi kan forvente, beregner via konditionstal; specielt dem givet i (17.2) og (17.3). Først beregner via `kappa_a`, konditionstallet $\kappa(\mathtt{a})$ for `a`:

```
u, s, vt = np.linalg.svd(a, full_matrices=False)
kappa_a = s[0] / s[-1]
print(f'{kappa_a:e}')
```

```
2.271777e+10
```

Her har vi brugt `print(f'{var:e}')` til at skrive `var` i videnskabelig notation. Udtrykket `f'...'` angiver en streng med formatering. De kaldes »formatted string literals« eller »f-strings« i den officielle dokumentation for python. Elementer i strengen af formen `{var}` erstattes af værdien af `var`. Repræsentationen af værdien kan specificeres ved at skrive `{var:...}` hvor `...` giver formatet. Formen `{var:e}` giver `var` i videnskabelig notation.

Vi ser at konditionstallet $\kappa(\mathtt{a})$ er ret stort.

For at beregne $\cos\theta$ og η har vi brug for en løsning. Vi satser på at SVD-metoden giver et godt svar.

```
proj_b = u @ (u.T @ b)
cos_theta = np.linalg.norm(proj_b) / np.linalg.norm(b)
print(np.arccos(cos_theta) * 180 / np.pi)
```

```
0.0002146251778574735
```

```
x = vt.T @ (np.diag(1/s) @ (u.T @ b))
eta = s[0] * np.linalg.norm(x) / np.linalg.norm(proj_b)
print(f'{eta:e}')
```

```
2.103560e+05
```

Vi kan så samle disse værdier til at få vores oplysning om konditionstal

```
kond_x_b = kappa_a / (eta * cos_theta)
kond_x_a_højst = (kappa_a +
  (kappa_a**2 * np.sqrt(1-cos_theta**2) / (eta * cos_theta)))
print(f'kond_x_b = {kond_x_b:e}')
print(f'kond_x_a_højst = {kond_x_a_højst:e}')
```

```
kond_x_b = 1.079968e+05
kond_x_a_højst = 3.190818e+10
```

Vi ser at den sidste er størrelsesorden 10^{10}. Dette medvirker at vi kan højst forvente vores beregning af x_0 er korrekt inden for $x_0 \times 10^{10} \times \epsilon_{\text{machine}} \approx 10^{-6}$.

Vi har allerede en beregning fra SVD-metoden. Denne har x_0

```
print(x[0,0])
```

```
1.000000068599447
```

som er indenfor 10^{-6} af det korrekte svar 1,0.

```
korrekt = 1.0
print(x[0,0] - korrekt)
```

```
6.859944701176346e-08
```

Så dette er så godt et svar, som vi kan forvente at beregne.

Lad os nu afprøve beregning via den forbedrede Gram-Schmidt metode. Vi skal bestemmer en QR-dekomponering $A = QR$, og løse derefter ligningen $Rx = Q^T b$.

```
def forbedret_gram_schmidt(a):
    _, k = a.shape
    q = np.copy(a)
    r = np.zeros((k, k))
    for i in range(k):
        r[i, i] = np.linalg.norm(q[:, i])
        q[:, i] /= r[i, i]
        r[[i], i+1:] = q[:, [i]].T @ q[:, i+1:]
        q[:, i+1:] -= q[:, [i]] @ r[[i], i+1:]
    return q, r

q, r = forbedret_gram_schmidt(a)
x_qr_fgs = np.linalg.solve(r, q.T @ b)
print(x_qr_fgs[0,0])
```

```
1.0007157941525924
```

Dette rammer målet mindre godt end SVD-metoden. Vi har en fejl på omtrent 10^{-3}.

En tredje metode er at bruge normalligninger. Her løser vi ligningen $A^T Ax = A^T b$.

```
print((np.linalg.solve(a.T @ a, a.T @ b))[0,0])
```

```
-0.24673590859846803
```

Dette er en katastrofe! Der er ingen korrekte cifre og selve fortegnet i svaret er forkert.

