Numerisk Lineær Algebra F2021
Notesæt 1

Andrew Swann

2. februar 2021

Sidst ændret: 1. februar 2021.
Versionskode: baf566b.

Indhold

Tabeller

0 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)
3.9.1 (default, Dec  8 2020, 11:42:15)
[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__)
1.19.5
>>> import matplotlib
>>> print(matplotlib.__version__)
3.3.4
```

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.

1 Indledning og første betragtninger

NUMERISK LINEÆR ALGEBRA er studiet af metode for behandling af beregningsproblemer indenfor matricer og vektorer. Det er et hjørnesten i computerberegninger baseret på matematiske modeller for problemstillinger fra alle videnskaber. Det er fag der har udviklet sig hastigt og er under stadig forandring. Grundlaget er teorien for vektorrum og lineære transformationer. Konkret arbejde med tal betyder at disse teori 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 problemstilling 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 kik 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 computerbilled 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 hurtig:

```
>>> 1.1 + 1.2
2.3
>>> 1.9 + 1.2
3.0999999999999996
>>> 1.1 - 1.2
-0.09999999999999987
>>> 2.1 * 2.7
5.670000000000001
>>> 2.1 / 2.0
1.05
>>> 2.1 / 2.5
0.8400000000000001
```

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.

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 arbejder 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

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

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

$$x' = x(1+\delta). \tag{1.1}$$

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

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

	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 $b' = 9{,}986$.

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 × 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 $a_{\min} \leqslant a \leqslant a_{\max}$ og $b_{\min} \leqslant b \leqslant b_{max}$ gælder*

(a) $a_{\min} + b_{\min} \leqslant a + b \leqslant a_{\max} + b_{\max}$,
(b) $a_{\min} - b_{\max} \leqslant a - b \leqslant a_{\max} - b_{\min}$.

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

(c) $a_{\min} b_{\min} \leqslant ab \leqslant a_{\max} b_{\max}$,
(d) $a_{\min}/b_{\max} \leqslant a/b \leqslant a_{\max}/b_{\min}$. □

1.2 Flydende-komma repræsentation

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

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

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

$$m = \text{“}d_1{,}d_2 \ldots d_r'' = d_1 + \frac{d_2}{10} + \cdots + \frac{d_r}{10^{r-1}}$$

er en decimalbrøk med r cifre, $d_i \in \{0, 1, 2, \ldots, 9\}$, $d_1 \neq 0$, og k er et heltal. Decimaltallet 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`:

```
>>> 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 $\text{float}_{\text{max}}$ og et mindste tal $\text{float}_{\text{min}}$ 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:

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

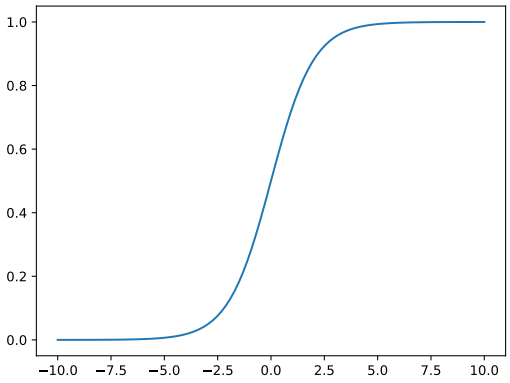
(d) Resultater af regneoperationer med tal af type `float` skal generelt afrundes for at give et svar der igen er af type `float`.

(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

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

som standard ledsaget af en advarsel

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

Hvis vi forsørger 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 $\sigma(750)$ bruger `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 $|\delta_x| \leqslant \epsilon_{\text{machine}}/2$. Dette er det samme, som at siger

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

For python `float`, vi kan forventer 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

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

med $|\delta_i| \leqslant \epsilon_{\text{machine}}/2$ 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 diskriminaten 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
```

Tilgengæld er udregning af r uden dette problem

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

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 $\operatorname{sgn}(b)$ af b.

Proposition 1.5. *Givet*

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

er rødderne r_1, r_2 af

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

givet ved

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

Python indeks

Indeks