S-114.100 Laskennallinen tiede

Antti Kuronen
Teknillinen korkeakoulu
Laskennallisen tekniikan laboratorio
PL 9400
02015 TKK

syyskuu 1999

Esittely

Kurssi **S-114.100 Laskennallinen tiede** kuuluu sähkö- ja tietoliikennetekniikan osaston

1) laskennallisen tekniikan pää/sivuaineeseen

2) sähköfysiikan opintosuunnan pakollisiin opintoihin.

Kurssin laajuus on 3 opintoviikkoa ja suoritus koostuu harjoituksista (osuus loppuarvosanassa 50%) ja lopputentistä (50%).

Syyslukukaudella 1999 luennoitsijana toimii

tutkija Antti Kuronen
Laskennallisen tekniikan laboratorio
puh. 451 4846
e-mail Antti.Kuronen@hut.fi,

Lukujärjestys syksyllä 1999 on seuraavanlainen:

luennot:	to klo 12-14 sali S2
harjoitukset:	ke klo 12-14 sali E110

Kurssin sisältö

Päämääränä on tarjota käytännön numeerisia työkaluja laskennallisen tieteen tutkijalle.

Sisältö

1. Yleistä

2. Numeerisen laskennan perusasioita

3. Lineaariset yhtälöryhmät

4. Interpolointi

5. Numeerinen integrointi

6. Epälineaariset yhtälöt ja yhtälöryhmät

7. Funktion minimointi

8. Satunnaislukujen tuottaminen

9. Datan tilastollinen kuvailu

10. Datan mallinnus

11. Fourier-muunnos

12. Monte Carlo -menetelmistä

Materiaalia

1. Kirja Numerical Recipes tarjoaa melko käytännönläheisen esityksen numeerisista menetelmistä. Kirjasta on olemassa kolme rinnakkaisversiota, jotka eroavat ainoastaan käytetyn ohjelmointikielen osalta:

William H. Press, Saul A. Teukolsky, William T. Vetterling, Brian P. Flannery: *Numerical Recipes in C : The Art of Scientific Computing*, Cambridge University Press 1992.

William H. Press, Saul A. Teukolsky, William T. Vetterling, Brian P. Flannery: *Numerical Recipes in Fortran : The Art of Scientific Computing*, Cambridge University Press 1992.

William H. Press, Saul A. Teukolsky, William T. Vetterling, Brian P. Flannery: *Numerical Recipes in Fortran 90: The Art of Parallel Scientific Computing*, Cambridge University Press 1996.

Nämä kirjat löytyvät myös NR:n WWW-sivulta PS- tai PDF-formaatissa. Linkin löydät kurssin kotisivulta.

2. CSC-Tieteellinen laskenta on julkaissut alan oppaita:

J. Haataja, J. Käpyaho ja J. Rahola: *Numeeriset menetelmät*, Yliopistopaino 1993.

J. Haataja, J. Rahola (toim.): *Matemaattiset ohjelmistot*, Yliopistopaino 1993.

3. Verkosta löytyy luonnollisesti paljon aiheeseen liittyvää materiaalia. Varsinaista opetusmateriaalia löydät allaolevasta osoitteesta:

Computational Science Education project (CSEP, WWW-projekti; **http://csep1.phy.ornl.gov/toc.html**)

4. Kurssin kotisivu löytyy paikasta

http://www.lce.hut.fi/teaching/S-114.100

Se sisältää hyödyllisiä linkkejä, kurssimateriaalia ja ilmoitusluontoisia asioita.

Esitiedot ja edellytykset

- Matematiikan peruskurssit S1-S3.

  - Koska kurssi pyrkii olemaan käytännönläheinen, asioiden käsittely ei ole kovin teoreettista.

- Jonkin ohjelmointikielen hallinta (C, Fortran, Java).

  - Kurssilla ei opeteta varsinaisesti mitään ohjelmointikieltä, vaikka numeerisen laskennan kannalta tärkeistä ohjelmointiin liittyvistä asioista puhutaankin.

- Tietokoneen käyttömahdollisuus.

1 Yleistä

1.1 Laskennallinen tiede ja tieteellinen laskenta

Teoreettisen ja kokeellisen metodin lisäksi tieteisiin on viime vuosina tullut kolmas alue: laskennallinen tiede. Sen nousuun on vaikuttanut tietokoneiden nopeuden huima kasvu ja algoritmien kehitys.

Laskennallista tiedettä tai tieteellistä laskentaa on hieman hankala määritellä. Alla yksi yritys:

> Tieteellinen laskenta on kokoelma välineitä, tekniikoita ja teorioita, joita käytetään ratkaistaessa tietokoneella tieteen ja tekniikan ongelmien matemaattisia malleja.[1]

Tai Wilsonin mukaan[2] laskennallisen tieteen ongelmalle on luonteenomaista se, että

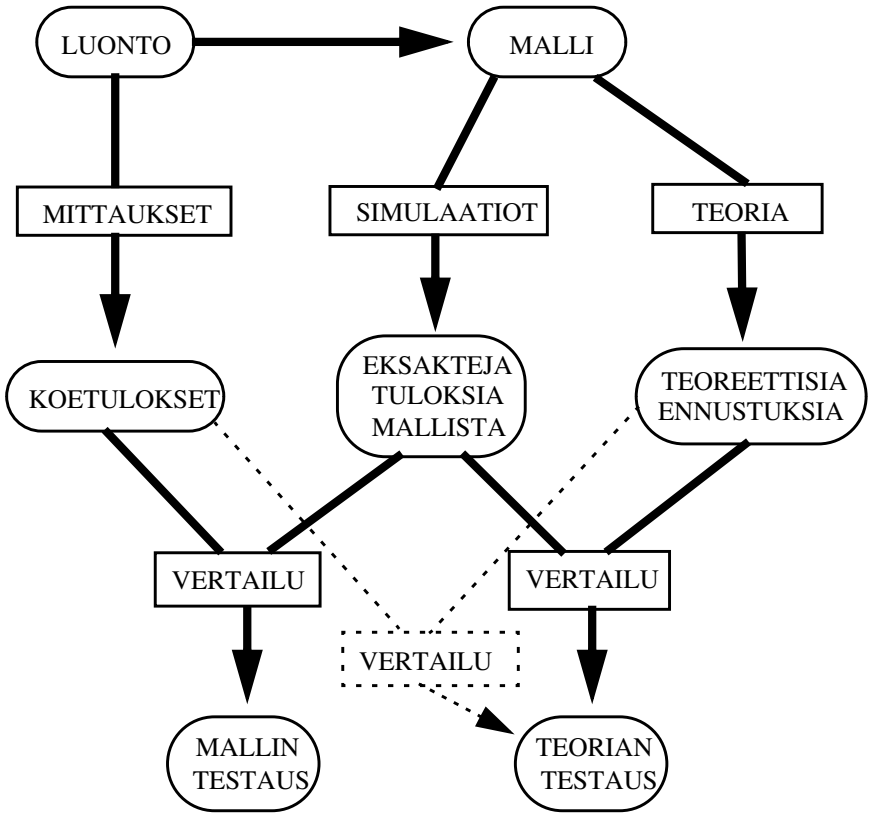
1. sillä on tarkka matemaattinen määrittely
2. se ei ole ratkaistavissa perinteisillä menetelmillä
3. se vaatii ratkaisijaltaan sovellusalueen syvällistä tuntemusta.

Laskennallisessa tieteessä on piirteitä sekä teoreettisesta että kokeellisesta alueesta. Kuten teorian avulla myös laskennallisen tutkimuksen avulla voidaan selittää koetuloksia ja ennustaa uusia. Toisaalta laskennan tuloksena saadaan usein suuri määrä luokittelematonta dataa, joka pitää jälkikäsitellä, jotta siitä pystyisi jotain päättelemään. Joskus puhutaankin tietokonekokeista (*computer experiments*).

---

1. Gene H. Golub ja James M. Ortega, *Scientific Computing and Differential Equations - An Introduction to Numerical Methods*, Academic Press, 1992
2. K. G. Wilson, *Basic Issues of Computational Science*, esitelmä konferenssissa International Conference In Computational Physics, Trieste, 1986

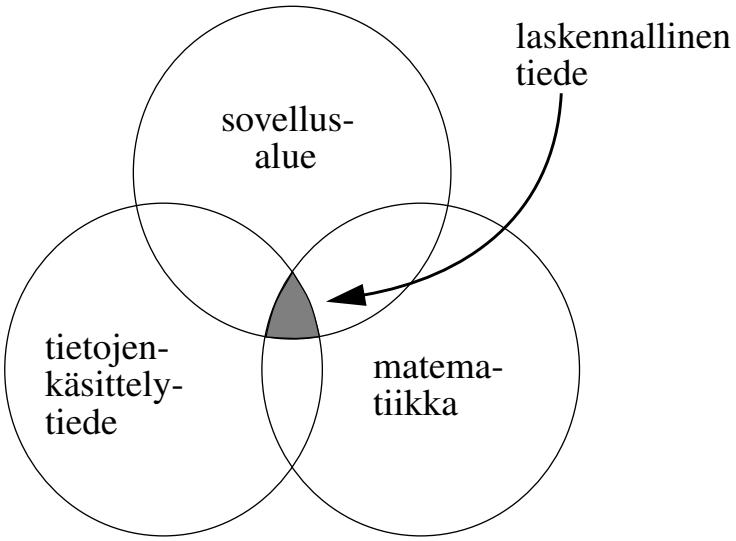
Eksakteissa luonnontieteissä (kuten fysiikassa) laskennallinen tiede esiintyy usein simulaatioiden muodossa.



Simulaatioilla
- testataan teorioita ja malleja
- kavennetaan kuilua teorian ja koetulosten välillä

Simulaatio
- on ‘algoritminen’ lähestymistapa: ei oikotietä lopputulokseen
- on monesti ainoa tapa saada tietyn mallin antamia tuloksia

Tietyllä tavalla laskennallinen tiede on hyvin poikkitieteellistä:



Laskennallinen tiede on tähän asti ollut enemmänkin eri tieteissä käytetty metodi.

Vaikuttaa kuitenkin siltä, että siitä on tulossa oma tieteenalansa. Tämä vaatii kuitenkin yhteyksien solmimista eri sovellusalojen laskijoiden kesken.

1.2 Työvälineistä

Kurssilla on tarkoitus oppia käyttämään laskennallisen tieteen harjoittamisessa tarvittavia numeerisia menetelmiä.

Näitä menetelmiä käytetään

a) itse kirjoitetuissa ohjelmissa (C tai Fortran90)

b) valmiissa aliohjelmakirjastoissa (BLAS, LAPACK, NAG, ...)

c) valmisohjelmissa (matlab, Mathematica, ...)

Pyrkimyksenä on, että opiskelija kurssin käytyään pystyy soveltamaan menetelmiä kirjoittamalla itse ohjelmia ja ymmärtämään valmisohjelmien ja kirjastojen käyttämiä menetelmiä.

Laskuharjoituksissa laaditaan useimmiten omia ohjelmia tai käytetään/muokataan valmiiksi annettua ohjelmarunkoa ongelman ratkaisemiseen.

Lisäksi joissain tehtävissä voidaan käyttää matlab-ohjelmistoa.

Kohdan a kielen valinta on tietysti kysymys, josta voisi väitellä pitkäänkin. Fortran90-kielen valintaa puoltavat kielen numeerista laskentaa helpottavat piirteet; esim. taulukkojen käsittely, liukulukuesitysten joustava valinta ym. Toisaalta C-kääntäjä löytyy löhes jokaisesta koneesta nykyään (jos ei löydy voi asentaa ilmaisen ja varsin hyvälaatuisen GNUn C-kääntäjän).

Kun vertaillaan erikielisten ohjelmien nopeutta, ei ainoastaan vertailla eri kieliä vaan myös kääntäjien tehokkuutta.

Molemmilla kielillä pystyy ilmaisemaan suunnilleen samat asiat. Alla on esitetty, miten esimerkiksi liukulukujen esitys voidaan tehdä helposti muutettavaksi Fortranilla tai C:llä.

```
program realdoubletest
implicit none

integer, parameter :: N=10000,R=4
real(kind=R),parameter:: dx=0.0001
real(kind=R) :: x,y,z
integer :: i

x=dx; y=0.0; z=0.0
do i=1,N
     x=x+dx
     y=y+sin(x)
     z=z+log(x)*cos(x)
enddo
write(0,*) x,y,z
stop
end program realdoubletest
```

```
#define REAL float
#define SIN sinf
#define LOG logf
#define COS cosf
#include <math.h>
void main()
{
  REAL x,y,z,dx=0.0001;
  int i,N=10000;

  x=dx ; y=0.0; z=0.0;
  for (i=1;i<=N;i++) {
    x+=dx;
    y+=SIN(x);
    z+=LOG(x)*COS(x);
  }
  printf(“%f %f %f\n”,x,y,z);
}
```

2

3 Numeerisen laskennan perusasioita

3.1 Virhelähteet

Laskennallista mallia muodostettaessa, ratkaisualgoritmia kehitettäessä ja algoritmin koodamisessa tietokoneohjelmaksi tehdään monia likimääräistyksiä. Näistä aiheutuvien virheiden suuruutta tulisi kontrolloida.

Virheet voidaan karkeasti jakaa neljään luokkaan: malli-, menetelmä-, lähtöarvo- ja pyöristysvirheet.[1]

3.1.1 Mallivirheet

Käytännön ongelman formulointi matemaattiseksi probleemaksi on tavallisesti mahdollista vain tekemällä yksinkertaistuksia, jotka aiheuttavat mallivirheen. Vähämerkityksisten tekijöiden poisjättäminen (ilmanvastuksen unohtaminen putoamisliikkeestä), riippuvuussuhteiden olettaminen lineaarisiksi tai tietyn teorian käyttö sen pätevyysalueen ulkopuolella (atomien kuvaus klassisia liikeyhtälöitä noudattavina massapisteinä kvanttimekaniikan sijaan) ovat esimerkkejä mallivirheistä.

Mallivirheiden käsittely kuuluu enemmänkin varsinaisen sovellusalueen vastuulle, joten niitä ei tässä käsitellä sen enempää.

3.1.2 Menetelmävirheet

Matemaattisen probleeman korvaaminen numeerisella probleemalla merkitsee myös virheen syntymistä. Tämä menetelmävirhe tehdään tietoisesti, ja sen käyttäytymistä pyritään kontrolloimaan niin, että pystytään arvioimaan, miten paljon numeerinen ratkaisu poikkeaa oikeasta ratkaisusta.

Menetelmävirhettä kutsutaan katkaisuvirheeksi, kun esimerkiksi ääretön sarja katkaistaan äärelliseksi:

$$x(t) = e^t = 1 + t + \frac{t^2}{2} + \ldots + \frac{t^n}{n!} + R_{n+1}(t) \tag{3.1}$$

$$y(t) = e^t = 1 + t + \frac{t^2}{2} + \ldots + \frac{t^n}{n!} \quad . \tag{3.2}$$

Katkaisuvirhe on $y(t) - x(t) = -R_{n+1}(t)$. Kun n riittävän suuri, on jäännöstermi $R_{n+1}(t)$ pieni, ja $y(t)$ on $x(t)$:n hyvä approksimaatio (merkitään $y(t) \approx x(t)$).

---

1. Matti Mäkelä, Olavi Nevanlinna, Juhani Virkkunen, *Numeerinen matematiikka*, Gaudeamus, 1984.

Kyseessä diskretointivirhe, jos jatkuva suure korvataan diskreetillä. Esimerkiksi derivaatan korvaaminen erotusosamäärällä:

$$x'(t) \approx y(t, h) = \frac{x(t+h) - x(t)}{h} \quad . \tag{3.3}$$

Tämä voidaan tulkita myös katkaisuvirheeksi, sillä

$$x(t+h) = x(t) + x'(t)h + R(t) \quad . \tag{3.4}$$

Menetelmävirhettä voi usein havainnollistaa graafisesti. Otetaan esimerkiksi määrätyn integraalin

$$I(x, a, b) = \int_a^b x(t)dt \tag{3.5}$$

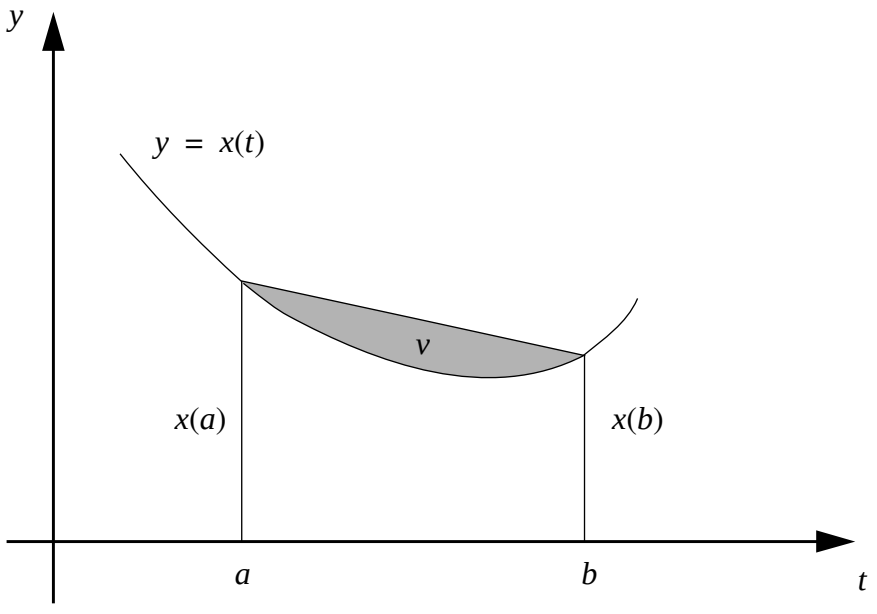
laskeminen puolisuunnikasapproksimaatiolla

$$Q(x, a, b) = \frac{b-a}{2}[x(a) - x(b)] \quad . \tag{3.6}$$

Tällöin menetelmävirhettä

$$v = Q(x, a, b) - I(x, a, b) \tag{3.7}$$

kuvaa allaolevan kuvion varjostettu alue.



Kuva 3.1: Puolisuunnikasaproksimaatio

Puolisuunnikasapproksimaatio on malliesimerkki linearisoinnista laskennallisessa tieteessä: käyrä $y = x(t)$ on korvattu suoralla.

3.1.3 Lähtöarvovirheet

Numeerisessa laskentatehtävässä voivat lähtöarvot sisältää epätarkkuuksia. Ne voivat olla esimerkiksi mittausarvoja, joilla on tietty epätarkkuus. Algoritmia suunniteltaessa on pidettävä huoli, ettei lähtöarvovirhettä laskujen aikana suurenneta siten, että pienikin virhe alkuarvoissa vahvistuu suureksi.

3.1.4 Pyöristysvirheet

Pyöristysvirheet johtuvat siitä, että liukulukujen esitykseen on varattu äärellinen määrä bittejä tietokoneen muistissa. Pyöristysvirhettä syntyy sekä lähtöarvoihin että laskujen kuluessa.

Alkuarvoihin pyöristysvirhettä voi syntyä sen vuoksi, että esitettävä luku on irrationaalinen (π, $\sqrt{2}$) tai sillä on päättymätön esitysmuoto binäärijärjestelmässä ($(0.1)_{10} = (0.00011001\ldots)_2$).

Laskutoimitusten tuloksia on lisäksi pyöristettävä. Kertolaskussa tuloksen tarkkaan esittämiseen tarvitaan suunnilleen kaksinkertainen määrä bittejä. Nämä on kuitenkin pyöristettävä normaaliin merkkimäärään.