Bemærk at konditionstallet for $A^T A$ er

```
_, sn, _ = np.linalg.svd(a.T @ a, full_matrices=False)
kond_at_a = sn[0] / sn[-1]
print(f'kond_at_a = {kond_at_a:e}')
```

```
kond_at_a = 7.507812e+17
```

som er størrelsesorden $1/\epsilon_{\text{machine}}$. Derfor kan man helle ikke forvente at løsningsmetoden via normalligninger har nogen som helst korrekte cifre.

17.3 Householder triangulering

Vi vil nu give en alternativ metode for at beregne QR-dekomponeringer. Dette er mere præcist end den forbedrede Gram-Schmidt process, og er faktisk også et element i computerberegning af SVD-dekomponeringer.

En fuld QR-dekomponeringer $A = QR \in \mathbb{R}^{n\times k}$ med $Q \in \mathbb{R}^{n\times n}$ ortogonal og $R \in \mathbb{R}^{n\times k}$ øvre triangulær er ækvivalent med

$$Q^T A = R = \begin{bmatrix} r_{00} & r_{01} & r_{02} & \dots \\ 0 & r_{11} & r_{12} & \dots \\ 0 & 0 & r_{22} & \dots \\ \vdots & \vdots & & \ddots \\ 0 & 0 & \dots & \dots \end{bmatrix}$$

da $Q^T = Q^{-1}$. Så for at bestemme en QR-dekomponering, er det not at finde en ortogonal matrix Q^T således at $Q^T A$ er øvre triangulær. Til dette formål kan vi bruge Householder transformationer.

Husk at en Householder matrix har formen

$$H = I_n - s v v^T$$

hvor $s = 2/\|v\|_2^2$. Proposition 9.14 fortæller os at givet vektoren $a_0 = (a_{00}, a_{10}, \dots, a_{n-1,0})$, findes der en Householder matrix således at $Ha_0 = (\pm\|a_0\|_2, 0, \dots, 0)$. Beregnes produkt HA, får vi så

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

Vi kan så få en øvre triangulær matricer ved at gentage ideen på følgende måde:

vi finder Householder matricer $H_0, H_1, \ldots, H_{k-1}$ således at

$$A \mapsto H_0 A = \begin{bmatrix} r_{00} & r_{01} & \cdots \\ 0 & * & \cdots \\ \vdots & \vdots & \\ 0 & * & \cdots \end{bmatrix} \mapsto H_1 H_0 A = \begin{bmatrix} r_{00} & r_{01} & r_{02} & \cdots \\ 0 & r_{11} & r_{12} & \cdots \\ 0 & 0 & * & \cdots \\ \vdots & \vdots & \vdots & \\ 0 & 0 & * & \cdots \end{bmatrix}$$

$$\mapsto H_2 H_1 H_0 A = \begin{bmatrix} r_{00} & r_{01} & r_{02} & r_{03} & \cdots \\ 0 & r_{11} & r_{12} & r_{13} & \cdots \\ 0 & 0 & r_{22} & r_{23} & \cdots \\ 0 & 0 & 0 & * & \cdots \\ \vdots & \vdots & \vdots & \vdots & \\ 0 & 0 & 0 & * & \cdots \end{bmatrix}$$

$$\cdots \mapsto H_{k-1} \cdots H_1 H_0 A = \begin{bmatrix} r_{00} & r_{01} & r_{02} & r_{03} & \cdots \\ 0 & r_{11} & r_{12} & r_{13} & \cdots \\ 0 & 0 & r_{22} & r_{23} & \cdots \\ 0 & 0 & 0 & r_{33} & \cdots \\ \vdots & \vdots & \vdots & & \ddots \\ 0 & 0 & 0 & \cdots & \cdots \end{bmatrix} = R$$

Vi sætter så

$$Q = (H_{k-1} \ldots H_1 H_0)^T = H_0 H_1 \ldots H_{k-1},$$

hvor den sidste lighed følger af $(AB)^T = B^T A^T$ og at $H_i^T = H_i$ for Householder matricer H_i.

