Sisältö

0.1. Epälineaarisista yhtälöistä ja kiintopisteiteraatiosta

0.1.1. Bisektio, välin puolitus, binäärihaku, sataosaa

2.3.2000 Tänne on siirretty tiedostosta nummenet.tex kohdasta sectionEpälineaariset yhtälöt

[1, ss. 39–94]

bisekt.m on kurssihakemistossa (toivottavasti), implementoi!

Projektityössä on mainittu teht. 15 "sataosaa", yleisemmin n_osaa . (Ei ole synonyymi ilmaisulle "en osaa"(edellisestä puuttuu aspiraatio).)

Formuloi ja todista "sataosaalause", "n_osaa-lause"(ilman aspiraatiota).

Nämä menetelmät rajaavat (sulkevat) 0-kohdan ("bracket a zero").

Johdattelua

Epälineaarisia yhtälöitä esiintyy kaikkialla ... [1, ss. 39–40] alkaa väestökasvumalliin liittyvällä esimerkillä.

0.1.2. Välin puolitus, bisektio, dekasekstio, satasektio

Ensimmäinen reaalifunktion nollakohdanhakumenetelmä nykypäivinä on kuvan piirto ja zoomaus. Jospa haluaisimme etsiä ratkaisua yhtälölle.

$$e^x - 1.5 - \arctan x = 0$$

MATLAB:ssa toimisimme tähän tapaan, kokeillen ensin sopivaa x-väliä. (Huomaa, että versiosta 5.3 alkaen voi pikkufunktioita määritellä suoraan istunnossa `inline`- tyylillä, toki voi aina kirjoittaa määrityksen m-tiedostoon, kuten ennekin.) Kuvaikkunaa zoomaamalla ja nollakohtaa hiiren vasemmalla napsuttelemalla pääsemme nopeasti arvioon $x_0 \approx 0.7676$. Tätä tarkempaan ei grafiikkaikkunan asteikkonumerointi yllä.

```
>> f=inline('exp(x)-1.5-atan(x)','x')
>> fplot(f,[-1.4,1.4]);grid;shg
>> fplot(f,[0.7676,0.7677]);grid;shg   % Tämä ei anna lisätietoa, kun asteikossa ei
                                       % ole enempää numeroita.
```

Samanlaiseen tarkkuuteen päästään vastaavassa MAPLE-istunnossa tähän tapaan (hieman hitaampaa tahtia, mutta hyvin reaaliajassa).

```
f:=x->exp(x)-1.5-arctan(x);
plot(f,-1.4..1.4);plot(f,0.7..0.8);plot(f,0.767..0.768);
```

Epävarmuusvälin kutistaminen

Klassinen algoritmi on bisektio, joka perustuu jatkuvien funktioiden väliarvolauseeseen, joka myös *Bolzanon lauseen* nimellä tunnetaan. *Jos jatkuvalla funktiolla on välin päätepisteessä erimerkkiset arvot, sillä on nollakohta ao. välillä.*

Alkuperäinen väli olkoon $[a, b]$ ja f vaihtakoon merkkiä. Bisektioalgoritmi voidaan kuvata näin:

Bisektioalgoritmi

1. $\alpha := a$, $\beta := b$, Laske $f_\alpha := f(\alpha)$, $f_\beta := f(\beta)$
2. $\gamma := \frac{1}{2}(\alpha + \beta)$. Laske $f_\gamma := f(\gamma)$
3. Jos $\text{sign}(f_\gamma) = \text{sign}(f_\alpha)$ niin $\alpha := \gamma$, muuten $\beta := \gamma$
4. Jos $\beta - \alpha > tol$, palaa kohtaan 2.

Tässä *tol* tarkoittaa virhetoleranssia.

