Numerisk Lineær Algebra F2021
Notesæt 15

Andrew Swann

6. april 2021

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

Indhold

Figurer

15 Forbedring af Gram-Schmidt

15.1 Numeriske problemer for klassisk Gram-Schmidt

Lad os betragte det følgende eksempel i python. Vi begynder med følgende vektorer i $\mathbb{R}^4$:

$$u_0 = \begin{bmatrix} 1 \\ s \\ 0 \\ 0 \end{bmatrix}, \quad u_1 = \begin{bmatrix} 1 \\ 0 \\ s \\ 0 \end{bmatrix}, \quad u_2 = \begin{bmatrix} 1 \\ 0 \\ 0 \\ s \end{bmatrix}$$

og vælger s til at være tæt på 0.

```
>>> import numpy as np
>>> s = 1e-8
>>> u0 = np.array([1.0,   s, 0.0, 0.0])[:, np.newaxis]
>>> u1 = np.array([1.0, 0.0,   s, 0.0])[:, np.newaxis]
>>> u2 = np.array([1.0, 0.0, 0.0,   s])[:, np.newaxis]
>>> a = np.hstack([u0, u1, u2])
>>> a
array([[1.e+00, 1.e+00, 1.e+00],
       [1.e-08, 0.e+00, 0.e+00],
       [0.e+00, 1.e-08, 0.e+00],
       [0.e+00, 0.e+00, 1.e-08]])
```

Vi udfører den klassiske Gram-Schmidt proces, med forventningen at det giver en ortonormal samling `v0`, `v1`, `v2`.

```
>>> def proj_på(v, u):
...     return np.vdot(v,u) / np.vdot(v,v) * v
...
>>> v0 = u0 / np.linalg.norm(u0)
>>> w1 = u1 - proj_på(v0, u1)
>>> v1 = w1 / np.linalg.norm(w1)
>>> w2 = u2 - proj_på(v0, u2) - proj_på(v1, u2)
>>> v2 = w2 / np.linalg.norm(w2)
>>> q = np.hstack([v0, v1, v2])
```

```
>>> q
array([[ 1.00000000e+00,  0.00000000e+00,  0.00000000e+00],
       [ 1.00000000e-08, -7.07106781e-01, -7.07106781e-01],
       [ 0.00000000e+00,  7.07106781e-01,  0.00000000e+00],
       [ 0.00000000e+00,  0.00000000e+00,  7.07106781e-01]])
```

Så har vi dannet en matrix `q` med søjler `v0`, `v1`, `v2`. Vi kan se om `v0`, `v1`, `v2` er faktisk ortonormal eller ej, ved at beregne Grammatricen for `q`:

```
>>> gram = q.T @ q
>>> gram
array([[ 1.00000000e+00, -7.07106781e-09, -7.07106781e-09],
       [-7.07106781e-09,  1.00000000e+00,  5.00000000e-01],
       [-7.07106781e-09,  5.00000000e-01,  1.00000000e+00]])
```

På diagonalen får vi pæne tal tæt på 1, så `v0`, `v1`, `v2` er enhedsvektorer. Desuden er de andre indre produkter med `v0` af den samme størrelsesorden som s, men de er skuffende langt væk fra machine epsilon. Det værste problem er dog $\langle$`v1`, `v2`$\rangle$, som burde være 0, men er tæt på $1/2$. Det vil sige vinklen mellem `v1` og `v2` er

```
>>> np.arccos(np.vdot(v1,v2) \
...           / (np.linalg.norm(v1)*np.linalg.norm(v2))) \
...   * 360/(2*np.pi)
59.99999999999999
```

omtrent 60°, så de er meget langt fra at være vinkelret på hinanden. Dette er udtryk for at den klassiske Gram-Schmidt proces er numerisk ustabil.