Pyöristysvirheiden analysointi ei ole kovin helppoa, joten usein se jää tekemättä. Enintään testataan algoritmi käyttäen yksinkertaisen tarkkuden lukujen sijasta kaksoistarkkuuta, ja jos tulokset ovat yhtenevät, huokaistaan helpotuksesta.

3.2 Lukujen esitys tietokoneessa

Lukujen esittämiseen tietokoneessa käytetään äärellinen määrä bittejä. Tästä seuraa luonnollisesti se, että itseisarvoltaan mielivaltaisen suuria lukuja ei voida esittää eikä reaalilukua voi esittää mielivaltaisella tarkkuudella.

Kokonaisluvut esitetään yleensä kahden komplementtimuodossa. Ylin bitti kertoo luvun merkin, joka positiivisilla luvuilla on nolla. Negatiivinen luku saadaan vastaavasta positiivisesta komplementoimalla kaikki bitit ja lisäämällä lukuun ykkönen. Esimerkiksi 32-bittinen kokonaisluku voi saada arvot välillä

$$2147483648 = (01111111\ 11111111\ 11111111\ 11111111)_2 \text{ ja}$$
$$-2147483647 = (10000000\ 00000000\ 00000000\ 00000000)_2$$

Useimmat kääntäjät eivät anna virheilmoitusta kokonaislukualueen ylityksestä tai alituksesta. Esimerkkinä seuraava lyhyt C-ohjelma.

```
void main()
{
 short int si;
 int i,k;
 si=1 ; i=1;
 for (k=1;k<33;k++) {
  si*=2;
  i*=2;
  printf("%3d%8d%12d\n",
         k,si,i);
  }
}
```

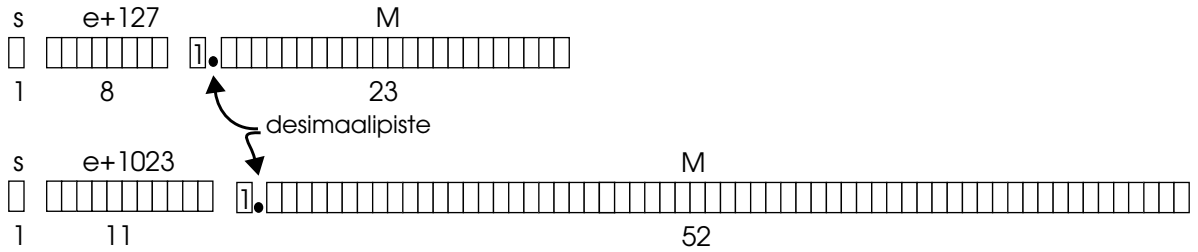
```
 2       4           4
 3       8           8
 4      16          16
 5      32          32
 6      64          64
 7     128         128
 8     256         256
 9     512         512
10    1024        1024
11    2048        2048
12    4096        4096
13    8192        8192
14   16384       16384
15  -32768       32768
16       0       65536
17       0      131072
18       0      262144
19       0      524288
20       0     1048576
21       0     2097152
22       0     4194304
23       0     8388608
24       0    16777216
25       0    33554432
26       0    67108864
27       0   134217728
28       0   268435456
29       0   536870912
30       0  1073741824
31       0 -2147483648
32       0           0
```

Reaaliluvut esitetään yleensä muodossa

$$a = s \times M \times 2^{e-E} , \tag{3.8}$$

missä s on merkkibitti, M on mantissa, e luvun eksponentti ja E vakio, jonka avulla voidaan ilmaista negatiiviset eksponentit ilman erillistä merkkibittiä. Liukulukujen esityksestä on olemassa standardi (IEEE 754[1]), jota monet tietokoneet enemmän tai vähemmän noudattavat. Se määrittelee ns. yksinkertaisen (32 bittiä eli 4 tavua) ja kaksoistarkkuuden (64 bittiä eli 8 tavua) liukuluvut.

Kuva 3.2: Yksinkertaisen tarkkuuden ja kaksoistarkkuuden liukulukujen esitykset.



Luvut normitetaan siten, että mantissan ensimmäinen bitti on aina ykkönen. Näin se voidaan jättää kokonaan pois ja säästetään yksi bitti. Puhutaan normitetuista liukuluvuista.

1. IEEE Standard for Binary Floating-Point Numbers, ANSI/IEEE std 754-1985 (IEEE, New York, 1985).

Pienin positiivinen (normitettu) liukuluku on ylläolevissa tapauksissa

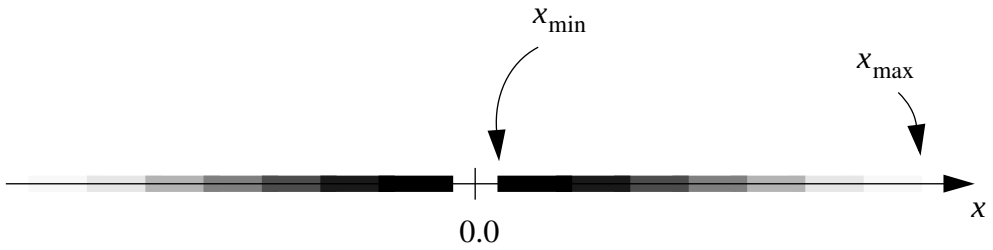
$$x_{\min} = 2^{e_{\min}} , \tag{3.9}$$

missä $e_{\min}$ on eksponentin minimiarvo. Vastaavasti suurin luku on

$$x_{\max} = 2^{e_{\max}} , \tag{3.10}$$

missä $e_{\max}$ on eksponentin maksimiarvo.

Allaoleva kuva havainnollistaa liukulukujen jakaumaa.



Kuva 3.3: Liukulukujen jakautuminen.

Kun merkkibitti on 1 on luku negatiivinen. Luvulla nolla kaikki bitit ovat nollia.

Lisäksi on määritelty etumerkillinen äärettömyys ($\pm\infty$), joka ilmaistaan asettamalla eksponentin kaikki bitit ykkösiksi ja mantissa nollaksi. Äärettömyydelle pätevät tietyt laskusäännöt:

$$\begin{aligned} \infty \pm x &= \infty \\ x \times \infty &= \infty \\ \frac{x}{\infty} &= 0 \ . \end{aligned} \tag{3.11}$$

Lisäksi on määritelty vakio NaN (not a number: mitähän lienee suomeksi?), joka ilmaistaan asettamalla luvun kaikki bitit ykkösiksi. Se saadaan tuloksena esimerkiksi kutsumalla arcuskosinia parametrilla, joka on ykköstä suurempi.

Ohessa esimerkkiohjelma C:llä:

```
#include <math.h>
void main()
{
 float a,x,y;
 x=acos(2.0);
 a=0.0;
 y=1.0/a;
 printf("%f\n",x);
 printf("%f %f %f\n",y,y+1.0e10,1.0/y);
```

```
 if (x != x) {
     printf(“Two NaNs are not equal.\n”);
 }
}
```

Käännös ja ajo (Digitalin Alpha -työasema):

```
~/cc> cc -o inftest -ieee inftest.c -lm
~/cc> inftest
NaNQ
INF INF 0.000000
Two NaNs are not equal.
~/cc> cc -o inftest inftest.c -lm
~/cc> inftest
Floating exception (core dumped)
~/cc>
```

Äärettömyyden ja NaN:n käsittely on hyvin konekohtaista; usein ohjelma vain kaatuu virheeseen nollalla jaon tms. seurauksena. Algoritmit ja ohjelmat tulisikin aina tehdä sellaisiksi, että ne itse tarkistavat, ettei laskujen tuloksena synny ko. vakioita. Esimerkiksi pitää varmistaa, että neliöjuuren argumentti on aina positiivinen.

Taulukko 3.1: Tyypillisiä liukulukujen esityksien ominaisuuksia. Parametrit on laskettu Numerical Recipes -kirjan Fortran90-ohjelmalla `machar` Digitalin Alpha-työasemassa. Ensinmainittu esitys vastaa C-kielen `float`-tyyppiä ja toinen `double`-tyyppiä. Fortran90:llä ilmaistuna tyypit ovat `real(kind=4)`, `real(kind=8)`ja `real(kind=16)`.

	Yksinkertainen tarkkuus	Kaksois-tarkkuus	Nelinkertainen tarkkuus
it[a]	24	53	113
iexp[b]	8	11	15
eps[c]	1.1920929×10^{-7}	$2.220446049250313\times10^{-16}$	$1.92592994438723585305597794258492 7\times10^{-34}$
epsneg[d]	5.9604645×10^{-8}	$1.110223024625157\times10^{-16}$	$9.62964972193617926527988971292463 7\times10^{-35}$
xmin[e]	1.1754944×10^{-38}	$2.225073858507201\times10^{-308}$	$3.36210314311209350626267781732175 3\times10^{-4932}$
xmax[f]	$3.4028235\times10^{+38}$	$1.797693134862316\times10^{+308}$	$1.18973149535723176508575932662800 7\times10^{+4932}$

a. bittien lukumäärä mantissassa
b. bittien lukumäärä eksponentissa
c. pienin luku, joka lisättynä numeroon 1.0 antaa jotain ykkösestä poikkeavaa
d. pienin luku, joka vähennettynä numerosta 1.0 antaa jotain ykkösestä poikkeavaa
e. pienin käyttökelpoinen liukuluku
f. suurin käyttökelpoinen liukuluku

```
x=dx; y=0.0; z=0.0
do i=1,N
 x=x+dx
 y=y+sin(x)
 z=z+log(x)*cos(x)
enddo
```

Mitä liukulukuesitystä sitten kannattaa käyttää? Kaksoistarkkuuden lukujen käsittely on hieman hitaampaa kuin yksinkertaisen tarkkuuden. Oikealla olevan Fortran90-silmukan laskemiseen Alpha-työasemalta meni yksinkertaisen tarkkuuden luvuilla 5.7 s ja kaksoistarkkuuden luvuilla 7.1 s. Ero on yllättävän pieni. Toisaalta Alpha-prosessorit ovat 64-bittisiä, joten datan käsittely pienemmissä yksiköissä ei tuo nopeutta paljon lisää. Linux/Intel-koneessa (NA Softwaren f90-kääntäjällä) eroa ei ollut ollenkaan. Se taas voi johtua siitä, että kaikki liukulukulaskenta tehdään kaksoistarkkuuden luvuilla.

Taulukossa (2.1) esiintyvää parametria $\mathrm{eps} \equiv \varepsilon_m$ voisi kutsua konetarkkuudeksi. Se on suhteellinen tarkkuus, jolla mielivaltainen reaaliluku pystytään esittämään tietokoneessa ja vastaa mantissan vähiten merkitsevän bitin 1:n muutosta.

Muunnettaessa reaaliluku (esimerkiksi ascii-muodossa luettu kymmenjärjestelmän luku) koneen sisäiseen esitysmuotoon tai sijoitettaessa esimerkiksi kertolaskun tulos joudutaan luku pyöristämään. Tällöin syntyy pyöristysvirhe, joka on maksimissaan $\varepsilon_m/2$. Jossain koneissa luku pyöristyksen sijasta katkaistaan, jolloin virhe on maksimissaan ε_m.

Jokaisen koneella tehtävän aritmeettisen operaation voidaan olettaa aiheuttavan ε_m:n suuruisen suhteellisen virheen tulokseen. Jos nämä virheet jakautuvat satunnaisesti ylös ja alaspäin on N:n operaation tuloksena virhe $\sqrt{N}\varepsilon_m$. Jos jostain syystä pyöristys tapahtuisi samaan suuntaan olisi virhe $N\varepsilon_m$.

Tietyissä tapauksissa tuloksen suhteellinen virhe voi kasvaa hyvinkin suureksi (*loss of significance*). Useimmiten kyseessä on kahden lähes yhtäsuuren luvun vähennys toisistaan.

Otetaan esimerkiksi lauseke $y = x - \sin x$. Olkoon meillä (kymmenjärjestelmän) liukulukuesitys, jossa luvulla on kymmenen merkitsevää numeroa. Olkoon $x = 1/15$. Tällöin liukulukuina

$$x = 0.66666667\times10^{-1}$$

$$\sin x = 0.66617295\times10^{-1}$$

$$\begin{aligned} x - \sin x &= 0.00049372\times10^{-1} \\ &= 0.4937175000\times10^{-4} \ . \end{aligned}$$

Eli alunperin kymmenen merkitsevän numeron tarkkuus pieneni kolmella. Ratkaisu ongelmaan on joko suuremman tarkkuuden omaavien liukulukujen käyttö tai (suositeltavin tapa) lausekkeen muokkaaminen sellaiseksi, että missään vaiheessa ei vähennetä samansuuruisia lukuja keskenään.

Esimerkiksi toisen asteen yhtälön

$$ax^2 + bx + c = 0 \tag{3.12}$$

toinen juuri saadaan lausekkeesta

$$x = \frac{-b + \sqrt{b^2 - 4ac}}{2a} \ . \tag{3.13}$$

Jos a tai c ovat pieniä ja $b > 0$, suoritetaan vähennyslasku $b - (b - [\text{jotain pientä}])$. Lauseke (2.13) voidaan kuitenkin sieventää muotoon

$$x = \frac{-2c}{\sqrt{b^2 - 4ac} + b} \ , \tag{3.14}$$

jossa ei tätä ongelmaa enää esiinny.

Muita esimerkkejä:

Mitä vikaa on lausekkeessa[1]

```
y = cos(x)**2-sin(x)**2 ?
```

Kun $x \approx \pi/4$, on $\sin x \approx \cos x$, ja tuloksena on merkitsevien numeroiden menetys. Käyttämällä trigonometrian kaavaa $\cos 2\theta = \cos^2\theta - \sin^2\theta$ saadaan lausekkeesta paremmin käyttäytyvä:

```
y = cos(2.0*x)
```

Merkitseviä numeroita voi hävitä myös seuraavalla tavalla. Koska funktio $\sin x$ on periodinen, laskettaessa sen arvoa argumentti x palautetaan välille $[0, 2\pi]$ (*range reduction*).

Olkoon meillä esimerkiksi $x = 12532.14$. Palautetaan tämä välille $[0, 2\pi]$: $x' = 12532.14 - 3988\pi = 3.47$. Eli alkuperäisen seitsemän merkitsevän numeron sijasta meillä on kolme. Eli lopputuloksessa $\sin(3.47)$ ei voi olla enempää kuin kolme merkitsevää numeroa.

Tällaisen virheen välttäminen on vaikeaa. Joskus kaksoistarkkuuden lukujen käyttäminen auttaa.

3.3 Ohjelmointityylistä

Hyvän numeerisen ohjelman tulisi olla

a) tehokkaasti koodattu (tietenkin)

b) helposti luettavissa ja muutettavissa (modulaarisuus)

c) helposti siirrettävissä koneelta toiselle

---

1. Niille, jotka eivät Fortrania tunne: a**b tarkoittaa a^b.

Tehokkuuteen vaikuttaa melko paljon se, miten lokaalisti ohjelma muistia käsittelee. Tämä johtuu siitä, että useat nykyaikaiset prosessorit nojaavat välimuistihierarkiaan.

Otetaan esimerkiksi allaoleva Fortran90-ohjelma, joka käsittelee taulukon alkioita joko järjestyksessä tai satunnaisesti [`unif` on reaalilukufunktio, joka palauttaa satunnaisluvun väliltä $[0, 1)$].

```
program arracc
implicit none
integer,parameter :: MAXI=10000000
real :: y,t(2),dtime,cpu,rmaxi,unif
integer :: a(0:MAXI),i,j,k,s

s=873421; a=0; cpu=dtime(t);
rmaxi=MAXI

do i=1,MAXI
     j=nint(rmaxi*unif(s))
     a(i)=j
enddo

cpu=dtime(t)
write(0,*) 'Sequential ',cpu

a=0
cpu=dtime(t)
rmaxi=MAXI

do i=1,MAXI
     j=nint(rmaxi*unif(s))
     a(j)=j
enddo
cpu=dtime(t)
write(0,*) 'Random     ',cpu

stop
end program arracc
```

Ohjelman tulostus:

```
~/f90> f90 arracc.f90
~/f90> a.out
 Sequential   1.748992
 Random       4.719936
```

Ohjelman muistinkäytön muuttaminen voi usein olla melko hankalaa. Toisaalta modernit kääntäjät osaavat tässä asiassa optimoida koodia varsin hyvin.

Muita tehokkuusnäkökohtia on esimerkiksi kaiken ylimääräisen siirtäminen pois ohjelman sisemmistä silmukoista: kaikki vakiotekijät kannattaa laskea ennen silmukoita. Kääntäjät osaavat melko hyvin optimoida tällaisiakin seikkoja.

Koodin optimointi kannattaa luonnollisesti keskittää ohjelman sellaisiin osiin, joissa kulutetaan eniten aikaa. Ajankäytön selvittäminen tapahtuu profilointityökaluilla tai lisäämällä koodiin CPU-ajan tarkkailu.

Kohtaan b kuuluvat sellaiset asiat kuten

i. Lähdekoodin kommentointi ja ulkoasu. Lähdekoodin ulkoasun tulisi heijastaa sen loogista rakennetta (sisennykset ym.). Tähän on olemassa hyviä työkaluja; suosittelen UNIXissa `GNUemacs`-editorin käyttöä.

ii. Ohjelman jakaminen aliohjelmiin. Aliohjelmien parametrien välitys tulisi olla selkeästi nähtävissä: mitä tietorakenteita kukin aliohjelma käyttää.

C:ssä tämän voi toteuttaa esimerkiksi niputtamalla muuttujat struktuureiksi ja välittämällä nämä aliohjelmille. Tällä tavalla parametrilistoista ei tule kovin pitkiä.

Fortran90:ssä normaali tapa on käyttää moduleita. Näiden haitta on se, että aliohjelman kutsussa ei näy, mitä muuttujia se käyttää.

Kaikki ohjelman parametrit, kuten esimerkiksi taulukkojen maksimikoot, tulisi määritellä vakioiksi ja koota yhteen paikkaan. Missään tapauksessa ei pidä kirjoittaa tällaisia numeroita eksplisiittisesti koodin sekaan.

C:ssä tämä toteutetaan header-tiedostoilla ja esiprosessorin `#define`-komennolla. Fortran90:ssä voidaan määritellä oma modulinsa tällaisille vakioille.
Alla sivulla pari esimerkkiä.