MATLABilla voitaisiin edellä olevan funktion f nollakohdan haku toteuttaa vaikka näin:

```
clear
f=inline('exp(x)-1.5-atan(x)','x')  % Määritellään (taas sama) funktio.

a(1)=-1.4;b(1)=5;                    % Alkuperäinen väli

for i=1:5                            % Tehdään viisi välin puolitusta.
fa=f(a(i));fb=f(b(i));
  x0=(a(i)+b(i))/2;
  f0=f(x0);
   if (sign(fa) == sign(f0))
     a(i+1)=x0; b(i+1)=b(i); fa=f0;
   elseif (sign(fb) == sign(f0))
     a(i+1)=a(i); b(i+1)=x0; fb=f0;
   end;
 end;
```

Estimme kaikki tulostukset. Katsotaan nyt, mitä saatiin. Havainnollista on laittaa a-pisteiden sarake ja b-pisteiden sarake vierekkäin 2-sarakkeiseksi matriisiksi: Välien pituudet antavat maksimivirheen kullakin iteraatiokierroksella.

** Seuraava kommentoitu vasta bisekt-funktion käytön yhteyteen.

```
format long          % Tulostustarkkuus maksimiarvoonsa
ab=[a' b']           % Transponoidaan a-vektori ja b-vektori ja laitetaan vierekkäin.
                     % näin nähdään havainnollisesti algoritmin eteneminen.
virheet=b-a          % Muodostetaan välien pituudet.
format short
[(1:6)' a' b']       % Laitetaan iteraatiokierrosnumero 1. sarakkeeksi.
ab =
  -1.40000000000000   5.00000000000000
  -1.40000000000000   1.80000000000000
   0.20000000000000   1.80000000000000
   0.20000000000000   1.00000000000000
   0.60000000000000   1.00000000000000
   0.60000000000000   0.80000000000000
virheet =
  Columns 1 through 4
```

```
   6.40000000000000   3.20000000000000   1.60000000000000   0.80000000000000
  Columns 5 through 6
   0.40000000000000   0.20000000000000
ans =
    1.0000   -1.4000    5.0000
    2.0000   -1.4000    1.8000
    3.0000    0.2000    1.8000
    4.0000    0.2000    1.0000
    5.0000    0.6000    1.0000
    6.0000    0.6000    0.8000
```

Kokoamme sitten edellisen istuntotiedoston funktioksi tiedostoon `bisekt.m`

```
function valit=bisekt(fun,vali,tol,nmax)
% Funktio paluttaa 2-sarakkeisen matriisin, josta näkyy epävarmuusvälin kehitys.
% Tarkoitus: Opetus ja havainnollistus.
% Kutsuesim:
%  1. % ab=bisekt('sin',[-1,1])   % oletustoleranssi ja nmax
%  2. % ab=bisekt(f,[-2,5],0.00001) % Oletus-nmax
%  3. % ab=bisekt(f,[-2,5],0.00001,100)

a(1)=vali(1);b(1)=vali(2);

if nargin < 4
   nmax=1000;
end;

if nargin==2
   tol=eps;
end;
i=1;
fa=feval(fun,a(i));fb=feval(fun,b(i));


if sign(fa) == sign(fb)
   error('Funktio ei saa erimerkkisiä arvoja päätepisteissä')
end;

while ( (b(i)-a(i) > tol) & (i < nmax ) )
  x0=(a(i)+b(i))/2;
  f0=feval(fun,x0);

  if (abs(f0)<eps)           % Välin keskellä oleva 0-kohtamahdollisuus
       a(i)=x0;b(i)=x0;      % otettava huomioon.  (esim. 1 yllä)
       valit=[a' b']; return; % Johtuu sign-testistä, 0 on aina erimerkkinen kuin ei-nolla
  end;                        % Ehkä vähäeleisemmin else-haaraksi alla

   if ( sign(fa) == sign(f0)  )
     a(i+1)=x0; b(i+1)=b(i);fa=f0;
```

```
   elseif  ( sign(fb) == sign(f0) )
     a(i+1)=a(i); b(i+1)=x0;fb=f0;
   end;
   i=i+1;
end;
valit=[a' b']; % Transponoidaan a-vektori ja b-vektori ja laitetaan vierekkäin.
               % näin nähdään havainnollisesti algoritmin eteneminen.
```

