
Matematikk 1000

Øvingsoppgaver i numerikk – leksjon 7

Løsningsforslag

Oppgave 1 – Halveringsmetoden – igjen

- a) I skriptet vårt fra leksjon 6 skal altså linje 16 erstattes med “`while abs(b-a)>1e-3`”. Når vi gjør det, ser skriptet slik ut:

```
1  % Implementering av halveringsmetoden for likninga f(x)=0 der
2  % f er kontinuerlig på [a,b] og f(a) og f(b) har ulike fortegn.
3
4  % Grenser
5  a=0;
6  b=pi/2;
7
8  % Gir funksjonen som skal være null
9  NullFunk=@(x) sqrt(x)-cos(x);
10
11 % Funksjonsverdier
12 Fa=NullFunk(a);
13 Fb=NullFunk(b);
14
15 % Starter while-løkke som kjøres til |b-a|<=1e-3
16 while abs(b-a)>1e-3
17     c=(a+b)/2;          % Midtpunktet
18     Fc=NullFunk(c); % Funksjonsverdi i midtpunktet
19     if Fa*Fc<0
20         b=c;            % Setter ny b til c
21     else
22         a=c;            % Setter ny a til c
23     end
24 end
25
26 % Regner ut nytt midtpunkt og skriver svaret til skjerm
27 x=(a+b)/2
```

Merk at vi her også har oppdatert kommentaren i linje 15. Vi kjører skriptet:

```
>> Halvering
x =
    0.6416
```

Vi fikk et noe annet svar enn det vi fikk i leksjon 6; der fikk vi $x = 0.6420$ med 10 iterasjoner. Selv om vi ikke vet om dette svaret er mer eller mindre nøyaktig enn det svaret vi fikk sist, har dette svaret en klar fordel: Vi vet at feilen ikke er større enn 0.001. Eller, siden vi har regna ut midtpunktet mellom a - og b -verdiene vi fikk til slutt, vet vi at feilen er mindre enn 0.0005. `while`-løkka har nemlig kjørt helt til intervallet $[a, b]$ har bredden 0.001. Selvsagt kan vi sette dette tallet enda lavere og få et mer nøyaktig svar. Om vi endrer linja til `while abs(b-a)>1e-5`, får vi dette svaret:

```
>> format long
>> Halvering
x =
    0.641716294950902
```

Her har vi skrevet ut svaret vårt med litt flere desimaler.

Dette poenget står ganske sentralt i kurset: *Selv om vi bare finner en tilnærma løsning, kan vi få løsningen til å være så nøyaktig som vi selv måtte ønske.* Og da er det kanskje ikke så farlig at den ikke er eksakt...

b) $2x - \sqrt{x} = 4$

Vår “nye” $f(x)$ blir altså $f(x) = 2x - \sqrt{x} - 4$. Siden $f(0) = -4$ og $f(4) = 2$, og f er kontinuerlig, må f ha minst ett nullpunkt på intervallet $[0, 4]$. Vi justerer `a` og `b` tilsvarende i linje 5 og 6, og oppdaterer $f(x)$ i linje 9. I `while`-linja, linje 16, setter vi nøyaktigheten til $2 \cdot 10^{-5}$. Med disse endringene blir skriptet vårt seende slik ut:

```
1  % Implementering av halveringsmetoden for likninga f(x)=0 der
2  % f er kontinuerlig på [a,b] og f(a) og f(b) har ulike fortegn.
3
4  % Grenser
5  a=0;
6  b=4;
7
8  % Gir funksjonen som skal være null
9  NullFunk=@(x) 2*x-sqrt(x)-4;
10
11 % Funksjonsverdier
12 Fa=NullFunk(a);
13 Fb=NullFunk(b);
14
15 % Starter while-løkke som kjøres til |b-a| er liten nok
16 while abs(b-a)>2e-5
17     c=(a+b)/2;          % Midtpunktet
```

```

18 Fc=NULLFunk(c); % Funksjonsverdi i midtpunktet
19 if Fa*Fc<0
20     b=c;          % Setter ny b til c
21 else
22     a=c;          % Setter ny a til c
23 end
24 end
25
26 % Regner ut nytt midtpunkt og skriver svaret til skjerm
27 x=(a+b)/2