Fortran90-ohjelma, joka laskee piin monihiukkaspotentiaalienergiaa:

```
program kd

use sizes
use latticedata
use potentialparameters

implicit none ! Fortran90:n tärkein komento

integer :: istat

integer :: i,j,k,n,iat

character (len=80) ::  argu
integer :: iargc

real (kind=realkind) :: rij,Ri,Rin,Zi,v2i,v3i,bexp,Gtheta,lexp,texp,fexp
real (kind=realkind) :: xij,xik,xjk,yij,yik,yjk,zij,zik,zjk,drij,drj
real (kind=realkind) :: rik,rjk,dotp,dthetajik,sqrt22,thetai,lnZi
real (kind=realkind) :: v2isum,v3isum,Epot
.
.
.
```

Taulukkojen koot ym. on määritelty modulissa `sizes`:

```
module sizes
  integer,parameter :: MAXPART=10000
  integer,parameter :: MAXTYPE=2
  integer,parameter :: realkind=8   ! 4=real, 8=double precision
  integer,parameter :: dpkind=8
  real (kind=realkind),parameter :: r2d=57.29577951
end module sizes
```

Muissa moduleissa käytetään ko. modulin vakioita; esim:

```
module latticedata
  use sizes

  integer :: ix,iy,iz
  real (kind=realkind) :: a0
  real (kind=realkind) :: x(MAXPART),y(MAXPART),z(MAXPART)
  real (kind=realkind) :: bx,by,bz,b2x,b2y,b2z,density
  real (kind=realkind) :: dfrac=0.05
  integer :: iatype(MAXPART)
  integer :: npart
  integer :: rseed=875659
  character (len=2) :: aname(MAXTYPE)
  character (len=11) :: xyzfname='KD_save.xyz'
end module latticedata
```

C:llä asia hoituu tähän tyyliin:

```
#include "sizes.h"
void main(argc,argv)
     int argc;
     char **argv;
{
        char c1[MAXCHAR],c2[MAXCHAR],c3[MAXCHAR];

        int i,j;
        FILE *fd;
        int nwr;
        float x[MAXIND],tfs;
        int itype[MAXIND],np3,np3si,np3sf;
         .
         .
         .
```

Ja tiedosto `sizes.h` näyttää seuraavalta:

```
#define MAXIND 300
#define MAXCHAR 100
     .
     .
     .
```

Toisaalta: tämä ei ole ehkä paras mahdollinen esimerkki, koska kaikkein joustavin tapa määritellä taulukot on allokoida niille muistia ohjelman ajon aikan tarvittava määrä.

Helppo siirrettävyys koneelta toiselle tarkoittaa sitä, että ohjelmassa ei käytetä mitään standardien ulkopuolisia piirteitä.

Tämä ei aina ole mahdollista. Esimerkiksi Fortran90:ssä CPU-ajan mittaaminen tehdään funktioilla `etime` ja `dtime`. Nämä löytyvät lähes kaikista koneista, mutta eivät kuulu f90-standardiin. Samanlaisia piirteitä löytyy myös C-kielestä.

Ratkaisu ongelmaan on esiprosessorin käyttö[1]. Esimerkiksi C-koodiin laitetaan esiprosessorin

komentoja, jotka ohjaavat käännöstä. Olkoon meillä kirjastoaliohjelma `sub1` käytössä joissain koneissa ja `sub2` toisissa.

```
#ifdef SUB1EXISTS
     sub1(x,y,z);
#else
     sub2(x,y,z);
#endif
```

Käännöskomennossa määritellään tai ei määritellä symboli `SUB1EXISTS`:

```
cc -DSUB1EXISTS cprog.c
```

3.3.1 Numerical Recipes -kirjan C-tyylistä

Numerical Recipes -kirjan C-ohjelmat käyttävät ANSI C:n funktioiden prototyyppejä. (Kirjan softan mukana saa myös vanhantyyppiset K&R-ohjelmat.)

Prototyyppihän auttaa kääntäjää tarkistamaan funktion parametrien oikeellisuuden. Jos funktio *määritellään* seuraavasti

```
int g(int x, int y, float z)
 ...
```

näyttää prototyyppi seuraavalta

```
int g(int, int, float);
```

Funktion prototyyppimäärittelyn tulee tietysti olla näkyvissä siinä modulissa tai tiedostossa, jossa funktiota *kutsutaan*. Kaikki NR:n funktioiden prototyypit on määritelty tiedostossa `nr.h`, joka löytyy laskaripapereissa annetuista URL:stä.

C-kielessä taulukoiden indeksointi alkaa nollasta: M-alkioisen taulukon indeksit osuvat välille $0 \ldots M-1$. Toisaalta usein on luonnollista indeksoida alkaen ykkösestä: $1 \ldots M$. Tämä on helppo toteuttaa:

```
float b[4],*bb;
bb=b-1;
```

Eli on määritelty alkiot `bb[1]`,`bb[2]`,`bb[3]`,`bb[4]`. Joskus taas on luonnollista indeksoida nollasta lähtien; esimerkiksi polynomin kertoimet.

NR:n ohjelmien mukana tulevassa aputiedostossa `nrutli.c`[1] on määritelty seuraavat funk-

1. Fortran90:lle ei ole olemassa standardia esiprosessoria. Mutta aina voi käyttää C-esiprosessoria.

tiot, joilla voidaan varata muistia mielivaltaisesti indeksoituja vektoreita varten:

```
float *vector(long nl, long nh)
/* allocate a float vector with subscript range v[nl..nh] */

int *ivector(long nl, long nh)
/* allocate an int vector with subscript range v[nl..nh] */

unsigned char *cvector(long nl, long nh)
/* allocate an unsigned char vector with subscript range v[nl..nh] */

unsigned long *lvector(long nl, long nh)
/* allocate an unsigned long vector with subscript range v[nl..nh] */

double *dvector(long nl, long nh)
/* allocate a double vector with subscript range v[nl..nh] */
```

Vastaavasti on määritelty deallokointirutiinit:
`void free_<allokointirutiinin nimi>`.

Numeerisessa laskennassa käsitellään hyvin paljon matriiseja, eli kaksiulotteisia taulukoita. Olisi kovin hankalaa käyttää aina kiinteän kokoisia matriiseja, joten aliohjelmiin tulisi pystyä välittämään muuttuvankokoisia kaksiulotteisia taulukoita.

Fortran90:ssä tämä on helppoa.

Pääohjelma:

```
integer,parameter :: imin=1,imax=50,jmin=0,jmax=20
real :: a(imin:imax,jmin:jmax)
...
call subr1(a,imin,imax,jmin,jmax)
...
```

Aliohjelma:

```
subroutine subr1(a,imin,imax,jmin,jmax)
integer :: imin,imax,jmin,jmax
real :: a(imin:imax,jmin:jmax)
...
```

1. Löytyy URL:stä **http://www.lce.hut.fi/teaching/S-114.100/nrsw/cc/**

C:ssä tämä ei ole suoraan mahdollista, joten NR:ssä matriisit määritellään pointtereina pointtereihin. Esimerkkinä alla muunnos kiinteästi määritellystä matriisista pointtereihin:

```
float a[13][9],**a;
int i;
aa=(float **) malloc((unsigned) 13*sizeof(float *));
for (i=0;i<=12;i++) aa[i]=a[i];
```

Käytössä on nyt matriisi `aa[0..12][0..8]`, jonka voi välittää aliohjelmalle, joka voi käsitellä sen alkioita tietämättä sen kokoa (tietysti koko on hyvä tietää, ettei yritä hakea alkoita taulukon ulkopuolelta).

Edellämainitussa aputiedostossa `nrutil.c` on määritelty funktiot, joilla varataan muistia matriiseja varten:

```
float **matrix(long nrl, long nrh, long ncl, long nch)
/* allocate a float matrix with subscript range m[nrl..nrh][ncl..nch] */

double **dmatrix(long nrl, long nrh, long ncl, long nch)
/* allocate a double matrix with subscript range m[nrl..nrh][ncl..nch] */

int **imatrix(long nrl, long nrh, long ncl, long nch)
/* allocate a int matrix with subscript range m[nrl..nrh][ncl..nch] */
```

Lisäksi löytyvät vastaavat deallokointirutiinit.

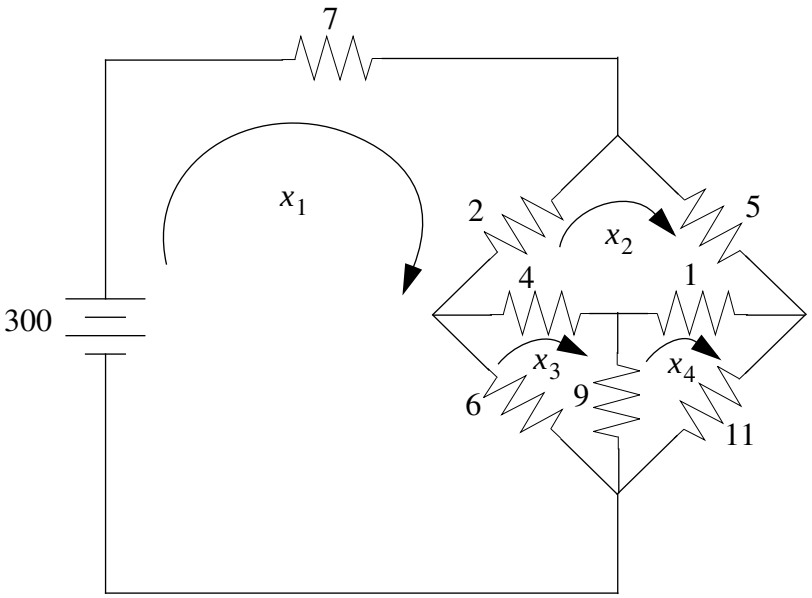
Koska C:ssä ei ole sisäänrakennettuna kompleksilukuaritmetiikkaa, on tarvittavat tyypit ja rutiinit määritelty tiedostoissa `complex.c` ja `complex.h`:

```
typedef struct FCOMPLEX {float r,i;} fcomplex;
fcomplex Cadd(fcomplex a, fcomplex b)
fcomplex Csub(fcomplex a, fcomplex b)
fcomplex Cmul(fcomplex a, fcomplex b)
fcomplex Complex(float re, float im)
fcomplex Conjg(fcomplex z)
fcomplex Cdiv(fcomplex a, fcomplex b)
```

3 Lineaariset yhtälöryhmät

3.1 Yleistä

Lineaarisia yhtälöryhmiä joudutaan ratkaisemaan hyvin monessa tekniikan ja tieteen ongelmassa. Alla esimerkkinä virtapiirin virtojen laskeminen, kun resistanssit ja lähdejännite tunnetaan.



Kuva 3.1: Virtapiiri

Systeemiä kuvaa yhtälöryhmä

$$\begin{cases} 15x_1 - 2x_2 - 6x_3 = 300 \\ -2x_1 + 12x_2 - 4x_3 - x_4 = 0 \\ -6x_1 - 4x_2 + 19x_3 - 9x_4 = 0 \\ -x_2 - 9x_3 + 21x_4 = 0 \end{cases}, \tag{3.1}$$

missä x_i ovat silmukoiden virrat. Kyseisen ongelman ratkaisu sattuu olemaan

$$\begin{cases} x_1 = 26.549158 \\ x_2 = 9.3537015 \\ x_3 = 13.254994 \\ x_4 = 6.1261261 \end{cases}. \tag{3.2}$$

Yleisesti lineaarinen yhtälöryhmä voidaan esittää muodossa

$$\mathbf{A} \cdot \mathbf{x} = \mathbf{b} \ . \tag{3.3}$$

Auki kirjoitettuna:

$$\begin{cases} a_{11}x_1 + a_{12}x_2 + a_{13}x_3 + \ldots + a_{1N}x_N = b_1 \\ a_{21}x_1 + a_{22}x_2 + a_{23}x_3 + \ldots + a_{2N}x_N = b_2 \\ a_{31}x_1 + a_{32}x_2 + a_{33}x_3 + \ldots + a_{3N}x_N = b_3 \\ \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \ldots \\ a_{M1}x_1 + a_{M2}x_2 + a_{M3}x_3 + \ldots + a_{MN}x_N = b_M \end{cases} . \tag{3.4}$$

N tuntematonta x_j, $j = 1, 2, \ldots, N$ on kytketty M:llä yhtälöllä. Kertoimet a_{ij} ja b_i ovat tunnettuja.

Jos $N = M$, yhtälöllä voi olla yksikäsitteinen ratkaisu $\mathbf{x}$ edellyttäen, että mikään M:stä yhtälöstä ei ole toisten yhtälöiden lineaarikombinaatio (rividegeneraatio) tai että mitkään muuttujista eivät esiinny yhtälöissä ainoastaan tietyissä lineaarikombinaatioissa (sarakedegeneraatio). Tämä on ekvivalenttia sen kanssa, että matriisilla $\mathbf{A}$ on olemassa käänteismatriisi $\mathbf{A}^{-1}$ tai että matriisin determinantti ei häviä: $\det(\mathbf{A}) \neq 0$.

Numeeriselta kannalta ratkaisun löytymisen voi estää, jos tietyt yhtälöt ovat lähes toistensa lineaarikombinaatioita. Tällöin laskun jossain vaiheessa pyöristysvirheet voivat aiheuttaa sen, että yhtälöt muuttuvat toistensa lineaarikombinaatioiksi.

Lisäksi pyöristysvirheet voivat aiheuttaa sen, että ratkaisu $\mathbf{x}$ saadaan laskettua, mutta se ei ole oikea. Tämä voidaan todeta sijoittamalla ratkaisu $\mathbf{x}$ yhtälöön.

Jos $M > N$, ei yhtälöryhmällä ole yleensä ratkaisua. Voidaan kuitenkin etsiä vektori $\mathbf{x}$, joka on kaikkein lähimpänä iokeaa ratkaisua, esimerkiksi pienimmän neliösumman mielessä. Tästä lisää myöhemmin.

Erilaisia lineaarialgebran tehtäviä ovat mm. seuraavat:

i. Matriisiyhtälön $\mathbf{A} \cdot \mathbf{x} = \mathbf{b}$ ratkaisun $\mathbf{x}$ etsiminen. $\mathbf{A}$ on neliömatriisi ja $\mathbf{b}$ on tunnettu vektori.

ii. Ratkaisun etsiminen useaan matriisiyhtälöön $\mathbf{A} \cdot \mathbf{x}_j = \mathbf{b}_j$ samanaikaisesti .

Jokaista ratkaisuvektoria $\mathbf{x}_j$, $j = 1, 2, \ldots$ vastaa tunnettu vektori $\mathbf{b}_j$. Tässä matriisi $\mathbf{A}$ on sama kaikille yhtälöille.

iii. Käänteismatriisin $\mathbf{A}^{-1}$ laskeminen: $\mathbf{A} \cdot \mathbf{A}^{-1} = \mathbf{1}$, missä $\mathbf{1}$ on yksikkömatriisi, jonka alkio ij on δ_{ij} .

iv. Neliömatriisin $\mathbf{A}$ determinantin laskeminen.

3.2 Gaussin eliminointi

Käsitellään aluksi hyvin yksinkertainen menetelmä yhtälön (3.3) ratkaisemiseksi: Gaussin eliminointi. Otetaan numeerinen esimerkki:

$$\begin{cases} 6x_1 - 2x_2 + 2x_3 + 4x_4 = 16 \\ 12x_1 - 8x_2 + 6x_3 + 10x_4 = 26 \\ 3x_1 - 13x_2 + 9x_3 + 3x_4 = -19 \\ -6x_1 + 4x_2 + x_3 - 18x_4 = -34 \end{cases}. \tag{3.5}$$

Ensimmäisessä vaiheessa vähennetään 1. yhtälö kerrottuna vakiolla muista yhtälöistä siten, että näistä eliminoituu x_1:

$$\begin{cases} 6x_1 - 2x_2 + 2x_3 + 4x_4 = 16 \\ -4x_2 + 2x_3 + 2x_4 = -6 \\ -12x_2 + 8x_3 + x_4 = -27 \\ 2x_2 + 3x_3 - 14x_4 = -18 \end{cases}. \tag{3.6}$$

Seuraavassa vaiheessa sama tehdään 3. ja 4. yhtälölle käyttäen 2. yhtälöä:

$$\begin{cases} 6x_1 - 2x_2 + 2x_3 + 4x_4 = 16 \\ -4x_2 + 2x_3 + 2x_4 = -6 \\ 2x_3 - 5x_4 = -9 \\ 4x_3 - 13x_4 = -21 \end{cases}. \tag{3.7}$$

Lopuksi käsitellään 4. yhtälö:

$$\begin{cases} 6x_1 - 2x_2 + 2x_3 + 4x_4 = 16 \\ -4x_2 + 2x_3 + 2x_4 = -6 \\ 2x_3 - 5x_4 = -9 \\ -3x_4 = -3 \end{cases}. \tag{3.8}$$

Yhtälö on nyt ns. yläkolmiomuodossa ja tuntematon voidaan ratkaista takaisinsijoituksella:

$$\begin{aligned} &x_4 = \frac{-3}{-3} = 1 \\ &2x_3 - 5 = -9 \Rightarrow x_3 = -2 \text{ jne.} \end{aligned} \tag{3.9}$$

Ratkaisu on siis

$$\mathbf{x} = \begin{bmatrix} 3 \\ 1 \\ -2 \\ 1 \end{bmatrix} . \tag{3.10}$$

Algoritmina eliminointi näyttää seuraavalta. Yhtälö on muotoa

$$\begin{cases} a_{11}x_1 + a_{12}x_2 + a_{13}x_3 + \ldots + a_{1N}x_N = b_1 \\ a_{21}x_1 + a_{22}x_2 + a_{23}x_3 + \ldots + a_{2N}x_N = b_2 \\ a_{31}x_1 + a_{32}x_2 + a_{33}x_3 + \ldots + a_{3N}x_N = b_3 \\ \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \ldots \\ a_{N1}x_1 + a_{N2}x_2 + a_{N3}x_3 + \ldots + a_{NN}x_N = b_N \end{cases} \tag{3.11}$$

Eliminointi koostuu $N-1$:stä askeleesta, k:nnella askeleella ($k = 1 \ldots N-1$) tehdään seuraavat sijoitukset yhtälöön i ($k+1 \le i \le N$):

$$\begin{cases} a_{ij} \leftarrow a_{ij} - \left(\dfrac{a_{ik}}{a_{kk}}\right) a_{kj} \\ \qquad\qquad\qquad , k \le j \le N \\ b_i \leftarrow b_i - \left(\dfrac{a_{ik}}{a_{kk}}\right) b_k \end{cases} . \tag{3.12}$$

Takaisinsijoitus taas tehdään seuraavasti

$$x_i = \frac{b_i - \sum_{j=i+1}^{N} a_{ij}x_j}{a_{ii}}, \; i = N, N-1, \ldots, 1 . \tag{3.13}$$

Gaussin eliminointimenetelmä sellaisenaan ei kuitenkaan ole kovin käyttökelpoinen. Otetaan esimerkiksi allaoleva triviaali yhtälö:

$$\begin{cases} 0x_1 + x_2 = 1 \\ x_1 + x_2 = 2 \end{cases} . \tag{3.14}$$

Ensimmäisen yhtälön kerroin nolla estää menetelmän käytön. Vaikka kerroin ei olisi täsmälleen nolla voi silti syntyä hankaluuksia. Esimerkiksi

$$\begin{cases} \varepsilon x_1 + x_2 = 1 \\ x_1 + x_2 = 2 \end{cases}, \tag{3.15}$$

missä ε on jotain hyvin pientä. Ensimmäisen eliminoinnin jälkeen yhtälö on muotoa

$$\begin{cases} \varepsilon x_1 + x_2 = 1 \\ \left(1 - \frac{1}{\varepsilon}\right) x_2 = 2 - \frac{1}{\varepsilon} \end{cases}. \tag{3.16}$$

Takaisinsijoituksen jälkeen saamme

$$x_2 = \frac{\left(2 - \frac{1}{\varepsilon}\right)}{\left(1 - \frac{1}{\varepsilon}\right)}, \; x_1 = \frac{1 - x_2}{\varepsilon}. \tag{3.17}$$

Koska ε on pieni, on $1/\varepsilon$ suuri ja $2 - 1/\varepsilon \approx 1 - 1/\varepsilon \approx -1/\varepsilon$.

Eli $x_2 \approx 1$ ja $x_1 \approx 0$, vaikka oikea ratkaisu on

$$x_1 = \frac{1}{1-\varepsilon} \approx 1, \; x_2 = \frac{1-2\varepsilon}{1-\varepsilon} \approx 1.$$

Eli 100% virhe x_1:n ratkaisussa!