Kombineret med opskrifterne fra proposition 9.14 kan vi nu bygge algoritmer for at implementere denne metode. Det må bemærkes at det er generelt unødvendigt at danne matricen Q; vi skal blot kende til de enkelte $H_i = I_n - s_i v_i v_i^T$, og disse er bestemt af vektoren v_i samt skalaren s_i.

Lad os først implementere metoden fra proposition 9.14. Vi beregner data for en Householder matrix, der sender en given vektor x til $\pm \|x\|_2 e_0$. Matematisk er metoden

HOUSE(x)
1 $u = x/\|x\|_2$
2 $\varepsilon = \begin{cases} -1 & \text{for } u_0 \geqslant 0 \\ +1 & \text{for } u_0 < 0 \end{cases}$
3 $s = 1 + |u_0|$
4 $v = (e_0 - \varepsilon u)/s$
5 **return** v og s

Dette kan oversættes til python, som

```
def house(x):
    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
```

Bemærk at $e_0 - \varepsilon u$ beregnes ved at finde $-\varepsilon u$ og så lægge 1 til den 0'te indgang. Tallet s kan godt beregnes fra v, men vi behøver ikke at gentage denne udregning senere, så vi vælger at returnere både vektoren v og skalaren s. Husk at vektoren v der konstrueres i proposition 9.14 har 0'te indgang 1.

Som nævnt ovenfor er det generelt ikke nødvendigt at beregne Q. I stedet for laver vi en funktion der samler den nødvendige data: matricen R, vektorerne v_i samt skalarene s_i. Matricen R er øvre triangulær, og vektoren v_i har formen

$$v_i = \begin{bmatrix} 0 \\ \vdots \\ 0 \\ 1 \\ (v_i)_1 \\ \vdots \\ (v_i)_{n-1-i} \end{bmatrix},$$

så v_i kan godt gemmes i samme matrix som R, under diagonalen:

$$\begin{bmatrix} r_{00} & r_{01} & r_{02} & \dots \\ (v_0)_1 & r_{11} & r_{12} & \dots \\ (v_0)_2 & (v_1)_1 & r_{22} & \dots \\ \vdots & \vdots & & \ddots \\ (v_0)_{n-1} & (v_1)_{n-2} & (v_2)_{n-3} & \end{bmatrix}. \tag{17.4}$$

HOUSEHOLDER QR DATA(A)
1 Beregn Householder v, s for
2 søjle 0 i A
3 søjle 0 i $(H_0A)_{[1:,1:]}$
4 søjle 0 i $(H_1H_0A)_{[2:,2:]}$
5 ...
6 Gem v data under diagonalen i $R = H_{k-1} \dots H_1H_0A$

Når vi implementerer dette i python må vi gerne huske at beregne $HA = (I_n - svv^T)A$, som $HA = A - sv(v^TA)$. Funktionen `householder_qr_data` gemmer dens output i en matrix vi kalder `data` $\in \mathbb{R}^{n\times k}$ af formen (17.4), og i en vektoren vi kalder `s` $\in \mathbb{R}^k$.

Matricen `data` overskrives med mellemregninger når vi bevæger os igennem algoritmen. Ved den `j`'te trin, er de første `j`-1 rækker af R på plads, og de første `j`-1 vektorer v_i og skalarer s_i er beregnede og gemt i deres pladser. Resten af matricen indeholder $(H_{j-1} \dots H_1H_0A)_{[j:,j:]}$. F.eks. for $j = 2$, har vi

$$\texttt{data} = \begin{bmatrix} r_{00} & r_{01} & r_{02} & \dots & r_{0,k-1} \\ (v_0)_1 & r_{11} & r_{12} & \dots & r_{1,k-1} \\ (v_0)_2 & (v_1)_1 & (H_1H_0A)_{22} & \dots & (H_1H_0A)_{2,k-1} \\ \vdots & \vdots & & \ddots & \vdots \\ (v_0)_{n-1} & (v_1)_{n-2} & (H_1H_0A)_{n-1,2} & \dots & (H_1H_0A)_{n-1,k-1} \end{bmatrix}.$$