Testiesimerkkinä katsomme viimeistä kommenttiesimerkkiä. Rajoitamme iteraatioiden lukumäärän kymmeneen tilan säästämiseksi.

```
format long                         % Tulostustarkkuus maksimiarvoonsa
f=inline('exp(x)-1.5-atan(x)','x')
ab=bisekt(f,[-2,5],0.00001,10)
format short                        % Takaisin oletusarvoon
virheet=diff(ab')                   % Muodostetaan välien pituudet.
[(1:10)' ab]                        % Laitetaan iteraatiokierrosnumero 1. sarakkeeksi.
ab =
  -2.00000000000000   5.00000000000000
  -2.00000000000000   1.50000000000000
  -0.25000000000000   1.50000000000000
   0.62500000000000   1.50000000000000
   0.62500000000000   1.06250000000000
   0.62500000000000   0.84375000000000
   0.73437500000000   0.84375000000000
   0.73437500000000   0.78906250000000
   0.76171875000000   0.78906250000000
   0.76171875000000   0.77539062500000
virheet =
  Columns 1 through 7
    7.0000    3.5000    1.7500    0.8750    0.4375    0.2188    0.1094
  Columns 8 through 10
    0.0547    0.0273    0.0137
ans =
    1.0000   -2.0000    5.0000
    2.0000   -2.0000    1.5000
    3.0000   -0.2500    1.5000
    4.0000    0.6250    1.5000
    5.0000    0.6250    1.0625
    6.0000    0.6250    0.8438
    7.0000    0.7344    0.8438
    8.0000    0.7344    0.7891
    9.0000    0.7617    0.7891
   10.0000    0.7617    0.7754
```

Jako sataan osaan

MATLAB opettaa ajattelemaan vektoroidusti ja rinnakkaisesti. Algoritmin tehokkuuden kannalta toisistaan riippumattomat operaatiot ovat eri asemassa kuin ne, joissa seuraava syöte riippuu edellisen laskennan tuloksesta. Matlab emuloi vektoriarkkitehtuuria sillä, että vektorioperaatiot ovat verrattoman nopeita.

Siten sata toisistaan riippumatonta funktion arvon laskentaa ei vaadi 100-kertaista aikaa verrattuna yhteen.

```
>> f=inline('exp(x)-1.5-atan(x)','x')
>> x=1;tic; f(x); t1=toc;
>>   x=linspace(-1,1,100);tic; f(x); t100=toc;
>>   x=linspace(-1,1,1000);tic; f(x); t1000=toc
>>   vektoriajat=[t1,t100,t1000]

vektoriajat =

    0.0009    0.0024    0.0024
>>   skalaariajat=[t1,100*t1,1000*t1]

skalaariajat =
    0.0009    0.0939    0.9390

>>    skalaariajat./vektoriajat

Kannattaa vielä katsoa yllä olevia. Ajat vaihtelevat, satunnaisuutta esiintyy.
Tässä on dramaattinen vertailu vektorilaskennan ja for-silmukan välillä.
ans =
    1.0000   39.8050  396.5372

» x=linspace(-1,1,10000);tic; f(x); toc
elapsed_time =
   0.05000000000000
» tic; for k=1:10000; f(x(k)); end; toc
elapsed_time =
   5.50000000000000
```

Nähdään, että tässä tapauksessa laskettaessa 100 kertaa funktion arvo skalaarisesti vaatii n. 40-kertaisen ajan verrattuna 100:aan saman funktion arvon laskentaan vektoroidustí. Karkeasti ottaen siis 100:n funktion arvon toisistaan riippumaton laskenta vie vain n. $2-3$- kertaisen ajan verrattuna yhden funktion arvon laskentaan!

```
>> t100/t1
ans =
    2.5122
```

*** Tässä on jotain yliluonollista ainakin t1000:n kohdalla, pitää testailla tarkemmin ***

Toki tämä on jossain määrin kone- ja tapausriippuvaista. Tilanne on vieläkin dramaattisempi silloin, kun MATLABia käytetään vektori- tai rinnakkaiskoneessa.