Jos yhtälöiden järjestystä muutetaan, Gaussin eliminointi toimii:

$$\begin{cases} x_1 + x_2 = 2 \\ \varepsilon x_1 + x_2 = 1 \end{cases} \Rightarrow \begin{cases} x_1 + x_2 = 2 \\ (1-\varepsilon) x_2 = 1 - 2\varepsilon \end{cases} \Rightarrow \begin{cases} x_2 = \frac{1-2\varepsilon}{1-\varepsilon} \approx 1 \\ x_1 = 2 - x_2 \approx 1 \end{cases}. \tag{3.18}$$

Johtopäätöksenä voimme todeta, että yhtälöiden järjestys ratkaisee.

Näin pääsemme tuentaan (pivoting).

Käsittelemme ns. osittaistuentaa, jossa vaihdetaan ainoastaan yhtlöiden järjestystä. Myös matriisin $\mathbf{A}$ sarakkeita voidaan vaihdella, mutta se on ohjelmateknisesti hankalampaa.

Indeksivektori $\mathbf{l} = \left[l_1 \; l_2 \; \ldots \; l_N \right]$ kertoo järjestyksen, jossa yhtälöitä käsitellään.

Alussa jokaiselle yhtälölle lasketaan skaalaustekijä s_i

$$s_i = \max_j(|a_{ij}|),\ 1 \le i \le N, \tag{3.19}$$

joista muodostetaan skaalausvektori $\mathbf{s} = \begin{bmatrix} s_1 & s_2 & \dots & s_N \end{bmatrix}$.

Ensimmäinen eliminointi suoritetaan käyttäen yhtälön 1 sijasta yhtälöä, jolle suhde $|a_{i1}|/s_i$ on suurin. Olkoon tämän yhtälön indeksi l_1. Nyt muista yhtälöistä vähennetään yhtälö l_1 siten, että saadaan x_1:n kertoimet nolliksi.

Kätevä tapa ylläpitää indeksejä on seuraava:

Alussa asetetaan $\mathbf{l} = \begin{bmatrix} 1 & 2 & \dots & N \end{bmatrix}$.

Valitaan j siten, että se on allaolevan suhteen maksimiarvo

$$\left\{ \frac{|a_{l_i 1}|}{s_{l_i}} \middle| 1 \le i \le N \right\}.$$

Vaihdetaan l_j ja l_1 indeksivektorissa $\mathbf{l}$. Seuraavaksi käytetään kertoimia

$$\frac{a_{l_i 1}}{a_{l_1 1}}$$

kerrottuna rivillä l_1 ja vähennetään ne riveistä l_i, $(2 \le i \le N)$.

Yleensä askeleella k valitaan indeksi j, joka vastaa maksimia suhteista

$$\left\{ \frac{|a_{l_i k}|}{s_{l_i}} \middle| k \le i \le N \right\} \quad ,$$

vaihdetaan l_j ja l_k.

Käytä kertoimia

$$\frac{a_{l_i k}}{a_{l_k k}}$$

vähennettäessä tuentayhtälö l_k yhtälöistä l_i $(k+1 \le i \le N)$.

Skaalausvektoria ei päivitetä laskun aikana. Yleinen uskomus on, että tästä aiheutuva ylimääräinen laskentatyö ei kannata.

Otetaan numeerinen esimerkki. Alkuperäinen yhtälö on

$$\begin{bmatrix} 3 & -13 & 9 & 3 \\ -6 & 4 & 1 & -18 \\ 6 & -2 & 2 & 4 \\ 12 & -8 & 6 & 10 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \end{bmatrix} = \begin{bmatrix} -19 \\ -34 \\ 16 \\ 26 \end{bmatrix} .$$

Alussa $\mathbf{l} = \begin{bmatrix} 1 & 2 & 3 & 4 \end{bmatrix}$ ja $\mathbf{s} = \begin{bmatrix} 13 & 18 & 6 & 12 \end{bmatrix}$.

Ensimmäisen tuentarivin määrittämiseksi laskemme suhteet

$$\left\{ \frac{|a_{l_i 1}|}{s_{l_i}} \middle| i = 1, 2, 3, 4 \right\} = \left\{ \frac{3}{13}, \frac{6}{18}, \frac{6}{6}, \frac{12}{12} \right\} .$$

Valitaan j ensimmäisen maksimiarvon indeksiksi: $j = 3$.

Eli indeksivektori tulee muotoon $\mathbf{l} = \begin{bmatrix} 3 & 2 & 1 & 4 \end{bmatrix}$.

Nyt vähennetään rivi 3 muista riveistä kerrottuna asianmukaisilla kertoimilla. Yhtälö tulee muotoon

$$\begin{bmatrix} 0 & -12 & 8 & 1 \\ 0 & 2 & 3 & -14 \\ 6 & -2 & 2 & 4 \\ 0 & -4 & 2 & 2 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \end{bmatrix} = \begin{bmatrix} -27 \\ -18 \\ 16 \\ -6 \end{bmatrix} .$$

Seuraavaksi etsimme maksimin joukosta

$$\left\{ \frac{|a_{l_i 2}|}{s_{l_i}} \middle| i = 2, 3, 4 \right\} = \left\{ \frac{2}{18}, \frac{12}{13}, \frac{4}{12} \right\} .$$

Saamme $j = 3$ eli indeksivektori tulee muotoon $\mathbf{l} = \begin{bmatrix} 3 & 1 & 2 & 4 \end{bmatrix}$. Nyt yhtälö 1 vähennetään yhtälöistä 2 ja 4 kerrottuina vakioilla -1/6 ja 1/3:

$$\begin{bmatrix} 0 & -12 & 8 & 1 \\ 0 & 0 & \frac{13}{3} & -\frac{83}{6} \\ 6 & -2 & 2 & 4 \\ 0 & 0 & -\frac{2}{3} & \frac{5}{3} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \end{bmatrix} = \begin{bmatrix} -27 \\ -\frac{45}{2} \\ 16 \\ 3 \end{bmatrix} .$$

Kolmannessa eli viimeisessä vaiheessa riittää tutkia suhteita

$$\left\{ \frac{|a_{l_i 3}|}{s_{l_i}} \middle| i = 3, 4 \right\} = \left\{ \frac{13/3}{18}, \frac{2/3}{12} \right\} .$$

Maksimi löytyy arvolla $j = 3$ eli indeksivektoria ei tarvitse muuttaa.
Tuentayhtälö on $l_3 = 2$ ja yhtälö 2 kerrottuna vakiolla -2/13 vähennetään yhtälöstä 4:

$$\begin{bmatrix} 0 & -12 & 8 & 1 \\ 0 & 0 & \frac{13}{3} & -\frac{83}{6} \\ 6 & -2 & 2 & 4 \\ 0 & 0 & 0 & -\frac{6}{13} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \end{bmatrix} = \begin{bmatrix} -27 \\ -\frac{45}{2} \\ 16 \\ -\frac{6}{13} \end{bmatrix} .$$

Indeksivektori on nyt $\mathbf{l} = \begin{bmatrix} 3 & 1 & 2 & 4 \end{bmatrix}$ ja ratkaisu saadaan takaisinsijoituksella käymällä läpi yhtälöt indeksivektorin lopusta lukien:

$$x_4 = \frac{-6/13}{-6/13} = 1$$

$$x_3 = \frac{(-45/2) + (83/6)(1)}{13/3} = -2$$

$$x_2 = \frac{-27 - 8(-2) - 1(1)}{-12} = 1$$

$$x_1 = \frac{16 + 2(1) - 2(-2) - 4(1)}{6} = 3 .$$

Ratkaisu on siis

$$\mathbf{x} = \begin{bmatrix} 3 \\ 1 \\ -2 \\ 1 \end{bmatrix} .$$

3.3 Gauss-Jordan-eliminointi

Gaussin eliminoinnissa tuloksena saamme ainoastaan yhtä vektoria $\mathbf{b}$ vastaavan ratkaisun. Gauss-Jordan-eliminoinnissa voimme samanaikaisesti laskea sekä useata vektoria $\mathbf{b}$ vastaavat ratkaisut ja matriisin $\mathbf{A}$ käänteismatriisin $\mathbf{A}^{-1}$.

'Lavennnetaan' yhtälö (3.3) muotoon

$$\mathbf{A} \cdot [\mathbf{x}_1 \cup \mathbf{x}_2 \cup \mathbf{x}_3 \cup \mathbf{Y}] = [\mathbf{b}_1 \cup \mathbf{b}_2 \cup \mathbf{b}_3 \cup \mathbf{1}] . \tag{3.20}$$

$\mathbf{1}$ on yksikkömatriisi, $\mathbf{x}_i$ ja $\mathbf{b}_i$ ovat pystyvektoreita ja operaatio $\mathbf{A} \cup \mathbf{B}$ tarkoittaa tässä matriisien yhdistämistä:

$$\begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{bmatrix} \cup \begin{bmatrix} b_{11} & b_{12} & b_{13} \\ b_{21} & b_{22} & b_{23} \\ b_{31} & b_{32} & b_{33} \end{bmatrix} = \begin{bmatrix} a_{11} & a_{12} & a_{13} & b_{11} & b_{12} & b_{13} \\ a_{21} & a_{22} & a_{23} & b_{21} & b_{22} & b_{23} \\ a_{31} & a_{32} & a_{33} & b_{31} & b_{32} & b_{33} \end{bmatrix} . \tag{3.21}$$

Helposti nähdään, että yhtälö (3.20) vastaa neljää yhtälöä

$$\mathbf{A} \cdot \mathbf{x}_1 = \mathbf{b}_1 , \mathbf{A} \cdot \mathbf{x}_2 = \mathbf{b}_2 , \mathbf{A} \cdot \mathbf{x}_3 = \mathbf{b}_3 \tag{3.22}$$

$$\mathbf{A} \cdot \mathbf{Y} = \mathbf{1} . \tag{3.23}$$

Helposti voimme nähdä, että seuraavat operaatiot eivät muuta yhtälöryhmää:

i. Kahden rivin vaihto matriisissa $\mathbf{A}$ ja vastaavien rivien vaihto vektoreissa $\mathbf{b}_i$ ja matriisissa $\mathbf{1}$.

ii. Tietyn rivin korvaaminen matriisissa $\mathbf{A}$ kaikkien rivien lineaarikombinaatiolla ja vastaavan muutoksen tekeminen vektoreissa $\mathbf{b}_i$ ja matriisissa $\mathbf{1}$.

iii. Kahden sarakkeen vaihto matriisissa $\mathbf{A}$ ja vastaavien rivien vaihto vektoreissa $\mathbf{x}_i$ ja matriisissa $\mathbf{Y}$.

Tuetussa Gauss-Jordan-eliminoinnissa (GJ) käytetään yllämainittuja operaatioita saattamaan matriisi $\mathbf{A}$ yksikkömatriisiksi. Tällöin yhtälöt (3.22) ja (3.23) tulevat muotoon

$$\mathbf{1} \cdot \mathbf{x}_1 = \mathbf{b}'_1 \, , \, \mathbf{1} \cdot \mathbf{x}_2 = \mathbf{b}'_2 \, , \, \mathbf{1} \cdot \mathbf{x}_3 = \mathbf{b}'_3 \tag{3.24}$$

$$\mathbf{1} \cdot \mathbf{Y} = \mathbf{A}^{-1} \, . \tag{3.25}$$

Eli saamme useaa vektoria $\mathbf{b}_i$ vastaavat ratkaisut $\mathbf{x}_i$ sekä $\mathbf{A}$:n käänteismatriisin $\mathbf{A}^{-1}$.

GJ-eliminointi (ilman tuentaa) etenee seuraavasti:

i. Ensimmäinen rivi jaetaan a_{11}:llä ja muista riveistä vähennetään 1. rivi skaalattuna siten, että kertoimet a_{i1} ovat nollia. Nyt ensimmäinen sarake vastaa yksikkömatriisia.

ii. Menemme toiseen sarakkeeseen ja jaamme toisen rivin kertoimella a_{22} ja vähennämme toisen rivin asianmukaisesti skaalattuna muista riveistä.

Esimerkiksi 3x3-matriisi.

$$\begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{bmatrix} \rightarrow \begin{bmatrix} 1 & a'_{12} & a'_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{bmatrix} \rightarrow \begin{bmatrix} 1 & a'_{12} & a'_{13} \\ 0 & a'_{22} & a'_{23} \\ 0 & a'_{32} & a'_{33} \end{bmatrix} \rightarrow$$

$$\begin{bmatrix} 1 & a'_{12} & a'_{13} \\ 0 & 1 & a''_{23} \\ 0 & a'_{32} & a'_{33} \end{bmatrix} \rightarrow \begin{bmatrix} 1 & 0 & a''_{13} \\ 0 & 1 & a''_{23} \\ 0 & 0 & a''_{33} \end{bmatrix} \rightarrow \begin{bmatrix} 1 & 0 & a''_{13} \\ 0 & 1 & a''_{23} \\ 0 & 0 & 1 \end{bmatrix} \rightarrow \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

Vastaavat muutokset tulee tietysti tehdä vektoreihin $\mathbf{b}_i$ ja oikealla puolella olevaan yksikkömatriisiin.

Kuten Gaussin eliminointi on GJ-eliminointikin virhealtis, ellei tehdä tuentaa eli valita edullisimmalla mahdollisimmalla tavalla se järjestys, jossa $\mathbf{A}$:n elementit käydään läpi. Ilman tuentaa se tapahtuu numerojärjestyksessä: a_{11}, a_{22}, a_{33}, ….

Kriteerin löytäminen edullisimmalle järjestykselle on teoreettisesti vaikea ongelma. Useimmiten valitaan mahdollisten alkioiden joukosta itseisarvoltaan suurin.

Tuenta voi olla osittaista, jos vaihdellaan vain rivien järjestystä, ja täydellistä, jos vaihdellaan sekä rivien että sarakkeiden järjestystä. Täydellinen tuenta on ohjelmointiteknisesti hankalampi toteuttaa kuin osittaistuenta.

NR:n aliohjelma `gaussj` tekee GJ-eliminoinnin käyttäen täydellistä tuentaa.

3.4 LU-hajotelma

Matriisin **A** LU-hajotelmaksi (**l**ower triangular-**u**pper triangular) kutustaan matriiseja **L** ja **U**, jotka toteuttavat seuraavan yhtälön

$$\mathbf{L} \cdot \mathbf{U} = \mathbf{A} \ . \tag{3.26}$$

Lisäksi **L** on alakolmiomatriisi ja **U** yläkolmiomatriisi. Esim. 4x4-matriisille

$$\begin{bmatrix} \alpha_{11} & 0 & 0 & 0 \\ \alpha_{21} & \alpha_{22} & 0 & 0 \\ \alpha_{31} & \alpha_{32} & \alpha_{33} & 0 \\ \alpha_{41} & \alpha_{42} & \alpha_{43} & \alpha_{44} \end{bmatrix} \cdot \begin{bmatrix} \beta_{11} & \beta_{12} & \beta_{13} & \beta_{14} \\ 0 & \beta_{22} & \beta_{23} & \beta_{24} \\ 0 & 0 & \beta_{33} & \beta_{34} \\ 0 & 0 & 0 & \beta_{44} \end{bmatrix} = \begin{bmatrix} a_{11} & a_{12} & a_{13} & a_{14} \\ a_{21} & a_{22} & a_{23} & a_{24} \\ a_{31} & a_{32} & a_{33} & a_{34} \\ a_{41} & a_{42} & a_{43} & a_{44} \end{bmatrix} \ . \tag{3.27}$$

Sijoittamalla hajotelma (3.26) yhtälöön (3.2) saamme

$$\mathbf{A} \cdot \mathbf{x} = (\mathbf{L} \cdot \mathbf{U}) \cdot \mathbf{x} = \mathbf{L} \cdot (\mathbf{U} \cdot \mathbf{x}) = \mathbf{b} \ . \tag{3.28}$$

Voimme ratkaista yhtälön kahdessa osassa

$$\mathbf{L} \cdot \mathbf{y} = \mathbf{b} \ , \tag{3.29}$$

$$\mathbf{U} \cdot \mathbf{x} = \mathbf{y} \ . \tag{3.30}$$

Etuna tässä on se, että yhtälöryhmän ratkaisu on triviaalia, kun kerroinmatriisi on kolmiomuotoinen.

Yhtälö (3.29) voidaan ratkaista eteenpäin sijoituksella (forward substitution):

$$\begin{aligned} y_1 &= \frac{b_1}{\alpha_{11}}, \\ y_i &= \frac{1}{\alpha_{ii}} \left[b_i - \sum_{j=1}^{i-1} \alpha_{ij} y_j \right] \ , \ i = 2, 3, \ldots, N \ . \end{aligned} \tag{3.31}$$

Yhtälö (3.30) taas voidaan ratkaista taaksepäin sijoituksella (backward substitution)

$$x_N = \frac{y_N}{\beta_{NN}},$$
$$x_i = \frac{1}{\beta_{ii}}\left[y_i - \sum_{j=i+1}^{N} \beta_{ij} x_j\right],\ i = N-1, N-2, \ldots, 1\ . \tag{3.32}$$

Menetelmän etuna on se, että kunhan olemme kerran tehneet LU-hajotelman, voimme käyttää sitä useita vektoreita **b** vastaavien ratkaisujen laskemiseen. Ylläolevat laskut vaativat CPU-aikaa $O(N^2)$, kun taas esimerkiksi Gaussin tai GJ-eliminointi vaativat aikaa $O(N^3)$. Itse LU-hajotelman laskeminen vaatii myös tuon $O(N^3)$.

Huomaa, että Gaussin eliminoinnissa itseasiassa muodostetaan matriisin **A** LU-hajotelma. Malliratkaisuohjelmassa LU-hajotelma palautetaan ratkaisualiohjelmasta matriisissa **A**. Hajotelma ei ole aivan yhtälön (3.27) muotoinen vaan

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ \alpha_{21} & 1 & 0 & 0 \\ \alpha_{31} & \alpha_{32} & 1 & 0 \\ \alpha_{41} & \alpha_{42} & \alpha_{43} & 1 \end{bmatrix} \cdot \begin{bmatrix} \beta_{11} & \beta_{12} & \beta_{13} & \beta_{14} \\ 0 & \beta_{22} & \beta_{23} & \beta_{24} \\ 0 & 0 & \beta_{33} & \beta_{34} \\ 0 & 0 & 0 & \beta_{44} \end{bmatrix} = \begin{bmatrix} a_{11} & a_{12} & a_{13} & a_{14} \\ a_{21} & a_{22} & a_{23} & a_{24} \\ a_{31} & a_{32} & a_{33} & a_{34} \\ a_{41} & a_{42} & a_{43} & a_{44} \end{bmatrix}\ . \tag{3.33}$$

Näin saadaan sekä **L** että **U** mahtumaan samaan taulukkoon. (Ykkösten saaminen **L**:n diagonaalille käy kertomalla se diagonaalimatriisilla; esim. 3×3-tapaus:

$$\begin{bmatrix} 1/\alpha_{11} & 0 & 0 \\ 0 & 1/\alpha_{22} & 0 \\ 0 & 0 & 1/\alpha_{33} \end{bmatrix} \cdot \begin{bmatrix} \alpha_{11} & 0 & 0 \\ \alpha_{21} & \alpha_{22} & 0 \\ \alpha_{31} & \alpha_{32} & \alpha_{33} \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 \\ \dfrac{\alpha_{21}}{\alpha_{22}} & 1 & 0 \\ \dfrac{\alpha_{31}}{\alpha_{33}} & \dfrac{\alpha_{32}}{\alpha_{33}} & 1 \end{bmatrix}\ .) \tag{3.34}$$

Entä miten tehdään tehokkaasti LU-hajotelma?

