Matematikk 1000

Øvingsoppgaver i numerikk – leksjon 6
Løsningsforslag

Oppgave 1 – Tredjegradslikninga

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

```
function X=TredjeGradFunk(a,b,c,d)

% Funksjon som løser tredjegradslikning
% ax^3+bx^2+cx+d=0.
% Input er parametrene a, b, c, d.
% Funksjonen gir de tre løsningene, x_1, x_2 og x_3
% som output. Her har vi valgt å gi de tre
% løsningene som komponentene i en vektor

% Kontrollerer at det faktisk ér en tredjegradslikning
if a==0
   disp('Dette er ikke ei tredjegradslikning')
   return
end

% Tilordner parametre som inngår i løsningsformelen
u=[1 (-1+sqrt(3)*i)/2 (-1-sqrt(3)*i)/2]
Delta0=b^2-3*a*c
Delta1=2*b^3-9*a*b*c+27*a^2*d
% Stor C. NB! Ulik input-variabelen c.
C=((Delta1+sqrt(Delta1^2-4*Delta0^3))/2)^(1/3)

% Løsninga
X=-1/(3*a)*(b+u*C+Delta0./(u*C));
```

En tilsvarende skript-versjon kan se slik ut:

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

% Gir koeffisientene fra kommandovinduet
a=input('Gi a: ');
b=input('Gi b: ');
c=input('Gi c: ');
d=input('Gi d: ');

% Kontrollerer at det faktisk ér en tredjegradslikning
if a==0
   disp('Dette er ikke ei tredjegradslikning')
   return
end

% Tilordner parametre som inngår i løsningsformelen
u=[1 (-1+sqrt(3)*i)/2 (-1-sqrt(3)*i)/2];
Delta0=b^2-3*a*c;
Delta1=2*b^3-9*a*b*c+27*a^2*d;
% Stor C. NB! Ulik input-variabelen c.
C=((Delta1+sqrt(Delta1^2-4*Delta0^3))/2)^(1/3);

% Løsninga
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 $(x-1)(x-2)(x-3) = x^3 - 6x^2 + 11x - 6$:

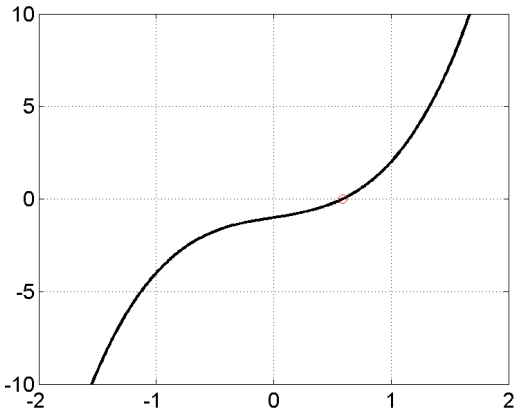
```
>> format compact
>> TredjeGradFunk(1,-6,11,-6)
ans =
   1.0000 + 0.0000i   3.0000 + 0.0000i   2.0000 - 0.0000i
```

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 én reell løsning og to komplekse som er de komplkskonjugerte av hverandre.



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

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
```

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.

Oppgave 2 – Halveringsmetoden – igjen

a) I skriptet vårt fra leksjon 5 skal altså linje 13 erstattes med "`while abs(b-a)>1e-3`". Når vi gjør det, ser skriptet slik ut:

```
1  % Implementering av halveringsmetoden for likninga
2  % sqrt(x)-cos(x)=0 med a=0 og b=pi/2 som start-grenser
3
4  % Grenser
5  a=0;
6  b=pi/2;
7
```

```
8  % Funksjonsverdier
9  Fa=sqrt(a)-cos(a);
10 Fb=sqrt(b)-cos(b);
11
12 % Starter for-løkke som kjøres 10 ganger
13 while abs(b-a)>1e-3
14   c=(a+b)/2;         % Midtpunktet
15   Fc=sqrt(c)-cos(c); % Funksjonsverdi i midtpunktet
16   if Fa*Fc<0
17     b=c;             % Setter ny b til c
18   else
19     a=c;             % Setter ny a til c
20   end
21 end
22
23 % Regner ut nytt midtpunkt og skriver svaret til skjerm
24 x=(a+b)/2
```

Vi kjører skriptet:

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

Vi fikk et noe annet svar enn det vi fikk i leksjon 5; 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 løsninga 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, 10 og 15. I `while`-linja setter vi nøyaktigheten til $2 \cdot 10^{-5}$. Med disse endringene blir skriptet vårt seende slik ut:

```
% Implementering av halveringsmetoden for likninga
% 2x-sqrt(x)-4=0 med a=0 og b=4 som start-grenser

% Grenser
a=0;
b=4;

