Numerisk Lineær Algebra F2021
Notesæt 28

Andrew Swann

18. maj 2021

Sidst ændret: 18. maj 2021.
Versionskode: d9cd21f.

Indhold

28 *LU*-dekomponering

Her genbesøger vi lineære ligningssystemer, med en metode der er baseret på systematisk brug af elementære rækkeoperationer. Metoden er relativ hurtigt, så er ofte valgt af nogen i anvendelse, men det er desværre ret svært at holde styr på hvor præcist resultatet bliver.

28.1 LU-dekomponering og Gausseliminering

Lad os betragte det følgende lineære ligningssystem

$$\begin{aligned} 2x + 3y - 4z &= 7, \\ 3x - 4y + z &= -2, \\ x + y + 2z &= 3, \end{aligned} \tag{28.1}$$

som i afsnit 6.1. Vi har tidligere løst systemet ved hjælp af elementære rækkeoperationer. Husk at disse er givet ved

(I)	$R_i \leftrightarrow R_j$	`a[ [i,j], :] = a[ [j,i], :]`
(II)	$R_i \to sR_i$ $(s \neq 0)$	`a[i, :] *= s`
(III)	$R_i \to R_i + tR_j$ $(j \neq i)$	`a[i, :] += t * a[j, :]`

og at de kan realiseres via matrixprodukter. F.eks. for $n = 2$ kan operationerne $R_0 \leftrightarrow R_1$, $R_1 \to sR_1$ og $R_1 \to R_1 + tR_0$ er realiseret via venstre multiplikation ved

$$\text{(I)} \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix} \quad \text{(II)} \begin{bmatrix} 1 & 0 \\ 0 & s \end{bmatrix} \quad \text{(III)} \begin{bmatrix} 1 & 0 \\ t & 1 \end{bmatrix}.$$

For $n = 3$, bemærk at kombinationen (a) $R_1 \to R_1 + tR_0$ efterfulgt af (b) $R_2 \to R_2 + rR_0$ kan realiseres, som $A \mapsto L_b L_a A = LA$, hvor

$$L = L_b L_a = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ r & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 \\ t & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 \\ t & 1 & 0 \\ r & 0 & 1 \end{bmatrix}.$$

Vi kan reducere systemet (28.1) til øvretriangulærform udelukkende ved brug af rækkeoperation (III) $R_i \to R_i + tR_j$ med $j < i$. Arbejdes der systematisk ved at indføre nultal under diagonalen i hver søjle får vi

$$A = \begin{bmatrix} 2 & 3 & -4 \\ 3 & -4 & 1 \\ 1 & 1 & 2 \end{bmatrix} \xrightarrow[R_2 \to R_2 - \frac{1}{2}R_0]{R_1 \to R_1 - \frac{3}{2}R_0} L_1 A = \begin{bmatrix} 2 & 3 & -4 \\ 0 & -17/2 & 7 \\ 0 & -1/2 & 4 \end{bmatrix}$$

$$\xrightarrow{R_2 \to R_2 - \frac{1}{17}R_1} L_2 L_1 A = \begin{bmatrix} 2 & 3 & -4 \\ 0 & -17/2 & 7 \\ 0 & 0 & 61/7 \end{bmatrix} = U$$

med slutmatricen øvre triangulær. Dette giver os en faktorisering $A = LU$, hvor

$$L = L_1^{-1} L_2^{-1} = \begin{bmatrix} 1 & 0 & 0 \\ 3/2 & 1 & 0 \\ 1/2 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 1/17 & 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 \\ 3/2 & 1 & 0 \\ 1/2 & 1/17 & 1 \end{bmatrix}$$

er nedre triangulær og U er øvre triangulær. Faktoriseringen $A = LU$ kaldes en *LU-dekomponering* af A.