Ryhmän (3.27) ij:s yhtälö on muotoa

$$\alpha_{i1}\beta_{1j} + \ldots = a_{ij}\ . \tag{3.35}$$

Se, montako termiä yhtälössä on riippuu i:n ja j:n suuruusjärjestyksestä:

$$\alpha_{i1}\beta_{i1} + \alpha_{i2}\beta_{i2} + \ldots + \alpha_{ii}\beta_{ij} = a_{ij}, \qquad i < j \tag{3.36}$$

$$\alpha_{i1}\beta_{i1} + \alpha_{i2}\beta_{i2} + \ldots + \alpha_{ii}\beta_{jj} = a_{ij}, \qquad i = j \tag{3.37}$$

$$\alpha_{i1}\beta_{i1} + \alpha_{i2}\beta_{i2} + \ldots + \alpha_{ij}\beta_{jj} = a_{ij}, \qquad i > j\ . \tag{3.38}$$

Yhtälöitä (3.36)-(3.38) on N^2 kappaletta yhteensä $N^2 + N$:lle tuntemattomalle (diagonaalielementit on jaettu kahteen osaan!). Voidaan osoittaa, että tästä ongelmasta päästään eroon asettamalla

$$\alpha_{ii} = 1\,,\ i = 1, \ldots, N\,. \tag{3.39}$$

NR:n käyttämä Croutin algoritmi ratkaisee yhtälöt asettamalla ne seuraavanlaiseen järjestykseen:

- Aseta $\alpha_{ii} = 1\ ,\ \ i = 1, \ldots, N\ \ .$

- Jokaiselle $j = 1, 2, \ldots, N$ tee seuraavaa:

  i. Kaikille $i = 1, 2, \ldots, j$ ratkaise β_{ij} yhtälöiden (3.36), (3.37) ja (3.39) avulla:

$$\beta_{ij} = a_{ij} - \sum_{k=1}^{i-1} \alpha_{ik}\beta_{kj}\ \ (i = 1 \rightarrow \text{ei summausta}) \tag{3.40}$$

  ii. Kaikille $i = j + 1, j + 2, \ldots, N$ ratkaise α_{ij} käyttäen yhtälöä (3.38):

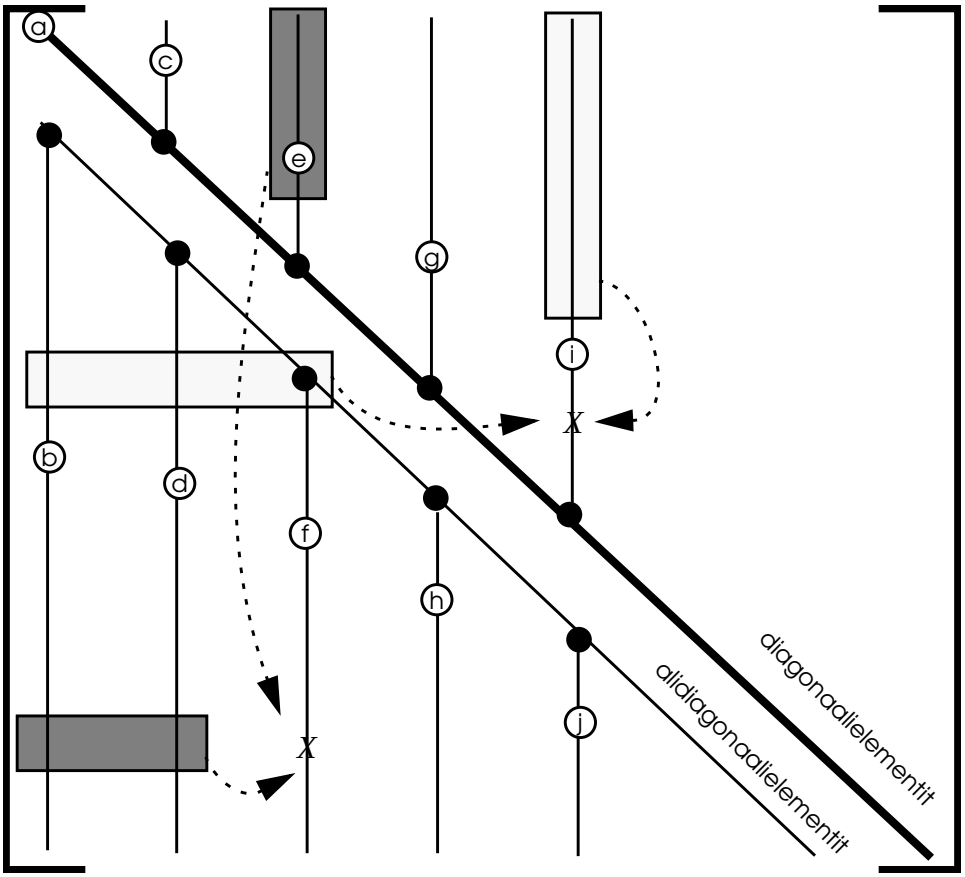
$$\alpha_{ij} = \frac{1}{\beta_{jj}}\left(a_{ij} - \sum_{k=1}^{j-1} \alpha_{ik}\beta_{kj} \right)\,. \tag{3.41}$$

Käymällä läpi muutaman iteraation, on helppo vakuuttua siitä, että ne α:t ja β:t, jotka esiintyvät yhtälöiden (3.40) ja (3.41) oikealla puolella ovat jo määrätty, kun niitä käytetään. Lisäksi jokaista kerrointa a_{ij} käytetään ainoastaan kerran koko laskennan aikana. Tästähän seuraa se, että matriisia $\mathbf{A}$ voidaan käyttää $\mathbf{L}$:n ja $\mathbf{U}$:n talletukseen. Eli $\mathbf{A}$:n tilalle syntyy matriisi

$$\begin{bmatrix} \beta_{11} & \beta_{12} & \beta_{13} & \beta_{14} \\ \alpha_{21} & \beta_{22} & \beta_{23} & \beta_{24} \\ \alpha_{31} & \alpha_{32} & \beta_{33} & \beta_{34} \\ \alpha_{41} & \alpha_{42} & \alpha_{43} & \beta_{44} \end{bmatrix}\,. \tag{3.42}$$

Croutin algoritmi täyttää siis ylläolevan matriisin sarakkeet vasemmalta oikealle ja jokaisen

sarakkeen ylhäältä alas.



Kuva 3.2: Croutin LU-hajotelma-algoritmi

Käydään läpi pienin mahdollinen ei-triviaali esimerkki: 3×3-matriisi:

$$\mathbf{A} = \begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{bmatrix} = \begin{bmatrix} 9 & 2 & 3 \\ 4 & 2 & 4 \\ 1 & 1 & 9 \end{bmatrix}.$$

Etsitään hajotelma (siis $\alpha_{ii} = 1$)

$$\begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 \\ \alpha_{21} & 1 & 0 \\ \alpha_{31} & \alpha_{32} & 1 \end{bmatrix} \cdot \begin{bmatrix} \beta_{11} & \beta_{12} & \beta_{13} \\ 0 & \beta_{22} & \beta_{23} \\ 0 & 0 & \beta_{33} \end{bmatrix}$$

$j = 1 \quad i = 1$ (3.40) $\beta_{11} = a_{11} = 9$

$i = 2$ (3.41) $\alpha_{21} = \frac{1}{\beta_{11}} a_{21} = \frac{4}{9} = 0.4444$

$$i = 3 \ (3.41) \ \alpha_{31} = \frac{1}{\beta_{11}} a_{31} = \frac{1}{9} = 0.1111$$

$j = 2 \quad i = 1 \ (3.40) \ \beta_{12} = a_{12} = 2$

$$i = 2 \ (3.40) \qquad \beta_{22} = a_{22} - \alpha_{21}\beta_{12} = 2 - \frac{4}{9}2 = 1.111$$

$$i = 3 \ (3.41) \ \alpha_{32} = \frac{1}{\beta_{22}}(a_{32} - \alpha_{31}\beta_{12}) = \frac{1}{1.111}\left(1 - \frac{2}{9}\right) = 0.7000$$

$j = 3 \quad i = 1 \ (3.40) \ \beta_{13} = a_{13} = 3$

$$i = 2 \ (3.40) \ \beta_{23} = a_{23} - \alpha_{21}\beta_{13} = 4 - \frac{4}{9}3 = 2.6667$$

$$i = 3 \ (3.40) \ \beta_{33} = a_{33} - \alpha_{31}\beta_{13} - \alpha_{32}\beta_{23} = \ldots = 6.8000$$

Kokoamalla tulokset saadaan:

$$\mathbf{L} = \begin{bmatrix} 1 & 0 & 0 \\ 0.4444 & 1 & 0 \\ 0.1111 & 0.7000 & 1 \end{bmatrix}$$

$$\mathbf{U} = \begin{bmatrix} 9 & 2 & 3 \\ 0 & 1.1111 & 2.6667 \\ 0 & 0 & 6.8000 \end{bmatrix} .$$

Tuloksen voi tietysti tarkastaa kertomalla matriisit keskenään.

Jotta Croutin algoritmi olisi stabiili, on hajotelma tehtävä käyttäen tuentaa.

NR:n aliohjelma `ludcmp` tekee LU-hajotelman käyttäen osittaistuentaa (vaihdellaan vain rivien järjestystä). Rutiini palauttaa alkuperäisessä matriisissa **A** ala- ja yläkolmimatriisin rivi-permutaation ja lisäksi vektorin, joka kertoo, mikä tuo permutaatio on:

```
void ludcmp(float **a, int n, int *indx, float *d)
```

Tässä `a` on `n`×`n`-matriisi, `indx` kertoo, missä järjestyksessä hajotetun matriisin rivit ovat ja `d` kertoo onko tehty parillinen (+1) vai pariton määrä rivien vaihtoja.

Aliohjelma `lubksb` taas laskee yhtälöryhmän ratkaisun takaisinsijoituksella:

```
void lubksb(float **a, int n, int *indx, float b[])
```

Vektori b on yhtälöryhmän oikea puoli ja siinä palautetaan ratkaisu.
Eli yhtälöryhmien $\mathbf{A} \cdot \mathbf{x} = \mathbf{b}$ ratkaisu sujuu näillä rutiineilla seraavasti:

```
float **a,*b,*b1,*b2,d;
int n, *indx;
.
.
.
ludcmp(a,n,indx,&d);
lubksb(a,n,indx,b);
.
.
.
lubksb(a,n,indx,b1);
.
.
.
lubksb(a,n,indx,b1);
.
.
.
```

Jos LU-hajotelma on tehty, on $\mathbf{A}$:n käänteismatriisin laskeminen helppoa:

```
#define N ...
float **a,**y,d,*col;
int i,j,*indx;
.
.
.
ludcmp(a,N,indx,&d);
for (j=1;j<=N;j++) {
     for (i=1;i<=N;i++) col[i]=0.0;
     col[j]=1.0;
     lubksb(a,N,indx,col);
     for (i=1;i<=N;i++) y[i][j]=col[i];
}
```

Matriisin $\mathbf{A}$ determinantti on helppo laskea LU-hajotelmasta. Koska

$$\det(\mathbf{X} \cdot \mathbf{Y}) = \det(\mathbf{X})\det(\mathbf{Y}) \tag{3.43}$$

ja lisäksi kolmiomatriisin determinantti on sen dioagonaalielementtien tulo saadaan (muista, että asetimme $\alpha_{ii} = 1$)

$$\det(\mathbf{A}) = \prod_{i=1}^{N} \beta_{ii} \ . \tag{3.44}$$

Lisäksi on muistettava, että edellämainitut rutiinit palauttavat permutoidut matriisit **L** ja **U**. Determinantti vaihtaa merkkiä, jos matriisin kaksi riviä vaihtaa keskenään. Näinollen tulos (3.44) on vielä kerrottava vakiolla `d`, jonka `ludcmp` palauttaa:

```
#define N ...
float **a,d;
int i,*indx;
.
.
.
ludcmp(a,N,indx,&d),
for (i=1;i<=N,i++) d *= a[i][i];
```

3.5 Iteratiiviset menetelmät

Varsinkin suurten yhtälöryhmien tapauksissa pyöristysvirheet vaikeuttavat edelläesitettyjen suorien menetelmien käyttöä. Tällöin voidaan apuna käyttää iteratiivisia menetelmiä.

Olkoon $\mathbf{x}$ tavalliseen tapaan yhtälön

$$\mathbf{A} \cdot \mathbf{x} = \mathbf{b} \tag{3.45}$$

ratkaisu. Emme tiedä sitä, mutta tunnemme likiarvon sille: $\mathbf{x} + \delta\mathbf{x}$, missä $\delta\mathbf{x}$ on tuntematon virhe. Sijoitettaessa tämä yhtälöryhmään saadaan hieman virheellinen oikea puoli

$$\mathbf{A} \cdot (\mathbf{x} + \delta\mathbf{x}) = \mathbf{b} + \delta\mathbf{b} \ . \tag{3.46}$$

Vähentämällä (3.45) (3.46):sta saadaan

$$\mathbf{A} \cdot \delta\mathbf{x} = \delta\mathbf{b} \ . \tag{3.47}$$

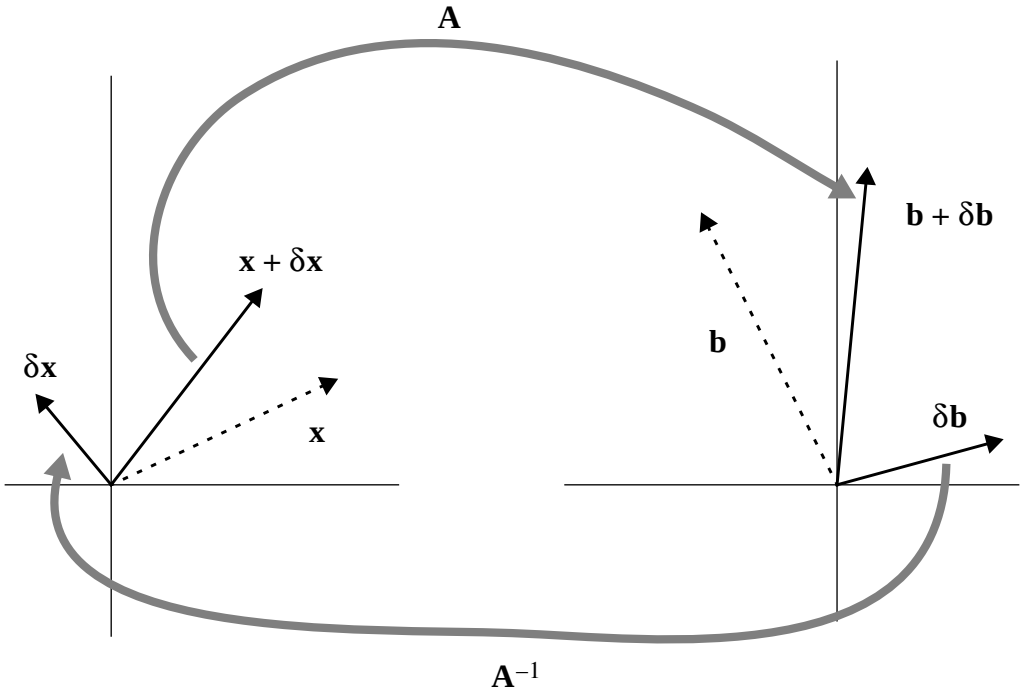
Ratkaisemalla $\delta\mathbf{b}$ (3.46):sta ja sijoittamalla se yhtälöön (3.47) saadaan

$$\mathbf{A} \cdot \delta\mathbf{x} = \mathbf{A} \cdot (\mathbf{x} + \delta\mathbf{x}) - \mathbf{b} \ . \tag{3.48}$$

Oikea puoli on nyt tiedossa, koska $\mathbf{x} + \delta\mathbf{x}$ on laskettu likiarvo ratkaisulle. Iteroivassa ratkaisussa ratkaistaan $\delta\mathbf{x}$ yhtälöstä (3.48) ja vähennetään se likiarvoratkaisusta. Näin saamme parannetun ratkaisun.

Asiaa helpottaa vielä se, että jos olemme laskeneet $\mathbf{A}$:n LU-hajotelman, ainoastaan takaisinsijoitus riittää (3.48):n ratkaisemiseksi.

Seuraava kuva havainnollistaa asiaa.



Kuva 3.3: Iteratiivinen ratkaisu

NR:n rutiini `mprove` tekee tämän iteroinnin.

Koska LU-hajotelma tallentaa tuloksen alkuperäiseen matriisiin `a`, ja iteroinnissa tarvitaan alkuperäistä matriisia, on se kopioitava talteen.

3.6 Harvat matriisit

Hyvin usein matriisi **A** on ns. harva matriisi eli suurin osa sen alkioista on nollia. Tällöin ei kannata käyttää edelläesiteltyjä suoria menetelmiä, sillä suurin osa laskenta-ajasta kuluu nolla-alkioiden käsittelyyn. Lisäksi muistia tuhlataan talletettaessa suuri määrä nollia.

Tiettyjen harvojen matriisien tapauksissa yhtälöryhmän ratkaisu esimerkiksi LU-hajotelmalla on triviaalia. Otetaan esimerkiksi tridiagonaalinen systeemi:

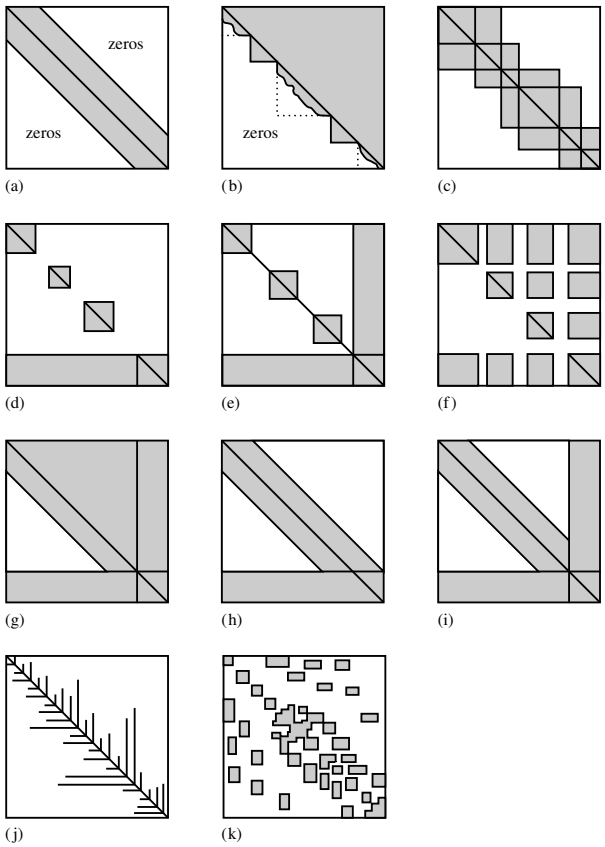
$$\begin{bmatrix} b_1 & c_1 & 0 & \dots & & & \\ a_2 & b_2 & c_2 & \dots & & & \\ & & & \dots & & & \\ & & & \dots & a_{N-1} & b_{N-1} & c_{N-1} \\ & & & & 0 & a_N & b_N \end{bmatrix} \cdot \begin{bmatrix} u_1 \\ u_2 \\ \dots \\ u_{N-1} \\ u_N \end{bmatrix} = \begin{bmatrix} r_1 \\ r_2 \\ \dots \\ r_{N-1} \\ r_N \end{bmatrix}, \qquad (3.49)$$

Eli jos $|i - j| > 1$ on $a_{ij} = 0$. Tällaiseen yhtälöön päädytään esimerkiksi kuutiosplini-interpolaatiossa (josta lisää myöhemmin). NR:n rutiin `tridag` ratkaisee ylläolevan yhtälöryhmän LU-hajotelmalla.

```
void tridag(float a[], float b[], float c[], float r[], float u[],
        unsigned long n)
{
        unsigned long j;
        float bet,*gam;

        gam=vector(1,n);
        if (b[1] == 0.0) nrerror("Error 1 in tridag");
        u[1]=r[1]/(bet=b[1]);
        for (j=2;j<=n;j++) {
                gam[j]=c[j-1]/bet;
                bet=b[j]-a[j]*gam[j];
                if (bet == 0.0) nrerror("Error 2 in tridag");
                u[j]=(r[j]-a[j]*u[j-1])/bet;
        }
        for (j=(n-1);j>=1;j--)
                u[j] -= gam[j+1]*u[j+1];
        free_vector(gam,1,n);
}
```

Useimmat harvojen matriisien käsittelyyn tarkoitetut algoritmit ja ohjelmat käyttävät edelläesitettyjä menetelmiä, mutta siten viritettyinä, että ne minimoivat turhat laskuoperaatiot ja *fill-in*-ilmiön (nolla-alkio muuttuu laskun aikana nollasta poikkeavaksi, jolloin sille on varattava tilaa).