% Funksjonsverdier
Fa=2*a-sqrt(a)-4;
Fb=2*b-sqrt(b)-4;

while abs(b-a)>2e-5
  c=(a+b)/2;          % Midtpunktet
  Fc=2*c-sqrt(c)-4; % Funksjonsverdi i midtpunktet
  if Fa*Fc<0
    b=c;              % Setter ny b til c
  else
    a=c;              % Setter ny a til c
  end
end

% Regner ut nytt midtpunkt og skriver svaret til skjerm
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 lar $u = \sqrt{x}$, slik at $x = u^2$, får vi

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

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) Vi lager ei funksjonsfil som implementerer $f(x)$. Vi kan godt velge å bruke den siste funksjonen, $2x - \sqrt{x} - 4$:

```
1  function f=FunkTilIntHalv(x)
2
3  % Funksjon som vi skal finne nullpunktet til
4  % Blir brukt av skriptet Halvering.m
5
6  %f=sqrt(x)-cos(x);
7  f=2*x-sqrt(x)-4;
```

Her ser vi også at vi har "gamle-funsjonen", $f(x) = \sqrt{x} - \cos(x)$ på lur; det er fort gjort å kommentere inn denne og ut den andre. (Ta bort prosenttegnet i linje 6 og sett det inn i linje 7.) Vi kan nå referere til denne funksjonen i halveringsmetode-skriptet vårt (linje 13, 14 og 18):

```
1  % Implementering av halveringsmetoden for likninga
2  % FunkTilIntHalv(x)=0, der funksjonen er gitt i ei eiga
3  % Funksjonfil. a og b er start-grensene for metoden.
4
5  % Grenser
6  a=0;
7  b=4;
8
9  % Presisjon
10 Pres=2e-5;
11
12 % Funksjonsverdier
13 Fa=FunkTilIntHalv(a);
14 Fb=FunkTilIntHalv(b);
15
16 while abs(b-a)>Pres
17   c=(a+b)/2;          % Midtpunktet
18   Fc=FunkTilIntHalv(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
```

På den måten er det enklere å oppdatere skriptet når vi skal løse andre likninger senere. Her har vi også definert presisjonen, `Pres`, i ei eiga linje (linje 10). Variabelen `Pres` dukker opp igjen i `while`-linja (linje 16).

d) 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:

```
% Implementering av halveringsmetoden for likninga
% FunkTilIntHalv(x)=0, der funksjonen er gitt i ei eiga
% Funksjonfil. a og b er start-grensene for metoden.

% Grenser
a=0;
b=4;

% Presisjon
Pres=2e-5;

% Teller antall iterasjoner
teller=0;

% Funksjonsverdier
Fa=FunkTilIntHalv(a);
Fb=FunkTilIntHalv(b);

while abs(b-a)>Pres
  c=(a+b)/2;         % Midtpunktet
  Fc=FunkTilIntHalv(c); % Funksjonsverdi i midtpunktet
  if Fa*Fc<0
    b=c;             % Setter ny b til c
  else
    a=c;             % Setter ny a til c
  end
  teller=teller+1;
end

% Regner ut nytt midtpunkt og skriver svaret til skjerm
x=(a+b)/2
% Skriver antall iterasjoner til skjerm
Iterasjoner=teller
```

Vi tester at det fungerer:

```
>> Halvering
x =
   2.843070983886719
Iterasjoner =
    19
```

Vi måtte iterere 19 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) 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:

```
% Skript som implementerer Newtons metode

% Bestemmer x_0
x=1;

for i=1:5
    x=x-(sqrt(x)-cos(x))/(1/(2*sqrt(x))+sin(x)) % Iterasjonsformel
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 fårhånd – 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;
```

```
% Preisjon
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 uten å måtte se utregninga for hver iterasjone:

```
>> 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.

b) 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  % Preisjon
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; vi sier at den *konvergerer* raskere. 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 nullpuktet skal ligge mellom a og b.

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

c) 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 iterasoner:

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

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

d) 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:

```
% Skript som forsøker å løse ei likning ved fikspunkt-iterasjon;
% x_{n+1}=f(x_n)

% Start-verdi
x=1;

for i=1:7
    x=(cos(x))^2
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:

```
% Skript som forsøker å løse ei likning ved fikspunkt-iterasjon;
% x_{n+1}=f(x_n)

% 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 intrykk av at dette er ei rimelig "laus kanon" av en metode, er det absolutt en nyttig metode i visse sammenhenger.

Ekstra-oppgave: Bursdag samme dag

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

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

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:

```
% Skript som regner ut hvor mange elever det må være i en
% skoleklasse for at der mest sannsynlig vil være minst to
% elever med bursdag på samme dag

% Setter sannsynligheten P til én
P=1;
% Setter antall elever til null
elever=0;

while P>1/2
  elever=elever+1;              % Øker antall elever med 1
  P=P*(365-elever+1)/365;       % Oppdaterer sannsynlighet
end

% Skriver resultatet til skjerm
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.