```

Vi kjører skriptet og får (med mange desimaler)

```

>> Halvering
x =
    2.843070983886719

```

Likninga kan løses eksakt. Det er en andregradslikning – ikke i x men i $\sqrt{x}$. Om vi setter $u = \sqrt{x}$, slik at $x = u^2$, får vi

$$\begin{aligned}
 2u^2 - u - 4 &= 0 \\
 u &= \frac{-(-1) \pm \sqrt{(-1)^2 - 4 \cdot 2 \cdot (-4)}}{2 \cdot 2} = \frac{1 \pm \sqrt{33}}{4} \\
 \sqrt{x} &= \frac{1 + \sqrt{33}}{4} \\
 x &= \left(\frac{1 + \sqrt{33}}{4} \right)^2 = \frac{(1 + \sqrt{33})^2}{16}
 \end{aligned}$$

Vi har her benyttet oss av at $\sqrt{x}$ ikke kan være negativ. Vi sammenligner løsninga vår med det eksakte svaret:

```

>> Eksakt=(1+sqrt(33))^2/16;
>> Eksakt-x
ans =
    -6.530694651729618e-07

```

Feilen er altså $6.53 \cdot 10^{-7}$; vi er godt innenfor.

- c) Før **while**-løkka lager vi oss en variabel vi kan kalle **teller** og setter denne til å være 0. Inni **while**-løkka lar vi denne øke med 1 for hver gang, **teller=teller+1**; . Til slutt, etter **while**-løkka, skriver vi **teller** til skjerm. Med dette kan skriptet se slik ut:

```

1 % Implementering av halveringsmetoden for likninga f(x)=0 der
2 % f er kontinuerlig på [a,b] og f(a) og f(b) har ulike fortegn.
3
4 % Grenser
5 a=0;

```

```

6  b=4;
7
8  % Gir funksjonen som skal være null
9  NullFunk=@(x) 2*x-sqrt(x)-4;
10
11 % Funksjonsverdier
12 Fa=NullFunk(a);
13 Fb=NullFunk(b);
14
15 % Teller antall iterasjoner
16 teller=0;
17
18 % Starter while-løkke som kjøres til |b-a| er liten nok
19 while abs(b-a)>2e-5
20     c=(a+b)/2;          % Midtpunktet
21     Fc=NullFunk(c); % Funksjonsverdi i midtpunktet
22     if Fa*Fc<0
23         b=c;            % Setter ny b til c
24     else
25         a=c;            % Setter ny a til c
26     end
27     teller=teller+1    % Lar antall iterasjoner øke med én
28 end
29
30 % Regner ut nytt midtpunkt og skriver svaret til skjerm
31 x=(a+b)/2
32 % Skriver antall iterasjoner til skjerm
33 Iterasjoner=teller

```

Vi tester at det fungerer:

```

>> Halvering
x =
    2.843070983886719
Iterasjoner =
    18

```

Vi måtte iterere 18 ganger for å oppnå den presisjonen vi ville ha.

Oppgave 3 – Newtons metode (og litt til)

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)} \quad .$$

- a) Om vi antar at $x_{n+1} = x_n$ og setter dette inn i iterasjonsuttrykket over, får vi

$$\begin{aligned} x_n &= x_n - \frac{f(x_n)}{f'(x_n)} \\ \frac{f(x_n)}{f'(x_n)} &= x_n - x_n \\ f(x_n) &= 0 \end{aligned}$$

- b) Med $f(x) = \sqrt{x} - \cos x$ blir iterasjonsskjemaet seende slik ut:

$$x_{n+1} = x_n \frac{\sqrt{x_n} - \cos x_n}{\frac{1}{2\sqrt{x_n}} + \sin x_n} \quad .$$

Newtons metode er nok enklest å implementere som ei **for**-løkke:

```
1 % Skript som implementerer Newtons metode
2
3 % Bestemmer x_0
4 x=1;
5
6 for i=1:5
7     x=x-(sqrt(x)-cos(x))/(1/(2*sqrt(x))+sin(x)) % Iterasjonsformel
8 end
```

Her har vi bestemt oss for å starte med $x_0 = 1$ og bruke 5 iterasjoner. Vi tester:

```
>> format long
>> NewtonMetFor
x =
    0.657318198203377
x =
    0.641746105250497
x =
    0.641714371002502
x =
    0.641714370872883
x =
    0.641714370872883
```

Etter bare 5 iterasjoner har vi tydeligvis fått svaret med 15 desimaler!