I begyndelsen af algoritmen er `data` lige med `a` $= A$; vi tager en kopi af `a`, så den oprindelige matrix `a` bliver ikke ændret af funktionen.

```
def householder_qr_data(a):
    data = np.copy(a)
    _, k = a.shape
    s = np.empty(k)
```

```
    for j in range(k):
        v, s[j] = house(data[j:, [j]])
        data[j:, j:] -= (s[j] * v) @ (v.T @ data[j:, j:])
        data[j+1:, [j]] = v[1:]
    return data, s
```

Den største del af arbejdet i denne funktion foregår i trinnet

```
        data[j:, j:] -= s[j] * v @ (v.T @ data[j:, j:])
```

Vi har `data[j:, j:]` $\in \mathbb{R}^{n-j,k-j}$, og $v \in \mathbb{R}^{n-j}$, så dette trin koster

$$\begin{aligned}
&\text{matrixdifference} + \text{skalar-vektorprodukt} \\
&\quad + \text{ydre produkt} + \text{vektor-matrixprodukt} \\
&\quad = (n-j)(k-j) + (n-j) \\
&\qquad + (n-j)(k-j) + 2(n-j)(k-j) \\
&\quad = 4(n-j)(k-j) + (n-j) \text{ flops.}
\end{aligned}$$

Trinnet udføres for $j = 0, 1, \ldots, k-1$, så i alt har vi

$$\begin{aligned}
&\sum_{j=0}^{k-1} (4(n-j)(k-j) + n - j) \\
&\quad = \sum_{j=0}^{k-1} \left(4j^2 - (4n+4k+1)j + n(4k+1)\right) \\
&\quad = 4\frac{1}{6}(k-1)k(2k-1) - (4n+4k+1)\frac{1}{2}(k-1)k + n(4k+1)k \\
&\quad = 2nk^2 - \frac{2}{3}k^3 + 3nk - \frac{1}{2}k^2 + \frac{7}{6}k \\
&\quad \sim 2nk^2 - \frac{2}{3}k^3 \text{ flops}
\end{aligned}$$

Her havde vi brug for formlen

Lemma 17.1.

$$\sum_{j=0}^{k-1} j^2 = \frac{1}{6}(k-1)k(2k-1).$$

Bevis. Sæt $T_r(k) = \sum_{j=0}^{k-1} j^r$, hvor vi sætter $0^0 = 1$. Vi har $T_0(k) = k$, da det er kun en sum af 1 tal, og ved at $T_1(k) = \frac{1}{2}(k-1)k$, se (15.1).

Bemærk først at $T_r(k+1) = T_r(k) + k^r$, da der er kun en ekstra led. Nu har vi

$$\begin{aligned} T_3(k) + k^3 &= T_3(k+1) \\ &= \sum_{j=0}^{k-1} (j+1)^3 = \sum_{j=0}^{k-1} j^3 + 3j^2 + 3j + 1 \\ &= T_3(k) + 3T_2(k) + 3T_1(k) + T_0(k). \end{aligned}$$