Denne fremgangsmåde virker generelt. Lad os arbejde med kvadratiske $(n \times n)$-matricer. Vi kan eliminere alle indgange under et diagonal element i en matrix X

$$X = \begin{bmatrix} x_{00} & \dots & x_{0,i-1} & x_{0i} & x_{0,i+1} & \dots & x_{0,n-1} \\ & \ddots & \vdots & \vdots & \vdots & & \vdots \\ 0 & & x_{i-1,i-1} & x_{i-1,i} & x_{i-1,i+1} & \dots & x_{i-1,n-1} \\ 0 & \dots & 0 & x_{ii} & x_{i,i+1} & \dots & x_{i,n-1} \\ 0 & \dots & 0 & x_{i+1,i} & x_{i+1,i+1} & \dots & x_{i+1,n-1} \\ \vdots & & \vdots & \vdots & \vdots & & \vdots \\ 0 & \dots & 0 & x_{n-1,i} & x_{n-1,i+1} & \dots & x_{n-1,n-1} \end{bmatrix}$$

ved flere rækkeoperationer af type (III) ved at danne produktet $L_i X$, hvor

$$L_i = \begin{bmatrix} 1 & & 0 & 0 & 0 & \dots & 0 \\ & \ddots & & \vdots & \vdots & & \vdots \\ 0 & & 1 & 0 & 0 & \dots & 0 \\ 0 & \dots & 0 & 1 & 0 & \dots & 0 \\ 0 & \dots & 0 & -x_{i+1,i}/x_{ii} & 1 & & 0 \\ \vdots & & \vdots & \vdots & & \ddots & \\ 0 & \dots & 0 & -x_{n-1,i}/x_{ii} & 0 & & 1 \end{bmatrix}.$$

Bemærk at matricen L_i kan skrives ved hjælp af et ydre produkt, som

$$L_i = I_n - w_i e_i^T, \quad \text{hvor } w_i = \begin{bmatrix} 0 \\ \vdots \\ 0 \\ x_{i+1,i}/x_{ii} \\ \vdots \\ x_{n-1,i}/x_{ii} \end{bmatrix}.$$

Denne matrix har invers

$$L_i^{-1} = I_n + w_i e_i^T, \tag{28.2}$$

da $e_i^T w_i = 0$ giver $(I_n - w_i e_i^T)(I_n + w_i e_i^T) = I_n - w_i e_i^T + w_i e_i^T - w_i e_i^T w_i e_i^T = I_n$.

Desuden er produktet i den ene rækkefølge af to af disse inverse nem at beregne,

$$L_i^{-1}L_{i+1}^{-1} = I_n + w_i e_i^T + w_{i+1}e_{i+1}^T \tag{28.3}$$

(produktet $L_{i+1}^{-1}L_i^{-1}$ har ikke så simpel en form).

Ligninger (28.2) og (28.3) betyder at vi kan nemt implementere denne metode. Vi får

GAUSSELIMINERING UDEN OMBYTNING($A \in \mathbb{R}^{n\times n}$)
1 $U = A, L = I_n$
2 **for** $i \in \{0, 1, \ldots, n-2\}$:
3 $\quad L_{[i+1:,[i]]} = U_{[i+1:,[i]]}/u_{ii}$
4 $\quad U_{[i+1:,i:]} = U_{[i+1:,i:]} - L_{[i+1:,[i]]}U_{[[i],i:]}$
5 **return** L, U

som bruger $\sim 2n^3/3$ flops.

Via en LU-dekomponering af A, kan vi løse et lineært ligningssystem $Ax = b$ på følgende måde

1 Beregn $A = LU$
2 Løs $Ly = b$ for y via forward substitution
3 Løs $Ux = y$ for x via backward substitution

Her kan backsubstitution realiseres via

BACKSUBSTITUTION($U \in \mathbb{R}^{n\times n}, b \in \mathbb{R}^{n\times 1}$)
1 **for** $i \in \{n-1, \ldots, 1, 0\}$:
2 $\quad x_i = (b_i - U_{[[i],i+1:]}x_{[i+1:]})/u_{ii}$
3 **return** x

og en tilsvarende algoritme giver forward substitution. Backsubstition bruger $\sim n^2$ flops, så løsning af $Ax = b$ via LU-dekomponering domineres af selve beregning af L og U, og i alt bruge Gausseliminering uden ombytning $\sim 2n^3/3$ flops.