Her har vi fiksert antall iterasjoner på forhånd – uten å vite hvor mange vi strengt tatt trenger. Dersom vi ønsker å bare iterere så mange ganger vi trenger for en gitt presisjon, kan vi bruke ei **while**-løkke. I skriptet `NewtonMetWhile.m` har vi implementert dette:

```
% Skript som implementerer Newtons metode

% Bestemmer x_0
x=1;

% Presisjon
Pres=1e-5;

% Lager en variabel for forrige x-verdi
xGml=100;

while abs(x-xGml)>Pres
    xGml=x;
    x=x-(sqrt(x)-cos(x))/(1/(2*sqrt(x))+sin(x)); % Iterasjonsformel
end

% Skriver svaret til skjerm
x
```

Her får vi bare skrevet det endelige svaret til skjerm:

```
>> NewtonMetWhile
x =
    0.641714370872883
```

Merk at vi i linje 10 har tilordna et nokså tilfeldig tall til variabelen `xGml`. Pass på at tallet er tilstrekkelig ulikt x_0 i linje 4 til at **while**-løkka kommer i gang.

c) Som i oppgave 2 innfører vi en variabel som teller antall iterasjoner:

```
1 % Skript som implementerer Newtons metode
2
3 % Bestemmer x_0
4 x=1;
5
6 % Presisjon
7 Pres=1e-5;
8
9 % Teller antall iterasjoner
10 teller=0;
11
12 % Lager en variabel for forrige x-verdi
```

```

13  xGml=100;
14
15  while abs(x-xGml)>Pres
16      xGml=x;
17      x=x-(sqrt(x)-cos(x))/(1/(2*sqrt(x))+sin(x)); % Iterasjonsformel
18      teller=teller+1;
19  end
20
21  % Skriver svaret til skjerm
22  x
23  % Skriver antall iterasjoner til skjerm
24  Iterasjoner=teller

```

Vi kjører skriptet igjen:

```

>> NewtonMetWhile
x =
    0.641714370872883
Iterasjoner =
    4

```

Her klarte vi oss med bare 4 iterasjoner for å få (minst) fire rette desimaler. Typisk vil Newtons metode gi et nøyaktig svar etter langt færre iterasjoner enn halveringsmetoden. Men halveringsmetoden har en klar fordel: Den er mye mer “idiotsikker”. Dersom vi har kontrollert at $f(a)$ og $f(b)$ har ulike fortegn og at $f(x)$ er kontinuerlig på $[a, b]$, kan vi være helt sikre på at vi har funnet et riktig nullpunkt med halveringsmetoden. Slik er det ikke med Newtons metode. Hvis vi har å gjøre med en funksjon som gir veldig høye verdier for den deriverte, kan vi få at $x_{n+1} \approx x_n$ selv om vi slett ikke har noe nullpunkt. Dessuten kan det hende at metoden ikke går mot noen bestemt verdi i det hele. Vi ser også at iterasjonsformelen bryter sammen dersom $f'(x_n) = 0$. Hvis den deriverte er *nær* 0, vil vi typisk finne et helt annet nullpunkt enn det vi “sikta oss inn på” da vi valgte x_0 ; der er ingen garanti for at Newtons metode, når den fungerer, vil gi det nullpunktet som ligger nærmest vårt valg av x_0 . Med halveringsmetoden, derimot, har vi i utgangspunktet sikra at nullpunktet skal ligge mellom a og b .

Så hva bruker vi i praksis, da – halveringsmetoden eller Newtons metode? -Stort sett Newtons metode.