Dette kan løses for $T_2(k)$, den ønskede sum, som

$$\begin{aligned} T_2(k) &= \frac{1}{3}(k^3 - 3T_1(k) - T_0(k)) \\ &= \frac{1}{3}\left(k^3 - \frac{3}{2}(k-1)k - k\right) = \frac{1}{6}k(2k^2 - 3(k-1) - 2) \\ &= \frac{1}{6}k(2(k-1)(k+1) - 3(k-1)) = \frac{1}{6}(k-1)k(2k-1), \end{aligned}$$

som påstået. □

Hvis man vil konstruere Q, er det mest effektiv at beregne det som

$$Q = H_0(H_1(\ldots(H_{k-1}((I_n)_{[:,:k]}))))$$

Grunden til dette ligger i at H_iB ændrer kun delmatricen $B_{[i:,i:]}$, så denne rækkefølge bruger ikke unødvendige flops. Den følgende funktion konstruerer Q og R fra output af `householder_qr_data`.

```
def householder_qr(a):
    data, s = householder_qr_data(a)
    n, k = a.shape
    r = np.triu(data[:k, :k])
    q = np.eye(n, k)
    for j in reversed(range(k)):
        x = data[j+1:, [j]]
        v = np.vstack([[1], x])
        q[j:, j:] -= (s[j] * v) @ (v.T @ q[j:, j:])
    return q, r
```

Lad os teste disse funktioner i et par eksempler. Først for en simpel matrix.

```
a_simpel = np.array([[ 1.0, 2.0],
                     [-1.0, 2.0],
                     [ 0.0, 1.0]])

q, r = householder_qr(a_simpel)
print('q = ', q)
print()
print('r = ', r)
```

```
q =  [[-0.70710678 -0.66666667]
 [ 0.70710678 -0.66666667]
 [ 0.         -0.33333333]]

r =  [[-1.41421356  0.        ]
 [ 0.         -3.        ]]
```

Vi tjekker at `q` er ortogonal og at `q @ r` er tæt på `a_simpel`:

```
print(q.T @ q)
print(a_simpel - q @ r)
```

```
[[1. 0.]
 [0. 1.]]
[[ 2.22044605e-16 -4.44089210e-16]
 [-2.22044605e-16 -4.44089210e-16]
 [ 0.00000000e+00  0.00000000e+00]]
```

Nu lad os kikke på vores eksempel fra afsnit 15.1.

```
s = 1e-8
a_s = np.array([[1.0, 1.0, 1.0],
                [  s, 0.0, 0.0],
                [0.0,   s, 0.0],
                [0.0, 0.0,   s]])

q, r = householder_qr(a_s)
```

```
print('q =', q)
print()
print('r =', r)
```

```
q = [[-1.00000000e+00  7.07106781e-09  4.08248290e-09]
 [-1.00000000e-08 -7.07106781e-01 -4.08248290e-01]
 [ 0.00000000e+00  7.07106781e-01 -4.08248290e-01]
 [ 0.00000000e+00  0.00000000e+00  8.16496581e-01]]

r = [[-1.00000000e+00 -1.00000000e+00 -1.00000000e+00]
 [ 0.00000000e+00  1.41421356e-08  7.07106781e-09]
 [ 0.00000000e+00  0.00000000e+00  1.22474487e-08]]
```

Grammatricen er nu

```
print(q.T @ q)
```

```
[[1. 0. 0.]
 [0. 1. 0.]
 [0. 0. 1.]]
```

som er en stor forbedring over beregningerne med forbedret Gram-Schmidt, hvor indre produkterne $\langle$`v1`, `v2`$\rangle$ havde størrelsesorden 10^{-16}, og $\langle$`v0`, `v1`$\rangle$ og $\langle$`v0`, `v2`$\rangle$ var størrelsesorden 10^{-8}. Vi har også at `q @ r` er pænt tæt på `a_s`:

```
print(a_s - q @ r)
```

```
[[ 0.00000000e+00  0.00000000e+00  0.00000000e+00]
 [ 0.00000000e+00 -1.65436123e-24 -1.65436123e-24]
 [ 0.00000000e+00  1.65436123e-24  8.27180613e-25]
 [ 0.00000000e+00  0.00000000e+00 -1.65436123e-24]]
```