Moneen osaan jako samanaikaisesti

Katsotaan malliksi vaikka sin-funktion 0-kohdan hakua välillä [-1,1]. Jaetaan aluksi 10:een osaan.

```
a=-5; b=5;
x=linspace(a,b,10);
y=sin(x)
I=find(sign(y(1)) ~= sign(y))  % Etsimme indeksejä, joita vastaavilla y-arvoilla on eri
                               % merkki kuin y(1):llä.
y =

  Columns 1 through 7

    0.9589    0.6797   -0.3558   -0.9954   -0.5274    0.5274    0.9954

  Columns 8 through 10

    0.3558   -0.6797   -0.9589


I =

     3     4     5     9    10
```

`y(3)` on ensimmäinen, jolla merkki vaihtuu, joten nollakohta on välillä `[y(2),y(3)]`. Jos kiinnitämme huomiomme ensimmäiseen (pienimpään) nollakohtaan, saamme sitä rajaavan välin ottamalla `Ia=I(1)-1`, `Ib=I(1)`. Siis näin:

```
» Ia=I(1)-1; Ib=I(1); [Ia Ib]

ans =

     2     3

» a=x(Ia);b=x(Ib);[a,b]

ans =

   -3.8889   -2.7778
```

Nyt voidaan toteuttaa vuorovaikutteinen $\boxed{\uparrow}$-iteraatio toistamalla nuolinäppäimen avulla jälkimmäistä riviä. Voidaan samantien jakaa 100:aan osaan, jolloin saadaan kaksi oikeaa desimaalia joka kierroksella.

```
a=0; b=5
x=linspace(a,b);y=sin(x);I=find(sign(y(1))~=sign(y));Ib=I(1);a=x(Ib-1),b=x(Ib)
```

Kolmen näpäyksen jälkeen saamme lyhyellä tarkkuudella samat, eli 3.1416. Tässä vaiheessa on syytä komentaa `format long`. Viisi lisänäpäystä antaa molemmille arvon 3.14159265358979. Sattumoisin samat numerot saamme kirjoittamalla

```
pi
```

Houkutus olisi suuri ryhtyä kehittämään algoritmia, joka etsii samanaikaisesti kaikkia merkinvaihtokohtia.

Monen nollakohdan etsintä samanaikaisesti

Emme pysty tuota houkutusta vastustamaan. Ideoidaanpa hiukan.

```
» bv=sign(y(1)) ~= sign(y)

bv =

     0     0     1     1     1     0     0     0     1     1
```

Kohdissa, joissa on peräkkäin 0 1 tai 1 0, tapahtuu merkin vaihto. Tässä tapauksessa haluaisimme poimia välit `x([2 3]), x([5 6]), x([8 9])` .

Entäpä, jos muodostamme `diff`-funktiolla erotukset, siis vektorin $(bv_{i+1} - bv_i)_{i=1}^{n-1}$. Tällöin saamme muutoskohdissa ± 1 ja muissa 0. Hieno idea, kokeillaanpa!

```
» diff(bv)

ans =

     0     1     0     0    -1     0     0     1     0
```

Toden totta! Kun tästä jatkamme ideointia, päädymme nopeasti havaintoon, että muutoskohdan vasemmat pisteet ("a-indeksit) ja oikeat ("b-indeksit) saadaan lausekkeilla

```
vasen=find(abs(diff(sign(y(1)) ~= sign(y))))
oikea=vasen+1
```

Nyt voimme kokeilla:

```
format compact
a=-5; b=5;
x=linspace(a,b,10);
y=sin(x);
vasen=find(abs(diff(sign(y(1)) ~= sign(y))))
oikea=vasen+1
a=x(vasen)
b=x(oikea)
valit=[a' b']

vasen =
     2     5     8
oikea =
     3     6     9
a =
   -3.8889   -0.5556    2.7778
b =
```

```
   -2.7778    0.5556    3.8889
valit =
   -3.8889   -2.7778
   -0.5556    0.5556
    2.7778    3.8889
```

