

Numerisk Lineær Algebra

Andrew Swann

Institut for Matematik
Aarhus Universitet

Forår 2021

Indhold

Indhold	iii
Figurer	viii
Tabeller	xi
1 Indledning og første betragtninger	1
1.1 Tal og fejl	2
1.2 Flydende-komma repræsentation	5
1.3 Fejl i flydende-komma tal	9
2 Plangeometri: vektorer og matricer	13
2.1 Vektorer, norm og vinkelbestemmelse	13
2.2 Indre produkt og matrixtransformationer	15
2.3 Første tegninger	21
3 Matricer og vektorer	25
3.1 Matricer	25
3.2 Matricer i python	26
3.3 Heatmap plot	28
3.4 Multiplikation med en skalar og matrixsum	31
3.5 Transponering	33
3.6 Vektorer	34
4 Matrix multiplikation	37
4.1 Række-søjleprodukt	37
4.2 Matrixprodukt	39
4.3 Regneregler for matrixprodukt	45
4.4 Identitetsmatricer	47
	iii

INDHOLD

4.5	Ydre produkt	48
5	Numerisk håndtering af matricer	51
5.1	Omkostninger ved matrixberegning	51
5.2	Pythons for-løkke	53
6	Lineære ligningssystemer	57
6.1	Elementære rækkeoperationer	57
6.2	Rækkeoperationer i python	59
6.3	Echelonform	62
6.4	Løsning via rækkeoperationer	63
7	Matrixinvers	69
7.1	Egenskaber af den inverse matrix	69
7.2	Eksistens af den inverse	73
7.3	Beregning af den inverse	75
8	Ortogonalitet og projektioner	77
8.1	Standard indre produkt	77
8.2	Vinkel mellem vektorer	80
8.3	Projektion på en linje	84
8.4	Ortogonalitet	86
9	Ortogonale matricer	95
9.1	Definition og første eksempler	95
9.2	Egenskaber	96
9.3	Ortonormale vektorer og ortogonale matricer	98
9.4	Householdermatrix	100
9.5	Udvidelse til ortonormal basis	105
9.6	Plan geometri: trigonometriske identiteter	106
10	Singulærværdidekomponering	111
10.1	Singulærværdier i dimension 2	112
10.2	SVD generelt	115
10.3	Eksempler på SVD	116
10.3.1	Punkter i planen	116
10.3.2	Billede med støj	118
10.3.3	Komprimering af et billede	120
10.4	Eksistens af SVD	124

11	Konditionstal	127
11.1	Konditionstal for differentiable funktioner	127
11.2	Matrixnorm	130
11.3	Konditionstal og lineære ligningssystemer	133
12	Generelle vektorrum	137
12.1	Vektorrum	137
12.2	Andre skalarer	139
12.3	Underrum	143
13	Indre produkter	147
13.1	Definitioner og eksempler	147
13.2	Ortogonale samlinger og tilnærmelse	152
13.3	Numerisk integration	155
14	Ortogonale samlinger: klassisk Gram-Schmidt	161
14.1	Den klassiske Gram-Schmidt proces	161
14.2	Klassisk Gram-Schmidt i \mathbb{R}^n	167
15	Forbedring af Gram-Schmidt	171
15.1	Numeriske problemer for klassisk Gram-Schmidt	171
15.2	Den forbedrede Gram-Schmidt proces	174
15.3	Klassisk kontra forbedret Gram-Schmidt	176
15.4	Flops	180
16	Mindste kvadraters metode	183
16.1	Problemformulering	183
16.2	Løsning via QR-dekomponering	184
16.3	Løsning via singularværdidekomponering	185
16.4	Polynomier	186
16.5	Løsning via normalligninger	193
17	Mindste kvadrater: konditionstal og Householder	195
17.1	Konditionstal for mindste kvadraters problemer	195
17.2	Et eksempel	197
17.3	Householder triangulering	201
17.4	Householder metoden	209
18	Lineære afbildninger og matricer	211

INDHOLD

18.1	Lineære afbildninger	211
18.2	Matrix af en lineær transformation	214
18.3	Kombinationer af lineære afbildninger	215
18.4	Kerne og billedmængde	216
19	Koordinater	219
19.1	Basis	221
19.2	Koordinater	223
19.3	Koordinatskift	225
19.4	Baser for nul- og søjlerum af en matrix	226
19.5	Dimension	228
20	Lineære transformationer og koordinatskift	229
20.1	Generelle matrixrepræsentationer	229
20.2	Basisskift	232
20.3	Afbildning på ét vektorrum	233
21	Egenverdier og egenvektorer	239
21.1	Et første eksempel	239
21.2	Definitioner og første regnemetoder	240
21.3	Andre eksempler	244
21.4	Determinanter	247
21.5	Et andet eksempel	249
22	Differentialligninger	253
22.1	Førsteordenssystemer	253
22.2	Komplekse egenverdier	258
22.3	Højere ordens lineære differentialligninger	264
23	Matrixfaktorisering fra egenverdier	267
23.1	Eksistens af egenverdier	267
23.2	Determinant og egenverdier	268
23.3	Unitære matricer og Schurdekomponeringen	269
23.4	Spektralsætningen	272
23.5	Egenskaber af determinanter	277
24	Potensmetoden og inverspotensmetoden for egenverdier	283
24.1	Potensmetoden	283
24.2	Page rank	288

24.3 Rayleighs kvotient	292
24.4 Inverspotensmetoden	293
25 Hessenberg- og tridiagonalform	299
25.1 Reduktion til Hessenbergform	302
25.2 Reduktion til tridiagonalform	305
26 Egendekomponering via QR-metoden	311
26.1 QR-metoden: første version	311
26.2 Ortogonal iteration	314
26.3 Ækvivalens af metoderne	317
26.4 QR-dekomponering af tridiagonale matricer	319
26.5 Praktisk QR-metode	322
27 Singulærværdier og principale komponenter	327
27.1 Singulærværdidekomponering og symmetriske matricer	327
27.2 Principalkomponent analyse	332
28 LU-dekomponering	339
28.1 LU-dekomponering og Gausseliminerings	339
A Det græske alfabet	351
B Python	353
Bibliografi	355
Python indeks	357
Indeks	361

Figurer

2.1	Afstand og vinkel i planen	13
2.2	Vinklen mellem to vektorer	16
2.3	Skalering	18
2.4	Rotation	18
2.5	Spejling	20
2.6	Projektion	20
2.7	Skævvridning	21
3.1	En første heatmap plot	29
3.2	Heatmap med colorbar	30
3.3	Heatmap med justeret farveskala	30
3.4	Vektorsum af $u = (2, 1)$ og $v = (1, 3)$	35
4.1	Linjen $x + 3y = 2$	39
4.2	Elektrisk kredsløb	44
4.3	Eksempler på ydre produkter	50
5.1	Oprindelig figur og dens rotationer	56
6.1	Rundkørsel	64
6.2	To linjer i planen, som skærer	66
6.3	Tre linjer i planen, som har intet fællespunkt	67
6.4	To parallelle linjer i planen har intet fællespunkt	67
8.1	Projektion på en linje	84
8.2	Ortogonal samling af polynomier	93
8.3	Numerisk approksimation af eksponentialfunktionen	93
8.4	Eksponentialfunktionen og dens Taylor udvikling	94
9.1	Effekten af en Householdermatrix	101

9.2	Spejling til εe_0	102
9.3	Cosinus og sinus for en enhedsvektor i planen	107
9.4	Enhedsvektoren u drejet igennem vinkel φ	107
9.5	Spejling i x -aksen	108
10.1	Cirkel efter matrixmultiplikation	112
10.2	SVD af punktsamling	118
10.3	Cifret 0, plus støj	120
10.4	SVD af billede med støj	121
10.5	Uppsala domkirke	122
10.6	Singulærværdier for billedet af domkirken	123
10.7	SVD komprimering	124
11.1	Trekantsuligheden	130
13.1	Fourier cosinus tilnærmelse for $1 - x$	154
13.2	Integral	156
13.3	Rektanglet fra den venstre funktionsværdi	156
13.4	Venstre- og højre-tilnærmelse til integral	157
13.5	Trapez-tilnærmelse til integral	158
14.1	Legendre og Chebychev polynomier	167
15.1	Klassisk kontra forbedret Gram-Schmidt	180
16.1	Andengradspolynomium igennem 3 datapunkter	188
16.2	Lineær tilnærmelse af data	190
16.3	Højere grad	191
17.1	Mindste kvadraters problemstilling	196
19.1	Standardkoordinater	219
19.2	Drejet ortogonal koordinatsystem	220
19.3	Generel lineært koordinatsystem	220
21.1	Tredjegradspolynomium	250
22.1	Beholdere forbudne med rør	256
22.2	Saltmængden i beholder A mod beholder B	259
22.3	Elektrisk kredsløb	261
22.4	Løsningen til kredsløb	263

FIGURER

22.5 System med klods og fjedre	265
24.1 En samling websider med links	289
26.1 En tridiagonal matrix	320
26.2 Q og R faktorer for en tridiagonal matrix	321
26.3 QR -trin på en tridiagonal matrix	321
27.1 Plot af data og den centrede version	335
27.2 Principale komponenter, uden normalisering	336
27.3 Principale komponenter, normaliserede	337

Tabeller

1.1	Python regneoperationer på tal	3
1.2	Fejl i beregninger til 3 decimaler	4
5.1	Omkostninger ved vektor- og matrixberegninger	53
6.1	Rækkeoperationer i python	60
17.1	Løsningsmetoder for mindste kvadraters problemer	210
27.1	Vægt og højde for nogle personer	332

Forord

Disse noter er skrevet for kurset Numerisk Lineær Algebra ved Aarhus Universitet, som kørte første gang i foråret af 2020. Denne version er redigeret ved kurset i 2021.

Lineær algebra et smukt matematisk emne og et stærkt værktøj, der anvendes i mange fag i forbindelse med matematiske modeller, og som er grundlæggende for mange teorier i naturvidenskabelige fag. Et område, hvor der er voksende fokus, er datavidenskab og analyse af store datamængde. I disse sammenhæng og ingeniørfag er det nødvendigt at behandle data på en computer og udfører operationer via et programmeringssprog.

Kurset har til formål både at indføre de studerende i sådanne beregningsmetode og til at forklare og begrunde den relevante teori. Programmeringssproget python anvendes igennem kurset, som eksempel sprog. Tidligere programmerings erfaring er ikke påkrævet, og den matematiske baggrund er begrænset til viden opnået på gymnasiet og i matematikkurser der ligger den første semester af universitetsuddannelser.

Der er mange bøger, der giver en god indledning til lineær algebra og dens anvendelser. En, der kan anbefales i forbindelse med dette kursus, er Leon (2015); andre bøger med lignende titler er lige så relevant. Men det er en vigtig pointe at de matematiske metoder der præsenteres i standard kurser og tekster om lineær algebra er ikke vedegnede til numerisk behandling af tal på en computer. Derfor lægges der vægt på også at beskrive metode, der har gode egenskaber i forhold til styring af fejl i beregninger. Der er gode kilder for sådanne materiale, jeg vil især nævne bogen Trefethen og Bau (1997), men disse tekster forventer at læseren har allerede gennemført et matematisk kursus i lineær algebra. Sådan et krav stilles ikke til læseren af disse noter. Vi indfører teorien samtidig med de varianter der er mest relevant i en numerisk kontekst, og lægger vægt på delene af teorien, som er bedst egnede til disse metoder og deres anvendelse.

FORORD

I løse form følger disse noter en kombination af de to ovennævnte kilder. Efter gennemførsel af kurset, kan bogen af Trefethen og Bau (1997) godt anbefales til viderelæsning. For dem der vil komme endnu dybere i faget er den store og uomgæelig værk Golub og Van Loan (2013), som har også været en god inspirationskilde for mig.

Dette notesæt er skrevet i \LaTeX ved hjælp af PythonTeX, Poore (2015).

Jeg takker alle de studerende og instruktører, som er kommet med kommentarer og rettelser til notesættet, og høre gerne om ydeligere forbedringsforslag.

Andrew Swann

Kapitel 1

Indledning og første betragtninger

NUMERISK LINEÆR ALGEBRA er studiet af metode for behandling af beregningsproblemer indenfor matricer og vektorer. Det er et hjørnesteen i computerberegninger baseret på matematiske modeller for problemstillinger fra alle videnskaber. Det er et fag der har udviklet sig hurtigt og er under stadig forandring. Grundlaget er teorien for vektorrum og lineære transformationer. Konkret arbejde med tal betyder at disse teorier skal vinkles og forfines for at behandle problemer baseret på eksperimentel eller observationsbaseret data og for at udføre beregninger på en computer. Moderne anvendelser ofte kræver store mængder data, og det kan ofte medføre at en direkte tilgang til implementering er ikke tilstrækkelig effektiv. Overraskende mange problemstillinger kan udtrykkes via matrixligninger, og ofte den bedste metode numerisk er at finde og anvende gode faktoriseringer af disse matricer. Af disse problemstillinger falder de fleste indfor variationer over en af de følgende to:

- (a) Givet en matrix A og en vektor b bestem alle vektorer x , som opfylder

$$Ax = b.$$

- (b) Givet en matrix A bestem alle vektorer v og alle tal λ således at

$$Av = \lambda v.$$

I dette første kapitel, tager vi et første kig på nogle af de grundlæggende betragtninger for emnet.

1.1 Tal og fejl

TAL I VORES DATA er yderst sjælden præcis. Måler man et lufttryk til at være 1005 hPa er der usikkerhed omkring det sidste tal; har man et computer-billed er den røde farve i et given punkt typisk gemt som et heltal mellem 0 og 255, og så kan ikke afspejle virkeligheden eksakt. Det er kun når vi tæller objekter 1, 2, 3, ... at vi får et eksakt tal at arbejde med; men også her hvis der er mange objekter eller objekterne er svære at tælle, som f.eks. ulve i Jylland, er tallet ikke nødvendigvis korrekt.

Selvom man har et eksakt tal opstår der også praktiske problemer når en computer skal opbevare og behandle den. Medmindre vi arbejder symbolsk, er tal typisk opbevaret i form der svarer til visse rationelle tal a/b . Det betyder at tal som $\sqrt{2} \approx 1,414\,21$, $e \approx 2,718\,28$, $\pi \approx 3,141\,59$ har ikke eksakte repræsentationer. Desuden er nævneren b typisk en potens af 2 (binærbrøk). Som konsekvens kan hverken $1/3 \approx 0,333\,33$ eller $1/7 \approx 0,142\,86$ behandles præcist.

Dette forværres af at computeren også giver os rige mulighed for at lave beregninger med mange tal. En computer vil sagtens kunne beregne en sum for et datasæt bestående af en million tal, men så risikerer man også at fejleffekterne bidrager en million gange.

Selv for heltal er der problemer. Computerhukommelse bliver billigere og billigere, men det er alligevel dyrt at arbejde med større heltal. Et heltal med mange cifre beslaglægger en del hukommelsesplads, og beregninger bliver langsommere.

I dette kursus har vi programmeringssproget python som eksempelsprog. Hvad et python program kan behandle er ikke fuldstændigt fastlagt og kan variere, afhængig af hvilke version og implementering der er tale om, samt egenskaber af den underliggende computer den køres på.

I python skal man skelner mellem heltal og decimalbrøk: skriver man 143 har man et heltal, ønskes derimod en decimalbrøk skrives 143.0 i stedet. Når vi bruger decimalbrøk, så viser nogle af de ovennævnte problemer sig ret hurtigt:

```
>>> 1.1 + 1.2
2.3
>>> 1.9 + 1.2
3.0999999999999996
>>> 1.1 - 1.2
-0.09999999999999987
```


python	matematik	navn
$x + y$	$x + y$	sum
$x - y$	$x - y$	differens
$x * y$	xy	produkt
x / y	x/y	kvotient
$x ** y$	x^y	potens

Tabel 1.1: Python regneoperationer på tal.

```
>>> 2.1 * 2.7
5.6700000000000001
>>> 2.1 / 2.0
1.05
>>> 2.1 / 2.5
0.8400000000000001
```

hvor vi har brugt pythons regneoperationer som i tabel 1.1. For reelle tal ville alle svar ovenfor har haft højst 2 decimaler. Desuden har vi

```
>>> 1.1 + (1.3 + 1.5)
3.9
>>> (1.1 + 1.3) + 1.5
3.9000000000000004
```

selvom begge sum er det samme for reelle tal. Når vi betragter regneoperationer, har manglende præcision nogle konsekvenser. Som eksempel lad os arbejde med decimalbrøk med kun 3 cifre efter kommaet. Så vil $a' = 10,153$ blive nødt til at repræsentere et hvilken som helst tal a mellem $a_{\min} = 10,1525$ og $a_{\max} = 10,1535$.

Hvis et reelt tal x er repræsenteret af tallet x' vil vi kalde

$$x' - x$$

for *fejlen*. Vi vil også kalde

$$\delta = \frac{x' - x}{x}$$

1 INDLEDNING OG FØRSTE BETRAGTNINGER

	beregnet	min værdi	max værdi	fejl	relativ fejl %
+	20,139	20,138 000	20,140 000	0,0010	0,004 97
−	0,167	0,166 000	0,168 000	0,0010	0,602 41
×	101,388	101,377 789	101,397 928	0,0102	0,010 07
/	1,017	1,016 622	1,016 824	0,0002	0,017 27

Tabel 1.2: Fejl i beregninger til 3 decimaler ved brug af repræsentanter $a' = 10,153$ og $b' = 9,986$.

for den *relative fejl*, så længe x er ikke 0. Den opfylder

$$x' = x(1 + \delta). \quad (1.1)$$

Ofte udtrykker vi den relative fejl i procent, ved at gange δ med 100.

For a mellem a_{\min} og a_{\max} ovenfor har vi så, at fejlen $a' - a$ ligger mellem $a' - a_{\max} = -0,0005$ og $a' - a_{\min} = 0,0005$, samt at den relative fejl ligger mellem

$$\frac{a' - a_{\max}}{a_{\max}} = -0,000\,049\,244\,1 \approx -0,004\,92\%$$

og

$$\frac{a' - a_{\min}}{a_{\min}} = 0,000\,049\,249\,0 \approx 0,004\,92\%.$$

Betragt nu $b' = 9,986$, repræsenterende et reelt tal b mellem $b_{\min} = 9,9855$ og $b_{\max} = 9,9865$. Fejlen for b' er den samme som for a' , mens den relative fejl er $\pm 0,005\,01\%$.

Summen forventes repræsenteret af $a' + b' = 20,139$, men $a + b$ ligger mellem $a_{\min} + b_{\min} = 20,138$ og $a_{\max} + b_{\max} = 20,140$. Fejlen er $\pm 0,001$, mens den relative fejl er mellem $\pm 0,004\,97\%$. For den plus + operation, ser vi at fejlen er dobbelt så stort som for a , mens i dette eksempel er den relative fejl er næsten uændret. For de andre regneoperationer difference −, produkt \times og kvotient /, kan fejlen regnes på tilsvarende måde. Resultaterne gives i tabel 1.2. Her ser vi en del udsving i fejl og relativ fejl. Mest bekymrende er den store relativ fejl for difference operationen, som afspejler færre betydende cifre i resultatet.

Generelle regneregler for disse type uligheder er givet ved:

Proposition 1.1. For reelle tal a og b med $a_{\min} \leq a \leq a_{\max}$ og $b_{\min} \leq b \leq b_{\max}$ gælder

1.2 FLYDENDE-KOMMA REPRÆSENTATION

$$(a) \ a_{\min} + b_{\min} \leq a + b \leq a_{\max} + b_{\max},$$

$$(b) \ a_{\min} - b_{\max} \leq a - b \leq a_{\max} - b_{\min}.$$

Hvis yderligere $a_{\min}, b_{\min} > 0$, så har vi

$$(c) \ a_{\min} b_{\min} \leq ab \leq a_{\max} b_{\max},$$

$$(d) \ a_{\min}/b_{\max} \leq a/b \leq a_{\max}/b_{\min}.$$

□

1.2 Flydende-komma repræsentation

Et par eksempler på tal i flydende-komma form, eller tal i videnskabelig notation, er

$$\begin{aligned} &1,639\,01 \cdot 10^2 \\ &-2,704\,62 \cdot 10^{-3} \\ &5,672\,91 \cdot 10^{100} \end{aligned}$$

Disse skrives i python som

```
>>> 1.63901e2
163.901
>>> -2.70462e-3
-0.00270462
>>> 5.67291e100
5.67291e+100
```

og som set vil tal, der er ikke for store, bliver skrevet ud som en almindelig decimalbrøk. Her benyttes grundtallet 10.

Generelt er en *flydende-komma repræsentation* med grundtal 10 er et reelt tal x af formen

$$x = \pm m \cdot 10^k$$

hvor

$$m = "d_1, d_2 \dots d_r" = d_1 + \frac{d_2}{10} + \dots + \frac{d_r}{10^{r-1}}$$

er en decimalbrøk med r cifre, $d_i \in \{0, 1, 2, \dots, 9\}$, $d_1 \neq 0$, og k er et heltal. Decimaltet m kaldes *mantissen* og k kaldes *eksponenten*. Så tallet $-2,704\,62 \cdot 10^{-3}$ har fortegn -1 , mantisse $2,704\,62$ og eksponent -3 .

I python repræsenteres alle decimaltal internt i en tilsvarende flydende-komma form, men med grundtallet 2 i stedet for 10, dvs. en binærbrøk ganget med en potens som 2^{17} . Denne type kaldes for *float*:

1 INDLEDNING OG FØRSTE BETRAGTNINGER

```
>>> type(143)
<class 'int'>
>>> type(143.0)
<class 'float'>
>>> type(-2.70462e-3)
<class 'float'>
```

Der afsættes en fast mængde hukommelse til enhver float. Vi vil arbejde med NumPy pakken. På min maskine fortæller

```
>>> import numpy as np
>>> np.finfo(float).bits
64
```

at der afsættes 64 bit = 8 byte for hver `float`. Din computer råder sikkert over hukommelse med adskillelige gigabyte = 2^{30} byte = 1 073 741 824 byte. I hver gigabyte er der så plads til at opbevare $134\,217\,728 \approx 1,3 \cdot 10^8$ tal i `float` form.

Da der afsættes 64 bit til en `float`, er der højst

$$2^{64} = 18\,446\,744\,073\,709\,551\,616 \approx 1,8 \cdot 10^{19}$$

forskellige reelle tal der kan bruges som repræsentanter i `float` form. Dette står i kontrast til at der er uendelige mange reelle tal, og har nogle væsentlige konsekvenser

(a) Der er et største tal float_{\max} og et mindste tal float_{\min} af typen `float`:

```
>>> np.finfo(float).max
1.7976931348623157e+308
>>> np.finfo(float).min
-1.7976931348623157e+308
```

(b) Der er et mindste tal $\epsilon_{\text{machine}} > 0$, *machine epsilon* eller *nøjagtigheden*, af type `float` således at $1 + \epsilon_{\text{machine}}$ er igen en `float` og $1 + \epsilon_{\text{machine}} > 1$:

```
>>> np.finfo(float).eps
2.220446049250313e-16
```

(c) Der er et mindste tal $\text{float}_{\text{tiny}}$ af type `float` som er skarpt større end 0:

1.2 FLYDENDE-KOMMA REPRÆSENTATION

```
>>> np.finfo(float).tiny
2.2250738585072014e-308
```

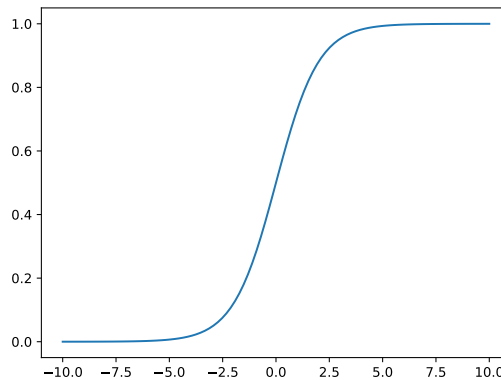
(d) Resultater af regneoperationer med tal af type `float` skal generelt afrundes for at give et svar der igen er af type `float`.

(e) Resultater der er for store kan ikke repræsenteres som `float`; dette kaldes *overflow*.

(f) Resultater der er for tæt på 0 riskærer at bliver afrundet til `0.0`; dette kaldes *underflow*.

Eksempel 1.2. I machine learning er der ofte brug for sigmoidfunktionen

$$\sigma(x) = \frac{e^x}{1 + e^x},$$



som er en differentiabel funktion med værdier skarpt mellem 0 og 1. Beregning af $\sigma(x)$ indebærer udregning af eksponentialfunktionen e^x , som hurtige antage meget store værdier for x positivt, og værdier tæt på 0 for x negativt. Eksponentialfunktionen er tilgængelig i `numpy` pakken som `np.exp`

```
>>> import numpy as np
>>> np.exp(1.75)
5.754602676005731
>>> np.exp(700.0)
1.0142320547350045e+304
>>> np.exp(-700.0)
9.85967654375977e-305
```

Prøver man en lidt større værdi, får man et svar der er ikke et `float` tal

1 INDLEDNING OG FØRSTE BETRAGTNINGER

```
>>> np.exp(750.0)
inf
```

som standard ledsaget af en advarsel

```
<stdin>:1: RuntimeWarning: overflow encountered in exp
```

Hvis vi forsøger at beregne $\sigma(750)$, får vi

```
>>> np.exp(750.0)/(1 + np.exp(750.0))
nan
```

```
<stdin>:1: RuntimeWarning: invalid value encountered in
↪ double_scalars
```

Men vi kan omskrive udtrykket for σ til

$$\sigma(x) = \frac{e^x}{1 + e^x} = \frac{e^x}{1 + e^x} \cdot \frac{e^{-x}}{e^{-x}} = \frac{1}{1 + e^{-x}}$$

Nu vil beregning af $\sigma(750)$ bruge $\text{np.exp}(-750.0)$, som afrundes til 0.0, uden nogen advarsel, og beregning af sigmoidfunktionen giver værdien

```
>>> 1/(1 + np.exp(-750.0))
1.0
```

som kan være mere brugbart. Der er en anden omskrivning

$$\sigma(x) = \frac{e^x}{1 + e^x} = \frac{e^x}{1 + e^x} \cdot \frac{e^{-x/2}}{e^{-x/2}} = \frac{e^{x/2}}{e^{x/2} + e^{-x/2}}$$

som undgår både overflow og underflow ved denne værdi

```
>>> np.exp(750.0/2)/(np.exp(750.0/2) + np.exp(-750.0/2))
1.0
```

men det hjælper ikke med udregning af $\sigma(1500)$.

△

Bemærkning 1.3. Det er ikke nødvendigvis godt, at overflow eller underflow opstår ubemærket. De er tegn på at noget er gået galt. Python kan tvinges til at stoppe ved sådan en fejl i `float`-beregninger via

```
>>> import numpy as np
>>> save_err = np.seterr(all = 'raise')
```

◇

1.3 Fejl i flydende-komma tal

Moderne computer plejer at følge en standard fastlagt af IEEE (Institute of Electrical and Electronics Engineers) for tal af type `float`. Ignorerer vi problemer med overflow og underflow, kan man generelt forvente en ideal implementering af float at

(F1) for ethvert reel tal $x \in \mathbb{R}$ findes der et tal $\text{float}(x)$ af type `float`, der repræsenterer x inden for en relativ fejl af højst $\epsilon_{\text{machine}}/2$, og

(F2) operationerne $+$, $-$, \times , $/$ og $\sqrt{\cdot}$ er implementeret på tal af type `float` således at svar er korrekt inden for en relative fejl af højst $\epsilon_{\text{machine}}/2$.

Punkt (F1) siger, at for alle reelle tal x har vi

$$\text{float}(x) = x(1 + \delta_x)$$

hvor den relative fejl δ_x opfylder $|\delta_x| \leq \epsilon_{\text{machine}}/2$. Dette er det samme, som at siger

$$|\text{float}(x) - x| \leq |x|\epsilon_{\text{machine}}/2.$$

For python `float`, vi kan forvente at mantissen afrundet til de første

```
>>> np.finfo(float).precision
15
```

cifre er korrekt.

Punkt (F2) siger tilsvarende, at for a og b af type `float` gælder

$$\begin{aligned} a +_{\text{float}} b &= (a + b)(1 + \delta_1), & a -_{\text{float}} b &= (a - b)(1 + \delta_2), \\ a *_{\text{float}} b &= ab(1 + \delta_3), & a /_{\text{float}} b &= (a/b)(1 + \delta_4), \\ \text{sqrt}(a) &= \sqrt{a}(1 + \delta_5), \end{aligned}$$

1 INDLEDNING OG FØRSTE BETRAGTNINGER

med $|\delta_i| \leq \epsilon_{\text{machine}}/2$ for alle relative fejl δ_i . Disse fejl δ_i varierer med a og b , men er altid kontrolleret af det samme tal, $\epsilon_{\text{machine}}/2$.

På trods af $\epsilon_{\text{machine}}$ er ret lille er vi stadigvæk plaget af problemerne i afsnit 1.1, især med hensyn til differenceoperationen og tab af betydende cifre.

Eksempel 1.4. Betragt andengradsligningen

$$2x^2 + 98x + \frac{1}{4} = 0.$$

Vi husker at $ax^2 + bx + c = 0$ har rødder

$$q = \frac{-b + \sqrt{b^2 - 4ac}}{2a} \quad \text{og} \quad r = \frac{-b - \sqrt{b^2 - 4ac}}{2a}.$$

I vores tilfælde er $b = 98$ og kvadratroden af diskriminanten er $\sqrt{b^2 - 4ac} = \sqrt{9602.0} \approx 97,990$. Dette betyder at når vi regner q , hvor skal vi tage differencen af b og $\sqrt{b^2 - 4ac}$, kan vi miste flere betydende cifre. Selvom vi bruger `float` kan det give fejl i den beregnede værdi for q . Vi kan delvis se dette ved at sætter q tilbage i polynomiet:

```
>>> import numpy as np
>>> a = 2.0 #kommentar: sæt 'a' til værdien 2.0
>>> b = 98.0
>>> c = 1.0 / 4.0
>>> sqrt_discriminant = np.sqrt(b**2 - 4*a*c)
>>> q = (-b + sqrt_discriminant)/(2*a)
>>> q
-0.0025511532323037045
>>> a*q**2 + b*q + c
-1.3367085216486885e-13
```

Tilgængæld er udregning af r uden dette problem

```
>>> r = (-b - sqrt_discriminant)/(2*a)
>>> r
-48.997448846767696
>>> a*r**2 + b*r + c
0.0
```


1.3 FEJL I FLYDENDE-KOMMA TAL

Husker vi at q og r opfylder

$$\begin{aligned} ax^2 + bx + c &= a(x - q)(x - r) \\ &= ax^2 - a(q + r)x + aqr \end{aligned}$$

kan vi sammenligne koefficienter af de forskellige potenser af x , som giver

$$b = -a(q + r) \quad \text{og} \quad c = aqr.$$

Den sidste kan løses for q :

$$q = \frac{c}{ar} = \frac{2c}{-b - \sqrt{b^2 - 4ac}}.$$

Bruges denne formel for q , får vi et bedre resultat:

```
>>> q_new = (2*c)/(-b - sqrt_discriminant)
>>> q_new
-0.0025511532323023406
>>> a*q_new**2 + b*q_new + c
0.0
```

Den relative fejl i q er så

```
>>> (q - q_new)/q_new
5.346312858502275e-13
```

som er væsentlig større end machine epsilon. △

Generelt afhænger valget af beregningsmetoden af fortegnet $\text{sgn}(b)$ af b .

Proposition 1.5. *Givet*

$$\text{sgn}(b) = \begin{cases} 1, & \text{for } b \geq 0, \\ -1, & \text{for } b < 0, \end{cases}$$

er rødderne r_1, r_2 af

$$ax^2 + bx + c = 0 \quad (a \neq 0, (b, c) \neq (0, 0))$$

givet ved

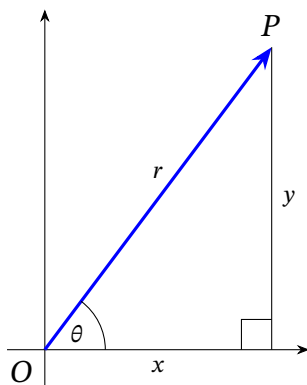
$$r_1 = \frac{-b - \text{sgn}(b)\sqrt{b^2 - 4ac}}{2a}, \quad r_2 = \frac{2c}{-b - \text{sgn}(b)\sqrt{b^2 - 4ac}}. \quad \square$$

Kapitel 2

Plangeometri: vektorer og matricer

2.1 Vektorer, norm og vinkelbestemmelse

Vi er vant til at repræsentere punkter P i planen via deres koordinater: $P = (x, y)$. F.eks. $P = (3, 4)$ er punktet med x -koordinat 3 og y -koordinat 4. Sådan et koordinatpar kan også betragtes som endepunkt for en pil \overrightarrow{OP} fra origo $O = (0, 0)$ til punktet $P = (x, y)$. Denne pil kan ses som den længste side i en retvinklet trekant med kateter af længde hhv. x og y , se figur 2.1. Den



Figur 2.1: Afstand og vinkel i planen.

2 PLANGEOMETRI: VEKTORER OG MATRICER

pythagoræiske læresætning fortæller os at \overrightarrow{OP} har længde $r = \sqrt{x^2 + y^2}$:

$$|\overrightarrow{OP}| = r = \sqrt{x^2 + y^2}.$$

Desuden kan vinklen θ bestemmes fra de trigonometriske relationer

$$c = \cos(\theta) = \frac{x}{\sqrt{x^2 + y^2}}, \quad s = \sin(\theta) = \frac{y}{\sqrt{x^2 + y^2}}.$$

Oftest har vi ikke brug for at kende selve værdien af θ , men kan nøjes med parret (c, s) .

Et koordinatpar (x, y) giver anledning til en vektor

$$v = \begin{bmatrix} x \\ y \end{bmatrix}.$$

Vi indfører *normen* af denne vektor til at være længde af \overrightarrow{OP} :

$$\|v\|_2 = \left\| \begin{bmatrix} x \\ y \end{bmatrix} \right\|_2 = \sqrt{x^2 + y^2} = r.$$

Skalæres figur 2.1 med faktoren $1/\|v\|_2$, fås en ny vektor

$$w = \frac{1}{\|v\|_2} v = \begin{bmatrix} x/\sqrt{x^2 + y^2} \\ y/\sqrt{x^2 + y^2} \end{bmatrix} = \begin{bmatrix} c \\ s \end{bmatrix}. \quad (2.1)$$

Denne vektor w har norm 1, da længden af kateteret i den nye retvinklede trekant er 1. Dette kan bekræftes med regnestykket

$$\|w\|_2^2 = \left(\frac{x}{\sqrt{x^2 + y^2}} \right)^2 + \left(\frac{y}{\sqrt{x^2 + y^2}} \right)^2 = \frac{x^2}{x^2 + y^2} + \frac{y^2}{x^2 + y^2} = 1.$$

Vektorer af norm 1 kaldes for *enhedsvektorer*. Relationen (2.1) kan bruges til at bestemme vektoren v ud fra længde r og parret (c, s) ved

$$v = \|v\|_2 w = r \begin{bmatrix} c \\ s \end{bmatrix} = \begin{bmatrix} rc \\ rs \end{bmatrix}. \quad (2.2)$$

Givet $r \geq 0$ og (c, s) med $c^2 + s^2 = 1$, fås fra (2.2) vektoren $v = \begin{bmatrix} rc \\ rs \end{bmatrix}$ med $\|v\|_2 = r$ og enhedsvektor $\frac{1}{\|v\|_2} v = \begin{bmatrix} c \\ s \end{bmatrix}$.

Bemærk at vektorerne e_0 og e_1 givet ved

$$e_0 = \begin{bmatrix} 1 \\ 0 \end{bmatrix} \quad \text{og} \quad e_1 = \begin{bmatrix} 0 \\ 1 \end{bmatrix}$$

er enhedsvektorer, som peger langs koordinataksene.

2.2 Indre produkt og matrixtransformationer

Givet to vektorer $u = \begin{bmatrix} a \\ b \end{bmatrix}$ og $v = \begin{bmatrix} x \\ y \end{bmatrix}$ sætter vi

$$\langle u, v \rangle = \left\langle \begin{bmatrix} a \\ b \end{bmatrix}, \begin{bmatrix} x \\ y \end{bmatrix} \right\rangle = ax + by.$$

Tallet $\langle u, v \rangle$ kaldes det *indre produkt* mellem u og v . Nogle steder skrives dette som $u \cdot v$, og dermed får det også navnet *prikproduktet*.

Vi får straks at

$$\langle v, v \rangle = \left\langle \begin{bmatrix} x \\ y \end{bmatrix}, \begin{bmatrix} x \\ y \end{bmatrix} \right\rangle = x^2 + y^2 = \|v\|_2^2,$$

dvs.

$$\|v\|_2 = \sqrt{\langle v, v \rangle}.$$

Med hensyn til parret $(c, s) = (\cos(\theta), \sin(\theta))$, der bestemmer vinklen θ , bemærker at vi har

$$\langle e_0, w \rangle = \left\langle \begin{bmatrix} 1 \\ 0 \end{bmatrix}, \begin{bmatrix} c \\ s \end{bmatrix} \right\rangle = 1 \cdot c + 0 \cdot s = c.$$

Givet to enhedsvektorer $w = \begin{bmatrix} \cos(\theta) \\ \sin(\theta) \end{bmatrix}$, $z = \begin{bmatrix} \cos(\varphi) \\ \sin(\varphi) \end{bmatrix}$ har vi

$$\begin{aligned} \langle w, z \rangle &= \left\langle \begin{bmatrix} \cos(\theta) \\ \sin(\theta) \end{bmatrix}, \begin{bmatrix} \cos(\varphi) \\ \sin(\varphi) \end{bmatrix} \right\rangle \\ &= \cos(\theta) \cos(\varphi) + \sin(\theta) \sin(\varphi) \\ &= \cos(\varphi - \theta) \end{aligned}$$

når vi bruger trigonometriske identiteter. Generelt har vi

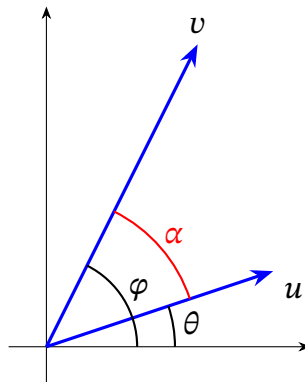
$$\langle u, v \rangle = \|u\|_2 \|v\|_2 \cos(\alpha)$$

hvor α er vinklen mellem u og v , se figur 2.2.

Det indre produkt kan skrives på en anden måde hvis indfører transponerings operation. For en vektor $u = \begin{bmatrix} a \\ b \end{bmatrix}$ er den *transponerede*

$$u^T = \begin{bmatrix} a & b \end{bmatrix},$$

2 PLANGEOMETRI: VEKTORER OG MATRICER



Figur 2.2: Vinklen mellem to vektorer.

en *rækkevektor*. Så sætter vi

$$u^T v = \begin{bmatrix} a & b \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} = ax + by = \langle u, v \rangle.$$

Denne notation har den fordel, at vi kan skrive mange indre produkter på en gang: Givet en til vektor $\tilde{u} = \begin{bmatrix} c \\ d \end{bmatrix}$ kan vi stable u^T og \tilde{u}^T ovenpå hinanden og danne matricen

$$A = \begin{bmatrix} u^T \\ \tilde{u}^T \end{bmatrix} = \begin{bmatrix} a & b \\ c & d \end{bmatrix}.$$

Vi kan nu stable deres indre produkter med v :

$$Av = \begin{bmatrix} u^T \\ \tilde{u}^T \end{bmatrix} v = \begin{bmatrix} u^T v \\ \tilde{u}^T v \end{bmatrix}, \quad (2.3)$$

dvs.

$$\begin{bmatrix} a & b \\ c & d \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} ax + by \\ cx + dy \end{bmatrix}. \quad (2.4)$$

Operationen $v \mapsto Av$ tager så et punkt i planen (x, y) og transformerer den til et nyt punkt $(ax + by, cx + dy)$. Forskellige matricer A giver forskellige transformationer. For at forstå disse transformationer, er det en ide at kikke på hvad transformationer gør med flere punkter. Givet vektorer

$$v_0 = \begin{bmatrix} x_0 \\ y_0 \end{bmatrix}, \quad v_1 = \begin{bmatrix} x_1 \\ y_1 \end{bmatrix}, \quad v_2 = \begin{bmatrix} x_2 \\ y_2 \end{bmatrix}, \quad \dots, \quad v_{n-1} = \begin{bmatrix} x_{n-1} \\ y_{n-1} \end{bmatrix}$$

2.2 INDRE PRODUKT OG MATRIXTRANSFORMATIONER

kan disse samles i en matrix

$$[v_0 \mid v_1 \mid v_2 \mid \dots \mid v_{n-1}] = \begin{bmatrix} x_0 & x_1 & x_2 & \dots & x_{n-1} \\ y_0 & y_1 & y_2 & \dots & y_{n-1} \end{bmatrix}.$$

Så kan deres indre produkter med u samles til en rækkevektor

$$\begin{aligned} u^T [v_0 \mid v_1 \mid v_2 \mid \dots \mid v_{n-1}] &= [u^T v_0 \quad u^T v_1 \quad u^T v_2 \quad \dots \quad u^T v_{n-1}] \\ &= [a \quad b] \begin{bmatrix} x_0 & x_1 & x_2 & \dots & x_{n-1} \\ y_0 & y_1 & y_2 & \dots & y_{n-1} \end{bmatrix} \\ &= [ax_0 + by_0 \quad ax_1 + by_1 \quad ax_2 + by_2 \quad \dots \quad ax_{n-1} + by_{n-1}]. \end{aligned}$$

Tilsvarende kan vi samle effekten af transformation $v \mapsto Av$ på v_0, \dots, v_{n-1} til en matrix med søjler Av_0, \dots, Av_{n-1}

$$A[v_0 \mid v_1 \mid v_2 \mid \dots \mid v_{n-1}] = [Av_0 \mid Av_1 \mid Av_2 \mid \dots \mid Av_{n-1}] \quad (2.5)$$

dvs.

$$\begin{aligned} \begin{bmatrix} a & b \\ c & d \end{bmatrix} \begin{bmatrix} x_0 & x_1 & x_2 & \dots & x_{n-1} \\ y_0 & y_1 & y_2 & \dots & y_{n-1} \end{bmatrix} \\ = \begin{bmatrix} ax_0 + by_0 & ax_1 + by_1 & ax_2 + by_2 & \dots & ax_{n-1} + by_{n-1} \\ cx_0 + dy_0 & cx_1 + dy_1 & cx_2 + dy_2 & \dots & cx_{n-1} + dy_{n-1} \end{bmatrix}. \end{aligned}$$

Lad os nu give nogle eksempler på sådanne transformationer.

Skalering

$$S = \begin{bmatrix} r & 0 \\ 0 & r \end{bmatrix}, \quad r > 0.$$

Dette sender $\begin{bmatrix} x \\ y \end{bmatrix}$ til $\begin{bmatrix} rx \\ ry \end{bmatrix} = r \begin{bmatrix} x \\ y \end{bmatrix}$, se figur 2.3.

Skalering med faktor $r = 1$ ændrer ikke på punktet (x, y) ; $\begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} x \\ y \end{bmatrix}$.
Matricen

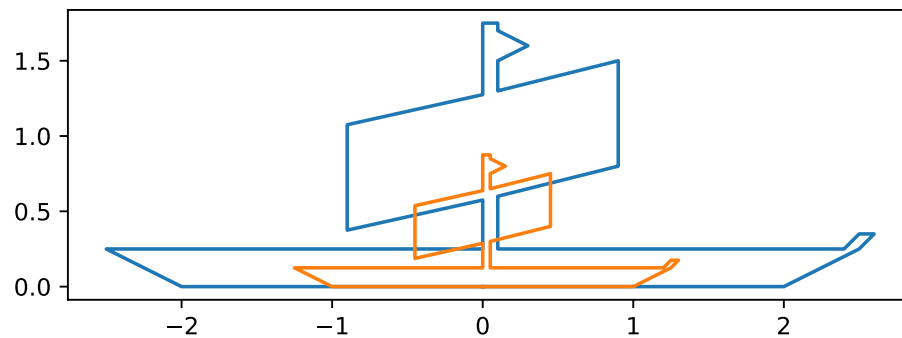
$$\begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}$$

kaldes *identitetsmatricen*.

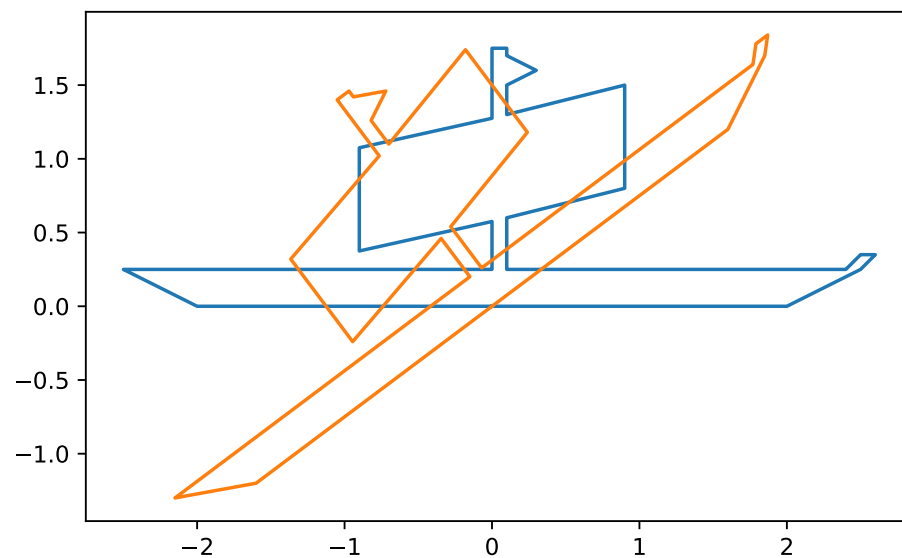
Ønsker man skalering med forskellige faktorer, r_1 i x -retningen og r_2 i y -retningen, bruger man matricen

$$\begin{bmatrix} r_1 & 0 \\ 0 & r_2 \end{bmatrix}.$$

2 PLANGEOMETRI: VEKTORER OG MATRICER



Figur 2.3: Skalering med faktor $r = 0,5$.



Figur 2.4: Rotation med $c = 0,8$, $s = \sqrt{1 - c^2} = 0,6$.

Rotation

$$R = \begin{bmatrix} c & -s \\ s & c \end{bmatrix} \quad c^2 + s^2 = 1$$

Dette sender e_0 til (c, s) , og e_1 til $(-s, c)$. Det er en rotation igennem vinklen θ , som opfylder $\cos(\theta) = c$, $\sin(\theta) = s$, se figur 2.4. I mange anvendelser har vi ikke brug for at beregne vinklen θ .

2.2 INDRE PRODUKT OG MATRIXTRANSFORMATIONER

Bemærkning 2.1. Hvis man skal bestemme θ , så kan funktionen `np.arctan2` fra NumPy pakken bruges. For en vektor $v = \begin{bmatrix} x \\ y \end{bmatrix}$ er vinklen θ givet via `np.arctan2(y, x)`, pas på rækkefølgen!

```
>>> import numpy as np
>>> np.arctan2(0, 1)
0.0
>>> np.arctan2(1, 0)
1.5707963267948966
>>> np.arctan2(1, 1)
0.7853981633974483
```

Svaret er i radianer. Ønsker man at vide hvilke multiplum af π det er skal man dele med `np.pi`. F.eks.

```
>>> np.pi
3.141592653589793
>>> np.arctan2(1, 0)/np.pi
0.5
>>> np.arctan2(1, 1)/np.pi
0.25
```

svarende til at $e_1 = (0, 1)$ har vinkel $\pi/2$ fra e_0 og at $(1, 1)$ har vinkel $\pi/4$. \diamond

Spejling

$$M = \begin{bmatrix} c & s \\ s & -c \end{bmatrix}, \quad c^2 + s^2 = 1.$$

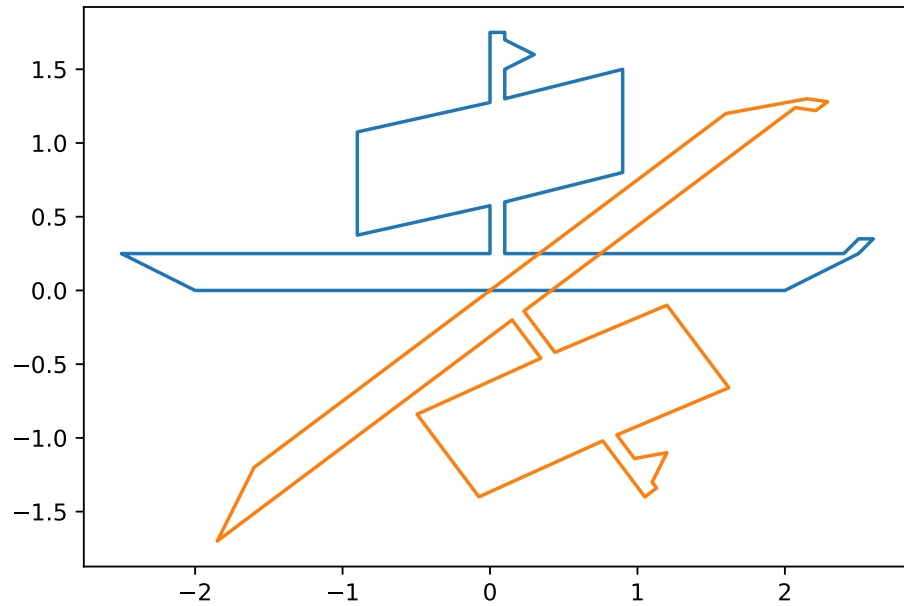
Bemærk fortegnsforskel fra rotationsmatricen; her ligger minusfortegnet på et element på diagonalen. Hvis $(c, s) = (\cos \theta, \sin \theta)$, så giver M en spejling i linjer igennem origo med vinkel $\theta/2$ til x -aksen, se figur 2.5.

Projektion

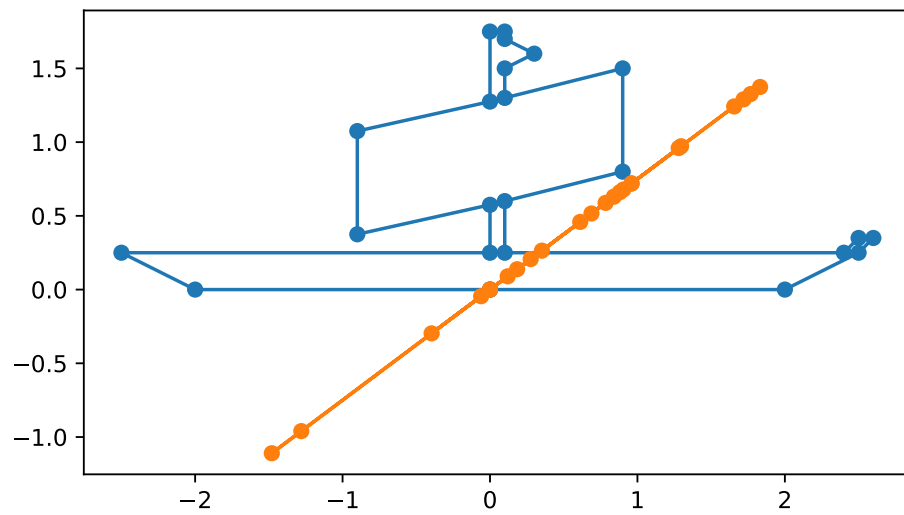
$$P = \begin{bmatrix} p^2 & pq \\ pq & q^2 \end{bmatrix}, \quad p^2 + q^2 = 1.$$

Dette sender alle punkter til punkter på linjen igennem origo og (p, q) , se figur 2.6. For $q = 0$, er $P = \begin{bmatrix} 1 & 0 \\ 0 & 0 \end{bmatrix}$ og $P \begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} x \\ 0 \end{bmatrix}$, så y -koordinatet sættes til 0, i dette tilfælde.

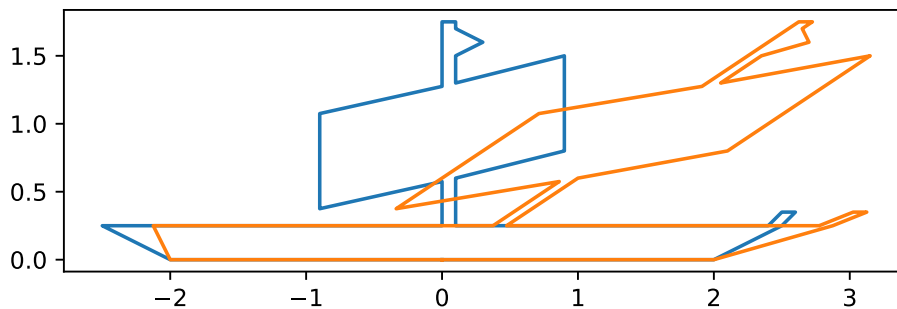
2 PLANGEOMETRI: VEKTORER OG MATRICER



Figur 2.5: Spejling med $c = 0,8$, $s = \sqrt{1 - c^2} = 0,6$.



Figur 2.6: Projektion med $p = 0,8$, $q = \sqrt{1 - p^2} = 0,6$.

Figur 2.7: Skævvridning med $t = 1,5$.**Skævvridning**

$$K = \begin{bmatrix} 1 & t \\ 0 & 1 \end{bmatrix}$$

forskubber punkter til højre ved en størrelse bestemt af y -koordinatet: $K \begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} x+ty \\ y \end{bmatrix}$, se figur 2.6.

2.3 Første tegninger

Lad os se hvordan man kan lave tegninger som ovenfor med python. Vi bruger pakken `matplotlib.pyplot` samt `numpy`:

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

Lad os begynder med nogle koordinatpar, som f.eks.

$$\begin{bmatrix} x_0 & x_1 & x_2 & x_3 \\ y_0 & y_1 & y_2 & y_3 \end{bmatrix} = \begin{bmatrix} 0,2 & 0,2 & -0,2 & 0,3 \\ -0,2 & 0,8 & 0,1 & 0,1 \end{bmatrix}.$$

Vi kan gemme x og y koordinater i to `np.array`:

```
>>> x = np.array([ 0.2, 0.2, -0.2, 0.3])
>>> y = np.array([-0.2, 0.8, 0.1, 0.1])
```

2 PLANGEOMETRI: VEKTORER OG MATRICER

Nu tegner vi linjen, som begynder i (x_0, y_0) og kører videre igennem (x_1, y_1) , (x_2, y_2) og hen til (x_3, y_3) .

```
>>> fig, ax = plt.subplots()
>>> ax.set_aspect('equal')
>>> ax.plot(x, y)
[<matplotlib.lines.Line2D object at 0x11d206c70>]
```

Afhængig af opsætningen, får man enten vist billedet straks eller man skal bede om at få den vist via

```
fig.show()
```

Lad os nu flytte vores tegning via en matrix

$$\begin{bmatrix} a & b \\ c & d \end{bmatrix} = \begin{bmatrix} 1,0 & 0,5 \\ 0,0 & 1,0 \end{bmatrix}.$$

Vi kan bruge formelen (2.4) næsten direkte; vores nye x -koordinater er $ax + by = a*x + b*y$, og den nye y -koordinater er $cx + dy = c*x + d*y$:

```
>>> a = 1.0
>>> b = 0.5
>>> c = 0.0
>>> d = 1.0
>>> fig, ax = plt.subplots()
>>> ax.set_aspect('equal')
>>> ax.plot(a*x+b*y, c*x+d*y)
[<matplotlib.lines.Line2D object at 0x11d25eac0>]
```

som vises frem på samme måde som ovenfor.

Figuren kan gemmes i en pdf fil via

```
fig.savefig('figure-name.pdf')
```

Man kan give tegningen en anden farve og/eller markere punkter ved f.eks.

```
ax.plot(a*x+b*y, c*x+d*y, color='r') #r = rød
ax.plot(a*x+b*y, c*x+d*y, marker='o') #o = cirkler/prikker
```

2.3 FØRSTE TEGNINGER

Plotgrænserne kan angives med

```
ax.set_xlim(-3.0, 3.0)  
ax.set_ylim(-2.0, 4.0)
```


Kapitel 3

Matricer og vektorer

3.1 Matricer

En *matrix* er et rektangulært skema af tal, som for eksempel

$$A = \begin{bmatrix} 3,1 & -2,1 & 4,0 & -1,6 \\ 7,2 & 3,6 & -2,7 & 11,3 \\ -5,6 & -1,2 & 3,5 & -17,2 \end{bmatrix}.$$

En matrix med m rækker og n søjler kaldes en $(m \times n)$ -matrix. Tallene der kan bruges i matricen kaldes *skalarer*. Tallet der er placeret i række i , søjle j , kaldes den (i, j) te indgang i matricen; for en matrix A skriver vi a_{ij} eller $A_{[i,j]}$ for denne indgang. Mht. konventionerne i python begynder nummerering ved $i = 0, j = 0$. Så i eksemplet overfor er A en (3×4) -matrix, med $a_{00} = 3,1$ og $a_{13} = 11,3$. Generelt, har vi

$$A = (a_{ij}) = \begin{bmatrix} a_{00} & a_{01} & \dots & a_{0,n-1} \\ a_{10} & a_{11} & \dots & a_{1,n-1} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m-1,0} & a_{m-1,1} & \dots & a_{m-1,n-1} \end{bmatrix}.$$

Mængden af alle $(m \times n)$ -matricer med reelle indgange skrives $\mathbb{R}^{m \times n}$.

To matricer A, B er *lige med hinanden* hvis de har samme størrelse $(m \times n)$ og samme indgange, dvs. $a_{ij} = b_{ij}$ for alle $i = 0, \dots, m-1, j = 0, \dots, n-1$.

3.2 Matricer i python

Pakken NumPy har en fundamental indbygget objekt `ndarray` for matricer (og vektorer og tensorer). Matricer oprettes med kommandoen `np.array`

```
>>> import numpy as np
>>> a = np.array([[3.1, -2.1, 4.0, -1.6],
...               [7.2, 3.6, -2.7, 11.3],
...               [-5.6, -1.2, 3.5, -17.2]])
>>> a
array([[ 3.1, -2.1,  4. , -1.6],
       [ 7.2,  3.6, -2.7, 11.3],
       [-5.6, -1.2,  3.5, -17.2]])
```

(prikkerne `...` i begyndelsen af linjen skrives ikke ind, de viser bare at input-linjen begyndende med `>>>` fortsætter). Dette er en 3×4 matrix

```
>>> a.shape
(3, 4)
```

I python siger man at `a` har 2 *akser*. Antallet af akser fås fra `ndim`

```
>>> a.ndim
2
```

Vi får adgang til enkelte indgange i `a` via `a[2, 3]` osv. Så de overnævnte indgange er

```
>>> a[0,0]
3.1
>>> a[1,3]
11.3
```

En hel søjle af `a` fås via `a[:, [2]]`, en række fås via `a[[1], :]`

```
>>> a[:, [2]] #søjle
array([[ 4. ],
       [-2.7],
```


3.2 MATRICER I PYTHON

```
[ 3.5]])  
>>> a[[1], :] #række  
array([[ 7.2,  3.6, -2.7, 11.3]])
```

Bemærk at der bruges ekstra parentes; glemmes disse får man resultater med kun én akse

```
>>> a[:, 2] #indgange i en søjle  
array([ 4. , -2.7,  3.5])  
>>> a[:, 2].ndim  
1  
>>> a[1, :] #indgange i en række  
array([ 7.2,  3.6, -2.7, 11.3])  
>>> a[1, :].ndim  
1
```

Indekserne begynder ved 0 og tæller op ad, men negative indekser kan bruges til at tælle ned fra den største værdi

```
>>> a[[-1], :] #sidste række  
array([[ -5.6,  -1.2,   3.5, -17.2]])  
>>> a[:, [-2]] #næstsidste søjle  
array([[ 4. ],  
       [-2.7],  
       [ 3.5]])
```

Elementerne i en ndarray har alle den samme datatype, dtype. Dette betyder at hvert element tager lige meget plads i computerens hukommelse, og gøre det nemmere og hurtigere for computeren til at finde et givent element. Vi har

```
>>> a.dtype  
dtype('float64')
```

Denne datatype bruges automatisk af python når mindst en indgang i a er givet som decimalbrøk. For at sikre, man får den rigtig datatype, kan man angive datatypen når matricen oprettes

3 MATRICER OG VEKTORER

```
>>> b = np.array([[1, 2],  
...               [3, 7]], dtype='float')  
>>> b  
array([[1., 2.],  
       [3., 7.]])  
>>> b.dtype  
dtype('float64')
```

En liste over forskellige datatypes NumPy kender som standard findes i begyndelsen af kapitel 4 af *NumPy user guide* (the NumPy community 2020). For det meste holder vi os til `float` typen. Lidt senere vil vi have brug for `complex` typen.

3.3 Heatmap plot

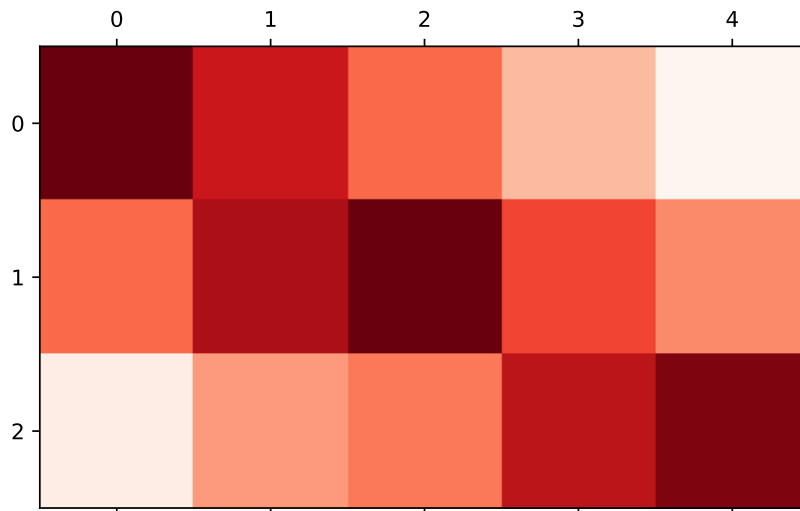
En nyttig metode for at visualisere en matrix er at farvelægge hver indgang med en farve bestemt af dens værdi. For dette kan man med fordel bruge `matshow` fra `matplotlib`.

```
import matplotlib.pyplot as plt  
import numpy as np  
  
a = np.array([[1.0, 0.5, 0.0, -0.5, -1.0],  
              [0.0, 0.7, 1.0, 0.2, -0.2],  
              [-0.9, -0.3, -0.1, 0.6, 0.9]])  
  
fig, ax = plt.subplots()  
ax.set_aspect('equal')  
ax.matshow(a, cmap='Reds')
```

Som giver figur 3.1.

Argumentet `cmap` angiver hvilke farve skala der skal anvendes. Navne som `'Reds'`, `'Blues'` eller `'Greys'` bruge en stærk farve for høje værdier og hvid for de lavste værdier. Det kan være nyttig at tilføje oplysning om hvilke farver svarer til hvilke værdi, og det vises fint med en colorbar.

3.3 HEATMAP PLOT



Figur 3.1: En første heatmap plot.

```
fig, ax = plt.subplots()
ax.set_aspect('equal')
im = ax.matshow(a, cmap='Blues')
fig.colorbar(im, shrink=0.65)
```

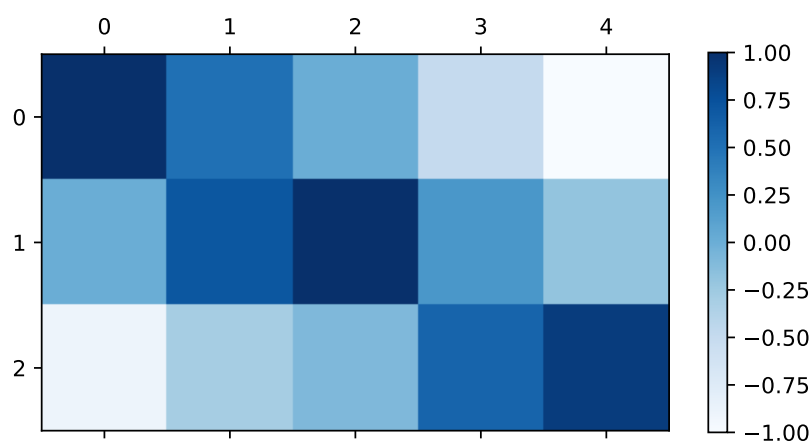
Størrelsen på colorbar styres af argumentet `shrink`, som skalere bjælken med den givne faktor.

Man kan justere hvilke værdier dækkes af farveskalaen, ved at sætte `clim`, f.eks. `clim = (-1.0, 1.0)` siger at farveskalaen skal bruges til at dække værdierne fra $-1,0$ til $1,0$.

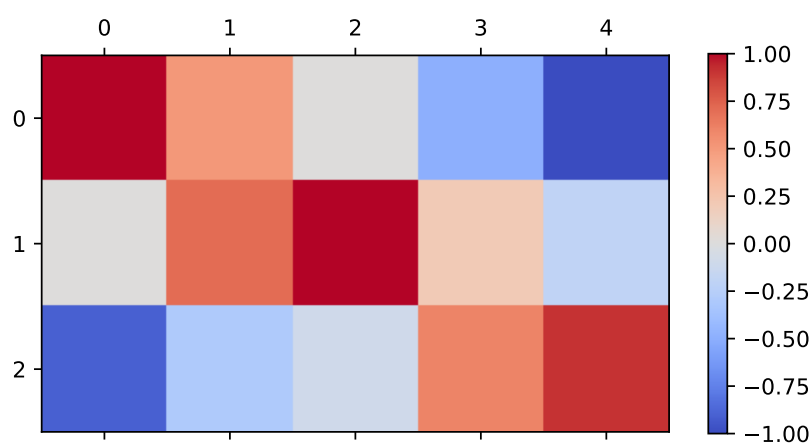
```
fig, ax = plt.subplots()
ax.set_aspect('equal')
im = ax.matshow(a, cmap='coolwarm', clim = (-1.0, 1.0))
fig.colorbar(im, shrink=0.65)
```

Her har vi brugt en farveskala der går fra blå til rød. Ved at bestemme grænser `clim = (-1.0, 1.0)` som er symmetrisk omkring $0,0$, opnås at negative værdier får en blå nuance, og positive værdier er rødlig nuance.

3 MATRICER OG VEKTORER



Figur 3.2: Heatmap med colorbar.



Figur 3.3: Heatmap med justeret farveskala.

3.4 Multiplikation med en skalar og matrixsum

Givet en matrix $A \in \mathbb{R}^{m \times n}$ som i afsnit 3.1 og en skalar $s \in \mathbb{R}$ defineres *multiplikation af A med en skalar s* som $(m \times n)$ -matricen sA hvor s er ganget ind på alle indgange i A enkeltvis:

$$sA = s(a_{ij}) := (sa_{ij}) = \begin{bmatrix} sa_{00} & sa_{01} & \dots & sa_{0,n-1} \\ sa_{10} & sa_{11} & \dots & sa_{1,n-1} \\ \vdots & \vdots & \ddots & \vdots \\ sa_{m-1,0} & sa_{m-1,1} & \dots & sa_{m-1,n-1} \end{bmatrix}.$$

I python er operationen givet via operatoren `*`

```
>>> a
array([[ 3.1, -2.1,  4. , -1.6],
       [ 7.2,  3.6, -2.7, 11.3],
       [-5.6, -1.2,  3.5, -17.2]])
>>> s = 2.0
>>> s * a
array([[ 6.2, -4.2,  8. , -3.2],
       [14.4,  7.2, -5.4, 22.6],
       [-11.2, -2.4,  7. , -34.4]])
```

I tilfældet hvor $s = 0$, er $sA = 0A = 0_{m \times n}$ *nulmatricen* i $\mathbb{R}^{m \times n}$, hvor alle indgange er 0. Hvis størrelsen af nulmatricen er underordnet, skriver vi blot 0 i stedet for $0_{m \times n}$. Funktionen `np.zeros()` giver en nulmatrix af en given størrelse; størrelsen specificeres som tuple, som f.eks. $(2, 3)$

```
>>> np.zeros((2,3))
array([[0., 0., 0.],
       [0., 0., 0.]])
```

Givet to matricer A, B af samme størrelse $m \times n$, defineres deres *matrixsum* $A + B \in \mathbb{R}^{m \times n}$ ved at dens (i, j) -indgang er summen af de tilsvarende (i, j) -

3 MATRICER OG VEKTORER

indgange i A og B :

$$A + B = (a_{ij}) + (b_{ij}) := (a_{ij} + b_{ij})$$

$$= \begin{bmatrix} a_{00} + b_{00} & a_{01} + b_{01} & \dots & a_{0,n-1} + b_{0,n-1} \\ a_{10} + b_{10} & a_{11} + b_{11} & \dots & a_{1,n-1} + b_{1,n-1} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m-1,0} + b_{m-1,0} & a_{m-1,1} + b_{m-1,1} & \dots & a_{m-1,n-1} + b_{m-1,n-1} \end{bmatrix}.$$

Dette gøres med almindelig plus operatoren $+$ i python

```
>>> a
array([[ 3.1, -2.1,  4. , -1.6],
       [ 7.2,  3.6, -2.7, 11.3],
       [-5.6, -1.2,  3.5, -17.2]])
>>> b = np.array([[2.0, 1.0, 0.0, -1.0],
...               [2.0, 1.0, 0.0, -1.0],
...               [2.0, 1.0, 0.0, -1.0]])
>>> b
array([[ 2.,  1.,  0., -1.],
       [ 2.,  1.,  0., -1.],
       [ 2.,  1.,  0., -1.]])
>>> a + b
array([[ 5.1, -1.1,  4. , -2.6],
       [ 9.2,  4.6, -2.7, 10.3],
       [-3.6, -0.2,  3.5, -18.2]])
```

Tilsvarende er *differencen* af to $(m \times n)$ -matricer

$$A - B = (a_{ij}) - (b_{ij}) := (a_{ij} - b_{ij})$$

og i python bruges almindelig minus $-$.

Proposition 3.1. *Skalar multiplikation og matrix sum har følgende regneregler:*

- (a) $0A = 0_{m \times n}$ og $1A = A$, for $0, 1 \in \mathbb{R}$, $A \in \mathbb{R}^{m \times n}$,
- (b) $A + 0_{m \times n} = A$, for $A, 0_{m \times n} \in \mathbb{R}^{m \times n}$,
- (c) $(s + t)A = sA + tA$, for $s, t \in \mathbb{R}$, $A \in \mathbb{R}^{m \times n}$,
- (d) $s(A + B) = sA + sB$, for $s \in \mathbb{R}$, $A, B \in \mathbb{R}^{m \times n}$,
- (e) $A + B = B + A$, for $A, B \in \mathbb{R}^{m \times n}$,

- (f) $A - B = A + (-1)B$, for $A, B \in \mathbb{R}^{m \times n}$,
 (g) $A + (B + C) = (A + B) + C$, for $A, B, C \in \mathbb{R}^{m \times n}$.

Bevis. Alle resultater checkes ved at sammenligne tilsvarende indgange på begge sider. \square

Skalarmultiplikation fra højre side defineres ved

$$As = (a_{ij})s := (a_{ij}s).$$

Da $a_{ij}s = sa_{ij}$, har vi

$$As = sA.$$

3.5 Transponering

Transponering er operationen $A \rightarrow A^T$ på matricer der ombytter rækker og søjler. Hvis $A \in \mathbb{R}^{m \times n}$, så er $A^T \in \mathbb{R}^{n \times m}$

```
>>> a
array([[ 3.1, -2.1,  4. , -1.6],
       [ 7.2,  3.6, -2.7, 11.3],
       [-5.6, -1.2,  3.5, -17.2]])
>>> a.T
array([[ 3.1,  7.2, -5.6],
       [-2.1,  3.6, -1.2],
       [ 4. , -2.7,  3.5],
       [-1.6, 11.3, -17.2]])
```

Mere visuelt har vi følgende

$$A = \begin{bmatrix} 3,1 & -2,1 & 4,0 & -1,6 \\ 7,2 & 3,6 & -2,7 & 11,3 \\ -5,6 & -1,2 & 3,5 & -17,2 \end{bmatrix} \mapsto \begin{bmatrix} 3,1 & 7,2 & -5,6 \\ -2,1 & 3,6 & -1,2 \\ 4,0 & -2,7 & 3,5 \\ -1,6 & 11,3 & -17,2 \end{bmatrix} = A^T$$

Matricen A bliver spejlet i diagonalen som består af de grønne indgange. I indeksnotation har vi

$$A = (a_{ij}) \mapsto A^T = (a_{ji}).$$

3.6 Vektorer

En *søjlevektor* $v \in \mathbb{R}^n$ er en $(n \times 1)$ -matrix, som vi skriver

$$v = (v_i) = \begin{bmatrix} v_0 \\ v_1 \\ \vdots \\ v_{n-1} \end{bmatrix}.$$

I tekst skriver vi $v = (v_0, v_1, \dots, v_{n-1})$. Da v er en $(n \times 1)$ -matrix en matrix, vi kan bruge de operationerne i afsnit 3.4 til at få skalarmultiplikation og vektorsum

$$sv = s(v_i) := (sv_i) = \begin{bmatrix} sv_0 \\ sv_1 \\ \vdots \\ sv_{n-1} \end{bmatrix},$$

$$u + v = (u_i) + (v_i) := (u_i + v_i) = \begin{bmatrix} u_0 + v_0 \\ u_1 + v_1 \\ \vdots \\ u_{n-1} + v_{n-1} \end{bmatrix}.$$

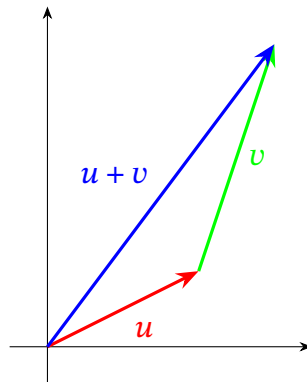
Som i kapitel 2 kan en vektor tænkes som en pil fra origo til punktet med koordinater $(v_0, v_1, \dots, v_{n-1})$. Vektorsum svarer til at sætte to piler efter hinanden, som i figur 3.4.

I python kan vi enten skriver en søjlevektor via sammen metoden for en $(n \times 1)$ -matrix

```
>>> v = np.array([[3.0], [4.0], [-0.7], [6.2]])
>>> v
array([[ 3. ],
       [ 4. ],
       [-0.7],
       [ 6.2]])
```

eller vi omformer en array med en akse

```
>>> np.array([3.0, 4.0, -0.7, 6.2])[:, np.newaxis]
array([[ 3. ],
```

Figur 3.4: Vektorsum af $u = (2, 1)$ og $v = (1, 3)$.

```
[ 4. ],
[-0.7],
[ 6.2]])
```

Transponering af en søjlevektor $v \in \mathbb{R}^n$ giver en *rækkevektor* $v^T \in \mathbb{R}^{1 \times n}$:

```
>>> v
array([[ 3. ],
       [ 4. ],
       [-0.7],
       [ 6.2]])
>>> v.T
array([[ 3. ,  4. , -0.7,  6.2]])
```


Kapitel 4

Matrix multiplikation

I afsnit 2.2 så vi hvordan man kan bruge en (2×2) -matrix A til at flytte på vektorer v i \mathbb{R}^2 , via $v \mapsto Av$, se (2.3). Ved at stille flere vektorer op efter hinanden fik så vi også (2.5) hvordan A flytter en $(2 \times n)$ -matrix $[v_0 \mid v_1 \mid v_2 \mid \dots \mid v_{n-1}]$ til $[Av_0 \mid Av_1 \mid Av_2 \mid \dots \mid Av_{n-1}]$. Lad os nu udvide disse processer til matricer af vilkårlig størrelse.

4.1 Række-søjleprodukt

Vi begynder med et produkt for en rækkevektor med en søjlevektor. Hvis $u^T \in \mathbb{R}^{1 \times n}$ og $v \in \mathbb{R}^{n \times 1}$ er en rækkevektor og en søjlevektor *med det samme antal indgange* så er deres *række-søjleprodukt* givet ved

$$u^T v = [u_0 \quad u_1 \quad \dots \quad u_{n-1}] \begin{bmatrix} v_0 \\ v_1 \\ \vdots \\ v_{n-1} \end{bmatrix} = u_0 v_0 + u_1 v_1 + \dots + u_{n-1} v_{n-1}.$$

Eksempel 4.1. Et eksempel med $n = 3$ er

$$\begin{bmatrix} 3 & -1 & 2 \end{bmatrix} \begin{bmatrix} 6 \\ 5 \\ -4 \end{bmatrix} = 3 \times 6 + (-1) \times 5 + 2 \times (-4) \\ = 18 - 5 - 8 = 5.$$

Δ

4 MATRIX MULTIPLIKATION

Eksempel 4.2. For $u^T = [1 \ 1 \ \dots \ 1] \in \mathbb{R}^{1 \times n}$ er

$$\begin{aligned} u^T v &= [1 \ 1 \ \dots \ 1] \begin{bmatrix} v_0 \\ v_1 \\ \vdots \\ v_{n-1} \end{bmatrix} = 1 \times v_0 + 1 \times v_1 + \dots + 1 \times v_{n-1} \\ &= v_0 + v_1 + \dots + v_{n-1} \end{aligned}$$

summen af indgangene i v . Ligeledes, for

$$w^T = (1/n)u^T = [1/n \ 1/n \ \dots \ 1/n] \in \mathbb{R}^{1 \times n}$$

er

$$w^T v = \frac{1}{n}(v_0 + v_1 + \dots + v_{n-1}),$$

som er middelværdien af indgangene i v . Δ

Eksempel 4.3. Betragt et polynomium

$$p(x) = a_0 + a_1 x + a_2 x^2 + \dots + a_{n-1} x^{n-1}.$$

Vi gemmer koefficienterne a_i i en rækkevektor u^T :

$$u^T = [a_0 \ a_1 \ \dots \ a_{n-1}] \in \mathbb{R}^{1 \times n}.$$

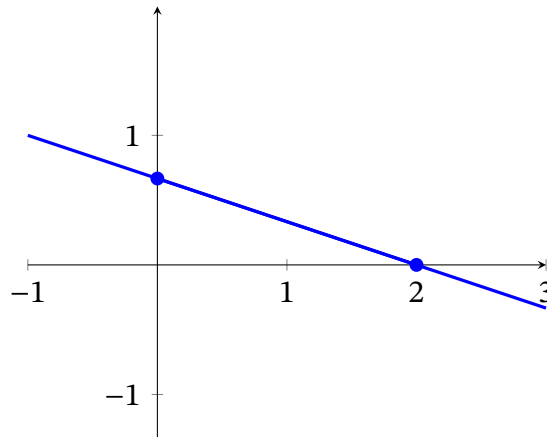
Givet et x_0 kan vi betragte Vandermondevektoren

$$v = \begin{bmatrix} 1 \\ x_0 \\ x_0^2 \\ \vdots \\ x_0^{n-1} \end{bmatrix} \in \mathbb{R}^{n \times 1}.$$

Række-søjleproduktet af u^T med v er så

$$\begin{aligned} u^T v &= [a_0 \ a_1 \ \dots \ a_{n-1}] \begin{bmatrix} 1 \\ x_0 \\ x_0^2 \\ \vdots \\ x_0^{n-1} \end{bmatrix} \\ &= a_0 + a_1 x_0 + a_2 x_0^2 + \dots + a_{n-1} x_0^{n-1} = p(x_0) \end{aligned}$$

evaluering af $p(x)$ i $x = x_0$. Δ

Figur 4.1: Linjen $x + 3y = 2$.