Matriisin harvuutta voi olla monelaista. Alla esimerkkejä.



Kuva 3.4: Harvoja matriiseja (*Numerical recipes, kuva 2.7.1*).

Harvojen matriisien tallettamiseen on luonnollisesti olemassa monia tapoja. Koska kuitenkin tarvitaan jonkinlainen indeksoitu menetelmä joudutaan käyttämään muistia enemmän kuin vain nollasta eroavien alkioita vastaava määrä.

Eräs tapa on seuraava:

> $N \times N$ -matriisin tallettamiseen tarvitaan kaksi yksiulotteista taulukkoa `sa` ja `ija`. Ensinmainittuun (`float`) talletetaan alkiot ja toiseen (`int`) indeksit. Talletussäännöt ovat seuraavat:
>
> i. `sa`:n N ensimmäistä alkiota sisältävät matriisin diagonaalielementit.
>
> ii. `ija`:n N ensimmäistä alkiota kertoo `sa`:n alkion, josta löytyy matriisin rivin ensimmäinen nollasta poikkeava ei-diagonaalielementti. Jos riviltä ei tällaista löydy asetetaan `ija`:n alkio yhtä suuremmaksi kuin edellisen rivin alkion indeksi.
>
> iii. `ija`:m alkio 1 on aina $N + 2$. Tästä saadaan selville matriisin koko.

iv. `ija`:n alkio $N+1$ on aina yhtä suurempi kuin viimeisen rivin viimeisen nollasta poikkeavan alkion indeksi `sa`:ssa. Tästä voidaan laskea matriisin nollasta poikkeavien alkioiden lukumäärä. Alkio $N+1$ `sa`:ssa ei ole käytössä.

v. `sa`:n alkiot $N+2$:sta eteenpäin sisältävät matriisin ei-diagonaalialkiot rivijärjestyksessä ja jokaisen rivin sisällä sarakejärjestyksessä.

vi. `ija`:n alkiot $N+2$:sta eteenpäin sisältävät `sa`:n vastaavan alkion sarakenumeron.

Esimerkiksi matriisi

$$\begin{bmatrix} 3. & 0. & 1. & 0. & 0. \\ 0. & 4. & 0. & 0. & 0. \\ 0. & 7. & 5. & 9. & 0. \\ 0. & 0. & 0. & 0. & 0. \\ 0. & 0. & 0. & 6. & 5. \end{bmatrix} \tag{3.50}$$

talletetaan seuraavasti:

k	1	2	3	4	5	6	7	8	9	10	11
ija(k)	7	8	8	10	11	12	3	2	4	5	4
sa(k)	3.	4.	5.	0.	5.	X	1.	7.	9.	2.	6.

NR:n ohjelmista löytyvät rutiinit, joilla ylläolevaa talletustapaa käyttäen voidaan käsitellä harvoja matriiseja (nimeltään `sprs*`).

Lisäksi `matlab`issa on mahdollista käsitellä harvoja matriiseja ja NAG- ja IMSL-kirjastoissa on rutiineja harvoille matriiseille.

3.7 Muita matriisityyppejä

On olemassa lukuisa joukko matriiseja, joiden erityispiirteitä voidaan käyttää hyväksi yhtälöryhmän tai käänteismatriisin ratkaisussa.

Niitä ovat esimerkiksi Vandermonden ja Toeplitzin matriisit (Numerical Recipes, kappale 2.8), symmetriset ja positiivisesti definiitit matriisit (Cholesky hajotelma, NR 2.9) ja QR-hajotelma (NR 2.10).

Emme käy näitä tässä läpi, jokainen voi itseopiskella NR:n avulla.

4 Interpolointi

4.1 Yleistä

Interpolointi- tai ekstrapolointitehtävä on yleensä seuraavanlainen:

i. Meillä on N kappaletta datapisteitä (x_i, y_i), $i = 1, 2, \ldots, N$.

ii. Haluamme tietää y:n arvon pisteessä $x \neq x_i$, $i = 1, \ldots, N$.

Interpoloinnissa $x_1 < x < x_N$ ja ekstrapoloinnissa $x < x_1$ tai $x > x_N$. Ekstrapolointi on tietysti hyvin vaarallista puuhaa. Sitä tehdään lähinnä differentiaaliyhtälöiden ratkaisun yhteydessä.

Lisäksi datapisteissä voi olla mukana kohinaa, jolloin interpolaatiokäyrän tulisi kulkea tasaisesti pistejoukon läpi, ei välttämättä kaikkien pisteiden kautta.

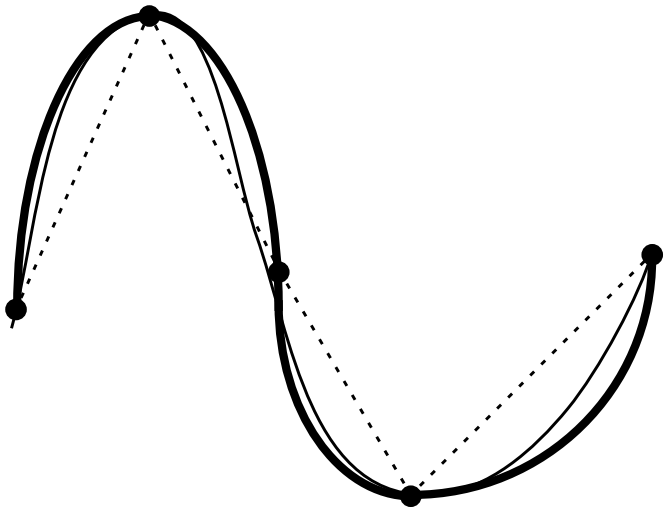
Kolmas sovellus on nopean approksimaation laskeminen jollekin funktiolle. Tällöin ‘datapisteitä’ on käytössä mielivaltainen määrä.

Usein approksimointiin käytetään pisteiden (tai tietyn osan pisteistä) kautta kulkevaa polynomia. Interpolointi tehdään käyttämällä hyväksi pisteen x lähiympäristöä. Tällöin interpolointikäyrän derivaatta on epäjatkuva, koska muutettaessa x:n arvoa jossain vaiheessa muuttuvat ne datapisteet, joiden perusteella interpolointi tehdään.

Ns. splinisovituksessa asetetaan lisäehdoksi myös derivaattojen jatkuvuus (tiettyyn asteeseen asti).

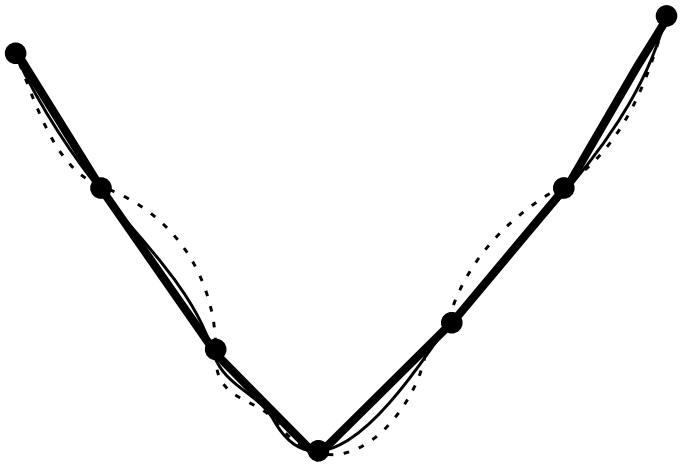
Polynomi-interpoloinnissa ei usein olla kiinnostuneita itse polynomin kertoimista vaan ainoastaan sen arvoista tietyissä pisteissä. Kertoimien laskeminen onkin paljon virhealttiimpaa kuin arvojen laskeminen.

Inerpoloinnissa käytettyjen pisteiden lukumäärä (miinus 1) on interpoloinnin aste. Sopiva aste riippuu interpoloitavan funktion tai pistejoukon ominaisuuksista. Alla esimerkkinä tasaisen funktion interpolointi.



Kuva 4.1: Tasaisen funktion interpolointi. Paksu viiva = alkuperäinen funktio, ohut viiva = korkea-asteinen interpoiointi, katkoviiva = matala-asteinen interpolointi.

Jos funktiossa on teräviä kulmia voi matala-asteinen interpolointi antaa paremman tuloksen.



Kuva 4.2: Teräväkulmaisen funktion interpolointi. Ohut viiva = matala-asteinen, katkoviiva = korkea-asteinen.

4.2 Polynomi-interpolointi

Olkoon meillä pistejoukko (x_i, y_i), $y_i = f(x_i)$, $i = 1, \ldots, N$. Tehtävänä on etsiä polynomi $P_{N-1}(x)$, joka toteuttaa ehdon

$$P_{N-1}(x_i) = y_i \ , \ i = 1, \ldots, N . \tag{4.1}$$

Voidaan helposti todistaa, että polynomi P_{N-1} on enintään astelukua $N-1$ ja että se on yksikäsitteinen kunhan kaikki x_i ovat erilaisia.

Suoraviivainen tapa olisi muodostaa yhtälöryhmä ja ratkaista se oppimillamme menetelmillä. Eli jos polynomi on muotoa

$$y = c_1 + c_2 x + c_3 x^2 + \ldots + c_N x^{N-1} \,, \tag{4.2}$$

saadaan ehdoista (4.1) yhtälöryhmä

$$\begin{cases} c_1 + c_2 x_1 + c_3 x_1^2 + \ldots + c_N x_1^{N-1} = y_1 \\ c_1 + c_2 x_2 + c_3 x_2^2 + \ldots + c_N x_2^{N-1} = y_2 \\ \qquad\qquad \ldots \\ c_1 + c_2 x_N + c_3 x_N^2 + \ldots + c_N x_N^{N-1} = y_N \end{cases} . \tag{4.3}$$

Matriisimuodossa

$$\begin{bmatrix} 1 & x_1 & x_1^2 & \ldots & x_1^{N-1} \\ 1 & x_2 & x_2^2 & \ldots & x_2^{N-1} \\ 1 & x_3 & x_3^2 & \ldots & x_3^{N-1} \\ \ldots & \ldots & \ldots & \ldots & \ldots \\ 1 & x_N & x_N^2 & \ldots & x_N^{N-1} \end{bmatrix} \cdot \begin{bmatrix} c_1 \\ c_2 \\ c_3 \\ \ldots \\ c_N \end{bmatrix} = \begin{bmatrix} y_1 \\ y_2 \\ y_3 \\ \ldots \\ y_N \end{bmatrix} . \tag{4.4}$$

Tuo neliömatriisi on ns. Vandermonden matriisi, ja se on hiukan huonosti käyttäytyvä (kuten harjoituksissa 2 ehkä huomasit).

Parempi tapa laskea polynomi P_{N-1} on ns. Lagrangen polynomien avulla. Olkoot polynomit $l_1, l_2, \ldots, l_N$ määritelty seuraavasti

$$l_i(x) = \prod_{\substack{j=1 \\ j \neq i}}^{N} \left(\frac{x - x_j}{x_i - x_j} \right) , \; i = 1, \ldots, N \,. \tag{4.5}$$

Funktioilla on ominaisuus:

$$l_i(x_j) = \delta_{ij} \text{ (Kroneckerin delta).} \tag{4.6}$$

Polynomi $P_{N-1}(x)$ voidaan nyt kirjoittaa muotoon

$$P_{N-1}(x) = \sum_{i=1}^{N} f(x_i) l_i(x) \ . \tag{4.7}$$

On helppo tarkistaa, että polynomi kulkee kaikkien pisteiden kautta. Koska l_i:t ovat enintään astetta $N-1$ on myös P_{N-1} enintään astelukua $N-1$.

Eli interpolointipolynomi saadaan muotoon (jätetään alaindeksi pois)

$$\begin{aligned} P(x) &= \frac{(x-x_2)(x-x_3)\ldots(x-x_N)}{(x_1-x_2)(x_1-x_3)\ldots(x_1-x_N)} y_1 \\ &+ \frac{(x-x_1)(x-x_3)\ldots(x-x_N)}{(x_2-x_1)(x_2-x_3)\ldots(x_2-x_N)} y_2 \\ &+ \ldots \\ &+ \frac{(x-x_1)(x-x_2)\ldots(x-x_{N-1})}{(x_N-x_1)(x_N-x_2)\ldots(x_N-x_{N-1})} y_N \end{aligned} \tag{4.8}$$

Periaatteessa polynomin voisi laskea suoraan kaavalla (4.8), mutta tehokkaampi menetelmä on ns. Nevillen algoritmi.

Olkoon P_1 nolla-asteisen, pisteen (x_1, y_1) kautta kulkevan polynomin arvo pisteessä x eli $P_1 = y_1$. Samalla tavoin määritellään polynomit $P_2, P_3, \ldots, P_N$.

Olkoon nyt P_{12} yksi-asteisen, pisteiden (x_1, y_1) ja (x_2, y_2) kautta kulkevan polynomin arvo pisteessä x. Samoin määritellään $P_{23}, P_{34}, \ldots, P_{(N-1)N}$. Edelleen määritellään korkeampiasteisia polynomeja, kunnes päästään polynomiin $P_{123\ldots N}$, joka on haluamamme interpolointipolynomi.

P:t voidaan kirjoittaa taulukkoon, jossa vasemmalla olevat ‘esi-isät’ johtavat yhteiseen ‘jälkeläiseen’ eli haluttuun polynomiin. Esim. $N = 3$:

$$
\begin{array}{lll}
x_1: & y_1 = P_1 & \\
 & & P_{12} \\
x_2: & y_2 = P_2 & P_{123} \\
 & & P_{23} \\
x_3: & y_3 = P_3 &
\end{array}
\tag{4.9}
$$

Nevillen algoritmi täyttää ylläolevaa taulukkoa vasemmalta oikealle, yksi sarake kerrallaan. Se perustuu rekursiokaavaan ‘vanhempien’ ja niiden jälkeläisten välillä:

$$
\begin{aligned}
& P_{i(i+1)\ldots(i+m)} \qquad . \\
& = \frac{(x - x_{i+m})P_{i(i+1)\ldots(i+m-1)} + (x_i - x)P_{(i+1)\ldots(i+m)}}{x_i - x_{i+m}}
\end{aligned}
\tag{4.10}
$$

Tämä rekursio toimii, koska vanhempipolynomeilla on samat arvot pisteissä $x_{i+1} \ldots x_{i+m-1}$.

Nevillen algoritmi perustuu Aitkenin lemmaan (hieman erilaiset merkinnät verrattuna edellä-esitettyyn):

Olkoon $P_{1n}(x)$ $(n-1)$-asteinen Lagrangen polynomi, joka toteuttaa ehdot

$$P_{1n}(x_i) = y_i,\ i = 1, \ldots, n\ . \tag{4.11}$$

Vastaavasti $P_{2,\,n+1}(x)$ on $(n-1)$-asteinen Lagrangen polynomi ja toteuttaa ehdot

$$P_{2,\,n+1}(x_i) = y_i,\ i = 2, \ldots, n+1\ . \tag{4.12}$$

Tällöin pistejoukossa (x_i, y_i), $i = 1, \ldots, n+1$ muodostettu polynomi $P_{1,\,n+1}(x)$ saadaan kaavasta

$$P_{1,\,n+1}(x) = \frac{(x - x_1)P_{2,\,n+1}(x) - (x - x_{n+1})P_{1n}(x)}{x_{n+1} - x_1}\ . \tag{4.13}$$

Tämähän on triviaalisti tosi, sillä $P_{1,\,n+1}(x)$ on n-asteinen polynomi, joka toteuttaa ehdot $P_{1,\,n+1}(x_i) = y_i$, $i = 1, \ldots, n+1$.

Hieman parempi tapa kuin laskea polynomin arvo suoraan kaavalla (4.10) on käyttää erotuksia $C_{m,\,i}$ ja $D_{m,\,i}$:

$$\begin{aligned} C_{m,\,i} &= P_{i\ldots(i+m)} - P_{i\ldots(i+m-1)} \\ D_{m,\,i} &= P_{i\ldots(i+m)} - P_{(i+1)\ldots(i+m)} \ . \end{aligned} \tag{4.14}$$

Näille erotuksille on helppo johtaa rekursiokaavat (4.10):n avulla:

$$\begin{aligned} D_{m+1,\,i} &= \frac{(x_{i+m+1} - x)(C_{m,\,i+1} - D_{m,\,i})}{x_i - x_{i+m+1}} \\ C_{m+1,\,i} &= \frac{(x_i - x)(C_{m,\,i+1} - D_{m,\,i})}{x_i - x_{i+m+1}} \qquad . \end{aligned} \tag{4.15}$$

Tasolla m korjaukset C ja D saattavat polynomin astetta korkeammaksi. Lopullinen polynomi $P_{1\ldots N}$ saadaan siis minkä tahansa y_i:n ja korjauksien C ja D summana, kun kuljetaan jotain polkua puussa (4.9) vasemmalta oikealle.

NR:n rutiini `polint` tekee tämän asian.

4.3 Rationaalifunktiointerpolointi

Usein pistejoukkoa tai funktiota ei ole mahdollista tyydyttävästi interpoloida polynomilla. Tämä tarkoittaa sitä, että esimerkiksi polynomi heilahtelee rajusti pisteiden välillä.

Tämän voi aiheuttaa usein se, että interpoloitavalla funktiolla on singulariteetti; ei välttämättä interpolointialueella eikä välttämättä edes reaalilukuakselilla.

Tässä tapauksessa kannattaa kokeilla rationaalifunktiota polynomin sijaan.