- d) Vi skal nå finne nullpunkt for funksjonen $2x - \sqrt{x} - 4$, som har derivert $2 - \frac{1}{2\sqrt{x}}$. Vi oppdaterer iterasjonsformelen i linje 17:

```

x=x-(2*x-sqrt(x)-4)/(2-1/(2*sqrt(x)));

```

Om vi beholder x_0 og presisjonen **Pres** vi hadde fra før, finner vi svaret etter 4 iterasjoner:

```
>> NewtonMetWhile
x =
    2.843070330817254
Iterasjoner =
    4
```

Hadde vi starta med $x_0 = 3$, hadde det holdt med 3 iterasjoner.

- e) Vi skal altså løse likninga $x = g(x)$. Om vi itererer på $x_{n+1} = g(x_n)$ og får at $x_{n+1} \approx x_n$, har vi jo at $x_n \approx g(x_n)$, og likninga er løst – tilnærma.
- e) Vi starter med likninga $\sqrt{x} = \cos x$ som også kan skrives som $x = \cos^2 x$ eller $x = \arccos \sqrt{x}$. Vi lager ei enkel **for**-løkke og tester begge formuleringene:

```
1 % Skript som forsøker å løse ei likning ved fikspunkt-iterasjon;
2 % x_{n+1}=f(x_n)
3
4 % Start-verdi
5 x=1;
6
7 for i=1:7
8     x=(cos(x))^2
9 end
```

Vi prøver:

```
>> FiksPunktFor
x =
    0.291926581726429
x =
    0.917172409958988
x =
    0.369745922832551
x =
    0.869405552848063
x =
    0.416387293352056
x =
    0.836412855210241
x =
    0.449073772577787
```

Dette ser dårlig ut. Ut fra de 7 første iterasjonene er det vanskelig å se om det konvergerer i det hele tatt. Om vi prøver med 50 iterasjoner, får vi $x = 0.66567$, med 100 iterasjoner får vi $x = 0.64466$, med 500 får vi $x = 0.64171$. Det viser seg at det går mot rett svar – men ikke fort!

Om vi prøver oss med den andre formuleringa, $x = \arccos \sqrt{x}$, derimot, går det ikke i det hele tatt; “svaret” blir komplekst.

Den andre likninga, $2x - \sqrt{x} = 4$ kan omformuleres til $x = (4 + \sqrt{x})/2$ eller til $x = (2x - 4)^2$. Vi velger $x_0 = 5$ og prøver den første varianten:

```

1  % Skript som forsøker å løse ei likning ved fikspunkt-iterasjon;
2  % x_{n+1}=f(x_n)
3
4  % Start-verdi
5  x=5;
6
7  for i=1:7
8      x=(4+sqrt(x))/2
9  end
10

```

I kommandovinduet får vi:

```

>> FiksPunktFor
x =
    3.118033988749895
x =
    2.882897784110638
x =
    2.848954914013494
x =
    2.843942372738431
x =
    2.843199616451886
x =
    2.843089499467863
x =
    2.843073172901953

```

Dette ser jo fint ut. Fullt så fint går det ikke med den andre varianten:

```

>> FiksPunktFor
x =
    36
x =
    4624
x =
    85451536
x =
    2.920785865181262e+16
x =
    3.412396028097062e+33
x =
    4.657778661029043e+67
x =
    8.677960822055000e+135

```

Selv om man her lett får inntrykk av at dette er ei rimelig “laus kanon” av en metode, er den absolutt en nyttig i visse sammenhenger. Og det finnes teori som sier noe om når metoden fungerer og når den ikke gjør det.

Ekstra-oppgave 1: Tredjegradslikninga

- a) Vi viser her hvordan det kan gjøres både som funksjonsfil og som skript. Vi starter med funksjonsfila:

```
1 function X=TredjeGradFunk(a,b,c,d)
2
3 % Funksjon som løser tredjegradslikning
4 %  $ax^3+bx^2+cx+d=0$ .
5 % Input er parametrene a, b, c, d.
6 % Funksjonen gir de tre løsningene, x_1, x_2 og x_3
7 % som output. Her har vi valgt å gi de tre
8 % løsningene som komponentene i en vektor
9
10 % Kontrollerer at det faktisk er ei tredjegradslikning
11 if a==0
12     disp('Dette er ikke ei tredjegradslikning')
13     return
14 end
15
16 % Tilordner parametre som inngår i løsningsformelen
17 u=[1 (-1+sqrt(3)*i)/2 (-1-sqrt(3)*i)/2]
18 Delta0=b^2-3*a*c
19 Delta1=2*b^3-9*a*b*c+27*a^2*d
20 % Stor C. NB! Ulik input-variabelen c.
21 C=((Delta1+sqrt(Delta1^2-4*Delta0^3))/2)^(1/3)
22
23 % Løsninga
24 X=-1/(3*a)*(b+u*C+Delta0./(u*C));
```