Eksempel 4.4. Givet en rækkevektor $u^T = [a \ b] \in \mathbb{R}^{1 \times 2}$ som er ikke nul, og en værdi $c \in \mathbb{R}$, kan vi kikke på punkterne (x, y) således at $v = \begin{bmatrix} x \\ y \end{bmatrix}$ opfylder $u^T v = c$. Skrives dette ud får vi

$$ax + by = c$$

som er ligningen for en ret linje i \mathbb{R}^2 . F.eks. for $(a, b) = (1, 3)$ og $c = 2$ har vi

$$x + 3y = 2.$$

Sættes $y = 0$, fås $x = 2$, så denne linje skær x -aksen i punktet $(2, 0)$. Tilsvarende hvis vi sætter $x = 0$ fås $y = 2/3$, så linjen går også igennem $(0, 2/3)$, se figur 4.1.

Tilsvarende for $u^T = [a \ b \ c] \in \mathbb{R}^{1 \times 3}$, $d \in \mathbb{R}$ og $v = \begin{bmatrix} x \\ y \\ z \end{bmatrix} \in \mathbb{R}^{3 \times 1}$, er $u^T v = d$ planen

$$ax + by + cz = d.$$

△

4.2 Matrixprodukt

4 MATRIX MULTIPLIKATION

Definition 4.5. For to matricer $A \in \mathbb{R}^{m \times n}$ og $B \in \mathbb{R}^{n \times r}$ defineres deres *matrixprodukt* til at være matricen $AB \in \mathbb{R}^{m \times r}$ hvis (i, j) -indgang er række-søjleproduktet af rækken i fra A med søjlen j fra B . Dvs. $C = AB$ har (i, j) -indgang

$$c_{ij} = a_{i0}b_{0j} + a_{i1}b_{1j} + a_{i2}b_{2j} + \cdots + a_{i,n-1}b_{n-1,j}.$$

Eksempel 4.6. For et eksempel med $m = 2$, $n = 3$, $r = 4$ tager vi

$$A = \begin{bmatrix} 3 & -1 & 2 \\ 0 & 1 & 1 \end{bmatrix} \in \mathbb{R}^{2 \times 3} \quad \text{og} \quad B = \begin{bmatrix} 6 & 1 & 0 & 0 \\ 5 & 1 & 1 & 2 \\ -4 & 1 & 0 & 2 \end{bmatrix} \in \mathbb{R}^{3 \times 4}.$$

Så er produktet AB givet ved

$$AB = \begin{bmatrix} 3 & -1 & 2 \\ 0 & 1 & 1 \end{bmatrix} \begin{bmatrix} 6 & 1 & 0 & 0 \\ 5 & 1 & 1 & 2 \\ -4 & 1 & 0 & 2 \end{bmatrix} = \begin{bmatrix} 5 & 4 & -1 & 2 \\ 1 & 2 & 1 & 4 \end{bmatrix}.$$

F.eks. indgangen nederst til højre er række-søjleproduktet

$$\begin{bmatrix} 0 & 1 & 1 \end{bmatrix} \begin{bmatrix} 0 \\ 2 \\ 2 \end{bmatrix} = 0 \times 0 + 1 \times 2 + 1 \times 2 = 4.$$

Bemærk at for disse matricer er produktet BA ikke defineret: $B \in \mathbb{R}^{3 \times 4}$ og $A \in \mathbb{R}^{2 \times 3}$, men $4 \neq 2$. △

I python er der naturligvis en funktion der vil beregne matrixproduktet for os, givet ved operatoren `@`. Så eksemplet ovenfor er

```
>>> import numpy as np
>>> a = np.array([[3.0, -1.0, 2.0],
...               [0.0, 1.0, 1.0]])
>>> b = np.array([[6.0, 1.0, 0.0, 0.0],
...               [5.0, 1.0, 1.0, 2.0],
...               [-4.0, 1.0, 0.0, 2.0]])
>>> a @ b
array([[ 5.,  4., -1.,  2.],
       [ 1.,  2.,  1.,  4.]])
```

Python er også enig at produktet BA er ikke defineret for disse to matricer

```

>>> b @ a
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: matmul: Input operand 1 has a mismatch in its
  ↳ core dimension 0, with gufunc signature
  ↳ (n?,k),(k,m?)->(n?,m?) (size 2 is different from 4)

```

Advarsel 4.7. Matrix multiplikation er *ikke* givet ved *, man skal bruge pythons @ operator. !

Matrixproduktet kan fortolkes på forskellige måde. Bemærk først at hvis vi deler A op i dens m rækker, og B op i dens r søjler har vi tre forskellige måde at se på produktet:

$$\begin{aligned}
 AB &= \begin{bmatrix} a_0^T \\ a_1^T \\ a_2^T \\ \vdots \\ a_{m-1}^T \end{bmatrix} [b_0 \mid b_1 \mid b_2 \mid \dots \mid b_{r-1}] \\
 &= \begin{bmatrix} a_0^T b_0 & a_0^T b_1 & a_0^T b_2 & \dots & a_0^T b_{r-1} \\ a_1^T b_0 & a_1^T b_1 & a_1^T b_2 & \dots & a_1^T b_{r-1} \\ a_2^T b_0 & a_2^T b_1 & a_2^T b_2 & \dots & a_2^T b_{r-1} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ a_{m-1}^T b_0 & a_{m-1}^T b_1 & a_{m-1}^T b_2 & \dots & a_{m-1}^T b_{r-1} \end{bmatrix} \quad (4.1)
 \end{aligned}$$

$$= [Ab_0 \mid Ab_1 \mid Ab_2 \mid \dots \mid Ab_{r-1}] \quad (4.2)$$

$$= \begin{bmatrix} a_0^T B \\ a_1^T B \\ a_2^T B \\ \vdots \\ a_{m-1}^T B \end{bmatrix}. \quad (4.3)$$

4 MATRIX MULTIPLIKATION

Udtrykket (4.1) er selve definitionen på matrixprodukt, som en matrix af række-søjleprodukter.

I (4.2), ser vi at søjlerne af AB består af A ganget på søjlerne af B . Hvert søjle b_i af B er et punkt i \mathbb{R}^n , og Ab_i taget dette punkt og giver et nyt punkt i \mathbb{R}^m . Produktet AB er så effekten af transformationen $v \mapsto Av$, $\mathbb{R}^n \rightarrow \mathbb{R}^m$, på søjlerne af B . For $m = n = 2$, er det dette syn vi havde på matrixprodukt i afsnit 2.2.

Her er et par eksempler i denne stil.

Eksempel 4.8. Lad

$$R_0 = \begin{bmatrix} 1 & 0 & 0 \\ 0 & c & -s \\ 0 & s & c \end{bmatrix} \quad \text{med } c^2 + s^2 = 1.$$

Så er R_0 en rotationsmatrix udvidet til en (3×3) -matrix. På et punkt $v = (x, y, z) \in \mathbb{R}^3$ er

$$R_0 v = \begin{bmatrix} 1 & 0 & 0 \\ 0 & c & -s \\ 0 & s & c \end{bmatrix} \begin{bmatrix} x \\ y \\ z \end{bmatrix} = \begin{bmatrix} x \\ cy - sz \\ sy + cz \end{bmatrix},$$

som er igen et punkt i \mathbb{R}^3 . Transformationen holder x -koordinatet fast, og udfører en rotation i (y, z) -planen.

Tilsvarende er

$$R_2 = \begin{bmatrix} c & -s & 0 \\ s & c & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

en rotation i (x, y) -planen og

$$R_1 = \begin{bmatrix} c & 0 & -s \\ 0 & 1 & 0 \\ s & 0 & c \end{bmatrix}$$

en rotation i (x, z) -planen. △

Eksempel 4.9. Betragt matricen

$$P = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \end{bmatrix}.$$

P er en (2×3) -matrix, så giver en transformation fra \mathbb{R}^3 til \mathbb{R}^2 . Mere præcist har vi for $v = (x, y, z) \in \mathbb{R}^3$ at

$$Pv = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \end{bmatrix} = \begin{bmatrix} x \\ y \end{bmatrix},$$

dvs. at Pv består kun af de første to koordinater fra v . △

Ligeledes beskriver udtrykket (4.3) rækkerne i matrixproduktet AB som effekten af en transformation $u^T \mapsto u^T B$, $\mathbb{R}^{1 \times n} \rightarrow \mathbb{R}^{1 \times r}$.

Eksempel 4.10. Følgende matrixprodukt

$$\begin{bmatrix} 1 & 0 & 0 & 0 & \dots & 0 \\ 1 & 1 & 0 & 0 & \dots & 0 \\ 1 & 1 & 1 & 0 & \dots & 0 \\ \vdots & & & \ddots & & \\ 1 & 1 & 1 & 1 & \dots & 1 \end{bmatrix} \begin{bmatrix} v_0 \\ v_1 \\ v_2 \\ \vdots \\ v_{n-1} \end{bmatrix} = \begin{bmatrix} v_0 \\ v_0 + v_1 \\ v_0 + v_1 + v_2 \\ \vdots \\ v_0 + v_1 + v_2 + \dots + v_{n-1} \end{bmatrix}$$

giver en løbende sum af indgangene i v . △




Især for matrix-vektorprodukter $Ax \in \mathbb{R}^m$, af $A \in \mathbb{R}^{m \times n}$ og $x \in \mathbb{R}^n$ er der to yderlige måde at anskue produktet. For det første, hvis vi betragte indgangene i x som variable og vælger et fast $b \in \mathbb{R}^m$, så er

$$Ax = b$$

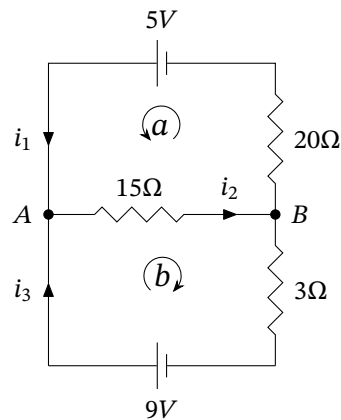
et lineært ligningssystem

$$\begin{array}{ccccccc} a_{00}x_0 + & a_{01}x_1 + \dots + & a_{0,n-1}x_{n-1} & = & b_0, \\ a_{10}x_0 + & a_{11}x_1 + \dots + & a_{1,n-1}x_{n-1} & = & b_1, \\ \vdots & \vdots & \vdots & & \vdots \\ a_{m-1,0}x_0 + a_{m-1,1}x_1 + \dots + a_{m-1,n-1}x_{n-1} & = & b_{m-1}, \end{array}$$

bestående af m ligninger i de n variable x_0, x_1, \dots, x_{n-1} . Sådanne systemer optræder i mange forskellige sammenhæng.

Eksempel 4.11. Betragt det elektriske kredsløb givet i figur 4.2. Det består af to batterier  og tre elektriske modstander  forbundet som vist. Pilene  angiver den elektriske strøm i ledningen.

4 MATRIX MULTIPLIKATION



Figur 4.2: Elektrisk kredsløb.

Den første lov af Kirchhoff fortæller at summen de elektriske strøm regnet med fortegn mod et knudepunkt er nul. Vores kreds har to knudepunkter markeret A og B . Kirchhoffs lov giver så to ligninger

$$\text{knudepunkt } A: \quad i_1 - i_2 + i_3 = 0,$$

$$\text{knudepunkt } B: \quad -i_1 + i_2 - i_3 = 0.$$

Kirchhoffs anden lov siger at sum af spændings fald over komponenter i hver delløkke af kredset er lige med spændingen fra batteriet, eller nul hvis der er ingen strømkilde i delkredsen. Spændingsfaldet V over en elektrisk modstand R er givet ved Ohms lov $V = IR$, hvor I er den elektriske strøm igennem komponenten. Vores figur har to lukkede delkredse, angivet som a og b , og vi har så

$$\text{løkke } a: \quad 20i_1 + 15i_2 = 5,$$

$$\text{løkke } b: \quad 15i_2 + 3i_3 = 9.$$

Vi har i alt 4 lineære ligninger i de 3 ubekendte i_1, i_2, i_3 . Disse 4 ligninger kan skrives som matrixligningen

$$\begin{bmatrix} 1 & -1 & 1 \\ -1 & 1 & -1 \\ 20 & 15 & 0 \\ 0 & 15 & 3 \end{bmatrix} \begin{bmatrix} i_1 \\ i_2 \\ i_3 \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ 5 \\ 9 \end{bmatrix}.$$

△

4.3 REGNEREGLER FOR MATRIXPRODUKT

En anden måde at anskue produktet på, især når vi har en matrix-vektorprodukt $Ax \in \mathbb{R}^m$, af $A \in \mathbb{R}^{m \times n}$ og $x \in \mathbb{R}^n$, er at dele A op i søjler og skrive x ud i koordinater

$$Ax = [v_0 \mid v_1 \mid v_2 \mid \dots \mid v_{n-1}] \begin{bmatrix} x_0 \\ x_1 \\ \vdots \\ x_{n-1} \end{bmatrix} = x_0 v_0 + x_1 v_1 + \dots + x_{n-1} v_{n-1}.$$

Dette er en *lineær kombination* af søjlerne v_0, v_1, \dots, v_{n-1} af A .

Eksempel 4.12. For

$$A = \begin{bmatrix} 3 & -1 & 2 \\ 0 & 1 & 1 \end{bmatrix} \quad \text{og} \quad x = \begin{bmatrix} -2 \\ 5 \\ 7 \end{bmatrix}$$

er

$$Ax = \begin{bmatrix} 3 & -1 & 2 \\ 0 & 1 & 1 \end{bmatrix} \begin{bmatrix} -2 \\ 5 \\ 7 \end{bmatrix} = (-2) \begin{bmatrix} 3 \\ 0 \end{bmatrix} + 5 \begin{bmatrix} -1 \\ 1 \end{bmatrix} + 7 \begin{bmatrix} 2 \\ 1 \end{bmatrix}.$$

△

4.3 Regneregler for matrixprodukt

Proposition 4.13. *Matrixproduktet opfylder følgende regneregler i forhold til sum og skalarmultiplikation*

- (a) $A(B + C) = AB + AC$, for $A \in \mathbb{R}^{m \times n}$, $B, C \in \mathbb{R}^{n \times r}$,
- (b) $(A + B)C = AC + BC$, for $A, B \in \mathbb{R}^{m \times n}$, $C \in \mathbb{R}^{n \times r}$,
- (c) $A(sB) = s(AB) = (sA)B$, for $s \in \mathbb{R}$, $A \in \mathbb{R}^{m \times n}$, $B \in \mathbb{R}^{n \times r}$. □

Især vigtigt er at

Sætning 4.14. For $A \in \mathbb{R}^{m \times n}$, $B \in \mathbb{R}^{n \times r}$, $C \in \mathbb{R}^{r \times p}$,

$$A(BC) = (AB)C.$$

Bevis. Skriv $A = (a_{ij})$, $B = (b_{jk})$, $C = (c_{k\ell})$ for $i \in \{0, \dots, m-1\}$, $j \in \{0, \dots, n-1\}$, $k \in \{0, \dots, r-1\}$, $\ell \in \{0, \dots, p-1\}$. Så er den (i, ℓ) te indgang af $A(BC)$ lige med

$$\sum_{j=0}^{n-1} a_{ij} \left(\sum_{k=0}^{r-1} b_{jk} c_{k\ell} \right).$$

4 MATRIX MULTIPLIKATION

Men dette udtryk er lige med

$$\sum_{j=0}^{n-1} \sum_{k=0}^{r-1} a_{ij} (b_{jk} c_{k\ell}) = \sum_{j=0}^{n-1} \sum_{k=0}^{r-1} a_{ij} b_{jk} c_{k\ell} = \sum_{k=0}^{r-1} \sum_{j=0}^{n-1} a_{ij} b_{jk} c_{k\ell} = \sum_{k=0}^{r-1} \left(\sum_{j=0}^{n-1} a_{ij} b_{jk} \right) c_{k\ell},$$

som er netop den (i, ℓ) te indgang af $(AB)C$. \square

Specielt for $v \in \mathbb{R}^r$ har vi

$$A(Bv) = (AB)v,$$

som siger at transformationen $v \mapsto (AB)v$ er det samme som at transformere v under B og derefter transformere resultatet med A .

Fra de overnævnte egenskaber ser det ud til at matrix multiplikation ligner meget produkt operationen for almindelig tal, men der er nogle væsentlige forskelle.

Advarsel 4.15. Produkterne AB og BA er generelt ikke ens. !

Vi så endda i eksempel 4.6 at selvom AB er defineret, kan det være at BA ikke giver mening. Selv når begge produkter eksisterer er de oftest forskellige. Dette kan vi nemt se i python, det er ikke et problem der skyldes unøjagtighed.

```
>>> import numpy as np
>>> a = np.array([[1.0, 2.0],
...               [0.0, 1.0]])
>>> b = np.array([[1.0, 0.0],
...               [1.0, 2.0]])
>>> a @ b
array([[3., 4.],
       [1., 2.]])
>>> b @ a
array([[1., 2.],
       [1., 4.]])
```

Advarsel 4.16. $AB = 0$ kan ske selvom A og B er ikke nulmatricer. !

```
>>> a = np.array([[1.0, 1.0],
...               [2.0, 2.0]])
```

```
>>> b = np.array([[1.0, 2.0],
...               [-1.0, -2.0]])
>>> a @ b
array([[0., 0.],
       [0., 0.]])
```

Som konsekvens har vi

Advarsel 4.17. $CA = CB$ medfører *ikke* at A er lig med B , selvom C er ikke en nulmatrix. !

4.4 Identitetsmatricer

Definition 4.18. Den $(n \times n)$ -identitetsmatrix er

$$I_n = \begin{bmatrix} 1 & 0 & 0 & \dots & 0 \\ 0 & 1 & 0 & \dots & 0 \\ 0 & 0 & 1 & \dots & 0 \\ \vdots & \vdots & & \ddots & \\ 0 & 0 & 0 & \dots & 1 \end{bmatrix} \in \mathbb{R}^{n \times n}.$$

Den (i, j) te indgang af I_n er 1 for $i = j$, og 0 for $i \neq j$.

Proposition 4.19. For $A \in \mathbb{R}^{m \times n}$ er

$$I_m A = A = A I_n.$$

□

I python er identitetsmatricer givet ved `np.eye`:

```
>>> import numpy as np
>>> a = np.array([[1.3, -3.0, 0.7, 1.4],
...               [2.0, 4.2, -5.6, 6.2],
...               [1.5, -2.5, 3.2, 5.4]])
>>> np.eye(3) @ a
array([[ 1.3, -3. ,  0.7,  1.4],
       [ 2. ,  4.2, -5.6,  6.2],
       [ 1.5, -2.5,  3.2,  5.4]])
```

4 MATRIX MULTIPLIKATION

```
>>> a @ np.eye(4)
array([[ 1.3, -3. ,  0.7,  1.4],
       [ 2. ,  4.2, -5.6,  6.2],
       [ 1.5, -2.5,  3.2,  5.4]])
```

4.5 Ydre produkt

Vi begyndte kapitlet med række-søjleproduktet, som giver en skalar. Omvendt hvis vi har en søjle vektor $v \in \mathbb{R}^{m \times 1}$ og en rækkevektor $w^T \in \mathbb{R}^{1 \times n}$ kan vi danne matrixproduktet $vw^T \in \mathbb{R}^{m \times n}$. Resultatet kaldes det *ydre produkt* af v med w^T . I komponenter har vi

$$\begin{aligned} vw^T &= \begin{bmatrix} v_0 \\ v_1 \\ \vdots \\ v_{m-1} \end{bmatrix} \begin{bmatrix} w_0 & w_1 & \dots & w_{n-1} \end{bmatrix} \\ &= \begin{bmatrix} v_0 w_0 & v_0 w_1 & \dots & v_0 w_{n-1} \\ v_1 w_0 & v_1 w_1 & \dots & v_1 w_{n-1} \\ \vdots & \vdots & \ddots & \vdots \\ v_{m-1} w_0 & v_{m-1} w_1 & \dots & v_{m-1} w_{n-1} \end{bmatrix} \in \mathbb{R}^{m \times n}. \end{aligned}$$

Vi kan bruge `np.matshow` til at plotte nogle eksempler på ydre produkter. Resultaterne gives i figur 4.3.

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

v = np.array([0.0, 1.0, 0.0, 1.0])[:, np.newaxis]
wt = np.array([[1.0, 0.0, 1.0, 1.0]])
v @ wt

fig, ax = plt.subplots()
ax.matshow(v @ wt, cmap='Reds')
```

```
w2t = np.linspace(0, 1, 10)[np.newaxis, :]
fig, ax = plt.subplots()
ax.matshow(v @ w2t, cmap='Blues')
```

```
v2 = np.linspace(0, 1, 10)[:, np.newaxis]
fig, ax = plt.subplots()
ax.matshow(v2 @ w2t, cmap='coolwarm')
```

For det sidste eksempel bruger vi nogle tilfældige vektorer. Disse frembringes ved hjælp en tilfældighedsgenerator `rng = np.random.default_rng()` og derefter en funktionen som `rng.random()`, for en uniform fordeling på $[0, 1)$, eller `rng.standard_normal()`, for en standard normalfordeling:

```
# opstil en tilfældighedsgenerator
rng = np.random.default_rng()
# 5 tilfældige tal i [0.0, 1.0)
vr = rng.random(5)[:, np.newaxis]
print(vr)
```

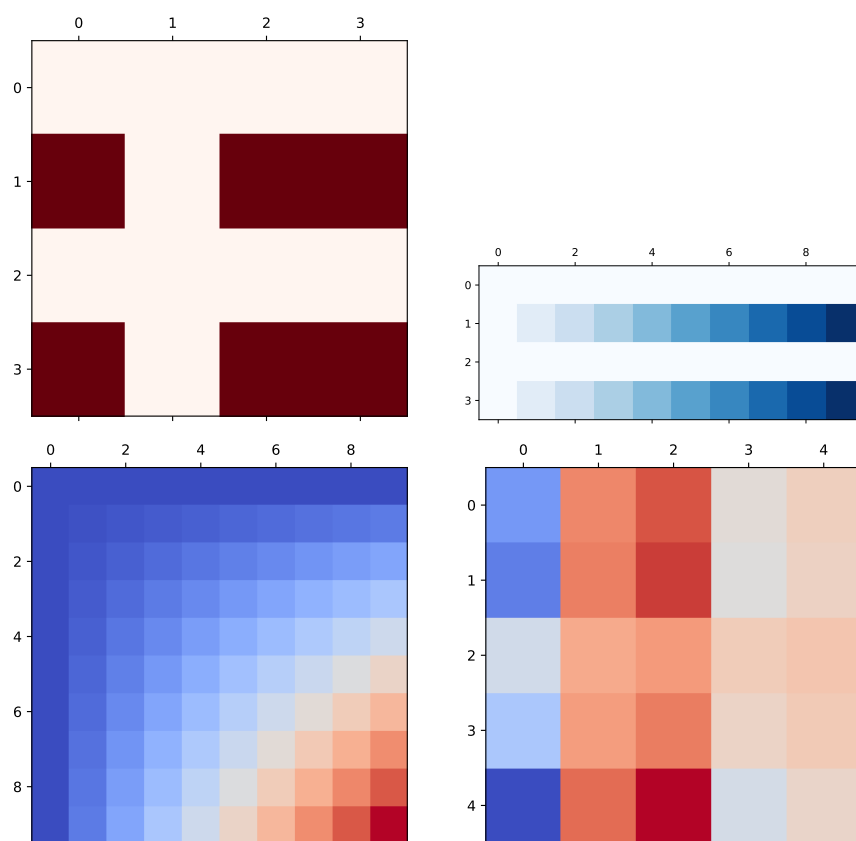
```
[[0.51865705]
 [0.59319632]
 [0.21433153]
 [0.34629232]
 [0.72597023]]
```

```
# 5 tilfældige tal normalfordelt med middelværdi 0
# og varians 1
wrt = rng.standard_normal(5)[np.newaxis, :]
print(wrt)
```

```
[[-1.12553805  0.35572775  0.62184594 -0.30729529 -0.15813279]]
```

```
fig, ax = plt.subplots()
ax.matshow(vr @ wrt, cmap='coolwarm')
```

4 MATRIX MULTIPLIKATION



Figur 4.3: Eksempler på ydre produkter.

Kapitel 5

Numerisk håndtering af matricer

5.1 Omkostninger ved matrixberegning

Matrixberegninger dækker over mange små aritmetiske operationer på indgangene. Det er derfor nødvendigt at have en idé om hvor mange ressourcer de beslaglægger ved computeren, og hvilke metoder er mindre ressourcekrævende end andre. Desuden har vi også set at hver operation med `float` risikerer at tilføje usikkerhed til resultatet, så det er godt at vide hvilke metoder bruger færre af disse operationer.

En simpel, men anvendelig, model, siger at hver af operationerne $+$, $-$, $*$ og $/$ på `floats` bruger den samme mængde tid eller ressource, kaldet en *flop*. Omkostninger ved en beregning gives så som et vist antal flops.

Som første eksempel lad os kikke på multiplikation af en vektor $v \in \mathbb{R}^{n \times 1}$ med en skalar $s \in \mathbb{R}$. Ved beregning af $u = sv$ udføres følgende operationer:

SKALAR-VEKTORPRODUKT(s, v)

```
1 for  $i \in \{0, 1, \dots, n - 1\}$ :  
2      $u_i = s * v_i$   
3 return  $u$ 
```

Dvs. for hvert heltal i fra 0 til $n - 1$, udføres én `float` operation $*$. Da der er n tal i $\{0, 1, \dots, n - 1\}$, koster SKALAR-VEKTORPRODUKT(s, v) $n \times 1 = n$ flops.

For et række-søjleprodukt $u^T v$, hvor $u^T \in \mathbb{R}^{1 \times n}$ og $v \in \mathbb{R}^{n \times 1}$ har vi tilsvarende

5 NUMERISK HÅNDTERING AF MATRICER

RÆKKE-SØJLEPRODUKT(u, v)

```

1   $c = 0$ 
2  for  $i \in \{0, 1, \dots, n-1\}$ :
3       $c = c + u_i * v_i$ 
4  return  $c$ 
```

Denne gang er der to **float**-operationer i linje 3: en gang $+$ og en gang $*$. Vi får så at RÆKKE-SØJLEPRODUKT(u, v) koster $2n$ flops.

Et matrixprodukt AB , for $A \in \mathbb{R}^{m \times n}$, $B \in \mathbb{R}^{n \times r}$ består blot af $m \times r$ række-søjleprodukter:

MATRIXPRODUKT(A, B)

```

1  for  $i \in \{0, 1, \dots, m-1\}$ :
2      for  $j \in \{0, 1, \dots, r-1\}$ :
3           $c_{ij} = \text{RÆKKE-SØJLEPRODUKT}(A_{[i,:]}, B[:,j])$ 
4  return ( $c_{ij}$ )
```

Omkostningen er så $mr \times 2n = 2mnr$ flops.

Bemærkning 5.1. Antag at $A \in \mathbb{R}^{m \times n}$, $B \in \mathbb{R}^{n \times r}$ og $C \in \mathbb{R}^{r \times p}$. Vi har set at matematisk er $(AB)C = A(BC)$. Men hvis vi beregner omkostningerne ved beregning af de to udtryk ser vi en forskel. Beregningen af $(AB)C$ koster

$$2mnr + 2mrp = 2(n+p)mr \text{ flops,}$$

hvorimod beregningen af $A(BC)$ koster

$$2nrp + 2mnp = 2(m+r)np \text{ flops.}$$

Da disse to udtryk er ikke ens; der kan være stor forskel i antallet af flops. \diamond

Eksempel 5.2. For $A \in \mathbb{R}^{2000 \times 2}$, $B \in \mathbb{R}^{2 \times 1000}$, $C \in \mathbb{R}^{1000 \times 100}$ har vi

$$(AB)C : 2 \times (2 + 100) \times 2000 \times 1000 = 4,08 \cdot 10^8 \text{ flops,}$$

$$A(BC) : 2 \times (2000 + 1000) \times 2 \times 100 = 1,2 \cdot 10^6 \text{ flops,}$$

så der er en faktor 400 til forskel. \triangle

Tabel 5.1 giver et overblik over omkostninger ved vektor- og matrixregneoperationer. Det skal noteres at disse vurderinger kan reduceres i nogle situationer, især hvis matricerne eller vektorerne har en bestemt struktur, f.eks. med mange indgange lige med 0.

operation				flops
vektorsum	$u + v$	$u, v \in \mathbb{R}^n$		n
skalar-vektorprodukt	sv	$s \in \mathbb{R}, v \in \mathbb{R}^n$		n
række-søjleprodukt	$u^T v$	$u^T \in \mathbb{R}^{1 \times n}, v \in \mathbb{R}^{n \times 1}$		$2n$
ydre produkt	vw^T	$v \in \mathbb{R}^{m \times 1}, w^T \in \mathbb{R}^{1 \times n}$		mn
matrixsum	$A + B$	$A, B \in \mathbb{R}^{m \times n}$		mn
skalar-matrixprodukt	sA	$s \in \mathbb{R}, A \in \mathbb{R}^{m \times n}$		mn
matrix-vektorprodukt	Au	$A \in \mathbb{R}^{m \times n}, u \in \mathbb{R}^{n \times 1}$		$2mn$
matrixprodukt	AB	$A \in \mathbb{R}^{m \times n}, B \in \mathbb{R}^{n \times r}$		$2mnr$

Tabel 5.1: Omkostninger ved vektor- og matrixberegninger.

Desuden for meget store vektorer og matricer er der også andre faktorer der skal tages i betragtning. En computer har typisk et begrænset mængde hukommelse som er hurtigt tilgængeligt (cache og RAM), så store store matricer skal nødvendigvis gemmes på en disk eller SSD. Men indhentning af data fra en disk eller SSD er væsentlige langsommere (f.eks. en ved faktor 1000) end fra cache eller RAM, så dette kan også påvirke hvor hurtigt en beregning kan udføres.

Overraskende nok findes der hurtige måde at gange matricer sammen end MATRIXPRODUKT ovenfor. For $A, B \in \mathbb{R}^{n \times n}$ koster MATRIXPRODUKT(A, B) $2n^3$ flops. I 1969 fandt Volker Strassen en beregningsmetode der bruger højst $c \times n^{2.807}$ flops, hvor c er en konstant uafhængigt af n . Strassens metode kan bruges effektivt på større matricer. Der findes andre nyere metoder hvis omkostning afhænger af en endnu mindre potens af n , men de har mest teoretisk interesse.

5.2 Pythons for-løkke

I de overstående eksempler har vi skrevet algoritmerne ikke i python med i generisk kode. Syntaksen er dog alligevel tæt på en konstruktion i python, som har den generelle form

```

1 for variabel in objekt:
2     udfør forskellige trin
3     der muligvis bruger variabel

```

5 NUMERISK HÅNDTERING AF MATRICER

Her har vi vist mellemrum eksplicit, da der er væsentligt at linjerne 2, 3, ..., der skal udføres i løkken, er rykket 4 mellemrum til højre i forhold til selve **for ... in ...** : linjen 1.

Objektet der typisk bruges i **for**-løkker er **range(...)**. For eksempel, skriver man **range(stop)** med stop et positivt heltal, fås rækken af heltal fra 0 til stop-1:

```
>>> for i in range(5):
...     print(i)
...
0
1
2
3
4
```

Man kan også skrive **range(start, stop)** for at angive en anden begyndelsesværdi end 0:

```
>>> for i in range(5, 10):
...     print(i)
...
5
6
7
8
9
```

Desuden kan man give en anden trinstørrelse via et tredje argument, som **range(start, stop, step)**:

```
>>> for i in range(0, 10, 2):
...     print(i)
...
0
2
4
```

```
6
8
```

F.eks. kan række-søjleproduktet ovenfor udregnes med følgende python kode

```
import numpy as np
rng = np.random.default_rng()
n = 20
ut = rng.random((1,n))
v = rng.random((n,1))
c = 0
for i in range(n):
    c = c + ut[0, i]*v[i, 0]
print(c)
```

som giver 6.249737049599422. Men NumPys egen @-operation er specielt skrevet til at gøre den type beregning mere effektivt med mere kompakt syntaks og kan tage højde for korrekt brug af float-operationer

```
c = (ut @ v)[0,0]
print(c)
```

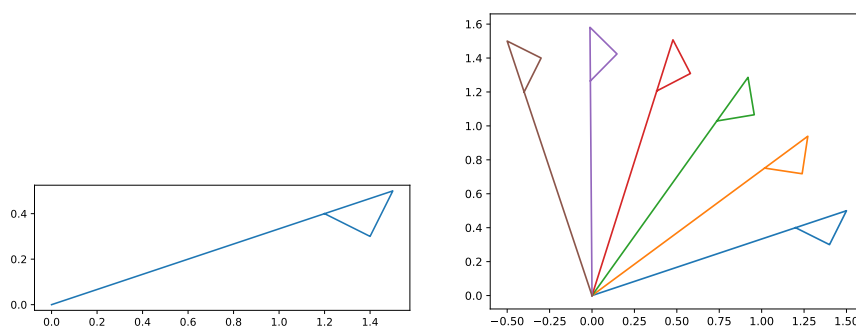
som giver 6.249737049599422. Generelt skal **for**-løkker ikke bruges til at erstatte operationer tilgængelig fra NumPy, men det kan være nyttigt når man afprøver nye beregningsmetode og i forbindelse med plot af figurer.

F.eks. hvis man vil illustrere effekten af gentagende brug af en rotationsmatrix på en simple figur. Vi begynder med en figur i planen angivet ved søjlerne af en (2×4) -matrix points

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

points = np.array([[0.0, 1.5, 1.4, 1.2],
                  [0.0, 0.5, 0.3, 0.4]])
fig, ax = plt.subplots()
ax.set_aspect('equal')
ax.plot(*points)
```

5 NUMERISK HÅNDTERING AF MATRICER



Figur 5.1: Oprindelig figur og dens rotationer.

som giver det første billede i figur 5.1. Vi har brugt lidt andet syntaks en tidligere, hvor en plot var typisk `ax.plot(x, y)`. I vores tilfælde er `x` og `y` rækkerne `points[0]` og `points[1]`, og `ax.plot(*points)` er en kort form for `ax.plot(points[0], points[1])`. Dvs. `*points` pakker `points` ud til `points[0]`, `points[1]`.

Nu stiller vi en rotationsmatrix op og bruger den til at drejer figuren 5 gange i en **for**-løkke:

```
c = np.cos(np.pi/10)
s = np.sin(np.pi/10)
R = np.array([[c, -s],
              [s,  c]])

fig, ax = plt.subplots()
ax.set_aspect('equal')
ax.plot(*points)
for i in range(5):
    points = R @ points
    ax.plot(*points)
```

Resultatet er det andet billede i figur 5.1.

Kapitel 6

Lineære ligningssystemer

Vi har set tidligere hvordan en simpel elektrisk kreds giver anledning til et lineært ligningssystem. I dette kapitel kikker vi på nogle matematiske metoder for at forstå og løse sådanne systemer eksakt. Det skal bemærkes at udmiddelbart er disse metoder ikke så hensigt mæssig for numerisk beregning med `floats`, men der et par udsagn, som har væsentlig betydning for metoderne der bruges senere og som skal være på plads inden vi kan komme til det numeriske.

6.1 Elementære rækkeoperationer

Betragt det lineære ligningssystem

$$2x + 3y - 4z = 7, \quad (6.1a)$$

$$3x - 4y + z = -2, \quad (6.1b)$$

$$x + y + 2z = 3. \quad (6.1c)$$

Det er et system af 3 lineære ligninger i ubekendte x, y, z . Vi er interesseret i at finde alle løsninger (x, y, z) til systemet.

Vi kunne løse systemet ved f.eks. at isolere variabelen z i (6.1c) og derefter indsæt for z i (6.1a) og (6.1b). Vi får $z = 3/2 - x/2 - y/2$ og (6.1a)–(6.1b) bliver

$$2x + 3y - 4\left(\frac{3}{2} - \frac{1}{2}x - \frac{1}{2}y\right) = 7,$$

$$3x - 4y + \left(\frac{3}{2} - \frac{1}{2}x - \frac{1}{2}y\right) = -2,$$

6 LINEÆRE LIGNINGSSYSTEMER

som forkortes til

$$4x + 5y = 13, \quad (6.2a)$$

$$\frac{5}{2}x - \frac{7}{2}y = -\frac{7}{2}. \quad (6.2b)$$

Processen kan så gentages ved at isolere y i (6.2b), og indsætte udtrykket i (6.2a). Der fås så én ligning med kun variabelen x , som kan løses for x , og derefter kan man finde først y og så z ved at indsætte den x -værdi.

En mere systematisk fremgangsmåde baserer sig på den følgende observation. Der er tre simple operationer vi kan udføre på system uden at ændre løsningsmængde, nemlig

- (I) byt om på rækkefølgen af ligninger,
- (II) gang en ligning igennem med en skalar som er ikke 0, eller
- (III) tilføj et multiplum af én ligning til en anden ligning.

Lad os skifte til matrixnotation for at lette arbejdet. Systemet (6.1a)–(6.1c) er ækvivalent med matrixligningen

$$\begin{bmatrix} 2 & 3 & -4 \\ 3 & -4 & 1 \\ 1 & 1 & 2 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \end{bmatrix} = \begin{bmatrix} 7 \\ -2 \\ 3 \end{bmatrix}.$$

Vi kan samle koefficienterne i en *udvidet matrix*

$$\left[\begin{array}{ccc|c} 2 & 3 & -4 & 7 \\ 3 & -4 & 1 & -2 \\ 1 & 1 & 2 & 3 \end{array} \right] \quad (6.3)$$

De tre elementære ligningsoperationer ovenfor er nu det samme som de følgende tre *elementære rækkeoperationer*

- (I) $R_i \leftrightarrow R_j$ byt række i med række j ,
- (II) $R_i \rightarrow sR_i$ $s \neq 0$ skalær R_i med en faktor s ,
- (III) $R_i \rightarrow R_i + tR_j$ $j \neq i$ læg t gange række j til række i .

Disse operationer kan bruges på (6.3), f.eks. på den følgende måde. Vores mål

er at have 0-tal under diagonalen.

$$\begin{aligned}
 & \left[\begin{array}{ccc|c} 2 & 3 & -4 & 7 \\ 3 & -4 & 1 & -2 \\ 1 & 1 & 2 & 3 \end{array} \right] \\
 & \sim_{R_0 \leftrightarrow R_2} \left[\begin{array}{ccc|c} 1 & 1 & 2 & 3 \\ 3 & -4 & 1 & -2 \\ 2 & 3 & -4 & 7 \end{array} \right] \sim_{R_1 \rightarrow R_1 - 3R_0} \left[\begin{array}{ccc|c} 1 & 1 & 2 & 3 \\ 0 & -7 & -5 & -11 \\ 2 & 3 & -4 & 7 \end{array} \right] \\
 & \sim_{R_2 \rightarrow R_2 - 2R_0} \left[\begin{array}{ccc|c} 1 & 1 & 2 & 3 \\ 0 & -7 & -5 & -11 \\ 0 & 1 & -8 & 1 \end{array} \right] \sim_{R_1 \leftrightarrow R_2} \left[\begin{array}{ccc|c} 1 & 1 & 2 & 3 \\ 0 & 1 & -8 & 1 \\ 0 & -7 & -5 & -11 \end{array} \right] \\
 & \sim_{R_2 \rightarrow R_2 + 7R_1} \left[\begin{array}{ccc|c} 1 & 1 & 2 & 3 \\ 0 & 1 & -8 & 1 \\ 0 & 0 & -61 & -4 \end{array} \right]
 \end{aligned} \tag{6.4}$$

Dette reduceret system kan du løses via *back substitution*: den sidste ligning siger $-61z = -4$, så $z = 4/61$, den anden ligning giver $y = 1 + 8z = (61 + 32)/61 = 93/61$ og nu den første fører til $x = 3 - y - 2z = (183 - 93 - 8)/61 = 82/61$. Dvs.

$$x = \frac{82}{61}, \quad y = \frac{93}{61}, \quad z = \frac{4}{61}.$$

Som alternativ kan man forsætter med (6.4), og bruge yderligere rækkeoperationer til at få 1-tal på diagonalen og 0-tal ellers:

$$\begin{aligned}
 & \left[\begin{array}{ccc|c} 1 & 1 & 2 & 3 \\ 0 & 1 & -8 & 1 \\ 0 & 0 & -61 & -4 \end{array} \right] \\
 & \sim_{R_2 \rightarrow -R_2/61} \left[\begin{array}{ccc|c} 1 & 1 & 2 & 3 \\ 0 & 1 & -8 & 1 \\ 0 & 0 & 1 & 4/61 \end{array} \right] \sim_{R_0 \rightarrow R_0 - R_1} \left[\begin{array}{ccc|c} 1 & 0 & 10 & 2 \\ 0 & 1 & -8 & 1 \\ 0 & 0 & 1 & 4/61 \end{array} \right] \\
 & \sim_{R_0 \rightarrow R_0 - 10R_2} \left[\begin{array}{ccc|c} 1 & 0 & 0 & 82/61 \\ 0 & 1 & -8 & 1 \\ 0 & 0 & 1 & 4/61 \end{array} \right] \sim_{R_1 \rightarrow R_1 + 8R_2} \left[\begin{array}{ccc|c} 1 & 0 & 0 & 82/61 \\ 0 & 1 & 0 & 93/61 \\ 0 & 0 & 1 & 4/61 \end{array} \right]
 \end{aligned} \tag{6.5}$$

6.2 Rækkeoperationer i python

Lad os se hvordan sådanne rækkeoperationer kan udføres i python. Husk at $a[i, :]$ er den i te række i a . Rækkeoperation (II) $R_i \rightarrow sR_i$ kan så udføres

operation	python
(I) $R_i \leftrightarrow R_j$	<code>a[[i,j], :] = a[[j,i], :]</code>
(II) $R_i \rightarrow sR_i (s \neq 0)$	<code>a[i, :] *= s</code>
(III) $R_i \rightarrow R_i + tR_j (j \neq i)$	<code>a[i, :] += t * a[j, :]</code>

Tabel 6.1: Rækkeoperationer i python.

via `a[i, :] = s * a[i, :]`. Men der er en nyttig forkortelse `*=`, som også hjælper med at undgå slåfejl: `x *= s` er det samme som `x = x * s`. Vi kan derfor lave rækkeoperation (II), som

```
a[i, :] *= s
```

Tilsvarende for operation (III) kan vi bruge `x += y` til som forkortelse af `x = x + y`, så operation (III) $R_i \rightarrow R_i + tR_j$ udføres via

```
a[i, :] += t * a[j, :]
```

For rækkeoperation (I) er der en nyttig udvidet indekserings notation: `a[[i,j], :]` er nemlig rækker *i* og *j* fra a ub i den givne rækkefølge. Så operation (I) $R_i \leftrightarrow R_j$ kan laves via

```
a[ [i,j], :] = a[ [j,i], :]
```

Disse rækkeoperationer er opsummeret i tabel 6.1.

Vi kan bruge disse operationer til at lave reduktionen (6.3) i python. Først lad os se hvordan man danner den udvidede matrix ved hjælp af `np.hstack`. Kommandoen `np.hstack([a, b])` giver en ny array med søjler fra *a* efterfulgt af søjler fra *b*.

```
>>> import numpy as np
>>> a = np.array([[2.0, 3.0, -4.0],
...              [3.0, -4.0, 1.0],
...              [1.0, 1.0, 2.0]])
>>> a
array([[ 2.,  3., -4.],
       [ 3., -4.,  1.]])
```

6.2 RÆKKEOPERATIONER I PYTHON

```
    [ 1.,  1.,  2.]])
>>> b = np.array([7.0, -2.0, 3.0])[:, np.newaxis]
>>> b
array([[ 7.],
       [-2.],
       [ 3.]])
>>> aub = np.hstack([a, b])
>>> aub
array([[ 2.,  3., -4.,  7.],
       [ 3., -4.,  1., -2.],
       [ 1.,  1.,  2.,  3.]])
```

Derefter bruger vi rækkeoperationerne fra tabel 6.1:

```
>>> aub
array([[ 2.,  3., -4.,  7.],
       [ 3., -4.,  1., -2.],
       [ 1.,  1.,  2.,  3.]])
>>> aub[ [0,2], :] = aub[ [2,0], :]
>>> aub
array([[ 1.,  1.,  2.,  3.],
       [ 3., -4.,  1., -2.],
       [ 2.,  3., -4.,  7.]])
>>> aub[1, :] += -3*aub[0, :]
>>> aub
array([[ 1.,  1.,  2.,  3.],
       [ 0., -7., -5., -11.],
       [ 2.,  3., -4.,  7.]])
>>> aub[2, :] += -2*aub[0, :]
>>> aub
array([[ 1.,  1.,  2.,  3.],
       [ 0., -7., -5., -11.],
       [ 0.,  1., -8.,  1.]])
>>> aub[ [1,2], :] = aub[ [2,1], :]
>>> aub
array([[ 1.,  1.,  2.,  3.],
       [ 0.,  1., -8.,  1.],
       [ 0., -7., -5., -11.]])
```

```
>>> aub[2, :] += 7*aub[1, :]
>>> aub
array([[ 1.,  1.,  2.,  3.],
       [ 0.,  1., -8.,  1.],
       [ 0.,  0., -61., -4.]])
```

Man kan forsætte i samme stil hvis man vil udføre operationerne i (6.5).

6.3 Echelonform

Generelt kan man ikke reducere en koefficientmatrix til en form med 1-tal på diagonalen, men man kan komme tæt på. Følgende matrix er i det der kaldes »echelonform«:

$$\begin{bmatrix} 1 & 0 & 2,3 & -1,2 & 0,5 \\ 0 & 0 & 1 & 3,2 & -2,2 \\ 0 & 0 & 0 & 1 & 0,3 \\ 0 & 0 & 0 & 0 & 0 \end{bmatrix} \quad (6.6)$$

Definition 6.1. En matrix er i *echelonform* hvis alle de følgende tre krav er opfyldte:

- (a) Den første ikke nul indgang i hver række er lige med 1. Indgangen kaldes et *pivotelement*.
- (b) Pivot elementer ligger længere til højre i efterfølgende rækker.
- (c) Nulrækker kommer til sidst.

I vores matrix er pivotelementerne fremhævet nedenfor

$$\begin{bmatrix} \boxed{1} & 0 & 2,3 & -1,2 & 0,5 \\ 0 & 0 & \boxed{1} & 3,2 & -2,2 \\ 0 & 0 & 0 & \boxed{1} & 0,3 \\ 0 & 0 & 0 & 0 & 0 \end{bmatrix}$$

og vi ser at alle indgange under trappestien dannet af pivotelementer er 0.

Proposition 6.2. *Enhver matrix kan reduceres til echelonform via rækkeoperationer.*

6.4 LØSNING VIA RÆKKEOPERATIONER

Bevis. Givet en matrix $A \in \mathbb{R}^{m \times n}$, find den første søjle som er ikke $0_{m \times 1}$ og vælg et element i denne søjle som er forskellig fra 0. Brug rækkeoperation (I) til at flytte dette element til den første række. Så har vi en matrix af formen

$$\begin{bmatrix} 0 & \dots & 0 & \times & * & \dots & * \\ 0 & \dots & 0 & * & * & \dots & * \\ \vdots & & \vdots & \vdots & \vdots & & \vdots \\ 0 & \dots & 0 & * & * & \dots & * \end{bmatrix},$$

hvor \times betegner et element forskelligt fra 0, og $*$ angiver vilkårlige tal.

Brug rækkeoperation (I) til at få et 1-tal i position \times . Hvis $m = 1$ har vi nu en matrix i echelonform og vi er færdig.

Hvis $m > 1$, kan vi antage at vi har vist at alle $((m-1) \times n)$ kan reduceres til echelonform.

Brug nu rækkeoperation (III) til at få en ny matrix med alle elementer i søjlen under 1-tallet lige med 0:

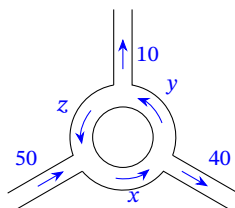
$$\begin{bmatrix} 0 & \dots & 0 & 1 & * & \dots & * \\ \dots & \dots & \dots & \dots & \dots & \dots & \dots \\ 0 & \dots & 0 & 0 & * & \dots & * \\ \vdots & & \vdots & \vdots & \vdots & & \vdots \\ 0 & \dots & 0 & 0 & * & \dots & * \end{bmatrix} = \begin{bmatrix} u^T \\ \cdot \\ B \\ \cdot \end{bmatrix}. \quad (6.7)$$

Matricen B under den punkterede linje har $B \in \mathbb{R}^{(m-1) \times n}$, så B kan rækkereduceres til echelonform. Bemærk at pivotelementer fra B ligger til højre fra pivotelementet 1 i rækken u^T i (6.7). Så ved at reducere B til echelonform får vi en echelonform for A . \square

6.4 Løsning af lineære ligningssystemer via rækkeoperationer

For matricen (6.6) i echelonform, er det tilsvarende system af lineære ligninger af formen

$$\begin{aligned} x_0 + 2,3x_2 - 1,2x_3 + 0,5x_4 &= 1,2, \\ x_2 + 3,2x_3 - 2,2x_4 &= 0,7, \\ x_3 + 0,3x_4 &= -0,2, \\ 0 &= 0, \end{aligned}$$



Figur 6.1: Rundkørsel.

hvor vi har indsat nogle tilfældige værdier på den højre side. I dette system ser vi at hvis vi giver x_4 , og det usynlige x_1 , vilkårlige værdier, så kan vi bestemme x_3 , x_2 og til sidst x_1 via back substitution. Vi siger at variablerne x_1 , x_4 er *frie*, og at x_0 , x_2 , x_3 er *bundne*. Systemet har så uendelig mange løsninger $(x_0, x_1, x_2, x_3, x_4)$, da x_1 , x_4 kan tage hvilket som helst reelle værdier. Bemærk at de bundne variabler svar netop til pivotsøjlerne, så

- (a) antallet af bundne variabler er antallet af pivotelementer, og
- (b) antallet af frie variabler er antallet af søjler i koefficientmatricen minus antallet af pivotelementer.

Eksempel 6.3. Betragt rundkørslen i figur 6.1. På de ensrettede tilsluttende veje er antallet af biler per time målt som angivet. Antallet x , y , z af biler per time på de forskellige dele af selve rundkørslen opfylder

$$z + 50 = x, \quad y + 40 = x \quad \text{og} \quad z + 10 = y.$$

Dette omskrives til systemet

$$\begin{aligned} x - z &= 50, \\ x - y &= 40, \\ y - z &= 10. \end{aligned}$$

Den tilsvarende udvidede matrix er

$$\begin{aligned} \left[\begin{array}{ccc|c} 1 & 0 & -1 & 50 \\ 1 & -1 & 0 & 40 \\ 0 & 1 & -1 & 10 \end{array} \right] &\sim_{R_1 \rightarrow R_1 - R_0} \left[\begin{array}{ccc|c} 1 & 0 & -1 & 50 \\ 0 & -1 & 1 & -10 \\ 0 & 1 & -1 & 10 \end{array} \right] \\ &\sim_{R_2 \rightarrow R_2 + R_1} \left[\begin{array}{ccc|c} 1 & 0 & -1 & 50 \\ 0 & -1 & 1 & -10 \\ 0 & 0 & 0 & 0 \end{array} \right] &\sim_{R_1 \rightarrow -R_1} \left[\begin{array}{ccc|c} 1 & 0 & -1 & 50 \\ 0 & 1 & -1 & 10 \\ 0 & 0 & 0 & 0 \end{array} \right] \end{aligned}$$

6.4 LØSNING VIA RÆKKEOPERATIONER

Systemet er konsistent. Pivotelementer er søjler 0 og 1, så z er fri, og x, y er bundne. Den generelle løsning til systemet er

$$x = 50 + z, \quad y = 10 + z, \quad z \in \mathbb{R}.$$

Realistiske løsninger skal dog være positive så vi må begrænse os til $z \geq 0$. \triangle

Generelt givet et lineært ligningssystem

$$\begin{aligned} a_{00}x_0 + a_{01}x_1 + \cdots + a_{0,n-1}x_{n-1} &= b_0, \\ a_{10}x_0 + a_{11}x_1 + \cdots + a_{1,n-1}x_{n-1} &= b_1, \\ &\vdots \\ a_{m-1,0}x_0 + a_{m-1,1}x_1 + \cdots + a_{m-1,n-1}x_{n-1} &= b_{m-1}, \end{aligned}$$

skriver vi det i matrixform

$$Ax = b$$

med $A \in \mathbb{R}^{m \times n}$, $b \in \mathbb{R}^m$ givne, og $x \in \mathbb{R}^n$ ukendt. Vi danner så den udvidede matrix

$$[A \mid b]$$

og reducerer til echelonform

$$[C \mid d]$$

Vores oprindelig lineære ligningssystem har de samme løsninger som systemet

$$Cx = d. \tag{6.8}$$

For et generelt system er der tre muligheder

(a) $[C \mid d]$ har en række af formen

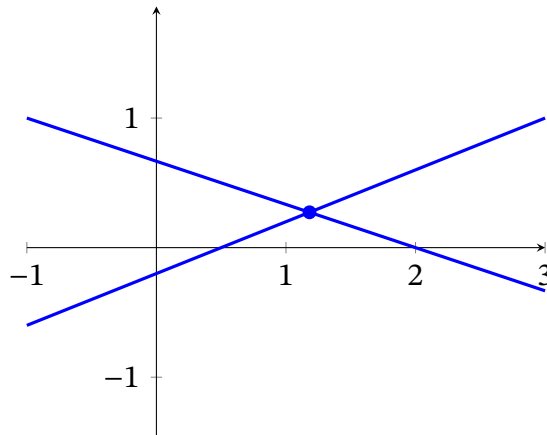
$$[0 \quad 0 \quad \dots \quad 0 \mid 1]$$

Dette betyder at den tilsvarende ligning siger $0 = 1$, som er en modstrid. Så har systemet $Ax = b$ ingen løsninger. Systemet siges at være *inkonsistent*.

(b) Systemet er konsistent og et af de følgende to tilfælde gælder:

- (i) Antallet af pivotelementer er lige med antallet af søjler. Så er der ingen frie variabler, alle variabler er bundne og systemet har en entydig løsning.
- (ii) Der er strengt færre pivotelementer end søjler. Så giver de andre søjler frie variabler og der er uendelige mange løsninger til systemet. Systemet er *underbestemt*.

6 LINEÆRE LIGNINGSSYSTEMER



Figur 6.2: To linjer i planen, som skærer.

Vi har derfor vist at:

Sætning 6.4. *Et lineært ligningssystem enten*

(a) *er en inkonsistent, eller*

(b) (i) *har præcis én løsning, eller*

(ii) *har uendelig mange løsninger bestemt af frie variabler.*

□

Bemærkning 6.5. For et konsistent system af m ligninger i n variabler, med $m < n$, er der altid frie variabler og så uendelig mange løsninger. ◇

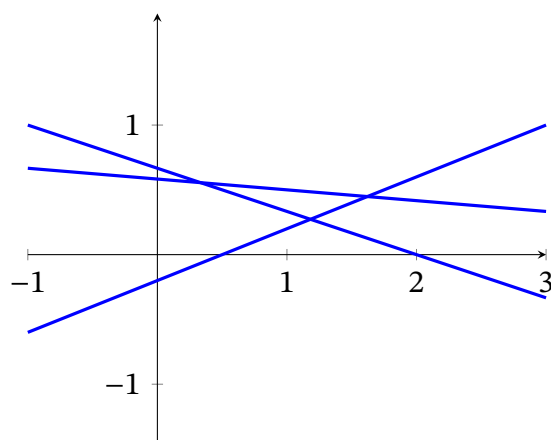
Eksempel 6.6. For systemer i to variabler kan vi tegne os frem til de forskellige udfald. En lineær ligning $ax + by = c$ beskriver en linje i planen, som vi så i eksempel 4.4. Løsningen på et system af sådanne ligninger er koordinaterne på punkter der ligger på alle de beskrevne linjer i planen, dvs. koordinater på fælles skæringspunkter.

Systemet

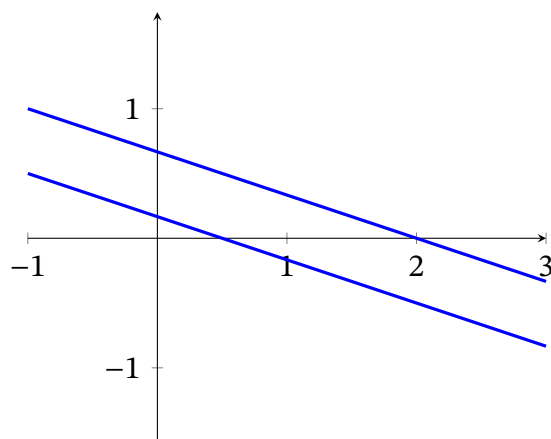
$$\begin{aligned}x + 3y &= 2, \\ 2x - 5y &= 1\end{aligned}$$

vises i figur 6.2, og det ses at der er netop ét skæringspunkt, dvs. en entydig løsning på systemet.

6.4 LØSNING VIA RÆKKEOPERATIONER



Figur 6.3: Tre linjer i planen, som har intet fællespunkt.



Figur 6.4: To parallelle linjer i planen har intet fællespunkt.

Derimod for systemet

$$\begin{aligned}x+3y &= 2, \\2x-5y &= 1, \\x+6y &= 5,\end{aligned}$$

er tegningen, som i figur 6.3, ses at tre linjer har ingen fællespunkt, så systemet har ingen løsning og er inkonsistent.

6 LINEÆRE LIGNINGSSYSTEMER

En anden situation uden fællesløsninger

$$\begin{aligned}x+3y &= 2, \\2x+6y &= 1.\end{aligned}$$

Som det ses i figur 6.4, er det to linjer parallelle, så har ingen skæringspunkt. Her er systemet inkonsistent. Men hvis vi justerer konstanterne på højre siden til

$$\begin{aligned}x+3y &= 2, \\2x+6y &= 4,\end{aligned}$$

så beskriver begge ligninger den samme linje i planen, og alle punkter på linjen er løsninger på systemet. Dvs. dette er et systemet med uendelige mange løsninger. △

Kapitel 7

Matrixinvers

For reelle tal er det meget nyttigt at man kan dele med et tal s , som er ikke nul. Dette er det sammen som at gange med den reciprokke $1/s$. For matricer er der en tilsvarende matrix, den inverse, som er især nyttig i eksakte regnestykker. Men modsat reelle tal er der mange matricer der ikke tillader en invers. For det første skal matricen være kvadratisk (definition 7.1 nedenfor), og for kvadratiske matricer er der ydeligere krav, der skal opfyldes.

7.1 Egenskaber af den inverse matrix

Definition 7.1. En matrix A er *kvadratisk* hvis den har det samme antal rækker som søjler. Dvs. en kvadratisk matrix er et $A \in \mathbb{R}^{n \times n}$.

Bemærk at identitetsmatricen I_n er kvadratisk og for alle $A \in \mathbb{R}^{n \times n}$ har vi

$$I_n A = A = A I_n.$$

Sætning 7.2. Lad $A \in \mathbb{R}^{n \times n}$ være en kvadratisk matrix. Hvis der findes en matrix $A^{-1} \in \mathbb{R}^{n \times n}$ således at

$$A^{-1} A = I_n \tag{7.1}$$

så (a) er A invertibel, (b) er A^{-1} den eneste matrix der opfylder (7.1), og (c)

$$A A^{-1} = I_n. \tag{7.2}$$

Omvendt, hvis en matrix A^{-1} opfylder (7.2), så er den entydigt bestemt og opfylder (7.1).

7 MATRIXINVERS

Beviset for dette resultat gives senere i kapitlet, se afsnit 7.2. Her giver vi nogle eksempler og fokuserer på nogle af egenskaberne for A^{-1} .

For $A \in \mathbb{R}^{2 \times 2}$ er der en simpel formel for A^{-1} . Matricen

$$A = \begin{bmatrix} a & b \\ c & d \end{bmatrix}$$

har invers

$$A^{-1} = \frac{1}{ad - bc} \begin{bmatrix} d & -b \\ -c & a \end{bmatrix} \quad \text{når } ad - bc \neq 0.$$

Dette kan bekræftes direkte ved beregning af matrixproduktet (7.1):

$$\begin{aligned} A^{-1}A &= \frac{1}{ad - bc} \begin{bmatrix} d & -b \\ -c & a \end{bmatrix} \begin{bmatrix} a & b \\ c & d \end{bmatrix} = \frac{1}{ad - bc} \begin{bmatrix} da - bc & db - bd \\ -ca + ac & -cb + ad \end{bmatrix} \\ &= \frac{1}{ad - bc} \begin{bmatrix} ad - bc & 0 \\ 0 & ad - bc \end{bmatrix} = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} = I_2. \end{aligned}$$

Ligningen (7.2) kan bekræftes ved en tilsvarende udregning. Bemærk betingelsen $ad - bc \neq 0$ for eksistens af den inverse i disse tilfælde. Tallet $ad - bc$ kaldes determinanten af $A \in \mathbb{R}^{2 \times 2}$.

Givet et lineært ligningssystem

$$Ax = b$$

med $A \in \mathbb{R}^{n \times n}$ kvadratisk og invertibel, kan man bruge inversen til at løse systemet. Vi ganger begge sider igennem med A^{-1} , til at få

$$A^{-1}Ax = A^{-1}b. \quad (7.3)$$

Men $A^{-1}A = I_n$, så $A^{-1}Ax = I_n x = x$, og ligning (7.3) er blot

$$x = A^{-1}b.$$

Dvs. vi har fundet den entydige løsning til systemet. Dette er ofte af teoretisk interesse; i en numerisk sammenhæng er det generelt bedst at undgå den inverse matrix og bruge andre metoder for sådanne systemer, som ikke er så følsom over for unøjagtigheder.

Eksempel 7.3. Betragt systemet

$$\begin{aligned} 2x + 3y &= 1, \\ x + 2y &= 2. \end{aligned}$$

7.1 EGENSKABER AF DEN INVERSE MATRIX

I matrixform er dette

$$\begin{bmatrix} 2 & 3 \\ 1 & 2 \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} 1 \\ 2 \end{bmatrix}.$$

Koefficientmatricen har $ad - bc = 2 \times 2 - 3 \times 1 = 1$, som er forskellig fra nul, så matricen er invertibel. Vi får så

$$\begin{aligned} \begin{bmatrix} x \\ y \end{bmatrix} &= \begin{bmatrix} 2 & 3 \\ 1 & 2 \end{bmatrix}^{-1} \begin{bmatrix} 1 \\ 2 \end{bmatrix} = \frac{1}{2 \times 2 - 3 \times 1} \begin{bmatrix} 2 & -3 \\ -1 & 2 \end{bmatrix} \begin{bmatrix} 1 \\ 2 \end{bmatrix} = \frac{1}{1} \begin{bmatrix} 2 \times 1 - 3 \times 2 \\ -1 \times 1 + 2 \times 2 \end{bmatrix} \\ &= \begin{bmatrix} -4 \\ 3 \end{bmatrix}. \end{aligned}$$

Dvs. $x = -4$, $y = 3$. △

Transformationen $b \mapsto A^{-1}b$ er den omvendte transformation af $x \mapsto Ax$. Så f.eks. den inverse til en rotation i \mathbb{R}^2 er den samme rotation i den modsatte retning. Som matricer har rotationsmatricen

$$R = \begin{bmatrix} c & -s \\ s & c \end{bmatrix}, \quad c^2 + s^2 = 1$$

invers

$$R^{-1} = \begin{bmatrix} c & s \\ -s & c \end{bmatrix}.$$

Bemærk at i dette tilfælde fås R^{-1} fra R via simpel ombytning af indgange. Generelt er beregning af den inverse væsentlig mere kompliceret, og resultatet kan være svært at kontrollere. Senere vil vi have særlig interesse for matricer hvor inversen er nemt tilgængelig.

Der er en simpel formel for den invers af et produkt af matricer:

Proposition 7.4. Hvis $A, B \in \mathbb{R}^{n \times n}$ er invertibel, så er AB invertibel med invers

$$(AB)^{-1} = B^{-1}A^{-1}.$$

Bevis. Det er nok at regne

$$(B^{-1}A^{-1})(AB) = B^{-1}(A^{-1}A)B = B^{-1}I_n B = B^{-1}B = I_n,$$

så $B^{-1}A^{-1}$ er den inverse til AB . □

Matrixinvers kan beregnes i python via `np.linalg.inv`. Her kan vi begynde at ane nogle af de numeriske problemer med den inverse.

7 MATRIXINVERS

```
>>> import numpy as np

>>> a = np.array([[1.0,          1.0],
...               [1.0000000001, 1.0]])
>>> a
array([[1., 1.],
       [1., 1.]])
>>> np.linalg.inv(a)
array([[ -9.99999916e+08,  9.99999916e+08],
       [ 9.99999917e+08, -9.99999916e+08]])
```

Vi ser at selvom denne matrix har alle indgange af størrelse 1, har den inverse indgange af størrelsesorden 10^9 . Dette stammer fra at i beregningen af den inverse kommer vi til at gange igennem med

$$\frac{1}{ad - bc} = \frac{1}{1.0 - 1.0000000001} = \frac{1}{-0.0000000001} = -10,0 \cdot 10^8.$$

Et eksempel af en anden type er

```
>>> n = 100
>>> a = np.triu(2*np.eye(n) - np.ones((n,n)))
>>> a
array([[ 1., -1., -1., ..., -1., -1., -1.],
       [ 0.,  1., -1., ..., -1., -1., -1.],
       [ 0.,  0.,  1., ..., -1., -1., -1.],
       ...,
       [ 0.,  0.,  0., ...,  1., -1., -1.],
       [ 0.,  0.,  0., ...,  0.,  1., -1.],
       [ 0.,  0.,  0., ...,  0.,  0.,  1.]])
>>> np.linalg.inv(a)[0,-1]
3.1691265005705735e+29
```

Matricen a har -1 i alle pladser over diagonalen, 1 på diagonalen, og 0 under diagonalen. Så alle indgange har størrelse højst 1 . Men den inverse har en indgang der er størrelsesorden 10^{29} . Faktisk indeholder den inverse matricer elementer med alle størrelsesordner mellem 1 og 10^{29} .

7.2 Eksistens af den inverse

Definition 7.5. En kvadratisk matrix $A \in \mathbb{R}^{n \times n}$ er *invertibel* hvis der findes et $B \in \mathbb{R}^{n \times n}$ således at

$$BA = I_n = AB. \quad (7.4)$$

Lemma 7.6. Matricen B i (7.4) er entydig.

Bevis. Antag at $C \in \mathbb{R}^{n \times n}$ opfylder Hvis vi har (7.4) og

$$CA = I_n = AC.$$

Så har vi

$$B = BI_n = B(AC) = (BA)C = I_n C = C,$$

dvs. $C = B$. Så B er entydig. \square

Når vi har dette resultat, giver det god mening at skrive $A^{-1} = B$ og at kalde A^{-1} den *inverse* til A .

Lemma 7.7. Hvis $A \in \mathbb{R}^{n \times n}$ er invertibel, så har det lineære ligningssystem

$$Ax = 0, \quad x \in \mathbb{R}^n \quad (7.5)$$

kun én løsning, nemlig $x = 0$.

Bevis. Bemærk først at $x = 0$ er en løsning til ligning, da $A0 = 0$.

Nu lad os gange A^{-1} på (7.5) fra den venstre side. Vi får

$$A^{-1}Ax = 0.$$

Men $A^{-1}A = I_n$, så $0 = A^{-1}Ax = I_n x = x$, dvs. $x = 0 \in \mathbb{R}^n$ er den eneste løsning. \square

Bemærkning 7.8. Beviset for lemma 7.7 bruger kun ligningen $A^{-1}A = I_n$. Så for vilkårlig A hvis B er en matrix således at $BA = I_n$, kan vi også konkludere at ligningen $Ax = 0$ har kun $x = 0$ som løsning. \diamond

Hvis ligningssystemet $Ax = 0$ har kun én løsning, så indeholder hver søjle ét pivotelement. For $A \in \mathbb{R}^{n \times n}$, betyder dette at A har echelonform

$$\begin{bmatrix} 1 & * & * & \cdots & * \\ 0 & 1 & * & \cdots & * \\ 0 & 0 & 1 & & * \\ \vdots & \vdots & & \ddots & \\ 0 & 0 & 0 & & 1 \end{bmatrix}. \quad (7.6)$$

7 MATRIXINVERS

Nu kan vi bruge yderlige rækkeoperationer af type (III) på (7.6) til at sætte alle elementer $*$ til 0, og dermed reducerer (7.6) til identitetsmatricen I_n . I andre ord, A kan rækkereduceres til I_n .

En bemærkelsesværdig ting er at rækkeoperationer kan realises via matrix-multiplikation via bestemte matricer.

Definition 7.9. En *elementær matrix* E er resultatet af en elementær rækkeoperation på I_n .

Eksempel 7.10. For $n = 2$, lad os kikke på et eksempel på hver type rækkeoperation.

Type (I): er den eneste mulighed $R_0 \leftrightarrow R_1$. Den giver den elementære matrix

$$E = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix}.$$

Type (II): rækkeoperationen $R_1 \rightarrow sR_1$ har elementær matrix

$$E = \begin{bmatrix} 1 & 0 \\ 0 & s \end{bmatrix}.$$

Type (III): rækkeoperationen $R_1 \rightarrow R_1 + tR_0$ svarer til matricen

$$E = \begin{bmatrix} 1 & 0 \\ t & 1 \end{bmatrix}.$$

△

Proposition 7.11. At udføre en elementære rækkeoperation på en $(m \times n)$ -matrix A er det samme som at tage matrixproduktet EA , hvor $E \in \mathbb{R}^{m \times m}$ er den elementære matrix svarende til den pågældende rækkeoperation.

Bevis. Hvis C har rækker $u_0^T, u_1^T, \dots, u_{n-1}^T$, så har produktet CA rækker $u_0^T A, u_1^T A, \dots, u_{n-1}^T A$. Det følger at udførelse af en elementær rækkeoperation på CA er det samme som at udføre den samme elementær rækkeoperation på C og derefter gange fra højre med A . Ved at tage $C = I_n$, fås resultatet. \square

Bemærk at elementære matricer er invertible: deres invers er blot den elementære matrix svarende til den omvendte rækkeoperation.

Bevis for sætning 7.2. Hvis $A^{-1}A = I_n$, så giver lemma 7.7 at det lineære ligningssystem $Ax = 0$ har kun én løsning, nemlig $x = 0$. Det følger at A kan rækkereduceres til I_n . Men det er det samme som at sige at der er elementære matricer $E_0, E_1, \dots, E_{r-2}, E_{r-1}$ således at

$$E_{r-1}E_{r-2} \dots E_1E_0A = I_n.$$

Ganges denne ligning fra venstre med først E_{r-1}^{-1} , så E_{r-2}^{-1} , osv., fås

$$A = E_0^{-1}E_1^{-1} \dots E_{r-2}^{-1}E_{r-1}^{-1}.$$

Men den højre side er invertibel med invers $E_{r-1}E_{r-2} \dots E_1E_0$. Så A selv er en invertibel matrix, og $AA^{-1} = I_n$.

For det omvendte resultat, bruger vi bare at $AA^{-1} = I_n$ fortæller at matricen $B = A^{-1}$ har en matrix $B^{-1} = A$ således at $B^{-1}B = I_n$. Det overstående argument viser at B er invertibel, med invers $B^{-1} = A$. Det siger at $B^{-1}B = I_n = BB^{-1}$, som er $AA^{-1} = I_n = A^{-1}A$, og vi har at A er invertibel. \square

7.3 Beregning af den inverse

Beviset for sætning 7.2 fører også til en metode for at beregne A^{-1} via elementære rækkeoperationer. Vi så at $A^{-1} = E_{r-1}E_{r-2} \dots E_1E_0$, men dette er resultatet af at udføre på I_n de rækkeoperationer, som reducerer A til I_n . Vi kan så beregne A^{-1} ved at starte med den udvidede matrix

$$[A \mid I_n]$$

og derefter rækkereducerer den til

$$[I_n \mid A^{-1}].$$

Eksempel 7.12. For matricen

$$A = \begin{bmatrix} 1 & 2 & 3 \\ -1 & 1 & 1 \\ 1 & 0 & 2 \end{bmatrix}$$

7 MATRIXINVERS

beregner vi inversen

$$\begin{aligned}
 & \left[\begin{array}{ccc|ccc} 1 & 2 & 3 & 1 & 0 & 0 \\ -1 & 1 & 1 & 0 & 1 & 0 \\ 1 & 0 & 2 & 0 & 0 & 1 \end{array} \right] \\
 & \sim_{R_1 \rightarrow R_1 + R_0} \left[\begin{array}{ccc|ccc} 1 & 2 & 3 & 1 & 0 & 0 \\ 0 & 3 & 4 & 1 & 1 & 0 \\ 1 & 0 & 2 & 0 & 0 & 1 \end{array} \right] \sim_{R_2 \rightarrow R_2 - R_0} \left[\begin{array}{ccc|ccc} 1 & 2 & 3 & 1 & 0 & 0 \\ 0 & 3 & 4 & 1 & 1 & 0 \\ 0 & -2 & -1 & -1 & 0 & 1 \end{array} \right] \\
 & \sim_{R_1 \rightarrow R_1 + R_2} \left[\begin{array}{ccc|ccc} 1 & 2 & 3 & 1 & 0 & 0 \\ 0 & 1 & 3 & 0 & 1 & 1 \\ 0 & -2 & -1 & -1 & 0 & 1 \end{array} \right] \sim_{R_2 \rightarrow R_2 + 2R_1} \left[\begin{array}{ccc|ccc} 1 & 2 & 3 & 1 & 0 & 0 \\ 0 & 1 & 3 & 0 & 1 & 1 \\ 0 & 0 & 5 & -1 & 2 & 3 \end{array} \right] \\
 & \sim_{R_0 \rightarrow R_0 - 2R_1} \left[\begin{array}{ccc|ccc} 1 & 0 & -3 & 1 & -2 & -2 \\ 0 & 1 & 3 & 0 & 1 & 1 \\ 0 & 0 & 5 & -1 & 2 & 3 \end{array} \right] \sim_{R_2 \rightarrow \frac{1}{5}R_2} \left[\begin{array}{ccc|ccc} 1 & 0 & -3 & 1 & -2 & -2 \\ 0 & 1 & 3 & 0 & 1 & 1 \\ 0 & 0 & 1 & -\frac{1}{5} & \frac{2}{5} & \frac{3}{5} \end{array} \right] \\
 & \sim_{R_0 \rightarrow R_0 + 3R_2} \left[\begin{array}{ccc|ccc} 1 & 0 & 0 & \frac{2}{5} & -\frac{4}{5} & \frac{1}{5} \\ 0 & 1 & 3 & 0 & 1 & 1 \\ 0 & 0 & 1 & -\frac{1}{5} & \frac{2}{5} & \frac{3}{5} \end{array} \right] \sim_{R_1 \rightarrow R_1 - 3R_2} \left[\begin{array}{ccc|ccc} 1 & 0 & 0 & \frac{2}{5} & -\frac{4}{5} & \frac{1}{5} \\ 0 & 1 & 0 & \frac{3}{5} & -\frac{1}{5} & -\frac{4}{5} \\ 0 & 0 & 1 & -\frac{1}{5} & \frac{2}{5} & \frac{3}{5} \end{array} \right].
 \end{aligned}$$

Fra dette aflæser vi at

$$A^{-1} = \begin{bmatrix} \frac{2}{5} & -\frac{4}{5} & -\frac{1}{5} \\ \frac{3}{5} & -\frac{1}{5} & -\frac{4}{5} \\ -\frac{1}{5} & \frac{2}{5} & \frac{3}{5} \end{bmatrix}.$$

Δ

Kapitel 8

Ortogonalitet og projektioner

8.1 Standard indre produkt

For vektorer $u, v \in \mathbb{R}^n$, definerer vi deres standard *indre produkt* til at være

$$\begin{aligned}\langle u, v \rangle &= u^T v \\ &= \begin{bmatrix} u_0 & u_1 & \dots & u_{n-1} \end{bmatrix} \begin{bmatrix} v_0 \\ v_1 \\ \vdots \\ v_{n-1} \end{bmatrix} \\ &= u_0 v_0 + u_1 v_1 + \dots + u_{n-1} v_{n-1}.\end{aligned}\tag{8.1}$$

Vi siger at u, v er *ortogonal*, eller at v står *vinkelret* på u , og skriver $u \perp v$, hvis

$$\langle u, v \rangle = 0.$$

Eksempel 8.1. For $u = (1, -2, 3)$ og $v = (0, 5, 7)$ har vi

$$\langle u, v \rangle = 1 \times 0 + (-2) \times 5 + 3 \times 7 = 0 - 10 + 21 = 11,$$

så de står ikke vinkelret på hinanden. Tilgængæld har vi for $w = (2, 1, 0)$ og $w' = (3, 0, -1)$ er

$$\langle u, w \rangle = 1 \times 2 + (-2) \times 1 + 3 \times 0 = 0,$$

$$\langle u, w' \rangle = 1 \times 3 + (-2) \times 0 + 3 \times 1 = 0,$$

så w og w' står vinkelret på u . Bemærk at w' er ikke proportionelt med w , så der kan være forskellige retninger der er vinkelret på en given vektor u . \triangle

8 ORTOGONALITET OG PROJEKTIONER

I python kan disse regnes i formen $u^T v$ via @, men det giver en (1×1) -matrix:

```
>>> import numpy as np
>>> u = np.array([1.0, -2.0, 3.0])[:, np.newaxis]
>>> v = np.array([0.0, 5.0, 7.0])[:, np.newaxis]
>>> u.T @ v
array([[11.]])
```

Så for at få en skalar skal vi tage indgangen $[0, 0]$:

```
>>> (u.T @ v)[0, 0]
11.0
```

Som praktisk alternativ er der funktionen `np.vdot`, som giver direkte en skalar og kræver ingen transponering:

```
>>> np.vdot(u, v)
11.0
```

Lemma 8.2. *Det indre produkt har de følgende regneegenskaber, for $u, v, w \in \mathbb{R}^n$ og $a, b \in \mathbb{R}$:*

- (a) $\langle u, v \rangle = \langle v, u \rangle$,
- (b) $\langle au + bw, v \rangle = a\langle u, v \rangle + b\langle w, v \rangle$,
- (c) $\langle u, av + bw \rangle = a\langle u, v \rangle + b\langle u, w \rangle$,
- (d) $\langle v, v \rangle \geq 0$,
- (e) for $v \neq 0$ er $\langle v, v \rangle > 0$.