Dette er dobbelt så hurtigt, som løsning via QR-faktorisering: At løse $Ax = b$, er et særtilfælde af en mindste kvadraters problem. Tabel 17.1 med $m = n$ fortæller at QR-faktorisering via Householder metoden bruger $\sim 2n^3 - (2n^3/3) = 4n^3/3$ flops.

Så LU-metoden er hurtigere, men desværre er den *ikke* stabil. Den kan hellere ikke anvendes hvis vi får $u_{ii} = 0$ i et trin undervejs. F.eks. for

$$A = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix}$$

fejler LU-dekomponering i det første trin da $a_{00} = 0$, selvom A er invertibel.

Det sidste problem kan undgås hvis vi også tillader rækkeombytning, så rækkeoperationer af type (I), udover dem af type (III). En implementering af dette er

GAUSSELIMINERING MED OMBYTNING($A \in \mathbb{R}^{n \times n}$)
1 $U = A, L = I_n, P = I_n$
2 **for** $i \in \{0, 1, \ldots, n-2\}$:
3 Vælg $j \geqslant i$ så at $|u_{ji}|$ er størst
4 Byt række i i $U_{[:,i:]}$ med række j
5 Byt række i i $L_{[:,:i]}$ med række j
6 $L_{[i+1:,[i]]} = U_{[i+1:,[i]]}/u_{ii}$
7 $U_{[i+1:,i:]} = U_{[i+1:,i:]} - L_{[i+1:,[i]]}U_{[[i],i:]}$
8 **return** L, U, P

som giver matricer L, U og P således at $PA = LU$. Matricen P har netop én ettal i hver række og i hver søjle, og indeholder oplysning om hvordan man bytter rækkerne i A.

Det kan vises at Gausseliminering med ombytning beregner løsninger indenfor en fejl der er

$$O(\max |u_{ij}| \epsilon_{\text{machine}} / \max |a_{ij}|).$$

Problemet er at denne fejl kan være ret stor.

Betragt for eksempel den følgende matrix

```python
import numpy as np

def eksempel_mat(n):
    a = np.eye(n, dtype=float)
    a -= np.tri(n, k=-1)
    a[:, -1] = 1
    return a

a = eksempel_mat(10)
print(a)
```

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

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

Her er indgangerne på diagonalen størst i deres søjle, så der sker ikke noget ombytning i Gausselimineringen.

```python
def gauss_uden_ombytning(a):
    n, _ = a.shape
    u = np.copy(a)
    l = np.eye(n)
    for i in range(n-1):
        l[i+1:, [i]] = u[i+1:, [i]] / u[i,i]
        u[i+1:, i:] -= l[i+1:, [i]] @ u[[i], i:]
    return l, u

l, u = gauss_uden_ombytning(a)

print('L =')
print(l)
print('U =')
print(u)
```

```
L =
[[ 1.  0.  0.  0.  0.  0.  0.  0.  0.  0.]
 [-1.  1.  0.  0.  0.  0.  0.  0.  0.  0.]
 [-1. -1.  1.  0.  0.  0.  0.  0.  0.  0.]
 [-1. -1. -1.  1.  0.  0.  0.  0.  0.  0.]
 [-1. -1. -1. -1.  1.  0.  0.  0.  0.  0.]
 [-1. -1. -1. -1. -1.  1.  0.  0.  0.  0.]
 [-1. -1. -1. -1. -1. -1.  1.  0.  0.  0.]
 [-1. -1. -1. -1. -1. -1. -1.  1.  0.  0.]
 [-1. -1. -1. -1. -1. -1. -1. -1.  1.  0.]
```

```
 [-1. -1. -1. -1. -1. -1. -1. -1. -1.  1.]]
U =
[[  1.   0.   0.   0.   0.   0.   0.   0.   0.   1.]
 [  0.   1.   0.   0.   0.   0.   0.   0.   0.   2.]
 [  0.   0.   1.   0.   0.   0.   0.   0.   0.   4.]
 [  0.   0.   0.   1.   0.   0.   0.   0.   0.   8.]
 [  0.   0.   0.   0.   1.   0.   0.   0.   0.  16.]
 [  0.   0.   0.   0.   0.   1.   0.   0.   0.  32.]
 [  0.   0.   0.   0.   0.   0.   1.   0.   0.  64.]
 [  0.   0.   0.   0.   0.   0.   0.   1.   0. 128.]
 [  0.   0.   0.   0.   0.   0.   0.   0.   1. 256.]
 [  0.   0.   0.   0.   0.   0.   0.   0.   0. 512.]]
```