Her har vi valgt å la både de tre løsningene x_1 , x_2 og x_3 og de tre u -verdiene bli gitt som én vektor i stedet for tre skalarer (se linje 17 og 24 i funksjonsfila).

En tilsvarende skript-versjon kan se slik ut:

```
1 % Skript som løser tredjegradslikning
2 %  $ax^3+bx^2+cx+d=0$ .
3 % Input er parametrene a, b, c, d.
4 % Skriptet gir de tre løsningene, x_1, x_2 og x_3
5 % som output. Her har vi valgt å gi de tre
6 % løsningene som komponentene i en vektor
7
```

```

8  % Gir koeffisientene fra kommandovinduet
9  a=input('Gi a: ');
10 b=input('Gi b: ');
11 c=input('Gi c: ');
12 d=input('Gi d: ');
13
14 % Kontrollerer at det faktisk er ei tredjegradslikning
15 if a==0
16     disp('Dette er ikke ei tredjegradslikning')
17     return
18 end
19
20 % Tilordner parametre som inngår i løsningsformelen
21 u=[1 (-1+sqrt(3)*i)/2 (-1-sqrt(3)*i)/2];
22 Delta0=b^2-3*a*c;
23 Delta1=2*b^3-9*a*b*c+27*a^2*d;
24 % Stor C. NB! Ulik input-variabelen c.
25 C=((Delta1+sqrt(Delta1^2-4*Delta0^3))/2)^(1/3);
26
27 % Løsninga
28 X=-1/(3*a)*(b+u*C+Delta0./(u*C))

```

Vi tester implementeringa vår. Vi kan jo begynne med å se om den finner røttene til $(x-1)(x-2)(x-3) = x^3 - 6x^2 + 11x - 6$:

```

>> format compact
>> TredjeGradFunk(1,-6,11,-6)
ans =
    1.0000 + 0.0000i    3.0000 + 0.0000i    2.0000 - 0.0000i

```

Dette stemte jo bra.

Hva så med tredjegradspolynomet $2x^3 + x - 1$?

```

>> X=TredjeGradFunk(2,0,1,-1)
X =
    0.5898 + 0.0000i   -0.2949 - 0.8723i   -0.2949 + 0.8723i

```

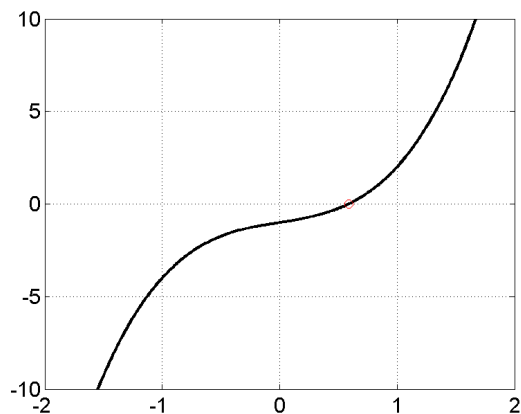
Vi har funnet ei reell løsning og to komplekse, som er de komplekskonjugerte av hverandre.

At den reelle løsninga ser ut til å stemme, ser vi av figur 1. Vi kan også kontrollere løsningene ved å sette dem inn i polynomet:

```

>> 2*X.^3+X-1
ans =
    1.0e-14 *
    0.1776 + 0.0000i    0.2220 + 0.0555i    0.2220 - 0.0555i

```



Figur 1: Tredjegradspolynomet $2x^3 + x - 1$ sammen med den ene reelle rota vi fant (tynn rød sirkel).

Vi ser at vi får (så godt som) null for alle tre løsningene.