Bevis. For del (a), bruger vi slutformlen i (8.1) til at få

$$\begin{aligned}\langle u, v \rangle &= u_0 v_0 + u_1 v_1 + \cdots + u_{n-1} v_{n-1} \\ &= v_0 u_0 + v_1 u_1 + \cdots + v_{n-1} u_{n-1} \\ &= \langle v, u \rangle,\end{aligned}$$

da $u_0 v_0 = v_0 u_0$, osv.

For dele (b) og (c) brug bare at $\langle u, v \rangle = u^T v$ og de tilsvarende relationer for matrixmultiplikation.

For del (d), beregner vi

$$\langle v, v \rangle = v_0^2 + v_1^2 + \cdots + v_{n-1}^2.$$

8.1 STANDARD INDRE PRODUKT

Men $v_k^2 \geq 0$, så $\langle v, v \rangle$ er sum af led større end eller lige med 0, og dermed er $\langle v, v \rangle \geq 0$. For del (e), hvis $v \neq 0$, så er der mindst en indgang v_k med $v_k \neq 0$. Men så har vi $v_k^2 > 0$, og $\langle v, v \rangle \geq v_k^2 > 0$, som påstået. \square

Vi sætter

$$\|v\|_2 = \sqrt{\langle v, v \rangle} = \sqrt{v_0^2 + v_1^2 + \cdots + v_{n-1}^2} \quad (8.2)$$

til at være **2-normen** af $v \in \mathbb{R}^n$, og siger at v er en **enhedsvektor** hvis $\|v\|_2 = 1$.

I python kan $\|v\|_2$ beregnes via `np.linalg.norm(v)`:

```
>>> import numpy as np
>>> v = np.array([1.0, -2.0, 3.0])
>>> np.linalg.norm(v)
3.7416573867739413
>>> np.sqrt(v[0]**2 + v[1]**2 + v[2]**2)
3.7416573867739413
```

Proposition 8.3 (Pythagoras sætning). *For u, v ortogonal er*

$$\|u + v\|_2^2 = \|u\|_2^2 + \|v\|_2^2.$$

Bevis. Vi beregner dette kun ved brug af regnereglerne fra lemma 8.2.

$$\begin{aligned} \|u + v\|_2^2 &= \langle u + v, u + v \rangle = \langle u, u + v \rangle + \langle v, u + v \rangle && \text{lemma 8.2(b),} \\ &= \langle u, u \rangle + \langle u, v \rangle + \langle v, u \rangle + \langle v, v \rangle && \text{lemma 8.2(c),} \\ &= \|u\|_2^2 + 2\langle u, v \rangle + \|v\|_2^2 && \text{lemma 8.2(a).} \end{aligned}$$

Men u, v er ortogonal, så $\langle u, v \rangle = 0$, og vi har resultatet. \square

Lemma 8.4. *Hvis v er en vektor i \mathbb{R}^n med $v \neq 0$, så er*

$$w = \frac{1}{\|v\|_2} v$$

en enhedsvektor.

Bevis. Vi begreger

$$\|w\|_2^2 = \langle w, w \rangle = \frac{1}{\|v\|_2^2} \langle v, v \rangle = 1,$$

da $\langle v, v \rangle = \|v\|_2^2$. \square

8.2 Vinkel mellem vektorer

Definition 8.5. For $u, v \in \mathbb{R}^n$ forskellig fra 0 siger vi at *vinklen* mellem u og v er $0 \leq \theta \leq \pi$ med

$$\cos \theta = \frac{\langle u, v \rangle}{\|u\|_2 \|v\|_2}.$$

For $u \perp v$ har vi $\langle u, v \rangle = 0$, så $\theta = \pi/2 = 90^\circ$, som stemmer overens med vores almindelig vinkelmål.

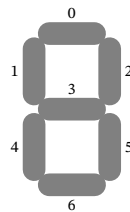
Eksempel 8.6. Vektorerne $u = (1, -2, 3)$ og $v = (0, 5, 7)$ fra eksempel 8.1 er ikke parallelle. De har $\langle u, v \rangle = 11$, $\|u\|_2 = \sqrt{1^2 + (-2)^2 + 3^2} = \sqrt{14}$, $\|v\|_2 = \sqrt{0^2 + 5^2 + 7^2} = \sqrt{74}$. Så

$$\cos \theta = \frac{\langle u, v \rangle}{\|u\|_2 \|v\|_2} = \frac{11}{\sqrt{14}\sqrt{74}} \approx \frac{11}{32,2} = 0,342,$$

som giver $\theta \approx \cos^{-1}(0,342) = 1,22 \text{ rad} = 70,0^\circ$. △

Lad os bemærke at for $u \neq 0 \neq v$ har vi at u, v er parallelle, dvs. $u = sv$ for et $s \in \mathbb{R}$, kun hvis $u/\|u\|_2 = \pm v/\|v\|_2$. Det sidste er det samme som at vinklen θ er enten 0 eller π .

Eksempel 8.7. Et digitalt ur viser cifre ved at oplyse nogle elementer, der er arrangeret som nedenfor:



Standard cifre kan gemmes i en vektor som har et 1-tal for hvert segment der er oplyst, og 0-tal ellers. F.eks.

$$v_0 = \begin{bmatrix} 1 \\ 1 \\ 1 \\ 1 \\ 0 \\ 1 \\ 1 \\ 1 \end{bmatrix} \quad \begin{array}{c} \text{0} \\ \text{1} \\ \text{2} \\ \text{3} \\ \text{4} \\ \text{5} \\ \text{6} \end{array} \quad v_1 = \begin{bmatrix} 0 \\ 0 \\ 1 \\ 0 \\ 0 \\ 0 \\ 1 \\ 0 \end{bmatrix} \quad \begin{array}{c} \text{0} \\ \text{1} \\ \text{2} \\ \text{3} \\ \text{4} \\ \text{5} \\ \text{6} \end{array} \quad v_2 = \begin{bmatrix} 1 \\ 0 \\ 1 \\ 1 \\ 1 \\ 0 \\ 0 \\ 1 \end{bmatrix} \quad \begin{array}{c} \text{0} \\ \text{1} \\ \text{2} \\ \text{3} \\ \text{4} \\ \text{5} \\ \text{6} \end{array} \quad \dots$$

En dag viser uret



(8.3)

Hvilke cifre ligner dette mest?

Vi kan danne en matrix med søjler repræsenterende cifre 0 til 9:

```
>>> import numpy as np
>>> v = np.empty((7,10), dtype=float)
>>> v[:,0] = np.array([1,1,1,0,1,1,1])
>>> v[:,1] = np.array([0,0,1,0,0,1,0])
>>> v[:,2] = np.array([1,0,1,1,1,0,1])
>>> v[:,3] = np.array([1,0,1,1,0,1,1])
>>> v[:,4] = np.array([0,1,1,1,1,0,0])
>>> v[:,5] = np.array([1,1,0,1,0,1,1])
>>> v[:,6] = np.array([0,1,0,1,1,1,1])
>>> v[:,7] = np.array([1,0,1,0,0,1,0])
>>> v[:,8] = np.array([1,1,1,1,1,1,1])
>>> v[:,9] = np.array([1,1,1,1,0,1,0])
>>> v
array([[1., 0., 1., 1., 0., 1., 0., 1., 1., 1.],
       [1., 0., 0., 0., 1., 1., 1., 0., 1., 1.],
       [1., 1., 1., 1., 1., 0., 0., 1., 1., 1.],
       [0., 0., 1., 1., 1., 1., 1., 0., 1., 1.],
       [1., 0., 1., 0., 1., 0., 1., 0., 1., 0.],
       [1., 1., 0., 1., 0., 1., 1., 1., 1., 1.],
       [1., 0., 1., 1., 0., 1., 1., 0., 1., 0.]])
```

(8.3) repræsenteres så af vektoren

```
>>> u = np.array([0,1,0,1,1,1,0])[:, np.newaxis]
>>> u
array([[0],
       [1],
       [0],
       [1],
```

8 ORTOGONALITET OG PROJEKTIONER

```
[1],  
[1],  
[0]])
```

Vi kan beregne indre produkterne mellem søjlerne af v og u ved

```
>>> v.T @ u  
array([[3.],  
       [1.],  
       [2.],  
       [2.],  
       [3.],  
       [3.],  
       [4.],  
       [1.],  
       [4.],  
       [3.]])
```

Dette tæller hvor mange tændte elementer u har til fælles med hvert cifre, og vi ser den har 4 tændte elementer til fælles med 6 og 8.

Hvis vi skalærer søjlerne af v og vektoren u til enhedsvektor, kan vi i stedet beregne cosinus til vinklerne mellem u og hvert søjle via matrix-vektorprodukt

```
>>> vn = np.empty_like(v)  
>>> for i in range(10):  
...     vn[:, i] = v[:, i]/np.linalg.norm(v[:, i])  
...  
>>> un = u/np.linalg.norm(u)  
  
>>> cosines = vn.T @ un  
>>> cosines  
array([[0.61237244],  
       [0.35355339],  
       [0.4472136 ],  
       [0.4472136 ],  
       [0.75       ],  
       [0.67082039],  
       [0.89442719],
```



```
[0.28867513],
[0.75592895],
[0.67082039]])
```

Her bruges `np.empty_like(v)` til at danne en ny matrix med samme antal rækker og søjler som `v`. Fra resultatet ser vi at cosinus til vinklen til 6-tallet er størst, så dette er den bedste kandidat for det korrekte cifre. \triangle

For at definitionen på vinklen giver mening har vi brug for at udtrykket for $\cos \theta$ ligger mellem -1 og 1 . Men dette følger fra:

Sætning 8.8 (Cauchy-Schwarz ulighed). *Givet $u, v \in \mathbb{R}^n$, gælder*

$$|\langle u, v \rangle| \leq \|u\|_2 \|v\|_2, \quad (8.4)$$

med lighed hvis og kun hvis u, v er parallelle.

Bevis. Bemærk først at hvis $u = 0$ eller $v = 0$, så er begge sidder af ulighed 0 og der er intet at vise. Vi derfor kikker på tilfældet hvor $u \neq 0 \neq v$. For $a, b \in \mathbb{R}$ betragter vi $\|au + bv\|_2^2$. Vi har

$$\begin{aligned} 0 &\leq \|au + bv\|_2^2 = \langle au + bv, au + bv \rangle \\ &= a^2 \|u\|_2^2 + 2ab \langle u, v \rangle + b^2 \|v\|_2^2. \end{aligned} \quad (8.5)$$

Det følger at

$$-2ab \langle u, v \rangle \leq a^2 \|u\|_2^2 + b^2 \|v\|_2^2. \quad (8.6)$$

Vælg nu $\varepsilon \in \{\pm 1\}$, $a = -\varepsilon \|v\|_2$ og $b = \|u\|_2$. Så bliver (8.6) til

$$2\varepsilon \|u\|_2 \|v\|_2 \langle u, v \rangle \leq 2 \|u\|_2^2 \|v\|_2^2.$$

Vi deler med $2\|u\|_2 \|v\|_2$, som er strengt positivt, da $u \neq 0 \neq v$, til at få

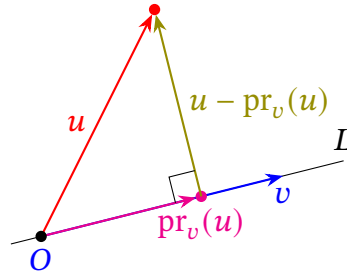
$$\varepsilon \langle u, v \rangle \leq \|u\|_2 \|v\|_2 \quad \text{for både } \varepsilon = +1 \text{ og } \varepsilon = -1.$$

Dette er det samme som (8.4).

Hvornår får vi lighed? Hvis $u = 0$ eller $v = 0$, er der intet at vise: u, v er parallelle og vi har lighed. For $u \neq 0 \neq v$, får vi lighed netop når vi har lighed i (8.5) for et af vores særlig valg af a og b , dvs. netop når

$$0 = \|- \varepsilon \|v\|_2 u + \|u\|_2 v\|_2^2$$

for et $\varepsilon \in \{\pm 1\}$. Men dette sker kun for $-\varepsilon \|v\|_2 u + \|u\|_2 v = 0$, dvs. $u/\|u\|_2 = \varepsilon v/\|v\|_2$, som siger at u, v er parallelle. \square



Figur 8.1: Projektion på en linje.

8.3 Projektion på en linje

Givet en vektor $v \in \mathbb{R}^n$, som er ikke nul, kan vi betragte den rette linje L igennem origo i retningen v :

$$L = \{sv \mid s \in \mathbb{R}\}.$$

Givet en anden vektor $u \in \mathbb{R}^n$, kan vi spørge hvilket punkt på L er tættest på u . Afstanden mellem u og sv er $\|u - sv\|_2$. At minimere $\|u - sv\|_2$ er det sammen som at minimere $\|u - sv\|_2^2$. Vi har

$$\|u - sv\|_2^2 = \langle u - sv, u - sv \rangle = \|u\|_2^2 - 2s\langle u, v \rangle + s^2\|v\|_2^2 =: p(s),$$

som er et andengradspolynomium i s . Toppunktet bestemmes via

$$0 = p'(s) = -2\langle u, v \rangle + 2s\|v\|_2^2,$$

så ligger ved $s = \langle u, v \rangle / \|v\|_2^2$. Dvs. punktet på L , som ligger tættest på u , er

$$\text{pr}_v(u) = \frac{\langle u, v \rangle}{\|v\|_2^2} v.$$

Vi kalder $\text{pr}_v(u)$ *projektionen* af u langs v .

Det skal bemærkes at $u - \text{pr}_v(u)$ er vinkelret på v , sammenlign med figur 8.1:

$$\begin{aligned} \langle u - \text{pr}_v u, v \rangle &= \langle u, v \rangle - \langle \text{pr}_v u, v \rangle = \langle u, v \rangle - \left\langle \frac{\langle u, v \rangle}{\|v\|_2^2} v, v \right\rangle \\ &= \langle u, v \rangle - \frac{\langle u, v \rangle}{\|v\|_2^2} \langle v, v \rangle = \langle u, v \rangle - \langle u, v \rangle \\ &= 0, \end{aligned}$$

8.3 PROJEKTION PÅ EN LINJE

da $\langle v, v \rangle = \|v\|_2^2$. Det følger fra Pythagoras sætning at

$$\|u\|_2^2 = \|\text{pr}_v(u)\|_2^2 + \|u - \text{pr}_v(u)\|_2^2,$$

så afstanden fra u til dens projektion er

$$\|u - \text{pr}_v(u)\|_2 = \sqrt{\|u\|_2^2 - \|\text{pr}_v(u)\|_2^2}.$$

Eksempel 8.9. Betragt $v = (1, 2) \in \mathbb{R}^2$ og $u = (2, 3)$. Vi har $\langle u, v \rangle = 2 \times 1 + 3 \times 2 = 8$, $\|v\|_2^2 = 1^2 + 2^2 = 5$, giver

$$\text{pr}_v(u) = \frac{8}{5} \begin{bmatrix} 1 \\ 2 \end{bmatrix} = \begin{bmatrix} 8/5 \\ 16/5 \end{bmatrix}.$$

Afstanden fra u til L er så

$$\begin{aligned} \sqrt{\|u\|_2^2 - \|\text{pr}_v(u)\|_2^2} &= \sqrt{(2^2 + 3^2) - ((8/5)^2 + (16/5)^2)} \\ &= \sqrt{13 - (64/5)} = \sqrt{1/5}. \end{aligned}$$

△

Observer også at for et punkt sv på L er projektionen punktet selv:

$$\text{pr}_v(sv) = sv,$$

og derved har vi

$$\text{pr}_v(\text{pr}_v(u)) = \text{pr}_v(u). \quad (8.7)$$

Projektion kan skrives som en matrix multiplikation

$$\text{pr}_v(u) = \frac{1}{\|v\|_2^2} v \langle v, u \rangle = \frac{1}{\|v\|_2^2} v v^T u = Pu$$

for matricen

$$P = \frac{1}{\|v\|_2^2} v v^T.$$

Ligning (8.7) siger at

$$P^2 = P.$$

Vi har desuden at

$$P^T = P,$$

dvs. at P er også en *symmetrisk* matrix.

8 ORTOGONALITET OG PROJEKTIONER

Eksempel 8.10. For $v = (1, 2) \in \mathbb{R}^2$, har vi

$$P = \frac{1}{5} \begin{bmatrix} 1 \\ 2 \end{bmatrix} \begin{bmatrix} 1 & 2 \end{bmatrix} = \frac{1}{5} \begin{bmatrix} 1 & 2 \\ 2 & 4 \end{bmatrix}.$$

△

Som eksempel i python har vi

```
>>> import numpy as np
>>> v = np.array([1.0, -2.0, 3.0])[:, np.newaxis]
>>> normsq = np.vdot(v, v)
>>> p = (1/normsq) * v @ v.T
>>> p
array([[ 0.07142857, -0.14285714,  0.21428571],
       [-0.14285714,  0.28571429, -0.42857143],
       [ 0.21428571, -0.42857143,  0.64285714]])
```

som er symmetrisk

```
>>> np.all(p.T == p)
True
```

og opfylder $P^2 = P$ inden for machine epsilon

```
>>> np.allclose(p @ p, p, atol = np.finfo(float).eps)
True
```

8.4 Ortogonalitet

Lad os nu kikke på et større antal vektorer.

Definition 8.11. En samling vektorer v_0, v_1, \dots, v_{k-1} er *ortogonal* hvis

$$\langle v_i, v_j \rangle = 0, \quad \text{for alle } i \neq j.$$

Hvis alle vektorerne v_0, v_1, \dots, v_{k-1} er desuden enhedsvektor, så siger vi at samlingen er *ortonormal*.

Eksempel 8.12. Samlingen

$$\begin{bmatrix} 1 \\ 1 \\ 1 \end{bmatrix}, \quad \begin{bmatrix} 2 \\ -2 \\ 0 \end{bmatrix}, \quad \begin{bmatrix} -1 \\ -1 \\ 2 \end{bmatrix}$$

er ortogonal, men ingen af vektorerne har længde 1, så samlingen er ikke ortonormal. \triangle

Eksempel 8.13. Standardvektorerne e_0, e_1, \dots, e_{n-1} i \mathbb{R}^n er ortonormal. \triangle

Eksempel 8.14. For $c^2 + s^2 = 1$ er vektorerne

$$\begin{bmatrix} c \\ s \end{bmatrix}, \quad \begin{bmatrix} -s \\ c \end{bmatrix}$$

ortonormal. Det samme gælder parret

$$\begin{bmatrix} c \\ s \end{bmatrix}, \quad \begin{bmatrix} s \\ -c \end{bmatrix}.$$

\triangle

Bemærk betingelsen for at v_0, v_1, \dots, v_{k-1} er ortonormal er det samme som

$$\langle v_i, v_j \rangle = \begin{cases} 1, & \text{for } i = j, \\ 0, & \text{for } i \neq j. \end{cases}$$

Ofte er det nemmere at finde vektorer der er ortogonal end ortonormal. Men så forskelligt er to begreber ikke:

Lemma 8.15. Hvis v_0, v_1, \dots, v_{k-1} er ortogonal og alle vektorer er forskellige fra nul, så er

$$\frac{1}{\|v_0\|_2} v_0, \frac{1}{\|v_1\|_2} v_1, \dots, \frac{1}{\|v_{k-1}\|_2} v_{k-1}$$

en ortonormal samling.

Bevis. Vi har $\langle v_i, v_j \rangle = 0$ for $i \neq j$. Desuden er $\langle v_i, v_i \rangle = \|v_i\|_2^2 > 0$, da $v_i \neq 0$. Resultatet følger nu fra lemma 8.4, og

$$\left\langle \frac{1}{\|v_i\|_2} v_i, \frac{1}{\|v_j\|_2} v_j \right\rangle = \frac{1}{\|v_i\|_2 \|v_j\|_2} \langle v_i, v_j \rangle = 0, \quad \text{for } i \neq j.$$

\square

8 ORTOGONALITET OG PROJEKTIONER

Eksempel 8.16. Vektorerne $u = (1, -2, 3)$, $w = (3, 0, -1)$ er ortogonal, som beregnet i eksempel 8.1. Men

$$\langle u, u \rangle = 1^2 + (-2)^2 + 3^2 = 14 \neq 1,$$

så de er ikke ortonormal. Til gengæld får vi en ortonormal samling ved at skalære disse vektorer til enhedsvektorer, dvs.

$$\frac{u}{\|u\|_2} = \frac{1}{\sqrt{14}} \begin{bmatrix} 1 \\ -2 \\ 3 \end{bmatrix}, \quad \frac{w}{\|w\|_2} = \frac{1}{\sqrt{10}} \begin{bmatrix} 3 \\ 0 \\ -1 \end{bmatrix}$$

er ortonormal. △

Lad v_0, v_1, \dots, v_{k-1} være en ortogonal samling af vektorer. En styrke ved sådan en samling er det er nemt at dekomponere andre vektor som kombinationer af v_0, v_1, \dots, v_{k-1} , uden at løse lineære ligningssystemer.

Proposition 8.17. *Hvis*

$$u = x_0 v_0 + x_1 v_1 + \dots + x_{k-1} v_{k-1} \tag{8.8}$$

hvor v_0, v_1, \dots, v_{k-1} er ortogonal, og ingen er 0, så er

$$x_0 = \frac{\langle u, v_0 \rangle}{\|v_0\|_2^2}, \quad x_1 = \frac{\langle u, v_1 \rangle}{\|v_1\|_2^2}, \quad \dots, \quad x_{k-1} = \frac{\langle u, v_{k-1} \rangle}{\|v_{k-1}\|_2^2}. \tag{8.9}$$

Desuden er

$$\|u\|_2^2 = x_0^2 \|v_0\|_2^2 + x_1^2 \|v_1\|_2^2 + \dots + x_{k-1}^2 \|v_{k-1}\|_2^2. \tag{8.10}$$

Den sidste ligning kendes som [Parsevals identitet](#).

Bevis. Lad os tage det indre produkt af (8.8) med v_i . Så får vi

$$\begin{aligned} \langle u, v_i \rangle &= \langle x_0 v_0 + x_1 v_1 + \dots + x_{k-1} v_{k-1}, v_i \rangle \\ &= x_0 \langle v_0, v_i \rangle + x_1 \langle v_1, v_i \rangle + \dots + x_{k-1} \langle v_{k-1}, v_i \rangle. \end{aligned}$$

Det eneste led på den højre side, som er ikke nul, er leddet $x_i \langle v_i, v_i \rangle = x_i \|v_i\|_2^2$. Det følger at $x_i = \langle u, v_i \rangle / \|v_i\|_2^2$, som påstået.

Tilsvarende har vi

$$\begin{aligned}\|u\|_2^2 &= \langle x_0 v_0 + \cdots + x_{k-1} v_{k-1}, x_0 v_0 + \cdots + x_{k-1} v_{k-1} \rangle \\ &= \sum_{i,j=0}^{k-1} x_i x_j \langle v_i, v_j \rangle.\end{aligned}$$

Men $\langle v_i, v_j \rangle$ er kun forskellig fra nul for $i = j$ og dens værdi er $\|v_i\|_2^2$, så

$$\|u\|_2^2 = \sum_{i=0}^{k-1} x_i^2 \|v_i\|_2^2,$$

som ønsket. \square

Eksempel 8.18. Lad v_0, v_1, v_2 være vektorerne fra eksempel 8.12. Vi har $\|v_0\|_2^2 = 3$, $\|v_1\|_2^2 = 8$ og $\|v_2\|_2^2 = 6$. For $u = (0, 1, 0)$, har vi så

$$\begin{aligned}u &= \begin{bmatrix} 0 \\ 1 \\ 0 \end{bmatrix} = \frac{1}{3} \begin{bmatrix} 1 \\ 1 \\ 1 \end{bmatrix} + \frac{-2}{8} \begin{bmatrix} 2 \\ -2 \\ 0 \end{bmatrix} + \frac{-1}{6} \begin{bmatrix} -1 \\ -1 \\ 2 \end{bmatrix} \\ &= \frac{1}{3} \begin{bmatrix} 1 \\ 1 \\ 1 \end{bmatrix} - \frac{1}{4} \begin{bmatrix} 2 \\ -2 \\ 0 \end{bmatrix} - \frac{1}{6} \begin{bmatrix} -1 \\ -1 \\ 2 \end{bmatrix}.\end{aligned}$$

\triangle

Givet (8.8) og (8.9) med $v_i, i = 0, \dots, k-1$, ortogonal og ingen 0, kan vi skrive den højre side af (8.8), som

$$\text{pr}_{v_0}(u) + \text{pr}_{v_1}(u) + \cdots + \text{pr}_{v_{k-1}}(u).$$

Husk at $\text{pr}_v(u) = Pu$ for $P = (1/\|v\|_2^2)vv^T$. Vi har så at den højre side af (8.8) er

$$\begin{aligned}&\frac{\langle v_0, u \rangle}{\|v_0\|_2^2} v_0 + \frac{\langle v_1, u \rangle}{\|v_1\|_2^2} v_1 + \cdots + \frac{\langle v_{k-1}, u \rangle}{\|v_{k-1}\|_2^2} v_{k-1} \\ &= \left(\frac{1}{\|v_0\|_2^2} v_0 v_0^T + \frac{1}{\|v_1\|_2^2} v_1 v_1^T + \cdots + \frac{1}{\|v_{k-1}\|_2^2} v_{k-1} v_{k-1}^T \right) u.\end{aligned}$$

Definition 8.19. For v_0, v_1, \dots, v_{k-1} ortogonal, med alle $v_i \neq 0$, er *projektionen langs v_0, v_1, \dots, v_{k-1}* givet ved Pu , hvor P er matricen

$$P = \frac{1}{\|v_0\|_2^2} v_0 v_0^T + \frac{1}{\|v_1\|_2^2} v_1 v_1^T + \cdots + \frac{1}{\|v_{k-1}\|_2^2} v_{k-1} v_{k-1}^T. \quad (8.11)$$

Proposition 8.20. For P som i definition 8.19 gælder

(a) $P^2 = P$ og

(b) $P^T = P$.

Desuden for vilkårlig $w \in \mathbb{R}^n$ har vi

(c) $w - Pw \perp v_i$ for alle i , og

(d) $\|w\|_2^2 = \|Pw\|_2^2 + \|w - Pw\|_2^2$.

Det følger at $u = Pw$ er vektoren af formen (8.8), som er tættest på w .

Bevis. For at vise $P^2 = P$ er det nok at bemærke at

$$(v_i v_i^T)(v_j v_j^T) = v_i(v_i^T v_j)v_j^T = \langle v_i, v_j \rangle v_i v_j^T = \begin{cases} 0, & \text{for } i \neq j, \\ \|v_i\|_2^2 v_i v_i^T, & \text{for } i = j. \end{cases}$$

Så følger resultatet ved at gange P^2 ud.

Ligningen $P^T = P$ følger fra at $v_i v_i^T$ er symmetrisk.

Observer at $\langle Pw, v_i \rangle = \langle v_i, Pw \rangle = v_i^T Pw$ og at $v_i^T v_j v_j^T = \langle v_i, v_j \rangle v_j^T$. Så har vi

$$\langle w - Pw, v_i \rangle = \langle w, v_i \rangle - v_i^T Pw = \langle w, v_i \rangle - \frac{1}{\|v_i\|_2^2} \|v_i\|_2^2 v_i^T w = \langle w, v_i \rangle - \langle v_i, w \rangle = 0.$$

Identiteten for $\|w\|_2^2$ følger nu fra Pythagoras sætning.

Antag at $y = y_0 v_0 + \dots + y_{k-1} v_{k-1}$ er ikke lige med Pw . Så er $Pw - y \neq 0$ og lemma 8.2(e) giver $\|Pw - y\|_2^2 > 0$. Da $w - Pw$ er vinkelret på hver v_i , er $w - Pw$ vinkelret på både y og Pw , og så vinkelret på $Pw - y$. Det følger fra Pythagoras at

$$\begin{aligned} \|w - y\|_2^2 &= \|(w - Pw) + (Pw - y)\|_2^2 = \|w - Pw\|_2^2 + \|Pw - y\|_2^2 \\ &> \|w - Pw\|_2^2. \end{aligned}$$

Dvs. at afstanden fra w til y er større end afstanden fra w til Pw . □

Eksempel 8.21. Lad $v_0 = (1, -2, 3)$, $v_1 = (3, 0, -1)$ være det ortogonale par u , w fra eksempel 8.1. Vi kan beregne

$$P \begin{bmatrix} 1 \\ 1 \\ 0 \end{bmatrix} = \frac{-1}{14} \begin{bmatrix} 1 \\ -2 \\ 3 \end{bmatrix} + \frac{3}{10} \begin{bmatrix} 3 \\ 0 \\ -1 \end{bmatrix} = \frac{1}{35} \begin{bmatrix} 29 \\ 5 \\ -3 \end{bmatrix}.$$

△

Eksempel 8.22. Lad os forsøge at approksimere funktionen $y = e^x$ for $-1 \leq x \leq 1$ med polynomier af grad højst 2. Vi vil gøre dette numerisk i python.

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

Vi start med funktioner 1, x og x^2 evalueret ved 100 punkter jævnt fordelt over $[-1, 1]$:

```
n = 100
x = np.linspace(-1, 1, 100)

# funktion 1
v0 = np.ones(100)[:, np.newaxis]
# funktion x
v1 = x[:, np.newaxis]
# funktion x**2
u2 = (x**2)[:, np.newaxis]
```

v_0 svarer til 1, v_1 til x og u_2 til x^2 . Vi ser at v_0 er stort set vinkelret på v_1

```
print(np.vdot(v0, v1))
```

8.215650382226158e-15

og at v_1 er næsten vinkelret på u_2

```
print(np.vdot(v1, u2))
```

7.216449660063518e-15

men u_2 er ikke vinkelret på v_0

```
print(np.vdot(v0, u2))
```

34.006734006734014

Sæt v_2 til at være u_2 minus projektionen af u_2 på v_0

8 ORTOGONALITET OG PROJEKTIONER

```
v2 = u2 - np.vdot(u2, v0) / np.vdot(v0, v0) * v0
```

(Vi har ikke brug for at trække projektionen på v_1 , da u_2 og v_0 er begge vinkelret på v_1 .) Så er v_2 stort set vinkelret på v_0 og v_1

```
print(np.vdot(v0, v2))
print(np.vdot(v1, v2))
```

```
-1.1102230246251565e-15
4.884981308350689e-15
```

og vi kan bruge v_0, v_1, v_2 som ortogonal samling. Disse funktioner er tegnet i figur 8.2.

```
fig, ax = plt.subplots()
ax.plot(x, v0[:, 0], label='v0')
ax.plot(x, v1[:, 0], label='v1')
ax.plot(x, v2[:, 0], label='v2')
ax.legend()
```

Lad u være repræsentationen af eksponentialfunktionen

```
u = np.exp(x)[:, np.newaxis]
```

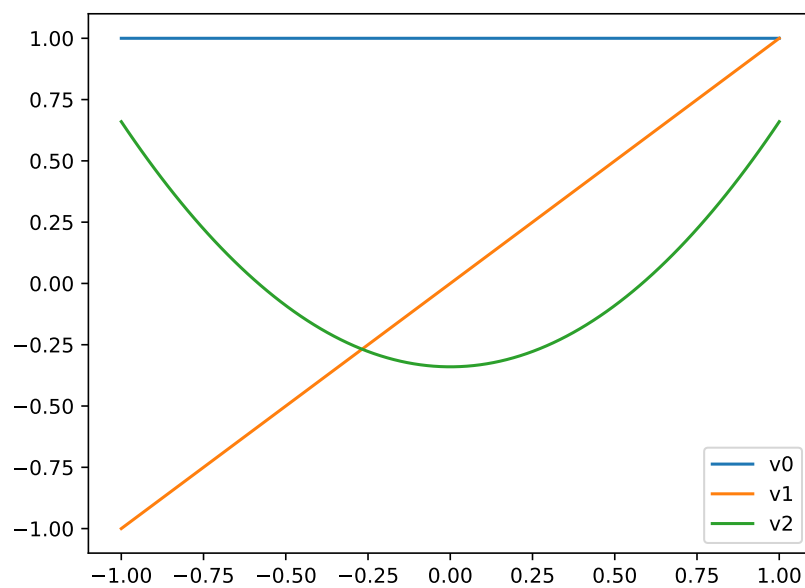
Dens projektion er så

```
u_proj = (1/np.vdot(v0, v0) * v0 @ (v0.T @ u)
          + 1/np.vdot(v1, v1) * v1 @ (v1.T @ u)
          + 1/np.vdot(v2, v2) * v2 @ (v2.T @ u))
```

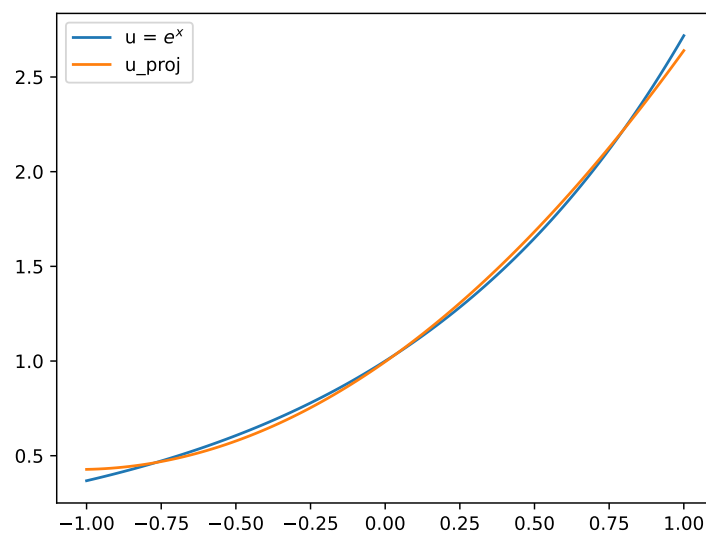
En plot af u og u_{proj} giver figur 8.3. Vi ser at u_{proj} ligger meget tæt på u over hele intervallet.

```
fig, ax = plt.subplots()
ax.plot(x, u[:, 0], label='u =  $e^{\{x\}}$ ')
ax.plot(x, u_proj[:, 0], label='u_proj')
ax.legend()
```

8.4 ORTOGONALITET

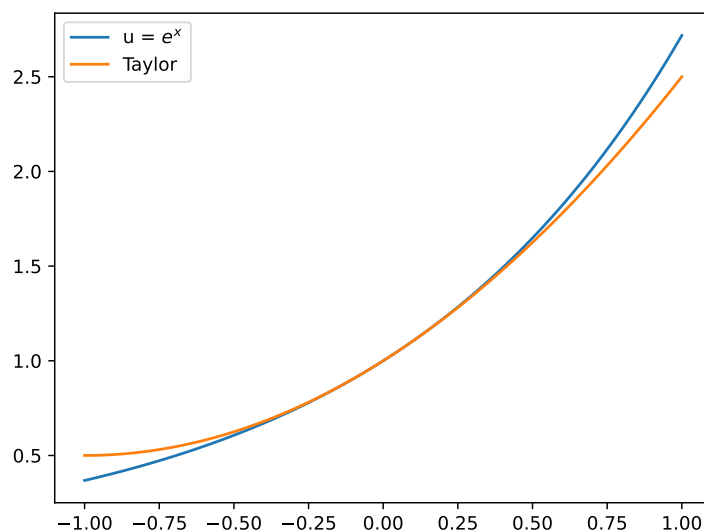


Figur 8.2: Den ortogonale samling v_0, v_1, v_2 af polynomier af højst anden grad.



Figur 8.3: Numerisk approksimation af $y = e^x$, $-1 \leq x \leq 1$.

8 ORTOGONALITET OG PROJEKTIONER



Figur 8.4: Approximation af $y = e^x$, $-1 \leq x \leq 1$, via dens anden ordens Taylor udvikling er kun godt tæt på $x = 0$.

Dette er i modsætning til den anden ordens Taylor udviklingen

$$e^x \approx 1 + x + \frac{1}{2}x^2$$

som kun er en god tilnærmelse til u tæt på $x = 0$, se figur 8.4.

```
fig, ax = plt.subplots()
ax.plot(x, u[:, 0], label='u =  $e^{\mathbf{x}}$ ')
ax.plot(x, (1. + x + (1/2.)*x**2), label='Taylor')
ax.legend()
```

△

Kapitel 9

Ortogonal matricer

Ortogonal og ortonormale samlinger af vektorer er tæt forbundne med ortogonale matricer. Desuden er ortogonale matricer det foretrukne værktøj for numerisk arbejde.

9.1 Definition og første eksempler

Definition 9.1. En kvadratisk matrix $A \in \mathbb{R}^{n \times n}$ er *ortogonal* hvis

$$A^T A = I_n.$$

Eksempel 9.2. For $n = 2$ er enhver rotationsmatrix ortogonal. Nemlig, for $c^2 + s^2 = 1$ har vi

$$R^T R = \begin{bmatrix} c & -s \\ s & c \end{bmatrix}^T \begin{bmatrix} c & -s \\ s & c \end{bmatrix} = \begin{bmatrix} c & s \\ -s & c \end{bmatrix} \begin{bmatrix} c & -s \\ s & c \end{bmatrix} = \begin{bmatrix} c^2 + s^2 & -cs + sc \\ -sc + cs & s^2 + c^2 \end{bmatrix} = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}.$$

Tilsvarende gælder for spejlingsmatricer

$$M^T M = \begin{bmatrix} c & s \\ s & -c \end{bmatrix} \begin{bmatrix} c & s \\ s & -c \end{bmatrix} = \begin{bmatrix} c^2 + s^2 & 0 \\ 0 & c^2 + s^2 \end{bmatrix} = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}.$$

Δ

Eksempel 9.3. Lad $u \in \mathbb{R}^n$ være en enhedsvektor. Så er

$$A = I_n - 2uu^T$$

9 ORTOGONALE MATRICER

er en ortogonal matrix: det ydre produkt uu^T er symmetrisk, og vi har

$$\begin{aligned} A^T A &= (I_n - 2uu^T)(I_n - 2uu^T) = I_n - 4uu^T + 4(uu^T)(uu^T) \\ &= I_n - 4uu^T + 4u(u^T u)u^T = I_n, \end{aligned}$$

da $u^T u = \langle u, u \rangle = 1$.

For eksempel for $n = 4$, er $u = (1/2, -1/2, 1/2, -1/2)$ en enhedsvektor, så

$$\begin{aligned} A &= \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} - 2 \begin{bmatrix} 1/2 \\ -1/2 \\ 1/2 \\ -1/2 \end{bmatrix} \begin{bmatrix} 1/2 & -1/2 & 1/2 & -1/2 \end{bmatrix} \\ &= \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} - \begin{bmatrix} 1/2 & -1/2 & 1/2 & -1/2 \\ -1/2 & 1/2 & -1/2 & 1/2 \\ 1/2 & -1/2 & 1/2 & -1/2 \\ -1/2 & 1/2 & -1/2 & 1/2 \end{bmatrix} \\ &= \begin{bmatrix} 1/2 & 1/2 & -1/2 & 1/2 \\ 1/2 & 1/2 & 1/2 & -1/2 \\ -1/2 & 1/2 & 1/2 & 1/2 \\ 1/2 & -1/2 & 1/2 & 1/2 \end{bmatrix} \end{aligned}$$

en ortogonal matrix. △

9.2 Egenskaber

Ortogonale matricer er netop dem, der bevarer det indre produkt. Dette medfører at ortogonale matricer også bevarer norm, afstand og vinkel, da disse er defineret ud fra det indre produkt. Så enhver figur bevarer dens form og størrelse under en transformation givet ved en ortogonal matrix.

For at vise at ortogonale matricer bevarer det indre produkt, har vi brug for at vide hvordan transponering påvirker matrixprodukter.

Proposition 9.4. For $B \in \mathbb{R}^{m \times n}$ og $C \in \mathbb{R}^{n \times p}$ har vi

$$(BC)^T = C^T B^T. \quad (9.1)$$

Observér at $C^T \in \mathbb{R}^{p \times n}$, og $B^T \in \mathbb{R}^{n \times m}$, så $C^T B^T$ er det eneste produkt via generelt kan danne ud fra C^T og B^T . Desuden er $(BC)^T$ og $C^T B^T$ begge matricer i $\mathbb{R}^{p \times m}$.

Bevis. Husk at BC har (i, j) -indgang $\sum_{k=0}^{n-1} b_{ik}c_{kj}$. Så dette er den (j, i) te indgang af $(BC)^T$. Men $C^T B^T$ har (j, i) -indgang

$$\sum_{k=0}^{n-1} (C^T)_{jk} (B^T)_{ki} = \sum_{k=0}^{n-1} c_{kj} b_{ik} = ((BC)^T)_{ji},$$

som påstået. \square

Proposition 9.5. For $A \in \mathbb{R}^{n \times n}$ ortogonal gælder

$$\langle Au, Av \rangle = \langle u, v \rangle, \quad \text{for alle } u, v \in \mathbb{R}^n. \quad (9.2)$$

Specielt gælder $\|Au\|_2 = \|u\|_2$ for alle $u \in \mathbb{R}^n$.

Omvendt, hvis $A \in \mathbb{R}^{n \times n}$ opfylder (9.2), så er A en ortogonal matrix.

Bevis. For A ortogonal har vi, ved brug af (9.1),

$$\langle Au, Av \rangle = (Au)^T (Av) = u^T A^T Av = u^T I_n v = u^T v,$$

så (9.2) holder.

Omvendt hvis (9.2) gælder, så er den (i, j) te indgang af $A^T A$ givet ved

$$(A^T A)_{ij} = e_i^T (A^T A) e_j = \langle Ae_i, Ae_j \rangle = \langle e_i, e_j \rangle = \begin{cases} 1, & \text{for } i = j, \\ 0, & \text{for } i \neq j, \end{cases}$$

som er den (i, j) te indgang af I_n . Så $A^T A = I_n$ og A er en ortogonal matrix. \square

Bemærkning 9.6. Numerisk er ortogonale matricer meget nyttig. Antag at en vektor $v \in \mathbb{R}^n$ har en fejl givet ved en vektor $w \in \mathbb{R}^n$ med $\|w\|_2 < c$, dvs. v repræsenteres af $v+w$. For A ortogonal har vi at Av repræsenteres af $A(v+w) = Av + Aw$, og fejlen er så Aw . Da A er ortogonal har vi $\|Aw\|_2 = \|w\|_2$, så fejlen i Av opfylder den samme estimering $\|Aw\|_2 < c$, som den oprindelig fejl w . \diamond

Desuden har ortogonale matricer andre pæne egenskaber. Specielt kan deres invers beregnes ved simpel ombytning af indgangerne.

Proposition 9.7. (a) Hvis A er ortogonal, så er A invertibel med invers A^T .

(b) For $A, B \in \mathbb{R}^{n \times n}$ ortogonale, er AB ortogonal.

Bevis. Del (a) følger direkte fra $A^T A = I_n$ og sætning 7.2.

For del (b), bruger vi (9.1):

$$(AB)^T (AB) = (B^T A^T) (AB) = B^T (A^T A) B = B^T I_n B = B^T B = I_n,$$

så AB er ortogonal. \square

9.3 Ortonormale vektorer og ortogonale matricer

Givet en samling vektorer v_0, v_1, \dots, v_{k-1} i \mathbb{R}^n kan vi danne en matrix V med disse vektorer som søjler

$$V = [v_0 \mid v_1 \mid \dots \mid v_{k-1}] \in \mathbb{R}^{n \times k}.$$

Matricen

$$G = V^T V \in \mathbb{R}^{k \times k}$$

kaldes *Grammatricen* for v_0, \dots, v_{k-1} , efter Jørgen Pedersen Gram (1850–1916). Vi har

$$\begin{aligned} G = V^T V &= \begin{bmatrix} v_0^T \\ v_1^T \\ \vdots \\ v_{k-1}^T \end{bmatrix} [v_0 \mid v_1 \mid \dots \mid v_{k-1}] \\ &= \begin{bmatrix} v_0^T v_0 & v_0^T v_1 & \dots & v_0^T v_{k-1} \\ v_1^T v_0 & v_1^T v_1 & \dots & v_1^T v_{k-1} \\ \vdots & \vdots & \ddots & \vdots \\ v_{k-1}^T v_0 & v_{k-1}^T v_1 & \dots & v_{k-1}^T v_{k-1} \end{bmatrix}, \end{aligned}$$

dvs. den (i, j) te indgang i G er det indre produkt mellem v_i og v_j :

$$G = (g_{ij}) = (\langle v_i, v_j \rangle).$$

Det følger at

Lemma 9.8. Samlingen v_0, v_1, \dots, v_{k-1} i \mathbb{R}^n er ortonormal hvis og kun hvis Grammatricen $G = V^T V \in \mathbb{R}^{k \times k}$ er lige med identitetsmatricen I_k :

$$G = V^T V = I_k.$$

□

Sætning 9.9. En ortonormal samling v_0, v_1, \dots, v_{k-1} i \mathbb{R}^n har højst $k = n$ vektorer.

9.3 ORTONORMALE VEKTORER OG ORTOGONALE MATRICER

Bevis. Hvis $k \geq n$, så betragter vi de første n vektorer og deres tilhørende matrix

$$A = [v_0 \mid v_1 \mid \dots \mid v_{n-1}] \in \mathbb{R}^{n \times n}.$$

Matricen A er kvadratisk, og lemma 9.8 giver at $A^T A = I_n$, så A er ortogonal. Det følger fra proposition 9.7(a) at A er invertibel med invers A^T . Lad $u \in \mathbb{R}^n$ være en vektor, som står vinkelret på alle de n vektorer v_0, v_1, \dots, v_{n-1} , dvs. $v_i^T u$ for $i = 0, \dots, n-1$. Da er

$$A^T u = \begin{bmatrix} v_0^T \\ v_1^T \\ \vdots \\ v_{n-1}^T \end{bmatrix} u = \begin{bmatrix} v_0^T u \\ v_1^T u \\ \vdots \\ v_{n-1}^T u \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ \vdots \\ 0 \end{bmatrix}.$$

Men A^T er invertibel, så lemma 7.7 giver at ligningssystemet $A^T u = 0$ har kun én løsning, nemlig $u = 0$. Så $u = 0$, og dermed kan den ortonormale samling v_0, v_1, \dots, v_{n-1} ikke tilføjes flere enhedsvektorer. \square

En ortonormal samling i \mathbb{R}^n med netop n vektorer kaldes en *ortonormal basis* for \mathbb{R}^n .

Lemma 9.8 giver straks det følgende resultat.

Proposition 9.10. Søjlerne af en ortogonal matrix $A \in \mathbb{R}^{n \times n}$ udgør en ortonormal basis for \mathbb{R}^n . \square

Eksempel 9.11. Fra en rotationsmatrix $R = \begin{bmatrix} c & -s \\ s & c \end{bmatrix}$, $c^2 + s^2 = 1$, har vi at

$$\begin{bmatrix} c \\ s \end{bmatrix}, \quad \begin{bmatrix} -s \\ c \end{bmatrix}$$

er en ortonormal basis for \mathbb{R}^2 .

Eksempel 9.3 giver at

$$\begin{bmatrix} 1/2 \\ 1/2 \\ -1/2 \\ 1/2 \end{bmatrix}, \quad \begin{bmatrix} 1/2 \\ 1/2 \\ 1/2 \\ -1/2 \end{bmatrix}, \quad \begin{bmatrix} -1/2 \\ 1/2 \\ 1/2 \\ 1/2 \end{bmatrix}, \quad \begin{bmatrix} 1/2 \\ -1/2 \\ 1/2 \\ 1/2 \end{bmatrix}$$

er en ortonormal basis for \mathbb{R}^4 . \triangle

9.4 Householdermatrix

Lad os skrive eksempel 9.3 på en lidt mere generel måde.

Definition 9.12. En *Householdermatrix* er en kvadratisk matrix $H \in \mathbb{R}^{n \times n}$, som er ortogonal og har formen

$$H = I_n - svv^T$$

for et $v \in \mathbb{R}^n$ og et $s \in \mathbb{R}$. Vektoren v kaldes den *Householdervektor*.

Hvis $v \neq 0$ og $s \neq 0$, har vi

$$s = \frac{2}{\|v\|_2^2}. \quad (9.3)$$

Dette følger fra først at bemærke at H er symmetrisk, og så regnestykket

$$\begin{aligned} I_n &= H^T H = (I_n - svv^T)(I_n - svv^T) = I_n - 2svv^T + s^2 v(v^T v)v^T \\ &= I_n + s((-2 + s\|v\|_2^2)vv^T), \end{aligned}$$

som giver $s\|v\|_2^2 = 2$ og dermed (9.3). Bemærk at vi har altid $s \geq 0$.

Proposition 9.13. Enhver Householdermatrix $H = I_n - svv^T$ opfylder

- (a) $H^T = H$,
- (b) $H^2 = I_n$,
- (c) $Hv = -v$, hvis $s \neq 0$, og
- (d) $Hw = w$ for alle $w \perp v$.

Det vil sige at H er en spejlingsmatrix, der sender v til $-v$ og fastholder alle vektorer, som står vinkelret på v . Se figur 9.1.

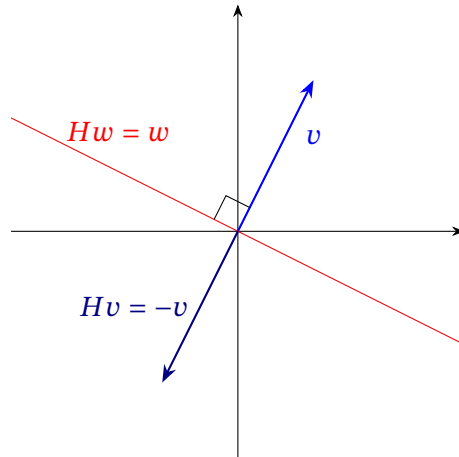
Bevis. Vi har allerede bemærket at $H^T = H$, og så er $H^2 = H^T H$, som vi har vist er lige med I_n . For de sidste to dele, først har vi

$$Hv = (I_n - svv^T)v = v - sv(v^T v) = v - s\|v\|_2^2 v = v - 2v = -v,$$

og for $w \perp v$ har vi $v^T w = 0$, så

$$Hw = w - sv(v^T w) = w - 0 = w,$$

som påstået. □



Figur 9.1: Effekten af en Householdermatrix.

I mange situationer, f.eks. løsning af ligningssystemer, vil vi gerne flytte en given vektor til en med mange indgange lige med 0. Householdermatricer er rigtig til god til dette. Faktisk kan vi flytte til et multiplum af $e_0 = (1, 0, \dots, 0)$. For numerisk sikkerhed er det bedst at være forsigtig med fortegnet af dette multiplum. I det næste resultat træffer vi det rigtige valg.

Proposition 9.14. Givet en enhedsvektor $u = (u_0, u_1, \dots, u_{n-1}) \in \mathbb{R}^n$, sæt

$$\varepsilon = \begin{cases} -1, & \text{hvis } u_0 \geq 0, \\ +1, & \text{hvis } u_0 < 0. \end{cases}$$

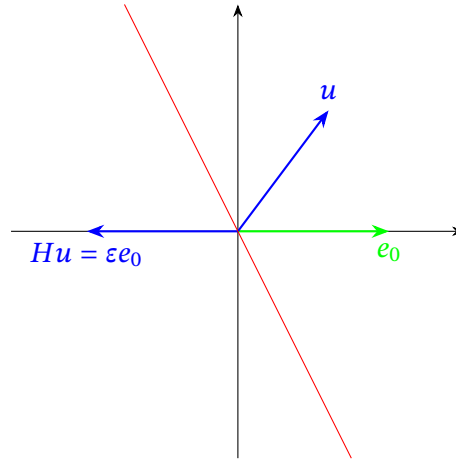
Så findes der en Householdermatrix $H = I_n - svv^T$ med

$$He_0 = \varepsilon u, \quad v = \begin{bmatrix} 1 \\ v_1 \\ \vdots \\ v_{n-1} \end{bmatrix} \quad \text{og} \quad s = 1 + |u_0|.$$

Bevis. Vi skal finde et passende v . Først har vi $He_0 = e_0 - sv(v^T e_0)$, men $v^T e_0 = 1$, så $He_0 = e_0 - sv$. Ligningen $He_0 = \varepsilon u$, giver $e_0 - sv = \varepsilon u$, som vi omskriver til

$$sv = e_0 - \varepsilon u.$$

9 ORTOGONALE MATRICER



Figur 9.2: Spejling af en enhedsvektor u til εe_0 via en Householdermatrix.

Kikkes på den 0te indgang har vi

$$s = 1 - \varepsilon u_0 = 1 + |u_0| \geq 1.$$

Sæt nu

$$v = \frac{1}{s}(e_0 - \varepsilon u). \quad (9.4)$$

Beregningen $\|v\|_2^2 = (1 + 2\varepsilon \langle e_0, u \rangle + \|u\|_2^2) / s^2 = (1 - 2|u_0| + 1) / s^2 = 2(1 + |u_0|) / s^2 = 2/s$, bekræfter at $s = 2/\|v\|_2^2$ og dermed at H er en Householdermatrix. \square

Husk at $H^2 = I_n$, så ligningen $He_0 = \varepsilon u$ giver

$$Hu = \varepsilon e_0 = \begin{bmatrix} \varepsilon \\ 0 \\ \vdots \\ 0 \end{bmatrix}.$$

Bemærkning 9.15. Fortegnet ε er valgt netop således at vi får $s \geq 1$. Da u er en enhedsvektor har vi $|u_0| \leq 1$, så vi har også $s \leq 2$. Det betyder at i ligningen (9.4) at der er ikke numeriske problemer med at dele med s . \diamond

Eksempel 9.16. Vi ønsker at bestemme en Householdermatrix H der flytter $x = (1, 2, 3)$ til et multiplum af e_0 .

9.4 HOUSEHOLDERMATRIX

Sæt $u = x/\|x\|_2 = x/\sqrt{14}$. Så er $u_0 = 1/\sqrt{14}$, $\varepsilon = -1$, $s = 1 + (1/\sqrt{14}) = (1 + \sqrt{14})/\sqrt{14}$ og

$$v = \frac{\sqrt{14}}{1 + \sqrt{14}} \begin{bmatrix} 1 + 1/\sqrt{14} \\ 2/\sqrt{14} \\ 3/\sqrt{14} \end{bmatrix} = \frac{1}{1 + \sqrt{14}} \begin{bmatrix} \sqrt{14} + 1 \\ 2 \\ 3 \end{bmatrix} = \begin{bmatrix} 1 \\ 2/(1 + \sqrt{14}) \\ 3/(1 + \sqrt{14}) \end{bmatrix}.$$

Vi kan lave dette beregning i python og bekræfter effekten af den frembragte Householdertransformation:

```
>>> import numpy as np
>>> x = np.array([1.0, 2.0, 3.0])[:, np.newaxis]
>>> u = x / np.linalg.norm(x)
>>> u[0,0]
0.2672612419124244
>>> eps = -1 if u[0,0] >=0 else 1
>>> eps
-1
>>> s = 1 + np.abs(u[0,0])
>>> s
1.2672612419124243
>>> v = (-eps/s) * u
>>> v[0,0] = 1
>>> v
array([[1.          ],
       [0.42179344],
       [0.63269017]])
>>> Hx = x - s * v @ (v.T @ x)
>>> Hx
array([[ -3.74165739e+00],
       [-4.44089210e-16],
       [-4.44089210e-16]])
>>> float(np.linalg.norm(x))
3.7416573867739413
```

△

Bemærk at vi implementer multiplikation med $H = I_n - svv^T$ på et x , som

$$Hx = x - sv(v^T x)$$

9 ORTOGONALE MATRICER

for at spare på antallet af flops. Desuden behøver vi hverken at konstruere eller gemme matricen H , men kan nøjes med at gemme skalaren s og vektoren v . (Da $v_0 = 1$ kan dette plads i v bruges til at gemme s .) Dette sparer hukommelsen. Det kan være nyttigt at oprette en funktion, som udfører Householdertransformationen givet af s og v på en anden vektor x .

```
import numpy as np

def house_transformation(s, v, x):
    return x - s * v @ (v.T @ x)
```

Dette bruges på følgende måde

```
rng = np.random.default_rng()
v = rng.random((10,1))
s = 2 / np.vdot(v,v)

x = rng.random((10,1))
Hx = house_transformation(s, v, x)
print(Hx)
```

```
[[-0.79417676]
 [-0.6750574 ]
 [-0.16794743]
 [-0.3987978 ]
 [ 0.44489081]
 [ 0.19773045]
 [ 0.69252731]
 [-0.38458744]
 [-0.23856773]
 [-0.36687549]]
```

Vi kan bekræfte at $H^2x = x$ og at H afbilder v til $-v$ indenfor machine epsilon:

```
print(np.allclose(house_transformation(s, v, Hx), x,
                  atol = np.finfo(float).eps))
print(np.allclose(house_transformation(s, v, v), -v,
                  atol = np.finfo(float).eps))
```

True
True

9.5 Udvidelse til ortonormal basis

Sætning 9.17. Givet en enhedsvektor $u \in \mathbb{R}^n$, så findes der en ortonormal basis v_0, v_1, \dots, v_{n-1} for \mathbb{R}^n med $v_0 = u$.

Bevis. Definér ε og H som i proposition 9.14. Da er $He_0 = \varepsilon u$ og H er en ortogonal matrix. Matricen $A = \varepsilon H$ er også ortogonal, da $\varepsilon^2 = 1$, og har u som 0te søjle. Vores ortonormal basis fås nu som søjlerne v_0, \dots, v_{n-1} af A . \square

Korollar 9.18. For u_0, u_1, \dots, u_{k-1} ortonormal i \mathbb{R}^n kan den udvides til en ortonormal basis $u_0, \dots, u_{k-1}, \dots, u_{n-1}$ for \mathbb{R}^n .

Bevis. Da ortonormale baser er det sammen som søjlerne af en ortogonal matrix, se proposition 9.10, er det nok at finde en ortogonal matrix A med søjler 0 til $k-1$ givet ved u_0 til u_{k-1} .

Brug proposition 9.14 til at finde ε og en Householdermatrix H med $He_0 = \varepsilon u_0$. Sæt $B = \varepsilon H$, som er ortogonal og har u_0 som 0te søjle. Vi har $u_0 = Be_0$ og $e_0 = Bu_0$. For $i > 0$ har vi også at $\langle Bu_i, e_0 \rangle = \langle Bu_i, Bu_0 \rangle = \langle u_i, u_0 \rangle = 0$. Dette siger at den 0te indgang i Bu_i er 0, så

$$Bu_i = \begin{bmatrix} 0 \\ w_i \end{bmatrix}, \quad i = 1, \dots, k-1$$

for vektorer $w_1, \dots, w_{k-1} \in \mathbb{R}^{n-1}$. Da $\langle w_i, w_j \rangle = \langle Bu_i, Bu_j \rangle = \langle u_i, u_j \rangle$, er w_1, \dots, w_{k-1} ortonormal i \mathbb{R}^{n-1} . Per induktion, findes der en ortogonal matrix $C \in \mathbb{R}^{(n-1) \times (n-1)}$ med $C = [w_1 \mid \dots \mid w_{k-1} \mid \dots]$. Dannes matricen

$$D = \left[\begin{array}{c|c} 1 & 0 \\ \hline 0 & C \end{array} \right] \in \mathbb{R}^{n \times n}$$

er D ortogonal, og

$$\begin{aligned} D &= [e_0 \mid Bu_1 \mid \dots \mid Bu_{k-1} \mid \dots] \\ &= [Bu_0 \mid Bu_1 \mid \dots \mid Bu_{k-1} \mid \dots] \\ &= B[u_0 \mid u_1 \mid \dots \mid u_{k-1} \mid \dots]. \end{aligned}$$

Matricen $A = B^T D$ er ortogonal og vi har

$$[u_0 \mid u_1 \mid \dots \mid u_{k-1} \mid \dots] = B^T D = A,$$

som ønsket. □

9.6 Plan geometri: trigonometriske identiteter

Isometrier i planen giver os muligheden for at vise nogle trigonometriske identiteter. Allerede i kapitel 2 brugte vi en formel for cosinus af en sum, men uden bevis. I dette afsnit vil vi vise de følgende to identiteter for cosinus og sinus af summen af vinkler.

$$\cos(\theta + \varphi) = \cos(\theta) \cos(\varphi) - \sin(\theta) \sin(\varphi), \quad (9.5)$$

$$\sin(\theta + \varphi) = \cos(\theta) \sin(\varphi) + \sin(\theta) \cos(\varphi). \quad (9.6)$$

Andre nyttige identiteter følger fra disse, som vi skal se om lidt.

For at dette skal give mening, har vi først brug for at definere sinusfunktionen. Vi arbejder i planen \mathbb{R}^2 . Givet to enhedsvektorer u og v , har vi defineret $\cos(\theta) = \langle u, v \rangle$, se definition 8.5. I planen vil vi tillade at θ ligger i intervallet $[0, 2\pi)$, intervallet $[-\pi, \pi)$ eller i \mathbb{R} .

Betragt nu matricen

$$J_2 = \begin{bmatrix} 0 & -1 \\ 1 & 0 \end{bmatrix}.$$

Vi sætter

$$\sin(\theta) = \langle u, J_2 v \rangle.$$

For $v = e_0$ er $Jv = e_1$, og $\sin(\theta)$ er dermed y -koordinaten af u . Dette stemmer overens med den sædvanlige definition på sinus, se figur 9.3. Bemærk at selve vinklen θ mellem u og e_0 er længden af buestykket langs enhedscirklen fra e_0 til u , målt med positiv omløbsretning.

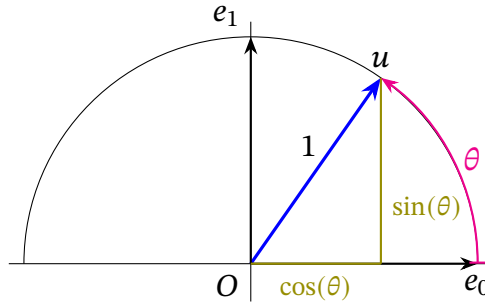
Hvis $A \in \mathbb{R}^{2 \times 2}$ er en isometri, så er $\cos(\theta)$ det samme for Au, Av som for u, v . For sinus har vi

$$\sin(\theta) = \langle u, J_2 v \rangle = \langle Au, AJ_2 v \rangle,$$

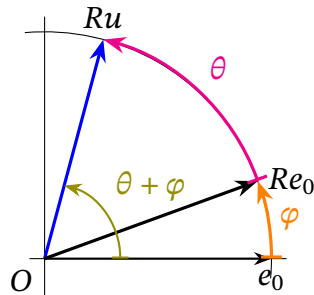
så vi får den samme sinusværdi hvis

$$AJ_2 = J_2 A. \quad (9.7)$$

9.6 PLAN GEOMETRI: TRIGONOMETRISKE IDENTITETER



Figur 9.3: Cosinus og sinus for en enhedsvektor i planen.



Figur 9.4: Enhedsvektoren u drejet igennem vinkel φ .

En lineær isometri af planen, som opfylder (9.7), siges at være *positiv*. Vi har så at positive isometrier bevarer både $\cos(\theta)$ og $\sin(\theta)$.

Et eksempel på en positiv isometri er givet ved matricen

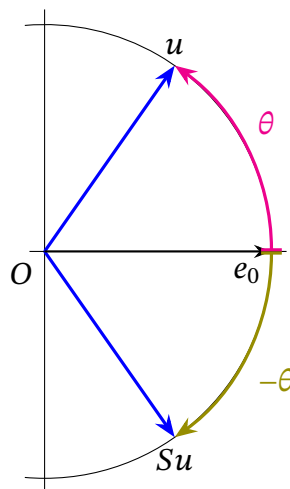
$$R = \begin{bmatrix} \cos(\varphi) & -\sin(\varphi) \\ \sin(\varphi) & \cos(\varphi) \end{bmatrix}.$$

Matricen er ortogonal og $RJ_2 = J_2R$ bekræftes ved direkte udregning. Bemærk at

$$Re_0 = \begin{bmatrix} \cos(\varphi) & -\sin(\varphi) \\ \sin(\varphi) & \cos(\varphi) \end{bmatrix} \begin{bmatrix} 1 \\ 0 \end{bmatrix} = \begin{bmatrix} \cos(\varphi) \\ \sin(\varphi) \end{bmatrix},$$

så R flytter e_0 til enhedsvektoren med vinkel φ til e_0

Lad $u = (\cos(\theta), \sin(\theta))$, som har vinkel θ til e_0 -aksen. Da R er en positiv isometri, er vinklen mellem Ru og Re_0 også θ . Dette illustreres i figur 9.4, hvor

Figur 9.5: Spejling i x -aksen.

man kan også se at vinklen mellem Ru og e_0 er $\theta + \varphi$. Dette giver

$$\begin{aligned} \begin{bmatrix} \cos(\theta + \varphi) \\ \sin(\theta + \varphi) \end{bmatrix} &= Ru = \begin{bmatrix} \cos(\varphi) & -\sin(\varphi) \\ \sin(\varphi) & \cos(\varphi) \end{bmatrix} \begin{bmatrix} \cos(\theta) \\ \sin(\theta) \end{bmatrix} \\ &= \begin{bmatrix} \cos(\theta) \cos(\varphi) - \sin(\theta) \sin(\varphi) \\ \cos(\theta) \sin(\varphi) + \sin(\theta) \cos(\varphi) \end{bmatrix}. \end{aligned}$$

De 0te og 1te koordinater i denne relation er netop ligninger (9.5) og (9.6), som ønsket.

Bemærk isometrien $S = \begin{bmatrix} 1 & 0 \\ 0 & -1 \end{bmatrix}$ sender $u = (\cos(\theta), \sin(\theta))$ til $Su = (\cos(\theta), -\sin(\theta))$. Men Su har vinkel $-\theta$ til e_0 , se figur 9.5, buelængden er det samme men vil skal tilføje et minus fortegn, da vi har den modsatte om-løbsretning. Dette giver at $Su = (\cos(-\theta), \sin(-\theta))$ og vi får dermed

$$\cos(-\theta) = \cos(\theta) \quad \text{og} \quad \sin(-\theta) = -\sin(\theta). \quad (9.8)$$

Som konsekvens har vi de følgende nyttige relationer.

$$2 \cos(\alpha) \cos(\beta) = \cos(\alpha + \beta) + \cos(\alpha - \beta), \quad (9.9)$$

$$2 \cos(\alpha) \sin(\beta) = \sin(\alpha + \beta) - \sin(\alpha - \beta), \quad (9.10)$$

$$2 \sin(\alpha) \sin(\beta) = \cos(\alpha + \beta) - \cos(\alpha - \beta). \quad (9.11)$$

9.6 PLAN GEOMETRI: TRIGONOMETRISKE IDENTITETER

Disse kan alle vises ved at begynde med den højre side, anvende (9.5) og (9.6) og derefter bruge (9.8). F.eks.

$$\begin{aligned}\cos(\alpha + \beta) - \cos(\alpha - \beta) &= \cos(\alpha + \beta) - \cos(\alpha + (-\beta)) \\ &= \cos(\alpha) \cos(\beta) - \sin(\alpha) \sin(\beta) \\ &\quad - (\cos(\alpha) \cos(-\beta) - \sin(\alpha) \sin(-\beta)) \\ &= \cos(\alpha) \cos(\beta) - \sin(\alpha) \sin(\beta) \\ &\quad - \cos(\alpha) \cos(\beta) + \sin(\alpha) \sin(-\beta) \\ &= 2 \cos(\alpha) \cos(\beta).\end{aligned}$$

Kapitel 10

Singulærværdidekomponering

En vektor $v \in \mathbb{R}^2$ i planen beskrives fint af længden $\|v\|_2$ og vinklen θ fra x -aksen (så længe $v \neq 0$). Som vi har set før kan denne vinkeloplysning erstattes af enhedsvektoren $\begin{bmatrix} c \\ s \end{bmatrix}$, $c^2 + s^2 = 1$, med $c = \cos(\theta)$, $s = \sin(\theta)$. Husk at $\begin{bmatrix} c \\ s \end{bmatrix} = v/\|v\|_2$.

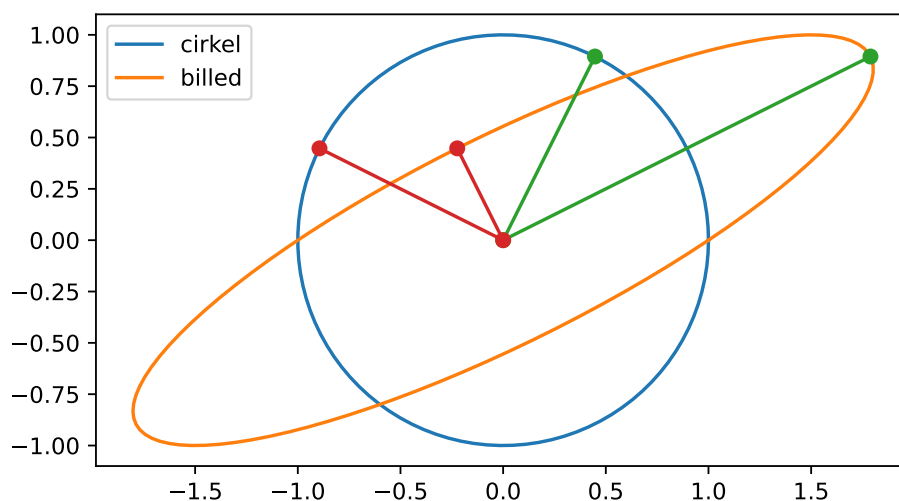
I højere dimensioner, $v \in \mathbb{R}^n$, $v \neq 0$, beskrives tilsvarende ved dens længde $\sigma = \|v\|_2$ og enhedsvektoren $u = v/\|v\|_2$. Betragtes v som en matrix $v \in \mathbb{R}^{n \times 1}$, har vi

$$v = \sigma u = \begin{bmatrix} u_0 \\ u_1 \\ \vdots \\ u_{n-1} \end{bmatrix} \begin{bmatrix} \sigma \\ 1 \end{bmatrix}$$

hvor $\begin{bmatrix} \sigma \end{bmatrix}$ og $\begin{bmatrix} 1 \end{bmatrix}$ er (1×1) -matricer. Dette er eksempel på en (tynd) singulærværdidekomponering (SVD) af en matrix. Tallet σ er en singulærværdi; vektorerne $u \in \mathbb{R}^n$ og $\begin{bmatrix} 1 \end{bmatrix} \in \mathbb{R}^1$ venstre- og højresingulærvektorer.

En tilsvarende dekomponering findes for vilkårlige matricer $A \in \mathbb{R}^{m \times n}$, og giver os god oplysning om deres struktur. Desuden findes der effektive og præcise metoder til at beregne singulærværdidekomponering på, så det er blevet til et vigtigt numerisk værktøj.

Singulærværdierne gemmer på oplysning om A der er relevant for at vurdere pålideligheden af f.eks. løsninger af lineære ligningssystemer $Ax = b$, og dekomponeringen kan bruges til approksimering af A i komprimeret form. Brug af singulærværdier og singulærvektorer er også meget relevant for analyse af datamængder: et kraftigt ofte brugt dataværktøj er principalkomponent



Figur 10.1: Billedet af en cirkel efter matrixmultiplikation med $A \in \mathbb{R}^{2 \times 2}$.

analyse; singulærværdidekomponering udføre dette analyse, og mere, på en numerisk mere pålidelig måde.

10.1 Singulærværdier i dimension 2

Lad os begynde med at betragte en (2×2) -matrix

$$A = \begin{bmatrix} a & b \\ c & d \end{bmatrix}.$$

Vi kan tegne billedet af enhedscirklen $(\cos(\theta), \sin(\theta))$ under transformationen $x \mapsto Ax$. Et typisk resultat vises i figur 10.1.

Vi ser at dette billede af cirklen er altid ellipseformet, uanset hvilke matrix A vi bruger. (Nogle gange vil denne ellipse være mast sammen til et linjestykke eller et enkelt punkt).

En ellipse har to akser: en stor akse og en lille akse. Halvdelen af længden af disse to akser er netop det vi kalder singulærværdierne σ_0 og σ_1 for A . Det to akser står vinkelret på hinanden og peger i retninger u_0 og u_1 , hvor vi tager u_i til at være enhedsvektorer.

10.1 SINGULÆRVÆRDIER I DIMENSION 2

Det er et overraskende faktum at der findes enhedsvektorer v_0 og v_1 i \mathbb{R}^2 , som står vinkelret på hinanden, således at

$$Av_0 = \sigma_0 u_0, \quad Av_1 = \sigma_1 u_1. \quad (10.1)$$

Dette er en uventet da en vilkårlig matrix A bevarer ikke vinkler mellem tilfældige vektorer.

Lad os samle u_0, u_1 og v_0, v_1 til (2×2) -matricer

$$U = [u_0 \mid u_1] \quad \text{og} \quad V = [v_0 \mid v_1].$$

Så er U og V ortogonale matricer, da deres søjler er enhedsvektorer, der står vinkelret på hinanden. Sættes

$$u_0 = \begin{bmatrix} x_0 \\ y_0 \end{bmatrix}, \quad u_1 = \begin{bmatrix} x_1 \\ y_1 \end{bmatrix}.$$

kan vi omskrive (10.1) til

$$\begin{aligned} AV &= [Av_0 \mid Av_1] = [\sigma_0 u_0 \mid \sigma_1 u_1] = \begin{bmatrix} x_0 \sigma_0 & x_1 \sigma_1 \\ y_0 \sigma_0 & y_1 \sigma_1 \end{bmatrix} \\ &= \begin{bmatrix} x_0 & x_1 \\ y_0 & y_1 \end{bmatrix} \begin{bmatrix} \sigma_0 & 0 \\ 0 & \sigma_1 \end{bmatrix} = U\Sigma, \end{aligned} \quad (10.2)$$

hvor

$$\Sigma = \begin{bmatrix} \sigma_0 & 0 \\ 0 & \sigma_1 \end{bmatrix}.$$

Da V er en ortogonal matrix, er den invertibel med invers V^T . Ganges (10.2) på den højre side med $V^{-1} = V^T$, fås

$$A = U\Sigma V^T. \quad (10.3)$$

Dette kaldes singulærværdidekomponeringen (SVD) af A .

Denne SVD kan beregnes i python via `np.linalg.svd`. Denne python funktionen giver tre objekter: (a) matricen U , (b) en ndarray med en akse, som indholder singulærværdierne σ_0, σ_1 , og (c) den transponerede matrix V^T .

```
>>> import numpy as np

>>> a = np.array([[1.0,  7.0],
```

10 SINGULÆRVÆRDIDEKOMPONERING

```
... [1.0, -1.0]])

>>> u, s, vt = np.linalg.svd(a)
>>> s
array([7.12310563, 1.12310563])
>>> u
array([[ -0.99250756,  0.12218326],
       [ 0.12218326,  0.99250756]])
>>> vt
array([[ -0.12218326, -0.99250756],
       [ 0.99250756, -0.12218326]])
```

Disse kan bruges til at rekonstruere a

```
>>> u @ np.diag(s) @ vt
array([[ 1.,  7.],
       [ 1., -1.]])
```

Her danner $\text{np.diag}(s)$ en diagonal matrix med indgange fra s sat langs diagonalen.