Nyt sitten tositoimiin rohkeasti,

```
format compact
a=-10; b=10;
x=linspace(a,b,1000);
y=sin(x);
vasen=find(abs(diff(sign(y(1)) ~= sign(y))));
oikea=vasen+1;
a=x(vasen);
b=x(oikea);
valit=[a' b']
valit =
   -9.4394   -9.4194
   -6.2963   -6.2763
   -3.1532   -3.1331
   -0.0100    0.0100
    3.1331    3.1532
    6.2763    6.2963
    9.4194    9.4394
» x0=(sum(valit'))/2
x0 =
   -9.4294   -6.2863   -3.1431         0    3.1431    6.2863    9.4294
```

Vaikuttavaa! Yhdellä askeleella (1000 riippumatonta funktion arvon laskentaa) saimme hyvät rajausvälit kullekin nollakohdalle. Laskimme vielä välien keskipistevektorin (`valit'` on matriisin transpoosi). Niistä voitaisiin jatkaa vektori-Newtonilla tai vastaavalla suoraan ja niin teemmekin hieman tuonnempana.

Niin, tarvitaanko nyt enää hienompia algoritmeja, kun tällä yksinkertaisella tehostuksella päästään vaikuttaviin tuloksiin. Vastaus riippuu käyttötarkoituksesta. Laajasti ottaen varmasti tarvitaan, usein kyseessä on osatehtävä ... Tässä suppeneminen on joka tapauksessa vain lineaarista, vaikka oikeita numeroita saadaan monta kerrallaan.

0.1.3. Iteratiivisten algoritmien lopetusehdoista

Ajatellaan, että jollain yhtälönratkaisualgoritmilla muodostetaan jono $(x_0, x_1, x_2, \ldots)$. Milloin lopetetaan?

Ehtoja:

$$|x_N - x_{N-1}| < \varepsilon_a \tag{1}$$

$$\frac{|x_N - x_{N-1}|}{|x_N|} < \varepsilon_r \tag{2}$$

$$|f(x_N)| < \varepsilon_f \tag{3}$$

Ehdot 1 ja 2 testaavat absoluuttista ja suhteellista virhettä vastaavasti ja 3 mittaa funktion arvon pienuutta.

Tärkein on yleensä 2, mutta nämä on harkittava tilanteen mukaan. On huomattava, että 3-ehdon takaama funktion arvon pienuus ei välttämättä takaa sitä, että ollaan lähellä oikeaa nollaktaa. Ajatellaanpa vaikka funktiota $f(x) = x^{10}$, jolla pistessa $x = 0.5$ on arvo 2^{-10}. Asiaa voidaan dramatisoida nostamalla potenssia.

0.1.4. Kiintopisteiteraatio

Tämä soveltuu esim. Newtonin menetelmän analysointiin.

Funktion f kiintopiste on luku p, jonka funktio pitää paikallaan, ts. $f(p) = p$. Kts. myös HAM.

Kiintopisteiteraatiot tulevat vastaan monissa yhteyksissä matematiikassa. Katsomme niitä tässä yhtälöiden ratkaisukeinoina. (Palataan myöhemmin ...)

Kiintopisteen etsiminen voidaan muuntaa funktion nollakohdan hakemiseksi ja kääntäen.

$f(x) = 0 \Longleftrightarrow g(x) = x$, kun määritellään $g(x) = f(x) + x$. Huom: apufunktio g voidaan valita mitä moninaisimmilla tavoilla, kuten vaikkapa $g(x) = 10f(x) + x$.

Kääntäen, jos tehtävämme on etsiä funktion f kiintopistettä, niin

$f(p) = p \Longleftrightarrow g(p) = 0$, kun määrittelemme $g(x) = f(x) - x$.

Kiintopistemuodossa on helpompi analysoida juurenhakuproblematiikkaa.

Huom! Juurenhaku voidaan muuntaa mitä moninaisimmilla tavoilla yhtälön ratkaisuksi. Toiset tavat voivat olla täysin kelvottomia, toiset taas loistokkaita.