- b) Løsninga vår skal “i prinsippet” fungere for alle mulige verdier av a, b, c og d dersom vi fra og med linje 23 i funksjonsfila over setter inn dette i stedet:

```
% Løsninga
if Delta0==0 & Delta1==0
    X=-b/(3*a);                                % Spesialtilfelle: 3 like røtter
elseif Delta0==0 & Delta1<0
    C=Delta1^(1/3);                             % Endrer C i dette tilfellet
    X=-1/(3*a)*(b+u*C+Delta0./(u*C));
else
    X=-1/(3*a)*(b+u*C+Delta0./(u*C));
end
```

Men dette viser seg å ikke fungere så godt i praksis. Om vi for eksempel setter $b = \sqrt{3}$, $a = c = 1$ og $d = -1$, får vi at $\Delta_0 = 0$ og $\Delta_1 \approx -32$. Dette tilsvarer tilfellet i `elseif`-satsen over. Men når vi tester, får vi følgende svar:

```
>> TredjeGradFunk(1,sqrt(3),1,-1)
ans =
    Inf    -Inf    Inf
```

Grunnen er at datamaskina ikke klarer å skille mellom tall som er eksakt null og tall som er veldig nær null. Litt enkelt kan man si at denne grensa går ved maskinpresisjonen, som man får opp når man skriver ‘`eps`’ i kommandovinduet. Typisk er denne av størrelsesorden 10^{-16} , men som navnet

tilsier, vil dette variere fra maskin til maskin. Man kan ta hensyn til dette på denne måten:

```
% Løsninga
if abs(Delta0)<1e-14 & abs(Delta1)<1e-14
    X=-b/(3*a); % Spesialtilfelle: 3 like røtter
elseif abs(Delta0)<1e-14 & Delta1<0
    C=Delta1^(1/3); % Endrer C i dette tilfellet
    X=-1/(3*a)*(b+u*C+Delta0./(u*C));
else
    X=-1/(3*a)*(b+u*C+Delta0./(u*C));
end
```

Her har vi sagt at dersom $|\Delta_0| < 10^{-14}$, skal vi regne den for å være null. Når vi nå tester skriptet, vil det fungere:

```
>> x=TredjeGradFunk(1,sqrt(3),1,-1)
x =
    -1.1076 - 0.9184i    0.4831 - 0.0000i    -1.1076 + 0.9184i
>> x.^3+sqrt(3)*x.^2+x-1
ans =
    1.0e-15 *
    0.2220 - 0.4441i    -0.1110 - 0.2497i    -0.4441 - 0.3331i
```

Her har vi også kontrollert løsningene; vi ser at $|x^3 + \sqrt{3}x^2 + x - 1| < 10^{-15}$ for alle de tre løsningene, så det må vel være godkjent.

Ekstra-oppgave 2: Bursdag samme dag

Ut fra hintet vi får, kan vi se at sannsynligheten for at n elever har bursdag på ulike datoer blir

$$\frac{365}{365} \cdot \frac{364}{365} \cdot \frac{363}{365} \cdot \dots \cdot \frac{365 - n + 1}{365}.$$

Spørsmålet er: Hva må n være for at dette produktet skal bli mindre enn $1/2$?

Det kan vi finne ut på flere måter. En måte kan være å lage et lite skript med ei `while`-løkke der vi for hver iterasjon multipliserer med neste faktor helt til produktet bikker $1/2$. Dette kan gjøres slik:

```
1 % Skript som regner ut hvor mange elever det må være i en
2 % skoleklasse for at der mest sannsynlig vil være minst to
3 % elever med bursdag på samme dag
4
5 % Setter sannsynligheten P til én
6 P=1;
7 % Setter antall elever til null
```

```

8  elever=0;
9
10 while P>1/2
11     elever=elever+1;           % Øker antall elever med 1
12     P=P*(365-elever+1)/365;   % Oppdaterer sannsynlighet
13 end
14
15 % Skriver resultatet til skjerm
16 elever

```

Vi kjører skriptet, som vi har kalt `SammeBursdag.m`, i kommandovinduet:

```

>> SammeBursdag
elever =
    23

```

Med andre ord må vi regne med det i en skoleklasse med 23 eller flere elever vil være elever med samme fødselsdag.