Lad os bytte rundt på vores operationer ovenfor:

```
>>> v0 = u0 / np.linalg.norm(u0)
>>> w1 = u1 - proj_på(v0, u1)
>>> x2 = u2 - proj_på(v0, u2)
>>> v1 = w1 / np.linalg.norm(w1)
>>> w2 = x2 - proj_på(v1, x2)
>>> v2 = w2 / np.linalg.norm(w2)
```

Her har vi delt dannelsen af w_2 op i to trin

$$\begin{aligned} w_2 &= u_2 - \mathrm{pr}_{v_0}(u_2) - \mathrm{pr}_{v_1}(u_2) \\ &= (u_2 - \mathrm{pr}_{v_0}(u_2)) - \mathrm{pr}_{v_1}(u_2 - \mathrm{pr}_{v_0}(u_2)) \\ &= x_2 - \mathrm{pr}_{v_1}(x_2), \qquad\qquad \text{for } x_2 = u_2 - \mathrm{pr}_{v_0}(u_2). \end{aligned}$$

Denne omskrivning er gyldigt, da $\mathrm{pr}_{v_0}(u_2)$ er parallelt med v_0 og $\mathrm{pr}_{v_1}(v_0) = 0$, så $\mathrm{pr}_{v_1}(\mathrm{pr}_{v_0}(u_2)) = 0$. Vi får nu

```
>>> q = np.hstack([v0, v1, v2])
>>> q
array([[ 1.00000000e+00,  0.00000000e+00,  0.00000000e+00],
       [ 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]])
```

og Grammatricen bliver

```
>>> gram = q.T @ q
>>> gram
array([[ 1.00000000e+00, -7.07106781e-09, -4.08248290e-09],
       [-7.07106781e-09,  1.00000000e+00, -1.11022302e-16],
       [-4.08248290e-09, -1.11022302e-16,  1.00000000e+00]])
```

Dette er en væsentlig forbedring, nu er alle indgange væk fra diagonalen af størrelsesorden højst 10^{-8}, og faktisk er `v2` og `v3` vinkelret på hinanden indenfor machine epsilon.

Hvorfor har denne omskrivning givet et bedre resultat? I beregningen via den klassiske Gram-Schmidt proces er `v0` lige med `u0`, da

```
>>> np.linalg.norm(u0)
1.0
```

Tallet `s` er netop valgt så at `s**2` er mindre en $\epsilon_{\text{machine}}$.

Beregning af `proj_på(v0, u1)` giver så `v0`, og derefter beregnes `w1` til at være $(0, -s, s, 0)$, som er vinkelret på `u2`. Dette medvirker til at `v1` bidrager ikke til udregningen af `w2`, og så ortogonalitet mellem `v1` og `v2` bliver ikke sikret.

Til gengæld for den anden version af udregning bliver `w1` $= (0, -\mathtt{s}, \mathtt{s}, 0)$ og `x2` $= (0, -\mathtt{s}, 0, \mathtt{s})$, som kun indeholde led af størrelsesorden `s`. Beregning af `w2` sikre nu en vektor (næsten) vinkelret på `w1` (og `v1`).

15.2 Den forbedrede Gram-Schmidt proces

For at implementere den overstående ide, lad os starte med $u_0, u_1, \ldots, u_{k-1}$ lineært uafhængige. Så går vi i gang med at konstruere ortonormale $v_0, v_1, \ldots, v_{k-1}$. I trinnet hvor vi danne v_r vil vi sørge for at vi kun arbejde med vektorer, som er vinkelret på $v_0, \ldots, v_{r-2}$. Dette udmøntes i den følgende.

FORBEDRET GRAM-SCHMIDT PROCES - SKITSE$(u_0, u_1, \ldots, u_{k-1})$

1 Sæt $w_i^{(0)} = u_i$, $i = 0, \ldots, k-1$, $\quad v_0 = w_0^{(0)} / \|w_0^{(0)}\|$.

2 Sæt $w_i^{(1)} = w_i^{(0)} - \mathrm{pr}_{v_0}(w_i^{(0)})$, $i = 1, \ldots, k-1$, $\quad v_1 = w_1^{(1)} / \|w_1^{(1)}\|$.

3 Sæt $w_i^{(2)} = w_i^{(1)} - \mathrm{pr}_{v_1}(w_i^{(1)})$, $i = 2, \ldots, k-1$, $\quad v_2 = w_2^{(2)} / \|w_2^{(2)}\|$.

$\ldots$

4 Sæt $w_i^{(r)} = w_i^{(r-1)} - \mathrm{pr}_{v_{r-1}}(w_i^{(r-1)})$, $i = r, \ldots, k-1$, $\quad v_r = w_r^{(r)} / \|w_r^{(r)}\|$.