Samaan tapaan kuin polynomin tapauksessa määritelään rationaalifunktio $R_{i(i+1)\ldots(i+m)}$ siten, että se kulkee pisteiden (x_i, y_i), …, (x_{i+m}, y_{i+m}) kautta:

$$R_{i(i+1)\ldots(i+m)}(x) = \frac{P_\mu(x)}{Q_\nu(x)} = \frac{p_0 + p_1 x + \ldots + p_\mu x^\mu}{q_0 + q_1 x + \ldots + q_\nu x^\nu} \ . \tag{4.16}$$

Nevillen algoritmin tapaan voidaan johtaa rekursiokaava

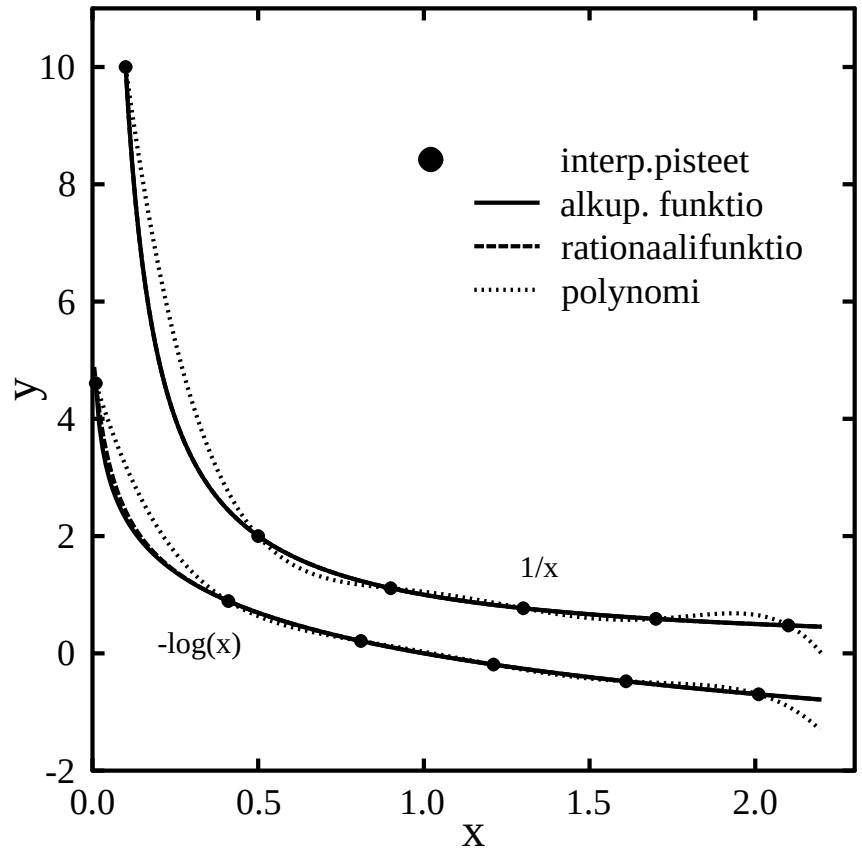
$$R_{i(i+1)\ldots(i+m)} = R_{(i+1)\ldots(i+m)} \tag{4.17}$$

$$+ \frac{R_{(i+1)\ldots(i+m)} - R_{i\ldots(i+m-1)}}{\left(\frac{x - x_i}{x - x_{i+m}}\right)\left(1 - \frac{R_{(i+1)\ldots(i+m)} - R_{i\ldots(i+m-1)}}{R_{(i+1)\ldots(i+m)} - R_{(i+1)\ldots(i+m-1)}}\right) - 1}$$

ja (4.15):n mukaiset korjaukset.

NR:n rutiini `ratint` tekee rationaalifunktiointerpoloinnin.

Alla pari esimerkkiä siitä, mihin rationaalifunktiointerpolointi soveltuu paremmin kuin polynomi-interpolointi.



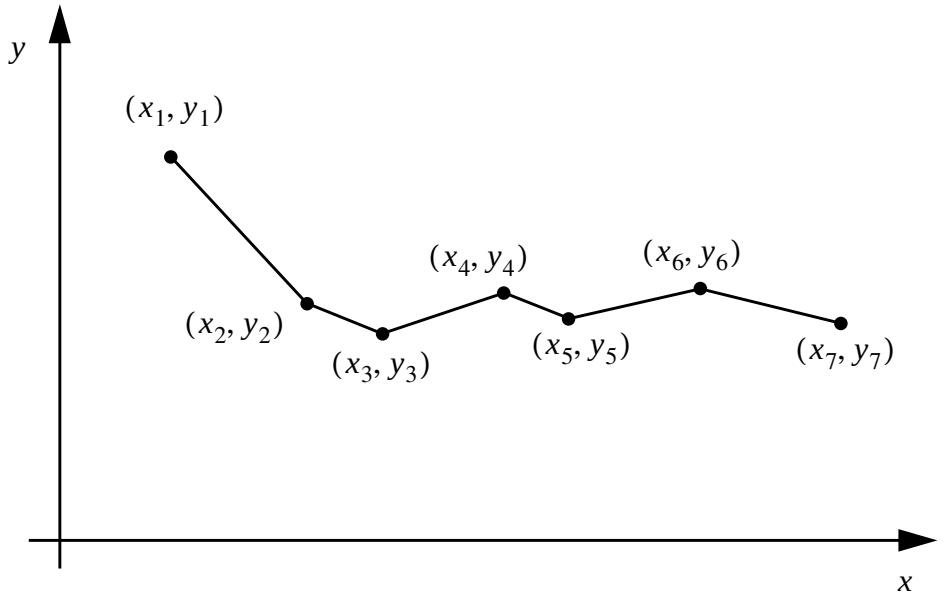
Kuva 4.3: Funktioiden $1/x$ ja $-\log x$ interpolointi rationaalifunktiolla ja polynomilla. Alkuperäinen funktio ja rationaali-interpolointi eivät juuri eroa toisistaan.

4.4 Splinifunktiot

Aikaisemmissa luvuissa koko datajoukkoa käytettiin interpolaatiofunktion laskemiseen. Varsinkin polynomien tapauksissa pisteiden lukumäärän (ja polynomin asteluvun) kasvaessa interpolointifunktio alkoi käyttäytyä villisti. Tästä syystä on usein edullista määrittää interpolointipolynomi tietyllä alueella käyttäen vain osaa pisteistä.

Splinifunktioksi kutsutaan tietyillä tasaisuusehdoilla yhteenliitetyistä palapolynomeista koostuvaa funktiota. Olkoon meillä taas pistejoukko $\{x_i, y_i\}$, $i = 1, \ldots, N$; lisäksi merkitään $y(x_i) = y_i$, $x_1 = a$, $x_N = b$. n-asteiseksi spliniksi kutsutaan palapolynomia, jonka $n-1$ ensimmäistä derivaattaa ovat jatkuvia pisteissä x_i.

Lineaarinen interpolaatio on ensimmäisen asteen splini:



Kuva 4.4: Lineaarinen interpolaatio

Yleisimmin käytetty on kuitenkin kuutiosplini. Sen käyttöä voidaan perustella 'energian minimointiominaisuudella': Kaikista välillä $[a, b]$ kahdesti jatkuvasti derivoituvista funktioista g, jotka toteuttavat ehdot $g(x_i) = y_i$, $g'(x_1) = y'_1$, $g'(x_N) = y'_N$ kuutiosplinillä S on pienin 'taivutusenergia' eli

$$\int_{x_1}^{x_N} [g''(x)]^2 dt \geq \int_{x_1}^{x_N} [S''(x)]^2 dt \ . \tag{4.18}$$

Tarkastellaan yksittäistä väliä $[x_j, x_{j+1}]$. Lineaarinen interpolointi tällä välillä antaa tuloksen

$$y = Ay_j + By_{j+1}, \tag{4.19}$$

missä

$$A = \frac{x_{j+1} - x}{x_{j+1} - x_j}\ ,\ B = 1 - A = \frac{x - x_j}{x_{j+1} - x_j}\ . \tag{4.20}$$

Solmupisteissä x_i, interpolointi (4.19) tuottaa epäjatkuvuuskohdan 1. derivaattaan ja lisäksi 2. derivaatta on määrittelemätön. Nyt haluamme muuttaa interpolointia siten, että 2. derivaatta jatkuva solmupisteissä. Olettakaamme, että käytössä on funktion arvojen lisäksi myös sen 2. derivaatan y'' arvot solmupisteissä y''_i.

Lisätään nyt (4.19):n oikealle puolelle kuutiotermi, jonka toinen derivaatta muuttuu lineaarisesti arvosta y''_j arvoon y''_{j+1}. Valitaan lisäksi tämä termi siten, että sen arvo on solmupisteissä nolla, jolloin se ei häiritse interpolointia.

Tämä voidaan saada aikaan seuraavalla muutoksella

$$y = Ay_j + By_{j+1} + Cy''_j + Dy''_{j+1}\ , \tag{4.21}$$

missä A ja B on määritelty yhtälössä (4.20) ja

$$\begin{aligned} C &= \frac{1}{6}(A^3 - A)(x_{j+1} - x_j)^2 \\ D &= \frac{1}{6}(B^3 - B)(x_{j+1} - x_j)^2\ . \end{aligned} \tag{4.22}$$

Derivoimalla (4.21) kahdesti x:n suhteen voidaan tarkistaa, että toinen derivaatta y'' todellakin on jatkuva solmupisteissä ja lineaarinen näiden välillä:

$$\begin{aligned} \frac{dy}{dx} = \frac{y_{j+1} - y_j}{x_{j+1} - x_j} &- \frac{3A^2 - 1}{6}(x_{j+1} - x_j)y''_j \\ &+ \frac{3B^2 - 1}{6}(x_{j+1} - x_j)y''_{j+1} \end{aligned}\ . \tag{4.23}$$

$$\frac{d^2y}{dx^2} = Ay''_j + By''_{j+1}\ . \tag{4.24}$$

Oletimme, että toiset derivaatat on tunnettu. Näinhän ei käytännössä ole; tarvitsemme siis lisäehtoja: 1. derivaattojen on oltava jatkuvia. Soveltamalla tätä ehtoa yhtälöön (4.23) pystymme laskemaan toiset derivaatat y''_j ja interpolointisplinin.

Eli asettamalla 1. derivaatta pisteessä x_j laskettuna välin $[x_{j-1}, x_j]$ splinillä samaksi kuin välin $[x_j, x_{j+1}]$ splinillä laskettu saadaan $N - 2$ yhtälöä ($j = 2, \ldots, N - 1$):

$$\frac{x_j - x_{j-1}}{6} y''_{j-1} + \frac{x_{j+1} - x_{j-1}}{3} y''_j + \frac{x_{j+1} - x_j}{6} y''_{j+1} \quad . \tag{4.25}$$
$$= \frac{y_{j+1} - y_j}{x_{j+1} - x_j} - \frac{y_j - y_{j-1}}{x_j - x_{j-1}}$$

Yhtälöryhmässä on $N-2$ yhtälöä, mutta N tuntematonta. Jotta ratkaisu olisi yksikäsitteinen tarvitaan vielä kaksi lisäehtoa. Yleisesti käytetään kahta tapaa:

i. Asetetaan $y''_1 = y''_N = 0$. Tästä syntyviä splinejä kutsutaan ns. luonnollisiksi splineiksi.

ii. Oletetaan 1. derivaatat alueen laidoilla tunnetuiksi: $y'_1 = c_1$, $y'_N = c_N$.

Yhtälöryhmä (4.25) on ns. tridiagonaalinen. Luonnollisten splinien tapauksessa se voidaan kirjoittaa muotoon

$$\begin{bmatrix} 1 & 0 & \dots & & & \\ h_1 & u_2 & h_2 & \dots & & \\ \dots & h_2 & u_3 & h_3 & \dots & \\ & & \dots & & & \\ & & \dots & h_{N-2} & u_{N-1} & h_{N-1} \\ & & & \dots & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} y''_1 \\ y''_2 \\ \dots \\ y''_{N-1} \\ y''_N \end{bmatrix} = \begin{bmatrix} 0 \\ v_2 \\ \dots \\ v_{N-1} \\ 0 \end{bmatrix}, \tag{4.26}$$

missä

$$h_i = \frac{x_{i+1} - x_i}{6}, \tag{4.27}$$

$$u_i = \frac{x_{i+1} - x_{i-1}}{3} \tag{4.28}$$

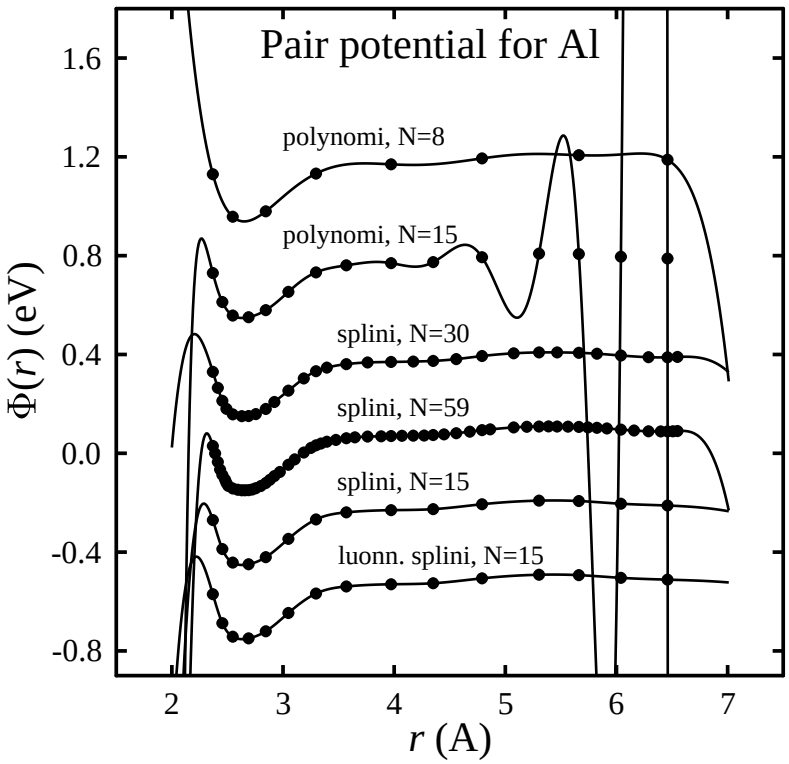
ja

$$v_i = \left(\frac{y_{i+1} - y_i}{x_{i+1} - x_i}\right) - \left(\frac{y_i - y_{i-1}}{x_i - x_{i-1}}\right) . \tag{4.29}$$

Edellämainitun kohdan ii. ehto saadaan yhtälöryhmään soveltamalla yhtälöä (4.23).

Tridiagonaalisen yhtälön ratkaiseminen on suoraviivainen tehtävä. NR:n rutiini `spline` tekee tämän. Sisäänmenoparametreina ovat pisteet (x_i, y_i) sekä lisäehtoina päätepisteiden 1. derivaatat tai vaihtoehtoisesti asetetaan näiden pisteiden 2. derivaatat nolliksi. Rutiini palauttaa toiset derivaatat y''_i, joiden perusteella [kaava (4.21)] lasketaan rutiinilla `splint` splinin arvo halutulla x:n arvolla.

Alla on esimerkkinä samaan pistejoukkoon sovitettu erilaisia polynomeja ja kuutiosplinejä. Kuva havainnollistaa polynomi-interpoloinnin heikkouksia suurilla pistemäärillä.



Kuva 4.5: Polynomi- ja splini-interpolointi. Alkuperäinen data on palloina ja kuvaa kahden alumiiniatomin keskinäistä potentiaalienergiaa niiden etäisyyden funktiona.

4.5 Kaksimuuttujaisen funktion interpolointi

Olkoon funktio y nyt riippuvainen kahdesta muuttujasta: $y = y(x_1, x_2)$. Data, jonka perusteella interpolointi tehdään on siis suorakulmaiseen hilaan asettuva pistejoukko $\{y_{ij}, x_1^{(i)}, x_2^{(j)}\}$, $i = 1, \ldots, m$, $j = 1, \ldots, n$. Tarkoituksena on tämän perusteella laskea y-arvo jollekin pisteelle (x_1, x_2).

Merkitään

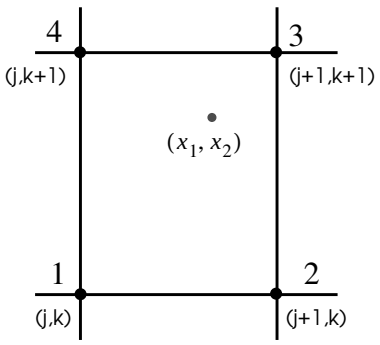
$$\texttt{ya(j,k)} = y(\texttt{x1a(j)}, \texttt{x2a(k)}) \; . \tag{4.30}$$

Ensin on tietysti löydettävä se hilan suorakulmio, jossa piste sijaitsee:

$$\begin{aligned} \texttt{x1a(j)} \le x_1 \le \texttt{x1a(j+1)} \\ \texttt{x2a(k)} \le x_2 \le \texttt{x2a(k+1)} \; . \end{aligned} \tag{4.31}$$

Merkitään lisäksi

$$\begin{aligned} y_1 &\equiv \texttt{ya(j,k)} \\ y_2 &\equiv \texttt{ya(j+1,k)} \\ y_3 &\equiv \texttt{ya(j+1.k+1)} \\ y_4 &\equiv \texttt{ya(j,k+1)} \; . \end{aligned} \tag{4.32}$$



Bilineaarisella interpolaatiolla haluttu y-arvo saadaan seuraavasti

$$\begin{aligned} t &= \frac{x_1 - \texttt{x1a(j)}}{\texttt{x1a(j+1)} - \texttt{x1a(j)}} \\ u &= \frac{x_2 - \texttt{x2a(k)}}{\texttt{x2a(k+1)} - \texttt{x2a(k)}} \; . \end{aligned} \tag{4.33}$$

$$y(x_1, x_2) = (1-t)(1-u)y_1 + t(1-u)y_2 + tuy_3 + (1-t)uy_4 \tag{4.34}$$

Tämä on hiukan karkea menetelmä. NR:ssä on kerrottu lisää, miten esimerkiksi voidaan käyttää kuutiosplinejä kaksiulotteisessa interpoloinnissa.

5 Numeerinen integrointi

5.1 Yleistä

Sähköopissa osoitetaan, että ympyränmuotoisen, r-säteisen virtasilmukan I aiheuttama magneettikenttä etäisyydellä x ympyrän keskipisteestä on $(0 \le x \le r)$

$$H(x) = \frac{4Ir}{r^2 - x^2} \int_0^{\pi/2} \left[1 - \left(\frac{x}{r} \right)^2 \sin^2(x) \right]^{1/2} d\theta \ .$$

Kun tunnemme x:n, r:n ja I:n voimme laskea magneettikentän. Tuo integraali vain on hankala ns. elliptinen integraali eikä se ratkea analyyttisesti. Se on integroitava numeerisesti.

Tässä kappaleessa perehdymme funktion numeeriseen integrointiin. Keskitymme lähinnä yksiulotteisten integraalien laskemiseen.

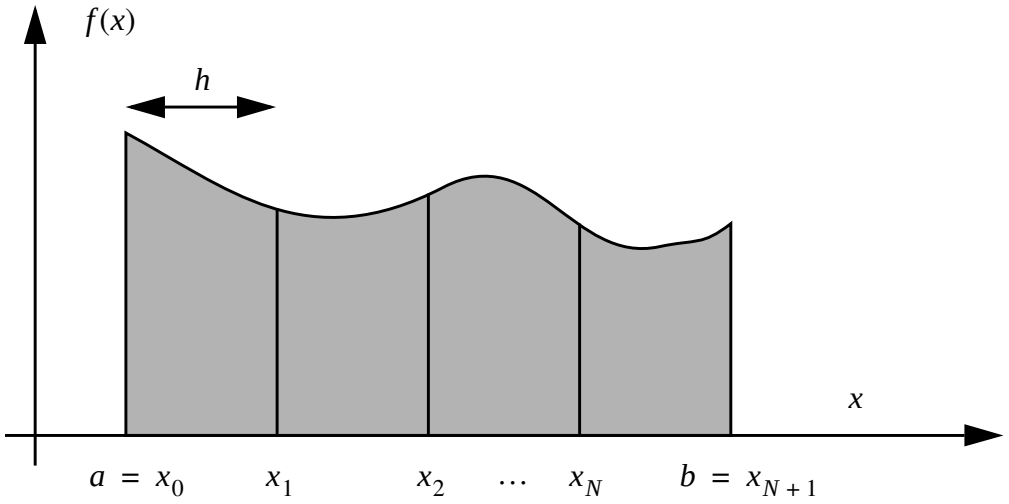
Useampiulotteisten integraalien lakeminen on varsin hankalaa. Usein käytetty menetelmä näille on Monte Carlo -integrointi. Itseasiassa voidaan osoittaa, että kun integraalin dimensio kasvaa yli neljän, käyttäytyy Monte Carlo -menetelmän virhe mielyttävämmin funktioevaluaatioiden lukumäärän funktiona kuin konventionaalisten menetelmien.