(10.3) kan også skrives via ydre produkter. Sæt

$$v_0 = \begin{bmatrix} a_0 \\ b_0 \end{bmatrix}, \quad v_1 = \begin{bmatrix} a_1 \\ b_1 \end{bmatrix}.$$

Så er (10.3)

$$\begin{aligned} U\Sigma V^T &= [\sigma_0 u_0 \mid \sigma_1 u_1] \begin{bmatrix} v_0^T \\ v_1^T \end{bmatrix} \\ &= \begin{bmatrix} x_0 \sigma_0 & x_1 \sigma_1 \\ y_0 \sigma_0 & y_1 \sigma_1 \end{bmatrix} \begin{bmatrix} a_0 & b_0 \\ a_1 & b_1 \end{bmatrix} = \begin{bmatrix} x_0 \sigma_0 a_0 + x_1 \sigma_1 a_1 & x_0 \sigma_0 b_0 + x_1 \sigma_1 b_1 \\ y_0 \sigma_0 a_0 + y_1 \sigma_1 a_1 & y_0 \sigma_0 b_0 + y_1 \sigma_1 b_1 \end{bmatrix} \\ &= \sigma_0 \begin{bmatrix} x_0 a_0 & x_0 b_0 \\ y_0 a_0 & y_0 b_0 \end{bmatrix} + \sigma_1 \begin{bmatrix} x_1 a_1 & x_1 b_1 \\ y_1 a_1 & y_1 b_1 \end{bmatrix} \\ &= \sigma_0 \begin{bmatrix} x_0 \\ y_0 \end{bmatrix} \begin{bmatrix} a_0 & b_0 \end{bmatrix} + \sigma_1 \begin{bmatrix} x_1 \\ y_1 \end{bmatrix} \begin{bmatrix} a_1 & b_1 \end{bmatrix} \\ &= \sigma_0 u_0 v_0^T + \sigma_1 u_1 v_1^T. \end{aligned} \tag{10.4}$$

10.2 SVD generelt

Sætning 10.1. Enhver matrix $A \in \mathbb{R}^{m \times n}$ har en [singulærværdidekomponering](#)

$$A = U \Sigma V^T \quad (10.5)$$

med $U \in \mathbb{R}^{m \times m}$ og $V \in \mathbb{R}^{n \times n}$ ortogonale matricer, med $\Sigma \in \mathbb{R}^{m \times n}$ en diagonal matrix

$$\Sigma = \text{diag}(\sigma_0, \dots, \sigma_{k-1}) = \begin{bmatrix} \sigma_0 & 0 & 0 & \dots \\ 0 & \sigma_1 & 0 & \dots \\ \vdots & & \ddots & \vdots \end{bmatrix}, \quad k = \min\{m, n\}, \quad (10.6)$$

og med $\sigma_0 \geq \sigma_1 \geq \dots \geq \sigma_{k-1} \geq 0$.

Tallene σ_i kaldes [singulærværdier](#) for A . Søjlerne af U er [venstresingulærvektorer](#); søjlerne af V er [højresingulærvektorer](#).

Beviset gives i afsnit 10.4. Her vil vi fokusere på fortolkningen og anvendelser.

Eksempel 10.2. For $A \in \mathbb{R}^{2 \times 3}$ har dens SVD formen

$$A = [u_0 \mid u_1] \begin{bmatrix} \sigma_0 & 0 & 0 \\ 0 & \sigma_1 & 0 \end{bmatrix} \begin{bmatrix} v_0^T \\ v_1^T \\ v_2^T \end{bmatrix}, \quad u_i \in \mathbb{R}^{2 \times 1}, v_j \in \mathbb{R}^{3 \times 1}.$$

Dette kan skrives via ydre produkter, som vi gjorde for (2×2) -matricer i (10.4), og giver

$$A = \sigma_0 u_0 v_0^T + \sigma_1 u_1 v_1^T.$$

Bemærk at dette form tager ikke vektoren v_2 i brug. Det svarer til matrixproduktet

$$A = [u_0 \mid u_1] \begin{bmatrix} \sigma_0 & 0 \\ 0 & \sigma_1 \end{bmatrix} \begin{bmatrix} v_0^T \\ v_1^T \end{bmatrix},$$

som kaldes en tynd SVD. △

Generelt er (10.5) det samme som

$$A = \sigma_0 u_0 v_0^T + \sigma_1 u_1 v_1^T + \dots + \sigma_{k-1} u_{k-1} v_{k-1}^T, \quad (10.7)$$

hvor igen $k = \min\{m, n\}$. Der er sørget for at u_i og v_j er enhedsvektorer, og at faktorerne σ_i er aftagende. Derfor er de led, der bidrager mest til A , dem der

kommer først i (10.7). Sagt med andre ord, er den største del af oplysningen i A indeholdt i de første led af (10.7).

I en del applikationer kan man nøjes med at approksimere A med de første r led i (10.7), $r \leq k$. Vi kalder

$$\sigma_0 u_0 v_0^T + \sigma_1 u_1 v_1^T + \cdots + \sigma_{r-1} u_{r-1} v_{r-1}^T$$

en *forkortet SVD* med r led.

10.3 Eksempler på SVD

10.3.1 Punkter i planen

Betragt 1000 punkter i \mathbb{R}^2 , som til venstre i Figur 10.2.

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

rng = np.random.default_rng()

n = 1000
a = (np.array([[1.0, 5.0], [1.0, -1.0]])
      @ rng.standard_normal((2,n)))

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

15.167098910781126

```
fig, ax = plt.subplots()

ax.set_xlim(-20, 20)
ax.set_ylim(-20, 20)
ax.set_aspect('equal')

ax.plot(*a, 'o', markersize = 2)
```

Hvis vi beregner en fuld SVD, får vi en V^T , som er meget stor.

```
u, s, vt = np.linalg.svd(a)
print(u.shape, s.shape, vt.shape)
```

```
(2, 2) (2,) (1000, 1000)
```

I stedet for beregner vi en tynd SVD ved hjælp af tilføjelsen

```
full_matrices=False
```

til `np.linalg.svd`:

```
u, s, vt = np.linalg.svd(a, full_matrices=False)
print(u.shape, s.shape, vt.shape)
```

Vi ser at den første singulærværdi er væsentlig større end den anden

```
print(s)
```

```
[159.15048709  37.63612895]
```

Denne singulærværdi har venstresingulærvektor, som er den først søjle af `u`

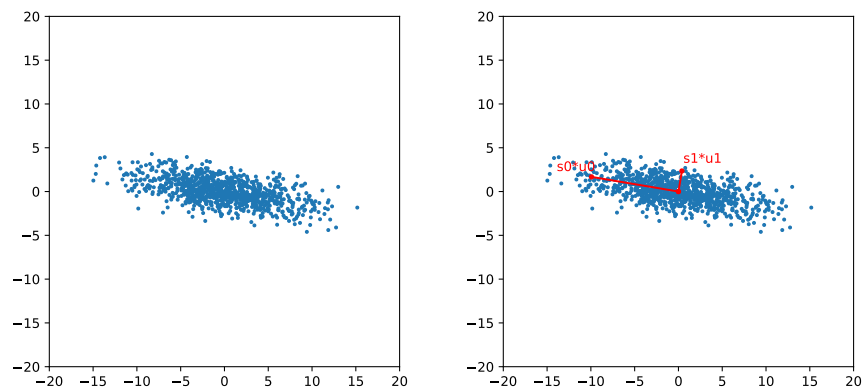
```
print(u)
```

```
[[-0.98610413  0.1661284 ]
 [ 0.1661284   0.98610413]]
```

```
origo = np.zeros((2,1))
scale = 2/np.sqrt(n)
tscale = 1.4*scale

fig, ax = plt.subplots()

ax.set_xlim(-20, 20)
ax.set_ylim(-20, 20)
ax.set_aspect('equal')
```



Figur 10.2: SVD af punktsamling i planen.

```
ax.plot(*a, 'o', markersize = 2)

ax.plot(*(np.hstack([origo, u[:, [0]]*s[0]*scale])),
        'red', marker='.')
ax.text(*u[:, [0]]*s[0]*tscale, 's0*u0', color='red')

ax.plot(*(np.hstack([origo, u[:, [1]]*s[1]*scale])),
        'red', marker='.')
ax.text(*u[:, [1]]*s[1]*tscale, 's1*u1', color='red')
```

Figur 10.2 demonstrerer at at de venstresingulærvektorer giver retningerne hvor variationen af punkterne er hhv. størst og mindst.

10.3.2 Billede med støj

For det næste eksempel, lad os danne en et simpelt billede af cifret 0:

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

a = np.zeros((20,14))
a[2, 2:12] = 1
a[3, 2:12] = 1
```

```

a[16, 2:12] = 1
a[17, 2:12] = 1
a[2:18, 2] = 1
a[2:18, 3] = 1
a[2:18, 10] = 1
a[2:18, 11] = 1

fig, ax = plt.subplots()
ax.matshow(a, cmap='coolwarm')

```

Dette giver billedet til venstre i figur 10.3.

Lad os tilføje støj til dette billede. Bemærk at støjniveauet er ret højt i forhold til den oprindelige matrix, se billedet til højre i figur 10.3.

```

rng = np.random.default_rng()
a += 1.5 * rng.random(a.shape)

fig, ax = plt.subplots()
ax.matshow(a, cmap='coolwarm')

```

Vi beregner en tynd SVD og kikker på singularværdierne

```

u, s, vt = np.linalg.svd(a, full_matrices=False)
np.set_printoptions(linewidth = 60)
print(np.round(s, 3))

```

```

[18.59  4.166  3.32  3.077  2.606  2.296  1.859  1.525
 1.413  1.291  0.93  0.709  0.589  0.39 ]

```

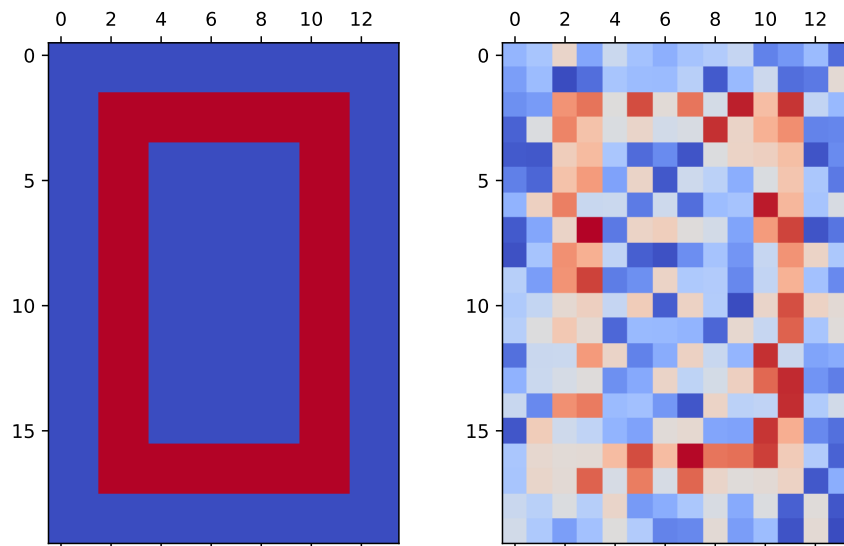
Vi ser at de første to til tre værdier er en del større end resten, så vi kan forvente bedste støjreduktion ved at bruge kun disse værdier. Lad os danne en funktion som angiver SVD for a forkortet til r led.

```

def svdapprox(u, s, vt, r):
    "Givet svd u, s, vt angiv forkortelse til niveau r."
    return u[:, :r] @ np.diag(s[:r]) @ vt[:r, :]

```

Vi kan nu plotte alle disse forkortede SVD



Figur 10.3: Billedet af cifret 0 samt en version med tilføjet støj.

```
fig, axs = plt.subplots(3, 4)
for r, (i,j) in enumerate(np.ndindex(axs.shape)):
    axs[i,j].matshow(svdapprox(u, s, vt, r), cmap='coolwarm')
plt.tight_layout() # justerer placering af hver subplot
```

Som kan ses i figur 10.4 giver den anden og tredje forkortede SVD gode gengivelse af det oprindelige billede

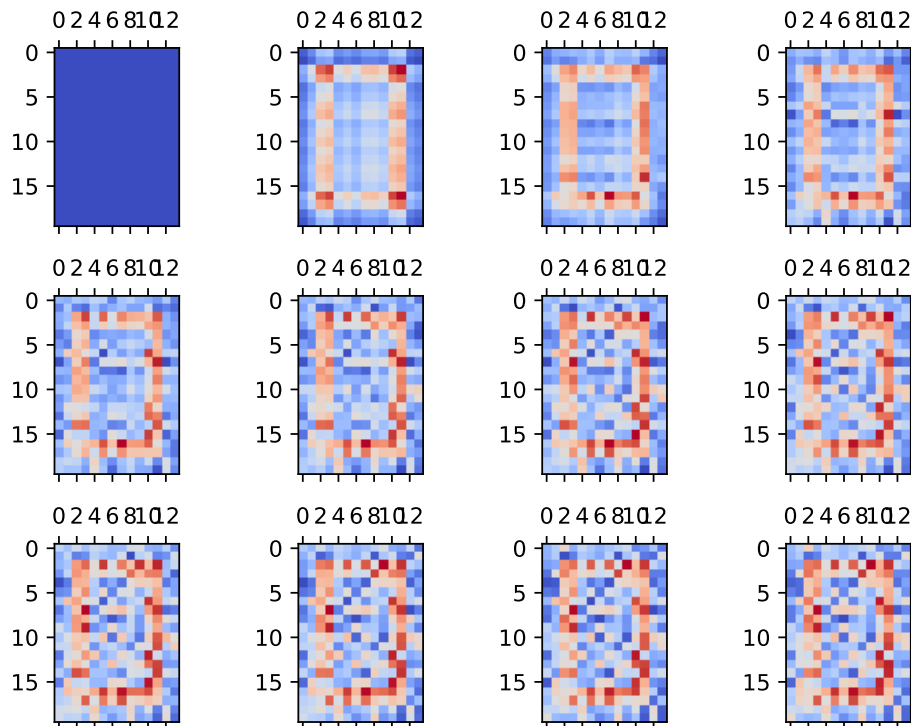
10.3.3 Komprimering af et billede

Lad os arbejde med et billede taget med mobiltelefonen, se figur 10.5, af domkirken i Uppsala. Billedfiler fra mobiltelefon er typisk i .jpg format, men man kan også arbejde med andre formatter, som .png.

Når vi har en .jpg kan billedfilen læses ind via `plt.imread` og vises via `imshow`:

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

10.3 EKSEMPLER PÅ SVD



Figur 10.4: SVD af billede med støj.

```
a = plt.imread('uppsala-img.jpg')  
  
fig, ax = plt.subplots()  
ax.imshow(a)
```

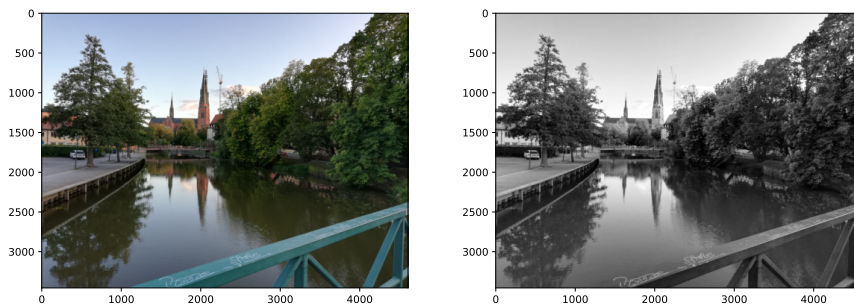
Dette giver en ndarray med shape

```
print(a.shape)
```

(3456, 4608, 3)

Den sidste indeks svarer til tal for rød, grøn og blå farver, i denne rækkefølge. (For en fil i .png format vil der også være et sidste tal for gennemsigtigheden.)

10 SINGULÆRVÆRDIDekomponering



Figur 10.5: Uppsala domkirke.

Billedet her har $3456 \times 4608 = 15925248$ pixels, dvs. $3456 \times 4608 / 2^{20} = 15.1875$ megapixels.

Typisk får vi nok med at kun kikke på den ene farve. Her vælger vi rød og får en almindelig matrix b

```
b = a[:, :, 0]
print(b.shape)
```

(3456, 4608)

som kan plottes i sort/hvid.

```
fig, ax = plt.subplots()
ax.imshow(b, cmap='Greys_r')
```

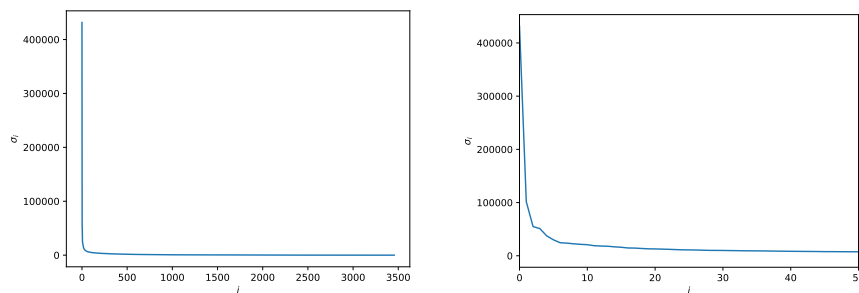
Lad os nu beregne den tynde SVD af b . Dette tager lidt tid, men mindre end 1 min på min maskine. Vi kan derefter plotte singulærværdierne, se figur 10.6.

```
u, s, vt = np.linalg.svd(b, full_matrices=False)

fig, ax = plt.subplots()
ax.set_xlabel(r'$i$')
ax.set_ylabel(r'$\sigma_{i}$')
ax.plot(s)
```

Vi ser at mange singulærværdier er meget tæt på 0. Vi kikker lidt nærmere på en plot af de første værdier.

10.3 EKSEMPLER PÅ SVD



Figur 10.6: Singulærværdier for billedet af domkirken.

```
fig, ax = plt.subplots()
ax.set_xlim(0, 50)
ax.set_xlabel(r'$i$')
ax.set_ylabel(r'$\sigma_{i}$')
ax.plot(s)
```

Til orientering kan vi se på nogle konkrete singulærværdier

```
print(np.round(s[[0, 2, 10, 50, 100]], 3))
```

```
[431654.686  54864.817  20826.709   7494.8    5049.833]
```

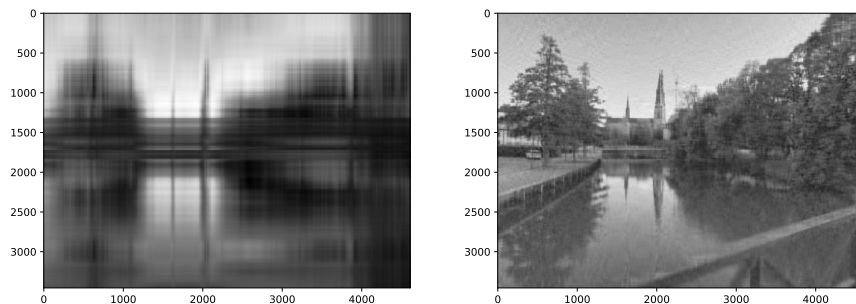
Vi kan forkorte SVD ved forskellige niveauer, og to af disse vises i figur 10.7.

```
def svdapprox(u, s, vt, r):
    "Givet svd u, s, vt angiv trunkering til niveau r."
    return u[:, :r] @ np.diag(s[:r]) @ vt[:r, :]
```

```
fig, ax = plt.subplots()
ax.imshow(svdapprox(u, s, vt, 3), cmap='Greys_r')
```

```
fig, ax = plt.subplots()
ax.imshow(svdapprox(u, s, vt, 50), cmap='Greys_r')
```

10 SINGULÆRVÆRDIDekomponering



Figur 10.7: SVD komprimeringer til hhv. 3 og 50 led.

En forkortning med kun 50 singulærværdier giver en god gengivelse af originalen. Dette er også en komprimering.

```
print('Bytes i a: ', a.nbytes)
print('Bytes i b: ', b.nbytes)
komprimeret_size = (u[:, :50].nbytes + s[:50].nbytes +
                    vt[:50, :].nbytes)
print('Bytes for det komprimerede billed: ',
      komprimeret_size)
print('Relativ størrelse af det komprimerede til b')
print('i procent: ', komprimeret_size / b.nbytes * 100)
```

```
Bytes i a: 47775744
Bytes i b: 15925248
Bytes for det komprimerede billed: 3226000
Relativ størrelse af det komprimerede til b
i procent: 20.257141364454732
```

Data for den forkortede SVD med 50 led fylder kun 2,5 % af størrelsen af det oprindelige billede

10.4 Eksistens af SVD

Bevis for sætning 10.1. Idet V er ortogonal er (10.5) det samme som

$$AV = U\Sigma.$$

Hvis V har søjler v_0, \dots, v_{n-1} , U har søjler u_0, \dots, u_{m-1} og Σ er som i (10.6), er denne ligning det samme som

$$[Av_0 \mid Av_1 \mid \dots] = [\sigma_0 u_0 \mid \sigma_1 u_1 \mid \dots]. \quad (10.8)$$

Så vores opgave er at finde ortonormale samlinger v_0, \dots, v_{n-1} og u_0, \dots, u_{m-1} samt skalarer $\sigma_0, \dots, \sigma_{k-1}$ således at (10.8) gælder.

Betragt funktionen

$$f_0(v) = \|Av\|_2^2, \quad \text{for } \|v\|_2 = 1.$$

Dette har en maksimumsværdi $s_0 = \|Av_0\|_2^2$ der opnås ved en enhedsvektor v_0 . Da $s_0 \geq 0$, kan vi skrive $\sigma_0 = \sqrt{s_0} = \|Av_0\|_2 \geq 0$. Hvis $\sigma_0 = 0$, har vi $A = 0$, og kan tage $\Sigma = 0$, $U = I_m$, $V = I_n$. Når $\sigma_0 > 0$ sætter vi $u_0 = Av_0/\sigma_0 = Av_0/\|Av_0\|_2$, som er en enhedsvektor. Så har vi

$$Av_0 = \sigma_0 u_0,$$

som er den ønskede ligning for den 0te søjle i (10.8).

Vi vil nu gerne gentage dette argument, men kun med vektorer ortogonale på v_0 . For at gøre dette har vi brug for den følgende observation:

v_0 er et maksimumspunkt for f_0 på enhedsvektorer medfører at $\langle u_0, Av \rangle = 0$ for alle v ortogonal på v_0 .

Dette kan ses på følgende måde. Det er nok at bekræfte udgagnet for v , der er en enhedsvektorer ortogonal på v_0 . Så er

$$w(t) = \cos(t)v_0 + \sin(t)v$$

en enhedsvektor for alle $t \in \mathbb{R}$. Funktionen $h(t) = f_0(w(t))$, opfylder at $h(0) = f_0(v_0) = s_0$, så $h(0) \geq h(t)$ for alle t , og specielt er $h'(0) = 0$. Men

$$\begin{aligned} h(t) &= f_0(w(t)) = \|\cos(t)Av_0 + \sin(t)Av\|_2^2 \\ &= \cos^2(t)\|Av_0\|_2^2 + 2\cos(t)\sin(t)\langle Av_0, Av \rangle + \sin^2(t)\|Av\|_2^2. \end{aligned}$$

Så

$$\begin{aligned} 0 &= h'(0) \\ &= (-2\cos(t)\sin(t)\|Av_0\|_2^2 + 2(\cos^2(t) - \sin^2(t))\langle Av_0, Av \rangle \\ &\quad + 2\sin(t)\cos(t)\|Av\|_2^2) \Big|_{t=0} \\ &= 2\langle Av_0, Av \rangle. \end{aligned}$$

Derfor er $\langle Av_0, Av \rangle = 0$ for alle v ortogonal på v_0 . Da u_0 er proportionelt med Av_0 , har vi så vist at

$$Av \perp u_0, \quad \text{for alle } v \perp v_0.$$

Vi kan nu gentage argumenterne ovenfor ved at kikke på maksimumsværdierne af

$$f_1(v) = \|Av\|_2^2, \quad \text{for } v \perp v_0 \text{ med } \|v\|_2 = 1,$$

og derefter

$$f_2(v) = \|Av\|_2^2, \quad \text{for } v \perp v_0, v_1 \text{ med } \|v\|_2 = 1,$$

osv. og stopper når f_r er identisk 0. Vi får $v_0, \dots, v_{r-1}, u_0, \dots, u_{r-1}$ og $\sigma_0 \geq \sigma_1 \geq \dots \geq \sigma_{r-1} > 0$ med

$$Av_j = \sigma_j u_j$$

samt

$\sigma_j^2 = \|Av_j\|_2^2$ er maksimalværdien af $\|Av\|_2^2$ over enhedsvektorer v ortogonal på v_0, \dots, v_{j-1}

for $j = 0, \dots, r-1$. Nu udvid til ortonormale baser $v_0, \dots, v_{n-1}, u_0, \dots, u_{m-1}$, ved hjælp af korollar 9.18, og sæt $\sigma_r = \dots = \sigma_{k-1} = 0$, til at få (10.8) og dermed (10.5), som ønsket. \square

Kapitel 11

Konditionstal

Vi har set at fejl kan opstå fra forskellige kilder: fejl i den oprindelige data, fejl ved repræsentation i computeren, fejl i beregningsoperationer. Konditionstal giver et praj om om hvor meget fejl sendes videre igennem beregninger og om hvor følsomt problemet er ovenfor fejl.

11.1 Konditionstal for differentiable funktioner

Betragt funktionen der beregner $f(x) = x^2$. Hvis x ændres til $x + h$, så ændres $f(x)$ til $f(x + h) = (x + h)^2 = x^2 + 2xh + h^2$. Ændringen i f er så $f(x + h) - f(x) = 2xh + h^2 = h(2x + h)$. Vi er interesseret i hvordan denne differens opfører sig for h tæt på 0. Man definerer det *absolutte konditionstal* til at være

$$\hat{\kappa}(x) = \lim_{h \rightarrow 0} \frac{|f(x + h) - f(x)|}{|h|} = |f'(x)|.$$

Så for vores funktion $f(x) = x^2$ er $\hat{\kappa}(x) = |2x| = 2|x|$. Dette siger at en lille ændring i x fører til en ændring i $f(x)$, som er $2|x|$ større.

Vi har set tidligere at især i forbindelse med *float* er det relevant at beregne relative fejl. Tilsvarende kan vi definere det relative konditionstal, eller blot *konditionstallet* $\kappa(x)$ til at være forholdet mellem de relative ændringer af $f(x)$ og x . Dette giver følgende definition for konditionstallet:

$$\begin{aligned} \kappa(x) &= \lim_{h \rightarrow 0} \left(\frac{|f(x + h) - f(x)|}{|f(x)|} \bigg/ \frac{|h|}{|x|} \right) = \lim_{h \rightarrow 0} \left(\frac{|f(x + h) - f(x)|}{|h|} \frac{|x|}{|f(x)|} \right) \\ &= \frac{|x||f'(x)|}{|f(x)|}, \end{aligned}$$

11 KONDITIONSTAL

forudsat at $f(x) \neq 0 \neq x$.

For funktionen $f(x) = x^2$ har vi $\kappa(x) = |x| |2x| / |x^2| = 2$. Dvs. en lille relative fejl i x afspejles i en relativ fejl, der er dobbelt stort, i $x^2 = f(x)$.

Eksempel 11.1. $f(x) = \sqrt{x}$, $x > 0$, har konditionstal

$$\kappa(x) = |x| \left| \frac{d}{dx} \sqrt{x} \right| \bigg/ |\sqrt{x}| = |x| \left| \frac{1}{2\sqrt{x}} \right| \bigg/ |\sqrt{x}| = \left| \frac{\sqrt{x}}{2} \right| \bigg/ |\sqrt{x}| = \frac{1}{2}.$$

△

Eksempel 11.2. $f(x) = e^x$ har konditionstal

$$\kappa(x) = |x| \left| \frac{d}{dx} e^x \right| \bigg/ |e^x| = |x| e^x / e^x = |x|.$$

Så beregning af e^x for x stort kan forventes at have en stor relativ fejl. △

Konditionstal af størrelsesorden 1 eller 10 er til at tåle; vi mister få betydende cifre i beregning af svaret. Omvendt betyder et konditionstal af størrelsesorden 10^6 , at en relativ fejl i x forstørres voldsomt, og dette er ofte uacceptabel.

Når vi arbejder med vektorer, skal vi justere lidt på de overnævnte definitioner. Først skal vi erstatte numerisk værdi med normen af vektoren. For $f: \mathbb{R}^n \rightarrow \mathbb{R}$ erstattes den afledede med gradienten

$$(\nabla f)_x = \left(\frac{\partial f}{\partial x_0}, \frac{\partial f}{\partial x_1}, \dots, \frac{\partial f}{\partial x_{n-1}} \right).$$

For en differentiabel $f: \mathbb{R}^n \rightarrow \mathbb{R}$, defineres *konditionstallet* i x til at være

$$\kappa(x) = \frac{\|x\|_2 \|(\nabla f)_x\|_2}{|f(x)|}.$$

Eksempel 11.3. Betragt funktionen $f: \mathbb{R}^2 \rightarrow \mathbb{R}$, som beregner differensen,

$$f(x_0, x_1) = x_1 - x_0$$

af tallene x_0 og x_1 . Gradienten af f er

$$(\nabla f)_x = \left(\frac{\partial(x_1 - x_0)}{\partial x_0}, \frac{\partial(x_1 - x_0)}{\partial x_1} \right) = (-1, 1).$$

11.1 KONDITIONSTAL FOR DIFFERENTIABLE FUNKTIONER

Konditionstallet for f beregnes til at være

$$\kappa(x) = \frac{\|x\|_2 \|(-1, 1)\|_2}{|x_1 - x_0|} = \sqrt{2} \frac{\sqrt{x_0^2 + x_1^2}}{|x_1 - x_0|}. \quad (11.1)$$

Vi ser at konditionstallet er stort for x_0 tæt på x_1 . Dette understøtter det vi har tidligere set (f.eks. eksempel 1.4) at differenceoperationen kan føre til stor fejl når tallene er tæt på hinanden.

Lad os kikke lidt nærmere på dette eksempel, for at understøtte definitionen på $\kappa(x)$.

Hvis (x_0, x_1) ændres med en vektor (h_0, h_1) , er ændringen i f :

$$\begin{aligned} f(x_0 + h_0, x_1 + h_1) - f(x_0, x_1) &= ((x_1 + h_1) - (x_0 + h_0)) - (x_1 - x_0) \\ &= h_1 - h_0. \end{aligned}$$

Vi kan skrive det sidste som et indre produkt

$$f(x_0 + h_0, x_1 + h_1) - f(x_0, x_1) = h_1 - h_0 = \left\langle \begin{bmatrix} -1 \\ 1 \end{bmatrix}, \begin{bmatrix} h_0 \\ h_1 \end{bmatrix} \right\rangle.$$

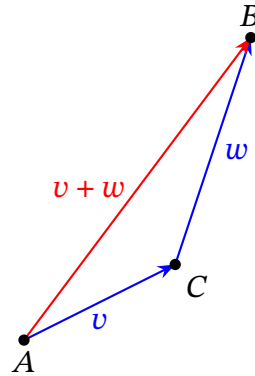
Bemærk at vektoren $(-1, 1)$ er $(\nabla f)_x$. Nu giver Cauchy-Schwarz uligheden at den absolutte fejl i f ved x er

$$\begin{aligned} |f(x_0 + h_0, x_1 + h_1) - f(x_0, x_1)| \\ = \left| \left\langle \begin{bmatrix} -1 \\ 1 \end{bmatrix}, \begin{bmatrix} h_0 \\ h_1 \end{bmatrix} \right\rangle \right| &\leq \left\| \begin{bmatrix} -1 \\ 1 \end{bmatrix} \right\|_2 \left\| \begin{bmatrix} h_0 \\ h_1 \end{bmatrix} \right\|_2 = \sqrt{2} \|h\|_2. \end{aligned}$$

Bemærk at Cauchy-Schwarz uligheden sikrer også at der er lighed for visse h : nemlig de h , som er parallel med $(-1, 1)$. Så den største mulige fejl er givet ved den højre side $\sqrt{2}\|h\|_2$. Beregner vi forholdet mellem den relative fejl i f og den relative ændring i x , får vi

$$\frac{|f(x+h) - f(x)|}{|f(x)|} \bigg/ \frac{\|h\|_2}{\|x\|_2} = \frac{|f(x+h) - f(x)|}{\|h\|_2} \frac{\|x\|_2}{|f(x)|} \leq \sqrt{2} \frac{\sqrt{x_0^2 + x_1^2}}{|x_1 - x_0|},$$

og denne grænse opnås for visse h . Denne maksimale værdi er netop konditionstallet $\kappa(x)$ i (11.1). \triangle



Figur 11.1: Trekantsuligheden.

11.2 Matrixnorm

Vi ønsker at udforske tilsvarende ændringer i relation til lineære ligningssystemer $Ax = b$. Det kræver dog at vi har et norm-begreb for matricen A .

Definition 11.4. En *norm* på \mathbb{R}^n er en funktion $\|\cdot\| : \mathbb{R}^n \rightarrow \mathbb{R}$ med

- (N1) $\|v\| \geq 0$ for alle $v \in \mathbb{R}^n$, og $\|v\| = 0$ kun for $v = 0$,
- (N2) $\|sv\| = |s|\|v\|$ for alle $s \in \mathbb{R}$,
- (N3) $\|v + w\| \leq \|v\| + \|w\|$.

Vi har set at 2-normen $\|\cdot\|_2 = \sqrt{\langle \cdot, \cdot \rangle}$ opfylder (N1) og (N2). Betingelse (N3) kaldes *trekantsuligheden*, da den siger at i figur 11.1 er vejen direkte fra A til B kortere end turen langs de andre to sider af trekanten via punktet C .

Trekantsuligheden gælder for $\|\cdot\|_2$, da

$$\begin{aligned}
 \|v + w\|_2^2 &= \langle v + w, v + w \rangle = \langle v, v \rangle + \langle v, w \rangle + \langle w, v \rangle + \langle w, w \rangle \\
 &= \|v\|_2^2 + 2\langle v, w \rangle + \|w\|_2^2 \\
 &\leq \|v\|_2^2 + 2\|v\|_2\|w\|_2 + \|w\|_2^2 \\
 &= (\|v\|_2 + \|w\|_2)^2,
 \end{aligned}$$

hvor vi har brugt Cauchy-Schwarz uligheden (8.4).

Der findes andre normer på \mathbb{R}^n . To første eksempler er *1-normen*

$$\|v\|_1 = \|(v_0, v_1, \dots, v_{n-1})\|_1 = |v_0| + |v_1| + \dots + |v_{n-1}|$$

og maksimumsnormen eller ∞ -normen

$$\|v\|_{\infty} = \|(v_0, v_1, \dots, v_{n-1})\|_{\infty} = \max\{|v_0|, |v_1|, \dots, |v_{n-1}|\}.$$

Numerisk kan det være hurtige at beregne $\|v\|_1$ eller $\|v\|_{\infty}$ end $\|v\|_2$. Desuden indeholder disse normer sammenlignelig oplysning om v , da for $v \in \mathbb{R}^n$ gælder

$$\|v\|_{\infty} \leq \|v\|_2 \leq \|v\|_1 \leq \sqrt{n}\|v\|_2 \leq n\|v\|_{\infty}.$$

Som navnene antyder er de overnævnte normer en del af en større familie. For hvert tal p med $1 \leq p \leq \infty$ er p -normen givet ved

$$\|v\|_p = \|(v_0, v_1, \dots, v_{n-1})\|_p = \left(\sum_{i=0}^{n-1} |v_i|^p \right)^{1/p}$$

For matricer $A \in \mathbb{R}^{m \times n}$ indfører vi en matrix eller operator 2 -norm på følgende måde

$$\|A\|_2 = \max_{\|x\|_2=1} \|Ax\|_2. \quad (11.2)$$

Dvs. $\|A\|_2$ måler den største skalering af længden af x når vi ganger med A . At (11.2) definerer en norm, følger direkte fra egenskaberne af $\|\cdot\|_2$ for vektorer. Ligning (11.2) er det samme som

$$\|A\|_2 = \max_{x \neq 0} \frac{\|Ax\|_2}{\|x\|_2}, \quad (11.3)$$

som ofte gives som definition. Dette ækvivalens gælder, da

$$\frac{\|Ax\|_2}{\|x\|_2} = \frac{1}{\|x\|_2} \|Ax\|_2 = \left\| \frac{1}{\|x\|_2} Ax \right\|_2 = \left\| A \left(\frac{1}{\|x\|_2} x \right) \right\|_2 = \left\| A \left(\frac{x}{\|x\|_2} \right) \right\|_2$$

og $x/\|x\|_2$ er enhedsvektor. Omskrivningen (11.3) giver den nyttige

$$\|Ax\|_2 \leq \|A\|_2 \|x\|_2, \quad \text{for alle } x \in \mathbb{R}^n. \quad (11.4)$$

Man kan definere en tilsvarende p -norm for A ved $\|A\|_p = \max_{\|x\|_p=1} \|Ax\|_p$, men vi vil holdes os mest til $\|A\|_2$.

Lemma 11.5. For $A \in \mathbb{R}^{m \times n}$, $B \in \mathbb{R}^{n \times p}$ gælder

$$\|AB\|_2 \leq \|A\|_2 \|B\|_2.$$

11 KONDITIONSTAL

Bevis. Det følger fra (11.4) at

$$\|ABx\|_2 = \|A(Bx)\|_2 \leq \|A\|_2 \|Bx\|_2 \leq \|A\|_2 \|B\|_2 \|x\|_2.$$

Så for $\|x\|_2 = 1$, gælder

$$\|ABx\|_2 \leq \|A\|_2 \|B\|_2.$$

Det følger at maksimumsværdien for $\|ABx\|_2$ over $\|x\|_2 = 1$ er højst $\|A\|_2 \|B\|_2$, men det er netop udsagnet $\|AB\|_2 \leq \|A\|_2 \|B\|_2$, der ønskes. \square

Dette 2-norm for A er faktisk noget vi har set før:

Lemma 11.6. Hvis $A \in \mathbb{R}^{m \times n}$ har singulærværdidekomponering $A = U\Sigma V^T$, med singulærværdier $\sigma_0 \geq \dots \geq \sigma_{k-1} \geq 0$, $k = \min\{m, n\}$, så er 2-normen af A givet ved

$$\|A\|_2 = \sigma_0,$$

hvor σ_0 er den største singulærværdi.

Bevis. Matricen $U \in \mathbb{R}^{m \times m}$ er ortogonal, så $\|Uy\|_2 = \|y\|_2$ for alle $y \in \mathbb{R}^m$, se proposition 9.5. Vi har så

$$\|Ax\|_2 = \|U\Sigma V^T x\|_2 = \|\Sigma V^T x\|_2. \quad (11.5)$$

For at bestemme $\|A\|_2$ skal vi så

$$\text{maksimere (11.5) over alle } x \text{ med } \|x\|_2 = 1. \quad (11.6)$$

Men V^T er også ortogonal, så bevarer $\|\cdot\|_2$ og er invertibel. Ved at skrive $w = V^T x$, får vi alle w med $\|w\|_2 = 1$, fra de forskellige x med $\|x\|_2 = 1$. Det giver at (11.6) er det sammen som at

$$\text{maksimere } \|\Sigma w\|_2 \text{ over alle } w \text{ med } \|w\|_2 = 1. \quad (11.7)$$

Bemærk nu at for $w = (w_0, \dots, w_{n-1})$ er

$$\begin{aligned} \|\Sigma w\|_2 &= \sqrt{\sigma_0^2 w_0^2 + \dots + \sigma_{k-1}^2 w_{k-1}^2} \\ &\leq \sqrt{\sigma_0^2 (w_0^2 + \dots + w_{n-1}^2)} = \sigma_0 \|w\|_2 \end{aligned}$$

og vi har lighed for $w = e_0$. Det følger at

$$\|A\|_2 = \max_{\|x\|_2=1} \|Ax\|_2 = \max_{\|w\|_2=1} \|\Sigma w\|_2 = \sigma_0,$$

som ønsket. \square

11.3 Konditionstal og lineære ligningssystemer

Lad os nu betragte et lineært ligningssystem med samme antal ligninger, som variable:

$$Ax = b$$

for $x, b \in \mathbb{R}^n$, $A \in \mathbb{R}^{n \times n}$. Lad os desuden antage at den kvadratiske matrix A er invertibel, så for hvert b er der en entydig løsning $x = A^{-1}b$. Der er flere forskellige måder at beregne et konditionstal for systemet, med det viser sig at alle kontrolleres af [konditionstallet for \$A\$](#)

$$\kappa(A) = \|A\|_2 \|A^{-1}\|_2.$$

Proposition 11.7. *Konditionstallet for en invertibel matrix $A \in \mathbb{R}^{n \times n}$, er givet ved*

$$\kappa(A) = \sigma_0 / \sigma_{n-1} \quad (11.8)$$

hvor $\sigma_0 \geq \dots \geq \sigma_{n-1} > 0$ er singularværdierne for A .

Bevis. Lemma 11.6 viser netop at $\|A\|_2 = \sigma_0$. Skriver vi A 's singularværdidekomponering, som $A = U\Sigma V^T$, har vi for A invertibel at $A^{-1} = V\Sigma^{-1}U^T$. Men

$$\Sigma^{-1} = \begin{bmatrix} \sigma_0 & 0 & \dots & 0 \\ 0 & \sigma_1 & & 0 \\ \vdots & & \ddots & \\ 0 & 0 & & \sigma_{n-1} \end{bmatrix}^{-1} = \begin{bmatrix} 1/\sigma_0 & 0 & \dots & 0 \\ 0 & 1/\sigma_1 & & 0 \\ \vdots & & \ddots & \\ 0 & 0 & & 1/\sigma_{n-1} \end{bmatrix}.$$

Det følger at A^{-1} har singularværdier $1/\sigma_{n-1} \geq \dots \geq 1/\sigma_0$, og den singularværdidekomponering af A^{-1} er

$$A^{-1} = [v_{n-1} \mid \dots \mid v_1 \mid v_0] \begin{bmatrix} 1/\sigma_{n-1} & 0 & \dots & 0 \\ & \ddots & & \vdots \\ 0 & & 1/\sigma_1 & 0 \\ 0 & \dots & 0 & 1/\sigma_1 \end{bmatrix} \begin{bmatrix} u_{n-1}^T \\ \vdots \\ u_1^T \\ u_0^T \end{bmatrix}$$

hvor u_i og v_j er søjlerne af U hhv. V . Vi har så at $\|A^{-1}\|_2 = 1/\sigma_{n-1}$, og resultatet følger. \square

11 KONDITIONSTAL

Lad os først betragte funktionen $f: \mathbb{R}^n \rightarrow \mathbb{R}^n$

$$f(x) = Ax,$$

for en given invertibel matrix A . Ændrer vi x til $x+h$, ændres $f(x)$ til $f(x+h) = A(x+h) = Ax + Ah$. Den relative ændring i f i forhold til den for x har størrelse

$$\begin{aligned} \frac{\|f(x+h) - f(x)\|_2}{\|f(x)\|_2} & \bigg/ \frac{\|h\|_2}{\|x\|_2} = \frac{\|Ah\|_2}{\|Ax\|_2} \bigg/ \frac{\|h\|_2}{\|x\|_2} \\ & = \frac{\|Ah\|_2}{\|h\|_2} \frac{\|x\|_2}{\|Ax\|_2}. \end{aligned}$$

Dens maksimal værdi er så

$$\kappa(x) = \|A\|_2 \frac{\|x\|_2}{\|Ax\|_2},$$

som opnås for h parallelt med v_0 . Når vi lader x variere, kan vi skrive $b = Ax$, så $x = A^{-1}b$, og har

$$\frac{\|x\|_2}{\|Ax\|_2} = \frac{\|A^{-1}b\|_2}{\|b\|_2}$$

som har maksimumsværdi $\|A^{-1}\|_2$. Det følger at

$$\max_{x \neq 0} \kappa(x) = \|A\|_2 \|A^{-1}\|_2 = \kappa(A), \quad (11.9)$$

med maksimum opnået for x parallelt med v_{n-1} .

Eksempel 11.8. Betragt ligningssystemet

$$\begin{aligned} 7x_0 + 3x_1 &= 5, \\ 9x_0 + 4x_1 &= -1. \end{aligned}$$

Den tilhørende matrix er

$$A = \begin{bmatrix} 7 & 3 \\ 9 & 4 \end{bmatrix},$$

hvis konditionstal kan beregnes i python:

```
>>> import numpy as np
>>> a = np.array([[7.0, 3.0],
...               [9.0, 4.0]])
>>> np.linalg.cond(a)
154.99354811853624
```

11.3 KONDITIONSTAL OG LINEÆRE LIGNINGSSYSTEMER

Tages x til at være vektoren v_1 fra SVD for A , og h til at være proportionelt med v_0 , ser vi at ændring i Ax til $A(x + h)$ er ret stort

```
>>> u, s, vt = np.linalg.svd(a)
>>> s
array([12.44964048,  0.0803236 ])
>>> x = vt[[1], :].T
>>> x
array([[ -0.40157455],
       [ 0.91582634]])
>>> a @ x
array([[ -0.06354287],
       [ 0.04913435]])
>>> h = 0.1 * vt[[0], :].T
>>> h
array([[ -0.09158263],
       [ -0.04015746]])
>>> a @ (x+h)
array([[ -0.82509367],
       [ -0.93573917]])
>>> ((np.linalg.norm(a @ h)/np.linalg.norm(h))
...  * (np.linalg.norm(x)/np.linalg.norm(a @ x)))
154.99354811853738
```

Dette er det størst muligt relativ fejl. Vælges x og h tilfældigt er fejlen typisk mindre

```
>>> rng = np.random.default_rng()
>>> x = rng.standard_normal((a.shape[1],1))
>>> a @ x
array([[ -8.01969767],
       [-10.25098793]])
>>> h = rng.normal(0.0, 0.1, (a.shape[1],1))
>>> a @ (x+h)
array([[ -8.49614759],
       [-10.86571357]])
>>> a @ (x+h)
array([[ -8.49614759],
```

11 KONDITIONSTAL

```
[-10.86571357]])  
>>> ((np.linalg.norm(a @ h)/np.linalg.norm(h))  
...   * (np.linalg.norm(x)/np.linalg.norm(a @ x)))  
1.3103327816675692
```

Faktisk er fejlen mindst for x parallelt med v_0 og h parallelt med $v_1 = v_{n-1}$. Δ

Hvis vi er interesseret i hvordan løsninger x til $Ax = b$ ændrer sig når b varierer, kan vi bruge den samme analyse, men for funktionen $b \mapsto x = A^{-1}b$. Dvs. A udskiftes med A^{-1} ovenfor. Men så er resultatet det samme udtryk (11.9); men rollerne af v_0 og v_{n-1} spilles nu af u_{n-1} og u_0 .

Desuden hvis man holder b fast, og lad A varierer, dvs. betragter funktionen $A \mapsto x = A^{-1}b$, kan man også vise at ændringerne styres også af (11.9).

For en generel matrix $A \in \mathbb{R}^{m \times n}$ kan man godt definere konditionstallet til at være $\kappa(A) = \sigma_0/\sigma_{k-1} \in [0, \infty]$, jævnfør (11.8). Nogle af de overnævnte resultater kan så overføres til denne situation.

Kapitel 12

Generelle vektorrum

12.1 Vektorrum

Indtil videre har vi kun betragtet vektorer som elementer i \mathbb{R}^n . Men i det forrige kapitel, så vi at matricer i $\mathbb{R}^{m \times n}$ kan også håndteres på en lignende måde. Både \mathbb{R}^n og $\mathbb{R}^{m \times n}$ er eksempler på vektorrum, men der er andre objekter der kan med fordel behandles på en tilsvarende måde.

Definition 12.1. Et *vektorrum* består af en mængde V , hvis elementer vi kalder *vektorer*. Der er to operationer defineret. Den første er en sum operation $u + v \in V$ for alle $u, v \in V$, som opfylder

- (V1) $u + v = v + u$, for alle $u, v \in V$,
- (V2) $(u + v) + w = u + (v + w)$, for alle $u, v, w \in V$,
- (V3) der findes $0 \in V$ således at $v + 0 = v$ for alle $v \in V$,
- (V4) for hver $v \in V$ findes der $-v \in V$ således at $v + (-v) = 0$.

Den anden operation er multiplikation med en skalar $sv \in V$ defineret for alle skalar s og alle $v \in V$, med følgende egenskaber

- (V5) $s(u + v) = su + sv$,
- (V6) $(s + t)v = sv + tv$,
- (V7) $(st)v = s(tv)$,
- (V8) $1v = v$,

for alle skalarer s og t , og alle $u, v \in V$.

Eksempel 12.2. $V = \mathbb{R}^n = \mathbb{R}^{n \times 1}$ med vektorsum og skalar multiplikation som vi har brugt tidligere opfylder dette. Den eneste der skal bemærkes er elementet 0

i del (V3) er vektoren

$$0 = 0_n = \begin{bmatrix} 0 \\ 0 \\ \vdots \\ 0 \end{bmatrix} \in \mathbb{R}^n.$$

△

Eksempel 12.3. Rummet af matricer $V = \mathbb{R}^{m \times n}$ er ligeledes et vektorrum. Denne gang er nulelementet matricen $0_{m \times n}$. Bemærk at rummet af matricer har yderligere matrixmultiplikation, som en ekstra operation ud over de to operationer i definition 12.1. △

Eksempel 12.4. Lad V bestå af alle polynomier af grad højst 2. Et typisk element i V er

$$p(x) = a_0 + a_1x + a_2x^2$$

med $a_0, a_1, a_2 \in \mathbb{R}$. Vi kan definere en sum operationer for $p(x)$ og $q(x) = b_0 + b_1x + b_2x^2$ ved

$$(p + q)(x) = (a_0 + b_0) + (a_1 + b_1)x + (a_2 + b_2)x^2.$$

Så kan man tjekke at dele (V1) til (V4) er opfyldt med nulpolynomiet givet ved

$$0(x) = 0 + 0x + 0x^2.$$

Med skalarmultiplikation givet ved

$$(sp)(x) = sa_0 + (sa_1)x + (sa_2)x^2,$$

er de resterende punkter i definition 12.1 opfyldt, og vi får et vektorrum. △

Eksempel 12.5. Eksempel 12.4 kan udvides til polynomier af grad højst n , eller til alle polynomier, på tilsvarende måde. △

Lemma 12.6. *Lad V være et vektorrum og lad $v \in V$.*

(a) *Så kan nulelementet beregnes som $0 = 0v$.*

(b) *Elementet $-v$ er givet som $-v = (-1)v$.*

Begge elementer er entydigt bestemt.

Bevis. Først er 0 entydig: hvis $v + u = v$, så har vi $0 = v - v = (v + u) - v = u$, dvs. $u = 0$. Nu gælder $v + 0v = 1v + 0v = (1 + 0)v = 1v = v$, så $0v = 0$.

Elementet $-v$ er entydig, da $v + w = 0$ giver $-v = -v + 0 = -v + (v + w) = (-v + v) + w = 0 + w = w$, så $w = -v$. Vi beregner nu $v + (-1)v = 1v + (-1)v = (1 - 1)v = 0v = 0$, så $(-1)v = -v$. □

Eksempel 12.7. Lad V bestå af alle differentiable funktioner $f: [a, b] \rightarrow \mathbb{R}$. Vi sætter $f + g$ til at være funktionen givet ved

$$(f + g)(x) = f(x) + g(x), \quad \text{for alle } x \in [a, b],$$

og definerer skalarmultiplikation ved

$$(sf)(x) = s(f(x)), \quad \text{for alle } x \in [a, b].$$

Bemærk at både $f + g$ og sf er differentiable funktioner. Vi får et vektorrum, med nulvektor funktionen $0(x) = 0$, for alle $x \in [a, b]$. Δ

12.2 Andre skalarer

Andre eksempler på vektorrum gives ved at bruge andre typer skalarer. Indtil videre har vores skalarer været reelle tal $s \in \mathbb{R}$. Men vektorrum kan dannes med andre koefficienter. Tænkes der på hvordan vi har løst lineære ligningssystemer, er det væsentlig at vi kan tage sum og differens af skalarer og at vi kan gange og dele med dem. Vigtigst for os vil være at vi kan bruge komplekse tal.

De *komplekse tal* betegnes \mathbb{C} . Et element $z \in \mathbb{C}$ skrives

$$z = x + iy$$

hvor $x, y \in \mathbb{R}$ og i er et symbol, med $i^2 = -1$. Vi kalder $x = \operatorname{Re}(z)$ den *reelle* del af z , og $y = \operatorname{Im}(z)$ den *imaginære* del. Beregning med komplekse tal udføres netop som for reelle tal blot med den ekstra relation at $i^2 = -1$.

Eksempel 12.8. For $z = 1 + 2i$, $w = -2 + 3i$ har vi

$$z + w = (1 + 2i) + (-2 + 3i) = 1 + (-2) + i(2 + 3) = -1 + 5i$$

og

$$\begin{aligned} zw &= (1 + 2i)(-2 + 3i) = 1 \times (-2) + 1 \times 3i + 2i \times (-2) + (2i) \times (3i) \\ &= -2 + 3i - 4i + 6i^2 = -2 - i - 6 = -8 - i. \end{aligned}$$

Δ

For $z = x + iy$, $w = s + it$ får vi så

$$z + w = (x + iy) + (s + it) = (x + s) + i(y + t), \quad (12.1)$$

$$zw = (x + iy)(s + it) = (xs - yt) + i(xt + ys). \quad (12.2)$$

12 GENERELLE VEKTORRUM

(12.1) er bare vektorsum i \mathbb{R}^2 oversat til denne ny notation. (12.2) kommer fra at gange $(x + iy)(s + it)$ ud på almindeligvis samt reglen $i^2 = -1$ som i eksemplet.

Vi kan godt regne med komplekse tal i python (her er python en del bedre end andre programmeringssprog). Vi skal dog gå opmærksom på at python bruger j for tallet i , i tråd med god traditioner for elektriske ingeniører:

```
>>> import numpy as np
>>> z = 1.0 + 2.0j
>>> type(z)
<class 'complex'>
>>> np.real(z)
1.0
>>> np.imag(z)
2.0
>>> w = -2.0 + 3.0j
>>> z + w
(-1+5j)
>>> z * w
(-8-1j)
```

Man skal være opmærksom på at der skal stå et tal direkte foran j , dvs. uden mellemrum og uden $*$.

Man kan også dele med komplekse tal, som er forskellig fra $0 = 0 + 0i$. Opskriften er

$$\frac{1}{z} = \frac{1}{x + iy} = \frac{x - iy}{x^2 + y^2}. \quad (12.3)$$

Ved direkte udregning kan man tjekke at $z(1/z) = 1$:

$$\begin{aligned} z \frac{1}{z} &= (x + iy) \frac{x - iy}{x^2 + y^2} = \frac{1}{x^2 + y^2} (x + iy)(x - iy) \\ &= \frac{1}{x^2 + y^2} (x^2 - ixy + ixy - i^2 y^2) = \frac{x^2 + y^2}{x^2 + y^2} = 1. \end{aligned}$$

Eksempel 12.9. For $z = 1 + 2i$ har vi

$$\frac{1}{z} = \frac{1}{1 + 2i} = \frac{1 - 2i}{1^2 + 2^2} = \frac{1 - 2i}{5} = \frac{1}{5} - \frac{2}{5}i,$$

som python kan bekræfte

```

>>> z = 1.0 + 2.0j
>>> 1/z
(0.2-0.4j)
>>> z * (1/z)
(1+0j)

```

△

Vi kan også arbejde med lineære ligningssystemer hvis koefficienter er komplekse tal på præcis den samme måde med rækkeoperationer, som vi gjorde tidligere.

Eksempel 12.10. Systemet

$$\begin{aligned} 2x + 3y - 4z &= 7 - i, \\ 3ix - (4 + i)y + z &= -2, \\ x + y + (2 + i)z &= 3 + i \end{aligned}$$

kan løses via rækkeoperationer. F.eks. i python

```

>>> import numpy as np
>>> a = np.array([[2.0, 3.0, -4.0],
...              [3.0j, -(4.0+1.0j), 1.0],
...              [1.0, 1.0, 2.0+1.0j]])
>>> a.dtype
dtype('complex128')
>>> b = np.array([7.0-1.0j, -2.0, 3.0+1.0j])[ :, np.newaxis]
>>> aub = np.hstack([a, b])
>>> aub
array([[ 2.+0.j,  3.+0.j, -4.+0.j,  7.-1.j],
       [ 0.+3.j, -4.-1.j,  1.+0.j, -2.+0.j],
       [ 1.+0.j,  1.+0.j,  2.+1.j,  3.+1.j]])
>>> aub[ [0,2], :] = aub[ [2,0], :]
>>> aub[1, :] += -3.0j * aub[0, :]
>>> aub[2, :] += -2.0 * aub[0, :]
>>> aub
array([[ 1.+0.j,  1.+0.j,  2.+1.j,  3.+1.j],
       [ 0.+0.j, -4.-4.j,  4.-6.j,  1.-9.j],
       [ 1.+0.j,  1.+0.j,  2.+1.j,  3.+1.j]])

```

12 GENERELLE VEKTORRUM

```
[ 0.+0.j, 1.+0.j, -8.-2.j, 1.-3.j]])
>>> aub[ [1,2], :] = aub[ [2,1], :]
>>> aub[0, :] += - aub[1, :]
>>> aub[2, :] += (4.0 + 4.0j) * aub[1, :]
>>> aub
array([[ 1. +0.j,  0. +0.j, 10. +3.j,  2. +4.j],
       [ 0. +0.j,  1. +0.j, -8. -2.j,  1. -3.j],
       [ 0. +0.j,  0. +0.j, -20.-46.j, 17.-17.j]])
>>> aub[2, :] *= 1/(-20.0-46.0j)
>>> aub[0, :] += -(10.0+3.0j) * aub[2, :]
>>> aub[1, :] += (8.0+2.0j) * aub[2, :]
>>> aub
array([[ 1.          +0.j          ,  0.          +0.j          ,
        0.          +0.j          ,  1.58108108-0.98648649j],
       [ 0.          +0.j          ,  1.          +0.j          ,
        0.          +0.j          ,  1.51351351+0.91891892j],
       [-0.          +0.j          , -0.          +0.j          ,
        1.          +0.j          ,  0.17567568+0.44594595j]])
>>> print('x = ', aub[0, -1])
x = (1.581081081081081-0.9864864864864868j)
>>> print('y = ', aub[1, -1])
y = (1.5135135135135136+0.9189189189189189j)
>>> print('z = ', aub[2, -1])
z = (0.17567567567567569+0.44594594594594594j)
>>> # Tjek af løsning
>>> v = aub[:, [-1]]
>>> a @ v - b
array([[ -8.88178420e-16-8.88178420e-16j],
       [ 1.11022302e-15-5.55111512e-16j],
       [ 0.00000000e+00-4.44089210e-16j]])
>>> np.allclose(a @ v, b, atol = np.finfo(float).eps)
True
```

Δ

Formlen (12.3) indeholder to væsentlig operationer på $z = x + iy$. Den første er den *konjugerede*

$$\bar{z} = \overline{x + iy} = x - iy.$$

```
>>> z = 1.0 + 2.0j
>>> np.conj(z)
(1-2j)
```

Den anden den *numeriske værdi*

$$|z| = |x + iy| = \sqrt{x^2 + y^2},$$

som er blot normen af vektoren (x, y) i \mathbb{R}^2 .

```
>>> import numpy as np
>>> z = 1.0 + 2.0j
>>> np.abs(z)
2.23606797749979
>>> np.sqrt(np.real(z)**2+np.imag(z)**2)
2.23606797749979
```

Vi kan så omskrive formlen (12.3), som

$$\frac{1}{z} = \frac{\bar{z}}{|z|^2}$$

Bemærk at $|z|^2 = x^2 + y^2 = (x + iy)(x - iy) = z\bar{z}$. Vi har også

$$|zw| = |z||w|$$

for alle komplekse tal z og w . Konjugation opfylder

$$\overline{\bar{z} + \bar{w}} = z + w, \quad \overline{zw} = \bar{z}\bar{w}.$$

12.3 Underrum

En nem og væsentlig kilde til vektorrum er visse delmængde af kendte vektorrum.

Definition 12.11. Lad V være et vektorrum. En ikke-tom delmængde $W \subset V$ er et *underrum* hvis sum og skalarmultiplikation sender element af W til elementer W . Dvs.

12 GENERELLE VEKTORRUM

- (a) W er ikke tom,
- (b) $v, w \in W$ medfører $v + w \in W$,
- (c) $w \in W$ og s en skalar medfører $sw \in W$.

Bemærk at nulvektoren 0 må nødvendigvis ligge i W , da for et $w \in W$ har vi $0 = 0w \in W$. Det kan godt tjekkes at et hvert underrum opfylder kravene i definition 12.1, så selv et vektorrum.

Eksempel 12.12. Givet $v \in \mathbb{R}^n$ med $v \neq 0$, er linjen L igennem v og origo, mængden

$$L = \{tv : t \in \mathbb{R}\}.$$

Dette mængde indeholder v selv, så er ikke tom. To elementer i L har formen t_0v og t_1v , så deres sum er $(t_0v) + (t_1v) = (t_0 + t_1)v$, som ligger i L . Til sidst er $s(tv) = (st)v$ i L . Så L er et underrum af \mathbb{R}^n , og dermed et vektorrum. Δ

Dette eksempel kan udvides til planer osv. på den følgende måde.

Definition 12.13. Rummet *udspændt* af vektorer $v_0, v_1, \dots, v_{k-1} \in V$ er

$$\begin{aligned} \text{Span}\{v_0, v_1, \dots, v_{k-1}\} \\ = \{x_0v_0 + x_1v_1 + \dots + x_{k-1}v_{k-1} \mid x_0, x_1, \dots, x_{k-1} \text{ skalarer}\}. \end{aligned}$$

Vektorerne

$$x_0v_0 + x_1v_1 + \dots + x_{k-1}v_{k-1}$$

er netop alle de mulige lineære kombinationer af v_0, \dots, v_{k-1} .

Eksempel 12.14. xy -planen i \mathbb{R}^3 består af alle vektorer

$$\begin{bmatrix} x \\ y \\ 0 \end{bmatrix}.$$

Men

$$\begin{bmatrix} x \\ y \\ 0 \end{bmatrix} = x \begin{bmatrix} 1 \\ 0 \\ 0 \end{bmatrix} + y \begin{bmatrix} 0 \\ 1 \\ 0 \end{bmatrix} = xe_0 + ye_1.$$

Så xy -planen er netop $\text{Span}\{e_0, e_1\}$. Δ

Definition 12.15. Lad $A \in \mathbb{R}^{m \times n}$. Så er *søjlerummet* $S(A)$ af A rummet udspændt af dens søjler.

Eksempel 12.16. Matricen

$$A = \begin{bmatrix} 1 & 0 & 2 \\ 0 & 1 & 1 \\ 0 & 0 & 0 \end{bmatrix}$$

har søjler

$$\begin{bmatrix} 1 \\ 0 \\ 0 \end{bmatrix}, \quad \begin{bmatrix} 0 \\ 1 \\ 0 \end{bmatrix}, \quad \begin{bmatrix} 2 \\ 1 \\ 0 \end{bmatrix}.$$

Så søjlerummet består af alle vektorer af formen

$$x_0 \begin{bmatrix} 1 \\ 0 \\ 0 \end{bmatrix} + x_1 \begin{bmatrix} 0 \\ 1 \\ 0 \end{bmatrix} + x_2 \begin{bmatrix} 2 \\ 1 \\ 0 \end{bmatrix} = \begin{bmatrix} x_0 + 2x_2 \\ x_1 + x_2 \\ 0 \end{bmatrix}. \quad (12.4)$$

Nar x_i varierer får vi alle vektorer af formen

$$\begin{bmatrix} s \\ t \\ 0 \end{bmatrix},$$

dvs. i dette tilfælde er søjlerummet $S(A)$ planen i \mathbb{R}^3 udspændt af e_0 og e_1 .

Bemærk at

$$Ax = \begin{bmatrix} 1 & 0 & 2 \\ 0 & 1 & 1 \\ 0 & 0 & 0 \end{bmatrix} \begin{bmatrix} x_0 \\ x_1 \\ x_2 \end{bmatrix} = \begin{bmatrix} x_0 + 2x_2 \\ x_1 + x_2 \\ 0 \end{bmatrix},$$

som er det samme som resultatet i (12.4). Så søjlerummet $S(A)$ består netop af alle vektorer der kan fås som Ax for $x \in \mathbb{R}^3$. \triangle

Den sidste bemærkning giver anledning til:

Proposition 12.17. For $A \in \mathbb{R}^{m \times n}$ er søjlerummet

$$S(A) = \{Ax \mid x \in \mathbb{R}^n\}.$$

Specielt har ligningssystemet

$$Ax = b$$

en løsning hvis og kun hvis b ligger i søjlerummet $S(A)$ af A .

Bevis. Skriv

$$A = [v_0 \mid v_1 \mid \dots \mid v_{n-1}].$$

Så har vi

$$Ax = [v_0 \mid v_1 \mid \dots \mid v_{n-1}] \begin{bmatrix} x_0 \\ x_1 \\ \vdots \\ x_{n-1} \end{bmatrix} = x_0 v_0 + x_1 v_1 + \dots + x_{n-1} v_{n-1}.$$

Det følger at de eneste vektor der kan skrives i formen Ax er netop lineære kombinationer af søjlerne af A , dvs. elementer af søjlerummet $S(A)$ for A . \square

Hvis vi har betragter singulærværdidekomponering i ydre produktform

$$A = \sigma_0 u_0 v_0^T + \sigma_1 u_1 v_1^T + \dots + \sigma_{k-1} u_{k-1} v_{k-1}^T$$

så har vi $\sigma_0 \geq \sigma_1 \geq \dots \geq \sigma_{k-1} \geq 0$. Lad $r \leq k$ være så at σ_{r-1} er den sidste singulærværdi som er ikke nul. Så er A givet ved

$$A = \sigma_0 u_0 v_0^T + \sigma_1 u_1 v_1^T + \dots + \sigma_{r-1} u_{r-1} v_{r-1}^T.$$

Beregner vi Ax , får vi

$$Ax = \sigma_0 \langle v_0, x \rangle u_0 + \sigma_1 \langle v_1, x \rangle u_1 + \dots + \sigma_{r-1} \langle v_{r-1}, x \rangle u_{r-1},$$

som er en lineær kombination af u_0, u_1, \dots, u_{r-1} . Ved at lade x løbe over v_0, v_1, \dots, v_{r-1} , ser vi at vi får alle sådanne lineære kombinationer på denne måde. Det betyder at

$$S(A) = \text{Span}\{u_0, u_1, \dots, u_{r-1}\}.$$

Eksempel 12.18. Betragt $v = (1, 0, -1) \in \mathbb{R}^3$. Sæt

$$W = v^\perp = \{w \in \mathbb{R}^3 \mid \langle v, w \rangle = 0\}.$$

Så er W et underrum: (a) $\langle v, 0 \rangle = 0$ giver $0 \in W$, så W er ikke tom, (b) $u, w \in W$ siger $\langle v, u \rangle = 0 = \langle v, w \rangle$, men så er $\langle v, u + w \rangle = \langle v, u \rangle + \langle v, w \rangle = 0 + 0 = 0$, som giver $u + w \in W$, (c) for $w \in W$ og $s \in \mathbb{R}$ har vi $\langle v, sw \rangle = s \langle v, w \rangle = 0$, så $sw \in W$. \triangle

Vi vil møde flere eksempler på underrum efterhånden.

Kapitel 13

Indre produkter

13.1 Definitioner og eksempler

Vi har allerede mødt det standard indre produkt på \mathbb{R}^n , og set nogle af dens egenskaber i lemma 8.2. Dette kan udvides udmiddelbart på den følgende måde.

Definition 13.1. Et *indre produkt* på et vektorrum V med enten reelle eller komplekse skalarer er skalarer $\langle u, v \rangle$ for alle $u, v \in V$ således at

- (I1) $\langle u, v \rangle = \overline{\langle v, u \rangle}$,
- (I2) $\langle su + tw, v \rangle = s\langle u, v \rangle + t\langle w, v \rangle$,
- (I3) $\langle v, v \rangle \in \mathbb{R}$ og $\langle v, v \rangle \geq 0$,
- (I4) for $v \neq 0$ er $\langle v, v \rangle > 0$,

for alle $u, v, w \in V$ og alle s, t skalar.

Vi får fra del (I1) og del (I2) at

$$\langle u, sv + tw \rangle = \overline{s}\langle u, v \rangle + \overline{t}\langle u, w \rangle. \quad (13.1)$$

Som tidligere, siger vi at u og v er *ortogonal* hvis $\langle u, v \rangle = 0$.

Bemærkning 13.2. Hvis vores skalarer er reelle tal så har konjugation ingen effekt og kan droppes fra det ovenstående. \diamond

Bemærkning 13.3. Sætters $\|v\| = \sqrt{\langle v, v \rangle}$, fås en *norm* på V , som opfylder det der svarer til definition 11.4, dvs.

- (N1) $\|v\| \geq 0$ for alle $v \in V$, og $\|v\| = 0$ kun for $v = 0$,
- (N2) $\|sv\| = |s|\|v\|$ for alle s skalar,
- (N3) $\|v + w\| \leq \|v\| + \|w\|$.

13 INDRE PRODUKTER

(Beviset for del (N3) kræver dog Cauchy-Schwarz uligheden nedenfor.) Vi kalder $\|\cdot\|$ *normen induceret* af $\langle \cdot, \cdot \rangle$.

I situationen hvor vores skalarer er reelle kan man definere *vinklen* mellem u og v stort set som i definition 8.5, dvs.

$$\cos \theta = \frac{\langle u, v \rangle}{\|u\| \|v\|},$$

når $u \neq 0 \neq v$. Igen er det Cauchy-Schwarz uligheden, som sikrer at dette giver mening. \diamond

Eksempel 13.4. På \mathbb{C}^n er det standard indre produkt givet ved

$$\langle u, v \rangle = u_0 \overline{v_0} + u_1 \overline{v_1} + \cdots + u_{n-1} \overline{v_{n-1}}. \quad (13.2)$$

Så

$$\begin{aligned} \left\langle \begin{bmatrix} 1+i \\ 2-i \end{bmatrix}, \begin{bmatrix} -1+i \\ 1+i \end{bmatrix} \right\rangle &= (1+i) \overline{(-1+i)} + (2-i) \overline{(1+i)} \\ &= (1+i)(-1-i) + (2-i)(1-i) \\ &= -1-i-i-i^2 + 2-2i-i+i^2 \\ &= 1-5i. \end{aligned}$$

Bemærk at

$$\langle u, v \rangle = \overline{v}^T u,$$

da $u_j \overline{v_j} = \overline{v_j} u_j$.

Den tilhørende norm er

$$\|v\|_2 = \|v\| = \sqrt{|v_0|^2 + |v_1|^2 + \cdots + |v_{n-1}|^2}.$$

Så

$$\left\| \begin{bmatrix} 1+3i \\ 2-4i \end{bmatrix} \right\|_2 = \sqrt{((1)^2 + 3^2) + (2^2 + (-4)^2)} = \sqrt{1+9+4+16} = \sqrt{30}.$$

I python kan `np.vdot` godt bruges til at beregne disse indre produkter, men man skal passe på rækkefølgen:

det indre produkt mellem komplekse vektorer u og v gives som `np.vdot(v, u)`, med u og v i modsat rækkefølge.

Funktionen `np.linalg.norm` virker, som det skal.