Esim: $f(x) = 0$

a) $g(x) = x - f(x)$

b) $g(x) = x + 3f(x)$

c) $g(x) = x - \frac{f(x)}{f'(x)}$

Huom! Jos f on jatkuva ja jos iteraatiolla muodostettu jono $x_n = f(x_{n-1})$ suppenee, niin $p = \lim_{n\infty} x_n$ on f:n kiintopiste, sillä toisaalta

$$\lim_{n\to\infty} x_n = p$$

ja toisaalta

$$\lim_{n\to\infty} x_n = \lim_{n\to\infty} f(x_{n-1}) = f(\lim_{n\to\infty} x_{n-1}) = f(p)$$

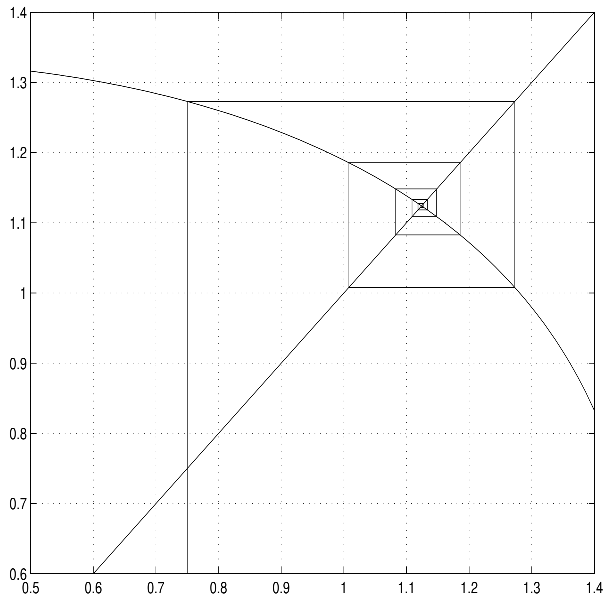
funktion f jatkuvuuden takia.

Lause 0.1.1. (Kiintopistelause) *Olkoon $g \in C[a,b]$ ja $g(x) \in [a,b] \forall x \in [a,b]$, ts. g kuvaa välin $[a,b]$ sille itselleen. Tällöin g:llä on kiintopiste välillä $[a,b]$. Jos lisäksi oletetaan, että g on derivoituva ja*

$$|g'(x)| \leq k \quad \forall x \in [a,b],$$

missä $k < 1$, niin kiintopiste on yksikäsitteinen ja iteraatiojono $x_n = g(x_{n-1}), n = 0, 1, \ldots$ suppenee mielivaltaisella alkupisteen x_0 valinnalla, $x_0 \in [a,b]$.

Tod:



Kuva 1:

(1) Olemassolo hoituu Bolzanon lauseella

Jos $g(a) = a$ tai $g(b) = b$, on kiintopiste löytynyt. Oletetaan, että näin ei ole, eli $g(a) \neq a$ ja $g(b) \neq b$. Oletuksemme mukaan täytyy silloin olla $g(a) > a$ ja $g(b) < b$ (muutenhan jompikumpi päätepiste kuvautuisi välin ulkopuolelle).

Määritellään apufunktio $h(x) = g(x) - x$. No nythän sitten $g(a) > 0$ ja $g(b) < 0$, joten Bolzano $\Rightarrow \exists p \in [a, b]$ *siten, että $h(p) = 0$, eli $f(p) = p$.*

(2) Yksikäsitteisyys seuraa suoraan väliarvolauseesta:

Jos olisi kaksi eri kiintopistettä $p \neq q$ välillä $[a, b]$, niin

$$p - q = f(p) - f(q) = f'(\xi)(p - q),$$

joten saataisiin ristiriitainen epäyhtälö

$$|p - q| < k|p - q| < |p - q|.$$

(3) Suppeneminen seuraa epäyhtälöketjusta:

$|x_n - p| = |g(x_{n-1}) - g(p)| = |g'(\xi)||x_{n-1} - p| \leq k|x_{n-1} - p| \leq \ldots \leq k^n|x_0 - p|$.

Koska $0 < k < 1$, niin $\lim_{n\to\infty} x_n = p$.