$\ldots$

5 **return** $v_0, v_1, \ldots, v_{k-1}$, en ortogonal samling,
der udspænder det samme rum som $u_0, u_1, \ldots, u_{k-1}$.

Matematisk er dette ækvivalent med den klassiske Gram-Schmidt proces.

Eksempel 15.1. Et eksempel, som billedliggør forskellen på klassisk og forbedret Gram-Schmidt er følgende. Betragt $V = \mathbb{R}^5$ og vektorer $u_0, u_1, \ldots, u_4$, som er søjlerne i matricen

$$A = \left[u_0 \mid u_1 \mid u_2 \mid u_3 \mid u_4\right] = \begin{bmatrix} 1 & * & * & * & * \\ 0 & 1 & * & * & * \\ 0 & 0 & 1 & * & * \\ 0 & 0 & 0 & 1 & * \\ 0 & 0 & 0 & 0 & 1 \end{bmatrix},$$

hvor $*$ repræsenterer vilkårlige tal. Da $u_0 = e_0$ er enhedsvektor, ændres denne vektor ikke af det første trin i den klassiske Gram-Schmidt process. Anden trin derimod sørger for at søjle 1 bliver til e_1, dvs. at indgangen a_{01} sættes til 0.

Næste trin giver e_2 i søjle 2, så a_{02} og a_{12} nulstilles, osv.

$$\left[\begin{array}{c|c|c|c|c} v_0 & v_1 & v_2 & v_3 & v_4 \end{array}\right] = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 \end{bmatrix}$$

Dvs. den klassiske Gram-Schmidt process fremstiller vektorerne $v_0, v_1, \ldots, v_4$ søjlevis, én efter én.

For den forbedrede Gram-Schmidt process, begynder vi på sammen måde, $v_0 = u_0$. Men beregnes $w_i^{(1)}$, og dette sørger for at der kommer 0-tal i resten af den første række, dvs. a_{01}, a_{02}, a_{03}, a_{04} nulstilles. Derefter opereres på $A_{[1:,1:]}$, og $*$-indgangerne nulstilles rækkevis

$$\left[\begin{array}{c|c|c|c|c} v_0 & v_1 & v_2 & v_3 & v_4 \end{array}\right] = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 \end{bmatrix}.$$

△

I $\mathbb{R}^n$ fører dette til den følgende algoritme:

FORBEDRET GRAM-SCHMIDT$(u_0, u_1, \ldots, u_{k-1})$
1 **for** $i \in \{0, 1, \ldots, k-1\}$**:**
2 $\quad w_i = u_i$
3 **for** $i \in \{0, 1, \ldots, k-1\}$**:**
4 $\quad r_{ii} = \|w_i\|_2$
5 $\quad v_i = w_i / r_{ii}$
6 $\quad$ **for** $j \in \{i+1, \ldots, k-1\}$**:**
7 $\quad\quad r_{ij} = v_i^T w_j$
8 $\quad\quad w_j = w_j - r_{ij} v_i$

Hvis det er nødvendigt kan hukommelsesplads spares ved at gemme v_i i w_i.

15.3 Klassisk kontra forbedret Gram-Schmidt

Lad os kikke på implementering af disse algoritme i python, og giver et eksempel, som viser hvor stærkt den forbedrede Gram-Schmidt process er i forhold til den klassiske.

Vi begynder ved at importere vores standardpakker

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

Den klassiske Gram-Schmidt implementeres på følgende vis

```
def klassisk_gram_schmidt(a):
    n, k = a.shape
    q = np.empty((n,k))
    r = np.zeros((k,k))
    for j in range(k):
        r[:j, [j]] = q[:, :j].T @ a[:, [j]]
        w = a[:, [j]] - q[:, :j] @ r[:j, [j]]
        r[j, j] = np.linalg.norm(w)
        q[:, [j]] = w / r[j, j]
    return q, r
```