```
>>> import numpy as np
>>> u = np.array([1.0 + 1.0j, 2.0 - 1.0j])[:, np.newaxis]
>>> v = np.array([-1.0 + 1.0j, 1.0 + 1.0j])[:, np.newaxis]
>>> np.vdot(v, u)
(1-5j)
>>> w = np.array([1.0 + 3.0j, 2.0 - 4.0j])[:, np.newaxis]
>>> np.linalg.norm(w)
5.477225575051661
>>> np.vdot(w, w)
(30+0j)
>>> np.sqrt(np.vdot(w, w))
(5.477225575051661+0j)
```

△

Eksempel 13.5. På \mathbb{R}^n givet reelle tal $w_0, \dots, w_{n-1} > 0$ er

$$\langle u, v \rangle := w_0 u_0 v_0 + w_1 u_1 v_1 + \dots + w_{n-1} u_{n-1} v_{n-1}$$

et indre produkt. Dette kan bruges når forskellige koordinatretninger måler forskellige type størrelse, f.eks. med forskellige enheder. Vi kalder dette et *vægtet indre produkt*.

På \mathbb{C} er det tilsvarende udtryk

$$\langle u, v \rangle := w_0 u_0 \overline{v_0} + w_1 u_1 \overline{v_1} + \dots + w_{n-1} u_{n-1} \overline{v_{n-1}}.$$

△

Eksempel 13.6. På $V = \mathbb{R}^{m \times n}$, kan vi definere

$$\langle A, B \rangle = \langle (a_{ij}), (b_{ij}) \rangle = \sum_{i=1}^m \sum_{j=1}^n a_{ij} b_{ij}.$$

Dette definerer et indre produkt, og den tilhørende norm kaldes *Frobeniusnormen*

$$\|A\|_F = \sqrt{\sum_{i=1}^m \sum_{j=1}^n (a_{ij})^2}.$$

Der skal understreges at $\|A\|_2 \neq \|A\|_F$ generelt. F.eks. har

$$A = \begin{bmatrix} 2 & 0 \\ 0 & 1 \end{bmatrix}$$

$$\|A\|_2 = \sigma_0 = 2, \text{ men } \|A\|_F = \sqrt{2^2 + 1^2} = \sqrt{5}. \quad \Delta$$

Eksempel 13.7. Lad V være vektorrummet, som består af kontinuerte funktioner $f: [a, b] \rightarrow \mathbb{R}$, $b > a$. Det *L^2 -indre produkt* er givet ved

$$\langle f, g \rangle = \int_a^b f(x)g(x) dx.$$

Det væsentlige her er at $\langle f, f \rangle = \int_a^b f(x)^2 dx \geq 0$. Hvis f er ikke identisk 0, så er der et punkt $x_0 \in [a, b]$ hvor $f(x_0) \neq 0$. Da f er kontinuert er der et interval $[c, d] \subset [a, b]$, $d > c$, omkring x_0 med $f(x)^2 \geq f(x_0)^2/2$ for $c \leq x \leq d$. Det følger at $\langle f, f \rangle \geq \int_c^d f(x_0)^2/2 dx > 0$, så del (I4) er opfyldt.

F.eks. på $[0, 1]$ har vi for funktionerne $f(x) = x$, $g(x) = x^2$ at

$$\langle f, g \rangle = \int_0^1 x \times x^2 dx = \int_0^1 x^3 dx = \left[\frac{1}{4}x^4 \right]_0^1 = \frac{1}{4} - 0 = \frac{1}{4}.$$

Givet $w: [a, b] \rightarrow \mathbb{R}$ kontinuert med $w(x) > 0$, for alle x , kan man definere et *vægtet indre produkt* ved

$$\langle f, g \rangle = \int_a^b f(x)g(x)w(x) dx.$$

Δ

Eksempel 13.8. For V vektorrummet af kontinuerte $f: [a, b] \rightarrow \mathbb{C}$ har man *L^2 -indre produkt*

$$\langle f, g \rangle = \int_a^b f(x)\overline{g(x)} dx.$$

Δ

Mange af vores begreber og resultater for den standard indre produkt gælder også for vilkårlige indre produkter. Specielt siger vi at u, v er ortogonal hvis $\langle u, v \rangle = 0$. Vi har

Proposition 13.9. *Lad $\langle \cdot, \cdot \rangle$ være et indre produkt på et vektorrum V med skalarer \mathbb{R} eller \mathbb{C} . Sæt $\|v\| = \sqrt{\langle v, v \rangle}$ til at være den inducerede norm. De følgende udsagn gælder.*

- (a) (Pythagoras) For u, v ortogonal, er $\|u + v\|^2 = \|u\|^2 + \|v\|^2$.
 (b) (Cauchy-Schwarz) For alle $u, v \in V$,

$$|\langle u, v \rangle| \leq \|u\| \|v\|,$$

med lighed kun hvis u og v er parallelle, dvs. enten $v = 0$ eller $u = sv$ for en skalar s .

Bevis. For Pythagoras sætningen gentager vi beviset for proposition 8.3 med $\|\cdot\|_2$ erstattet af $\|\cdot\|$: $\langle u, v \rangle = 0$ giver $\langle v, u \rangle = \overline{\langle u, v \rangle} = 0$, så for u, v ortogonal har vi

$$\begin{aligned} \|u + v\|_2^2 &= \langle u + v, u + v \rangle = \langle u, u + v \rangle + \langle v, u + v \rangle && \text{definition 13.1(I2),} \\ &= \langle u, u \rangle + \langle u, v \rangle + \langle v, u \rangle + \langle v, v \rangle && \text{ligning (13.1),} \\ &= \|u\|_2^2 + \|v\|_2^2. \end{aligned}$$

For Cauchy-Schwarz uligheden, bemærk først at der intet at vise hvis $v = 0$, da $\langle u, 0 \rangle = \langle u, 0 \times 0 \rangle = 0 \langle u, 0 \rangle = 0$, og $\|v\| = 0$.

Hvis $v \neq 0$, betagter vi

$$w = u - tv, \quad \text{med } t = \langle u, v \rangle / \|v\|^2.$$

Når $u = sv$ for en skalar s , har vi $\langle u, v \rangle = s \langle v, v \rangle = s \|v\|^2$, og så $t = s$ og $w = sv - tv = 0$. Generelt er

$$\begin{aligned} 0 &\leq \|w\|^2 = \langle u - tv, u - tv \rangle \\ &= \langle u, u \rangle - \langle tv, u \rangle - \langle u, tv \rangle + \langle tv, tv \rangle \\ &= \|u\|^2 - t \langle v, u \rangle - \bar{t} \langle u, v \rangle + |t|^2 \|v\|^2 \\ &= \|u\|^2 - t \overline{\langle u, v \rangle} - \bar{t} \langle u, v \rangle + |t|^2 \|v\|^2 \\ &= \|u\|^2 - \frac{1}{\|v\|^2} \langle u, v \rangle \overline{\langle u, v \rangle} - \frac{1}{\|v\|^2} \overline{\langle u, v \rangle} \langle u, v \rangle + \frac{1}{\|v\|^4} |\langle u, v \rangle|^2 \|v\|^2 \\ &= \|u\|^2 - \frac{1}{\|v\|^2} |\langle u, v \rangle|^2 - \frac{1}{\|v\|^2} |\langle u, v \rangle|^2 + \frac{1}{\|v\|^2} |\langle u, v \rangle|^2 \\ &= \|u\|^2 - \frac{1}{\|v\|^2} |\langle u, v \rangle|^2. \end{aligned}$$

Vi har så

$$|\langle u, v \rangle|^2 \leq \|u\|^2 \|v\|^2.$$

Tages kvadratroden, fås Cauchy-Schwarz uligheden. Vi har lighed kun når $w = 0$, dvs. kun når $u = tv$, så u er parallel med v . \square

13.2 Ortogonale samlinger og tilnærmelse

Med vores nye definitioner, kan vi arbejde med ortogonale og ortonormale samlinger vektorer stort set som førhen. Specielt gælder proposition 8.17: hvis v_0, \dots, v_{k-1} er ortogonal og $u \in \text{Span}\{v_0, \dots, v_{k-1}\}$, så har vi

$$u = x_0 v_0 + \dots + x_{k-1} v_{k-1} \quad \text{med } x_j = \frac{\langle u, v_j \rangle}{\|v_j\|^2}, \quad j = 0, 1, \dots, k-1.$$

På en tilsvarende måde er projektionen P på $W = \text{Span}\{v_0, v_1, \dots, v_{k-1}\}$, når v_0, v_1, \dots, v_{k-1} er ortogonal, med ingen 0, givet ved

$$P(u) = \frac{\langle u, v_0 \rangle}{\|v_0\|^2} v_0 + \frac{\langle u, v_1 \rangle}{\|v_1\|^2} v_1 + \dots + \frac{\langle u, v_{k-1} \rangle}{\|v_{k-1}\|^2} v_{k-1}. \quad (13.3)$$

Vektoren $w = P(u)$ er den vektor i W , som er tættest på u , dvs. har $\|u - w\|$ mindst muligt. Beviset er det samme, som vi har givet i \mathbb{R}^n , for projektion langs v_0, v_1, \dots, v_{k-1} .

Eksempel 13.10. Lad os betragte rummet V af kontinuerte funktioner $[0, \pi] \rightarrow \mathbb{R}$ udstyret med det L^2 -indre produkt. Vi påstår at

$$1, \cos(x), \cos(2x), \dots, \cos((k-1)x)$$

er en ortogonal samling i V . Dette gælder da for $n > 0$

$$\langle 1, \cos nx \rangle = \int_0^\pi \cos(nx) dx = \left[\frac{1}{n} \sin(nx) \right]_0^\pi = \frac{1}{n} (\sin(n\pi) - \sin(0)) = 0,$$

så 1 er vinkelret på de andre funktioner. Desuden har vi for $m \neq n$ at

$$\begin{aligned} \langle \cos(mx), \cos(nx) \rangle &= \int_0^\pi \cos(mx) \cos(nx) dx \\ &= \frac{1}{2} \int_0^\pi \cos((m+n)x) + \cos((m-n)x) dx \\ &= \frac{1}{2} \left[\frac{1}{m+n} \sin((m+n)x) + \frac{1}{m-n} \sin((m-n)x) \right]_0^\pi = 0. \end{aligned}$$

Så funktionerne er vinkelret på hinanden, som påstået.

Bemærk at

$$\|1\|^2 = \int_0^\pi 1^2 dx = \pi,$$

og

$$\|\cos(nx)\|^2 = \int_0^\pi (\cos(nx))^2 dx = \frac{1}{2} \int_0^\pi (\cos(2nx) + 1) dx = \frac{\pi}{2}.$$

Lad os beregne projektionen af funktionen $1 - x$ på

$$\text{Span}\{1, \cos(x), \cos(2x), \dots, \cos((k-1)x)\}.$$

Vi har

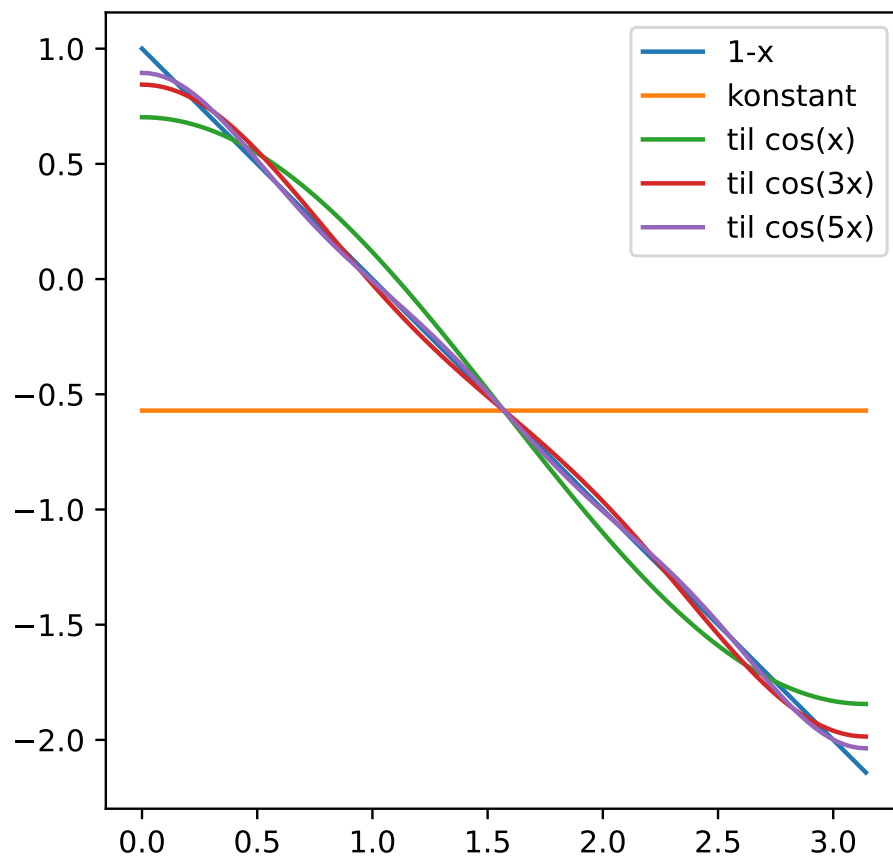
$$\langle 1 - x, 1 \rangle = \int_0^\pi (1 - x)1 dx = \left[x - \frac{1}{2}x^2 \right]_0^\pi = \pi - \frac{1}{2}\pi^2 = \frac{\pi}{2}(2 - \pi)$$

og, for $n > 0$,

$$\begin{aligned} \langle 1 - x, \cos(nx) \rangle &= \int_0^\pi \cos(nx) - x \cos(nx) dx = 0 - \int_0^\pi x \cos(nx) dx \\ &= -\left[\frac{1}{n} x \sin(nx) \right]_0^\pi + \int_0^\pi \frac{1}{n} \sin(nx) dx \\ &= 0 + \left[-\frac{1}{n^2} \cos(nx) \right]_0^\pi = -\frac{1}{n^2}(-\cos(n\pi) + 1) \\ &= \frac{1}{n^2}((-1)^n - 1) = \begin{cases} -\frac{2}{n^2}, & \text{for } n \text{ ulige,} \\ 0, & \text{for } n \text{ lige.} \end{cases} \end{aligned}$$

Sættes ind i (13.3), får vi

$$\begin{aligned} P(1 - x) &= \frac{\langle 1 - x, 1 \rangle}{\|1\|^2} 1 + \frac{\langle 1 - x, \cos(x) \rangle}{\|\cos(x)\|^2} \cos(x) + \frac{\langle 1 - x, \cos(2x) \rangle}{\|\cos(2x)\|^2} \cos(2x) \\ &\quad + \dots + \frac{\langle 1 - x, \cos((k-1)x) \rangle}{\|\cos((k-1)x)\|^2} \cos((k-1)x) \\ &= \frac{\pi(2 - \pi)/2}{\pi} 1 + \frac{2/1^2}{\pi/2} \cos(x) + 0 + \frac{2/3^2}{\pi/2} \cos(3x) + 0 \\ &\quad + \frac{2/5^2}{\pi/2} \cos(5x) + \dots + \frac{2/r^2}{\pi/2} \cos(rx) \\ &= \frac{1}{2}(2 - \pi) + \frac{4}{\pi} \cos(x) + \frac{4}{9\pi} \cos(3x) \\ &\quad + \frac{4}{25\pi} \cos(5x) + \dots + \frac{4}{r^2\pi} \cos(rx), \end{aligned} \tag{13.4}$$

Figur 13.1: Fourier cosinus tilnærmelse for $1 - x$.

hvor r er det største ulige tal $\leq k - 1$. Dette kaldes en *Fourier cosinus udvikling* af funktionen $1 - x$. Billedfilformatet jpeg er baseret på en diskret udgave af Fourier cosinus rækkeudvikling.

△

Eksempel 13.11. Fourier rækker generelt kommer fra at betragte kontinuerte funktioner $[-\pi, \pi] \rightarrow \mathbb{R}$. Her er

$$1, \cos(x), \sin(x), \cos(2x), \sin(2x), \dots, \cos((k-1)x), \sin((k-1)x) \quad (13.5)$$

ortogonal med

$$\|1\|^2 = 2\pi$$

og

$$\|\cos(nx)\|^2 = \pi = \|\sin(nx)\|^2.$$

Man kan beregne en tilnærmelse til andre kontinuerte funktioner på $[-\pi, \pi]$ ved at projicere på rummet udspændt af (13.5):

$$\begin{aligned} P(f) = & a_0 + c_1 \cos(x) + b_1 \sin(x) \\ & + c_2 \cos(2x) + b_2 \sin(2x) \\ & + c_3 \cos(3x) + b_3 \sin(3x) \\ & + \cdots + c_{k-1} \cos((k-1)x) + b_{k-1} \sin((k-1)x), \end{aligned}$$

med

$$\begin{aligned} a_0 &= \frac{\langle f, 1 \rangle}{\|1\|^2} = \frac{1}{2\pi} \int_{-\pi}^{\pi} f(x) dx, \\ c_m &= \frac{\langle f, \cos(mx) \rangle}{\|\cos(mx)\|^2} = \frac{1}{\pi} \int_{-\pi}^{\pi} f(x) \cos(mx) dx \\ b_m &= \frac{\langle f, \sin(mx) \rangle}{\|\sin(mx)\|^2} = \frac{1}{\pi} \int_{-\pi}^{\pi} f(x) \sin(mx) dx. \end{aligned}$$

I det næste afsnit vil vi se hvordan, udregning af denne type kan udføres numerisk i python. △

13.3 Numerisk integration

Hvis vi vil regne eksempler som 13.10 i python, skal vi have styr på hvordan vi kan beregne integraler numerisk.

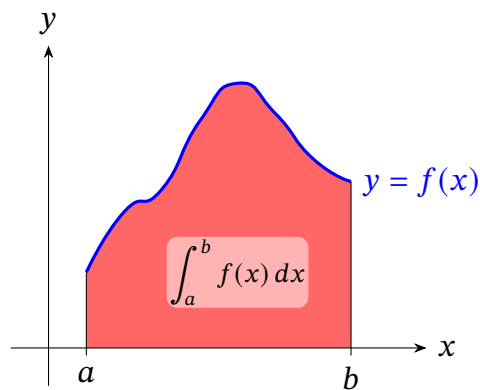
Givet en kontinuert funktion $f: [a, b] \rightarrow \mathbb{R}$ er integralet $\int_a^b f(x) dx$ arealet under dens graf, se figur 13.2. Der er forskellige måder vi kan lave en tilnærmelse til dette areal.

Lad x_0, x_1, \dots, x_{n-1} være n punkter ligelig fordelt fra $a = x_0$ til $b = x_{n-1}$. Sådant en samling får vi fra `np.linspace(a, b, n)`. For hvert x_j kan vi betragte rektanglet R med venstre side ved x_j , højre side ved x_{j+1} , og med højde $f(x_j)$, se figur 13.3. Rektanglet har areal

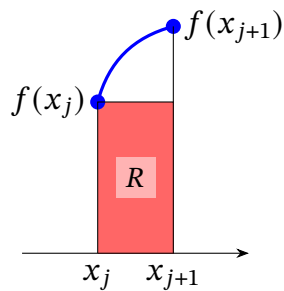
$$hf(x_j)$$

hvor $h = x_{j+1} - x_j$. Vi har sådan et rektangel for hvert venstre start punkt x_0, x_1, \dots, x_{n-2} , da rektanglet til højre fra x_{n-1} ligger uden for det område vi

13 INDRE PRODUKTER



Figur 13.2: Integral.



Figur 13.3: Rektanglet fra den venstre funktionsværdi.

arbejder med. En mulig tilnærmelse til integralet er summen af arealerne på disse rektangler:

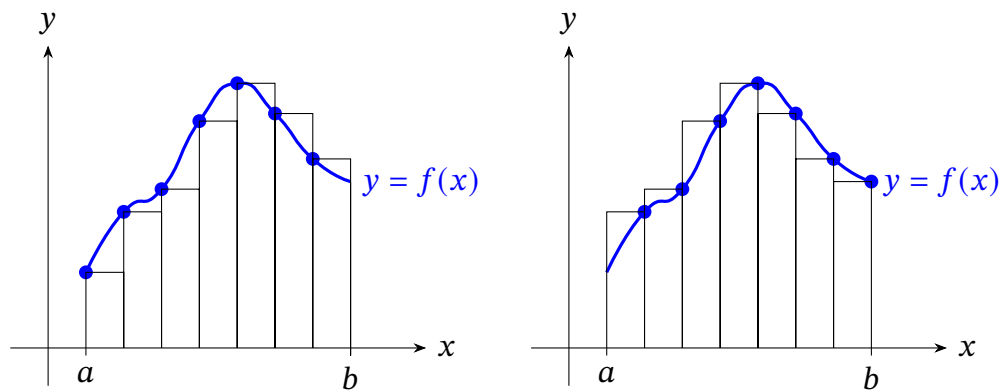
$$\begin{aligned} & hf(x_0) + hf(x_1) + \cdots + hf(x_{n-2}) \\ &= h(f(x_0) + f(x_1) + \cdots + f(x_{n-2})). \end{aligned}$$

Værdien h kan regnes, som

$$h = \frac{1}{n-1}(b-a),$$

da vi har delt $[a, b]$ op i $(n-1)$ -delintervaller. Ved brug af `np.linspace()` er det ikke nødvendigt at beregne h , man kan bede `np.linspace()` at returnere det ved hjælp af `np.linspace(a, b, n, retstep=True)`. F.eks. tager vi

13.3 NUMERISK INTEGRATION



Figur 13.4: Venstre- og højre-tilnærmelse til integral.

funktionen $f(x) = x^3$ på $[0, 2]$, hvor vi ved at $\int_0^2 x^3 dx = [x^4/4]_0^2 = 4$, kan vi regne i python med $n = 100$:

```
>>> import numpy as np
>>> n = 100
>>> x, h = np.linspace(0, 2, n, retstep=True)
>>> h * np.sum((x**3)[: -1])
3.919600040812163
```

Dette er lidt mindre end 4, da funktionen er voksende og alle rektangler ligger under grafen.

Som alternativ kunne man brug funktionsværdien i den højre endepunkt af intervallet $[x_j, x_{j+1}]$, se figur 13.4, og får tilnærmelsen

$$h(f(x_1) + f(x_2) + \cdots + f(x_{n-1}))$$

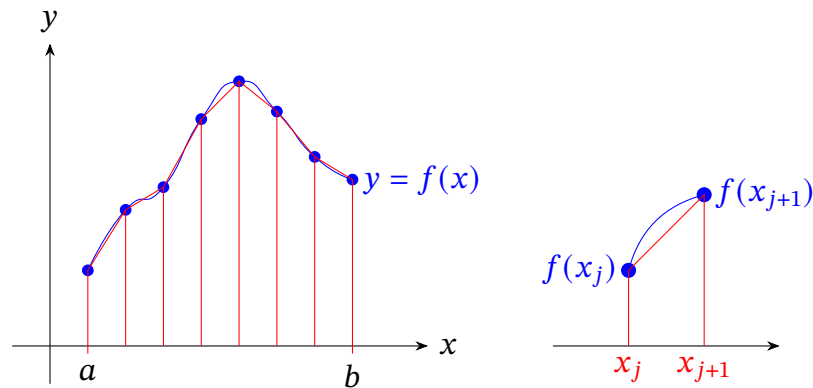
```
>>> h * np.sum((x**3)[1:])
4.081216202428324
```

Denne gang, for funktionen $f(x) = x^3$, får vi et svar der lidt for stort.

Et kompromis er at erstatte rektanglet med trapezet, som i figur 13.5. Dette trapez har areal

$$h \frac{1}{2} (f(x_j) + f(x_{j+1})).$$

13 INDRE PRODUKTER



Figur 13.5: Trapez-tilnærmelse til integral.

Dette giver trapez-tilnærmelsen til integralet

$$\frac{1}{2}h(f(x_0) + 2f(x_1) + 2f(x_2) + \cdots + 2f(x_{n-2}) + f(x_{n-1})). \quad (13.6)$$

Denne regel er implementeret i python som `np.trapz()`, som kan bruges på følgende måde

```
>>> np.trapz(x**3, dx=h)
4.000408121620244
```

som er væsentligt tættere på den rigtige værdi for integralet.

Generelt kan det vises at denne trapezregel (13.6) på funktioner, som har et andet afledte, beregner integraler $\int_a^b f(x) dx$ inden for en fejl af

$$\frac{1}{12}h^3(b-a) \max\{|f''(x)| \mid a \leq x \leq b\}.$$

Trapezformlen (13.6) kommer fra at erstattet f med linjestykker. Man kan i stedet for bruge højre ordens polynomier. Andengradspolynomier giver anledning til Simpsons regel

$$\begin{aligned} \int_a^b f(x) dx \approx \frac{1}{3}h(f(x_0) + 4f(x_1) + 2f(x_2) + 4f(x_3) + 2f(x_4) \\ + \cdots + 2f(x_{n-3}) + 4f(x_{n-2}) + f(x_{n-1})) \end{aligned} \quad (13.7)$$

for n ulig. Denne gang er fejlen i beregning af integralet højst

$$\frac{1}{90}h^4(b-a)\max\{|f''''(x)| \mid a \leq x \leq b\}.$$

Lad os anvender trapezreglen for beregner eksempel 13.10 numerisk i python.

```
>>> import numpy as np
>>> n = 100
>>> x, h = np.linspace(0, np.pi, 100, retstep=True)
```

Vores ortogonale funktioner er 1, $\cos(x)$, $\cos(2x)$, osv. Den første gemmer vi i

```
>>> konstant = np.ones_like(x)
```

de andre kan bare regnes som $\text{np.cos}(m*x)$. Lad os bekræfter at et par af disse er ortogonal. Indre produktet er $\langle f, g \rangle = \int_0^\pi f(x)g(x) dx$, så vi kan bruge $\text{np.trapz}()$ på $f*g$:

```
>>> def indre_produkt(f, g, h):
...     return np.trapz(f * g, dx=h)
...
>>> indre_produkt(konstant, np.cos(3*x), h)
-2.0816681711721685e-17
>>> indre_produkt(np.cos(2*x), np.cos(3*x), h)
-2.7755575615628914e-17
>>> indre_produkt(np.cos(5*x), np.cos(3*x), h)
-9.020562075079397e-17
```

som viser at de er ortogonale inden for machine epsilon. Vi har også beregnet $\|1\|^2 = \pi$, $\|\cos(mx)\|^2 = \pi/2$, som kan også bekræftes i python

```
>>> def norm_sq(f, h):
...     return indre_produkt(f, f, h)
...
>>> norm_sq(konstant, h) - np.pi
-4.440892098500626e-16
>>> norm_sq(np.cos(x), h) - np.pi/2
```

13 INDRE PRODUKTER

```
-2.220446049250313e-16
>>> norm_sq(np.cos(4*x), h) - np.pi/2
0.0
```

Dette er igen korrekt inden for machine epsilon.

I eksempel 13.10 beregnede vi projektionerne af $1 - x$ på de første k basis vektorer. Lad os indføre en funktion der gøre dette for os

```
>>> def proj(f, k, x, h):
...     out = (indre_produkt(f, konstant, h)
...           / norm_sq(konstant, h)
...           * konstant)
...     for m in range(1, k, 2):
...         out += (indre_produkt(f, np.cos(m*x), h)
...                / norm_sq(np.cos(m*x), h)
...                * np.cos(m*x))
...     return out
... 
```

Plotter man $1-x$, og $\text{proj}(1-x, k, x, h)$, for $k = 1, 2, 4, 6$ fås figur 13.1. Bemærk at

```
>>> np.allclose(proj(1-x, 3, x, h), proj(1-x, 2, x, h),
...             atol = np.finfo(float).eps)
True
```

som bekræfter at $\cos(2x)$ bidrager ikke til denne projektion, se formlen (13.4). Sådanne bidrag falder væk da indre produkterne $\langle 1 - x, \cos(mx) \rangle$ er nul for m lige:

```
>>> for m in range(2, 7, 2):
...     print(indre_produkt(1-x, np.cos(m*x), h))
... 
```

```
-5.551115123125783e-17
-5.551115123125783e-17
-2.7755575615628914e-17
```

Kapitel 14

Ortogonal samlinger: klassisk Gram-Schmidt

Vi har brugt ortogonale samlinger for at finde tilnærmelser til vektorer og funktioner. Her vil vi undersøge forskellige metode for at konstruere sådanne samlinger.

14.1 Den klassiske Gram-Schmidt proces

Lad V være et vektorrum med indre produkt $\langle \cdot, \cdot \rangle$. Husk at projektion på linjen udspændt af $v \neq 0$ er givet ved

$$\text{pr}_v(u) = \frac{\langle u, v \rangle}{\|v\|^2} v.$$

Vektoren

$$u - \text{pr}_v(u)$$

er derefter vinkelret på v . Vi har også set at for v_0, \dots, v_{k-1} ortogonal, med ingen 0, er projektionen til $\text{Span}\{v_0, \dots, v_{k-1}\}$ givet ved

$$\text{pr}_{v_0, \dots, v_{k-1}}(u) = \frac{\langle u, v_0 \rangle}{\|v_0\|^2} v_0 + \dots + \frac{\langle u, v_{k-1} \rangle}{\|v_{k-1}\|^2} v_{k-1}$$

og at $u - \text{pr}_{v_0, \dots, v_{k-1}}(u)$ er ortogonal på v_0, \dots, v_{k-1} .

Vi kan bruge dette til at opbygge en samling ortogonale vektorer nogenlunde på følgende måde. Input er vektorer $u_0, u_1, \dots, u_{k-1} \in V$.

14 ORTOGONALE SAMLINGER: KLASSISK GRAM-SCHMIDT

KLASSISK GRAM-SCHMIDT PROCES—SKITSE(u_0, u_1, \dots, u_{k-1})

- 1 Sæt $w_0 = u_0$, $v_0 = c_0 w_0$ for et valgfrit $c_0 \neq 0$.
- 2 Sæt $w_1 = u_1 - \text{pr}_{v_0}(u_1)$, $v_1 = c_1 w_1$, $c_1 \neq 0$.
- 3 Sæt $w_2 = u_2 - \text{pr}_{v_0, v_1}(u_2)$, $v_2 = c_2 w_2$, $c_2 \neq 0$.
- ...
- 4 Sæt $w_r = u_r - \text{pr}_{v_0, \dots, v_{r-1}}(u_r)$, $v_r = c_r w_r$, $c_r \neq 0$.
- ...
- 5 **return** v_0, v_1, \dots, v_{k-1} , en ortogonal samling,
der udspænder det samme rum som u_0, u_1, \dots, u_{k-1} .

Vælges $c_r = 1/\|w_r\|$ i hvert trin, så bliver v_0, v_1, \dots, v_{k-1} ortonormal.

Eksempel 14.1. I \mathbb{R}^4 med det standard indre produkt betragt

$$u_0 = \begin{bmatrix} 1 \\ 1 \\ 0 \\ 0 \end{bmatrix}, \quad u_1 = \begin{bmatrix} 2 \\ 1 \\ 1 \\ 1 \end{bmatrix}, \quad u_2 = \begin{bmatrix} -1 \\ 3 \\ 1 \\ -1 \end{bmatrix}.$$

Vi sætter

$$v_0 = w_0 = u_0 = \begin{bmatrix} 1 \\ 1 \\ 0 \\ 0 \end{bmatrix}.$$

Bemærk $\|v_0\|_2^2 = 1^2 + 1^2 + 0^2 + 0^2 = 2$. Vi har $\langle u_1, v_0 \rangle = 2 \times 1 + 1 \times 1 + 1 \times 0 + 1 \times 0 = 3$.
Så

$$\begin{aligned} w_1 &= u_1 - \text{pr}_{v_0}(u_1) \\ &= u_1 - \frac{\langle u_1, v_0 \rangle}{\|v_0\|_2^2} v_0 = \begin{bmatrix} 2 \\ 1 \\ 1 \\ 1 \end{bmatrix} - \frac{3}{2} \begin{bmatrix} 1 \\ 1 \\ 0 \\ 0 \end{bmatrix} = \frac{1}{2} \begin{bmatrix} 1 \\ -1 \\ 2 \\ 2 \end{bmatrix}. \end{aligned}$$

Lad os gange i gennem med $c_1 = 2$ for at fjerne brøken og sætte

$$v_1 = 2w_1 = \begin{bmatrix} 1 \\ -1 \\ 2 \\ 2 \end{bmatrix},$$

14.1 DEN KLASSISKE GRAM-SCHMIDT PROCES

som har $\|v_2\|_2^2 = 1 + 1 + 4 + 4 = 10$. Vi regner så at

$$\begin{aligned} w_2 &= u_2 - \text{pr}_{v_0, v_1}(u_2) \\ &= u_2 - \frac{\langle u_2, v_0 \rangle}{\|v_0\|_2^2} v_0 - \frac{\langle u_2, v_1 \rangle}{\|v_1\|_2^2} v_1 = \begin{bmatrix} -1 \\ 3 \\ 1 \\ -1 \end{bmatrix} - \frac{-1+3}{2} \begin{bmatrix} 1 \\ 1 \\ 0 \\ 0 \end{bmatrix} - \frac{-1-3+2-2}{10} \begin{bmatrix} 1 \\ 1 \\ 2 \\ 2 \end{bmatrix} \\ &= \begin{bmatrix} -1 \\ 3 \\ 1 \\ -1 \end{bmatrix} - \begin{bmatrix} 1 \\ 1 \\ 0 \\ 0 \end{bmatrix} + \frac{2}{5} \begin{bmatrix} 1 \\ 1 \\ 2 \\ 2 \end{bmatrix} = \frac{1}{5} \begin{bmatrix} -8 \\ 8 \\ 9 \\ -1 \end{bmatrix}. \end{aligned}$$

Sættes $v_2 = 5w_2$, fås at

$$v_0 = \begin{bmatrix} 1 \\ 1 \\ 0 \\ 0 \end{bmatrix}, \quad v_1 = \begin{bmatrix} 1 \\ -1 \\ 2 \\ 2 \end{bmatrix}, \quad v_2 = \begin{bmatrix} -8 \\ 8 \\ 9 \\ -1 \end{bmatrix}$$

er en ortogonal samling, som udspænder $\text{Span}\{u_0, u_1, u_2\}$ i \mathbb{R}^4 .

Det tilsvarende regnestykke i python er

```
>>> import numpy as np

>>> u0 = np.array([1.0, 1.0, 0.0, 0.0])[:, np.newaxis]
>>> u1 = np.array([2.0, 1.0, 1.0, 1.0])[:, np.newaxis]
>>> u2 = np.array([-1.0, 3.0, 1.0, -1.0])[:, np.newaxis]

>>> def proj_på(v, u):
...     return np.vdot(v, u) / np.vdot(v, v) * v
...
>>> v0 = u0
>>> v0
array([[1.],
       [1.],
       [0.],
       [0.]])
>>> w1 = u1 - proj_på(v0, u1)
>>> v1 = 2*w1
```

```

>>> v1
array([[ 1.],
       [-1.],
       [ 2.],
       [ 2.]])
>>> w2 = u2 - proj_på(v0, u2) - proj_på(v1, u2)
>>> v2 = 5*w2
>>> v2
array([[ -8.],
       [  8.],
       [  9.],
       [-1.]])

```

△

Ved beskrivelsen af den klassiske Gram-Schmidt proces ovenfor, brugte vi ordet »nogenlunde«. Grunden til dette er at vi har brug for at w_r , og dermed v_r , er ikke nul i hvert trin. Dette er tilfældet hvis og kun hvis u_0, u_1, \dots, u_{k-1} opfylder:

Definition 14.2. En samling vektorer u_0, u_1, \dots, u_{k-1} i V er *lineært uafhængig* hvis den eneste løsning på ligningen

$$s_0 u_0 + s_1 u_1 + \dots + s_{k-1} u_{k-1} = 0 \quad (14.1)$$

med s_i skalarer er $s_0 = 0 = s_1 = \dots = s_{k-1}$.

Bemærkning 14.3. En måde at tjekke denne betingelse på er at køre algoritmen ovenfor: Hvis $w_r = 0$ for et r , så er $u_r = \text{pr}_{v_0, v_1, \dots, v_{r-1}}(u_r)$. Men denne projektion er en lineær kombination $s_0 u_0 + s_1 u_1 + \dots + s_{r-1} u_{r-1}$ af u_0, \dots, u_{r-1} . Sættes $s_r = -1$ og $s_i = 0$ for $i > r$, fås en løsning til (14.1) med $s_r \neq 0$. Så samlingen er ikke lineært uafhængig.

Omvendt hvis u_0, u_1, \dots, u_{k-1} er lineært uafhængig, så er ingen u_r en lineær kombination af u_0, \dots, u_{r-1} . Specielt er u_r ikke lige med $\text{pr}_{v_0, v_1, \dots, v_{r-1}}(u_r)$, så $w_r \neq 0$ for alle r . ◇

Eksempel 14.4. Lad $u_0, u_1, \dots, u_{k-1} \in \mathbb{R}^n$ og betragt matricen

$$A = [u_0 \mid u_1 \mid \dots \mid u_{k-1}] \in \mathbb{R}^{n \times k}$$

14.1 DEN KLASSISKE GRAM-SCHMIDT PROCES

hvis søjler er $u_r, r = 0, \dots, k-1$. Så er u_0, u_1, \dots, u_{k-1} lineært uafhængig hvis og kun hvis ligningen

$$Ax = 0, \quad x \in \mathbb{R}^k$$

har kun én løsningen, nemlig $x = 0$. Dette gælder fordi $Ax = x_0 u_0 + x_1 u_1 + \dots + x_{k-1} u_{k-1}$, så ligning (14.1) er $Ax = 0$ efter udskiftning af variable. Reduceres A til echelonform får vi så at u_0, u_1, \dots, u_{k-1} er lineært uafhængig hvis alle søjler i A indeholder et pivotelement.

F.eks. vektorerne $u_0 = (1, 0, 2)$, $u_1 = (0, 2, 1)$, $u_2 = (1, 2, 3)$ er ikke lineært uafhængig da

$$\begin{bmatrix} 1 & 0 & 1 \\ 0 & 2 & 2 \\ 2 & 1 & 3 \end{bmatrix} \sim_{R_2 \rightarrow R_2 - 2R_0} \begin{bmatrix} 1 & 0 & 1 \\ 0 & 2 & 2 \\ 0 & 1 & 1 \end{bmatrix} \sim_{R_1 \leftrightarrow R_2} \begin{bmatrix} 1 & 0 & 1 \\ 0 & 1 & 1 \\ 0 & 2 & 2 \end{bmatrix} \sim_{R_2 \leftrightarrow R_2 - 2R_1} \begin{bmatrix} 1 & 0 & 1 \\ 0 & 1 & 1 \\ 0 & 0 & 0 \end{bmatrix}$$

har kun pivot elementer i de første 2 søjler, og ikke i den sidste. Δ

Eksempel 14.5. For $a < b$, betragt vektorrummet af differentiable funktioner $f: [a, b] \rightarrow \mathbb{R}$. Så er polynomierne

$$1, x, x^2, \dots, x^{k-1}$$

lineært uafhængig i V . For at vise dette, antag at

$$p(x) = s_0 1 + s_1 x + s_2 x^2 + \dots + s_{k-1} x^{k-1} = 0(x) = 0, \quad \text{for alle } a \leq x \leq b.$$

Betragt den $(k-1)$ 'te afledte $d^{k-1}p/dx^{k-1}$, som er nødvendigvis 0. Vi får

$$0 = \frac{d^{k-1}p}{dx^{k-1}} = (k-1)! s_{k-1},$$

som giver $s_{k-1} = 0$. Kikker man derefter på den $(k-2)$ 'te afledte, får man s_{k-2} , og fortsætter man i på denne måde, ser man at $s_r = 0$ for alle r .

Bruger man den klassiske Gram-Schmidt på $1, x, x^2, \dots$ for forskellige indre produkter får man nogle kendte ortogonale systemer af polynomier:

(a) *Legendre polynomier* $[a, b] = [-1, 1]$, $\langle f, g \rangle = \int_{-1}^1 f(x)g(x) dx$, giver

$$P_0(x) = 1,$$

$$P_1(x) = x,$$

$$P_2(x) = 3x^2 - 2,$$

$$P_3(x) = 5x^3 - 3x.$$

(b) *Chebyshev polynomier* $[a, b] = [-1, 1]$,

$$\langle f, g \rangle = \int_{-1}^1 \frac{f(x)g(x)}{\sqrt{1-x^2}} dx$$

giver

$$T_0(x) = 1,$$

$$T_1(x) = x,$$

$$T_2(x) = 2x^2 - 1,$$

$$T_3(x) = 4x^3 - 3x,$$

$$T_4(x) = 8x^4 - 8x^2 + 1.$$

Disse og flere andre polynomier af denne type er indbygget i NumPy.

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

from numpy.polynomial import Polynomial as P
from numpy.polynomial import Legendre as L
from numpy.polynomial import Chebyshev as T

x, h = np.linspace(-1, 1, 100, retstep=True)

fig, ax = plt.subplots()
for i in range(6):
    ax.plot(x, L.basis(i)(x))
```

```
fig, ax = plt.subplots()
for i in range(6):
    ax.plot(x, T.basis(i)(x))
```

Vi får koefficienter for de givne polynomier vist via

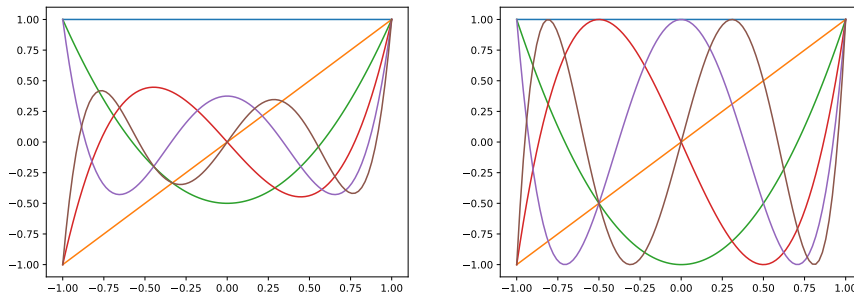
```
np.polynomial.set_default_printstyle('ascii')
print(T.basis(4).convert(kind=P))
```

```
1.0 + 0.0 x**1 - 8.0 x**2 + 0.0 x**3 + 8.0 x**4
```

som passer med vores udtryk for $T_4(x)$.

△

14.2 KLASSISK GRAM-SCHMIDT I \mathbb{R}^n



Figur 14.1: Legendre og Chebychev polynomier.

14.2 Klassisk Gram-Schmidt i \mathbb{R}^n

Vores klassisk Gram-Schmidt starter med u_0, u_1, \dots, u_{k-1} lineært uafhængig. Lad os indføre lidt andet notation i tilfældet hvor vi konstruere en ortonormal samling v_0, v_1, \dots, v_{k-1} . I hvert trin danne vi

$$w_j = u_j - \text{pr}_{v_0, \dots, v_{j-1}}(u_j) = u_j - r_{0j}v_0 - \dots - r_{j-1,j}v_{j-1}. \quad (14.2)$$

Da $\langle v_i, w_j \rangle = 0$ og $\|v_i\|_2 = 1$, kan vi tage indre produktet af v_i med (14.2), som dermed giver $r_{ij} = \langle v_i, u_j \rangle = v_i^T u_j$. For at få en ortonormal samling, sætter vi så

$$v_j = \frac{1}{r_{jj}} w_j, \quad \text{hvor } r_{jj} = \|w_j\|_2. \quad (14.3)$$

Alt i alt har vi den følgende algoritme

KLASSISK GRAM-SCHMIDT(u_0, u_1, \dots, u_{k-1})

```

1  for  $j \in \{0, 1, \dots, k-1\}$ :
2       $w_j = u_j$ 
3      for  $i \in \{0, 1, \dots, j-1\}$ :
4           $r_{ij} = v_i^T u_j$ 
5           $w_j = w_j - r_{ij}v_i$ 
6       $r_{jj} = \|w_j\|_2$ 
7       $v_j = w_j / r_{jj}$ 
```

Givet v_i og r_{ij} kan vi omskrive (14.2) og (14.3) til at få

$$\begin{aligned} u_0 &= r_{00}v_0, \\ u_1 &= r_{01}v_0 + r_{11}v_1, \\ u_2 &= r_{02}v_0 + r_{12}v_1 + r_{22}v_2, \\ &\vdots \\ u_{k-1} &= r_{0,k-1}v_0 + r_{1,k-1}v_1 + \cdots + r_{k-1,k-1}v_{k-1}. \end{aligned} \quad (14.4)$$

Dette kan skrives i matrixform, som

$$\begin{aligned} A &= [u_0 \mid u_1 \mid u_2 \mid \cdots \mid u_{k-1}] \\ &= [v_0 \mid v_1 \mid v_2 \mid \cdots \mid v_{k-1}] \begin{bmatrix} r_{00} & r_{01} & r_{02} & \cdots & r_{0,k-1} \\ 0 & r_{11} & r_{12} & \cdots & r_{1,k-1} \\ 0 & 0 & r_{22} & \cdots & r_{2,k-1} \\ \vdots & \vdots & & \ddots & \vdots \\ 0 & 0 & 0 & & r_{k-1,k-1} \end{bmatrix} \\ &= QR. \end{aligned} \quad (14.5)$$

Her er $A \in \mathbb{R}^{n \times k}$, matricen $Q \in \mathbb{R}^{n \times k}$ har ortonormale søjler v_0, \dots, v_{k-1} , og $R \in \mathbb{R}^{k \times k}$ er øvre triangulær. En dekomponering af denne form kaldes en *tynd QR-dekomponering* af A . En *fuld QR-dekomponering* fås ved at udvide R med 0-tal til en matrix i $\mathbb{R}^{n \times k}$, og at tilføje søjler til Q , f.eks. ved hjælp af korollar 9.18, så vi får en ortogonal matrix i $\mathbb{R}^{n \times n}$.

Hvis samlingen u_0, u_1, \dots, u_{k-1} er lineært uafhængig, giver den klassiske Gram-Schmidt proces os en tynd QR-dekomponering med $r_{ii} > 0$ for alle i .

Når A er en $n \times k$ -matrix, med $n \geq k$, hvis søjler er ikke lineært uafhængig, kan vi køre den klassiske Gram-Schmidt proces med en lille ændring: når et w_j er 0, kan vi bare vælge v_j til at være en tilfældig enhedsvektor vinkelret på v_0, v_1, \dots, v_{j-1} . På denne måde får vi en QR-dekomponering for A med R øvre triangulær, men nogle r_{ii} kan være 0.

Dette giver

Proposition 14.6. *Alle matricer $A \in \mathbb{R}^{n \times k}$, for $n \geq k$, tillader en tynd og end fuld QR-dekomponering.*

Hvis søjlerne af A er lineært uafhængig og $r_{ii} > 0$ for alle i , så er denne tynd QR-dekomponering entydig. \square

14.2 KLASSISK GRAM-SCHMIDT I \mathbb{R}^n

Bemærkning 14.7. Som vi vil se i det næste kapitel er der numeriske problemer med den klassiske Gram-Schmidt process. Vi vil give andre metoder, der er numerisk mere hensigts mæssig. \diamond

Advarsel 14.8. Numpy har en indbygget funktion `np.linalg.qr`. Den producerer ikke en QR-dekomponering af den overstående type, da den tillader $r_{ii} < 0$. $!$

Kapitel 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 v_0, v_1, v_2 .

```
>>> 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 v_0, v_1, v_2 . Vi kan se om v_0, v_1, v_2 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 v_0, v_1, v_2 er enhedsvektorer. Desuden er de andre indre produkter med v_0 af den samme størrelsesorden som s , men de er skuffende langt væk fra machine epsilon. Det værste problem er dog $\langle v_1, v_2 \rangle$, som burde være 0, men er tæt på $1/2$. Det vil sige vinklen mellem v_1 og v_2 er

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

15.1 NUMERISKE PROBLEMER FOR KLASSISK GRAM-SCHMIDT

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

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

Denne omskrivning er gyldigt, da $\text{pr}_{v_0}(u_2)$ er parallelt med v_0 og $\text{pr}_{v_1}(v_0) = 0$, så $\text{pr}_{v_1}(\text{pr}_{v_0}(u_2)) = 0$. 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 v_2 og v_3 vinkelret på hinanden indenfor machine epsilon.

15 FORBEDRING AF GRAM-SCHMIDT

Hvorfor har denne omskrivning givet et bedre resultat? I beregningen via den klassiske Gram-Schmidt proces er v_0 lige med u_0 , 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 v_0 , og derefter beregnes w_1 til at være $(0, -s, s, 0)$, som er vinkelret på u_2 . Dette medvirker til at v_1 bidrager ikke til udregningen af w_2 , og så ortogonalitet mellem v_1 og v_2 bliver ikke sikret.

Til gengæld for den anden version af udregning bliver $w_1 = (0, -s, s, 0)$ og $x_2 = (0, -s, 0, s)$, som kun indeholde led af størrelsesorden s . Beregning af w_2 sikre nu en vektor (næsten) vinkelret på w_1 (og v_1).

15.2 Den forbedrede Gram-Schmidt proces

For at implementere den overstående ide, lad os starte med u_0, u_1, \dots, u_{k-1} lineært uafhængige. Så går vi i gang med at konstruere ortonormale v_0, v_1, \dots, 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, \dots, v_{r-2} . Dette udmøntes i den følgende.

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

- 1 Sæt $w_i^{(0)} = u_i, i = 0, \dots, k-1, \quad v_0 = w_0^{(0)} / \|w_0^{(0)}\|.$
- 2 Sæt $w_i^{(1)} = w_i^{(0)} - \text{pr}_{v_0}(w_i^{(0)}), i = 1, \dots, k-1, \quad v_1 = w_1^{(1)} / \|w_1^{(1)}\|.$
- 3 Sæt $w_i^{(2)} = w_i^{(1)} - \text{pr}_{v_1}(w_i^{(1)}), i = 2, \dots, k-1, \quad v_2 = w_2^{(2)} / \|w_2^{(2)}\|.$
- ...
- 4 Sæt $w_i^{(r)} = w_i^{(r-1)} - \text{pr}_{v_{r-1}}(w_i^{(r-1)}), i = r, \dots, k-1, \quad v_r = w_r^{(r)} / \|w_r^{(r)}\|.$
- ...
- 5 **return** v_0, v_1, \dots, v_{k-1} , en ortogonal samling,
der udspænder det samme rum som u_0, u_1, \dots, 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, \dots, u_4 , som er

15.2 DEN FORBEDREDE GRAM-SCHMIDT PROCES

søjlerne i matricen

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

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.

$$[v_0 \mid v_1 \mid v_2 \mid v_3 \mid v_4] = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 \end{bmatrix}$$

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

For den forbedrede Gram-Schmidt process, begynder vi på samme 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

$$[v_0 \mid v_1 \mid v_2 \mid v_3 \mid v_4] = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 \end{bmatrix}.$$

△

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

15 FORBEDRING AF GRAM-SCHMIDT

FORBEDRET GRAM-SCHMIDT(u_0, u_1, \dots, u_{k-1})

```
1 for  $i \in \{0, 1, \dots, k-1\}$ :
2      $w_i = u_i$ 
3 for  $i \in \{0, 1, \dots, k-1\}$ :
4      $r_{ii} = \|w_i\|_2$ 
5      $v_i = w_i / r_{ii}$ 
6     for  $j \in \{i+1, \dots, k-1\}$ :
7          $r_{ij} = v_i^T w_j$ 
8          $w_j = w_j - r_{ij} v_i$ 
```

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, :j].T @ a[:j, [j]]
        w = a[:j, [j]] - q[:j, :j] @ r[:j, [j]]
        r[j, j] = np.linalg.norm(w)
        q[:j, [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 167, af den klassiske Gram-Schmidt process

15.3 KLASSISK KONTRA FORBEDRET GRAM-SCHMIDT

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 bekræfte 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)
```

15 FORBEDRING AF GRAM-SCHMIDT

```
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 singularværdier, nemlig

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

Vi gør dette ved beregne singularværdidekomponeringen af en tilfældig matrix og bruger dens u og v til at danne en ny matrix a :

15.3 KLASSISK KONTRA FORBEDRET GRAM-SCHMIDT

```
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* give 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)

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

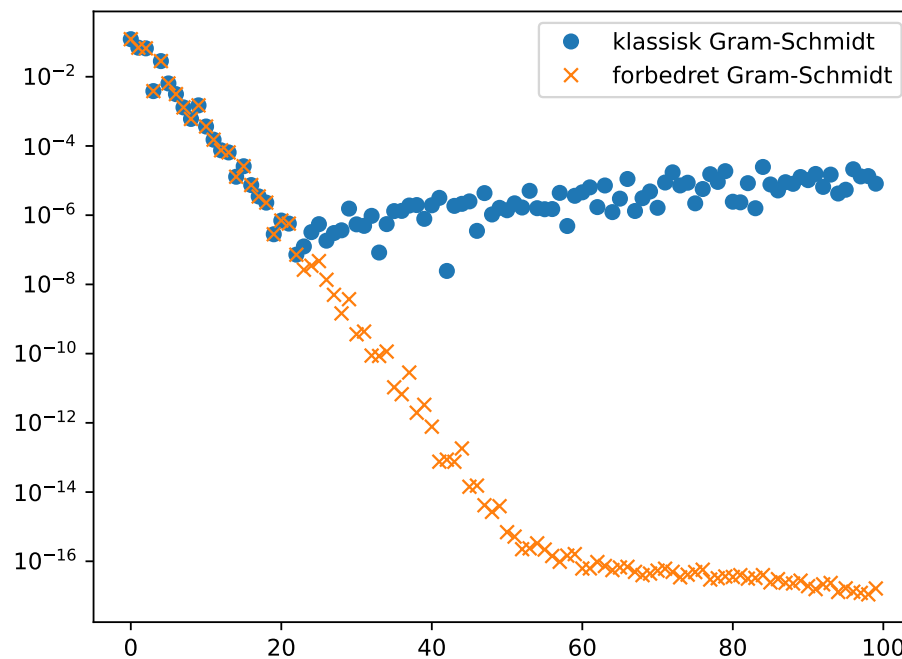
```
print(2.**-55)
```

```
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))
```

15 FORBEDRING AF GRAM-SCHMIDT



Figur 15.1: Klassisk kontra forbedret Gram-Schmidt.

1.4901161193847656e-08

Dette er ret typisk for opførelsen af de to algoritmer.

15.4 Flops

Lad os tælle antallet af **float** operationer forbundet med den forbedrede Gram-Schmidt process. Optællingen for den klassiske er nogenlunde den samme. Vi kikker på algoritmen side 176. Løkken i linjerne 1 og 2 tæller ingen flops. Kikker vi på den anden **for**-løkke, som starter på linje 3, har vi først

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

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

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

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

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

som følger af

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

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.

Kapitel 16

Mindste kvadraters metode

Det er ikke usædvanligt at vi ønsker at estimere nogle parametre i en model. Nogle gange kan dette gøres ved at udføre konkrete eksperimenter og målinger. For at få en god estimering af parametrene tager man gerne så mange målinger, som muligt. Men typisk er disse målinger forbundet med fejl, så de opnåede datapunkter ligger ikke præcis på den kurve der forventes, og de resulterende ligninger er inkonsistent. Den mindste kvadraters metode forsøger at give et bedste bud på disse parametre.

16.1 Problemformulering

Givet et lineært ligningssystem

$$Ax = b \tag{16.1}$$

med flere ligninger end variabler, er der for typisk b ingen løsning. Lad os tage $A \in \mathbb{R}^{m \times n}$, med $m > n$, så $x \in \mathbb{R}^n$ og $b \in \mathbb{R}^m$. Systemet (16.1) kan skrives

$$x_0 a_0 + x_1 a_1 + \cdots + x_{n-1} a_{n-1} = b,$$

hvor a_0, a_1, \dots, a_{n-1} er søjlerne af A . Det følger at systemet har en løsning kun hvis b ligger i søjlerummet $S(A)$ af A .

For generel $b \in \mathbb{R}^m$, betragter vi *restvektoren*

$$r = b - Ax \in \mathbb{R}^m.$$

I stedet for at løse (16.1), arbejder vi på at finde $x \in \mathbb{R}^n$ således at restvektoren er så lille, som muligt. Lad os måle vektorstørrelse via det standard indre produkt.

Så er vores problem: givet $A \in \mathbb{R}^{m \times n}$ og $b \in \mathbb{R}^m$, bestem $x \in \mathbb{R}^n$, som minimerer

$$\|r\|_2 = \|b - Ax\|_2.$$

Da x varierer, giver Ax alle vektorer i søjlerummet $S(A)$. Så $\|r\|_2$ minimeres når Ax er projektionen Pb af b på søjlerummet af A . Så den mindste kvadraters metode er ækvivalent med at løse

$$Ax = Pb. \quad (16.2)$$

Lad os give to forskellig løsningsmetoder.

16.2 Løsning via QR-dekomponering

Givet en tynd QR-dekomponering af $A = QR$, har vi

$$Ax = QRx = Qy = y_0q_0 + y_1q_1 + \cdots + y_{n-1}q_{n-1}$$

hvor q_0, q_1, \dots, q_{n-1} er søjlerne af Q . Lad os antage at $S(Q) = S(A)$. Så er projektion på søjlerummet af A givet ved

$$\begin{aligned} P &= q_0q_0^T + q_1q_1^T + \cdots + q_{n-1}q_{n-1}^T \\ &= QQ^T, \end{aligned}$$

da q_i er ortogonal af længde 1, sammenlign (13.3). Så vi ønsker at løse

$$QRx = QQ^Tb. \quad (16.3)$$

Men Q har ortonormale søjler q_0, q_1, \dots, q_{n-1} . Så dens Grammatrix er $Q^TQ = I_n$. Fra (16.3) har vi $Rx = I_nRx = Q^TRx = Q^TQQ^Tb = I_nQ^Tb = Q^Tb$. Omvendt $Rx = Q^Tb$ medfører $QRx = QQ^Tb$. Så ser vi at (16.3) er det samme som

$$Rx = Q^Tb. \quad (16.4)$$

Bemærk at R er øvre triangulær, så vi kan forvente at løse (16.4) via back substitution.

Vi får

MINDSTE KVADRATER VIA $QR(A, b)$

Krav: søjlerne af A er lineært uafhængig

- 1 Beregn en tynd QR-dekomponering $A = QR$
- 2 Beregn Q^Tb
- 3 Løs $Rx = Q^Tb$ via back substitution

Antallet af flops for denne metode er cirka $2mn^2$ hvis vi beregner QR-dekomponering via den forbedrede Gram-Schmidt proces.

16.3 Løsning via singulærværdidekomponering

En alternative metode er at udnytte singulærværdidekomponering. Husk at SVD dekomponering af $A = U\Sigma V^T \in \mathbb{R}^{m \times n}$ er ækvivalent med

$$A = \sigma_0 u_0 v_0^T + \sigma_1 u_1 v_1^T + \cdots + \sigma_{k-1} u_{k-1} v_{k-1}^T,$$

hvor $\sigma_0 \geq \sigma_1 \geq \cdots \geq \sigma_{k-1} \geq 0$, $k = \min\{m, n\}$.

Lad r være det største tal så at $\sigma_{r-1} > 0$. Så er

$$A = \sigma_0 u_0 v_0^T + \sigma_1 u_1 v_1^T + \cdots + \sigma_{r-1} u_{r-1} v_{r-1}^T$$

og

$$A = U\Sigma V^T$$

med $U \in \mathbb{R}^{m \times r}$, $\Sigma \in \mathbb{R}^{r \times r}$, $V \in \mathbb{R}^{n \times r}$. Dette kaldes den *reducerede* singulærværdidekomponering.

Bemærk at u_0, u_1, \dots, u_{r-1} er nu en ortonormal basis for $S(A)$. Vi kalder *rangen* af A , og observerer at den er lige med antallet af pivotsøjler i echelonform for A .

Givet den reducerede SVD $A = U\Sigma V^T$, er u_0, u_1, \dots, u_{r-1} ortonormal basis for $S(A)$ og projektionen på $S(A)$ er

$$P = UU^T.$$

For de mindste kvadraters metode ønsker vi at løse

$$U\Sigma V^T x = UU^T b$$

Da søjlerne af U er ortonormal er dette ækvivalent med

$$\Sigma V^T x = U^T b$$

Husk at $\Sigma = \text{diag}(\sigma_0, \sigma_1, \dots, \sigma_{r-1})$, med alle $\sigma_i > 0$. Det giver at den inverse er

$$\Sigma^{-1} = \text{diag}(1/\sigma_0, 1/\sigma_1, \dots, 1/\sigma_{r-1}),$$

og vi kan nemt løse $\Sigma y = U^T b$ for et y , som $y = \Sigma^{-1} U^T b$.

Desuden er VV^T projektion på $S(V)$. Så ligningen $y = V^T x$ har en løsning givet som $x = Vy$. Alt i alt har vi

$$x = V\Sigma^{-1}U^T b. \tag{16.5}$$

Det kan bekræftes ved udregningen at

$$\begin{aligned} Ax &= U\Sigma V^T V\Sigma^{-1}U^T b = U\Sigma I_r \Sigma^{-1}U^T b \\ &= UI_r U^T b = UU^T b = Pb, \end{aligned}$$

som ønsket.

Matricen $A^+ = V\Sigma^{-1}U^T$ i (16.5) kaldes *pseudoinversen* til A , og (16.5) er løsningen til $Ax = Pb$, som har $\|x\|_2$ mindst muligt.

MINDSTE KVADRATER VIA SVD(A, b)

- 1 Beregn en reduceret SVD $A = U\Sigma V^T$
- 2 Beregn $U^T b$
- 3 Løs $\Sigma y = U^T b$
- 4 Sæt $x = Vy$

Antallet af flops for denne metode er cirka $2mn^2 + 11n^3$. Så generelt er denne metode langsommere end via QR-dekomponering. Men beregnes sådan en QR-dekomponering via den forbedrede Gram-Schmidt metode er resultatet ikke så præcis, som den fra SVD metoden; vi vil se et eksempel om lidt. Desuden er SVD metoden gyldig når søjlerne af A er ikke nødvendigvis lineært uafhængig, i modsætning til QR-metoden.

16.4 Polynomier

Lad os kikke på et par eksempler hvor vi tilpasser polynomier til given data.

Vi begynder med et simpelt eksempel hvor der er kun 3 datapunkter.

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

x = np.array([0.0, 1.0, 2.0])
y = np.array([0.2, 0.5, 1.2])

fig, ax = plt.subplots()
ax.plot(x, y, 'o')
```

Her har vi indlæst punkterne og plottet dem som den første plot i figur 16.1. Da der er kun 3 punkter, findes der et entydig andengradspolynomium igennem

disse punkter. Dette polynomium kan findes på følgende måde. Vi danner Vandermondematrixen hvis søjler er x -koordinaterne opløftet i potenserne 2, 1 og 0:

```
cols = len(x)
print(cols)

a = np.vander(x, cols)
print(a)
```

```
3
[[0. 0. 1.]
 [1. 1. 1.]
 [4. 2. 1.]]
```

Bestemmelsen af polynomiet $ax^2 + bx + c$, som har de 3 givne y -værdier i de givne punkter, er det samme som at løse det lineære ligningssystem

$$\begin{bmatrix} x_0^2 & x_0 & 1 \\ x_1^2 & x_1 & 1 \\ x_2^2 & x_2 & 1 \end{bmatrix} \begin{bmatrix} a \\ b \\ c \end{bmatrix} = \begin{bmatrix} y_0 \\ y_1 \\ y_2 \end{bmatrix}.$$

Dette kan gøres i python på følgende vis

```
koeffs = np.linalg.solve(a, y[:, np.newaxis])
print(koeffs)
```

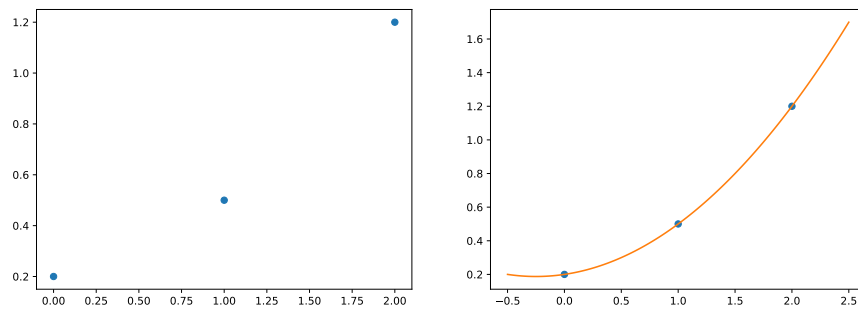
```
[[0.2]
 [0.1]
 [0.2]]
```

Nu kan vi plotte kurven fra andengradspolynomiet

```
t = np.linspace(-0.5, 2.5, 100)

fig, ax = plt.subplots()
ax.plot(x, y, 'o')
ax.plot(t, np.vander(t, cols) @ koeffs)
```

16 MINDSTE KVADRATERS METODE



Figur 16.1: Andengradspolynomium igennem 3 datapunkter.

Resultatet vises til højre i figur 16.1.

Som alternativt kan vi finde en ret linje, dss. polynomium af grad 1, som ligger tæt på den givne punkter, ved hjælp af den mindste kvadraters metode. Dette kaldes også lineær regression. Lad os bruge QR-metoden via den forbedrede Gram-Schmidt.

```
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
```

Vi sætter graden til 1, men det andet argument til `np.vander` er antallet af søjle i Vandermondematricen, som er én mere end graden:

```
cols = 2  
a = np.vander(x, cols)  
print(a)
```

```
[[0. 1.]
 [1. 1.]
 [2. 1.]]
```

QR-dekomponeringen af a beregnes

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

```
[[ 0.          0.91287093]
 [ 0.4472136   0.36514837]
 [ 0.89442719 -0.18257419]]
```

```
[[2.23606798 1.34164079]
 [0.         1.09544512]]
```

Vektoren $c = Q^T b$ er så

```
c = q.T @ y[:, np.newaxis]
print(c)
```

```
[[1.29691943]
 [0.14605935]]
```

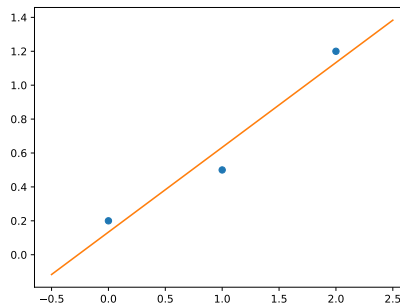
og koefficienterne til førsteordenspolynomiet er nu

```
koeffs = np.linalg.solve(r, c)
print(koeffs)
```

```
[[0.5      ]
 [0.13333333]]
```

En plot af resultatet

16 MINDSTE KVADRATERS METODE



Figur 16.2: Lineær tilnærmelse af data.

```
t = np.linspace(-0.5, 2.5, 100)

fig, ax = plt.subplots()
ax.plot(x, y, 'o')
ax.plot(t, np.vander(t, cols) @ koeffs)
```

gives i figur 16.2.

Denne samme fremgangsmåde kan anvendes på flere datapunkter. Her vil man ofte gerne bruge polynomier af højere grad. Lad os tage et eksempel og først finde et polynomium, der går igennem alle datapunkter.

```
x = np.array([-2.1, -1.9, -1.5, -0.8, -0.3, 0.1,
              0.5, 1.2, 1.3, 1.7, 2.4])
y = np.array([0.1, 0.4, -0.1, -0.6, -0.5, 0.1,
              1.0, 1.3, 0.7, 0.1, 0.2])

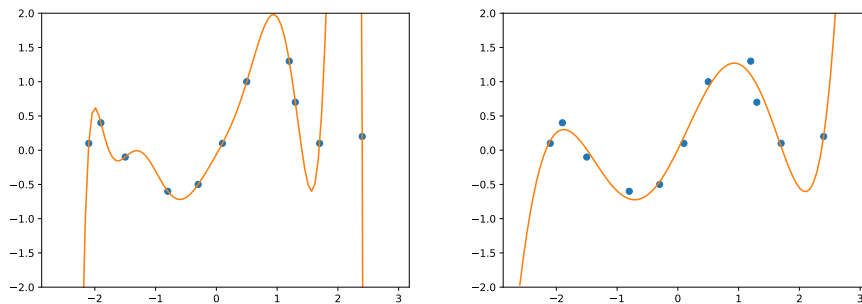
cols = len(x)
print(cols)
```

11

```
a = np.vander(x, cols)
koeffs = np.linalg.solve(a, y[:, np.newaxis])

t = np.linspace(x.min()-0.5, x.max()+0.5, 100)
```

190



Figur 16.3: Højere grad.

```
fig, ax = plt.subplots()
ax.set_ylim(-2.0, 2.0)
ax.plot(x, y, 'o')
ax.plot(t, np.vander(t, cols) @ koeffs)
```

Kikker vi på plottet til venstre i figur 16.3, ses at dette polynomium er en meget dårlig repræsentation af formen af datapunkterne, den stræber for meget efter at gå igennem hver eneste punkt. Dette kaldes *over fitting*. Reduceres graden af polynomiet, og bruges de mindste kvadraters metode

```
cols = 7
a = np.vander(x, cols)
q, r = forbedret_gram_schmidt(a)
c = q.T @ y[:, np.newaxis]
koeffs = np.linalg.solve(r, c)

fig, ax = plt.subplots()
ax.set_ylim(-2.0, 2.0)
ax.plot(x, y, 'o')
ax.plot(t, np.vander(t, cols) @ koeffs)
```

fås en bedre gengivelse, se plottet til højre i figur 16.3. Afvigelserne af dette polynomium fra de givne datapunkter ligger i restvektoren

```
rest = y[:, np.newaxis] - np.vander(x, cols) @ coeffs
print(rest)
```

```
[[-0.04036569]
 [ 0.09963581]
 [-0.11044335]
 [ 0.11108815]
 [-0.03960429]
 [-0.09563144]
 [ 0.08603725]
 [ 0.19850296]
 [-0.25445747]
 [ 0.04795903]
 [-0.00272098]]
```

Et godt mål for den samlede afvigelsen er normen

```
print(np.linalg.norm(rest))
```

```
0.400837595209125
```

Beregningen via SVD i stedet for QR-dekomponering, er givet nedenfor. I dette tilfælde er resultatet meget tæt på den forrige fra QR-metoden, så vi nøjes med at give længden af restvektoren.

```
u, s, vt = np.linalg.svd(a, full_matrices=False)
coeffs_svd = vt.T @ (np.diag(1/s) @ (u.T @ y[:, np.newaxis]))
rest_svd = y[:, np.newaxis] - np.vander(x, cols) @ coeffs_svd
print(np.linalg.norm(rest_svd))
```

```
0.4008375952091247
```

Differencen mellem restvektorene for de to metoder har længde

```
print(np.linalg.norm(rest_svd - rest))
```

```
1.9638163267674327e-14
```

i dette eksempel.

16.5 Løsning via normalligninger

Der er en tredje løsningsmetode for det mindste kvadraters problem, som burde nævnes, nemlig via normalligningerne. Der er mange tekster, som giver kun denne løsningsmetode, men i det næste kapitel vil vi se at den er numerisk meget upræcist. Så i praksis er metoderne ovenfor altid at foretrække.

Vi antager at $A \in \mathbb{R}^{m \times n}$, $m \geq n$, har lineært uafhængige søjler. Hvis A har reduceret SVD $A = U\Sigma V^T$, så har vi at $r = \text{rang}(A) = n$ og at

$$\begin{aligned} A^T A &= (U\Sigma V^T)^T (U\Sigma V^T) = V\Sigma^T U^T U\Sigma V^T \\ &= V\Sigma_r^T \Sigma V^T = V\Sigma^2 V^T \end{aligned}$$

er invertibel med invers $V\Sigma^{-2}V^T$. Dette giver

$$\begin{aligned} (A^T A)^{-1} A^T &= V\Sigma^{-2}V^T V\Sigma U^T = V\Sigma^{-2}I_n \Sigma U^T \\ &= V\Sigma^{-1}U^T = A^+. \end{aligned}$$

Løsning af den mindste kvadraters problem $Ax = Pb$ for A med lineært uafhængige søjler er ækvivalent med løsning af

$$A^T Ax = A^T b,$$

som kaldes *normalligningerne*. Som nævnt ovenfor er denne metode dog numerisk upræcist og anbefales ikke.

Kapitel 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

$$Ax = b$$

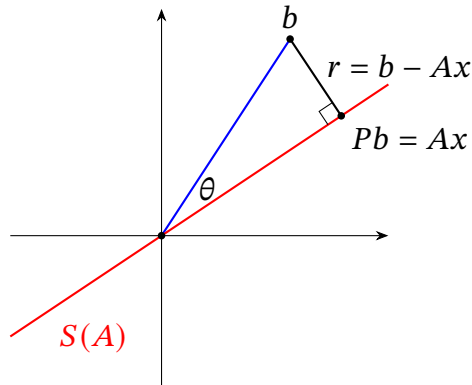
hvor $A \in \mathbb{R}^{m \times n}$, $m > n$, og $b \in \mathbb{R}^m$ er givet. Som vi har set, er dette ækvivalent med at løse

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

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.

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



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

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 give 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 ,

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

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

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

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

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

17.2 Et eksempel

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

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

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 $f(t_i)$, $i = 0, \dots, 99$.

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

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

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 κ_a , konditionstallet $\kappa(a)$ 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 $\kappa(a)$ 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])
```

1.0000000068599447

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 bestemme 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 $A^T A x = A^T b$.

```
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 -dekomponering $A = QR \in \mathbb{R}^{n \times k}$ 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 nok 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

$$H = I_n - svv^T$$

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

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

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

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

$$\begin{aligned}
 A \mapsto H_0 A &= \begin{bmatrix} r_{00} & r_{01} & \cdots \\ 0 & * & \cdots \\ \vdots & \vdots & \\ 0 & * & \cdots \end{bmatrix} \mapsto H_1 H_0 A = \begin{bmatrix} r_{00} & r_{01} & r_{02} & \cdots \\ 0 & r_{11} & r_{12} & \cdots \\ 0 & 0 & * & \cdots \\ \vdots & \vdots & \vdots & \\ 0 & 0 & * & \cdots \end{bmatrix} \\
 \mapsto H_2 H_1 H_0 A &= \begin{bmatrix} r_{00} & r_{01} & r_{02} & r_{03} & \cdots \\ 0 & r_{11} & r_{12} & r_{13} & \cdots \\ 0 & 0 & r_{22} & r_{23} & \cdots \\ 0 & 0 & 0 & * & \cdots \\ \vdots & \vdots & \vdots & \vdots & \\ 0 & 0 & 0 & * & \cdots \end{bmatrix} \\
 \cdots \mapsto H_{k-1} \cdots H_1 H_0 A &= \begin{bmatrix} r_{00} & r_{01} & r_{02} & r_{03} & \cdots \\ 0 & r_{11} & r_{12} & r_{13} & \cdots \\ 0 & 0 & r_{22} & r_{23} & \cdots \\ 0 & 0 & 0 & r_{33} & \cdots \\ \vdots & \vdots & \vdots & & \ddots \\ 0 & 0 & 0 & \cdots & \cdots \end{bmatrix} = R
 \end{aligned}$$

Vi sætter så

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

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  $\varepsilon = \begin{cases} -1 & \text{for } u_0 \geq 0 \\ +1 & \text{for } u_0 < 0 \end{cases}$ 
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 skalar 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 skalarerne 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:

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

HOUSEHOLDER QR DATA(A)

- 1 Beregn Householder v, s for
- 2 søjle 0 i A
- 3 søjle 0 i $(H_0 A)_{[1:,1:]}$
- 4 søjle 0 i $(H_1 H_0 A)_{[2:,2:]}$
- 5 ...
- 6 Gem v data under diagonalen i $R = H_{k-1} \dots H_1 H_0 A$

Når vi implementerer dette i python må vi gerne huske at beregne $HA = (I_n - svv^T)A$, som $HA = A - sv(v^T A)$. 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 $(H_{j-1} \dots H_1 H_0 A)_{[j:,j:]}$. F.eks. for $j = 2$, har vi

$$\text{data} = \begin{bmatrix} r_{00} & r_{01} & r_{02} & \dots & r_{0,k-1} \\ (v_0)_1 & r_{11} & r_{12} & \dots & r_{1,k-1} \\ (v_0)_2 & (v_1)_1 & (H_1 H_0 A)_{22} & \dots & (H_1 H_0 A)_{2,k-1} \\ \vdots & \vdots & & \ddots & \vdots \\ (v_0)_{n-1} & (v_1)_{n-2} & (H_1 H_0 A)_{n-1,2} & \dots & (H_1 H_0 A)_{n-1,k-1} \end{bmatrix}.$$

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

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

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

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

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 effektivt at beregne det som

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

Grunden til dette ligger i at $H_i B$ 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)
```

17 MINDSTE KVADRATER: KONDITIONSTAL OG HOUSEHOLDER

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

```
q = [[-1.000000000e+00  7.07106781e-09  4.08248290e-09]
      [-1.000000000e-08 -7.07106781e-01 -4.08248290e-01]
      [ 0.000000000e+00  7.07106781e-01 -4.08248290e-01]
      [ 0.000000000e+00  0.000000000e+00  8.16496581e-01]]

r = [[-1.000000000e+00 -1.000000000e+00 -1.000000000e+00]
      [ 0.000000000e+00  1.41421356e-08  7.07106781e-09]
      [ 0.000000000e+00  0.000000000e+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 $\langle v_1, v_2 \rangle$ havde størrelsesorden 10^{-16} , og $\langle v_0, v_1 \rangle$ og $\langle v_0, v_2 \rangle$ var størrelsesorden 10^{-8} . Vi har også at $q @ r$ er pænt tæt på a_s :

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

```
[[ 0.000000000e+00  0.000000000e+00  0.000000000e+00]
 [ 0.000000000e+00 -1.65436123e-24 -1.65436123e-24]
 [ 0.000000000e+00  1.65436123e-24  8.27180613e-25]
 [ 0.000000000e+00  0.000000000e+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)
```

```
[[1.000000000e+00 0.000000000e+00 0.000000000e+00]
 [0.000000000e+00 1.000000000e+00 5.55111512e-17]
 [0.000000000e+00 5.55111512e-17 1.000000000e+00]]
[[ 0.000000000e+00  0.000000000e+00  0.000000000e+00]
 [ 0.000000000e+00  3.30872245e-24  4.13590306e-24]
 [ 0.000000000e+00 -1.65436123e-24 -1.65436123e-24]
 [ 0.000000000e+00  0.000000000e+00  0.000000000e+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 er ikke så svært at implementere back substitution selv, men vi overlader det, som en opgave for læseren.

Kapitel 18

Lineære afbildninger og matricer

18.1 Lineære afbildninger

Et vektorrum V har to fundamentale operationer: sum $u + v$ og skalar multiplikation sv . En lineær transformation, eller lineær afbildning, er en funktion, der respekterer disse operationer.

Definition 18.1. En *lineær transformation* er en afbildning $L: V \rightarrow W$ fra et vektorrum V til et vektorrum W således at

(a) $L(u + v) = L(u) + L(v)$ og

(b) $L(sv) = sL(v)$

for alle $u, v \in V$ og alle skalarer s .

Eksempel 18.2. Afbildningen $L: \mathbb{R} \rightarrow \mathbb{R}$, $L(x) = 5x$ er lineær, da

$$L(x + y) = 5(x + y) = 5x + 5y = L(x) + L(y),$$

$$L(sx) = 5(sx) = s5x = sL(x)$$

for alle $x, y, z \in \mathbb{R}$. △

Eksempel 18.3. Afbildningen $F: \mathbb{R} \rightarrow \mathbb{R}$, $F(x) = x^2$ er *ikke* lineær. Dette vises bedst ved at finde konkrete elementer hvor den ene af del (a) eller del (b) i definition 18.1 holder ikke. For denne afbildning F kan vi kikke på del (a) ved $u = 1 = v$. Vi har

$$F(1 + 1) = F(2) = 4,$$

$$F(1) + F(1) = 1 + 1 = 2,$$

så

$$F(1+1) = 4 \neq 2 = F(1) + F(1)$$

og del (a) er ikke opfyldt. Δ

Eksempel 18.4. Lad V være rummet af funktioner $[-1, 2] \rightarrow \mathbb{R}$, som er to gange differentiable, og lad W være rummet af en gang differentiable funktioner på det samme interval. Så er afbildning

$$L: V \rightarrow W, \quad L(f) = f'$$

lineær. Dette er konsekvens af de almindelige regneregler for den afledte: $(f+g)' = f' + g'$ og $(cf)' = cf'$, for c konstant. Δ

Eksempel 18.5. Lad V være rummet af kontinuerte funktioner $[0, 3] \rightarrow \mathbb{R}$. Så er afbildningerne

$$L: V \rightarrow V, \quad L(f) = \int_0^x f(t) dt,$$

$$M: V \rightarrow \mathbb{R}, \quad M(f) = \int_0^3 f(t) dt$$

begge lineære. Igen er dette konsekvens af standard regneregler for integraler: $\int_0^x f(t) + g(t) dt = \int_0^x f(t) dt + \int_0^x g(t) dt$, og $\int_0^x cf(t) dt = c \int_0^x f(t) dt$ for c konstant. Δ

Eksempel 18.6. Forholdet mellem elektrisk spænding V over og elektrisk strøm I igennem flere komponenter er lineær:

(a) Elektrisk modstand R :

$$I \mapsto V = RI, \quad V \mapsto I = \frac{1}{R}V;$$

(b) Elektrisk kondensator, kapacitans C :

$$I(t) \mapsto V(t) - V(t_0) = \frac{1}{C} \int_{t_0}^t I(s) ds, \quad V(t) \mapsto I(t) = C \frac{dV}{dt};$$

(c) Induktionsspole, induktans L :

$$I(t) \mapsto V(t) = L \frac{dI}{dt}, \quad V(t) \mapsto I(t) - I(t_0) = \frac{1}{L} \int_{t_0}^t V(s) ds.$$

Dette gør lineær algebra særligt relevant for analyse af elektriske kredsløb. \triangle

Eksempel 18.7. $L: \mathbb{R}^n \rightarrow \mathbb{R}$

$$L \begin{pmatrix} x_0 \\ x_1 \\ x_2 \\ \vdots \\ x_{n-1} \end{pmatrix} = x_0 + 2x_1 + 3x_2 + \cdots + nx_{n-1}$$

er lineær. \triangle

Eksempel 18.8. Givet $A \in \mathbb{R}^{m \times n}$, $L: \mathbb{R}^n \rightarrow \mathbb{R}^m$

$$L(x) = Ax$$

er lineær, da

$$A(x + y) = Ax + Ay \quad \text{og} \quad A(sx) = sAx$$

\triangle

Bemærkning 18.9. Der skal noteres at notationen i definition 18.1 gemmer et par væsentlige detaljer. I del (a) er de to plustegn »+« fra forskellige vektorrum: den på den venstre side er sumoperationen i V ; den på den højre side af ligningen er sumoperationen i W . Ligeledes er de to skalarmultiplikationer i del (b) fra V og henholdsvis W . \diamond

Eksempel 18.10. For V rummet af funktioner $[0, 1] \rightarrow \mathbb{R}$, er afbildningen

$$f \mapsto f(0)$$

lineær, da

$$f + g \mapsto (f + g)(0) = f(0) + g(0)$$

og

$$sf \mapsto (sf)(0) = s f(0)$$

fra vores definition af operationerne i V . \triangle

Lad os fremhæve to væsentlige egenskaber af lineære afbildninger.

Proposition 18.11. For $L: V \rightarrow W$ lineær gælder

(a) $L(0) = 0$,

(b) $L(su + tv) = sL(u) + tL(v)$.

Bevis. For (a) har vi $L(0) = L(0v) = 0L(v) = 0$, hvor vi har brugt definition 18.1(b).

Del (b) følger fra brug af begge egenskaber af L : $L(su + tv) = L(su) + L(tv) = sL(u) + tL(v)$. \square

Eksempel 18.12. Det følger at $F(x) = x + 1$ er ikke lineær, da $F(0) = 1 \neq 0$. \triangle

Del (b) af proposition 18.11 giver mere generelt

$$\begin{aligned} L(s_0v_0 + s_1v_1 + \cdots + s_{k-1}v_{k-1}) \\ = s_0L(v_0) + s_1L(v_1) + \cdots + s_{k-1}L(v_{k-1}). \end{aligned} \quad (18.1)$$

18.2 Matrix af en lineær transformation

Eksempel 18.8 er faktisk typisk for lineære afbildninger $\mathbb{R}^n \rightarrow \mathbb{R}^m$.

Proposition 18.13. For $L: \mathbb{R}^n \rightarrow \mathbb{R}^m$ lineær, definér $A = [u_0 \mid u_1 \mid \cdots \mid u_{n-1}] \in \mathbb{R}^{m \times n}$ ved

$$u_j = L(e_j)$$

hvor $e_j = (0, \dots, 0, 1, 0, \dots, 0)$ er den j 'te standard basisvektor, med j 'te indgang 1 og alle andre indgange 0.

Så gælder

$$L(x) = Ax \quad \text{for alle } x \in \mathbb{R}^n.$$

Definition 18.14. Matricen A i proposition 18.13 kaldes *den standard matrixrepræsentation* (SMR) af L .