□

Esim.Luentoesimerkki. Yksikäsitteinen kiinopiste voi toki olla ilmankin lauseen ehtoja, se lienee tuiki ilmeistä ilmankin esimerkkiä, otetaan vaan jokin jyrkästi nouseva funktio, kuten e^x.

Lause 0.1.2. (Virhearvio) *Jos g toteuttaa kiintopistelauseen ehdot, niin*

$$|x_n - p| \leq k^n \max\{x_0 - a, b - x_0\} \tag{4}$$

$$|x_n - p| \leq \frac{k^n}{1-k}|x_0 - x_1| \quad \forall n \geq 1. \tag{5}$$

Tod: *Samanhenkisesti kuin edellä, kts. [1, s. 53]*

□

Stabiilisuus, attraktiivisuus

Usein iteraatiojono ajatellaan dynaamiseksi systeemiksi ja x-muuttuja ajaksi.

Määritelmä 0.1.3. *Kiintopistettä p sanotaan* attraktiiviseksi *tai* stabiiliksi, *jos on olemassa sellainen p:n ympäristö $U_\varepsilon(p)$, että $x_0 \in U_\varepsilon(p) \Rightarrow x_n \to p$.*

Vastakohta on repulsiivinen *eli* epästabiili. *Valittiinpa tällöin miten pieni ympäristö tahansa, niin aina löytyy alkupisteitä, joista lähdettäessä iteraatiojono ei suppene kohti p:tä.*

□

Olkoon p funktion g kiintopiste. On helppo nähdä, että

- *Jos $g'(p) < 1$, niin p on attraktiivinen.*
- *Jos $g'(p) > 1$, niin p on repulsiivinen.*

Jos $g'(p) = 1$, niin kumpi tahansa on mahdollinen.

Kts. esim. [1, teht. 22, s. 56]

Tehtäviä

1. *BF s. 92 teht. 17: Vuonna 1224 Leonardo Pisalainen, paremmin tunnettu Fibonaccina vastasi ...*
   $x^3 + 2x^2 + 10x = 20$

   $$1 + 22\frac{1}{60} + 7(\frac{1}{60})^2 + 42(\frac{1}{60})^3 + 33(\frac{1}{60})^4 + 4(\frac{1}{60})^5 + 40(\frac{1}{60})^6$$

   Kuinka tarkka oli hänen approksimaationsa?

2. *NMS-kirja s. 260: Cold snap.*

3. *BF s. 56 teht. 21 Putoava esine (kuten vaikkapa hampurilainen) toteuttaa:*

0.1.5. Newton-Raphson, sekantti, Regula Falsii

Olemme jo luennolla nähneet: Newton suppenee kvadraattisesti kohti yksinkertaista 0-kohtaa, jos alkuarvo on riittävän hyvä. (Todistus ei ollut ihan kunnollinen sikäli, että se sanoi oikeastaan vain, että jos jono suppenee, niin se suppenee kvadraattiseti.) Kiintopistelauseen avulla saadaan tyydyttävämmin ja tarkemmin.

Palataan siihen hieman myöhemmin.

0.1.6. Vektori-Newton

```
function [f,df]=funder(x)
f=sin(x); df=cos(x);  % Tämä rivi vaihtuva, editoidaan ongelmakohtaiseti
```

Newton-askel

```
x0=...; x=x0;
[fx,dfx]=funder(x); x=x-fx./dfx   % Iteroidaan nuolinäppäimellä
                                  %  tai:
N=10;
for i=1:N
[fx,dfx]=funder(x); x=x-fx./dfx
end;
```

Kun käytimme "pistejakoa"(./) , voimme antaa koko joukon alkupisteitä samantien. Siivotaan samaa kohti (ilmeisesti) suppenevat pois. Iteroidaan joitakin kierroksia ja siivotaan:

```
function [f,df]=funder(x)
f=sin(x); df=cos(x);  % Tämä rivi vaihtuva ongelmakohtaiseti

N=5;kynnys=0.5;
for i=1:N
[fx,dfx]=funder(x); x=x-fx./dfx ; x=x(diff(x)>kynnys)
end;
```