Algoritmen returnerer matricerne `q` og `r` for en tynd QR-dekomponering af `a`. Vores tidligere fremstilling, side 8, af den klassiske Gram-Schmidt process indebar to **for**-løkker. Ovenfor har vi omskrevet den indre **for**-løkke via matrixprodukter. Vores input vektorer er søjlerne af `a`; den `j`'te søjle er `a[:, [j]]`. De ortonormale vektorer der produceres af processen bliver til søjlerne i `q`. Ved det `j`'te trin har vi dannet ortonormale søjler `0` til `j-1` af `q`. De nye indgang i `r` gives via indre produkter med `a[:, [j]]`, som kan samles i produktet `q[:, :j] @ a[:, [j]]`. Projektionen af `a[:, [j]]` langs de første `j` søjler af `q` er så `q[:, :j] @ r[:j, [j]]`. Vi overskrive `w` hver gang med den nye w_j, dens længde gemmes i `r[j, j]` og den tilsvarende enhedsvektor bliver til `q[:, [j]]`.

For den forbedrede Gram-Schmidt laver vi tilsvarende omskrivninger til matrixprodukter.

```
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
```

Initialiseringstrinnet er blot at kopiere `a` til `q`; denne kopiering er vigtig, da indgangerne i `q` overskrives i løbet af processen. Ved det `i`'te trin i processen inderholder søjler `0` til `i-1` af `q` ortonormale vektorer, som en del af slut resultatet, resten af `q` indeholder vektorerne $w_j^{(i)}$, som bliver overskrevet gentagende gange. Matricen `r` dannes rækkevis.

Lad os først bekræfter at disse implementering giver de samme resultater, som vi fik for eksemplet i begyndelsen af kapitlet.

```
s = 1e-8
a = 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 = klassisk_gram_schmidt(a)
print(a - q @ r)
print()
print(q.T @ q)
```

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

[[ 1.00000000e+00 -7.07106781e-09 -7.07106781e-09]
 [-7.07106781e-09  1.00000000e+00  5.00000000e-01]
 [-7.07106781e-09  5.00000000e-01  1.00000000e+00]]
```

Bemærk at selvom `q` er langt fra at have ortonormale søjler, vi har trods alt at `q @ r` er meget tæt på `a`.

```
q, r = forbedret_gram_schmidt(a)
print(a - q @ r)
```

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

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

[[ 1.00000000e+00 -7.07106781e-09 -4.08248290e-09]
 [-7.07106781e-09  1.00000000e+00  1.11022302e-16]
 [-4.08248290e-09  1.11022302e-16  1.00000000e+00]]
```

Igen `q @ r` ligger tæt på `a`, men denne gang er `q` væsentligt tættere på at have ortonormale søjler.

Nu udfører vi et eksperiment. Vi vil danne en tilfældig (100×100)-matrix med bestemte singulærværdier, nemlig

$$(\sigma_0, \ldots, \sigma_{99}) = (1, 1/2, 1/4, 1/8, \ldots, 1/2^{99}).$$

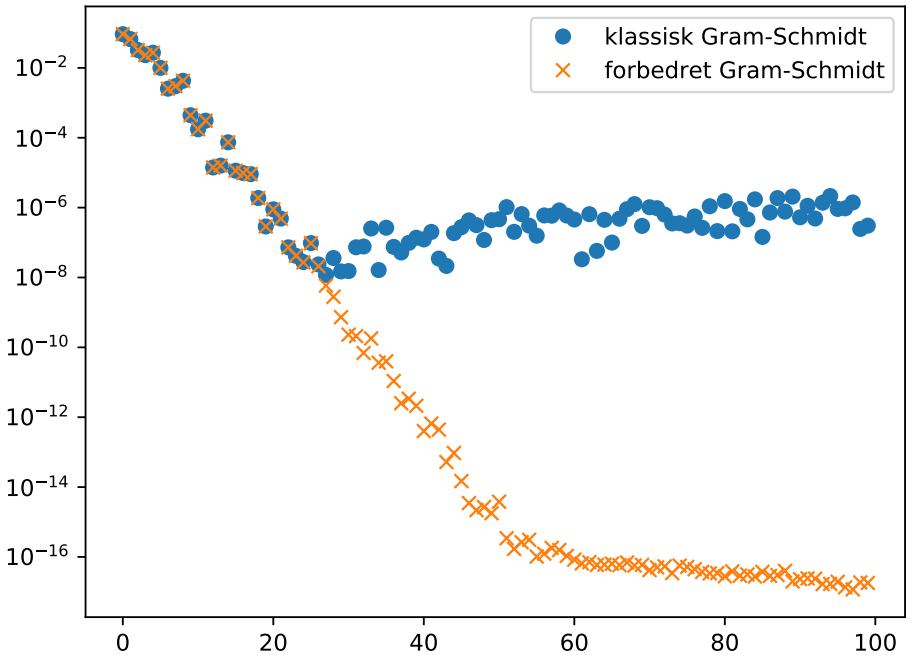
Vi gør dette ved beregne singulærværdidekomponeringen af en tilfældig matrix og bruger dens `u` og `vt` til at danne en ny matrix `a`:

```
rng = np.random.default_rng()
n = 100
u, _, vt = np.linalg.svd(rng.random((n, n)))
i = np.arange(n)
s = np.array(2.0 ** (-i))
a = u @ np.diag(s) @ vt
print(s[:5])
```

```
[1.     0.5    0.25   0.125  0.0625]
```