Til sammenligning kan vi bruge den indbyggede QR-dekomponering i NumPy fra funktionen `np.linalg.qr`

```
q, r = np.linalg.qr(a_s)
print(q.T @ q)
print(a_s - q @ r)
```

```
[[1.00000000e+00 0.00000000e+00 0.00000000e+00]
 [0.00000000e+00 1.00000000e+00 5.55111512e-17]
 [0.00000000e+00 5.55111512e-17 1.00000000e+00]]
[[ 0.00000000e+00  0.00000000e+00  0.00000000e+00]
 [ 0.00000000e+00  3.30872245e-24  4.13590306e-24]
 [ 0.00000000e+00 -1.65436123e-24 -1.65436123e-24]
 [ 0.00000000e+00  0.00000000e+00  0.00000000e+00]]
```

Vi ser at vores Householder implementering er lige så god, som den indbyggede.

17.4 Householder metoden for mindste kvadraters problemer

Det overstående kan godt bruges til at løse mindste kvadraters problemer, men det er bedst at bruge `householder_qr_data` i stedet for at beregne Q. Vi skal løse $Rx = Q^T b$, men

$$Q^T b = H_{k-1}(\dots(H_1(H_0 b)))_{[:k]}.$$

Dette fører til følgende python kode

```
def householder_lsq(a, b):
    data, s = householder_qr_data(a)
    _, k = a.shape
    r = np.triu(data[:k, :k])
    c = np.copy(b)
    for j in range(k):
        x = data[j+1:, [j]]
        v = np.vstack([[1], x])
        c[j:] -= (s[j] * np.vdot(v, c[j:])) * v
    return np.linalg.solve(r, c[:k])
```

Lad os afprøve denne metode på vores eksempel fra afsnit 17.2.

```
print(householder_lsq(a, b)[0,0])
```

```
0.9999999602219982
```

metode	opstilling	løsning x fra	$\sim$ flops
QR via Gram-Schmidt	$A = QR$	$Rx = Q^T b$	$2mn^2$
QR via Householder	$A = QR$	$Rx = Q^T b$	$2mn^2 - \frac{2}{3}n^3$
reduceret SVD	$A = U\Sigma V^T$	$x = V(\Sigma^{-1}(U^T b))$	$2mn^2 + 11n^3$
normalligning	$A^T Ax = A^T b$	$(A^T A)x = (A^T b)$	$mn^2 + \frac{1}{3}n^3$

Tabel 17.1: Forskellige løsningsmetoder for mindste kvadraters problemer $Ax = Pb$, $A \in \mathbb{R}^{m \times n}$, $m > n$.

Dette ligger inden for 10^{-6} af det korrekte svar 1,0

```
print(householder_lsq(a, b)[0,0] - korrekt)
```

```
-3.9778001781343164e-08
```

så er lige så godt, som SVD-resultatet. Vi ser at det er ikke selve QR-metoden der er problematisk, men derimod hvordan man udregner data for denne dekomponering. Householder metoden er meget præcis og bruges, som en del af standardmetoder for at udregne SVD-dekomponeringer. Derfor er det til at foretrække at løsning af mindste kvadraters problemer foregår via Householder metoden, som kræver færre flops.

Tabel 17.1 giver en opsummering af de forskellige metoder vi har. Bortset fra SVD-metoden, forudsætter de andre metoder i tabel 17.1 at søjlerne af A er lineært uafhængig.

Bemærk at det er ofte bedst at løse en reduceret lineær ligningssystem, end at beregne en invers matrix. F.eks. for QR-metoderne, er det bedre at bruge back substitution til at løse $Rx = Q^T b$ end at finde x som $x = R^{-1}(Q^T b)$. Det har vi gjort ovenfor ved at bruge `np.linalg.solve` i stedet for `np.linalg.inv`. Det ikke så svært at implementere back substitution selv, men vi overlader det, som en opgave for læseren.

Bibliografi

Trefethen, L. N. og D. Bau III (1997). *Numerical linear algebra*. Society for Industrial og Applied Mathematics (SIAM), Philadelphia, PA. ISBN: 0-89871-361-7. DOI: 10.1137/1.9780898719574.

Python indeks

Indeks