Bemærk at u har en indgang der er `2.**(10 - 1) = 512.`, så fejlen har også denne størrelsesorden

```
print(np.max(np.abs(u)) / np.max(np.abs(a)))
```

```
512.0
```

ganget med machine epsilon

```
meps = np.finfo(float).eps
print(np.max(np.abs(u)) * meps / np.max(np.abs(a)))
```

```
1.1368683772161603e-13
```

For generel n har den tilsvarende matrix a fejlfaktoren kontrolleret af 2^{n-1}. Så for stort n vil det være et overvældende problem.

Det kan vises at 2^{n-1} er den størst mulige fejlfaktor for Gausseliminering på en $(n \times n)$-matrix.

Lad og kikke på løsning til $Ax = b$ for denne matrix a og et tilfældigt valgt b.

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

b = rng.normal(0.0, 5.0, (a.shape[0], 1))
print(b)
```

```
[[-3.33507522]
 [ 6.64707323]
 [12.27108391]
 [ 0.54061189]
 [-2.53559401]
 [ 3.09731124]
 [-4.24772021]
 [ 6.15079156]
 [ 9.86630353]
 [-1.01063967]]
```

Først bruger vi LU-metoden

```
def back_subs(u, b):
    n, _ = u.shape
    x = np.empty((n, 1))
    for i in reversed(range(n)):
        x[i] = (b[i] - u[[i], i+1:] @ x[i+1:]) / u[i, i]
    return x

def forward_subs(l, b):
    n, _ = l.shape
    y = np.empty((n, 1))
    for j in range(n):
        y[j] = (b[j] - l[[j], :j] @ y[:j] ) / l[j, j]
    return y

x_lu = back_subs(u, forward_subs(l, b))
print(x_lu)
```

```
[[-4.87427526]
 [ 0.23359793]
 [ 6.09120655]
 [ 0.45194107]
 [-2.17232376]
 [ 1.28825774]
 [-4.76851598]
 [ 0.86147981]
```

```
 [ 5.4384716 ]
 [ 1.53920004]]
```

Vi ser hvor tæt den er på at være en løsning ved at kikke på forskellen med Ax og b:

```
print(a @ x_lu - b)
```

```
[[ 0.00000000e+00]
 [ 0.00000000e+00]
 [ 1.77635684e-15]
 [-1.55431223e-15]
 [-1.77635684e-15]
 [ 5.32907052e-15]
 [-1.06581410e-14]
 [ 2.66453526e-15]
 [-2.30926389e-14]
 [-5.04041253e-14]]
```

Dette er ikke så dårligt, og er lidt bedre end, men i tråd med, vores forventning ovenfor, som ville give en absolut fejl på

```
print(np.max(np.abs(b))
      * np.max(np.abs(u)) * meps / np.max(np.abs(a)))
```

```
1.3950607251055856e-12
```

Hvis vi bruger QR-metoden, har vi ikke det samme problem med matricer med store indgange

```
q, r = householder_qr(a)
print(np.max(np.abs(r)) / np.max(np.abs(a)))
```

```
3.162277660168379
```

som giver en relativfejl på

```
print(np.max(np.abs(r)) * meps / np.max(np.abs(a)))
```

```
7.021666937153401e-16
```

Det bekræftes af at løsningen

```
x_qr = back_subs(r, q.T @ b)
```

er lidt mere præcist.