Da singulærværdierne aftager hurtigt, burde beregning af QR-dekomponering af `a` giver diagonalindgange i `r`, som er tæt på singulærværdierne for `a`. Vi kan beregne `r` via de to algoritmer og plotte disse diagonalværdier.

```
qk, rk = klassisk_gram_schmidt(a)
qf, rf = forbedret_gram_schmidt(a)
```



Figur 15.1: Klassisk kontra forbedret Gram-Schmidt.

```
fig, ax = plt.subplots()
ax.set_yscale('log')
ax.plot(i, rk[i,i], 'o', label='klassisk Gram-Schmidt')
ax.plot(i, rf[i,i], 'x', label='forbedret Gram-Schmidt')
ax.legend()
```

Resultatet visers i figur 15.1. Her ses tydeligt at den forbedrede Gram-Schmidt beregne mange flere indgange korrekt end den klassiske. Den forbedrede giver korrekte værdier til omkring indeks 55, men

```
print(2.**-55)
```

```
2.7755575615628914e-17
```

er tæt på machine epsilon. Til gengæld er den klassiske Gram-Schmidt kun korrekt for de første cirka 25 indgange, og værdier efterfølgende ligger om 10^{-8}, som er tæt på kvadratroden af machine epsilon

```
print(np.sqrt(np.finfo(float).eps))
```

```
1.4901161193847656e-08
```

Dette er ret typisk for opførelsen af de to algoritmer.

15.4 Flops

Lad os tæller antallet af `float` operationer forbundet med den forbedrede Gram-Schmidt process. Optællingen for den klassiske er nogenlunde den samme. Vi kikker på algoritmen side 6. Løkken i linjerne 1 og 2 tæller ingen flops. Kikker vi på den anden **for**-løkke, som starter på linje 3, har vi først

$$\begin{aligned} r_{ii} = \|w_i\|_2\colon &\quad 2n+1 \text{ flops} \\ v_i = w_i / r_{ii}\colon &\quad n \text{ flops} \end{aligned}$$

så efter k omganger bidrager dette med $k(3n+1)$ flops. I den indre **for**-løkke, som starter på linje 6 har vi

$$\begin{aligned} r_{ij} = v_i^T w_j\colon &\quad 2n \text{ flops} \\ w_j = w_j - r_{ij} v_i\colon &\quad 2n \text{ flops} \end{aligned}$$

Så denne indre løkke bidrager med $4n(k-i-1)$ flops for hvert $i \in \{0, 1, \ldots, k-1\}$, dvs.

$$4n \sum_{i=0}^{k-1} (k-i-1) = 4n \sum_{j=0}^{k-1} j = 2n(k-1)k \text{ flops.}$$

Det sidste bruger formlen

$$\sum_{j=0}^{k-1} j = \frac{1}{2}(k-1)k, \tag{15.1}$$

som følger af

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

Alt i alt har vi

$$2n(k-1)k + k(3n+1) = 2nk^2 + kn + k \sim 2nk^2 \text{ flops}$$

for den forbedrede Gram-Schmidt process.

Python indeks

Indeks