Bevis for proposition 18.13. For $x \in \mathbb{R}^n$, har vi

$$x = \begin{bmatrix} x_0 \\ x_1 \\ \vdots \\ x_{n-1} \end{bmatrix} = x_0e_0 + x_1e_1 + \cdots + x_{n-1}e_{n-1}$$

Ved brug af (18.1), får vi

$$\begin{aligned} L(x) &= L(x_0e_0 + x_1e_1 + \cdots + x_{n-1}e_{n-1}) \\ &= x_0L(e_0) + x_1L(e_1) + \cdots + x_{n-1}L(e_{n-1}) \\ &= x_0u_0 + x_1u_1 + \cdots + x_{n-1}u_{n-1} \\ &= [u_0 \mid u_1 \mid \cdots \mid u_{n-1}]x = Ax. \end{aligned}$$

Dvs. $L(x) = Ax$ for alle $x \in \mathbb{R}^n$, som ønsket. \square

Eksempel 18.15. $L((x, y)) = (2x - y, y + x, y - 2x)$ har

$$u_0 = L(e_0) = L\left(\begin{bmatrix} 1 \\ 0 \end{bmatrix}\right) = \begin{bmatrix} 2 \\ 1 \\ -2 \end{bmatrix}$$

og

$$u_1 = L(e_1) = L\left(\begin{bmatrix} 0 \\ 1 \end{bmatrix}\right) = \begin{bmatrix} -1 \\ 1 \\ 1 \end{bmatrix}$$

Så L har SMR

$$A = [u_0 \mid u_1] = \begin{bmatrix} 2 & -1 \\ 1 & 1 \\ -2 & 1 \end{bmatrix}.$$

\triangle

18.3 Kombinationer af lineære afbildninger

Lineære afbildninger kan kombineres på forskellige vis til at danne nye lineære afbildninger. Dette betyder man kan bygge komplicerede lineære afbildninger op fra simple eksempler.

Proposition 18.16. Hvis $L_0, L_1: V \rightarrow W$ er lineære og s er en skalar, så er

- (a) $L_0 + L_1$ og
- (b) sL_0

lineære transformationer.

Bevis. Husk at $(L_0 + L_1)(v)$ defineres til at være $L_0(v) + L_1(v)$. For del (a), har vi

$$\begin{aligned} (L_0 + L_1)(u + v) &= L_0(u + v) + L_1(u + v) = L_0(u) + L_0(v) + L_1(u) + L_1(v) \\ &= (L_0 + L_1)(u) + (L_0 + L_1)(v) \end{aligned}$$

og

$$\begin{aligned} (L_0 + L_1)(sv) &= L_0(sv) + L_1(sv) = sL_0(v) + sL_1(v) \\ &= s(L_0(v) + L_1(v)) = s(L_0 + L_1)(v). \end{aligned}$$

Redegørelsen for del (b) er tilsvarende, med udgangspunkt i $(sL)(v) = s(L(v))$. \square

Eksempel 18.17. Afbildninger $f \mapsto f'$ og $f \mapsto f(0)$ er lineære, så det følger at $f \mapsto 3f' - 2f(0)$ er også lineær. Δ

Proposition 18.18. Hvis $L: V \rightarrow W$ og $M: U \rightarrow V$ er lineære afbildninger så er

$$L \circ M: U \rightarrow W$$

lineær.

Bevis. Her er $(L \circ M)(u) = L(M(u))$. Vi har

$$\begin{aligned}(L \circ M)(u_0 + u_1) &= L(M(u_0 + u_1)) = L(M(u_0) + M(u_1)) \\ &= L(M(u_0)) + L(M(u_1)) = (L \circ M)(u_0) + (L \circ M)(u_1),\end{aligned}$$

og en tilsvarende begrundelse gælder for $(L \circ M)(su) = s(L \circ M)(u)$. \square

Eksempel 18.19. Da $f \mapsto f'$ er lineære er $f \mapsto f'' = (f')'$ også lineær. Δ

18.4 Kerne og billedmængde

Lineære afbildninger bevarer de to fundamental operationer i vektorrum. Ligeledes er underrum også defineret ud fra disse to operationer. Dette fører til at lineære afbildninger sender underrum til underrum og at der er to særlige underrum, som en lineær afbildning giver anledning til.

Definition 18.20. Lad $L: V \rightarrow W$ være lineær. Så er *kernen* af L

$$\ker L = \{v \in V \mid L(v) = 0\},$$

mængden af alle vektorer i V , som L afbildninger i $0 \in W$.

Billedmængden af L er

$$\operatorname{im} L = L(V) = \{L(v) \mid v \in V\} \subset W$$

mængden af alle værdier af L .

Eksempel 18.21. For $L: \mathbb{R}^n \rightarrow \mathbb{R}^m$ med SMR A kender vi disse mængder i forvejen, blot med andre navne. Billedmængden er

$$\operatorname{im} L = S(A)$$

søjlerummet af A . Kernen

$$\ker L = N(A) = \{x \in \mathbb{R}^n \mid Ax = 0\}$$

er rummet af løsninger til $Ax = 0$. Dette kaldes *nulrummet* af A . Δ

Proposition 18.22. *Lad $L: V \rightarrow W$ være lineær. Så er $\ker L \subset V$ et underrum af V og $\operatorname{im} L \subset W$ er et underrum af W .*

Bevis. Vi begynder med $\ker L$. Vi skal tjekke at de tre betingelser i definition 12.11 er opfyldt.

Først ser vi at $\ker L$ er ikke tom, da $L(0) = 0$ giver $0 \in \ker L$. Bemærk nu at $u, v \in \ker L$ betyder $L(u) = 0 = L(v)$. Da gælder

$$L(u + v) = L(u) + L(v) = 0 + 0 = 0,$$

som siger at $u + v \in \ker L$. Desuden har vi

$$L(sv) = sL(v) = s0 = 0$$

så $sv \in \ker L$. Dermed er $\ker L$ et underrum af V .

Diskussionen for billedrummet er tilsvarende, men kræver lidt andre overvejelser. Først $\operatorname{im} L$ er ikke tom da $L(0) = 0$ giver $0 \in \operatorname{im} L$. Vi har at $w, y \in \operatorname{im} L$ betyder at der findes $u, v \in V$ med $L(u) = w$ og $L(v) = y$. Da gælder

$$w + y = L(u) + L(v) = L(u + v),$$

som siger $w + y \in \operatorname{im} L$. Desuden har vi

$$sy = sL(v) = L(sv)$$

så $sy \in \operatorname{im} L$, og $\operatorname{im} L$ er et underrum af W . □

Eksempel 18.23. Lad P være projektion på $\operatorname{Span}\{v_0, v_1, \dots, v_{k-1}\}$ for v_0, \dots, v_{k-1} ortonormal:

$$P(u) = \langle v_0, u \rangle v_0 + \langle v_1, u \rangle v_1 + \dots + \langle v_{k-1}, u \rangle v_{k-1}$$

Vi har $\operatorname{im} P = \operatorname{Span}\{v_0, v_1, \dots, v_{k-1}\}$, da $P(v_i) = v_i$ medfører at $v_i \in \operatorname{im} P$, og da hvert element af $\operatorname{im} P$ er en lineær kombination af v_0, v_1, \dots, v_{k-1} .

Kernen $\ker P$ består af de $v \in V$ med $\langle v_i, v \rangle = 0$, for alle $i = 0, 1, \dots, k-1$. △

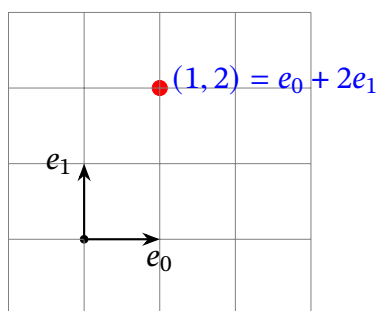
Kapitel 19

Koordinater

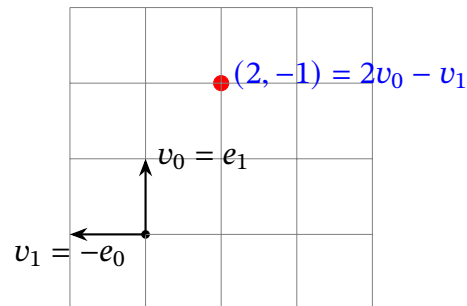
Lad os betragte et punkt i planen, som i figur 19.1. Med hensyn til det sædvanlige koordinatsystem, har punktet koordinater $(1, 2) = e_0 + 2e_1$. Dette siger, at vi kan bevæge os fra origo til punktet ved at gå langs en kopi af e_0 og derefter langs to kopier af e_1 . På denne måde er det parret e_0, e_1 , der bestemmer koordinaterne af vores punkt.

I figur 19.2 kikker vi på det samme punkt, men erstatter parret e_0, e_1 med parret $v_0 = e_1, v_1 = -e_0$. Da $e_0 + 2e_1 = -v_1 + 2v_0 = 2v_0 + (-1)v_1$, kommer vi fra origo til punktet ved at bevæge os langs to kopier af v_0 og derefter langs en kopi af $-v_1$. Med hensyn til v_0, v_1 har vores punkt koordinater $(2, -1)$.

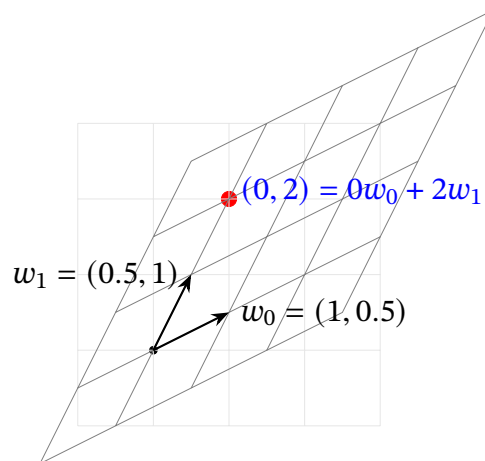
Figur 19.3 viser det samme punkt, men nu arbejder vi ud fra vektorparret w_0, w_1 , hvor $w_0 = (1, 0, 0, 5)$ og $w_1 = (0, 5, 1, 0)$. Da $e_0 + 2e_1 = 2w_1 = 0w_0 + 2w_1$,



Figur 19.1: Standardkoordinater.



Figur 19.2: Drejet ortogonal koordinatsystem.



Figur 19.3: Generel lineært koordinatsystem.

har vores punkt koordinater $(0, 2)$ i koordinatsystemet bestemt af w_0, w_1 .

Vi ser at det samme punkt har forskellige beskrivelser afhængig af hvilke vektorpar vi bruger til at danne koordinater. Hvilke par vektorer må vi bruge? Her i planen er det rimelig klart at begge vektor skal være forskellig fra 0 og at de må ikke være parallelle. Dette er faktisk også tilstrækkeligt. I højere dimensioner er der lidt flere krav der skal stilles: disse dækkes af begrebet »basis«, som vi vil nu studere.

19.1 Basis

Lad os lige huske et par definitioner vi har mødt før. Først er en samling vektorer v_0, \dots, v_{n-1} lineært uafhængig hvis

$$s_0 v_0 + s_1 v_1 + \dots + s_{k-1} v_{k-1} = 0 \quad \text{medfører} \quad s_0 = 0 = s_1 = \dots = s_{k-1}.$$

Rummet udspændt af disse vektorer er $\text{Span}\{v_0, v_1, \dots, v_{n-1}\}$, som består af alle lineære kombinationer

$$s_0 v_0 + s_1 v_1 + \dots + s_{k-1} v_{k-1}$$

for alle skalarer s_i .

Definition 19.1. En samling vektorer v_0, v_1, \dots, v_{n-1} er en *basis* E for V hvis

- (a) v_0, v_1, \dots, v_{n-1} er lineært uafhængig, og
- (b) $\text{Span}\{v_0, v_1, \dots, v_{n-1}\} = V$.

I \mathbb{R}^n har vi nogle gode værktøj for at tjekke om en samling vektorer er faktisk en basis.

Proposition 19.2. Lad v_0, v_1, \dots, v_{k-1} være en samling vektorer i \mathbb{R}^n , og sæt $A = [v_0 \mid v_1 \mid \dots \mid v_{k-1}]$. Så er v_0, v_1, \dots, v_{k-1} en basis for \mathbb{R}^n hvis og kun hvis ligningssystemet

$$Ax = b$$

en entydig løsning for hvert $b \in \mathbb{R}^n$.

Dette tvinger $k = n$ for en basis af \mathbb{R}^n .

Bevis. Husk at $Ax = x_0 v_0 + x_1 v_1 + \dots + x_{k-1} v_{k-1}$. Fra dette kan vi bemærke to ting.

- (i) At betingelsen (a), at samlingen er lineært uafhængig, er ens betydende med at $Ay = 0$ har $y = 0$, som entydig løsning.
- (ii) At betingelsen (b), at samlingen udspænder $V = \mathbb{R}^n$, er ens betydende med at $Ax = b$ har mindst en løsning.

Antag at $Ax = b$ har en entydig løsning for hvert b . At vi har en løsning for hvert b , giver (ii) at samlingen udspænder \mathbb{R}^n . Tages $b = 0$, har vi en entydig løsning $x = 0$. Så fra (i) har vi at samlingen er også lineært uafhængig. Dermed er samlingen en basis for \mathbb{R}^n .

Omvendt hvis samlingen er en basis, har vi fra (ii) eksistens af løsninger til $Ax = b$ for hvert b , og ved fra (i) at løsningen er entydigt i tilfældet $b = 0$.

Vi mangler så at vise, at for vilkårlig b er løsningen til $Ax = b$ entydig. Antag at vi har to løsninger x og z for samme b . Så $Ax = b$ og $Az = b$. Så har vi $A(x - z) = Ax - Az = b - b = 0$. Da $Ay = 0$ har kun løsningen $y = 0$, konkluderer vi at $x - z = 0$. Dette siger at $x = z$, så løsningen på $Ax = b$ er entydig.

For at vise at $k = n$, udelukker vi de andre muligheder.

Hvis $k > n$, så har A flere søjler end pivotelementer. Det betyder at systemet $Ax = b$ med $b = 0$ har frie variabler, og dermed er løsninger ikke entydig. Så samlingen er ikke lineært uafhængig, og dermed ikke en basis.

Hvis $k < n$, er der flere rækker i A end pivotelementer. Vi kan derfor finde b så at $Ax = b$ er ikke konsistent. Så har systemet ingen løsning i dette tilfælde, og dermed at b ikke ligger i $\text{Span}\{v_0, v_1, \dots, v_{n-1}\}$. Dvs. $\text{Span}\{v_0, v_1, \dots, v_{n-1}\} \neq V$, så samlingen er ikke en basis. \square

Eksempel 19.3. Standard basis for \mathbb{R}^n er e_0, e_1, \dots, e_{n-1} . For at se at dette er en basis, bemærk at den tilsvarende matrix A er I_n . Men systemerne $I_n x = b$, $b \in \mathbb{R}^n$, har altid den entydige løsning $x = b$, så søjlerne af I_n udgør en basis for \mathbb{R}^n . \triangle

I tilfældet hvor vi har n vektorer i \mathbb{R}^n er der nu mange forskellige måder at afgøre om de udgør en basis.

Proposition 19.4. Givet v_0, v_1, \dots, v_{n-1} i \mathbb{R}^n sæt $A = [v_0 \mid v_1 \mid \dots \mid v_{n-1}] \in \mathbb{R}^{n \times n}$. Så er udsagnet

- (a) v_0, v_1, \dots, v_{n-1} er en basis for \mathbb{R}^n
- ækvivalent med hver af de følgende
- (b) hver søjle af A er en pivotsøjle,
- (c) A er invertibel,
- (d) Gram-Schmidt på v_0, v_1, \dots, v_{n-1} kan gennemføres til at give en ortogonal samling bestående af n vektorer.

Bevis. Når vi har n vektorer i \mathbb{R}^n er matricen A kvadratisk. Derfor er hvert søjle en pivotsøjle hvis og kun hvis der er ingen frie variabler. Dette er ens betydende med at $Ax = b$ har altid en entydig løsning, så proposition 19.4 giver at samlingen er en basis. Dermed er del (a) ækvivalent med del (b).

Hvis A er invertibel har $Ax = b$ den entydige løsning $x = A^{-1}b$, så v_0, v_1, \dots, v_{n-1} er en basis. Omvendt, hvis vi har del (a), så er der entydige løsninger w_0, w_1, \dots, w_{n-1} til ligningerne $Aw_i = e_i$, $i = 0, 1, \dots, n-1$. Vi kan sætte $B =$

$[w_0 \mid w_1 \mid \dots \mid w_{n-1}]$ og ser at

$$AB = [Aw_0 \mid Aw_1 \mid \dots \mid Aw_{n-1}] = [e_0 \mid e_1 \mid \dots \mid e_{n-1}] = I_n.$$

Det følger fra sætning 7.2 at A er invertibel, med invers $A^{-1} = B$. Vi har så del (c).

Hvis Gram-Schmidt processen kan ikke gennemføres, er det fordi en vektor er en lineær kombination af de forrige. Vektorerne er dermed ikke lineært uafhængig og den første betingelse for basis er ikke opfyldt. Omvendt hvis Gram-Schmidt processen kan gennemføres, er enhver vektor i \mathbb{R}^n en lineær kombination af de ortogonale vektorer konstrueret, og dermed en lineær kombination af de oprindelige vektorer. Det følger at $Ax = b$ har altid en løsning, så hver række indeholder et pivotelement. Men A er kvadratisk, så hvert søjler også indeholder et pivotelement. Dette giver ækvivalens af del (d) med de andre udsagn. \square

Eksempel 19.5. For (2×2) -matricer, kan vi tjekke invertibilitet via determinanten. Så f.eks.

$$\begin{bmatrix} 1 & 0,5 \\ 0,5 & 1 \end{bmatrix}$$

har determinant $1 \times 1 - 0,5 \times 0,5 = 1 - 0,25 = 0,75 \neq 0$, giver at matricen er invertibel, og dermed at søjlerne udgør en basis for \mathbb{R}^2 . \triangle

19.2 Koordinater

Lad $E: v_0, v_1, \dots, v_{n-1}$ være en basis for et vektorrum V . Enhver vektor $b \in V$ kan skrives som

$$b = x_0 v_0 + x_1 v_1 + \dots + x_{n-1} v_{n-1} \quad (19.1)$$

for nogle skalarer x_i , da definition 19.1(b) sikrer en løsning. Disse skalarer er endda entydig, som konsekvens af definition 19.1(a): hvis z_0, \dots, z_{n-1} er en anden løsning, har vi

$$\begin{aligned} 0 &= b - b = (x_0 v_0 + x_1 v_1 + \dots + x_{n-1} v_{n-1}) - (z_0 v_0 + z_1 v_1 + \dots + z_{n-1} v_{n-1}) \\ &= (x_0 - z_0) v_0 + (x_1 - z_1) v_1 + \dots + (x_{n-1} - z_{n-1}) v_{n-1}. \end{aligned}$$

Men lineært uafhængighed giver nu at $x_i - z_i = 0$ for alle i . Dvs. $z_i = x_i$ for alle i , og løsningen er entydig.

Definition 19.6. Lad $E: v_0, v_1, \dots, v_{n-1}$ være en basis for V . En vektor $b \in V$ har *koordinatvektor* $[b]_E$ mht. E givet ved

$$[b]_E = \begin{bmatrix} x_0 \\ x_1 \\ \vdots \\ x_{n-1} \end{bmatrix},$$

hvor x_0, x_1, \dots, x_{n-1} er den entydig løsning til (19.1).

Eksempel 19.7. Betragt den følgende basis for \mathbb{R}^2 , $E: v_0, v_1$, hvor

$$v_0 = \begin{bmatrix} 3 \\ 1 \end{bmatrix}, \quad v_1 = \begin{bmatrix} -1 \\ 2 \end{bmatrix}.$$

Dette er en basis, da matricen $\begin{bmatrix} 3 & -1 \\ 1 & 2 \end{bmatrix}$ har determinant $3 \times 2 - (-1) \times 1 = 6 + 1 = 7 \neq 0$.

Lad os bestemme koordinatvektoren for $b = (1, -1)$ med hensyn til basen E . Vi skal løse

$$x_0 \begin{bmatrix} 3 \\ 1 \end{bmatrix} + x_1 \begin{bmatrix} -1 \\ 2 \end{bmatrix} = \begin{bmatrix} 1 \\ -1 \end{bmatrix},$$

som er det samme som at løse det lineære ligningssystem

$$\begin{bmatrix} 3 & -1 \\ 1 & 2 \end{bmatrix} \begin{bmatrix} x_0 \\ x_1 \end{bmatrix} = \begin{bmatrix} 1 \\ -1 \end{bmatrix}.$$

Dette kan gøres vi rækkeoperationer på den tilsvarende udvidede matrix

$$\begin{aligned} \left[\begin{array}{cc|c} 3 & -1 & 1 \\ 1 & 2 & -1 \end{array} \right] &\sim_{R_0 \leftrightarrow R_1} \left[\begin{array}{cc|c} 1 & 2 & -1 \\ 3 & -1 & 1 \end{array} \right] \sim_{R_1 \rightarrow R_1 - 3R_0} \left[\begin{array}{cc|c} 1 & 2 & -1 \\ 0 & -7 & 4 \end{array} \right] \\ &\sim_{R_1 \rightarrow (-1/7)R_1} \left[\begin{array}{cc|c} 1 & 2 & -1 \\ 0 & 1 & -4/7 \end{array} \right] \sim_{R_0 \rightarrow R_0 - 2R_1} \left[\begin{array}{cc|c} 1 & 0 & 1/7 \\ 0 & 1 & -4/7 \end{array} \right], \end{aligned}$$

så $x_0 = 1/7$, $x_1 = -4/7$ og

$$[b]_E = \begin{bmatrix} x_0 \\ x_1 \end{bmatrix} = \begin{bmatrix} 1/7 \\ -4/7 \end{bmatrix}.$$

△

19.3 Koordinatskift

Når vi har to forskellige baser for det samme vektorrum, vil man gerne vide hvordan de tilhørende koordinater er relaterede til hinanden. Lad os begynde med at kikke på skift af koordinater i vektorrummet \mathbb{R}^n .

Lad $E: v_0, v_1, \dots, v_{n-1}$ være en basis for \mathbb{R}^n , og sæt $A = [v_0 \mid v_1 \mid \dots \mid v_{n-1}]$. Så er koordinatvektoren $x = [b]_E$ løsningen til systemet

$$Ax = b.$$

Dette er det samme som

$$[b]_E = x = A^{-1}b. \quad (19.2)$$

Hvis $F: w_0, \dots, w_{n-1}$ er en anden basis, sætter vi $B = [w_0 \mid w_1 \mid \dots \mid w_{n-1}]$. Så er koordinatvektoren af b mht. F givet på tilsvarende måde ved

$$[b]_F = B^{-1}b. \quad (19.3)$$

Kombinerer vi ligningerne (19.2) og (19.3), får vi

$$[b]_F = B^{-1}A[b]_E,$$

som giver relationen mellem de to koordinatvektorer for b . Der er derfor naturligt at kalde matricen $S = B^{-1}A$ for *koordinatskiftsmatricen* fra E til F .

Bemærkning 19.8. Det følger at $A = I_n^{-1}A$ er koordinatskiftsmatricen fra E til standardbasen e_0, \dots, e_{n-1} . Desuden har vi at $A^{-1} = A^{-1}I_n$ er koordinatskiftsmatricen fra standardbasen e_0, \dots, e_{n-1} to E \diamond

Lad os nu kikke på et vektorrum af polynomier. For at være konkret, lad være V vektorrummet af polynomier af grad højst 2. Et typisk element i V er så et polynomium

$$p(x) = ax^2 + bx + c,$$

med a, b, c skalarer.

En basis for V er $E: x^2, x, 1$. Vores polynomium p kan skrives, som en lineær kombination af basis elementer fra E via

$$p(x) = ax^2 + bx + c1.$$

Dette giver koordinatvektoren

$$[p]_E = \begin{bmatrix} a \\ b \\ c \end{bmatrix}$$

af p mht. E .

En anden basis for V er $F: 3x^2 - 1, 2x + 2, 3$. Lad os først beregne koordinatskiftet fra F til E . Vi har

$$\begin{aligned}[3x^2 - 1]_E &= (3, 0, -1), \\ [2x + 2]_E &= (0, 2, 2), \\ [3]_E &= (0, 0, 3).\end{aligned}$$

Hvis $[p]_F = (r, s, t)$, betyder dette at

$$p(x) = r(3x^2 - 1) + s(2x + 2) + t(3).$$

Vi har så

$$\begin{aligned}[p]_E &= r[3x^2 - 1]_E + s[2x + 2]_E + t[3]_E \\ &= \left[[3x^2 - 1]_E \mid [2x + 2]_E \mid [3]_E \right] \begin{bmatrix} r \\ s \\ t \end{bmatrix} \\ &= \begin{bmatrix} 3 & 0 & 0 \\ 0 & 2 & 0 \\ -1 & 2 & 3 \end{bmatrix} \begin{bmatrix} r \\ s \\ t \end{bmatrix} = \begin{bmatrix} 3 & 0 & 0 \\ 0 & 2 & 0 \\ -1 & 2 & 3 \end{bmatrix} [p]_F.\end{aligned}$$

Dette betyder at koordinatskiftsmatricen S fra F til E er

$$S = \begin{bmatrix} 3 & 0 & 0 \\ 0 & 2 & 0 \\ -1 & 2 & 3 \end{bmatrix},$$

dvs. S er matricen hvis søjlerne er koordinatvektorerne mht. E for basiselementerne af F . Koordinatskiftsmatricen den omvendte vej, fra E til F , er nu S^{-1} , som kan beregnes til at være

$$S^{-1} = \begin{bmatrix} \frac{1}{3} & 0 & 0 \\ 0 & \frac{1}{2} & 0 \\ \frac{1}{9} & -\frac{1}{3} & \frac{1}{3} \end{bmatrix}.$$

19.4 Baser for nul- og søjlerum af en matrix

Lad os betragte matricen

$$A = \begin{bmatrix} 1 & 1 & 2 & 1 & 0 \\ 1 & 1 & 3 & 2 & 1 \\ 2 & 2 & 5 & 4 & 1 \end{bmatrix}.$$

Denne matrix har reduceret echelonform

$$\begin{bmatrix} 1 & 1 & 0 & 0 & -2 \\ 0 & 0 & 1 & 0 & 1 \\ 0 & 0 & 0 & 1 & 0 \end{bmatrix},$$

så dens pivotsøjler er søjler 0, 2 og 3, og i ligningssystemet $Ax = b$ er x_1 og x_4 frie variable. For enhver x er Ax en lineær kombination af pivotsøjlerne. Så vektorerne

$$\begin{bmatrix} 1 \\ 1 \\ 2 \end{bmatrix}, \quad \begin{bmatrix} 2 \\ 3 \\ 5 \end{bmatrix}, \quad \begin{bmatrix} 1 \\ 2 \\ 4 \end{bmatrix},$$

som er pivotsøjlerne 0, 2, 3 af A , udspænder $S(A)$. Dette er også en basis for $S(A)$: Betingelsen at en lineær kombination af disse søjler er 0, er det samme som $Ax = 0$ hvor $x \in \mathbb{R}^5$ har formen $x = (x_0, 0, x_2, x_3, 0)$. Men fra echelonformen, ser vi at dette giver $x_0 = 0 = x_2 = x_3$, så $x = 0$, og søjlerne er dermed lineært uafhængig. Dermed har vi at pivotsøjlerne udgør en basis for $S(A)$.

Lad os nu betragte nulrummet $N(A)$ af A . Dette består af alle løsninger $x \in \mathbb{R}^5$ til $Ax = 0$. Fra den reducerede echelonform af A har vi at ligningssystemet $Ax = 0$ er ækvivalent med

$$\begin{aligned} x_0 + x_1 - 2x_4 &= 0, \\ x_2 + x_4 &= 0, \\ x_3 &= 0. \end{aligned}$$

Som vi har bemærket før er de frie variable x_1 og x_4 . Så den generelle løsning til $Ax = 0$ er

$$\begin{bmatrix} x_0 \\ x_1 \\ x_2 \\ x_3 \\ x_4 \end{bmatrix} = \begin{bmatrix} -x_1 + 2x_4 \\ x_1 \\ -x_4 \\ 0 \\ x_4 \end{bmatrix} = x_1 \begin{bmatrix} -1 \\ 1 \\ 0 \\ 0 \\ 0 \end{bmatrix} + x_4 \begin{bmatrix} 2 \\ 0 \\ -1 \\ 0 \\ 1 \end{bmatrix}.$$

Derfor udspænder disse to vektorer $(-1, 1, 0, 0, 0)$, $(2, 0, -1, 0, 1)$ nulrummet for A . Dette par er også lineært uafhængig: I hver vektor er der et 1-tal i pladsen der svarer til den tilsvarende frie variabel, og i de andre vektorer er der et 0-tal. Så en lineær kombination af disse vektorer kan kun give 0 hvis de tilsvarende frie variable er 0, dvs. hvis alle koefficienter er 0. De to vektorer udgør derfor en basis for $N(A)$ er $(-1, 1, 0, 0, 0)$, $(2, 0, -1, 0, 1)$.

Tilsvarende overvejelser for en generel matrix giver

Proposition 19.9. *Lad A være en $(m \times n)$ -matrix. Så udgør pivotsøjlerne af A en basis for søjlerummet $S(A)$.*

Hvis A har r pivotsøjler, så er der $n - r$ frie variable $x_{i_0}, \dots, x_{i_{n-r-1}}$ for systemet $Ax = 0$. En basis for nulrummet $N(A)$, gives for $j = 0, \dots, n - r - 1$ af de løsninger til $Ax = b$ hvor $x_{i_j} = 1$ og de øvrige $x_{i_k} = 0$. \square

19.5 Dimension

Definition 19.10. Antallet af elementer i en basis for V kaldes *dimensionen* $\dim V$ af V .

I eksemplet lige før har vi så

$$\dim S(A) = 3 \quad \text{og} \quad \dim N(A) = 2.$$

Som direkte konsekvens af proposition 19.9 har vi

Proposition 19.11. *For $A \in \mathbb{R}^{m \times n}$ er $\dim S(A)$ lige med antallet af pivotsøjler i A og $\dim N(A)$ lige med antallet af frie variable i $Ax = 0$. Det følger at*

$$\dim S(A) + \dim N(A) = n.$$

Bevis. Proposition 19.9 giver

$$\text{antal pivotsøjler} + \text{antal frie variable} = n$$

og resultater følger. \square

Kapitel 20

Lineære transformationer og koordinatskift

20.1 Generelle matrixrepræsentationer

Lad $L: V \rightarrow W$ være en lineær transformation. Antag at $E: v_0, v_1, \dots, v_{n-1}$ er en basis for V og at $F: w_0, w_1, \dots, w_{m-1}$ er en basis for W .

Definition 20.1. Vi definerer *matricen A af L mht. E og F* til at være matricen $A = [a_0 \mid a_1 \mid \dots \mid a_{n-1}] \in \mathbb{R}^{m \times n}$ givet ved

$$a_j = [L(v_j)]_F. \quad (20.1)$$

Vi siger også at A er *matrixrepræsentationen* af L mht. E og F .

Denne matrixrepræsentation er nyttigt da det følgende resultat fortæller os at lineære transformationer er blot multiplikation med en matrix. Mere præcis kan vi erstatte L med matrixmultiplikation ved $A \in \mathbb{R}^{m \times n}$, hvis vi skriver vektorerne ud, som koordinatvektorer mht. de givne baser. Vi er glad for matrixmultiplikation, da det er en operation vi kan udføre i python.

Proposition 20.2. *Lad A være matricen af $L: V \rightarrow W$ mht. baserne E for V og F for W . Da gælder for alle $v \in V$ at*

$$[L(v)]_F = A[v]_E.$$

Bevis. Lad os skrive

$$[v]_E = \begin{bmatrix} x_0 \\ x_1 \\ \vdots \\ x_{n-1} \end{bmatrix}.$$

Så har vi $v = x_0v_0 + x_1v_1 + \cdots + x_{n-1}v_{n-1}$. Vi kan nu regne

$$\begin{aligned} L(v) &= L(x_0v_0 + x_1v_1 + \cdots + x_{n-1}v_{n-1}) \\ &= x_0L(v_0) + x_1L(v_1) + \cdots + x_{n-1}L(v_{n-1}). \end{aligned}$$

Skrives $L(v)$ ud i koordinater, får vi så

$$\begin{aligned} [L(v)]_F &= x_0[L(v_0)]_F + x_1[L(v_1)]_F + \cdots + x_{n-1}[L(v_{n-1})]_F \\ &= x_0a_0 + x_1a_1 + \cdots + x_{n-1}a_{n-1} \\ &= Ax. \end{aligned}$$

Det sidste udtryk er lige med $A[v]_E$, som ønsket. \square

Eksempel 20.3. Lad V være vektorrummet af polynomier af grad højst 2 og lad W være vektorrummet af polynomier af grad højst 1. Funktionen $L: V \rightarrow W$ givet ved $L(p) = p'$, den afledede til p , er en lineær afbildning.

Lad os vælge baserne $E: x^2, x, 1$ for V , og $F: x, 1$ for W . Vi finder søjlerne a_0, a_1, a_2 af matricen $A \in \mathbb{R}^{2 \times 3}$ af L mht. E og F ved

$$\begin{aligned} a_0 &= [L(x^2)]_F = [2x]_F = [2 \times x + 0 \times 1]_F = \begin{bmatrix} 2 \\ 0 \end{bmatrix}, \\ a_1 &= [L(x)]_F = [1]_F = [0 \times x + 1 \times 1]_F = \begin{bmatrix} 0 \\ 1 \end{bmatrix}, \\ a_2 &= [L(1)]_F = [0]_F = \begin{bmatrix} 0 \\ 0 \end{bmatrix}. \end{aligned}$$

Dette giver

$$A = [a_0 \mid a_1 \mid a_2] = \begin{bmatrix} 2 & 0 & 0 \\ 0 & 1 & 0 \end{bmatrix}.$$

Proposition 20.2 siger nu at L svarer til multiplikation med matricen A .

I dette tilfælde har vi for $p(x) = ax^2 + bx + c$ at $L(p(x)) = 2ax + b$ kan udføres, som matrixmultiplikation

$$[L(p(x))]_F = A[p(x)]_E = \begin{bmatrix} 2 & 0 & 0 \\ 0 & 1 & 0 \end{bmatrix} \begin{bmatrix} a \\ b \\ c \end{bmatrix} = \begin{bmatrix} 2a \\ b \end{bmatrix},$$

og dermed implementeres f.eks. i python. Δ

Som konsekvens kan vi nu vise den følgende sætning. Formlen der fremkommer kaldes ofte *rang-nullitetsformlen*, da $\dim \operatorname{im} L$ er *rangen* af L , og $\dim \ker L$ er dens *nullitet*.

Sætning 20.4. *Lad $L: V \rightarrow W$ være lineær, lad E være en basis for V , lad F være en basis for W og lad A være matricen af L mht. E og F . Da gælder*

(a) $v \in \ker L$ hvis og kun hvis $[v]_E \in N(A)$,

(b) $w \in \operatorname{im} L$ hvis og kun hvis $[w]_F \in S(A)$.

Det følger at forskellige baser for V har det samme antal elementer og at

$$\dim \operatorname{im} L + \dim \ker L = \dim V.$$

Bevis. For del (a), har vi $v \in \ker L$ betyder at $L(v) = 0$. For sådan et $v \in \ker L$ har vi nu $A[v]_E = [L(v)]_F = [0]_F = 0$, så $[v]_E \in N(A)$. Omvendt $[v]_E \in N(A)$, giver $[L(v)]_F = A[v]_E = 0$, så $L(v) = 0$ og v ligger i $\ker L$.

Del (b) er tilsvarende: $w \in \operatorname{im} L$ betyder at $w = L(v)$ for et $v \in V$. Så for $w = L(v)$ har vi $[w]_F = [L(v)]_F = A[v]_E$, som siger at $[w]_F$ er en lineær kombination af søjle af A , dvs. $[w]_F \in S(A)$. Omvendt $[w]_F \in S(A)$ betyder $[w]_F = Ax$ for et $x \in \mathbb{R}^n$. Hvis basen E består af v_0, \dots, v_{n-1} , sætter vi $v = x_0 v_0 + \dots + v_{n-1} x_{n-1}$ og får at $[v]_E = x$. Vi har nu $[L(v)]_F = Ax = [w]_F$, så $L(v) = w$ og $w \in \operatorname{im} L$.

Bemærk nu at

(a) u_0, \dots, u_{k-1} er en basis for $\ker L$ hvis og kun hvis $[u_0]_E, \dots, [u_{k-1}]_E$ er en basis for $N(A)$, og

(b) f_0, \dots, f_{r-1} er en basis for $\operatorname{im} L$ hvis og kun hvis $[f_0]_F, \dots, [f_{r-1}]_F$ er en basis for $S(A)$.

Ved at bruge proposition 19.11, har vi

$$\dim \operatorname{im} L + \dim \ker L = r + k = \dim S(A) + \dim N(A) = n = \dim V,$$

som er den ønskede relation mellem dimensioner.

For at se at forskellige baser for V har det samme antal elementer, antag nu at E og F er begge baser for V , hvor E har n elementer, og F har m elementer. Lad $L = \operatorname{Id}: V \rightarrow V$, $\operatorname{Id}(v) = v$ være identitetsafbildningen. Denne afbildning er lineær, så vi kan betragte matricen $A \in \mathbb{R}^{m \times n}$ af $L = \operatorname{Id}$ mht. E og F . Vi har $\ker \operatorname{Id} = \{0\}$ og $\operatorname{im} \operatorname{Id} = V$, så $N(A) = \{0\}$, $S(A) = \mathbb{R}^m$. Vi får nu at

$$n = \dim S(A) + \dim N(A) = m + 0 = m,$$

så både E og F har n elementer. \square

Eksempel 20.5. I eksempel 20.3 kan vi godt beregne $\ker L$. Et polynomium $p(x) = ax^2 + bx + c$ ligger i $\ker L$ hvis $L(p(x)) = p'(x) = 2ax + b$ er 0-polynomiet. Dette er det samme som $a = 0 = b$, så $\ker L$ består af alle konstante polynomier $p(x) = c$, med c en vilkårlig skalar. Bemærk at disse polynomier har $[p(x)]_E = [c]_E = (0, 0, c)$.

Matricen

$$A = \begin{bmatrix} 2 & 0 & 0 \\ 0 & 1 & 0 \end{bmatrix}$$

af L mht. E og F har $v = (x_0, x_1, x_2) \in N(A)$ kun hvis $Av = 0$. Men

$$Av = \begin{bmatrix} 2 & 0 & 0 \\ 0 & 1 & 0 \end{bmatrix} \begin{bmatrix} x_0 \\ x_1 \\ x_2 \end{bmatrix} = \begin{bmatrix} 2x_0 \\ x_1 \end{bmatrix},$$

så $Av = 0$ kun for $v = (0, 0, x_2)$, med x_2 vilkårlig. Vektorer af denne form er netop koordinatvektorer for elementer af $\ker L$.

Bemærk at vi har $\dim \ker L = 1 = \dim N(A)$, da der er kun én fri parameter.

Billedmængden $\operatorname{im} L$ består af alle polynomier af formen $2ax + b$, så af alle polynomier af grad højst 1. Vi har dermed at $\dim \operatorname{im} L = 2$, og kan bekræfte at

$$\dim \operatorname{im} L + \dim \ker L = 2 + 1 = 3 = \dim V$$

i overensstemmelse med sætning 20.4.

△

20.2 Basisskift

Vi fortsætter med at betragte en lineær afbildning $L: V \rightarrow W$. Hvis E og E' er to baser for V , har vi en invertibel koordinatshiftmatrix S fra E til E' , som opfylder

$$[v]_{E'} = S[v]_E.$$

Tilsvarende hvis F og F' er baser for W , har vi en koordinatshiftmatrix T fra F til F' med

$$[w]_{F'} = T[w]_F.$$

Vi kan nu beregne matricen B af L mht. E' og F' , givet matricen A af L mht. E og F . Vi har

$$B[v]_{E'} = [L(v)]_{F'} \quad \text{og} \quad A[v]_E = [L(v)]_F,$$

så

$$BS[v]_E = B[v]_{E'} = [L(v)]_{F'} = T[L(v)]_F = TA[v]_E$$

for alle v . Det følger at

$$BS = TA,$$

som er det samme som

$$B = TAS^{-1}. \quad (20.2)$$

Omvendt, hver gang vi har en matrixformel af formen (20.2) med $A, B \in \mathbb{R}^{m \times n}$, og invertibel $S \in \mathbb{R}^{n \times n}$, $T \in \mathbb{R}^{m \times m}$, kan vi betragte dette som sigende at A og B er matrixrepræsentationer for den samme lineære transformation, blot i forhold til forskellige baser.

Vi kender allerede en formel af formen (20.2), nemlig den fulde SVD: $A = U\Sigma V^T$, som er $A = U\Sigma V^{-1}$, da V er ortogonal. Dette siger at A mht. standard baser har samme effekt som Σ mht. baser fra søjlerne af V og U : $Av_i = \sigma_i u_i$. Dette er et godt eksempel på at et nyt valg af baser kan gøre effekten af A mere tydeligt.

20.3 Afbildning på ét vektorrum

Betragt nu en lineær afbildning $L: V \rightarrow V$, som afbilder V ind i sig selv. Vælger vi en basis $E: v_0, \dots, v_{n-1}$ for V , har vi en *matrix A af L mht. kun E* givet ved (20.1) med $F = E$, dvs. $A = [a_0 \mid a_1 \mid \dots \mid a_{n-1}] \in \mathbb{R}^{n \times n}$ med

$$a_j = [L(v_j)]_E.$$

Skifter vi til en anden basis $F: w_0, w_1, \dots, w_{n-1}$ af V , har vi en koordinatmatrix S fra E til F . Ved brug af (20.2), ser vi nu at matricen B af L mht. kun F er relateret til A via

$$B = SAS^{-1}.$$

Eksempel 20.6. I et lille land bor 500 000 mennesker i storbyen og 700 000 mennesker i landområder. Der observeres at hvert år flyttes 3 % af landbefolkningen til byen, og 2 % af bybefolkningen på landet. Lad os undersøge hvordan befolkningsgrupperne udvikler sig med tid.

Vi starter med befolkningsfordelingen

$$b_0 = \begin{bmatrix} 500\,000 \\ 700\,000 \end{bmatrix},$$

hvor vi skrive antallet i byen øverst, og antallet på landet nederst. Ændringen fra år til år gives ved matricen

$$A = \begin{bmatrix} 0,98 & 0,03 \\ 0,02 & 0,97 \end{bmatrix}.$$

Vi kan så beregne befolkningsfordeling over de kommende år, som

$$b_1 = Ab_0, \quad b_2 = Ab_1, \quad b_3 = Ab_2, \dots$$

I python

```
import numpy as np
b0 = np.array([5000000., 7000000.])[:, np.newaxis]
a = np.array([[0.98, 0.03],
              [0.02, 0.97]])
b = b0
for i in range(5):
    print(f'b{i} = ', b)
    b = a @ b
```

```
b0 = [[5000000.]
      [7000000.]]
b1 = [[5110000.]
      [6890000.]]
b2 = [[5214500.]
      [6785500.]]
b3 = [[5313777.5]
      [6686222.5]]
b4 = [[5408008.625]
      [659191.375]]
```

Det ses at bybefolkningen vokser, men det er ikke klart hvad der sker over længere tid.

Lad os prøve at skifte fra standard basen, til basen F : $w_0 = (3, 2)$, $w_1 = (1, -1)$. Koordinatshiftmatricen fra F til standardbasen er

$$T = [w_0 \mid w_1] = \begin{bmatrix} 3 & 1 \\ 2 & -1 \end{bmatrix}.$$

Effekten af A mht. F er givet ved matricen $C = T^{-1}AT$. Vi har at

$$\begin{aligned} C = T^{-1}AT &= \frac{1}{3 \times (-1) - 1 \times 2} \begin{bmatrix} -1 & -1 \\ -2 & 3 \end{bmatrix} \begin{bmatrix} 0,98 & 0,03 \\ 0,02 & 0,97 \end{bmatrix} \begin{bmatrix} 3 & 1 \\ 2 & -1 \end{bmatrix} \\ &= -\frac{1}{5} \begin{bmatrix} -1,0 & -1,0 \\ -1,9 & 2,85 \end{bmatrix} \begin{bmatrix} 3 & 1 \\ 2 & -1 \end{bmatrix} = -\frac{1}{5} \begin{bmatrix} -5 & 0 \\ 0 & -4,75 \end{bmatrix} \\ &= \begin{bmatrix} 1 & 0 \\ 0 & 0,95 \end{bmatrix} \end{aligned}$$

Bemærk at

$$A = TCT^{-1}$$

og at

$$\begin{aligned} A^2 &= (TCT^{-1})(TCT^{-1}) = TC(T^{-1}T)CT^{-1} = TC^2T^{-1} \\ A^3 &= A^2A = TC^2T^{-1}TCT^{-1} = TC^3T^{-1}. \end{aligned}$$

Så generelt har vi

$$A^k = TC^kT^{-1}.$$

Men matricen C^k er nem at udregne: det er en diagonalmatrix med indgange de k 'te potenser af de tilsvarende indgange i C . Dette giver

$$\begin{aligned} b_k &= A^k b_0 = TC^k T^{-1} b_0 \\ &= T \begin{bmatrix} 1 & 0 \\ 0 & 0,95^k \end{bmatrix} T^{-1} b_0 \\ &= \begin{bmatrix} 3 & 0,95^k \\ 2 & -0,95^k \end{bmatrix} \begin{bmatrix} 240\,000 \\ 220\,000 \end{bmatrix} = \begin{bmatrix} 720\,000 \\ 480\,000 \end{bmatrix} + 0,95^k \begin{bmatrix} 240\,000 \\ 220\,000 \end{bmatrix}. \end{aligned} \tag{20.3}$$

Vi kan nu se hvad sker når k bliver stort. For $k \rightarrow \infty$, har vi $(0,95)^k \rightarrow 0$, da $|0,95| < 1$. Sml.

```
for k in range(0, 1000, 57):
    print(0.95**k)
```

```
1.0
0.053733545982740286
0.0028872939638792646
0.00015514454297379496
```

```

8.336466433853638e-06
4.479479024570453e-07
2.4069829214547706e-08
1.2933572748966046e-09
6.949667260276838e-11
3.734302652948301e-12
2.006573233156666e-13
1.0782029509155957e-14
5.793566784174943e-16
3.113088872015411e-17
1.6727730405279714e-18
8.988402709189803e-20
4.8297875028563765e-21
2.595216088715975e-22

```

Så fra (20.3) ser vi at

$$b_k \rightarrow \begin{bmatrix} 720\,000 \\ 480\,000 \end{bmatrix} \quad \text{for } k \rightarrow \infty.$$

Dvs. efter mange år vil befolkningen blive fordelt med tæt på 720 000 i byen, og 480 000 på landet.

Alternativt kan vi se at

$$A^k = TC^kT^{-1} = T \begin{bmatrix} 1 & 0 \\ 0 & (0,95)^k \end{bmatrix} T^{-1} \rightarrow T \begin{bmatrix} 1 & 0 \\ 0 & 0 \end{bmatrix} T^{-1}.$$

Så har vi $b_k = A^k b_0 \rightarrow T \operatorname{diag}(1, 0) T^{-1} b_0$ for $k \rightarrow \infty$. I python har vi

```

t = np.array([[3.0, 1.0],
              [2.0, -1.0]])
print(t)

```

```

[[ 3.  1.]
 [ 2. -1.]]

```

og matricen $C = TAT^{-1}$ er

```

c = np.linalg.inv(t) @ (a @ t)
print(c)

```

20.3 AFBILDNING PÅ ÉT VEKTORRUM

```
[[ 1.00000000e+00 -2.77555756e-17]
 [ 0.00000000e+00  9.50000000e-01]]
```

Grænsen for b_k når $k \rightarrow \infty$ er nu

```
print(t @ np.diag([1.0, 0.0]) @ np.linalg.inv(t) @ b0)
```

```
[[720000.]
 [480000.]]
```

som stemmer overens med grænsen fundet ovenfor.

△

Kapitel 21

Eigenverdier og egenvektorer

21.1 Et første eksempel

Lad os kikke igen på eksempel 20.6. Problemet er styret af matricen

$$A = \begin{bmatrix} 0,98 & 0,03 \\ 0,02 & 0,97 \end{bmatrix}$$

og vi valgt en særlig basis F : $w_0 = (3, 2)$, $w_1 = (1, -1)$. Hvor kommer denne basis fra?

Lad os lige beregne Aw_0 og Aw_1 :

$$\begin{aligned} Aw_0 &= \begin{bmatrix} 0,98 & 0,03 \\ 0,02 & 0,97 \end{bmatrix} \begin{bmatrix} 3 \\ 2 \end{bmatrix} = \begin{bmatrix} 3 \\ 2 \end{bmatrix} = w_0 \\ Aw_1 &= \begin{bmatrix} 0,98 & 0,03 \\ 0,02 & 0,97 \end{bmatrix} \begin{bmatrix} 1 \\ -1 \end{bmatrix} = \begin{bmatrix} 0,95 \\ -0,95 \end{bmatrix} = 0,95w_1. \end{aligned}$$

Vi ser at i begge tilfælde har vi en ligning af formen

$$Aw_i = \lambda_i w_i, \tag{21.1}$$

hvor $\lambda_0 = 1$ og $\lambda_1 = 0,95$.

Dette giver at vi kan beregne effekten af A på b ved at skrive b , som lineær kombination af basisvektorerne w_0, w_1 . Hvis $b = x_0 w_0 + x_1 w_1$, har vi

$$\begin{aligned} Ab &= A(x_0 w_0 + x_1 w_1) = x_0 Aw_0 + x_1 Aw_1 \\ &= x_0 w_0 + 0,95 x_1 w_1. \end{aligned}$$

Denne fremstilling gøre det nemt at gentage multiplikation med A og at se en videreudvikling: Vi har

$$A^k w_i = \lambda_i^k w_i,$$

som fører til

$$\begin{aligned} A^k b &= A^k(x_0 w_0 + x_1 w_1) = x_0 A^k w_0 + x_1 A^k w_1 \\ &= x_0 w_0 + (0,95)^k x_1 w_1. \end{aligned}$$

For eksempel, da $(0,95)^k \rightarrow 0$ for $k \rightarrow \infty$, kan vi nu aflæse at $A^k b \rightarrow x_0 w_0$, så for alle startvektorer $b = x_0 w_0 + x_1 w_1$ får vi en fast grænsevektor $x_0 w_0$ for $A^k b$ når $k \rightarrow \infty$.

Det væsentlige i disse regnestykker er relationerne (21.1).

21.2 Definitioner og første regnemetoder

Definition 21.1. Lad A være en kvadratisk matrix af størrelse $n \times n$. En skalar λ er *egenværdi* for A hvis der findes en vektor v med $v \neq 0$ således at

$$Av = \lambda v.$$

Vektoren $v \neq 0$ er så en *egenvektor* hørende til λ .

I eksemplet ovenfor har vi 2 egenværdier $\lambda_0 = 1$ og $\lambda_1 = 0,95$, med $w_0 = (3, 2)$ en egenvektor hørende til λ_0 og $w_1 = (1, -1)$ en egenvektor hørende til λ_1 .

Hvis λ er en egenværdi for A og $v \neq 0$ er en tilhørende egenvektor, så kan vi omskrive $Av = \lambda v$ til

$$(A - \lambda I_n)v = 0.$$

Vi ser så at λ er en egenværdi for A hvis og kun hvis $A - \lambda I_n$ er ikke invertibel. For $n = 2$, ved vi at en (2×2) -matrix B er invertibel hvis og kun hvis $\det B \neq 0$. Vi får så

Proposition 21.2. En skalar λ er en egenværdi for en $(n \times n)$ -matrix A hvis og kun hvis $A - \lambda I_n$ er ikke invertibel.

For $n = 2$, er λ en egenværdi hvis og kun hvis

$$\det(A - \lambda I_2) = 0.$$

□

Eksempel 21.3. Betragt $A \in \mathbb{R}^{2 \times 2}$ givet ved

$$A = \begin{bmatrix} 3 & 1 \\ 2 & 2 \end{bmatrix}.$$

Vi har

$$\begin{aligned} \det(A - \lambda I_2) &= \det \begin{bmatrix} 3 - \lambda & 1 \\ 2 & 2 - \lambda \end{bmatrix} = (3 - \lambda)(2 - \lambda) - 1 \times 2 \\ &= \lambda^2 - 5\lambda + 4. \end{aligned}$$

Hvis λ er en egen værdi af A , har vi så $\lambda^2 - 5\lambda + 4 = 0$. Denne andengradslikning har løsninger

$$\frac{5 \pm \sqrt{25 - 16}}{2} = \frac{5 \pm 3}{2} = 1 \text{ og } 4.$$

For at finde tilhørende egenvektorer skal vi finde $v \neq 0$ således at $(A - \lambda I_2)v = 0$. Det kan vi godt gøre ved hjælp af rækkeoperationer.

Vi skal løse et lineært ligningssystem af formen $Bx = c$, med $c = 0$. Sædvanligvis ville vi stille en udvidet matrix $[B \mid c]$ op. Men når $c = 0$, forbliver den sidste søjle 0 under alle rækkeoperationer, og vi kan nøjes med at arbejde blot med B .

For $\lambda_0 = 1$, har vi

$$A - \lambda_0 I_2 = \begin{bmatrix} 3 - 1 & 1 \\ 2 & 2 - 1 \end{bmatrix} = \begin{bmatrix} 2 & 1 \\ 2 & 1 \end{bmatrix} \sim_{R_1 \rightarrow R_1 - R_0} \begin{bmatrix} 2 & 1 \\ 0 & 0 \end{bmatrix}.$$

Hvis $v_0 = (x, y)$, siger dette reduktion at $2x + y = 0$, så en løsning er $v_0 = (1, -2)$.

For $\lambda_1 = 4$, beregnes

$$A - \lambda_1 I_2 = \begin{bmatrix} 3 - 4 & 1 \\ 2 & 2 - 4 \end{bmatrix} = \begin{bmatrix} -1 & 1 \\ 2 & -2 \end{bmatrix} \sim_{R_1 \rightarrow R_1 + 2R_0} \begin{bmatrix} -1 & 1 \\ 0 & 0 \end{bmatrix}.$$

For $v_1 = (x, y)$ har vi så ligningen $-x + y = 0$, dvs. $y = x$ og en løsning er $v_1 = (1, 1)$. △

Bemærk at $\det(A - \lambda I_2)$ er et andengradspolynomium. Dette kaldes det *karakteristiske polynomium* for $A \in \mathbb{R}^{2 \times 2}$.

Når vi skal bestemme en egenvektor v hørende til λ , får vi et system med (mindst) et fri variabel. Dette bekræfter at matricen $A - \lambda I_n$ er ikke invertibel, dvs. at vi har fat i korrekte egen værdier. Det understreger også at egenvektorerne

er ikke entydigt bestemt: hvis v er en egenvektor for A hørende til λ , så har vi $Av = \lambda v$; hvis $s \neq 0$ er en skalar, har vi nu $A(sv) = \lambda sv$, så sv er også en egenvektor hørende til λ .

For $n > 2$, kan vi godt bestemme egenvektorer, som ovenfor, hvis vi er givet en eller flere egenverdier på forhånd.

Eksempel 21.4. Betragt matricen

$$A = \begin{bmatrix} 3 & -1 & 2 \\ 1 & 2 & -1 \\ 4 & 1 & 1 \end{bmatrix}.$$

Jeg påstår at $\lambda = 1$ er en egenverdi for A .

Dette kan bekræftes ved at kikke på $A - \lambda I_3$ og vise at $(A - \lambda I_3)v = 0$ har en løsning med $v \neq 0$.

Vi har

$$\begin{aligned} A - \lambda I_3 &= \begin{bmatrix} 3-1 & -1 & 2 \\ 1 & 2-1 & -1 \\ 4 & 1 & 1-1 \end{bmatrix} = \begin{bmatrix} 2 & -1 & 2 \\ 1 & 1 & -1 \\ 4 & 1 & 0 \end{bmatrix} \\ &\sim_{R_0 \leftrightarrow R_1} \begin{bmatrix} 1 & 1 & -1 \\ 2 & -1 & 2 \\ 4 & 1 & 0 \end{bmatrix} \sim_{R_1 \rightarrow R_1 - 2R_0} \begin{bmatrix} 1 & 1 & -1 \\ 0 & -3 & 4 \\ 4 & 1 & 0 \end{bmatrix} \\ &\sim_{R_2 \rightarrow R_2 - 4R_0} \begin{bmatrix} 1 & 1 & -1 \\ 0 & -3 & 4 \\ 0 & -3 & 4 \end{bmatrix} \sim_{R_2 \rightarrow R_2 - R_1} \begin{bmatrix} 1 & 1 & -1 \\ 0 & -3 & 4 \\ 0 & 0 & 0 \end{bmatrix}. \end{aligned}$$

Vi ser så at vi har en fri variabel, fra den sidste søjle, så $\lambda = 1$ er en egenverdi. En tilhørende egenvektor fås ved at sætte denne frie variable lige med 1, for eksempel. Det giver $v = (-1/3, 4/3, 1)$. Et alternativ valg er vektoren $3v = (-1, 4, 3)$. Begge er egenvektorer hørende til $\lambda = 1$. Δ

Senere vil vi begynde at se på metoder for at finde egenverdier generelt, men vi vil erfare i det næste afsnit at det er ikke altid et problem der tillader en løsning. Desuden er de første metoder vi kikker på ikke gode for numeriske beregninger.

Lad os begynde med at bemærke det følgende resultat.

Proposition 21.5. Hvis $\lambda_0, \lambda_1, \dots, \lambda_{k-1}$ er forskellige egenverdier for A og v_0, v_1, \dots, v_{k-1} er tilhørende egenvektorer, så er v_0, v_1, \dots, v_{k-1} lineært uafhængig.

Det følger at hvis en $(n \times n)$ -matrix A har n forskellige egenverdier, så udgør de til hørende egenvektorer v_0, \dots, v_{n-1} en basis.

21.2 DEFINITIONER OG FØRSTE REGNEMETODER

Bevis. Lad os kikke først på tilfældet $k = 2$. Antag at

$$x_0 v_0 + x_1 v_1 = 0. \quad (21.2)$$

Ganger vi matricen A på denne ligning får vi fra $Av_i = \lambda_i v_i$ at

$$x_0 \lambda_0 v_0 + x_1 \lambda_1 v_1 = 0.$$

Trækker vi λ_1 gange ligning (21.2) fra, fås

$$x_0(\lambda_0 - \lambda_1)v_0 = 0.$$

Men $v_0 \neq 0$, så $x_0(\lambda_0 - \lambda_1) = 0$. Da vi har antaget at $\lambda_0 \neq \lambda_1$, får vi $x_0 = 0$. Ligning (21.2) er nu $x_1 v_1 = 0$. Men $v_1 \neq 0$, giver $x_1 = 0$. Så $x_0 = 0 = x_1$, og samlingen v_0, v_1 er lineært uafhængig.

Generelt hvis har vist at v_0, \dots, v_{k-2} er lineært uafhængig, kan vi kikke på ligningen

$$x_0 v_0 + \dots + x_{k-2} v_{k-2} + x_{k-1} v_{k-1} = 0. \quad (21.3)$$

Ganger vi matricen A på denne ligning får vi

$$x_0 \lambda_0 v_0 + \dots + x_{k-2} \lambda_{k-2} v_{k-2} + x_{k-1} \lambda_{k-1} v_{k-1} = 0.$$

Trækkes λ_{k-1} ganger (21.3) fra, har vi

$$x_0(\lambda_0 - \lambda_{k-1})v_0 + \dots + x_{k-2}(\lambda_{k-2} - \lambda_{k-1})v_{k-2} = 0.$$

Da v_0, \dots, v_{k-2} er lineært uafhængig, fås $x_i(\lambda_i - \lambda_{k-1}) = 0$ for $i = 0, \dots, k-2$. Men $\lambda_i \neq \lambda_{k-1}$, så $x_i = 0$. Vi har tilbage $x_{k-1} v_{k-1} = 0$, så får også $x_{k-1} = 0$. Dermed er v_0, v_1, \dots, v_{k-1} lineært uafhængig. \square

Proposition 21.6. Hvis en $(n \times n)$ -matrix A har en basis v_0, v_1, \dots, v_{n-1} af egenvektorer, så kan A skrives som

$$A = V \Lambda V^{-1}, \quad (21.4)$$

hvor $V = [v_0 \mid v_1 \mid \dots \mid v_{n-1}]$ og

$$\Lambda = \text{diag}(\lambda_0, \lambda_1, \dots, \lambda_{n-1}).$$

Bevis. Vi bemærker at

$$\begin{aligned} AV &= A[v_0 \mid v_1 \mid \dots \mid v_{n-1}] = [Av_0 \mid Av_1 \mid \dots \mid Av_{n-1}] \\ &= [\lambda_0 v_0 \mid \lambda_1 v_1 \mid \dots \mid \lambda_{n-1} v_{n-1}] = V\Lambda. \end{aligned}$$

Da V er en $(n \times n)$ -matrix med lineært uafhængige søjler, er V invertibel og ligning (21.4) følger. \square

Definition 21.7. En matrix A der kan skrives i formen (21.4) kaldes *diagonaliserbar*.

Eksempel 21.8. I eksempel 21.3 fandt vi at

$$A = \begin{bmatrix} 3 & 1 \\ 2 & 2 \end{bmatrix}$$

har egenverdier 1 og 4 med tilhørende egenvektorer $(1, -2)$ hhv. $(1, 1)$. Det følger at

$$A = V\Lambda V^{-1}$$

med

$$V = \begin{bmatrix} 1 & 1 \\ -2 & 1 \end{bmatrix} \quad \text{og} \quad \Lambda = \begin{bmatrix} 1 & 0 \\ 0 & 4 \end{bmatrix}.$$

Dette kan bekræftes ved

```
>>> import numpy as np
>>> v = np.array([[ 1.0, 1.0],
...               [-2.0, 1.0]])
>>> lambda_mat = np.diag([1.0, 4.0])
>>> v @ lambda_mat @ np.linalg.inv(v)
array([[3., 1.],
       [2., 2.]])
```

\triangle

21.3 Andre eksempler

Eksempel 21.9. Betragt matricen $A \in \mathbb{R}^{2 \times 2}$ givet ved

$$A = \begin{bmatrix} 1 & 2 \\ -2 & 1 \end{bmatrix}.$$

Det karakteristiske polynomium er

$$\det(A - \lambda I_2) = \det \begin{bmatrix} 1 - \lambda & 2 \\ -2 & 1 - \lambda \end{bmatrix} = (1 - \lambda)^2 + 4 = \lambda^2 - 2\lambda + 5.$$

Egenverdier er løsninger til $p(\lambda) = \lambda^2 - 2\lambda + 5 = 0$. Men p har toppunkt i 1 og $p(1) = 4$, så p har ingen rødder, som reel tal, og dermed A har ingen egenverdier og ingen egenvektorer i \mathbb{R}^2 .

Arbejder vi dog med komplekse skalarer, dvs. betragter vi A som en matrix i $\mathbb{C}^{2 \times 2}$, kan vi finde rødder via den sædvanlige formel

$$\frac{2 \pm \sqrt{4 - 20}}{2} = \frac{2 \pm \sqrt{-16}}{2} = \frac{2 \pm 4i}{2} = 1 \pm 2i.$$

Lad os finde tilsvarende egenvektorer i \mathbb{C}^2 ved hjælp af python.

Først for $\lambda_0 = 1 + 2i$,

```
>>> import numpy as np

>>> a = np.array([[ 1.0, 2.0],
...               [-2.0, 1.0]])
>>> lambda_0 = 1.0 + 2.0j
>>> b = a - lambda_0 * np.eye(2)
>>> b
array([[ 0.-2.j,  2.+0.j],
       [-2.+0.j,  0.-2.j]])
>>> b[0, :] /= -2.0j
>>> b
array([[ 1.+0.j, -0.+1.j],
       [-2.+0.j,  0.-2.j]])
>>> b[1, :] += 2.0 * b[0, :]
>>> b
array([[ 1.+0.j, -0.+1.j],
       [ 0.+0.j,  0.+0.j]])
```

så $v_0 = (-i, 1)$ er en tilhørende egenvektor. For $\lambda_1 = 1 - 2i$, har vi

```
>>> lambda_1 = 1.0 - 2.0j
>>> c = a - lambda_1 * np.eye(2)
```

21 EGENVÆRDIER OG EGENVEKTORER

```
>>> c
array([[ 0.+2.j,  2.+0.j],
       [-2.+0.j,  0.+2.j]])
>>> c[0, :] /= 2.0j
>>> c
array([[ 1.+0.j,  0.-1.j],
       [-2.+0.j,  0.+2.j]])
>>> c[1, :] += 2.0 * c[0, :]
>>> c
array([[1.+0.j,  0.-1.j],
       [0.+0.j,  0.+0.j]])
```

og $v_1 = (i, 1)$ er en tilhørende egenvektor. Det følger at v_0, v_1 er en basis for \mathbb{C}^2 og at

$$A = V\Lambda V^{-1}$$

med

$$V = [v_0 \mid v_1] = \begin{bmatrix} -i & i \\ 1 & 1 \end{bmatrix} \quad \text{og} \quad \Lambda = \begin{bmatrix} 1+2i & 0 \\ 0 & 1-2i \end{bmatrix}.$$

Dette kan bekræftes i python

```
>>> v = np.array([-1.j, 1.j],
...               [1.0, 1.0]])
>>> lambda_mat = np.diag([1.0+2.0j, 1.0-2.0j])
>>> v @ lambda_mat @ np.linalg.inv(v)
array([[ 1.+0.j,  2.+0.j],
       [-2.+0.j,  1.+0.j]])
```

Δ

Eksempel 21.10. Betragt matricen

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

Det karakteristiske polynomium er

$$\det(A - \lambda I_2) = \det \begin{bmatrix} 1-\lambda & 1 \\ 0 & 1-\lambda \end{bmatrix} = (1-\lambda)^2,$$

så det eneste egen værdi er $\lambda = 1$. Lad os finde de tilhørende egenvektorer. Vi har

$$A - \lambda I_2 = \begin{bmatrix} 1-1 & 1 \\ 0 & 1-1 \end{bmatrix} = \begin{bmatrix} 0 & 1 \\ 0 & 0 \end{bmatrix},$$

så enhver egenvektor er et multiplum af $v = (1, 0)$. Det følger at der findes ingen basis af \mathbb{R}^2 (eller af \mathbb{C}^2), som består af egenvektorer til A . \triangle

21.4 Determinanter

For (2×2) -matricer har vi at

$$\det A = \det \begin{bmatrix} a_{00} & a_{01} \\ a_{10} & a_{11} \end{bmatrix} = a_{00}a_{11} - a_{01}a_{10}$$

og at matricen A er invertibel hvis og kun hvis dens $\det A \neq 0$. Denne definition kan udvides til $(n \times n)$ -matricer på den følgende måde. Først for (1×1) -matricer, sætter vi

$$\det [a_{00}] = a_{00}.$$

For $n > 1$, og for $0 \leq i, j \leq n-1$ definerer vi den (i, j) 'te *undermatrix* M_{ij} , til at være den $((n-1) \times (n-1))$ -delmatrix af A der fås ved at slette hele række i og hele søjle j fra A . Dvs. at M_{ij} er A fraregnet rækken $A_{[i,:]}$ og søjle $A_{[:,j]}$.

Eksempel 21.11. For den følgende matrix A , bestemmer vi den $(1, 2)$ 'te undermatrix M_{12} ved

$$A = \begin{bmatrix} 2 & -1 & 3 & 4 & 5 \\ 6 & -2 & -5 & 2 & -1 \\ -1 & 3 & 0 & -6 & 3 \\ 4 & 1 & -3 & -3 & 2 \\ 5 & 2 & 1 & -1 & -2 \end{bmatrix} = \begin{matrix} A_{[1,:]} \\ \begin{bmatrix} 2 & -1 & 3 & 4 & 5 \\ 6 & -2 & -5 & 2 & -1 \\ -1 & 3 & 0 & -6 & 3 \\ 4 & 1 & -3 & -3 & 2 \\ 5 & 2 & 1 & -1 & -2 \end{bmatrix} \\ A_{[:,2]} \end{matrix}$$

giver

$$M_{12} = \begin{bmatrix} 2 & -1 & 4 & 5 \\ -1 & 3 & -6 & 3 \\ 4 & 1 & -3 & 2 \\ 5 & 2 & -1 & -2 \end{bmatrix}$$

I python kan dette undermatrix fås vi `np.delete`:

```

>>> import numpy as np
>>> a = np.array([[2.0, -1.0, 3.0, 4.0, 5.0],
...               [6.0, -2.0, -5.0, 2.0, -1.0],
...               [-1.0, 3.0, 0.0, -6.0, 3.0],
...               [4.0, 1.0, -3.0, -3.0, 2.0],
...               [5.0, 2.0, 1.0, -1.0, -2.0]])

>>> m12 = np.delete(np.delete(a, 1, 0), 2, 1)
>>> m12
array([[ 2., -1.,  4.,  5.],
       [-1.,  3., -6.,  3.],
       [ 4.,  1., -3.,  2.],
       [ 5.,  2., -1., -2.]])

```

Her sletter `np.delete(a, 1, 0)` række 1, og `np.delete(a, 2, 1)` sletter søjle 2. Det er den sidste argument til `np.delete` der bestemmer om der er tale om en række eller en søjle der skal slettes: 0 for rækker, 1 for søjler. Δ

Bemærk nu at for $n = 2$, har vi $M_{00} = [a_{11}]$ og $M_{01} = [a_{10}]$, så vi kan skrive

$$\det A = a_{00}a_{11} - a_{01}a_{10} = a_{00} \det(M_{00}) - a_{01} \det(M_{01}).$$

Definition 21.12. For A en $(n \times n)$ -matrix, $n > 1$, sætter vi *determinanten* til at være

$$\det A = a_{00} \det(M_{00}) - a_{01} \det(M_{01}) + \cdots + (-1)^{n-1} a_{0,n-1} \det(M_{0,n-1}),$$

hvor fortegnene er skiftevis $+$, $-$, $+$, $-$, \dots .

Eksempel 21.13. Lad os beregne determinanten af en (3×3) -matrix

$$\begin{aligned}
 \det \begin{bmatrix} 1 & -1 & 2 \\ 3 & 2 & -2 \\ -1 & 4 & 5 \end{bmatrix} &= 1 \times \det \begin{bmatrix} 2 & -2 \\ 4 & 5 \end{bmatrix} - (-1) \times \det \begin{bmatrix} 3 & -2 \\ -1 & 5 \end{bmatrix} + 2 \times \det \begin{bmatrix} 3 & 2 \\ -1 & 4 \end{bmatrix} \\
 &= 1 \times (10 - (-8)) + 1 \times (15 - 2) + 2 \times (12 - (-2)) \\
 &= 1 \times 18 + 1 \times 13 + 2 \times 14 \\
 &= 59.
 \end{aligned}$$

Δ

Sætning 21.14. For en kvadratisk matrix A er A invertibel hvis og kun hvis $\det A \neq 0$.

Vi vil ikke give et bevis for dette endnu, det gemmes til afsnit 22.3 især side 281, men sætningen har den følgende konsekvens:

Proposition 21.15. For en $(n \times n)$ -matrix A er λ en egenværdi for A hvis og kun hvis

$$\det(A - \lambda I_n) = 0.$$

□

Polynomiet $p(\lambda) = \det(A - \lambda I_n)$ kaldes det *karakteristiske polynomium* for A .

21.5 Et andet eksempel

Lad os beregne egenværdier og egenvektorer for den følgende (3×3) -matrix

$$A = \begin{bmatrix} -3 & 4 & 2 \\ 1 & 0 & -1 \\ -6 & 6 & 5 \end{bmatrix}.$$

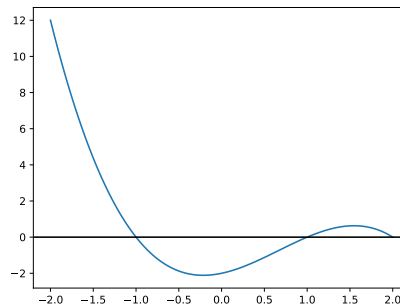
Vi finder først det karakteristiske polynomium

$$\begin{aligned} \det(A - \lambda I_3) &= \det \begin{bmatrix} -3 - \lambda & 4 & 2 \\ 1 & -\lambda & -1 \\ -6 & 6 & 5 - \lambda \end{bmatrix} \\ &= (-3 - \lambda) \det \begin{bmatrix} -\lambda & -1 \\ 6 & 5 - \lambda \end{bmatrix} - 4 \det \begin{bmatrix} 1 & -1 \\ -6 & 5 - \lambda \end{bmatrix} + 2 \det \begin{bmatrix} 1 & -\lambda \\ -6 & 6 \end{bmatrix} \\ &= (-3 - \lambda)(\lambda^2 - 5\lambda + 6) - 4(5 - \lambda - 6) + 2(6 - 6\lambda) \\ &= -\lambda^3 + 2\lambda^2 + \lambda - 2. \end{aligned}$$

Nu har vi et tredjegradspolynomium der skal løses. Dette er ikke så nemt, der findes en formel, men den er ikke helt lige til. Vi kan dog plote funktionen

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

21 EGENVÆRDIER OG EGENVEKTORER



Figur 21.1: Tredjegradspolynomium.

```
x = np.linspace(-2, 2, 100)
fig, ax = plt.subplots()
ax.plot(x, -x**3+2*x**2+x-2)
ax.axhline(0, color='black') # tegn x akser
```

Vi ser fra plottet i figur 21.1 at der er rødder tæt på $\lambda = -1, 1$ og 2 . Vi har faktisk

$$\begin{aligned}\det(A - \lambda I_3) &= -\lambda^3 + 2\lambda^2 + \lambda - 2 \\ &= -(\lambda + 1)(\lambda^2 - 3\lambda + 2) = -(\lambda + 1)(\lambda - 1)(\lambda - 2),\end{aligned}$$

så rødderne er netop $\lambda_0 = -1, \lambda_1 = 1$ og $\lambda_2 = 2$.

Vi kan nu finde tilsvarende egenvektorer. For $\lambda_0 = -1$,

$$A - \lambda_0 I_3 = \begin{bmatrix} -2 & 4 & 2 \\ 1 & 1 & -1 \\ -6 & 6 & 6 \end{bmatrix} \sim_{R_0 \leftrightarrow R_1} \begin{bmatrix} 1 & 1 & -1 \\ -2 & 4 & 2 \\ -6 & 6 & 6 \end{bmatrix} \sim \begin{bmatrix} 1 & 1 & -1 \\ 0 & 6 & 0 \\ 0 & 12 & 0 \end{bmatrix} \sim \begin{bmatrix} 1 & 1 & -1 \\ 0 & 1 & 0 \\ 0 & 0 & 0 \end{bmatrix}$$

giver $v_0 = (1, 0, 1)$. For $\lambda = 1$,

$$A - \lambda_1 I_3 = \begin{bmatrix} -4 & 4 & 2 \\ 1 & -1 & -1 \\ -6 & 6 & 4 \end{bmatrix} \sim \begin{bmatrix} 1 & -1 & -1 \\ -4 & 4 & 2 \\ -6 & 6 & 4 \end{bmatrix} \sim \begin{bmatrix} 1 & -1 & -1 \\ 0 & 0 & -2 \\ 0 & 0 & -2 \end{bmatrix} \sim \begin{bmatrix} 1 & -1 & -1 \\ 0 & 0 & 1 \\ 0 & 0 & 0 \end{bmatrix}$$

og $v_1 = (1, 1, 0)$. Til sidst for $\lambda_2 = 2$,

$$A - \lambda_2 I_3 = \begin{bmatrix} -5 & 4 & 2 \\ 1 & -2 & -1 \\ -6 & 6 & 3 \end{bmatrix} \sim \begin{bmatrix} 1 & -2 & -1 \\ -5 & 4 & 2 \\ -6 & 6 & 3 \end{bmatrix} \sim \begin{bmatrix} 1 & -2 & -1 \\ 0 & -6 & -3 \\ 0 & -6 & -3 \end{bmatrix} \sim \begin{bmatrix} 1 & 0 & 0 \\ 0 & 2 & 1 \\ 0 & 0 & 0 \end{bmatrix}$$

og $v_2 = (0, 1, -2)$.

Det følger at

$$A = V\Lambda V^{-1}$$

med

$$V = [v_0 \mid v_1 \mid v_2] = \begin{bmatrix} 1 & 1 & 0 \\ 0 & 1 & 1 \\ 1 & 0 & -2 \end{bmatrix} \quad \text{og} \quad \Lambda = \begin{bmatrix} \lambda_0 & 0 & 0 \\ 0 & \lambda_1 & 0 \\ 0 & 0 & \lambda_2 \end{bmatrix} = \begin{bmatrix} -1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 2 \end{bmatrix}.$$

Som det kan ses er denne fremgangsmåde ikke velegnet til generelle numeriske eksempler. Den største forhindring ligger i at bestemme egenverdierne. Vi vil finde numeriske metoder senere der finder egenverdier og egenvektorer samtidigt.

Kapitel 22

Differentialligninger

En væsentlig anvendelse af egenverdier og egenvektorer er til løsninger af lineære differentialligninger. Problemer for generelle lineære differentialligninger kan omskrives til lineære systemer af førsteordens differentialligninger, så det er disse systemer vi begynder med.

22.1 Førsteordenssystemer

Lad os begynde med en konkret førsteordensdifferentialligning for én ukendt funktion $y(t)$:

$$y'(t) = ay(t), \quad (22.1)$$

hvor a er en given konstant. Funktionen $y(t) = ce^{at}$ har

$$y'(t) = \frac{d}{dt}(ce^{at}) = cae^{at} = ay(t),$$

så er en løsning til (22.1).