```
print(a @ x_qr - b)
```

```
[[ 1.77635684e-15]
 [ 8.88178420e-16]
 [-1.77635684e-15]
 [-1.11022302e-15]
 [ 8.88178420e-16]
 [ 8.88178420e-16]
 [ 2.66453526e-15]
 [ 8.88178420e-16]
 [ 1.77635684e-15]
 [ 8.88178420e-16]]
```

For større n er der dog en meget tydelig forskel

```
a = eksempel_mat(200)

b = rng.normal(0.0, 5.0, (a.shape[0], 1))
print('Største indgang i b:', np.max(np.abs(b)))

l, u = gauss_uden_ombytning(a)
x_lu = back_subs(u, forward_subs(l, b))
print('Afvigelse Ax fra b' )
print('LU metoden:', np.max(np.abs(a @ x_lu - b)))

q, r = householder_qr(a)
x_qr = back_subs(r, q.T @ b)
print('QR metoden:', np.max(np.abs(a @ x_qr - b)))
```

```
Største indgang i b: 15.673819799307369
Afvigelse Ax fra b
LU metoden: 16.07084503905439
QR metoden: 1.3322676295501878e-13
```

Her ser vi at fejlen i LU-metoden er af samme størrelsesorden, som selve indgangerne i b, hvor QR-metoden har kun en fejl på størrelse 10^{-13}.

Så meget fristende at afskrive LU-metoden. Men det mystiske er at disse stor fejl viser sig at være atypisk. Lad os først implementere Gauss eliminering med ombytning, og bekræfter at det virker i et tilfældig eksempel.

```
def gauss(a):
    n, _ = a.shape
    u = np.copy(a)
    l = np.eye(n)
    p = np.eye(n)
    for i in range(n-1):
        j = np.argmax(np.abs(u[i:, i])) + i
        p[[i,j], :] = p[[j,i], :]
        u[[i,j], i:] = u[[j,i], i:]
        l[[i,j], :i] = l[[j,i], :i]
        l[i+1:, [i]] = u[i+1:, [i]] / u[i,i]
        u[i+1:, i:] -= l[i+1:, [i]] @ u[[i], i:]
    return l, u, p

a = rng.normal(0.0, 5.0, (10, 10))
l, u, p = gauss(a)
print(np.max(np.abs(p @ a - l @ u)))
```

```
2.6645352591003757e-15
```

Vi definerer tilvækstraten for en matrix $A = LU$ til at være faktoren

$$\max |u_{ij}| / \max |a_{ij}|,$$

som styre den relative fejl.

```
def tilvækst(a):
    l, u, p = gauss(a)
    return np.max(np.abs(u)) / np.max(np.abs(a))
```

Nu kører vi et eksperiment hvor vi tager tilfældige matricer og beregner denne tilvækstrate. Vi samler på den største tilvækstrate, som opstår undervejs.

```
n = 200
max = 0
for i in range(1000):
    t = tilvækst(rng.normal(0.0, 5.0, (n, n)))
    if t > max:
        max = t

print(t)
```

```
6.275912018212276
```

Vi ser at dette er væsentlig mindre end raten `2^199` for det værste tilfælde, nemlig matricen `eksempel_mat(200)`.

Desværre findes der ikke en god forklaring for dette, endnu, men det medfører at i praksis kan LU-dekomponering med ombytning alligevel være brugbar.

Python indeks

Indeks