Tässä muodossa ei ihan vielä toimi. Järjestys ei välttämättä säily Newtonin iteraatiossa. On siten syytä järjestää välillä. Tosin riittänee tehdä se kerran, eikä sinänsä ole fataalia, jos ei tehdäkään. Sensijaan fataalia algoritmille on, se että diff(x) on yhtä lyhyempi kuin x. Tällöin viimeinen alkio jää varmasti poimimatta, joten x-vektorista putoaa joka kierroksella (mahd. väärä) alkio pois. Siksi lisäämme poimintabittivektorin loppuun 1:n, joka on kaiken kukkuraksi konvertoitava logical-tyyppiseksi. Suoritetaan siten kaksivaiheinen iterointi. (Montako kummassakin, tässä kumpaakin 3.) Lisätään tuo `sort`.

```
N=3;kynnys=0.5;x=-10:10;
for i=1:N
[fx,dfx]=funder(x); x=x-fx./dfx
end;
% Ehkä olisi sopivaa tehdä yksi sort tässä välissä:
% x=sort(x);
% Alla teemme sen suhteen tuhlaillen.
for i=1:N
[fx,dfx]=funder(x); x=x-fx./dfx;x=sort(x), x=x([diff(x)>kynnys,logical(1)])
end;
```

0.1.7.

0.1.8.

Luku 1

Usean muuttujan funktiokäsite ja graafinen kuvaaminen

Mitä tarkoittaa funktio, noin niinkuin yleensä?

Mieti:

- *Yhden muuttujan funktio*
- *Äärellisessä joukossa määritelty funktio (esim. permutaatio)*
- *Matriisin avulla määritelty funktio (lineaarikuvaus)*
- *Tietokoneohjelman määrittelemä funktoi (aliohjelma, proseduuri)*
- *Usean reaalimuuttujan funktio*

Esim.Ympyräpohjaisen lieriön tilavuus voidaan laskea kaavalla $V = \pi r^2 h$, missä r on pohjan säde ja h lieriön korkeus. Voimme hyvällä syyllä sanoa, että tilavuus on kahden muuttujan, r ja h funktio:

$$f(r, h) = \pi r^2 h$$

Olennaista funktiolle on *niin tässä kuin kaikissa muissakin yhteyksissä, että on annettu* sääntö, *jolla syötteeksi annetuista argumenteista voidaan laskea yksikäsitteinen tulos. Toisin sanoen, funktioon menee jotain syötteitä sisään ja se palauttaa niistä riippuvan tuloksen.*

$$\textit{Syöte} \rightarrow \boxed{\textit{Laskenta-algoritmi}} \rightarrow \textit{tulos}$$

Voimme nyt muotoilla määritelmän juuri tätä usean reaalimuuttujan (eli $\mathbb{R}^n$:n vektorin) tapausta varten.

Määritelmä 1.0.4. *Funktio $f : \mathbb{R}^n \rightarrow \mathbb{R}$ on* sääntö, *joka liittää jokaiseen määrittelyjoukon $D(f) \subset \mathbb{R}^n$ (pisteeseen (vektoriin)) $(x_1, \ldots, x_n)$ yksikäsitteisen luvun $f(x_1, \ldots, x_n)$. Lukujoukkoa, joka koostuu kaikista kuvapisteistä $f(x_1, \ldots, x_n)$, kun $(x_1, \ldots, x_n)$ käy läpi kaikki määrittelyjoukon $D(f)$ pisteet, sanotaan* arvojoukoksi *tai* kuvajoukoksi.

Englannink. nimitykset: *Määrittelyjoukko: "domain", arvojoukko: "range".*

Käytäntö määrittelyjoukon suhteen *Tavallisesti ymmärretään funktion määrittelyjoukko laajimmaksi* $\mathbb{R}^n$*:n osajoukoksi, jossa määrittelylauseke voidaan laskea (siis suljetaan pois nimittäjän nollakohdat tms.) Toisin on tietysti, jos määrittelyjoukko spesifioidaan erikseen.*

Kirjallisuutta

[1] R.L. Burden and J.D. Faires. Numerical Analysis. *PWS-KENT, 1997.* 5^{th} *edition.*