To ting skal bemærkes. Hvis $y(t) = ce^{at}$ og en startværdi $y(t_0) = r_0$ er givet, så har vi en entydig løsning for c , givet ved $ce^{at_0} = r_0$, dvs. $c = r_0e^{-at_0}$. Desuden hvis $z(t)$ er en anden løsning til (22.1), så kan vi betragte $w(t) = z(t)e^{-at}$, som opfylder

$$w'(t) = \frac{d}{dt}(z(t)e^{-at}) = z'(t)e^{-at} - az(t)e^{-at} = (z'(t) - az(t))e^{-at} = 0$$

for alle t . Det følger at $w(t)$ er konstant, og dermed at $z(t) = ce^{at}$ for en konstant c . Vi har så følgende resultat:

Proposition 22.1. *Alle løsninger $t \mapsto y(t) \in \mathbb{R}$ til differentialligningen (22.1) har formen $y(t) = ce^{at}$ med c konstant. Løsninger er entydigt bestemt af en begyndelsesværdi $y(t_0) = r_0$. \square*

Et *førsteordenssystem af lineære differentialligninger* er et system af formen

$$\begin{aligned} y_0'(t) &= a_{00}y_0(t) + a_{01}y_1(t) + \cdots + a_{0,n-1}y_{n-1}(t), \\ y_1'(t) &= a_{10}y_0(t) + a_{11}y_1(t) + \cdots + a_{1,n-1}y_{n-1}(t), \\ &\vdots \\ y_{n-1}'(t) &= a_{n-1,0}y_0(t) + a_{n-1,1}y_1(t) + \cdots + a_{n-1,n-1}y_{n-1}(t), \end{aligned}$$

med $A = (a_{ij}) \in \mathbb{R}^{n \times n}$ en kvadratisk matrix af konstanter. Her er der n ukendte funktioner $y_i(t)$.

Systemet kan skrives i vektornotation ved at sætte $y(t) = (y_0(t), \dots, y_{n-1}(t))$. Vi får

$$y'(t) = Ay(t). \quad (22.2)$$

Dette system er lineær, da for løsninger $y(t)$, $z(t)$ og konstanter a, b har vi at $ay(t) + bz(t)$ er også en løsning, jævnfor

$$(ay(t) + bz(t))' = ay'(t) + bz'(t) = aAy(t) + bAz(t) = A(ay(t) + bz(t)).$$

Hvis $A = \text{diag}(\lambda_0, \lambda_1, \dots, \lambda_{n-1})$ er en diagonalmatrix, er systemet blot

$$\begin{aligned} y_0'(t) &= \lambda_0 y_0(t), \\ y_1'(t) &= \lambda_1 y_1(t), \\ &\vdots \\ y_{n-1}'(t) &= \lambda_{n-1} y_{n-1}(t), \end{aligned}$$

som er n kopier af (22.1). Det følger at løsningen i dette tilfælde er

$$\begin{aligned} \begin{bmatrix} y_0(t) \\ y_1(t) \\ \vdots \\ y_{n-1}(t) \end{bmatrix} &= \begin{bmatrix} c_0 e^{\lambda_0 t} \\ c_1 e^{\lambda_1 t} \\ \vdots \\ c_{n-1} e^{\lambda_{n-1} t} \end{bmatrix} \\ &= c_0 \begin{bmatrix} 1 \\ 0 \\ \vdots \\ 0 \end{bmatrix} e^{\lambda_0 t} + c_1 \begin{bmatrix} 0 \\ 1 \\ \vdots \\ 0 \end{bmatrix} e^{\lambda_1 t} + \cdots + c_{n-1} \begin{bmatrix} 0 \\ 0 \\ \vdots \\ 1 \end{bmatrix} e^{\lambda_{n-1} t} \\ &= c_0 e_0 e^{\lambda_0 t} + c_1 e_1 e^{\lambda_1 t} + \cdots + c_{n-1} e_{n-1} e^{\lambda_{n-1} t}, \end{aligned}$$

for konstante c_0, c_1, \dots, c_{n-1} . Vektorene i resultatet er standardbasisvektorerne e_0, \dots, e_{n-1} . For en diagonalmatrix $A = \text{diag}(\lambda_0, \lambda_1, \dots, \lambda_{n-1})$ opfylder vektorerne e_i at $Ae_i = \lambda_i e_i$, dvs. de er egenvektorer for A .

Generelt hvis $v \in \mathbb{R}^n$ er en egenvektor for A med $Av = \lambda v$, har vi at $y(t) = cve^{\lambda t}$ løser (22.2):

$$y'(t) = \frac{d}{dt}(cve^{\lambda t}) = cv \frac{d}{dt}e^{\lambda t} = cv\lambda e^{\lambda t} = ce^{\lambda t}Av = A(cve^{\lambda t}) = Ay(t).$$

Fra linearitet følger det, at hvis v_0, v_1, \dots, v_{k-1} er egenvektorer for A med egenverdier $\lambda_0, \lambda_1, \dots, \lambda_{k-1}$, så er

$$y(t) = c_0 v_0 e^{\lambda_0 t} + c_1 v_1 e^{\lambda_1 t} + \dots + c_{k-1} v_{k-1} e^{\lambda_{k-1} t},$$

med c_i konstant, en løsning til (22.2).

Proposition 22.2. Hvis A har en basis bestående af egenvektorer v_0, v_1, \dots, v_{n-1} med egenverdier $\lambda_0, \lambda_1, \dots, \lambda_{n-1}$ så er alle løsninger til (22.2) af formen

$$y(t) = c_0 v_0 e^{\lambda_0 t} + c_1 v_1 e^{\lambda_1 t} + \dots + c_{n-1} v_{n-1} e^{\lambda_{n-1} t},$$

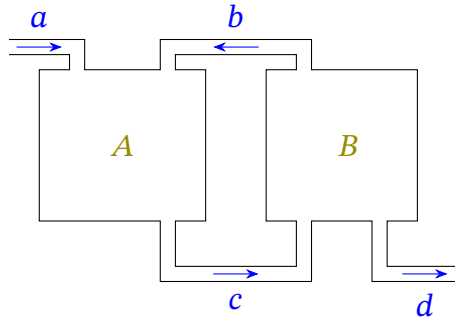
med c_i konstant. Givet en vektor af begyndelsesverdier $y(t_0) = r \in \mathbb{R}^n$, har (22.2) en entydig løsning, som opfylder med disse begyndelsesverdier.

Bevis. Ligningssystemet er $y' = Ay = V\Lambda V^{-1}y$, som er ækvivalent med $(V^{-1}y)' = \Lambda(V^{-1}y)$. Vi får også $(V^{-1}y)(t_0) = V^{-1}r$. Skrives $z(t) = V^{-1}y(t)$ har vi derfor at systemet er ækvivalent med $z' = \Lambda z$, med begyndelsesvektor $z(t_0) = V^{-1}r$. Da Λ er diagonal, har vi set ovenfor at eksistens og entydig følger nu fra tilfældet for $n = 1$, dvs. det følger fra proposition 22.1. \square

Eksempel 22.3. To beholdere A og B er forbundet med rør, som i figur 22.1. Som udgangspunkt indeholder A 200 l vand med 60 g salt, og B indeholder 200 l rent vand. Vandtilstand udvikler sig ved at der tilføjes 15 l/min rent vand via rør a , der flyder 5 l/min blanding i gennem rør b i den anviste retning, og der flyder 20 l/min i gennem rør c . Igennem rør d forlader 15 l/min saltvand. Hvor meget salt er der i hver beholder efter tid t ?

Vi sætter $y_0(t)$ til at være mængden af salt i A ved tid t min, og $y_1(t)$ til mængden af salt i B . For beholder A mistes der $20(y_0(t)/200) = y_0(t)/10$ gram salt per minut, og der tilføjes $5(y_1(t)/200) = y_1(t)/40$ gram salt per minut. Så

$$y_0'(t) = -\frac{1}{10}y_0(t) + \frac{1}{40}y_1(t).$$



Figur 22.1: Beholdere forbudne med rør.

For beholder B har vi

$$y_1'(t) = \frac{1}{10}y_0(t) - \frac{5+15}{200}y_1(t) = \frac{1}{10}y_0(t) - \frac{1}{10}y_1(t).$$

Så systemet er

$$y'(t) = Ay, \quad A = \begin{bmatrix} -\frac{1}{10} & \frac{1}{40} \\ \frac{1}{10} & -\frac{1}{10} \end{bmatrix},$$

med begyndelsesværdi

$$y(0) = \begin{bmatrix} 60 \\ 0 \end{bmatrix}.$$

Nu vi vil løse systemet. Det karakteristiske polynomium for A er

$$\det \begin{bmatrix} -\frac{1}{10} - \lambda & \frac{1}{40} \\ \frac{1}{10} & -\frac{1}{10} - \lambda \end{bmatrix} = \left(-\frac{1}{10} - \lambda\right)^2 - \frac{1}{10} \frac{1}{40} = \lambda^2 + \frac{1}{5}\lambda + \frac{3}{400},$$

som har rødder

$$\frac{-(1/5) \pm \sqrt{(1/25) - (3/100)}}{2} = \frac{-(1/5) \pm (1/10)}{2} = -\frac{1}{20}, -\frac{3}{20}$$

Matricen A er en (2×2) -matrix, og der to forskellige rødder, så A er diagonaliserbar.

For $\lambda_0 = -1/20$ har vi

$$\begin{bmatrix} -\frac{1}{10} - \lambda_0 & \frac{1}{40} \\ \frac{1}{10} & -\frac{1}{10} - \lambda_0 \end{bmatrix} = \begin{bmatrix} -\frac{1}{20} & \frac{1}{40} \\ \frac{1}{10} & -\frac{1}{20} \end{bmatrix} \sim \begin{bmatrix} -2 & 1 \\ 2 & -1 \end{bmatrix} \sim \begin{bmatrix} 2 & -1 \\ 0 & 0 \end{bmatrix}$$

så en egenvektor er $v_0 = (1, 2)$. For $\lambda_1 = -3/20$ har vi

$$\begin{bmatrix} -\frac{1}{10} - \lambda_1 & \frac{1}{40} \\ \frac{1}{10} & -\frac{1}{10} - \lambda_1 \end{bmatrix} = \begin{bmatrix} \frac{1}{20} & \frac{1}{40} \\ \frac{1}{10} & \frac{1}{20} \end{bmatrix} \sim \begin{bmatrix} 2 & 1 \\ 0 & 0 \end{bmatrix}$$

og en egenvektor er $v_1 = (1, -2)$. Proposition 22.2 fortæller os så at løsningen har formen

$$y(t) = c_0 \begin{bmatrix} 1 \\ 2 \end{bmatrix} e^{-t/20} + c_1 \begin{bmatrix} 1 \\ -2 \end{bmatrix} e^{-3t/20}.$$

Konstanterne c_0 og c_1 kan bestemmes ud fra vores begyndelsesdata. Ved $t = 0$ har vi

$$y(0) = c_0 \begin{bmatrix} 1 \\ 2 \end{bmatrix} + c_1 \begin{bmatrix} 1 \\ -2 \end{bmatrix} = \begin{bmatrix} 1 & 1 \\ 2 & -2 \end{bmatrix} \begin{bmatrix} c_0 \\ c_1 \end{bmatrix} = \begin{bmatrix} 60 \\ 0 \end{bmatrix},$$

som har løsning $c_0 = c_1 = 30$. Så saltmængderne er givet ved

$$y(t) = \begin{bmatrix} y_0(t) \\ y_1(t) \end{bmatrix} = 30 \begin{bmatrix} 1 \\ 2 \end{bmatrix} e^{-t/20} + 30 \begin{bmatrix} 1 \\ -2 \end{bmatrix} e^{-3t/20} = \begin{bmatrix} 30e^{-t/20} + 30e^{-3t/20} \\ 60e^{-t/20} - 60e^{-3t/20} \end{bmatrix}.$$

Da $e^{-t} \rightarrow 0$ for $t \rightarrow \infty$, ser vi at løsningen går mod 0 når t vokser. Men $e^{-t/20}$ går langsommere mod 0 end $e^{-3t/20}$, så forholdet mellem $y_0(t)$ og $y_1(t)$ nærmer sig fordeling fra indgangerne i v_0 , dvs. 1 til 2.

Vi kan plotte $(y_0(t), y_1(t))$, og får figuren til venstre i figur 22.2. Punkterne bevæger langs kurven fra højre til venstre, og kurvens tangentretning nærmer sig v_0 efterhånden.

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

v0 = np.array([1.0, 2.0])[:, np.newaxis]
v1 = np.array([1.0, -2.0])[:, np.newaxis]
lambda0 = -1.0 / 20.0
lambda1 = -3.0 / 20.0

t = np.linspace(0, 100, 100)
løsning = (30 * v0 * np.exp(lambda0 * t)
           + 30 * v1 * np.exp(lambda1 * t))

fig, ax = plt.subplots()
```

```
ax.set_aspect('equal')
ax.plot(*løsning,
        marker='o', markevery=10, markersize=4)
```

Vi kan også lave plots med andre startværdier.

```
v = np.hstack([v0, v1])

def sol(r0, r1, v, lambda0, lambda1, t):
    c = np.linalg.solve(v, np.array([r0, r1])[:, np.newaxis])
    return (c[0] * v[:, [0]] * np.exp(lambda0 * t)
            + c[1] * v[:, [1]] * np.exp(lambda1 * t))

t = np.linspace(0, 100, 100)

marks = dict(marker='o', markevery=10, markersize=3)

fig, ax = plt.subplots()
ax.set_aspect('equal')
for s in np.linspace(10, 60, 6):
    ax.plot(*sol(s, 0, v, lambda0, lambda1, t), **marks)
    ax.plot(*sol(s, 60, v, lambda0, lambda1, t), **marks)
for s in np.linspace(10, 60, 6):
    ax.plot(*sol(0, s, v, lambda0, lambda1, t), **marks)
    ax.plot(*sol(60, s, v, lambda0, lambda1, t), **marks)
```

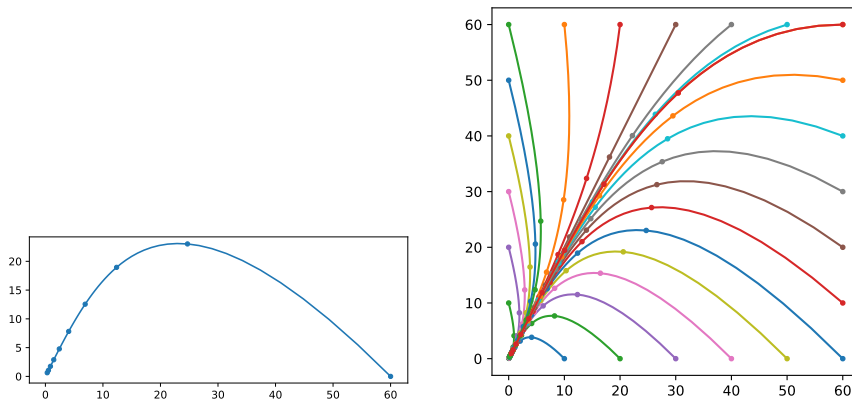
Disse visses til højre i figur 22.2.

△

22.2 Komplekse egenverdier

Selvom A har reelle indgange, kan egenverdierne godt være komplekse tal. Det har vi set i det forrige kapitel. Vi vil dog gerne følge metoden ovenfor. Så vi begynder med at kikke på ligning (22.1) i det komplekse tilfælde. Hvis $\lambda = a + ib$, $a, b \in \mathbb{R}$, har vi brug for funktioner $t \mapsto y(t) \in \mathbb{C}$, der opfylder

$$y'(t) = \lambda y(t) = (a + ib)y(t).$$



Figur 22.2: Saltmængden i beholder A mod beholder B.

Heldigvis findes sådanne funktioner. Betragt

$$y(t) = e^{at}(\cos(bt) + i \sin(bt)).$$

Vi har

$$\begin{aligned} y'(t) &= \frac{d}{dt}(e^{at}(\cos(bt) + i \sin(bt))) \\ &= \left(\frac{d}{dt}(e^{at})\right)(\cos(bt) + i \sin(bt)) + e^{at} \frac{d}{dt}(\cos(bt) + i \sin(bt)) \\ &= ae^{at}(\cos(bt) + i \sin(bt)) + e^{at}(-b \sin(bt) + ib \cos(bt)) \\ &= e^{at}((a \cos(bt) - b \sin(bt)) + i(a \sin(bt) + b \cos(bt))) \\ &= e^{at}(a + ib)(\cos(bt) + i \sin(bt)) \\ &= (a + ib)y(t), \end{aligned}$$

som er netop det vi ønsker.

Vi definerer $e^{\lambda t}$ for kompleks $\lambda = a + ib$ til at være

$$e^{\lambda t} = e^{at}(\cos(bt) + i \sin(bt)),$$

og kalder dette *den komplekse eksponentialfunktion*.

Vi har lige vist at

$$\frac{d}{dt}e^{\lambda t} = \lambda e^{\lambda t}.$$

Ved hjælp af standard trigonometriske identiteter kan man desuden tjekke at

$$e^{\lambda+\mu} = e^{\lambda} e^{\mu}$$

for komplekse tal λ og μ . Så denne eksponentialfunktion for komplekse tal har mange af de samme egenskaber, som dem vi kender for reelle tal.

Bemærk at vi har nu den smukke *Eulers identitet*

$$e^{i\pi} = -1,$$

da $e^{i\pi} = \cos(\pi) + i \sin(\pi) = -1 + i0 = -1$.

Hvis $A \in \mathbb{R}^{n \times n}$ har en kompleks egen værdi $\lambda = a + ib \in \mathbb{C}$ med egenvektor $v \in \mathbb{C}^n$, så har vi $Av = \lambda v$. Det følger at $\bar{\lambda} = a - ib$ er også en egen værdi for A over \mathbb{C} med egenvektor \bar{v} , da $\bar{A} = A$ giver

$$A\bar{v} = \bar{A}\bar{v} = \overline{Av} = \overline{\lambda v} = \bar{\lambda}\bar{v}.$$

Så opfylder både $e^{\lambda t}v$ og $e^{\bar{\lambda}t}\bar{v}$ systemet $y' = Ay$. De er dog løsninger i \mathbb{C}^n .

For at få reelle løsninger kan vi skrive $\lambda = a + ib$, $a, b \in \mathbb{R}$, og tilsvarende $v = w + iz$, med $w, z \in \mathbb{R}^n$. De følgende to lineære kombinationer af løsningerne $e^{\lambda t}v$ og $e^{\bar{\lambda}t}\bar{v}$,

$$\begin{aligned} u_0(t) &= \frac{1}{2}(e^{\lambda t}v + e^{\bar{\lambda}t}\bar{v}) = \operatorname{Re}(e^{\lambda t}v), \\ u_1(t) &= \frac{1}{2i}(e^{\lambda t}v - e^{\bar{\lambda}t}\bar{v}) = \operatorname{Im}(e^{\lambda t}v), \end{aligned}$$

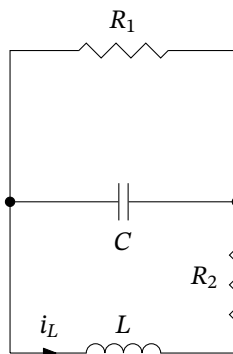
er løsninger til $y' = Ay$ i \mathbb{R}^n . Ved at bruge a, b og w, z , har vi $e^{\lambda t}v = e^{at}(\cos(bt) + i \sin(bt))(w + iz)$, og vi finder at

$$\begin{aligned} u_0(t) &= e^{at}(\cos(bt)w - \sin(bt)z), \\ u_1(t) &= e^{at}(\sin(bt)w + \cos(bt)z). \end{aligned}$$

Eksempel 22.4. Betragt kredsløbet i figur 22.3 med $R_1 = 5 \Omega$, $R_2 = 0,8 \Omega$, $L = 0,4 \text{ H}$, $C = 0,1 \text{ F}$. Den elektriske strøm i_L igennem L og spændingsfaldet v_C opfylder systemet

$$\begin{bmatrix} i'_L(t) \\ v'_C(t) \end{bmatrix} = \begin{bmatrix} -R_2/L & -1/L \\ 1/C & -1/(R_1 C) \end{bmatrix} \begin{bmatrix} i_L(t) \\ v_C(t) \end{bmatrix}.$$

Sættes $y(t) = (i_L(t), v_C(t))$, har vi



Figur 22.3: Elektrisk kredsløb.

$$y'(t) = Ay(t), \quad A = \begin{bmatrix} -2,0 & -2,5 \\ 10,0 & -2,0 \end{bmatrix}.$$

Det karakteristiske polynomium for A , $p(\lambda) = \det(A - \lambda I_2) = a\lambda^2 + b\lambda + c$ har koefficienter

```
a = 1.0
b = -2 * (-2.0)
c = (-2.0) * (-2.0) - (-2.5) * (10.0)
print(a, b, c)
```

```
1.0 4.0 29.0
```

Den tilsvarende diskriminant Δ er

```
diskriminant = b**2 - 4 * a * c
print(diskriminant)
```

```
-100.0
```

som er negative. Det følger at en kompleks egen værdi er $(-b + i\sqrt{-\Delta})/(2a)$:

```
lambda_c = (-b + 1.0j * np.sqrt(-diskriminant))/(2 * a)
print(lambda_c)
```

```
(-2+5j)
```

Dvs. $\lambda = -2 + 5i$. Vi finder en tilsvarende egenvektor af formen $v = (z, 1)$:

```
a_mat = np.array([[-2.0, -2.5],
                  [10.0, -2.0]])
a_mat_c = a_mat - lambda_c * np.eye(2)
print(a_mat_c)
v_c = np.array([-a_mat_c[0,1]/a_mat_c[0,0],
                1.0])[:, np.newaxis]
print(v_c)
```

```
[[ 0. -5.j -2.5+0.j]
 [10. +0.j  0. -5.j]]
[[0.+0.5j]
 [1.+0.j ]]
```

Dvs. $v = (i/2, 1)$. Dette giver de følgende komplekse løsninger

$$ve^{\lambda t} = \begin{bmatrix} i/2 \\ 1 \end{bmatrix} e^{(-2+5i)t}, \quad \bar{v}e^{\bar{\lambda}t} = \begin{bmatrix} -i/2 \\ 1 \end{bmatrix} e^{(-2-5i)t}.$$

Da

$$v = \begin{bmatrix} i/2 \\ 1 \end{bmatrix} = \begin{bmatrix} 0 \\ 1 \end{bmatrix} + i \begin{bmatrix} 1/2 \\ 0 \end{bmatrix} =: w + iz,$$

er de tilsvarende reelle løsninger

$$u_0(t) = e^{-2t} \left(\cos(5t) \begin{bmatrix} 0 \\ 1 \end{bmatrix} - \sin(5t) \begin{bmatrix} 1/2 \\ 0 \end{bmatrix} \right) = e^{-2t} \begin{bmatrix} -(1/2) \sin(5t) \\ \cos(5t) \end{bmatrix},$$

$$u_1(t) = e^{-2t} \left(\sin(5t) \begin{bmatrix} 0 \\ 1 \end{bmatrix} + \cos(5t) \begin{bmatrix} 1/2 \\ 0 \end{bmatrix} \right) = e^{-2t} \begin{bmatrix} (1/2) \cos(5t) \\ \sin(5t) \end{bmatrix}.$$

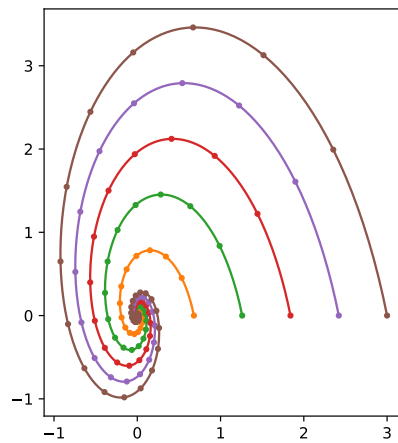
Den generelle løsning til systemer er så $y(t) = c_0 u_0(t) + c_1 u_1(t)$.

Givet begyndelsesværdierne $y(0) = (i_L(0), v_C(0))$, kan vi løse

$$y(0) = c_0(0, 1) + c_1(1/2, 0)$$

for c_0, c_1 . Vi kan plotte disse løsninger som før

```
def sol_c(r0, r1, v_c, lambda_c, t):
    w = v_c.real
```



Figur 22.4: Løsningen til kredsløb.

```

z = v_c.imag
v = np.hstack([w, z])
a = lambda_c.real
b = lambda_c.imag
c = np.linalg.solve(v, np.array([r0, r1])[:, np.newaxis])
return (c[0] * np.exp(a*t)
        * (np.cos(b*t) * w - np.sin(b*t) * z)
        + c[1] * np.exp(a*t)
        * (np.sin(b*t) * w + np.cos(b*t) * z))

t = np.linspace(0, 4, 1000)

marks = dict(marker='o', markevery=20, markersize=3)

fig, ax = plt.subplots()
ax.set_aspect('equal')
for s in np.linspace(0.1, 3, 6):
    ax.plot(*sol_c(s, 0, v_c, lambda_c, t), **marks)

```

△

22.3 Højere ordens lineære differentiaalligninger

Højere ordens differentiaalligninger kan omskrives til systemer af førsteordensligninger. Lad os vise dette for lineære differentiaalligninger af orden 2.

Betragt

$$y''(t) + ay'(t) + by(t) = 0, \quad y(t) \in \mathbb{R}. \quad (22.3)$$

Vi indfører nye funktioner $y_0(t)$ og $y_1(t)$ givet ved

$$y_0(t) = y(t), \quad y_1(t) = y'(t).$$

Ligning (22.3) kan nu skrives ved hjælp af $y_0(t)$, $y_1(t)$ og den første ordens afledede $y_1'(t)$:

$$y_1'(t) + ay_1(t) + by_0(t) = 0. \quad (22.4)$$

Husker vi definitionen på $y_1(t)$, får vi også en ligning for den første ordens afledede $y_0'(t)$:

$$y_0'(t) = y'(t) = y_1(t). \quad (22.5)$$

Ligninger (22.4) og (22.5) kan nu omskrives, som et førsteordenssystem

$$\begin{aligned} y_0'(t) &= y_1(t), \\ y_1'(t) &= -by_0(t) - ay_1(t). \end{aligned}$$

Vi skriver dette i matrixform, som

$$\begin{bmatrix} y_0'(t) \\ y_1'(t) \end{bmatrix} = \begin{bmatrix} 0 & 1 \\ -b & -a \end{bmatrix} \begin{bmatrix} y_0(t) \\ y_1(t) \end{bmatrix}. \quad (22.6)$$

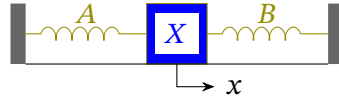
Sådan et system kan nu løses ved hjælp af de metoder vi indførte tidligere i kapitlet.

Bemærk at en begyndelsesværdi for systemet (22.6) er en angivelse af $y_0(t_0) = y(t_0)$ og $y_1(t_0) = y'(t_0)$, dvs. en angivelse af y og dens første ordens afledede i tid t_0 . Derfor forventer vi at løsninger er entydigt bestemt af sådanne begyndelsesværdier.

Eksempel 22.5. Betragt figur 22.5 hvor en klods X af masse m er forbundet med fjedre A og B med fjederkonstanter k_A og k_B . Klodsen kan bevæge sig til venstre og til højre. Vi angiver dens position ved et vandret koordinat x_0 , som er 0 når X er i dens ligevægtsstilling, og er positiv når X bevæger sig til højre.

Bevægelsesligningerne følger fra Newtons anden lov, som siger

$$\text{masse} \times \text{acceleration} = \text{kraft}.$$



Figur 22.5: System med klods og fjedre.

Fjeder A bidrager med en kraft $-k_A x(t)$: minus fortegnet skyldes at $x(t)$ måles til højre, men sammentrækningskraften er til venstre for $x(t)$ positiv. Ligeledes bidrager fjeder B med kraften $-k_B x(t)$: for $x(t)$ positiv, skubber fjedren klodsens til venstre. I alt får vi

$$mx''(t) = -k_A x(t) - k_B x(t) = -(k_A + k_B)x(t).$$

Lad os tage $m = 1$, $k_A = k_B = 1$. Vi sætter $y_0(t) = x(t)$ og $y_1(t) = x'(t)$ og får

$$\begin{aligned} y_0'(t) &= x'(t) = y_1(t), \\ y_1'(t) &= x''(t) = -2x(t) = -2y_0(t). \end{aligned}$$

Så har vi et førsteordenssystem

$$\begin{bmatrix} y_0'(t) \\ y_1'(t) \end{bmatrix} = \begin{bmatrix} 0 & 1 \\ -2 & 0 \end{bmatrix} \begin{bmatrix} y_0(t) \\ y_1(t) \end{bmatrix},$$

dvs.

$$\begin{bmatrix} y_0'(t) \\ y_1'(t) \end{bmatrix} = A \begin{bmatrix} y_0(t) \\ y_1(t) \end{bmatrix} \quad \text{med } A = \begin{bmatrix} 0 & 1 \\ -2 & 0 \end{bmatrix}.$$

For at løse systemet, beregne vi først det karakteristiske polynomium for koefficientmatricen A :

$$\det(A - \lambda I_2) = \det \begin{bmatrix} -\lambda & 1 \\ -2 & -\lambda \end{bmatrix} = \lambda^2 + 2$$

Polynomiet har rødder $\pm i\sqrt{2} \in \mathbb{C}$. Nu finder vi en kompleks egenvektor v for $\lambda = i\sqrt{2}$:

$$\begin{bmatrix} -\lambda & 1 \\ -2 & -\lambda \end{bmatrix} = \begin{bmatrix} -i\sqrt{2} & 1 \\ -2 & -i\sqrt{2} \end{bmatrix} \sim \begin{bmatrix} -i\sqrt{2} & 1 \\ 0 & 0 \end{bmatrix},$$

giver en egenvektor $v = (1, i\sqrt{2}) \in \mathbb{C}^2$. Vi får en kompleks løsning $e^{\lambda t} v = e^{i\sqrt{2}t} (1, i\sqrt{2}) = (\cos(\sqrt{2}t) + i \sin(\sqrt{2}t))(1, i\sqrt{2})$. Fra afsnit 22.2 har vi at den

generelle løsning er nu en lineær kombination af de reelle og imaginære del af den komplekse løsning:

$$\begin{aligned} \begin{bmatrix} x(t) \\ x'(t) \end{bmatrix} &= \begin{bmatrix} y_0(t) \\ y_1(t) \end{bmatrix} = c_0(\cos(\sqrt{2}t) \operatorname{Re}(v) - \sin(\sqrt{2}t) \operatorname{Im}(v)) \\ &\quad + c_1(\sin(\sqrt{2}t) \operatorname{Re}(v) + \cos(\sqrt{2}t) \operatorname{Im}(v)) \\ &= \begin{bmatrix} c_0 \cos(\sqrt{2}t) + c_1 \sin(\sqrt{2}t) \\ -c_0 \sin(\sqrt{2}t)\sqrt{2} + c_1 \cos(\sqrt{2}t)\sqrt{2} \end{bmatrix}. \end{aligned}$$

Så vi har

$$x(t) = c_0 \cos(\sqrt{2}t) + c_1 \sin(\sqrt{2}t),$$

dvs. en svingning med svingningstid $2\pi/\sqrt{2} \approx 4,44$ s.

△

Kapitel 23

Matrixfaktorisering fra egenværdier

Egenværdier og egenvektorer giver anledning til forskellige matrixfaktorisering. Her vil vi kikke på nogle enkle betingelser der garanterer eksistens af sådanne faktoriseringer.

23.1 Eksistens af egenværdier

Polynomiet

$$p(x) = x^2 + 1$$

har ingen rødder i \mathbb{R} . Men tillader vi komplekse tal så har den to rødder $i, -i$ i \mathbb{C} . At man har bedre kontrol over eksistens af rødder over komplekse tal er indholdet i:

Sætning 23.1 (Algebraens fundamentalsætning). *Ethvert polynomium $p(x) = a_0 + a_1x + \dots + a_nx^n$ har en rod i \mathbb{C} .* \square

Givet en rod z_0 , kan vi så skrive $p(x) = (x - z_0)q(x)$ med $q(x)$ er polynomium af grad $n - 1$ og anvender sætningen igen på $q(x)$. Vi får til sidst at $p(x)$ kan faktorerises, som

$$p(x) = a_n(x - z_0)(x - z_1) \dots (x - z_{n-1})$$

for nogle $z_0, z_1, \dots, z_{n-1} \in \mathbb{C}$.

Egenværdier er netop rødder til det karakteristiske polynomium $p(\lambda) = \det(A - \lambda I_n)$, så vi får som konsekvens af sætning [23.1](#) at

Proposition 23.2. *Enhver matrix $A \in \mathbb{C}^{n \times n}$ har en egenværdi.* \square

Dette siger for enhver $A \in \mathbb{C}^{n \times n}$ findes der en $\lambda \in \mathbb{C}$ og $v \in \mathbb{C}^n$, med $v \neq 0$, således at

$$Av = \lambda v.$$

Bemærk at vi kan altid erstatte v med $v/\|v\|_2$, og får dermed en egenvektor af længde 1, skulle det ønskes.

23.2 Determinant og egenværdier

Lad os notere nogle egenskaber af determinanten og giver nogle simple konsekvenser for bestemmelse af egenværdier. For det første opfører determinanten sig godt i forhold til matrixmultiplikation.

Sætning 23.3. *For A og B ($n \times n$)-matricer gælder $\det(AB) = \det(A) \det(B)$.* \square

Vi giver et bevis for sætningen senere, se afsnit 23.5, side 282. Her er en vigtig anvendelse.

Proposition 23.4. *For $A = VBV^{-1}$ har A og B det samme karakteristiske polynomium, og så de samme egenværdier.*

Bevis. Bemærk at sætning 23.3 giver

$$\det(AB) = \det(A) \det(B) = \det(B) \det(A) = \det(BA).$$

Det følger at

$$\begin{aligned} \det(A - \lambda I_n) &= \det(VBV^{-1} - \lambda I_n) = \det(VBV^{-1} - V\lambda I_n V^{-1}) \\ &= \det(V(B - \lambda I_n)V^{-1}) = \det((B - \lambda I_n)V^{-1}V) \\ &= \det(B - \lambda I_n), \end{aligned}$$

dvs. at A og B har det samme karakteristiske polynomium. Da egenværdier er netop rødder af det karakteristiske polynomium, får vi det sidste påstand. \square

Proposition 23.5. *For en øvre triangulær matrix*

$$B = \begin{bmatrix} \lambda_0 & * & \dots & * \\ 0 & \lambda_1 & \dots & * \\ \vdots & & \ddots & \vdots \\ 0 & 0 & & \lambda_{n-1} \end{bmatrix}$$

er

$$\det(B) = \lambda_0 \lambda_1 \dots \lambda_{n-1}$$

og $\lambda_0, \lambda_1, \dots, \lambda_{n-1}$ er egenverdierne af B .

Igen giver vi beviset senere, se afsnit 23.5, side 278. Lad os kikke på et eksempel.

Eksempel 23.6. Matricen

$$A = \begin{bmatrix} 0,8 & 2,6 \\ -8,4 & 10,2 \end{bmatrix} = \begin{bmatrix} 1 & 3 \\ 2 & 1 \end{bmatrix} \begin{bmatrix} 6 & -10 \\ 0 & 5 \end{bmatrix} \begin{bmatrix} 1 & 3 \\ 2 & 1 \end{bmatrix}^{-1}$$

har samme egenverdier som

$$B = \begin{bmatrix} 6 & -10 \\ 0 & 5 \end{bmatrix}$$

Men B er øvre triangulær, så egenverdierne i B er blot dens diagonale indgange. Dvs. $\lambda_0 = 6$ og $\lambda_1 = 5$. △

23.3 Unitære matricer og Schurdekomponeringen

Standardindreproduktet i \mathbb{C}^n er

$$\langle u, v \rangle = \bar{v}^T u.$$

Som analog til ortogonale matricer i $\mathbb{R}^{n \times n}$ har vi følgende definition over \mathbb{C} .

Definition 23.7. En matrix $U \in \mathbb{C}^{n \times n}$ er *unitær* hvis

$$\bar{U}^T U = I_n.$$

Bemærk at U er unitær hvis og kun hvis U er invertibel med

$$U^{-1} = \bar{U}^T.$$

Det følger at $U \bar{U}^T = I_n$ også.

For generel U , kan vi skrive $U = [u_0 \mid u_1 \mid \dots \mid u_{n-1}]$, og danne en kompleks Grammatrix:

$$\overline{U}^T U = \begin{bmatrix} \overline{u_1}^T u_1 & \overline{u_1}^T u_2 & \dots & \overline{u_1}^T u_{n-1} \\ \overline{u_2}^T u_1 & \overline{u_2}^T u_2 & \dots & \overline{u_2}^T u_{n-1} \\ \vdots & \vdots & \ddots & \vdots \\ \overline{u_{n-1}}^T u_1 & \overline{u_{n-1}}^T u_2 & \dots & \overline{u_{n-1}}^T u_{n-1} \end{bmatrix}$$

Dette er den $(n \times n)$ -matrix med (i, j) -indgang $\langle u_j, u_i \rangle$. Det følger at $U \in \mathbb{C}^{n \times n}$ er unitær hvis og kun hvis dens søjler u_0, u_1, \dots, u_{n-1} er ortonormal i \mathbb{C}^n .

Sætning 23.8 (Schurdekomponering). *For enhver matrix $A \in \mathbb{C}^{n \times n}$ findes der en unitær matrix $U \in \mathbb{C}^{n \times n}$, $\overline{U}^T = U^{-1}$, således at $T = \overline{U}^T A U$ er øvre triangulær, dvs. $A = U T \overline{U}^T$.*

Bevis. Lad os først betragte tilfældet $n = 2$. Fra proposition 23.2 ved vi at matricen A har en egen værdi $\lambda_0 \in \mathbb{C}$ og en egenvektor $u_0 \in \mathbb{C}^2$. Sættes $v_0 = u_0 / \|u_0\|_2 = (z, w)$ fås en egenvektor af længde 1, så med $|z|^2 + |w|^2 = 1$.

Vi kan nu finde en ortonormal basis for \mathbb{C}^2 ved at bruge v_0 og vektoren $v_1 = (-\overline{w}, \overline{z})$. Vi har $\|v_1\|_2^2 = |w|^2 + |z|^2 = 1$ og $\langle v_0, v_1 \rangle = -wz + zw = 0$, så v_0, v_1 er ortonormal. Det følger at $U = [v_0 \mid v_1]$ er en unitær matrix.

Da $A v_0 = \lambda_0 v_0$ og U er invertibel, har vi

$$\begin{aligned} A U &= A [v_0 \mid v_1] = [A v_0 \mid A v_1] = [\lambda_0 v_0 \mid x_1] = [v_0 \mid v_1] \begin{bmatrix} \lambda_0 & z_1 \\ 0 & w_1 \end{bmatrix} \\ &= U \begin{bmatrix} \lambda_0 & z_1 \\ 0 & w_1 \end{bmatrix}, \end{aligned}$$

hvor $x_1 = A v_1$ og $(z_1, w_1) = U^{-1} x_1$. Vi har så at

$$\overline{U}^T A U = \begin{bmatrix} \lambda_0 & * \\ 0 & * \end{bmatrix},$$

en øvre triangulær, som ønsket.

For generel n , kan vi begynde på samme måde med en egen værdi λ_0 og en egenvektor v_0 af længde 1. Vi kan så brug Gram-Schmidt processen på v_0, e_0, \dots, e_{n-1} til at finde en ortonormal basis v_0, v_1, \dots, v_{n-1} af \mathbb{C}^n . Sættes $U_0 = [v_0 \mid v_1 \mid \dots \mid v_{n-1}]$ har vi en unitær matrix.

23.3 UNITÆRE MATRICER OG SCHURDEKOMPONERENGEN

Nu ser vi at

$$\begin{aligned} AU_0 &= A[v_0 \mid v_1 \mid \dots \mid v_{n-1}] = [Av_0 \mid Av_1 \mid \dots \mid Av_{n-1}] \\ &= [\lambda_0 v_0 \mid x_1 \mid \dots \mid x_{n-1}] = U_0(\overline{U}_0^T)[\lambda_0 v_0 \mid x_1 \mid \dots \mid x_{n-1}] \\ &= U_0 \begin{bmatrix} \lambda_0 & z_1 & \dots & z_{n-1} \\ 0 & b_{00} & \dots & b_{0,n-2} \\ \vdots & \vdots & & \vdots \\ 0 & b_{n-2,0} & \dots & b_{n-2,n-2} \end{bmatrix}, \end{aligned}$$

hvor i det sidste trin vi bruge at $\overline{v}_i^T v_0$ er 1, for $i = 0$, og 0 ellers. Matricen $B = (b_{ij}) \in \mathbb{C}^{n-1 \times n-1}$ har mindre størrelse end A , så per induktion kan vi skrive $B = VS\overline{V}^T$ med $V = (v_{ij}) \in \mathbb{C}^{n-1 \times n-1}$ unitær og $S = (s_{ij}) \in \mathbb{C}^{n-1 \times n-1}$ øvre triangulær. Sættes

$$U_1 = \begin{bmatrix} 1 & 0 & \dots & 0 \\ 0 & v_{00} & \dots & v_{0,n-2} \\ \vdots & \vdots & & \vdots \\ 0 & v_{n-2,0} & \dots & v_{n-2,n-2} \end{bmatrix} \quad \text{og} \quad T = \begin{bmatrix} \lambda_0 & z_1 & \dots & z_{n-1} \\ 0 & s_{00} & \dots & s_{0,n-2} \\ \vdots & & \ddots & \vdots \\ 0 & 0 & & s_{n-2,n-2} \end{bmatrix},$$

har vi at U_1 er unitær, T er øvre triangulær og

$$AU_0 = U_0 U_1 T \overline{U}_1^T,$$

så

$$AU_0 U_1 = U_0 U_1 T.$$

Men $U = U_0 U_1$ er unitær:

$$\begin{aligned} \overline{U}^T U &= \overline{U_0 U_1}^T U_0 U_1 = (\overline{U_0} \overline{U_1})^T U_0 U_1 \\ &= \overline{U_1}^T \overline{U_0}^T U_0 U_1 = \overline{U_1}^T I_n U_1 \\ &= I_n. \end{aligned}$$

Det følger at $AU = UT$, eller

$$T = \overline{U}^T AU,$$

som ønsket. □

23.4 Spektralsætningen

Vi kan få diagonaliseringsresultater hvis vi sætter ekstra krav på vores matrix. Vi giver først et resultat for reelle symmetriske matricer, og derefter en version af samme resultat for visse komplekse matricer.

Sætning 23.9 (Spektralsætningen for reelle symmetriske matricer). *For $A \in \mathbb{R}^{n \times n}$ symmetrisk findes der en ortogonal matrix $V \in \mathbb{R}^{n \times n}$ således at*

$$A = V \Lambda V^T$$

for en diagonal matrix $\Lambda = \text{diag}(\lambda_0, \dots, \lambda_{n-1}) \in \mathbb{R}^{n \times n}$. Specielt er A diagonaliserbar med reelle egenverdier $\lambda_0, \dots, \lambda_{n-1}$.

Inden vi går i gang med beviset lad os kikke på et par eksempler.

Eksempel 23.10. Betragt

$$A = \begin{bmatrix} 1 & 2 \\ 2 & 1 \end{bmatrix},$$

som er symmetrisk. Dens karakteristiske polynomium er $p(\lambda) = (1 - \lambda)^2 - 4 = \lambda^2 - 2\lambda - 3$, så de er to egenverdier $\lambda_0 = 3, \lambda_1 = -1$.

Vi kikker efter egenvektorer. Først for λ_0 :

$$A - \lambda_0 I_2 = \begin{bmatrix} 1-3 & 2 \\ 2 & 1-3 \end{bmatrix} = \begin{bmatrix} -2 & 2 \\ 2 & -2 \end{bmatrix} \sim \begin{bmatrix} -1 & 1 \\ 0 & 0 \end{bmatrix},$$

giver alle egenvektorer $v_0 = (x, y)$ for λ_0 har $x = y$. Vi får en enhedsvektor ved at sætter f.eks. $x = 1 = y$ og dele derefter med normen:

$$v_0 = \frac{1}{\|(1, 1)\|_2} \begin{bmatrix} 1 \\ 1 \end{bmatrix} = \begin{bmatrix} 1/\sqrt{2} \\ 1/\sqrt{2} \end{bmatrix}.$$

For λ_1 har vi

$$A - \lambda_1 I_2 = \begin{bmatrix} 1 - (-1) & 2 \\ 2 & 1 - (-1) \end{bmatrix} = \begin{bmatrix} 2 & 2 \\ 2 & 2 \end{bmatrix} \sim \begin{bmatrix} 1 & 1 \\ 0 & 0 \end{bmatrix}$$

så kan tage

$$v_1 = \frac{1}{\|(1, -1)\|_2} \begin{bmatrix} 1 \\ -1 \end{bmatrix} = \begin{bmatrix} 1/\sqrt{2} \\ -1/\sqrt{2} \end{bmatrix}.$$

Bemærk at $\langle v_0, v_1 \rangle = 0$, så

$$V = [v_0 \mid v_1]$$

er en ortogonal matrix, og vi har

$$A = V\Lambda V^T \quad \text{med } \Lambda = \begin{bmatrix} 3 & 0 \\ 0 & -1 \end{bmatrix}.$$

△

Eksempel 23.11. Betragt

$$A = \begin{bmatrix} 2 & 0 & 0 \\ 0 & 1 & -1 \\ 0 & -1 & 1 \end{bmatrix},$$

som er symmetrisk med karakteristisk polynomium

$$\begin{aligned} p(\lambda) &= \det(A - \lambda I_3) = \det \begin{bmatrix} 2-\lambda & 0 & 0 \\ 0 & 1-\lambda & -1 \\ 0 & -1 & 1-\lambda \end{bmatrix} \\ &= (2-\lambda)((1-\lambda)^2 - (-1)^2) = (2-\lambda)(\lambda^2 - 2\lambda) \\ &= \lambda(2-\lambda)^2, \end{aligned}$$

så A har egenverdier 0 og 2.

For $\lambda = 0$, er

$$v_0 = \frac{1}{\|(0, 1, 1)\|_2} \begin{bmatrix} 0 \\ 1 \\ 1 \end{bmatrix} = \begin{bmatrix} 0 \\ 1/\sqrt{2} \\ 1/\sqrt{2} \end{bmatrix}$$

en egenvektor.

For $\lambda = 2$, har vi

$$A - 2I_3 = \begin{bmatrix} 0 & 0 & 0 \\ 0 & -1 & -1 \\ 0 & -1 & -1 \end{bmatrix} \sim \begin{bmatrix} 0 & 1 & 1 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix}.$$

Der er to frie variabler (x_0 og x_2), og dermed den generelle løsning er $v = (x_0, -x_2, x_2)$. Bemærk at alle disse vektorer er vinkelret på v_0 . Sættes $x_0 = 1$, $x_2 = 0$, får vi en enhedsvektor

$$v_1 = \begin{bmatrix} 1 \\ 0 \\ 0 \end{bmatrix}.$$

Vi mangler nu kun en enhedsvektor v_2 , som er både en egenvektor med egen­værdi $\lambda = 2$ og er vinkelret på v_1 . Egenvektorerne $v = (x_0, -x_2, x_2)$, som er vinkelret på v_1 har $x_0 = 0$ og så er generelt af formen $v = (0, -x_2, x_2)$. Ved at tage $x_2 = 1$ og dele med normen får vi en enhedsvektor

$$v_2 = \frac{1}{\|(0, -1, 1)\|_2} \begin{bmatrix} 0 \\ -1 \\ 1 \end{bmatrix} = \begin{bmatrix} 0 \\ -1/\sqrt{2} \\ 1/\sqrt{2} \end{bmatrix},$$

Nu er v_0, v_1, v_2 en ortonormal samling af egenvektorer.

Sættes

$$V = [v_0 \mid v_1 \mid v_2] = \begin{bmatrix} 0 & 1 & 0 \\ 1/\sqrt{2} & 0 & -1/\sqrt{2} \\ 1/\sqrt{2} & 0 & 1/\sqrt{2} \end{bmatrix}$$

fås en ortogonal matrix med

$$A = V\Lambda V^T, \quad \text{for } \Lambda = \begin{bmatrix} 0 & 0 & 0 \\ 0 & 2 & 0 \\ 0 & 0 & 2 \end{bmatrix}.$$

Vi bekræfter dette i python

```
import numpy as np

s = 1/np.sqrt(2.0)
v = np.array([[0.0, 1.0, 0.0],
              [s, 0, -s],
              [s, 0, s]])
egenværdier = np.array([0.0, 2.0, 2.0])

print(v @ np.diag(egenværdier) @ v.T)
```

```
[[ 2.  0.  0.]
 [ 0.  1. -1.]
 [ 0. -1.  1.]]
```

som er vores oprindelig matrix A . △

Lemma 23.12. Hvis $A \in \mathbb{R}^{n \times n}$ er symmetrisk, så har A en egen­værdi $\lambda \in \mathbb{R}$.

Bevis. Vi betragter A som en kompleks matrix $A \in \mathbb{C}^{n \times n}$, med $\bar{A} = A = A^T$. Proposition 23.2 giver at der findes en kompleks egen værdi $\lambda \in \mathbb{C}$. Lad $v \in \mathbb{C}^n$ være en tilhørende egenvektor, så $Av = \lambda v$ og $v \neq 0$.

Vi beregner $\bar{v}^T Av$ på to forskellige måder. Først har vi

$$\bar{v}^T Av = \bar{v}^T (Av) = \bar{v}^T \lambda v = \lambda \|v\|_2^2.$$

Desuden har vi

$$\bar{v}^T Av = (\bar{v}^T \bar{A}^T) v = (\overline{Av})^T v = (\overline{\lambda v})^T v = \bar{\lambda} \|v\|_2^2.$$

Disse to resultater må være ens, så vi har

$$\lambda \|v\|_2^2 = \bar{\lambda} \|v\|_2^2.$$

Men $v \neq 0$ medfører $\|v\|_2 \neq 0$, så vi kan dele med $\|v\|_2^2$ og få $\lambda = \bar{\lambda}$. Dette siger netop at $\lambda \in \mathbb{R}$. Nu er $\det(A - \lambda I_n) = 0$, så λ er en egen værdi for den reelle matrix $A \in \mathbb{R}^{n \times n}$. \square

Bevis for sætning 23.9. Vælg en egen værdi $\lambda_0 = \lambda \in \mathbb{R}$ for A . Lad $v = v_0 \in \mathbb{R}^n$ være en enhedsvektor med $Av_0 = \lambda_0 v_0$.

Vi kan finde en ortogonal matrix af formen $V_0 = [v_0 \mid * \mid \dots \mid *] \in \mathbb{R}^{n \times n}$, med v_0 , som den 0'te søjle. F.eks. man kan vælge en Householder matrix. Som i beviset for sætning 23.8, får vi

$$AV_0 = V_0 \begin{bmatrix} \lambda_0 & * & \dots & * \\ 0 & * & \dots & * \\ \vdots & \vdots & & \vdots \\ 0 & * & \dots & * \end{bmatrix}.$$

Da V_0 er ortogonal er $V^{-1} = V^T$ og det giver

$$A = V_0 \begin{bmatrix} \lambda_0 & * & \dots & * \\ 0 & * & \dots & * \\ \vdots & \vdots & & \vdots \\ 0 & * & \dots & * \end{bmatrix} V_0^T.$$

Men A er symmetrisk, $A = A^T$, og

$$A^T = V_0 \begin{bmatrix} \lambda_0 & 0 & \dots & 0 \\ * & * & \dots & * \\ \vdots & \vdots & & \vdots \\ * & * & \dots & * \end{bmatrix} V_0^T.$$

Sammenlignes disse to udtryk, har vi

$$A = V_0 \begin{bmatrix} \lambda_0 & 0 & \dots & 0 \\ 0 & b_{00} & \dots & b_{0,n-2} \\ \vdots & \vdots & & \vdots \\ 0 & b_{n-2,0} & \dots & b_{n-2,n-2} \end{bmatrix} V_0^T,$$

hvor $B = (b_{ij}) \in \mathbb{R}^{n-1 \times n-1}$ med $B^T = B$. Da B er symmetrisk kan vi finde en ortogonal matrix V_1 således at

$$A = V_0 V_1 \begin{bmatrix} \lambda_0 & 0 & 0 & \dots & 0 \\ 0 & \lambda_1 & 0 & \dots & 0 \\ 0 & 0 & c_{00} & \dots & c_{0,n-3} \\ \vdots & \vdots & \vdots & & \vdots \\ 0 & 0 & c_{n-3,0} & \dots & c_{n-3,n-3} \end{bmatrix} V_1^T V_0^T$$

med $C = (c_{ij}) \in \mathbb{R}^{n-2}$ symmetrisk. Forsættes fås

$$A = V_0 V_1 \dots V_{n-1} \begin{bmatrix} \lambda_0 & 0 & \dots & 0 \\ 0 & \lambda_1 & & 0 \\ \vdots & & \ddots & \\ 0 & 0 & & \lambda_{n-1} \end{bmatrix} V_0^T V_1^T \dots V_{n-1}^T = V \Lambda V^T$$

med $V = V_0 V_1 \dots V_{n-1}$ ortogonal. \square

Det tilsvarende resultat over komplekse tal bruger det følgende begreb: en matrix $A \in \mathbb{C}^{n \times n}$ er *hermitisk* hvis $\overline{A}^T = A$. Beviserne ovenfor kan udvides til dette tilfælde og vi får

Sætning 23.13 (Spektralsætningen for hermitiske matricer). *Hvis $A \in \mathbb{C}^{n \times n}$ er hermitisk, så er dens egenverdier reelle tal og der findes en unitær matrix $U \in \mathbb{C}^{n \times n}$ således at*

$$A = U \Lambda \overline{U}^T.$$

\square

23.5 Egenskaber af determinanter

I dette afsnit samler vi flere egenskaber af determinant og beviser nogle af de tidlige resultater vi har brugt. Et nøgle værktøj er at determinanter opfører sig godt i forhold til rækkeoperationer.

Proposition 23.14. *De følgende relationer gælder for determinanter af $(n \times n)$ -matricer A og B , hvor B fås fra A via den pågældende rækkeoperation.*

- (I) Hvis $A \sim_{R_i \leftrightarrow R_j} B$, $i \neq j$, så er $\det B = -\det A$.
- (II) Hvis $A \sim_{R_i \rightarrow sR_i} B$, så er $\det B = s \det A$.
- (III) Hvis $A \sim_{R_i \rightarrow R_i + tR_j} B$, så er $\det B = \det A$.

Bemærk at del (II) er gyldigt for alle skalarer s , men giver kun en elementær rækkeoperation hvis $s \neq 0$.

Vi har også

Proposition 23.15. *For A en $(n \times n)$ -matrix er $\det(A^T) = \det A$.*

Dette giver at vi kan udvikle determinanter via deres først søjle i stedet for dens første række:

Korollar 23.16. *For A en $(n \times n)$ -matrix gælder*

$$\det A = a_{00} \det(M_{00}) - a_{10} \det(M_{10}) + \cdots + (-1)^{n-1} a_{n-1,0} \det(M_{n-1,0}).$$

Bevis. Der er intet at vise for $n = 1$. For $n > 1$, har vi fra definition 21.12

$$\det A = a_{00} \det(M_{00}) - a_{01} \det(M_{01}) + \cdots + (-1)^{n-1} a_{0,n-1} \det(M_{0,n-1}).$$

Lad os anvende den formel på $B = A^T$. Vi har $b_{ij} = a_{ji}$ og at M_{ij}^B , som fås fra B ved at slette række i og søjle j , er M_{ji}^T . Dette giver

$$\begin{aligned} \det(A^T) &= \det B \\ &= b_{00} \det(M_{00}^B) - b_{01} \det(M_{01}^B) + \cdots + (-1)^{n-1} b_{0,n-1} \det(M_{0,n-1}^B) \\ &= a_{00} \det(M_{00}^T) - a_{10} \det(M_{10}^T) + \cdots + (-1)^{n-1} a_{n-1,0} \det(M_{n-1,0}^T). \end{aligned}$$

Men M_{ij} har størrelse $(n-1) \times (n-1)$, så per induktion er $\det(M_{ij}^T) = \det(M_{ij})$, og resultatet følger. \square

Eksempel 23.17. Kombineret med proposition 23.14 giver korollar 23.16 en relativ praktisk måde at regne determinanter. For eksempel,

$$\begin{aligned}
 & \det \begin{bmatrix} 2 & 3 & 1 & -1 \\ 1 & 1 & 0 & 2 \\ 0 & 2 & 3 & 1 \\ -1 & 2 & 1 & 2 \end{bmatrix} \\
 &=_{R_0 \leftrightarrow R_1} -\det \begin{bmatrix} 1 & 1 & 0 & 2 \\ 2 & 3 & 1 & -1 \\ 0 & 2 & 3 & 1 \\ -1 & 2 & 1 & 2 \end{bmatrix} =_{\substack{R_1 \rightarrow R_1 - 2R_0 \\ R_3 \rightarrow R_3 + R_0}} -\det \begin{bmatrix} 1 & 1 & 0 & 2 \\ 0 & 1 & 1 & -5 \\ 0 & 2 & 3 & 1 \\ 0 & 3 & 1 & 4 \end{bmatrix} \\
 &= -(1) \times \det \begin{bmatrix} 1 & 1 & -5 \\ 2 & 3 & 1 \\ 3 & 1 & 4 \end{bmatrix} + 0 \times \det(M_{10}) - 0 \times \det(M_{20}) + 0 \times \det(M_{30}) \\
 &=_{\substack{R_1 \rightarrow R_1 - 2R_0 \\ R_2 \rightarrow R_2 - 3R_0}} -\det \begin{bmatrix} 1 & 1 & -5 \\ 0 & 1 & 11 \\ 0 & -2 & 19 \end{bmatrix} = -(1) \times \det \begin{bmatrix} 1 & 11 \\ -2 & 19 \end{bmatrix} \\
 &= -(1 \times 19 - 11 \times (-2)) = -19 - 22 = -41.
 \end{aligned}$$

△

Bevis for proposition 23.5. Vi bruger korollar 23.16, til at udvikle via den første søjle:

$$\begin{aligned}
 \det B &= \lambda_0 \det(M_{00}) - 0 \times \det(M_{10}) + \cdots + (-1)^{n-1} 0 \times \det(M_{n-1,0}) \\
 &= \lambda_0 \det(M_{00}) = \lambda_0 \det \begin{bmatrix} \lambda_1 & \cdots & * \\ & \ddots & \vdots \\ 0 & & \lambda_{n-1} \end{bmatrix} \\
 &= \lambda_0 \lambda_1 \det \begin{bmatrix} \lambda_2 & \cdots & * \\ & \ddots & \vdots \\ 0 & & \lambda_{n-1} \end{bmatrix} = \cdots = \lambda_0 \lambda_1 \cdots \lambda_{n-1}.
 \end{aligned}$$

For det karakteristiske polynomium har vi så

$$\begin{aligned} p(\lambda) &= \det(B - \lambda I_n) = \det \begin{bmatrix} \lambda_0 - \lambda & * & \dots & * \\ 0 & \lambda_1 - \lambda & \dots & * \\ \vdots & & \ddots & \vdots \\ 0 & 0 & & \lambda_{n-1} - \lambda \end{bmatrix} \\ &= (\lambda_0 - \lambda)(\lambda_1 - \lambda) \dots (\lambda_{n-1} - \lambda), \end{aligned}$$

som ønsket. \square

Bevis for proposition 23.14. Vi skal regne effekten af rækkeoperationer på determinanten.

Vi begynder med at vise del (I) for $R_0 \leftrightarrow R_1$. Vi definitionen

$$\begin{aligned} \det A &= a_{00} \det(M_{00}) - a_{01} \det(M_{01}) + \dots + (-1)^{n-1} a_{0,n-1} \det(M_{0,n-1}) \\ &= \sum_{j=0}^{n-1} (-1)^j a_{0j} \det(M_{0j}). \end{aligned}$$

Nu skal vi anvende denne definition på hvert M_{0j} . Lad $N_{ia;jb}$ være delmatricen af A størrelsen $(n-2) \times (n-2)$, som fås ved at slette rækker i og a samt søjler j og b fra A . Den første række i M_{0j} er $a_{10}, \dots, a_{1,j-1}, a_{1,j+1}, \dots, a_{1,n-1}$. Så har vi

$$\det(M_{0j}) = \sum_{k=0}^{j-1} (-1)^k a_{1k} \det(N_{01;jk}) - \sum_{\ell=j+1}^{n-1} (-1)^\ell a_{1\ell} \det(N_{01;j\ell}).$$

Bemærk at $N_{01;jb} = N_{01;b,j}$, så for optræder $\det(N_{01;jk})$ to steder i $\det A$, fra leddene $a_{0j} \det(M_{0j})$ og $a_{0k} \det(M_{0k})$. Koefficienten af $\det(N_{01;jk})$ i $\det A$ er så

$$(-1)^j a_{0j} (-1)^k a_{1k} - (-1)^k a_{0k} (-1)^j a_{1j} = (-1)^{j+k} (a_{0j} a_{1k} - a_{0k} a_{1j}). \quad (23.1)$$

Når vi bytter række 0 med række 1 er $N_{01;jk}$ uændret, og (23.1) skifter fortegn. Dette giver denne version af del (I).

For at få del (I) generelt, bemærk først at for tilfældet $R_i \leftrightarrow R_{i+1}$ kan vi skrive $\det A$ som en sum af led af formen $(-1)^r a_{0p} a_{1q} \dots a_{i-1,t}$ gange determinanten af en kvadratisk undermatrix N af $A_{[i:,:]}$. Når vi bytter R_i med R_{i+1} bytter vi den 0'te og 1'te række af N , så $\det N \mapsto -\det N$. Det følger at $\det A \mapsto -\det A$.

For det generel tilfælde $R_i \leftrightarrow R_j$ med $i < j$ fås ved den følgende række af operationer af typen $R_k \leftrightarrow R_{k+1}$: ryk R_i trinvis ned til R_j , derefter ryk R_j trinvis op til R_i , som illustreres for $R_1 \leftrightarrow R_4$ ved

$$\begin{array}{c}
 \begin{bmatrix} A_{[0,:]} \\ \textcolor{red}{A}_{[1,:]} \\ A_{[2,:]} \\ A_{[3,:]} \\ \textcolor{blue}{A}_{[4,:]} \\ A_{[5,:]} \end{bmatrix} \xrightarrow{\sim R_1 \leftrightarrow R_2} \begin{bmatrix} A_{[0,:]} \\ A_{[2,:]} \\ \textcolor{red}{A}_{[1,:]} \\ A_{[3,:]} \\ \textcolor{blue}{A}_{[4,:]} \\ A_{[5,:]} \end{bmatrix} \xrightarrow{\sim R_2 \leftrightarrow R_3} \begin{bmatrix} A_{[0,:]} \\ A_{[2,:]} \\ A_{[3,:]} \\ \textcolor{red}{A}_{[1,:]} \\ \textcolor{blue}{A}_{[4,:]} \\ A_{[5,:]} \end{bmatrix} \\
 \xrightarrow{\sim R_3 \leftrightarrow R_4} \begin{bmatrix} A_{[0,:]} \\ A_{[2,:]} \\ A_{[3,:]} \\ \textcolor{blue}{A}_{[4,:]} \\ \textcolor{red}{A}_{[1,:]} \\ A_{[5,:]} \end{bmatrix} \xrightarrow{\sim R_2 \leftrightarrow R_3} \begin{bmatrix} A_{[0,:]} \\ A_{[2,:]} \\ \textcolor{blue}{A}_{[4,:]} \\ A_{[3,:]} \\ \textcolor{red}{A}_{[1,:]} \\ A_{[5,:]} \end{bmatrix} \xrightarrow{\sim R_1 \leftrightarrow R_2} \begin{bmatrix} A_{[0,:]} \\ \textcolor{blue}{A}_{[4,:]} \\ A_{[2,:]} \\ A_{[3,:]} \\ \textcolor{red}{A}_{[1,:]} \\ A_{[5,:]} \end{bmatrix}.
 \end{array}$$

Hver af disse operationer skifter fortegnet på $\det A$. Det sker $(j-i) + (j-i-1)$ gange, da rækken i flyttes ned $j-i$ gange, og derefter række j , som er nu i plads $j-1$, flyttes op $j-i-1$ gange. Dette er et ulig antal fortegnsskift, så i alt har $\det A \mapsto -\det A$.

For del (II), har vi for $i = 0$, at

$$\begin{aligned}
 \det B &= (sa_{00}) \det(M_{00}) - (sa_{01}) \det(M_{01}) + \cdots + (-1)^{n-1} (sa_{n-1,0}) \\
 &= s \det A.
 \end{aligned}$$

For generel i , er $A \sim_{R_i \rightarrow sR_i} B$ det samme som

$$A \sim_{R_0 \leftrightarrow R_i} A' \sim_{R_0 \rightarrow sR_0} A'' \sim_{R_0 \leftrightarrow R_i} B.$$

Under disse operationer har

$$\det A \mapsto -\det A \mapsto -s \det A \mapsto s \det A,$$

så $\det B = s \det A$.

For del (III), kikke vi først på tilfældet $i = 0$. Vi har

$$\begin{aligned}
 \det B &= (a_{00} + ta_{j0}) \det(M_{00}) - (a_{01} + ta_{j1}) \det(M_{01}) \\
 &\quad + \cdots + (-1)^{n-1} (a_{0,n-1} + ta_{j,n-1}) \det(M_{0,n-1}) \\
 &= \det A + t \det C,
 \end{aligned}$$

hvor C er A med række 0 erstattet af række j . Dvs. i C er række 0 og række j ens. Matricen C er uændret under $R_0 \leftrightarrow R_j$, men fra del (I) skifter dens determinant fortegn. Vi har så at $\det C = 0$, og har vist del (III) for $i = 0$.

For general i gør vi lige som i del (II), byt række i med række 0, udføre rækkeoperationen på række 0, og så byt række 0 tilbage til række i . Determinant skifter fortegn i alt to gange under den følge af operationer, så er uændret til sidst. Dette giver del (III). \square

Husk at elementære rækkeoperationer kan udføres som matrixmultiplikation ved en elementær matrix, se proposition 7.11.

Lemma 23.18. *Hvis E er en elementærmatrix svarende til en elementær rækkeoperation, så er $\det(EA) = \det(E) \det(A)$.*

Bevis. Som i beviset overfor kan vi reducere til tilfældende hvor $i = 0$ og $j = 1$. Vores elementære matricer har så formen

$$E = \begin{bmatrix} E' & 0 \\ 0 & I_{n-2} \end{bmatrix},$$

hvor E' er en elementær matrix af størrelse (2×2) . Vi har $\det E = \det E'$, og for hver type rækkeoperation har vi de følgende determinanter.

Type (I):

$$E' = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix}, \quad \det E = \det E' = -1.$$

Type (II):

$$E' = \begin{bmatrix} s & 0 \\ 0 & 1 \end{bmatrix}, \quad \det E = \det E' = s.$$

Type (III):

$$E' = \begin{bmatrix} 1 & t \\ 0 & 1 \end{bmatrix}, \quad \det E = \det E' = 1.$$

Disse skaleringsfaktor er de samme, som giver ændringerne i $\det A$ i proposition 23.14, og vi har resultatet. \square

Bevis for sætning 21.14. Vi skal vise at A er invertibel hvis og kun hvis $\det A \neq 0$.

A er en kvadratisk matrix, så den kan rækkereduceres til dens reduceret echelonform F . Køres disse operationer baglæns får vi A via rækkeoperationer fra F . Vi har så nogle elementære matricer E_0, E_1, \dots, E_{k-1} således at

$$A = E_0 E_1 \dots E_{k-1} F. \quad (23.2)$$

Fra lemma 23.18 har vi

$$\det A = \det(E_0) \det(E_1) \dots \det(E_{k-1}) \det(F).$$

Men hvert $\det(E_i)$ er forskellig fra 0, så $\det A = 0$ hvis og kun hvis $\det F = 0$.

Hvis F har en nulrække er A ikke invertibel. Vi har $\det F = 0$, så $\det A = 0$.

Hvis F ikke har en nulrække er alle søjle pivot søjle, A er invertibel og $F = I_n$. Det følger at $\det F \neq 0$ og så $\det A \neq 0$, som ønsket. \square

Bevis for sætning 23.3. Vi vil vise at $\det(AB) = \det(A) \det(B)$.

Som i beviset før, skriv

$$A = E_0 E_1 \dots E_{k-1} F,$$

som produkt af elementære matricer E_i og en reduceret echelonmatrix F .

Vi påstår at $\det(FB) = \det(F) \det(B)$. Hvis $F = I_n$, så er $\det F = 1$, og lignen gælder. Hvis F er ikke I_n , er dens sidste række nul. Så er den sidste række af FB også nul, og FB er ikke invertibel, og dermed er $\det(FB) = 0 = \det(F) \det(B)$.

Nu har vi

$$\begin{aligned} \det(AB) &= \det(E_0 E_1 \dots E_{k-1} FB) = \det(E_0 E_1 \dots E_{k-1}) \det(FB) \\ &= \det(E_0 E_1 \dots E_{k-1}) \det(F) \det(B) = \det(A) \det(B), \end{aligned}$$

som ønsket. \square

Bevis for proposition 23.15. Skriv igen A i formen (23.2):

$$A = E_0 E_1 \dots E_{k-1} F.$$

Vi har nu

$$A^T = F^T E_{k-1}^T \dots E_1^T.$$

Vi bemærker at for hver type elementær matrix, har vi $\det(E_i^T) = \det(E_i)$ via direkte udregning.

For $F = I_n$ er $F^T = I_n$, så $\det(F^T) = \det F$.

Hvis $F \neq I_n$, er den sidste række af F nul, så den sidste søjle af F^T er nul. Dette betyder der er højst $n - 1$ pivot søjle i F^T , så F^T er ikke invertibel, og $\det(F^T) = 0 = \det(F)$.

I begge tilfælde har vi $\det(F^T) = \det F$ og så

$$\begin{aligned} \det(A^T) &= \det(F^T E_{k-1}^T \dots E_1^T) = \det(F^T) \det(E_{k-1}^T) \dots \det(E_1^T) \\ &= \det(E_1) \dots \det(E_{k-1}) \det(F) = \det(E_1 \dots E_{k-1} F) = \det A, \end{aligned}$$

som ønsket. \square

Kapitel 24

Potensmetoden og inverspotensmetoden for egenværdier

For generel $(n \times n)$ -matricer findes der ingen algebraiske formel for rødder af det karakteristiske polynomium. Numeriske metoder for egenværdier og egenvektorer er derfor nødt til bruger iteration og kan kun beregne tilnærmelser. I dette kapitel kikker vi på to metoder der tillader beregning af en enkelt egenværdi og en tilhørende egenvektor. Den første kan kun finde den egenværdi der har størst numerisk værdi. Den anden metode finder en egenværdi i nærheden af et første gæt. Vi vil diskutere disse metoder anvendt på symmetriske matricer, men metoderne kan også bruges på andre matricer. Fordelen med symmetriske matricer er at vi har spektralsætningen, sætning 23.9, der garanterer at matricen faktisk har en reel egenværdi.

24.1 Potensmetoden

Lad $A \in \mathbb{R}^{n \times n}$ være en kvadratisk matrix. Potensmetoden er den følgende algoritme:

POTENSMETODEN(A)

```

1 Begynd med et vilkårligt  $w^{(0)}$  af længde 1
2 for  $k \in \{0, 1, 2, \dots\}$ :
3      $v = Aw^{(k-1)}$ 
4      $w^{(k)} = v / \|v\|_2$ 
5      $\lambda^{(k)} = (w^{(k)})^T Aw^{(k)}$ 

```

Under visse betingelser konvergerer $w^{(k)}$ til en egenvektor for A , og $\lambda^{(k)}$ konvergerer til dens egen værdi.

Eksempel 24.1. Lad os regne i python

```

import numpy as np

rng = np.random.default_rng()

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

w = rng.standard_normal((a.shape[0], 1))
n = 20
lambda_out = np.empty(n)

for i in range(n):
    v = a @ w
    w = v / np.linalg.norm(v)
    lambda_out[i] = w.T @ (a @ w)

np.set_printoptions(linewidth = 60)
print(lambda_out)

```

```

[4.93108775  5.15486396  5.20144103  5.211479    5.213689
  5.2141794   5.21428849  5.21431278  5.21431819  5.2143194
  5.21431967  5.21431973  5.21431974  5.21431974  5.21431974
  5.21431974  5.21431974  5.21431974  5.21431974  5.21431974]

```

Vi ser at værdierne $\lambda^{(k)} = \text{lambda_out}[k]$ ser ud til at stabilisere. Hvis vi beregne $Aw^{(k)}$ og sammenligne med $\lambda^{(k)}w^{(k)}$ ser vi at $w^{(k)}$ er tæt på at være en egenvektor for A :

```
print(a @ w)
print(lambda_out[-1] * w)
print(np.allclose(a @ w, lambda_out[-1] * w,
                  atol = np.finfo(float).eps))
```

```
[[2.07067192]
 [2.71487435]
 [3.94092698]]
[[2.07067204]
 [2.71487474]
 [3.94092664]]
```

True

△

Vi analyserer potensmetoden for A en symmetrisk matrix. Så er A diagonaliserbar via en ortogonal matrix, pga. sætning 23.9. Søjlerne af den ortogonale matrix giver en ortonormal basis v_0, \dots, v_{n-1} af egenvektorer for A . Skriv $\lambda_0, \dots, \lambda_{n-1}$ for de tilhørende egenverdier, så $Av_i = \lambda_i v_i$ for hvert i . Da vi har en basis, kan vi skrive en vilkårlig $w \in \mathbb{R}^n$ som en lineær kombination af v_0, \dots, v_{n-1} :

$$w = s_0 v_0 + s_1 v_1 + \dots + s_{n-1} v_{n-1}.$$

Da $Av_i = \lambda_i v_i$, kan vi nu beregne Aw som

$$\begin{aligned} Aw &= s_0 Av_0 + s_1 Av_1 + \dots + s_{n-1} Av_{n-1} \\ &= s_0 \lambda_0 v_0 + s_1 \lambda_1 v_1 + \dots + s_{n-1} \lambda_{n-1} v_{n-1}. \end{aligned}$$

I potensmetoden er $w^{(k)}$ en enhedsvektor i retningen $A^k w$. Vi ser at

$$\begin{aligned} A^k w &= s_0 \lambda_0^k v_0 + s_1 \lambda_1^k v_1 + \dots + s_{n-1} \lambda_{n-1}^k v_{n-1} \\ &= \lambda_0^k \left(s_0 v_0 + s_1 \left(\frac{\lambda_1}{\lambda_0} \right)^k v_1 + \dots + s_{n-1} \left(\frac{\lambda_{n-1}}{\lambda_0} \right)^k v_{n-1} \right), \end{aligned} \quad (24.1)$$

hvis $\lambda_0 \neq 0$.

Antag nu at $|\lambda_0| > |\lambda_1| \geq \dots \geq |\lambda_{n-1}| \geq 0$ og at $s_0 \neq 0$. For hvert $i > 0$ er $|\lambda_i/\lambda_0| < 1$, så faktoren $(\lambda_i/\lambda_0)^k$ konvergerer mod 0 når $k \rightarrow \infty$. Dette betyder at for stort k domineres $A^k w$ af leDET

$$\lambda_0^k s_0 v_0,$$

som er parallel med v_0 . Da $w^{(k)}$ og v_0 er enhedsvektorer, har vi dermed at for alle store k ligger $w^{(k)}$ tæt på $\varepsilon_k v_0$, hvor $\varepsilon_k \in \{\pm 1\}$.

Proposition 24.2. *Antag at $A \in \mathbb{R}^{n \times n}$ symmetrisk, med egenverdier der opfylder $|\lambda_0| > |\lambda_1| \geq \dots \geq |\lambda_{n-1}| \geq 0$. Lad v_0 være en enhedsvektor, som er en egenvektor for A med egenverdi λ_0 , og lad $w \in \mathbb{R}^n$ opfylder $\langle w, v_0 \rangle \neq 0$. Lad $w^{(0)} = w$, og lad $w^{(k)}, \lambda^{(k)}$ være frembragt af potensmetoden.*

Så findes der $\varepsilon_k \in \{\pm 1\}$ og konstanter $K_0, K_1 \geq 0$ således at for alle k tilstrækkelig stor gælder

$$\|w^{(k)} - \varepsilon_k v_0\|_2 \leq K_0 \left| \frac{\lambda_1}{\lambda_0} \right|^k \quad \text{og} \quad |\lambda^{(k)} - \lambda_0| \leq K_1 \left| \frac{\lambda_1}{\lambda_0} \right|^{2k}.$$

□

Dette siger at efter et vis trin, forbedres tilnærmelsen til λ_0 med en faktor der er højst $|\lambda_1/\lambda_0|^2$ for hver iteration af potensmetoden. Med andre ord, har vi kvadratisk konvergens. Vi skriver

$$|\lambda^{(k)} - \lambda_0| \leq O\left(\left|\frac{\lambda_1}{\lambda_0}\right|^{2k}\right) \quad \text{for } k \rightarrow \infty.$$

Notationen $O(\cdot)$ kaldes *stor-»O«-notationen*. Det følger at potensmetoden virker godt hvis $|\lambda_1|$ er væsentlig mindre end $|\lambda_0|$.

Bevis. Vores antagelser sikrer at $\lambda_0^k s_0$, for $s_0 = \langle w, v_0 \rangle$, er forskellig fra 0. Lad ε_k være fortegnet af $\lambda_0^k s_0$. Vi har $w^{(k)} = A^k w / \|A^k w\|_2$. Men fra (24.1) har vi

$$\langle w^{(k)}, \varepsilon_k v_0 \rangle = \frac{\langle A^k w, \varepsilon_k v_0 \rangle}{\|A^k w\|_2} = \frac{\langle \lambda_0^k s_0 v_0, \varepsilon_k v_0 \rangle}{\|A^k w\|_2} = \frac{\varepsilon_k \lambda_0^k s_0}{\|A^k w\|_2} > 0,$$

da $\langle v_i, v_0 \rangle = 0$, for $i > 0$. Ved at bruge $\langle w^{(k)}, \varepsilon_k v_0 \rangle \geq 0$, følger det at

$$\begin{aligned} \|w^{(k)} - \varepsilon_k v_0\|_2^2 &= \|w^{(k)}\|_2^2 + \|v_0\|_2^2 - 2\langle w^{(k)}, \varepsilon_k v_0 \rangle = 2 - 2\langle w^{(k)}, \varepsilon_k v_0 \rangle \\ &\leq (2 - 2\langle w^{(k)}, \varepsilon_k v_0 \rangle)(1 + \langle w^{(k)}, \varepsilon_k v_0 \rangle) = 2(1 - \langle w^{(k)}, \varepsilon_k v_0 \rangle^2). \end{aligned}$$

Men

$$1 - \langle w^{(k)}, \varepsilon_k v_0 \rangle^2 = 1 - \frac{\langle A^k w, \varepsilon_k v_0 \rangle^2}{\|A^k w\|_2^2} = \frac{\|A^k w\|_2^2 - \langle A^k w, \varepsilon_k v_0 \rangle^2}{\|A^k w\|_2^2}$$

Da v_i er ortonormal, er

$$\|A^k w\|_2^2 = s_0^2 \lambda_0^{2k} + s_1^2 \lambda_1^{2k} + \dots + s_{n-1}^2 \lambda_{n-1}^{2k}$$

og $\langle A^k w, \varepsilon_k v_0 \rangle^2 = s_0^2 \lambda_0^{2k}$, som er netop den første led i $\|A^k w\|_2^2$. Det følger at

$$\begin{aligned} 1 - \langle w^{(k)}, \varepsilon_k v_0 \rangle^2 &= \frac{s_1^2 \lambda_1^{2k} + \dots + s_{n-1}^2 \lambda_{n-1}^{2k}}{s_0^2 \lambda_0^{2k} + s_1^2 \lambda_1^{2k} + \dots + s_{n-1}^2 \lambda_{n-1}^{2k}} \\ &\leq \frac{s_1^2 \lambda_1^{2k} + \dots + s_{n-1}^2 \lambda_{n-1}^{2k}}{s_0^2 \lambda_0^{2k}} \\ &\leq \frac{(s_1^2 + \dots + s_{n-1}^2) \lambda_1^{2k}}{s_0^2 \lambda_0^{2k}} = \frac{1 - s_0^2}{s_0^2} \left(\frac{\lambda_1}{\lambda_0} \right)^{2k}, \end{aligned}$$

og den første ulighed i proposition holder med

$$K_0 = \frac{\sqrt{2(1 - s_0^2)}}{|s_0|}.$$

Tilsvarende har man

$$\begin{aligned} \lambda^{(k)} - \lambda_0 &= (w^{(k)})^T A w^{(k)} - \lambda_0 = \frac{(A^k w)^T A^{k+1} w}{\|A^k w\|_2^2} - \lambda_0 \\ &= \frac{w^T A^{2k+1} w}{w^T A^{2k} w} - \lambda_0 \\ &= \frac{s_0^2 \lambda_0^{2k+1} + s_1^2 \lambda_1^{2k+1} + \dots + s_{n-1}^2 \lambda_{n-1}^{2k+1}}{s_0^2 \lambda_0^{2k} + s_1^2 \lambda_1^{2k} + \dots + s_{n-1}^2 \lambda_{n-1}^{2k}} - \lambda_0 \\ &= \frac{s_1^2 \lambda_1^{2k} (\lambda_1 - \lambda_0) + \dots + s_{n-1}^2 \lambda_{n-1}^{2k} (\lambda_{n-1} - \lambda_0)}{s_0^2 \lambda_0^{2k} + s_1^2 \lambda_1^{2k} + \dots + s_{n-1}^2 \lambda_{n-1}^{2k}} \end{aligned}$$

så

$$\begin{aligned}
 |\lambda^{(k)} - \lambda_0| &\leq \frac{|s_1^2 \lambda_1^{2k} (\lambda_1 - \lambda_0) + \cdots + s_{n-1}^2 \lambda_{n-1}^{2k} (\lambda_{n-1} - \lambda_0)|}{s_0^2 \lambda_0^2} \\
 &\leq \frac{(s_1^2 + \cdots + s_{n-1}^2) \lambda_1^{2k} \max_{i=1, \dots, n-1} |\lambda_i - \lambda_0|}{s_0^2 \lambda_0^2} \\
 &\leq \left(\frac{1 - s_0^2}{s_0^2} \max_{i=1, \dots, n-1} |\lambda_i - \lambda_0| \right) \left(\frac{\lambda_1}{\lambda_0} \right)^{2k},
 \end{aligned}$$

som er den anden ønskede ulighed. \square

24.2 Page rank

En anvendelse af potensmetoden er til rangering af web sider via page rank, som bruges af Google i forbindelse med prioritering af resultater fra internet søgninger.

Lad os antage at vi vil gerne bestemme rangering af n websider. En webside j har links til $k(j)$ andre forskellige sider. Så indfører vi en $(n \times n)$ -matrix $M = (m_{ij})$ med

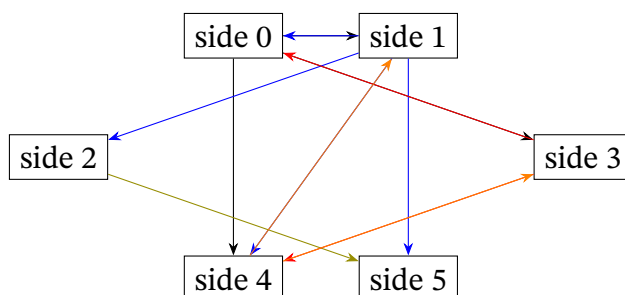
$$m_{ij} = \begin{cases} \frac{1}{k(j)}, & \text{hvis der er en link fra side } j \text{ til side } i, \\ 0, & \text{hvis der er ingen link } j \text{ til } i \text{ og } k(j) \neq 0, \\ \frac{1}{n}, & \text{hvis } j \text{ har ingen links til andre sider.} \end{cases}$$

Vi opstiller en model for hvilket sider en bruger, der læser på nettet, besøger. Brugeren starter på en af siderne. Brugeren går videre til en ny side ved enten (a) at følge en link på den nuværende side eller (b) at springe direkte til en tilfældig side.

Vi antager at tilfældet (a) sker med sandsynlighed $0 < p < 1$, og at tilfældet (b) sker med sandsynlighed $1 - p$. Ofte tages $p = 0,85$.

Sandsynligheden for at brugeren går fra side j direkte til side i er nu

$$a_{ij} = pm_{ij} + (1 - p) \frac{1}{n} \quad (24.2)$$



Figur 24.1: En samling websider med links.

Bemærk at valget af værdien $m_{ij} = 1/n$, når side j har ingen links til andre sider, giver $a_{ij} = p(1/n) + (1-p)1/n = 1/n$ i sådan en situation. Dette repræsenterer korrekt at brugeren er tvunget til at vælge en tilfældig side, når der er ingen links at følge.

Eksempel 24.3. For samlingen af 6 websider givet i figur 24.1 har vi

$$M = \begin{bmatrix} 0 & 1/4 & 0 & 1/2 & 0 & 1/6 \\ 1/3 & 0 & 0 & 0 & 1/2 & 1/6 \\ 0 & 1/4 & 0 & 0 & 0 & 1/6 \\ 1/3 & 0 & 0 & 0 & 1/2 & 1/6 \\ 1/3 & 1/4 & 0 & 1/2 & 0 & 1/6 \\ 0 & 1/4 & 1 & 0 & 0 & 1/6 \end{bmatrix}.$$

△

Som eksemplet viser, har hver søjle af M sum 1. Matricer med dette egenskab kaldes *stokastiske matricer*. Formlen (24.2) for $A = (a_{ij})$, giver at A er også en stokastisk matrix. Bemærk desuden at A har at alle a_{ij} opfylder $a_{ij} \geq (1-p)/n > 0$.

Proposition 24.4. Hvis $B \in \mathbb{R}^{n \times n}$ er en stokastisk matrix så er $\lambda = 1$ en egen værdi.

Bevis. I matricen $B - I_n$ er hvert søljesum lige med $1 - 1 = 0$. Det følger at summen af alle rækker i $B - I_n$ er rækkevektoren 0, så echelonformen for $B - I_n$ indeholder en nulrække. Det følger at $(B - I_n)x = 0$ har en løsning med $x \neq 0$, og dermed $Bx = I_n x = x$. Så x er egenvektor med egen værdi 1. □

Et dybere resultat er

Sætning 24.5 (Perron-Frobenius). Hvis A er en stokastisk matrix med alle indgange > 0 , så har egenværdien $\lambda = 1$ en egenvektor v med positive indgange og alle andre egenværdier λ_i for A har $|\lambda_i| < 1 = \lambda$. \square

Page rank er den positive egenvektor v med $\|v\|_2 = 1$ og egenværdi 1 for matricen A af (24.2). Sætningen sikre at vi kan bruge potensmetoden til at bestemme den.

Eksempel 24.6. Vi fortsætter eksempel 24.3. Vi opstiller matricen m og definere a via (24.2).

```
import numpy as np

m = np.array([[ 0., 1/4., 0., 1/2., 0., 1/6.],
               [1/3., 0., 0., 0., 1/2., 1/6.],
               [ 0., 1/4., 0., 0., 0., 1/6.],
               [1/3., 0., 0., 0., 1/2., 1/6.],
               [1/3., 1/4., 0., 1/2., 0., 1/6.],
               [ 0., 1/4., 1.0, 0., 0., 1/6.]])

p = 0.85
n = m.shape[0]
a = p * m + ((1-p)/n) * np.ones_like(m)

print(np.array_str(a, precision=4))
```

```
[[0.025  0.2375 0.025  0.45   0.025  0.1667]
 [0.3083 0.025  0.025  0.025  0.45   0.1667]
 [0.025  0.2375 0.025  0.025  0.025  0.1667]
 [0.3083 0.025  0.025  0.025  0.45   0.1667]
 [0.3083 0.2375 0.025  0.45   0.025  0.1667]
 [0.025  0.2375 0.875  0.025  0.025  0.1667]]
```

Her har vi brugt `np.array_str(a, precision=4)` for at gøre fremvisningen af matricen a mere overskueligt.

Nu bruger vi potensmetoden, skrevet på en måde, der kun kræver en matrix-vektormultiplikation per løkke. Bemærk at da vi ønsker en egenvektor med positive indgange og alle indgange i a er positive, kan vi begynde med en vektor

v med positive indgange. Vi vælger at stoppe når de sidste to kandidater for egenvektorer er næste parallelle med hinanden

Koden nedenfor bruger en **while** løkke. Gentagelse af løkken kontrolleres af en test, som vi sætter til altid at give sandt, **True**. Løkken afbrydes i stedet med **break** kommandoen, når indre produktet mellem v_{ny} og v er tilstrækkelig tæt på 1.0.

```
rng = np.random.default_rng()
v = rng.random((n, 1))
v /= np.linalg.norm(v)

nøjagtighed = 1e-9

while True:
    v_ny = a @ v
    v_ny /= np.linalg.norm(v_ny)
    if np.vdot(v_ny, v) > 1.0 - nøjagtighed:
        print(v_ny)
        break
    else:
        v = v_ny

print('lambda = ', (v_ny.T @ (a @ v_ny))[0, 0])

print(v_ny.argmax())
```

```
[[0.39646895]
 [0.44276155]
 [0.20828893]
 [0.44276155]
 [0.50879674]
 [0.38532894]]
lambda = 1.0000008881273814
4
```

Vi ser at den beregnede $\lambda = v_{ny}.T @ (a @ v_{ny})$ er tæt på 1, og side 4 har fået den største page rank, tæt efterfulgt af sider 1 og 3. Vi kan få rangeringen af sider bestemt af v_{ny} fra `np.argsort()`, som vil give indekserne for ordningen

af indgangerne i `v_ny` i voksende rækkefølge. Vi er mere interesseret i den omvendte rækkefølge, så vi bruger `np.flip()` bagefter.

```
print(np.flip(np.argsort(v_ny, axis=0)))
```

```
[[4]
 [3]
 [1]
 [0]
 [5]
 [2]]
```

△

24.3 Rayleighs kvotient

I potensmetoden brugt vi udtrykket $(w^{(k)})^T A w^{(k)}$ som vores tilnærmelse til egenværdien λ_0 . Generelt hvis $v \neq 0$, er *Rayleighs kvotient* tallet

$$r(v) = \frac{v^T A v}{v^T v}.$$

Når v er en egenvektor for A med $Av = \lambda v$, er

$$r(v) = \frac{v^T A v}{v^T v} = \frac{v^T \lambda v}{v^T v} = \lambda$$

den tilhørende egenværdi.

Hvis w er tæt på en egenvektor v , er $r(w)$ en god tilnærmelse til egenværdien λ : Skalaen $r(x)$ er den mindste kvadraters løsning på systemet $xr = Ax$, da systemet har normalligninger $x^T x r = x^T A x$.

For A symmetrisk har vi

$$\begin{aligned} \frac{\partial r(x)}{\partial x_j} &= \frac{\partial}{\partial x_j} \frac{\sum_{k,\ell=0}^{n-1} x_k a_{k\ell} x_\ell}{\sum_{i=0}^{n-1} x_i^2} \\ &= \frac{\sum_{\ell=0}^{n-1} a_{j\ell} x_\ell + \sum_{k=0}^{n-1} x_k a_{kj}}{x^T x} - \frac{x^T A x \cdot 2x_j}{(x^T x)^2} \\ &= \frac{1}{x^T x} (2(Ax)_j - r(x) 2x_j) = \frac{2}{x^T x} (Ax - r(x)x)_j. \end{aligned}$$

Det følger at x er et kritisk punkt for $r(x)$ hvis og kun hvis x er en egenvektor. Vi får også, at hvis v er egenvektor for A , så gælder

$$r(x) - r(v) = O(\|x - v\|_2^2) \quad \text{for } x \rightarrow v.$$

Dette siger at $r(x)$ giver en estimering af λ , som er korrekt til anden orden i $\|x - v\|_2$.

24.4 Inverspotensmetoden

En stor ulempe med potensmetoden er at den kun finde den egen værdi, der er numerisk størst. Har man interesse i andre egen værdier, eller hvis man har et godt gæt for den største, kan man bruge inverspotensmetoden, som bruger et gæt μ for den ønskede egen værdi.

INVERSPOTENSMETODEN(A, μ)

- 1 Begynd med et vilkårligt $w^{(0)}$ af længde 1
- 2 **for** $k \in \{1, 2, \dots\}$:
- 3 Løs $(A - \mu I_n)v = w^{(k-1)}$ for $v \neq 0$
- 4 $w^{(k)} = v / \|v\|_2$
- 5 $\lambda^{(k)} = (w^{(k)})^T A w^{(k)}$

Under visse antagelser konvergerer $\lambda^{(k)}$ til egen værdien af A , som er numerisk tættest på μ . Desuden vil $w^{(k)}$ konvergerer mod en egenvektor for denne egen værdi.

For at gøre disse udsagn mere præcis, antager vi at $A \in \mathbb{R}^{n \times n}$ er symmetrisk og at $\mu \in \mathbb{R}$ er givet. Lad λ_p være egen værdien af A , som minimerer $|\mu - \lambda_p|$, og lad λ_q være egen værdien af A med $|\mu - \lambda_q|$ næst mindst. Vi antager så at egen værdierne $\lambda_0, \lambda_1, \dots, \lambda_{n-1}$ opfylder

$$|\mu - \lambda_p| < |\mu - \lambda_q| \leq |\mu - \lambda_i| \quad \text{for alle } i \neq p.$$

I tilfældet hvor $\mu \neq \lambda_i$ for alle i , er $A - \mu I_n$ både invertibel og diagonaliserbar, med egen værdier $\lambda_i - \mu$. Den inverse $(A - \mu I_n)$ har så egen værdier $1/(\lambda_i - \mu)$, og den største af disse numerisk er $1/|\lambda_p - \mu|$. Den inversepotensmetoden er stort set potensmetoden for $(A - \mu I_n)^{-1}$.

Lad v_p være en egenvektor med egen værdi λ_p og $\|v_p\|_2 = 1$, og antag at begyndelsesvektoren $w^{(0)}$ i inverspotensmetoden opfylder $s_p = \langle w, v_p \rangle \neq 0$. Så har vi:

Proposition 24.7. For $w^{(k)}$ og $\lambda^{(k)}$ produceret af den inverspotensmetoden, findes der $\varepsilon_k \in \{\pm 1\}$ således at

$$\|w^{(k)} - \varepsilon_k v_p\|_2 = O\left(\left|\frac{\mu - \lambda_q}{\mu - \lambda_p}\right|^k\right) \quad \text{og} \quad |\lambda^{(k)} - \lambda_0| = O\left(\left|\frac{\mu - \lambda_q}{\mu - \lambda_p}\right|^{2k}\right).$$

□

Vi får igen kvadratisk konvergens for egenværdien, og denne konvergens er hurtig hvis μ ligger væsentlige tættere på λ_p end på λ_q . Dette er en foretrukken metoden for beregning af konkrete egenværdier.

Eksempel 24.8. Lad os kikke igen på eksempel 24.1.

```
import numpy as np

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

Denne gang bruger vi inverspotensmetoden. Først lad os kikke efter en egen-værdi, som er tættest på 1,0

```
mu = 1

rng = np.random.default_rng()

w = rng.standard_normal((a.shape[0], 1))
w /= np.linalg.norm(w)

n = 20
lambda_out = np.empty(n)

for i in range(n):
    v = np.linalg.solve(a - mu * np.eye(a.shape[0]), w)
    w = v / np.linalg.norm(v)
    lambda_out[i] = w.T @ (a @ w)

np.set_printoptions(linewidth = 60)
print(lambda_out)
```

24.4 INVERSPOTENSMETODEN

```
[2.82497299 2.33421508 1.5719425  1.34014703 1.32563388
 1.32490697 1.324871  1.32486922 1.32486913 1.32486913
 1.32486913 1.32486913 1.32486913 1.32486913 1.32486913
 1.32486913 1.32486913 1.32486913 1.32486913 1.32486913]
```

Vi ser at $\lambda^{(k)}$ stabiliserer relativt hurtigt, og vi kan nemt tjekke hvor godt et par λ, w vi har fået

```
print(np.allclose(a @ w, lambda_out[-1] * w,
                  atol = np.finfo(float).eps))
```

True

Fra eksempel 24.1 ved vi at den største egen værdi ligger tæt på 5,2, dette kan sættes ind i inverspotensmetoden, som værdien for μ

```
mu = 5.2

w = rng.standard_normal((a.shape[0], 1))
w /= np.linalg.norm(w)

n = 20
lambda_out = np.empty(n)

for i in range(n):
    v = np.linalg.solve(a - mu * np.eye(a.shape[0]), w)
    w = v / np.linalg.norm(v)
    lambda_out[i] = w.T @ (a @ w)

print(lambda_out)
```

```
[5.21369195 5.21431973 5.21431974 5.21431974 5.21431974
 5.21431974 5.21431974 5.21431974 5.21431974 5.21431974
 5.21431974 5.21431974 5.21431974 5.21431974 5.21431974
 5.21431974 5.21431974 5.21431974 5.21431974 5.21431974]
```

som giver hurtige konvergens end potensmetoden.

△

Hvis vi udnytter at vi ændrer på valget af μ undervejs, kan vi få endnu bedre konvergens. Forfines af inverspotensmetoden på denne måde, får vi Rayleighkvotientmetoden:

RAYLEIGHKVOTIENTMETODEN(A, w)

- 1 $w^{(0)} = w$ en endhedsvektor
- 2 $\lambda^{(0)} = w^T A w$
- 3 **for** $k \in \{1, 2, \dots\}$:
- 4 Løs $(A - \lambda^{(k-1)} I_n) v = w^{(k-1)}$ for $v \neq 0$
- 5 $w^{(k)} = v / \|v\|_2$
- 6 $\lambda^{(k)} = (w^{(k)})^T A w^{(k)}$

som har tredjeordens konvergens med en faktor der forbedres for hvert skridt:

Proposition 24.9. *For A symmetrisk, konvergerer Rayleighkvotientmetoden til en egenværdi λ_p af A og for stort k gælder*

$$\|w^{(k+1)} - \varepsilon_{k+1} v_p\|_2 = O(\|w^{(k)} - \varepsilon_k v_p\|_2^3) \quad \text{og} \quad |\lambda^{(k+1)} - \lambda_p| = O(|\lambda^{(k)} - \lambda_p|^3),$$

med $\varepsilon_k \in \{\pm 1\}$. □

Eksempel 24.10. For matricen i (24.1), ser vi konvergens af både egenværdien og egenvektoren indenfor machine epsilon ofte efter kun 3 iterationer:

```
for j in range(5):
    w = rng.standard_normal((a.shape[0], 1))
    w /= np.linalg.norm(w)

    for i in range(20):
        lambda_est = (w.T @ (a @ w))[0,0]
        print(lambda_est)
        if np.allclose(a @ w, lambda_est * w,
                        atol = np.finfo(float).eps):
            print(f'    efter {i} iterationer\n')
            break
        b = a - lambda_est * np.eye(a.shape[0])
        v = np.linalg.solve(b, w)
        w = v / np.linalg.norm(v)
```

24.4 INVERSPOTENSMETODEN

2.3132036613119693
2.4088854324552305
2.46067490160736
2.460811127187129
 efter 3 iterationer

2.4382363526628175
2.460176976136701
2.4608111269611066
2.46081112718911
 efter 3 iterationer

2.922682980131502
2.4781139187652723
2.4608070973307674
2.460811127189111
 efter 3 iterationer

2.421189144421365
2.4598130946040735
2.460811126312255
2.4608111271891104
 efter 3 iterationer

4.071162028649015
4.508267813452456
5.106954957279769
5.214136457097686
5.214319743376724
 efter 4 iterationer

Bemærk at forskellige egenverdier kan beregnes på denne måde. △

Potensmetoden på $A \in \mathbb{R}^{n \times n}$ bruger $O(n^2)$ flops per skridt til at estimere den numerisk størst egenverdi. For inverspotensmetoden, løser vi et lineært ligningssystem med den samme koefficientmatrix $A - \mu I_n$ hver gang, og dette koster $O(n^3)$ flops. Hvis vi først beregne en QR -dekomponering af $A - \mu I_n$ i begyndelsen, kan denne bruges til at reducere arbejdet per skridt fra $O(n^3)$ flops

til $O(n^2)$ flops. Til gengæld, kan sådan en forbedring ikke bruges til Rayleighkvotientmetoden, som bruger $O(n^3)$ flops per skridt. Men dette opvejes af at Rayleighkvotientmetoden bruger meget få skridt til at få meget høj nøjagtighed.

Kapitel 25

Hessenberg- og tridiagonalform

Vi begynder nu arbejdet for at beskrive metoder for at bestemme alle egenverdier og en basis af tilhørende egenvektorer 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

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

Hessenbergform er lidt svagere end øvre triangulær, 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

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

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 vælge et Householder matrix $H_0 = H_0^T = H_0^{-1}$ således at H_0A har kun nultal under diagonalen i den første søjle

$$A = \begin{bmatrix} a_{00} & * & \dots & * \\ a_{10} & * & \dots & * \\ \vdots & \vdots & \ddots & \vdots \\ a_{n-1,0} & * & \dots & * \end{bmatrix} \mapsto H_0A = \begin{bmatrix} \pm \|a_0\|_2 & * & \dots & * \\ \mathbf{0} & * & \dots & * \\ \vdots & \vdots & \ddots & \vdots \\ \mathbf{0} & * & \dots & * \end{bmatrix}$$

men disse nultal bliver overskrevet når vi beregner

$$H_0AH_0^{-1} = H_0AH_0 = \begin{bmatrix} * & * & \dots & * \\ * & * & \dots & * \\ \vdots & \vdots & \ddots & \vdots \\ * & * & \dots & * \end{bmatrix},$$

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)
```

```
[[ 3.15423995 -0.7885905  2.65861868 -2.25771234]
 [-0.7885905  3.63459687  0.19818514  1.86885935]
 [ 2.65861868  0.19818514 -4.76429025  1.44709128]
 [-2.25771234  1.86885935  1.44709128  2.32170292]]
```

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(np.array_str(h0 @ a, precision=6))
```

```
[[ -4.768294e+00  1.897127e+00  1.615654e+00  2.095006e+00]
 [-1.387779e-16  3.367267e+00  3.019994e-01  1.435600e+00]
 [ 4.996004e-16  1.099450e+00 -5.114285e+00  2.907763e+00]
 [-2.220446e-16  1.103501e+00  1.744309e+00  1.081293e+00]]
```

Vi ser at denne give tal tæt på 0 i den 0'te søjle underdiagonalen. Men $H_0 A H_0^T = H_0 A H_0$ har

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

```
[[ 3.55911648  1.06823645  4.41013989 -0.2780852 ]
 [ 1.06823645  3.26093701  0.66047481  1.13118078]
 [ 4.41013989  0.66047481 -3.63434425  1.65098965]
 [-0.2780852   1.13118078  1.65098965  1.16054025]]
```

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

$$\begin{bmatrix} a_{00} & a_{01} & a_{02} & \dots & a_{0,n-1} \\ a_{10} & * & * & \dots & * \\ a_{20} & * & * & \dots & * \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ a_{n-1,0} & * & * & \dots & * \end{bmatrix} \mapsto H_0 A = \begin{bmatrix} a_{00} & a_{01} & a_{02} & \dots & a_{0,n-1} \\ * & * & * & \dots & * \\ 0 & * & * & \dots & * \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & * & * & \dots & * \end{bmatrix}.$$

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 $H_0 A H_0$ ved at gange

25.1 REDUKTION TIL HESSENBERGFORM

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** $k \in \{0, 1, \dots, n-3\}$:
- 3 Beregn Householder H_k for $B_{[k+1:,k]}$
- 4 $B = H_k B H_k$
- 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)
```

25 HESSENBERG- OG TRIDIAGONALFORM

```
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:')
print(q @ h @ q.T)

print('Hessenbergmatrix')
print(np.array_str(h, precision=3))
```

```
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
[[ 0.000e+00 -5.477e+00  1.077e-15 -7.826e-16 -5.128e-17]
 [-2.739e+01  6.000e+01  2.236e+01 -1.376e-14  7.938e-15]
 [ 0.000e+00  4.472e+00 -4.736e-16  1.763e-15  3.496e-15]
 [ 0.000e+00  0.000e+00 -2.448e-17 -9.831e-16 -2.449e-15]
 [ 0.000e+00  0.000e+00  0.000e+00 -8.523e-17 -2.086e-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

$$\begin{bmatrix} a_{00} & a_{01} & a_{02} & \dots & a_{0,n-1} \\ a_{10} & * & * & \dots & * \\ a_{20} & * & * & \dots & * \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ a_{n-1,0} & * & * & \dots & * \end{bmatrix} \mapsto H_0 A = \begin{bmatrix} a_{00} & a_{01} & a_{02} & \dots & a_{0,n-1} \\ * & * & * & \dots & * \\ 0 & * & * & \dots & * \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & * & * & \dots & * \end{bmatrix}$$

Denne gang når vi beregne $H_0 A H_0 = H_0 A H_0^T$, med A symmetrisk, får vi igen en symmetrisk matrix. Så

$$H_0 A H_0 = \begin{bmatrix} a_{00} & * & 0 & \dots & 0 \\ * & * & * & \dots & * \\ 0 & * & * & \dots & * \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & * & * & \dots & * \end{bmatrix}$$

og dette begynder at ligne den tridiagonalform vi ønsker.

Vi fortsætter ved at vælge en Householdermatrix H_1 så at

$$\begin{bmatrix} a_{00} & * & 0 & 0 & 0 \\ * & * & * & * & * \\ 0 & * & * & * & * \\ 0 & * & * & * & * \\ 0 & * & * & * & * \end{bmatrix} \mapsto H_1 H_0 A H_0 = \begin{bmatrix} a_{00} & * & 0 & 0 & 0 \\ * & * & * & * & * \\ 0 & * & * & * & * \\ 0 & 0 & * & * & * \\ 0 & 0 & * & * & * \end{bmatrix}$$

og får dermed

$$H_1 H_0 A H_0 H_1 = \begin{bmatrix} a_{00} & * & 0 & 0 & 0 \\ * & * & * & 0 & 0 \\ 0 & * & * & * & * \\ 0 & 0 & * & * & * \\ 0 & 0 & * & * & * \end{bmatrix},$$

osv.

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

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

hvor

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

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))

```

25.2 REDUKTION TIL TRIDIAGONALFORM

```
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.

Kapitel 26

Egendekomponering via QR-metoden

Vi vil nu indføre QR-metoden for beregningen af samtlige egenverdier 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 egenverdier. Ideen er meget enkelt, men det er ikke oplagt hvorfor det skulle virke.

FØRSTE QR-METODE(A)

```
1  $A^{(0)} = A$ 
2 for  $k \in \{1, 2, \dots\}$ :
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

$$RQ = Q^T Q R Q = Q^T A Q,$$

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

Vi kan dog kikke på et eksempel. Betragt matrixen

```
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

26.1 QR-METODEN: FØRSTE VERSION

```
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 egenverdier 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 egenverdierne inden for cirka 3 decimaler.

26.2 Ortogonal iteration

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

Metoden begynder med en vilkårlig matrix

$$W = [w_0 \mid w_1 \mid \dots \mid w_{m-1}] \in \mathbb{R}^{n \times m}$$

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 nærme 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 ortogonal iteration nedenfor.)

Søjlerne af $Q^{[k]}$ fås ved at anvende en version af Gram-Schmidt processen på $A^k w_0, A^k w_1, \dots, A^k w_{m-1}$. 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, \dots, v_{n-1} være en ortonormal basis for A med egenverdier $\lambda_0, \lambda_1, \dots, \lambda_{n-1}$. Hvis

$$|\lambda_0| > |\lambda_1| > \dots > |\lambda_{m-1}| > |\lambda_m| \geq \dots \geq |\lambda_{n-1}| \quad (26.1)$$

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 $j = 0, 1, \dots, m-1$.

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** $k \in \{1, 2, \dots\}$:
- 3 $V^{(k)} = A W^{(k-1)}$
- 4 $Q^{(k)} R^{(k)} = V^{(k)}$
- 5 $W^{(k)} = Q^{(k)}$

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

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

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

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

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 egenverdier.

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)
```

```
[[-0.52387877  0.03506249  0.63404784]
 [-0.50228613 -0.58659547 -0.25764956]
 [-0.40070458  0.72311933  0.05138146]
 [-0.36536139 -0.28996113  0.19918437]
 [-0.42332796  0.21839722 -0.69948951]]
```

Så laver vi ortogonal iteration

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

print(w)
```

```
[[-0.29673807 -0.25043225 -0.08290442]
 [-0.48181475 -0.16389715  0.03906892]
 [-0.31761222 -0.36874371 -0.7798104 ]
 [-0.60619091 -0.20342845  0.58089918]
 [-0.45984358  0.85619294 -0.21459814]]
```

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

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

```
[7.362313  4.5812424  2.29207848]
```

Vi kan sammenligne Aq_i med $\lambda_i q_i$

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

```
[[ 0.00050783 -0.00310784 -0.01365435]
 [ 0.00032776  0.0027861  -0.00983023]
 [ 0.00078017 -0.03468466 -0.01498089]
 [ 0.00038021  0.02746393 -0.00704491]
 [-0.0017112  -0.00878829  0.03885317]]
```

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

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

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

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

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

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

da

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

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

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

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

Konvergens resultater for ortogonal iteration giver nu:

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

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

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

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

Igen er dette kun lineær konvergens.

26.4 QR-dekomponering af tridiagonale matricer

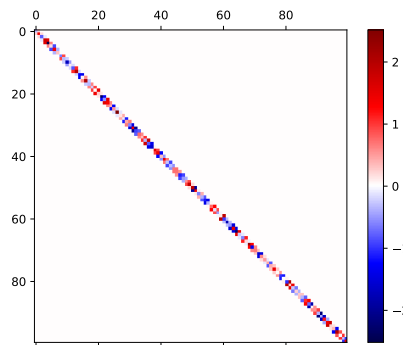
Som noteret i kapitel 25, er den første trin i en typisk beregning af egenverdier for en symmetrisk matrix, at vi reducerer til tridiagonal form. Det viser sig at QR-dekomponering af tridiagonale matricer har pæne egenskaber, som er relevant når vi skal anvende QR-metoden på dem.

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
```

26 EGENDEKOMPONERING VIA QR -METODEN



Figur 26.1: En tridiagonal matrix.

```
im = ax.matshow(a, cmap='seismic', clim = (-r, r))
fig.colorbar(im)
```

Lad os kikke på en tilfældig symmetrisk tridiagonal matrix t

```
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

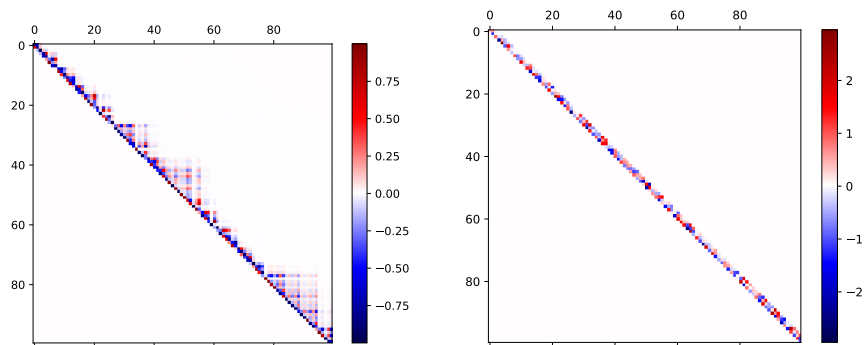
```
q, r = householder_qr(t)

heat_map(q)
```

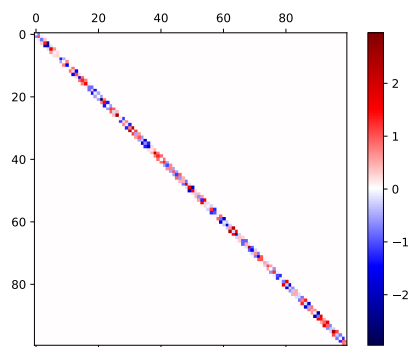
```
heat_map(r)
```

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

26.4 QR-DEKOMPONERING AF TRIDIAGONALE MATRICER



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



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

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

True

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

$$RQ = Q^T Q R Q = Q^T T Q$$

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 egen værdi 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  $k \in \{1, 2, \dots\}$ :
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 egen værdier som $T^{(k-1)}$ da

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

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)

```

```

[ 18.1917529   16.83348654  14.83812455 -17.49983151
 -13.146672    5.08877601  -9.53865578   1.59057962
 -3.96595715  -2.0645045 ]
iterationer: 20

```

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

```

numpy_eig = np.linalg.eigvals(a)
print(numpy_eig)

```

```

[ 18.24000563  16.85691943  14.76904771 -17.92945345
 -12.71987708  -9.68034981   5.23068829   1.59057962
 -3.96595715  -2.0645045 ]

```

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

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

```
[ 4.29621937e-01 -4.26794918e-01  1.41694036e-01
  2.04281037e-14 -4.44089210e-16 -1.08071863e-09
 -1.41912276e-01  6.90768469e-02 -2.34328893e-02
 -4.82527355e-02]
```

Egenverdierne, der beregnes bedst, er dem, der har relativt stort afstand fra nabo egenverdier. 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 egenverdier 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

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

hvor

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

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

```
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))

```

```

[ 18.24000563  16.85691943  14.76904771   5.23068829
 -17.92945345 -12.71987708  -9.68034981   1.59057962
 -3.96595715  -2.0645045 ]

```

26 EGENDEKOMPONERING VIA QR -METODEN

samlet iterationer: 28

afvigelser

```
[-1.06581410e-14  5.32907052e-15  8.88178420e-15  
-3.99680289e-15 -2.22044605e-15  6.43929354e-15  
 6.21724894e-15  6.39488462e-14  3.55271368e-15  
 6.03961325e-14]
```

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,
- egenverdier 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 a , da QR -metoden på en tridiagonal matrix kan omskrives til $O(n^2)$.

Kapitel 27

Singulærværdi beregning og principalkomponent analyse

Vi har brugt nogle kræfter på at vise hvordan egenverdier og egenvektorer for symmetriske matricer kan beregnes. Dette er et interessant problem i sig selv, men den er også relevant for beregning af singulærværdier, som vil vise i dette kapitel. Vil vi også skitsere hvordan singulærværdidekomponering giver en god måde at lave principalkomponent analyse, en ofte brugt værktøj i dataanalyse.

27.1 Singulærværdidekomponering og symmetriske matricer

Det er (mindst) tre måder man kan danne en ny symmetrisk matrix fra en vilkårlig matrix $A \in \mathbb{R}^{m \times n}$:

- (a) $A^T A \in \mathbb{R}^{n \times n}$,
- (b) $AA^T \in \mathbb{R}^{m \times m}$,
- (c) $\begin{bmatrix} 0_{n \times n} & A^T \\ A & 0_{m \times m} \end{bmatrix} \in \mathbb{R}^{(n+m) \times (n+m)}$.

Hvis man bruger QR-metoden til at beregne egenverdier for disse tre symmetriske matricer, vil det koster hhv. (a) $\sim 4n^3/3$ flops, (b) $\sim 4m^3/3$ flops og (c) $\sim 4(n+m)^3/3$ flops, Da $4(n+m)^3/3 > (4n^3/3) + (4m^3/3)$, virker den sidste metode udmiddelbart til at være ret bekostelig.

Eksempel 27.1. Lad os kikke på matricen

$$A = \begin{bmatrix} 1 & -1 \\ 2 & 1 \\ 4 & 1 \\ 3 & -1 \end{bmatrix}.$$

Vi har

$$(a) \quad A^T A = \begin{bmatrix} 30 & 2 \\ 2 & 4 \end{bmatrix},$$

$$(b) \quad AA^T = \begin{bmatrix} 2 & 1 & 3 & 4 \\ 1 & 5 & 9 & 5 \\ 3 & 9 & 17 & 11 \\ 4 & 5 & 11 & 10 \end{bmatrix} \text{ og}$$

$$(c) \quad \begin{bmatrix} 0_{n \times n} & A^T \\ A & 0_{m \times m} \end{bmatrix} = \begin{bmatrix} 0 & 0 & 1 & 2 & 4 & 3 \\ 0 & 0 & -1 & 1 & 1 & -1 \\ 1 & -1 & 0 & 0 & 0 & 0 \\ 2 & 1 & 0 & 0 & 0 & 0 \\ 4 & 1 & 0 & 0 & 0 & 0 \\ 3 & -1 & 0 & 0 & 0 & 0 \end{bmatrix}.$$

Δ

Hvad siger SVD om disse symmetriske matricer? Lad os skrive

$$A = U\Sigma V^T \in \mathbb{R}^{m \times n}$$

med $U \in \mathbb{R}^{m \times m}$ og $V \in \mathbb{R}^{n \times n}$ ortogonale, og $\Sigma = \text{diag}_{m \times n}(\sigma_0, \sigma_1, \dots, \sigma_{k-1})$, hvor $k = \min\{m, n\}$.

For (a) har vi

$$A^T A = V\Sigma^T U^T U\Sigma V^T = V\Sigma^T \Sigma V^T = VD_n V^T$$

hvor

$$D_n = \text{diag}(\sigma_0^2, \sigma_1^2, \dots, \sigma_{k-1}^2, 0, \dots, 0) \in \mathbb{R}^{n \times n}.$$

Tilsvarende for (b) er

$$AA^T = U\Sigma V^T V\Sigma^T U^T = U\Sigma \Sigma^T U^T = UD_m U^T$$

med $D_m \in \mathbb{R}^{m \times m}$, som ovenfor. Vi ser at i begge tilfælder giver egenværdierne af den symmetriske matrix singulærværdierne for A i anden potens. Da

27.1 SINGULÆRVÆRDIDEKOMPONERING OG SYMMETRISKE MATRICER

singulærværdier er positive, er de entydigt bestemt af disse egenverdier. Det betyder at vi kunne beregne singulærværdier af A ved at forme enten $A^T A$ eller AA^T og bruge metoden fra det forrige kapitel til at beregne egenverdier af den symmetriske matrix.

Der er dog flere problemer med at beregne singulærværdier af via egenverdierne af $A^T A$ (eller AA^T). De stammer fra tre kilder:

- (i) QR-metoden beregner egenverdier af $A^T A$ indenfor $\|A^T A\|_2 \epsilon_{\text{machine}} = \sigma_0^2 \epsilon_{\text{machine}}$,
- (ii) for $m \geq n$ har $A^T A$ har konditionstal $\kappa(A^T A) = \sigma_0^2 / \sigma_{n-1}^2 = \kappa(A)^2$,
- (iii) selve float beregning af $A^T A$ kan have betydningsfulde fejl.

Punkterne (i) og (ii) peger begge i retning af en halvering af det forventede antal korrekte cifre i hver beregnede singulærværdi. Punkt del (iii) kan også være ret alvorligt.

Eksempel 27.2. For (iii) betragt det følgende eksempel.

```
>>> import numpy as np
>>> s = 1e-9
>>> a = np.array([[1.0, 1.0],
...               [ s, 0.0],
...               [0.0, s]])
```

Vi har

```
>>> a.T @ a
array([[1., 1.],
       [1., 1.]])
```

og alle indgange i matricen $a.T @ a$ er præcis lige med 1.0 , som float:

```
>>> np.all(a.T @ a == np.ones((a.T @ a).shape))
True
```

Dette betyder at beregnede egenverdier af $a.T @ a$ er nødvendigvis 2.0 og 0.0 fra egenvektorerne hhv. $[1.0], [1.0]$ og $[1.0], [-1.0]$, som ville give singulærværdier $\text{np.sqrt}(2.0)$ og 0.0 . Men de korrekte singulærværdier for a er $\sqrt{2 + s^2}$ og s ; ingen af disse er 0 .

```
>>> np.linalg.svd(a, compute_uv=False)
array([1.41421356e+00, 1.00000000e-09])

>>> np.allclose(np.linalg.svd(a, compute_uv=False),
...             [np.sqrt(2.0 + s**2), s],
...             atol = np.finfo(float).eps)
True
```

Problemet stammer fra dannelsen af $a.T @ a$, hvor f.eks. den øverste venstre indgang er

```
>>> (1.0)**2 + (s)**2 + (0.0)**2
1.0
```

men $s**2$ er mindre end machine epsilon, så dens sum med 1.0 giver blot 1.0 .

For denne matrix kunne problemet delvis forbigås ved at betragte i stedet den større matrix $a @ a.T$

```
>>> a @ a.T
array([[2.e+00, 1.e-09, 1.e-09],
       [1.e-09, 1.e-18, 0.e+00],
       [1.e-09, 0.e+00, 1.e-18]])

>>> np.sort(np.sqrt(np.abs(np.linalg.eigvals(a @ a.T))))
array([1.38777878e-17, 1.00000000e-09, 1.41421356e+00])

>>> np.sort(np.linalg.svd(a, compute_uv=False))
array([1.00000000e-09, 1.41421356e+00])
```

Men generelt kan AA^T være væsentlige større end $A^T A$ og der kan være de samme underliggende problemer i forhold til float beregning. Δ

Indtil videre har vi kun kikket på de symmetriske matrice af type (a) og (b). For den tredje symmetrisk matrix (c), som er af størrelse $(n + m) \times (n + m)$, har vi

$$\begin{aligned} \begin{bmatrix} 0_{n \times n} & A^T \\ A & 0_{m \times m} \end{bmatrix} &= \begin{bmatrix} 0 & V \Sigma^T U^T \\ U \Sigma V^T & 0 \end{bmatrix} = \begin{bmatrix} V & 0 \\ 0 & U \end{bmatrix} \begin{bmatrix} 0 & \Sigma^T U^T \\ \Sigma V^T & 0 \end{bmatrix} \\ &= \begin{bmatrix} V & 0 \\ 0 & U \end{bmatrix} \begin{bmatrix} 0 & \Sigma^T \\ \Sigma & 0 \end{bmatrix} \begin{bmatrix} V^T & 0 \\ 0 & U^T \end{bmatrix}. \end{aligned}$$

27.1 SINGULÆRVÆRDIDEKOMPONERING OG SYMMETRISKE MATRICER

Her er de ydre matricer ortogonale med

$$\begin{bmatrix} V & 0 \\ 0 & U \end{bmatrix} \begin{bmatrix} V^T & 0 \\ 0 & U^T \end{bmatrix} = \begin{bmatrix} VV^T & 0 \\ 0 & UU^T \end{bmatrix} \begin{bmatrix} I_n & 0 \\ 0 & I_m \end{bmatrix} = I_{n+m}.$$

Desuden er matricen i midten symmetrisk, og vi kan beregne dens egenverdier. For hver singulærværdi σ har vi en delmatrix af formen

$$\begin{bmatrix} 0 & \sigma \\ \sigma & 0 \end{bmatrix} = \frac{1}{\sqrt{2}} \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix} \begin{bmatrix} \sigma & 0 \\ 0 & -\sigma \end{bmatrix} \frac{1}{\sqrt{2}} \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix}.$$

Vi ser så at matricen

$$\begin{bmatrix} 0 & \Sigma^T \\ \Sigma & 0 \end{bmatrix}$$

er diagonaliserbar med egenverdier

$$\sigma_0, -\sigma_0, \dots, \sigma_{k-1}, -\sigma_{k-1}, 0, \dots, 0.$$

Så den symmetriske matrice af type (c) har samme 2-norm og konditionstal, som A . Det betyder at vi kan forvente en præcise beregning af dens egenverdier, og dermed singulærværdierne for A , via QR-metoden. Selvfølgelig er ulempen at matricen er en $((n+m) \times (n+m))$ -matrix, som kan være meget stort.

I praksis beregnes SVD via QR-metoden for $A^T A$. Men man undlader at danne matricen $A^T A$ og i stedet for kikker på tilsvarende operationerne på den oprindelige matrix A . For eksempel, er det første trin at reducere A til en bidiagonal form

$$A = U_0 B V_0^T, \quad B = \begin{bmatrix} * & * & 0 & 0 & 0 \\ 0 & * & * & 0 & 0 \\ 0 & 0 & * & * & 0 \\ 0 & 0 & 0 & * & * \\ 0 & 0 & 0 & 0 & * \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \end{bmatrix}.$$

Dette svarer netop til at $B^T B$ er tridiagonal. Derefter findes oplysningen til » $Q^{(1)}$ « fra QR-metoden for $B^T B$ med shift $\mu^{(1)}$ givet ved en egenverdi af den sidste (2×2) -blok af $B^T B$. Dette anvendes på B , men ødelægger dens bidiagonal form. Så man gentager bidiagonaliseringstrinnet og forsætter. Til sidst vil metoden konvergere mod en SVD for A .

Person	0	1	2	3	4
Vægt (kg)	54	56	56	61	66
Højde (m)	1,54	1,52	1,63	1,72	1,82

Tabel 27.1: Vægt og højde for nogle personer.

27.2 Principalkomponent analyse

Relationerne mellem $(m \times n)$ -matricer og symmetriske matricer bruges i en del forskellige sammenhæng. Et eksempel er principalkomponent analyse.

Betragt datasættet givet i tabel 27.1. For en given række

$$x = [x_0 \ x_1 \ \dots \ x_{n-1}] \in \mathbb{R}^{1 \times n}$$

kan vi beregne dens *middelværdien*

$$\bar{x} = \frac{1}{n} \sum_{i=0}^{n-1} x_i \in \mathbb{R}$$

og dens *varians*

$$\text{var}(x) = \frac{1}{n} \sum_{i=0}^{n-1} (x_i - \bar{x})^2 \in \mathbb{R}.$$

Middelværdi er den gennemsnitlige værdi af x_0, x_1, \dots, x_{n-1} . Variansen måler hvor tæt disse værdier ligger til middelværdien. Er variansen 0, så er alle x_i lige med middelværdien; er variansen stort, så er der stor spredning i x_i værdierne.

Ved at sætte $1_n = [1 \ 1 \ \dots \ 1] \in \mathbb{R}^{1 \times n}$ kan variansen skrives

$$\text{var}(x) = \frac{1}{n} (x - \bar{x}1_n)(x - \bar{x}1_n)^T \in \mathbb{R}.$$

For to rækker kan vi også beregne deres indbyrdes *kovarians*

$$\text{cov}(x, y) = \frac{1}{n-1} (x - \bar{x}1_n)(y - \bar{y}1_n)^T.$$

Kovarians kan betragtes, som et mål for hvor meget y_i -værdier følger de tilsvarende x_i -værdier. Er $\text{cov}(x, y) = 0$, betragtes variablerne x og y , som værende uafhængig af hinanden.

Har man flere datarækker kan man udføre disse operationer i matrix form. Lad os antage at tabellen har m rækker og n søjler. F.eks. har vi n personer hvor vi har taget m forskellige mål. Så kan vi betragte vores data givet ved n søjlevektorer $v_0, \dots, v_{n-1} \in \mathbb{R}^m$. Hver vektor består af de forskellige målinger for en enkelt person.

Vektoren af middelværdier er så

$$\bar{v} = \frac{1}{n}(v_0 + \dots + v_{n-1}) \in \mathbb{R}^m.$$

For tabel 27.1 er $\bar{v} \in \mathbb{R}^2$, med den 0'te indgang middelvægten for gruppen og 1'te indgang middelhøjden.

Sæt

$$w_i = v_i - \bar{v}$$

og saml disse vektorer i en matrix

$$W = [w_0 \mid w_1 \mid \dots \mid w_{n-1}] \in \mathbb{R}^{m \times n}.$$

Hver række i W har middelværdi 0, så er centreret omkring sin middelværdi. Nu kan vi danne *kovariansmatricen*

$$C = \frac{1}{n-1}WW^T \in \mathbb{R}^{m \times m}.$$

For $i = j$ er c_{ii} variansen af række i ; for $i \neq j$ er c_{ij} kovariansen mellem række i og række j .

Kovariansmatricen er en symmetrisk matrix. Dens egenvektorer giver forskellige lineære kombinationer af datarække. Den tilhørende egenværdier er variansen for denne kombination. Oftest er man interesseret i de kombinationer der giver den største varians, da det beskriver bedst hvordan datapunkter adskiller sig mest. Dette svarer til den (numerisk) største egenværdi.

Som i vores diskussion ovenfor, er det mest præcist at beregne egenværdier af C fra singularværdierne af W . Hvis $W = U\Sigma V^T$ er en SVD for W , så har vi

$$C = \frac{1}{n-1}U\Sigma\Sigma^T U^T = U\left(\frac{1}{n-1}\Sigma\Sigma^T\right)U^T.$$

Egenværdierne af C er dermed $\frac{1}{n-1}\sigma_i^2$, hvor σ_i er singularværdierne for W .

Lad os nu betragte $U^T W$. Dette har kovariansmatrix

$$\begin{aligned} \frac{1}{n-1}(U^T W)(U^T W)^T &= \frac{1}{n-1}(\Sigma V)(\Sigma V)^T = \frac{1}{n-1}\Sigma V V^T \Sigma^T \\ &= \frac{1}{n-1}\Sigma\Sigma^T, \end{aligned}$$

som er diagonal. Dette er udtryk for at rækkerne i $U^T W$ er uafhængig af hinanden. Det vil sige disse lineær kombinationer af målinger er uafhængige af hinanden.

Eksempel 27.3. For datasættet i tabel 27.1 har vi

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

v = np.array([[54, 56, 56, 61, 66],
              [1.54, 1.52, 1.63, 1.72, 1.82]])

fig, ax = plt.subplots()
ax.plot(*v, 'o')
```

Middelværdien af en række kan beregnes via `np.mean`

```
print(np.mean(v[0]))
print(np.mean(v[1]))
```

```
58.6
1.6459999999999997
```

Vi kan beregne middelværdierne for alle rækker på en gang ved at bruge `np.mean` på `v` og angive `axis=1`

```
print(np.mean(v, axis=1))
```

```
[58.6    1.646]
```

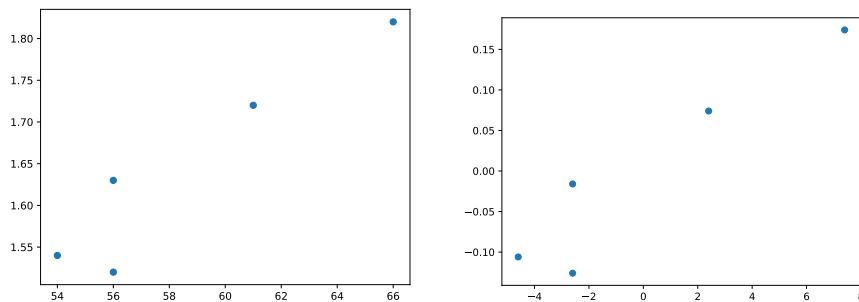
Resultatet kan fås som søjlevektor ved hjælp af `keepdims=True`

```
v_bar = np.mean(v, axis=1, keepdims=True)
print(v_bar)
```

```
[[58.6 ]
 [ 1.646]]
```

så den centrede matrix `w` er blot

27.2 PRINCIPALKOMPONENT ANALYSE



Figur 27.1: Plot af data og den centrede version.

```
w = v - v_bar  
print(w)
```

```
[[ -4.6  -2.6  -2.6   2.4   7.4 ]  
 [-0.106 -0.126 -0.016  0.074  0.174]]
```

```
fig, ax = plt.subplots()  
ax.plot(*w, 'o')
```

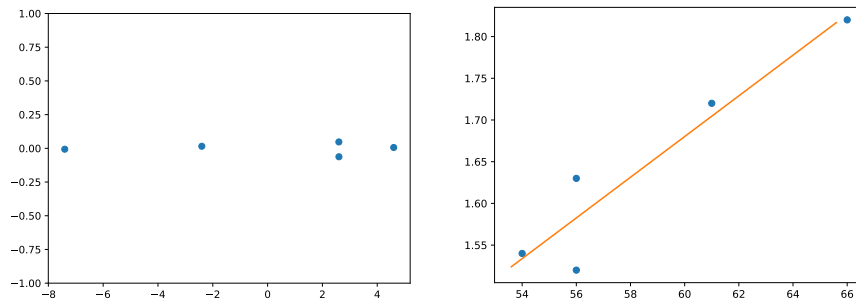
Se figur 27.1 for plots af den oprindelige datamatrix v og den centrede matrix w .
Kovariansmatricen er nu

```
m, n = v.shape  
c = (1 / (n - 1)) * w @ w.T  
print(c)
```

```
[[2.380e+01  5.805e-01]  
 [5.805e-01  1.578e-02]]
```

som viser større variation i vægterne end i højderne. Beregner vi SVD for w , får vi følgende singulærværdier

```
u, s, _ = np.linalg.svd(w, full_matrices=False)  
print(s)
```



Figur 27.2: Principale komponenter, uden normalisering.

```
[9.75995078 0.08050347]
```

Så den første retning har en relativt stort variation. En plot af den datasættet i via de nye måle kombinationer er givet i figur 27.2: til venstre plote via de to nye målekombinationer mod hinanden; til højre tegner vi den første principalretning, svarende til den største singulærværdi og dens målekombination, ovenpå den oprindelige datasæt.

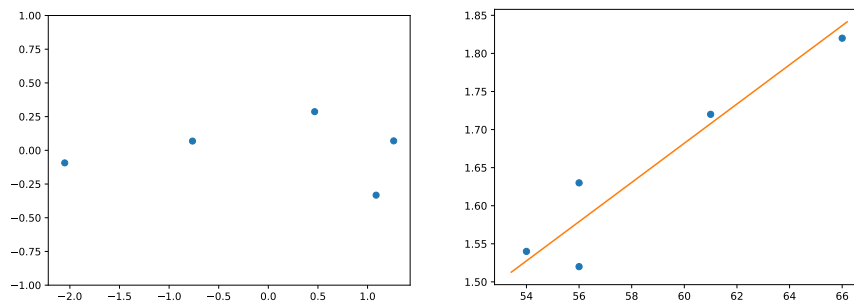
```
fig, ax = plt.subplots()
ax.set_ylim(-1, 1)
ax.plot(*u.T @ w, 'o')
```

```
fig, ax = plt.subplots()
ax.plot(*v, 'o')
ax.plot(*u[:, [0]] * np.linspace(-7, 5, 2) + v_bar))
```

En del af forklaringen på variationsstørrelsen er de forskellige enheder, og så numeriske størrelser, brugt for de forskellige målinger (vægt og højde). En standard måde at kompensere for dette, er at sørge for at dataet justeres så hver række har middelværdi 0 og varians 1. Disse skalæringsfaktorer er givet via kvadratroden af de diagonale indgange i kovariansmatricen c :

```
skalering = np.sqrt(np.diag(c))[:, np.newaxis]
w_ny = w / skalering
c_ny = (1 / (n - 1)) * w_ny @ w_ny.T
print(c_ny)
```

27.2 PRINCIPALKOMPONENT ANALYSE



Figur 27.3: Principale komponenter, normaliserede.

```
[[1.          0.94724047]
 [0.94724047 1.          ]]
```

Vi kan beregne principale komponenter for det normaliserede data vi singulær-værdidekomponeringen

```
u_ny, s_ny, _ = np.linalg.svd(w_ny)
print(s_ny)
```

```
[2.79087117 0.45938884]
```

De nye plots disse giver er

```
fig, ax = plt.subplots()
ax.set_ylim(-1, 1)
ax.plot(*(u_ny.T @ w_ny), 'o')
```

```
fig, ax = plt.subplots()
ax.plot(*v, 'o')
ax.plot*((u_ny * skalering)[: , [0]]
         * np.linspace(-2.2, 1.5, 2) + v_bar))
```

Se figur 27.3.

△

Kapitel 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 Gausseliminerings

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

$$\begin{aligned}\text{(I)} \quad R_i &\leftrightarrow R_j & a[i, j], : &= a[j, i], : \\ \text{(II)} \quad R_i &\rightarrow sR_i \ (s \neq 0) & a[i, :] &*= s \\ \text{(III)} \quad R_i &\rightarrow R_i + tR_j \ (j \neq i) & a[i, :] &+= t * a[j, :]\end{aligned}$$

og at de kan realiseres via matrixprodukter. F.eks. for $n = 2$ kan operationerne $R_0 \leftrightarrow R_1$, $R_1 \rightarrow sR_1$ og $R_1 \rightarrow R_1 + tR_0$ 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) $R_1 \rightarrow R_1 + tR_0$ efterfulgt af (b) $R_2 \rightarrow R_2 + rR_0$ 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) $R_i \rightarrow R_i + tR_j$ med $j < i$. Arbejdes der systematisk ved at indføre nultal under diagonalen i hver søjle får vi

$$\begin{aligned} A = \begin{bmatrix} 2 & 3 & -4 \\ 3 & -4 & 1 \\ 1 & 1 & 2 \end{bmatrix} &\xrightarrow[\substack{R_1 \rightarrow R_1 - \frac{3}{2}R_0 \\ R_2 \rightarrow R_2 - \frac{1}{2}R_0}]{L_1 A} \begin{bmatrix} 2 & 3 & -4 \\ 0 & -17/2 & 7 \\ 0 & -1/2 & 4 \end{bmatrix} \\ &\xrightarrow{R_2 \rightarrow R_2 - \frac{1}{17}R_1} L_2 L_1 A = \begin{bmatrix} 2 & 3 & -4 \\ 0 & -17/2 & 7 \\ 0 & 0 & 61/7 \end{bmatrix} = U \end{aligned}$$

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

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

28.1 LU-DEKOMPONERING OG GAUSSELIMINERING

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

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

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

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

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 nemt at beregne,

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

(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  $i \in \{0, 1, \dots, n-2\}$ :
3       $L_{[i+1:, [i]]} = U_{[i+1:, [i]]} / u_{ii}$ 
4       $U_{[i+1:, i]} = U_{[i+1:, i]} - L_{[i+1:, [i]]} U_{[[i], i]}$ 
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** $i \in \{n-1, \dots, 1, 0\}$:
- 2 $x_i = (b_i - U_{[i, i+1:]} x_{[i+1:]}) / u_{ii}$
- 3 **return** x

og en tilsvarende algoritme giver forward substitution. Backsubstitution 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 Gausseliminerings 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** $i \in \{0, 1, \dots, n-2\}$:
- 3 Vælg $j \geq i$ 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

28.1 LU-DEKOMPONERING OG GAUSSELIMINERING

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 Gausseliminerings 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

```
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 Gausseliminerings.

```
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 $2 \cdot (10 - 1) = 512$., så fejlen har også

28.1 LU-DEKOMPONERING OG GAUSSELIMINERING

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 Gausseliminerings på en $(n \times n)$ -matrix.

Lad os kigge 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)
```

```
[[ 7.57469357]  
 [ 6.69341503]  
 [ 2.30839297]  
 [-0.38952093]  
 [-4.13018685]  
 [ 9.06129482]  
 [-0.7264079 ]  
 [-4.99335204]  
 [-0.59959028]  
 [-1.50109608]]
```

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)

```

```

[[ 1.86655782]
 [ 2.8518371 ]
 [ 1.31865213]
 [-0.06060963]
 [-3.86188519]
 [ 5.46771129]
 [ 1.14771985]
 [-1.97150443]
 [ 0.4507529 ]
 [ 5.70813575]]

```

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]
 [ 8.88178420e-16]
 [-4.44089210e-16]

```


28.1 LU-DEKOMPONERING OG GAUSSELIMINERING

```
[ 2.77555756e-16]
[ 1.77635684e-15]
[ 1.42108547e-14]
[-6.32827124e-15]
[-9.41469125e-14]
[-1.63313807e-13]
[-3.22852856e-13]]
```

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)))
```

1.030149953393586e-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)
```

```
[[-8.88178420e-16]
 [-2.66453526e-15]
 [-1.33226763e-15]
 [-1.94289029e-15]
 [ 8.88178420e-16]
 [-7.10542736e-15]
 [ 1.11022302e-16]
 [ 1.77635684e-15]
 [ 9.99200722e-16]
 [ 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: 15.67081517439916
Afvigelse Ax fra b
LU metoden: 14.147568678476691
QR metoden: 1.7053025658242404e-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 eliminerings med ombytning, og bekræfter at det virker i et tilfældig eksempel.

28.1 LU-DEKOMPONERING OG GAUSSELIMINERING

```
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)
```

7.771589244938637

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.

Appendiks A

Det græske alfabet

α	A	alpha	ν	N	nu
β	B	beta	ξ	Ξ	xi
γ	Γ	gamma	o	O	omicron
δ	Δ	delta	π	Π	pi
ε	E	epsilon	ρ	R	rho
ζ	Z	zeta	σ	Σ	sigma
η	H	eta	τ	T	tau
θ	Θ	theta	υ	Y	ypsilon
ι	I	jota	φ	Φ	phi
κ	K	kappa	χ	X	chi
λ	Λ	lambda	ψ	Ψ	psi
μ	M	mu	ω	Ω	omega

Appendiks B

Python

Programmeringssproget python kan hentes fra

<https://www.python.org>

Vær opmærksom på at der er nogle væsentlige forskel mellem python version 2.* og 3.*. Vi bruger python 3, mere præcist

```
>>> import sys
>>> print(sys.version)
3.9.5 (default, May 8 2021, 11:22:23)
[Clang 11.0.3 (clang-1103.0.32.62)]
```

Nogle computer har python 2 installeret i forvejen, men ikke python 3. I sådanne tilfælde bliver man nødt til at installere python 3 selv.

Desuden bruger vi pakker fra SciPy samlingen, som findes ved

<https://www.scipy.org>

Specielt NumPy og Matplotlib

```
>>> import numpy
>>> print(numpy.__version__)
1.20.3
>>> import matplotlib
>>> print(matplotlib.__version__)
3.4.2
```

B PYTHON

Installationsvejledning findes ved samlingens hjemmeside.

For at skrive python filer er det nyttigt at have jupyter lab som kan hentes fra

<https://jupyter.org>

I jupyter lab kan man oprette notebooks, som kombinerer kode og almindelig tekst.

Der findes nogle distributioner som anaconda,

<https://www.anaconda.com>

der inkluderer python og alle de overnævnte tillægspakker.

Bibliografi

- Golub, G. H. og C. F. Van Loan (2013). *Matrix computations*. 4. udg. Johns Hopkins Studies in the Mathematical Sciences. Johns Hopkins University Press, Baltimore, MD. ISBN: 978-1-4214-0794-4.
- Leon, S. J. (2015). *Linear algebra with applications*. ninth edition, global edition. Harlow: Prentice Hall. ISBN: 978-1-292-07059-9.
- Poore, G. M. (jul. 2015). »PythonTeX: reproducible documents with LaTeX, Python, and more«. I: *Comput. Sci. Discov.* 8.1, 014010. DOI: [10.1088/1749-4699/8/1/014010](https://doi.org/10.1088/1749-4699/8/1/014010).
- The NumPy community (29. jun. 2020). *NumPy user guide*. Vers. release 1.19.0. 214 s. URL: <https://www.scipy.org/docs.html>.
- 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](https://doi.org/10.1137/1.9780898719574).

Python indeks

- (differens)
 - array, [32](#)
 - float, [2](#), [3](#)
- # (kommentar), [10](#)
- () (tupel), [31](#)
- * (produkt)
 - float, [2](#), [3](#)
 - skalar-array, [22](#), [31](#)
- * (unpack), [56](#)
- ** (potens)
 - float, [3](#)
- ** (unpack dictionary), [258](#)
- *=, [60](#)
- + (sum)
 - array, [32](#)
 - float, [2](#), [3](#)
- +=, [60](#)
- / (kvotient)
 - float, [2](#), [3](#)
- // (floor divide), [6](#)
- /=, [177](#)
- : (indeksering), [26](#)
- =, [177](#)
- = (sæt et variabel), [10](#)
- @ (matrixprodukt), [40](#)

A

[all](#), [86](#), [312](#)
[allclose](#), [86](#)

[arccos](#), [80](#)
[arctan2](#), [19](#)
[argsort](#), [291](#)
[argwhere](#), [324](#)
[array](#), [21](#), [26](#)
[array_str](#), [290](#)
[axhline](#), [249](#)

B

[back_subs](#), [345](#)
[break](#), [291](#)

C

[Chebyshev](#), [166](#)
[clim](#), [29](#)
[cmap](#), [28](#)
[color](#)

- [plot](#), [22](#)

[colorbar](#), [28](#)
[complex](#), [140](#)
[compute_uv](#), [329](#)
[cond](#)

- [linalg.cond](#), [134](#)

[conj](#), [143](#)
[convert](#)

- [polynomial](#), [166](#)

[copy](#), [177](#)

D

[def](#), [104](#)

PYTHON INDEKS

default_rng, 49
delete, 247
diag, 114
dtype, 27
 float, 27

E

e
 i float, 5
eksempel_mat, 343
empty, 81
empty_like, 82
enumerate, 119
exp, 7, 92
eye, 206

F

False, 117
f'...:e' (videnskabelig notation), 198
finfo (float information), 6
flip, 292
float, 5
 dtype, 27
for, 53, 56
forbedret_gram_schmidt, 177
forward_subs, 345
from, 166
full_matrices, 117

G

gauss, 348
gauss_uden_ombytning, 343

H

heat_map, 320
hessenberg, 304
hessenberg_data, 303
hessenberg_qh, 304

house, 203, 301
house_plus, 307
house_transformation, 104
householder_lsqr, 209
householder_qr, 206
householder_qr_data, 204
hstack, 60

I

if...else, 103
imag
 complex, 140
import, 6
imread, 120
imshow, 120
indeksering, 27
 ndarray, 26
inv
 linalg.inv, 71

J

j, 140

K

keepdims, 334
klassisk_gram_schmidt, 176

L

label
 plot, 92
legend, 92
Legendre, 166
linalg
 linalg.cond, 134
 linalg.inv, 71
 linalg.norm, 79, 149
 linalg.qr, 169
 linalg.solve, 187
 linalg.svd, 113, 117, 122

linspace, 48
retstep, 156

M

marker
plot, 22
markersize, 116
markevery, 257
matplotlib, 21
matshow, 28, 48, 119
mean, 334

N

nbytes
ndarray, 124
ndarray, 26
ndim, 26
ndindex, 119
norm
linalg.norm, 79, 149
normal, se rng
np, se numpy
numpy, 6

O

ones, 72, 91
ones_like, 159

P

pi, 19
plot, 22
polynomial
Chebyshev, 166
convert, 166
Legendre, 166
Polynomial, 166
praktisk_qr_metode, 324
print, 49, 54

Q

qr
householder_qr, 206
linalg.qr, 169

R

random, 49
range, 54
real
complex, 140
reversed, 206
rng, se også default_rng
normal, 135
random, 49
standard_normal, 49

S

savefig, 22
seismic, 319
semi_praktisk_qr_metode, 322
set_aspect, 22
set_printoptions, 119
set_xlabel, 122
set_xlim, 23
set_ylabel, 122
set_ylim, 23
seterr, 9
shape
ndarray, 26
show
billede, 120
matrix, 28
plot, 22
shrink, 29
solve
linalg.solve, 187
sqrt, 10
standard_normal, se rng

PYTHON INDEKS

subplots, [22](#), [119](#)

svd

 linalg.svd, [113](#), [117](#), [122](#)

svdapprox, [119](#), [123](#)

T

T (transponering), [33](#)

text

 subplot, [117](#)

tight_layout, [119](#)

tilvækst, [349](#)

trapz, [158](#)

tri, [343](#)

tridiagonal_data, [307](#)

tridiagonal_qt, [307](#)

triu, [72](#)

True, [291](#)

V

vander, [187](#)

vdot, [104](#), [149](#)

W

while, [291](#)

Z

zeros, [31](#)

Indeks

1-norm, 130

2-norm, 79

kompleks, 148

matrix, 131

∞ -norm, 131

A

afbildning

lineær, 211

akse, 26

Algebraens fundamentalsætning,
267

andengradslikning, 10, 11

B

back substitution, 59, 342

basis, 221

ortonormal, 99

billedmængde, 216

bundne variabler, 64

C

cache, 53

Cauchy-Schwarz ulighed, 83

Chebychev polynomier, 166

cosinus, 80

D

determinant, 70, 248

diagonaliserbar, 244

differens

float, 3

differentialligninger, 253

førsteordenssystem, 254

dimension, 228

E

echelonform, 62

egenvektor, 240

egenværdi, 240

eksponent, 5

eksponentialfunktion, 7, 253

kompleks, 259

elektrisk

induktionsspole, 212

kondensator, 212

kredsløb, 43, 213

modstand, 212

elementær

matrix, 74

elementære

operationer på ligninger, 58

rækkeoperationer, 58

enhedsvektor, 14, 79

epsilon

machine, 6

Eulers identitet, 260

INDEKS

F

F1, *se* float, aksiomer

F2, *se* float, aksiomer

farvelægning

af en matrix, 28

farveskala, 28

grænser, 29

fejl, 3

relativ, 4

float, 5

aksiomer, 9

flop, 51

flydende-komma repræsentation, 5

fordeling

standard normal-, 49

uniform-, 49

for-løkke, 51, 53

Fourier cosinus række, 154

Fourier række, 154

frie variabler, 64

Frobeniusnorm, 149

G

Gausseliminering

med ombytning, 342

uden ombytning, 341

Grammatrix, 98

Gram-Schmidt

forbedret, 174, 175, 177

klassisk, 161, 167, 176

H

heatmap plot, 28

hermitisk

matrix, 276

Hessenbergform, 299

Householder

matrix, 100

vektor, 100

I

identitetsmatrix, 17, 47, 69

imaginær del, 139

indeksering, 26, 27

matrix, 25

negative, 27

indgang, 25

indre produkt, 15, 77, 147

L^2 -, 150

over \mathbb{C} , 148

vægtet, 149

L^2 -, 150

induceret norm, 148

induktionsspole, 212

integration

numerisk, 155

invers

af produkt, 71

invers matrix, 69, 73

2×2 , 70

via rækkeoperationer, 75

inverspotensmetoden, 293

invertibel, 69, 73

K

karakteristisk polynomium, 249

$n = 2$, 241

kerne, 216

Kirchhoffs lov, 43, 44

kompleks tal, 139

eksponentialfunktion, 259

imaginær del, 139

konjugerede, 142

numerisk værdi, 143

reel del, 139

kondensator

elektrisk, 212
 konditionstal, 127, 128
 absolut, 127
 for en invertibel matrix, 133
 for en matrix, 136
 mindste kvadraters metode, 196
 konjugerede
 kompleks tal, 142
 koordinatskiftmatrix, 225
 koordinatvektor, 224
 kovarians, 332
 kovariansmatrix, 333
 kredsløb
 elektrisk, 43
 kvadratisk matrix, 69
 kvotient
 float, 3

L
 L^2 -indre produkt, 150
 Legendre polynomier, 165
 ligningssystem
 lineær, 43, 70
 lineær
 afbildning, 211
 standard matrixrepræsentation, 214
 kombination, 45, 144
 ligningssystem, 43, 70, 73
 regression, 188
 transformation, 211
 lineært
 uafhængig, 164
 linje
 ret, 39
 LU-dekomponering, 340

M
 machine epsilon, 6
 maksimumsnorm, 131
 mantisse, 5
 matplotlib, 21
 matrix, 25
 $(m \times n)$ -, 25
 af en lineær transformation, 229, 233
 diagonaliserbar, 244
 difference, 32
 elementær, 74
 Gram, 98
 Hermitisk, 276
 Hessenbergform, 299
 Householder, 100
 identitets-, 17, 47, 69
 invers, 69, 70, 73
 via rækkeoperationer, 75
 koordinatskift, 225
 kvadratisk, 69
 lighed, 25
 nul-, 31
 ortogonal, 95
 produkt, se matrixprodukt
 projektion, 85, 89
 pseudoinvers, 186
 rang, 185
 repræsentation
 generel, 229
 standard, 214
 rotation, 95, 107
 søjlerum, 144
 spejling, 95
 stokastisk, 289
 sum, 31, 32
 symmetrisk, 85
 tridiagonal, 300, 319

INDEKS

- udvidet, 58
- under, 247
- unitær, 269
- Vandermonde, 187
- matrixprodukt, 40, 52
 - Strassen, 53
- matrix-vektorprodukt, 45
- middelværdi, 332
- mindste kvadraters metode, 183, 184
 - konditionstal, 196
 - via QR, 184
 - via SVD, 186
- modstand
 - elektrisk, 212
- multiplikation
 - med en skalar, 31–34, 137

N

- nøjagtighed
 - af float, 6
- norm, 14, 130, 147
 - ∞ -, 131
 - 1-, 130
 - 2-, 79, 148
 - 2- for matrix, 131
 - Frobenius-, 149
 - induceret af indre produkt, 147
 - maksimums-, 131
 - operator, 131
 - p -, 131
- normalfordeling
 - standard, 49
- normalligninger, 193
- nullitet, 231
- nulmatrix, 31
- numerisk værdi, 143

O

- Ohms lov, 44
- operatornorm, 131
- ortogonal
 - iteration, 315
 - matrix, 95
 - vektorer, 77, 86, 147
- ortogonale polynomier
 - Chebyshev, 166
 - Legendre, 165
- ortonormal
 - basis, 99
 - vektorer, 86
- over fitting, 191
- overflow, 7

P

- page rank, 290
- parallel, 80
- Parsevals identitet, 88
- Perron-Frobenius sætning, 290
- π , 19
- pil, 13, 34
- pivotelement, 62
- plan, 39
- plot
 - heatmap, 28
- p -norm, 131
- polynomium, 38
- potens
 - float, 3
- potensmetoden, 283
- prikprodukt, 15
- principalkomponent, 332
- produkt
 - float, 3
 - indre, 77, 147
 - matrix, 40, 52

- matrix-vektor, 45
- række-søjle, 37, 51
- skalar-vektor, 51
- ydre, 48
- projektion, 89
 - matrix, 85, 89
 - på en linje, 84
- pseudoinvers, 186
- Pythagoras sætning, 79
- Q**
- QR-dekomponering, 168
 - fuld, 168
 - tynd, 168, 176, 184
- QR-metoden, 311, 322
- R**
- $\mathbb{R}^{m \times n}$, 25
- RAM, 53
- rang, 185, 231
- rang-nullitetsformlen, 231
- Rayleighkvotientmetoden, 296
- Rayleighs kvotient, 292
- reel del, 139
- regression
 - lineær, 188
- relativ fejl, 4
- restvektoren, 183
- ret linje, 39
- række
 - af en matrix, 26
- rækkeoperationer, 58
 - i python, 60
- række-søjleprodukt, 51
- rækkevektor, 16, 35
- rotation, 95
- S**
- $S(A)$, 144
- Simpsons regel, 158
- singulærvektor, 115
 - højre-, 115
 - venstr-, 115
- singulærværdi, 111, 115
- singulærværdidekomponering, 111, 113, 115
 - forkortet, 116
 - reduceret, 185
 - tynd, 111, 115, 119, 122
 - ydre produkt, 114, 115
- sinus, 106
- skalar, 25, 137
- skalarmultiplikation, se multiplikation, med en skalar
- skalar-vektorprodukt, 51
- SMR, se standard matrixrepræsentation
- søjle
 - af en matrix, 26
- søjlerum, 144
- søjlevektor, 34
- Span, 144
- spejling, 95
- SSD, 53
- standard matrixrepræsentation, 214
- stokastisk matrix, 289
- stor-»O«-notationen, 286
- Strassen
 - matrixprodukt, 53
- sum
 - float, 3
 - matrix, 31, 32
 - vektor, 34
- SVD, se singulærværdidekomponering
- symmetrisk matrix, 85

INDEKS

T

tilfældighedsgenerator, 49
transformation
 lineær, 211
transponering
 af produkt, 96
 matrix, 33
 vektor, 15
trapezregel, 158
trekantsulighed, 130
tridiagonal matrix, 300, 319
trigonometriske identiter, 106
tupel, 31

U

udspændt, 144
udvidet matrix, 58
underflow, 7
undermatrix, 247
underrum, 143
uniformfordeling, 49
unitær
 matrix, 269

V

Vandermode
 vektor, 38
Vandermonde
 matrix, 187
varians, 332
vektor, 34, 137
 enheds-, 79
 parallelle, 80
 række, 35
 søjle, 34
 sum, 34
vektorrum, 137
vinkel, 80, 148
 cos, 80
vinkelret, 77
vægtet indre produkt, 149

W

Wilkinsons shift, 324

Y

ydre produkt, 48