5.2 Puolisuunnikassääntö

Tutkitaan määrättyä integraalia

$$I = \int_a^b f(x)dx \ . \tag{5.1}$$

Tämähän on tietysti funktion f piirtämän käyrän ja x-akselin väliin jäävä pinta-ala.



Kuva 5.1: Funktion f integraali.

Jaetaan väli $[a, b]$ tasavälisiin pätkiin siten että

$$x_i = x_0 + ih\,,\ i = 0, 1, \ldots, N, N+1\ . \tag{5.2}$$

Funktion f arvot pisteissä x_i on tunnettu

$$f(x_i) = f_i\ . \tag{5.3}$$

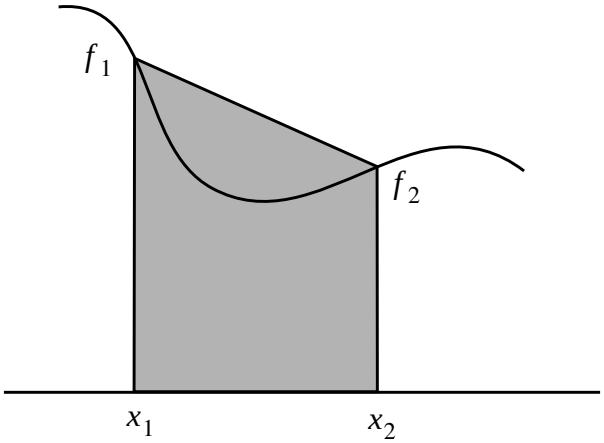
Integraali (5.1) lasketaan yleensä summana osista, jotka muodostetaan yhdestä tai useammasta osavälistä $[x_i, x_{i+1}]$:

$$\int_{x_i}^{x_{i+k}} f(x)dx\,. \tag{5.4}$$

Mitä enemmän osavälejä otetaan mukaan, sitä korkeampiasteiselle polynomille tulos on eksakti.

Yksinkertaisin mahdollinen on puolisuunnikassääntö

$$\int_{x_1}^{x_2} f(x)dx = h\left(\frac{1}{2}f_1 + \frac{1}{2}f_2\right) + O(h^3 f'')\ . \tag{5.5}$$



Kuva 5.2: Puolisuunnikassääntö.

Eli koko integraali tulee muotoon

$$I_T = \frac{h}{2}\sum_{i=0}^{N}[f_i + f_{i+1}] + O(h^2 f'') \quad . \tag{5.6}$$
$$= h\sum_{i=1}^{N} f_i + \frac{h}{2}[f_0 + f_{N+1}] + O(h^2 f'')$$

Tuo integraalin virhe on tarkemmin

$$I - I_T = -\frac{1}{12}(b-a)h^2 f''(\zeta) \ , \ \zeta \in [a, b] \ . \tag{5.7}$$

Tätä voidaan tietyissä tapauksissa käyttää arvioitaessa, miten pieni h:n tulee olla, jotta saavutetaan haluttu tarkkuus integraalissa.

Esimerkki:

Kuinka monta pistettä tarvitaan, kun halutaan laskea integraali

$$\int_0^1 e^{-x^2} dx \tag{5.8}$$

siten, että virhe on pienempi kuin 0.5×10^{-4} ?

Yhtälön (5.7) mukaan virhe on $-(h^2 f''(\zeta))/12$. Nyt $f(x) = \exp(-x^2)$, $f'(x) = -2x\exp(-x^2)$ ja $f''(x) = (4x^2 - 2)\exp(-x^2)$.

Välillä $[0, 1]$ pätee $|f''(x)| \le 2$ eli virhe on enintään $h^2/6$. Jotta virhe olisi vähemmän kuin 0.5×10^{-4} on oltava

$$\frac{1}{6}h^2 < 0.5\times10^{-4} \rightarrow h < 0.01732 \ .$$

Integrointipisteitä tarvitaan siis:

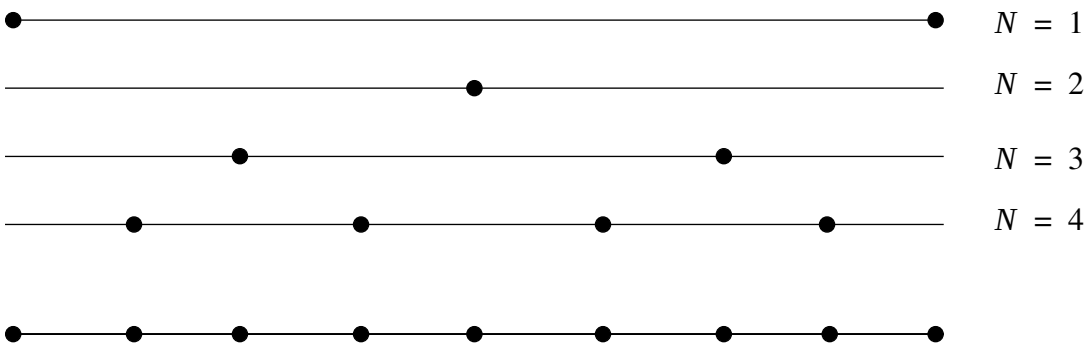
$$h = \frac{b-a}{n-1} = \frac{1}{n-1} \rightarrow n = \frac{1}{h} + 1 = 59 \text{ kappaletta.}$$

Aina ei tietenkään ole mahdollista arvioida f'':n suuruutta integrointivälillä.

Käytännössä virheen arviota ei yleensä tehdä tällä tavalla, vaan integraali lasketaan lisäämällä aina integrointipisteitä ja tarkastelemalla integraalin arvon suppenemista.

Kaavasta (5.6) huomataan, että siihen on helppo lisätä pisteitä siten, ettei jo laskettua informaatiota hävitetä. Eli integraali voidaan laskea iteroimalla: aloitetaan käyttämällä vain päätepisteitä ja jokaisella iteraatiokierroksella lisätään aikaisemmin käytettyjen pisteiden väliin uusia

pisteitä.



Kuva 5.3: Integraalin iteratiivinen laskeminen

NR:n aliohjelma `trapz` laskee integraalin perättäisiä iteraatioita.

Puolisuunnikassäännön käyttö sopii periaatteessa parhaiten funktioille, jotka eivät ole kovin tasaisia.

5.3 Korkeamman kertaluvun menetelmät

Ottamalla osavälin integraaliin (5.4) mukaan useampia pisteitä saadaan korkeamman kertaluvun menetelmiä.

Esimerkiksi kolmen pisteen tapauksessa saadaan Simpsonin sääntö

$$\int_{x_1}^{x_3} f(x)dx = h\left[\frac{1}{3}f_1 + \frac{4}{3}f_2 + \frac{1}{3}f_3\right] + O(h^5 f^{(4)}) \ . \tag{5.9}$$

Simpsonin sääntö voidaan johtaa seuraavasti:

Puolisuunnikassäännössä $f(x)$ oletettiin lineaariseksi integrointivälillä. Tässä tapauksessa oletetaan, että funktiota kuvaa hyvin 2. asteen polynomi pisteen x_2 ympäristössä:

$$f(x) = f_2 + \frac{f_3 - f_1}{2h}(x - x_2) + \frac{f_3 - 2f_2 + f_1}{2h^2}(x - x_2)^2 \quad . \tag{5.10}$$
$$+ O((x_2 - x)^3)$$

(Tämähän on Taylorin sarja x_2:n ympäristössä.) Tämä voidaan helposti integroida:

$$\int_{x_1}^{x_3} f(x)dx = \int_{x_1}^{x_3}\left[f_2 + \frac{f_3 - f_1}{2h}(x - x_2) + \frac{f_3 - 2f_2 + f_1}{2h^2}(x - x_2)^2\right]dx \qquad (5.11)$$
$$= \int_{-h}^{h}\left[f_2 + \frac{f_3 - f_1}{2h}x + \frac{f_3 - 2f_2 + f_1}{2h^2}x^2\right]dx$$

Summaamalla näitä integraalinpätkiä siten, etteivät ne mene toistensa kanssa päällekkäin, saadaan koko integraalille lauseke

$$\int_{x_1}^{x_N} f(x)dx \qquad (5.12)$$
$$= h\left[\frac{1}{3}f_1 + \frac{4}{3}f_2 + \frac{2}{3}f_3 + \frac{4}{3}f_4 + \ldots + \frac{2}{3}f_{N-2} + \frac{4}{3}f_{N-1} + \frac{1}{3}f_N\right]$$

Simpsonin menetelmää ei voi aivan suoraan käyttää integraalin iteratiiviseen laskemiseen, vaan tarvitsemme hieman aputuloksia.

Ns. Eulerin-Maclaurinin summakaava kertoo puolisuunnikassäännön virheen

$$\int_{x_1}^{x_N} f(x)dx = h\left[\frac{1}{2}f_1 + f_2 + f_3 + \ldots + f_{N-1} + \frac{1}{2}f_N\right] \quad , \qquad (5.13)$$
$$-\frac{B_2h^2}{2!}(f'_N - f'_1) - \ldots - \frac{B_{2k}h^{2k}}{(2k)!}(f_N^{(2k-1)} - f_1^{(2k-1)}) - \ldots$$

missä B_{2k} on Bernoullin luku. Tärkeää on huomata, että vain h:n parilliset potenssit esiintyvät kaavassa.

Oletetaan, että olemme laskeneet integraalille approksimaation S_N puolisuunnikassäännöllä (5.6) käyttäen N:ää pistettä ja tämän jälkeen approksimaation S_{2N} 2N pisteellä. Ensimmäinen virhetermi S_{2N}:ssä on neljäsosa S_N:n virhetermistä. Kun laskemme näiden 'painotetun keskiarvon'

$$S = \frac{4}{3}S_{2N} - \frac{1}{3}S_N \ , \qquad (5.14)$$

huomaamme, että ensimmäinen virhetermi $((B_2h^2)/2!)(f'_N - f'_1)$ häviää. Näinollen S:n virhe on $O(h^4)$ eli sama kuin Simpsonin menetelmällä. Itseasiassa kirjoittamalla auki yhtälö (5.14), päädytään täsmälleen Simpsonin sääntöön (5.12).

NR:n rutiini `qsimp` laskee integraalin tällä Simpsonin menetelmän variaatiolla.

5.4 Rombergin menetelmä

Ylläesitetty proseduuri voidaan yleistää myös korkeamman asteen integrointimenetelmiin. Perusidea on soveltaa puolisuunnikassääntöä peräkkäin k kertaa, jolloin ensimmäinen virhetermi kaavassa (5.13) on $O(h^{2k})$. Simpsonin menetelmälle $k = 2$.

NR:n rutiini `qromb` laskee integraalin halutulla arvolla k (Rombergin menetelmä). Siihen tutustutaan tarkemmin laskuharjoituksissa.

Rombergin menetelmä perustuu ns. Richardsonin ekstrapolointiin. Esitellään se lyhyesti tässä.

Otetaan esimerkiksi numeerinen derivointi. Ensimmäiseksi tulee mieleen erotusosamäärä

$$f'(x) = \frac{f(x+h) - f(x)}{h} \ . \tag{5.15}$$

Mikä on tämän kaavan virhe? Käytetään Taylorin teoreemaa:

$$f(x+h) = f(x) + h f'(h) + \frac{1}{2}h^2 f''(\xi) \ , \ \xi \in [x, x+h] \tag{5.16}$$

eli

$$f'(x) = \frac{f(x+h) - f(x)}{h} - \frac{1}{2}h f''(\xi) \ . \tag{5.17}$$

Virhe tässä on siis $O(h)$. Haluamme yleensä kuitenkin virheen, joka pienenee nopeammin h:n funktiona.

Katsotaanpa seuraavia Taylorin sarjoja

$$\begin{cases} f(x+h) = f(x) + h f'(x) + \frac{1}{2!}h^2 f''(x) + \frac{1}{3!}h^3 f'''(x) + \frac{1}{4!}h^4 f^{(4)}(x) + \dots \\ f(x-h) = f(x) - h f'(x) + \frac{1}{2!}h^2 f''(x) - \frac{1}{3!}h^3 f'''(x) + \frac{1}{4!}h^4 f^{(4)}(x) - \dots \end{cases} \tag{5.18}$$

Vähentämällä ne toisistaan saadaan

$$f(x+h) - f(x-h) = 2h f'(x) + \frac{2}{3!}h^3 f'''(x) + \frac{2}{5!}h^5 f^{(5)}(x) + \dots \tag{5.19}$$

eli saamme approksimaation derivaatalle

$$f'(x) = \frac{f(x+h) - f(x-h)}{2h} - \frac{h^2}{3!} f'''(x) - \frac{h^4}{5!} f^{(4)}(x) - \dots \ . \tag{5.20}$$

tai

$$f'(x) \approx \frac{f(x+h) - f(x-h)}{2h} \ . \tag{5.21}$$

Tämän virhe on $O(h^2)$.

Taylorin teoreeman antamassa muodossa nuo sarjat olisivat muotoa

$$\begin{cases} f(x+h) = f(x) + h f'(x) + \frac{1}{2} h^2 f''(x) + \frac{1}{6} h^3 f'''(\xi_1) \\ f(x-h) = f(x) - h f'(x) + \frac{1}{2} h^2 f''(x) - \frac{1}{6} h^3 f'''(\xi_2) \end{cases} \tag{5.22}$$

ja vähennys antaa tuloksen

$$f'(x) = \frac{f(x+h) - f(x-h)}{2h} - \frac{h^2}{6} \left[\frac{f'''(\xi_1) + f'''(\xi_2)}{2} \right] . \tag{5.23}$$

Tässä taas $\xi_1 \in [x, x+h]$ ja $\xi_2 \in [x-h, x]$. Voimme yksinkertaistaa tämän muotoon

$$f'(x) = \frac{f(x+h) - f(x-h)}{2h} - \frac{h^2}{6} f'''(\xi) \ , \tag{5.24}$$

missä $\xi \in [x-h, x+h]$.

Voimme kirjoittaa (5.20):n muotoon

$$f'(x) = \frac{f(x+h) - f(x-h)}{2h} + a_2 h^2 + a_4 h^4 + a_6 h^6 + \dots \ , \tag{5.25}$$

missä a_i riippuvat vain f:stä ja x:stä, eivät h:sta. Tähän perustuu Richardsonin ekstrapolointi.

Määritellään funktio ϕ seuraavasti (f ja x kiinteitä)

$$\phi(h) = \frac{f(x+h) - f(x-h)}{2h} \ . \tag{5.26}$$

Se on siis approksimaatio $f'(x)$:lle ja sen virhe on $O(h^2)$. Päämääränä on laskea raja-arvo

$$\lim_{h \to 0} \phi(h) = f'(x) \ . \tag{5.27}$$

Voimme siis laskea ϕ:n mille hyvänsä sarjalle h_n ja käyttää näitä derivaatan approksimaatioina. Fiksumpaa on kuitenkin laskea ϕ jollekin sarjalle käyttäen aina edellistä tulosta hyväksi.

Oletetaan, että olemme laskeneet approksimaatiot $\phi(h)$ ja $\phi(h/2)$. Kaavan (5.25) mukaan

$$\begin{aligned} \phi(h) &= f'(x) - a_2 h^2 - a_4 h^4 - a_6 h^6 - \dots \\ \phi(\frac{h}{2}) &= f'(x) - a_2\left(\frac{h}{2}\right)^2 - a_4\left(\frac{h}{2}\right)^4 - a_6\left(\frac{h}{2}\right)^6 - \dots \end{aligned} \quad . \tag{5.28}$$

Vähennetään alempi yhtälö 4:llä kerrottuna ylemmästä:

$$\phi(h) - 4\phi(\frac{h}{2}) = -3f(x) - \frac{3}{4}a_4 h^4 - \frac{15}{16}a_6 h^6 - \dots \ , \tag{5.29}$$

josta saadaan edelleen

$$\phi(\frac{h}{2}) + \frac{1}{3}\left[\phi(\frac{h}{2}) - \phi(h)\right] = f'(x) + \frac{1}{4}a_4 h^4 + \frac{5}{16}a_6 h^6 + \dots \ . \tag{5.30}$$

Ja, hokkus-pokkus, vain lisäämällä termin $(1/3)[\phi(h/2) - \phi(h)]$ $\phi(h/2)$:een olemme saaneet tarkemman arvion derivaatalle f'. Voimme jatkaa tätä proseduuria poistamalla aina yhä korkeampiasteisia virhetermejä. Tätä kutsutaan Richardsonin ekstrapoloinniksi (RE). (Ekstrapolointia mielessä $h \to 0$.)

Yleisemmin RE tapahtuu seuraavasti: Olkoon ϕ funktio, jolle pätee

$$\phi(h) = L - \sum_{k=1}^{\infty} a_{2k} h^{2k} \ . \tag{5.31}$$

Kertoimia a_{2k} ei tunneta. Tarkoituksena on approksimoida suuretta L käyttäen funktiota ϕ eli halutaan laskea raja-arvo $L = \lim_{x \to 0}\phi(x)$.

Valitaan h:lle jokin arvo ja lasketaan luvut

$$D(n, 1) = \phi(\frac{h}{2^n}), \ n = 1, 2, \dots \ . \tag{5.32}$$

Yhtälön (5.31) mukaan

$$D(n, 1) = L + \sum_{k=1}^{\infty} A(k, 1)\left(\frac{h}{2^n}\right)^{2k} \ , \tag{5.33}$$

missä $A(k, 1) = -a_{2k}$. Luvut $D(n, 1)$ antavat karkean arvion haluamallemme suureelle L. Tarkempi arvio saadaan RE:llä:

$$D(n, m+1) = \frac{4^m}{4^m - 1} D(n, m) - \frac{1}{4^m - 1} D(n-1, m) \ . \tag{5.34}$$

Voidaan osoittaa, että suureille $D(n, m)$ pätee seuraava kaava

$$D(n, m) = L + \sum_{k=m}^{\infty} A(k, m) \left(\frac{h}{2^n} \right)^{2k} \ . \tag{5.35}$$

Olennaista on, että summaus k:n yli alkaa termistä m, jolloin ensimmäinen virhetermi on $O(h^{2m})$.

Luvut $D(n, m)$ voidaan järjestää kolmioon kuvan mukaisesti.

$$\begin{array}{lllll} D(1,1) & & & & \\ D(2,1) & D(2,2) & & & \\ D(3,1) & D(3,2) & D(3,3) & & \\ \ldots & \ldots & \ldots & & \\ D(N,1) & D(N,2) & D(N,3) & \ldots & D(N,N) \end{array} \tag{5.36}$$

Käytännössä approksimaation laskeminen tapahtuu seuraavasti:

i. Kirjoita aliohjelma funktiolle ϕ .

ii. Valitse sopivat arvot suureille N ja h.

iii. Laske $D(i, 1) = \phi(h/2^i)$, $i = 1, 2, \ldots, N$.

iv. Laske $D(i, j+1) = D(i, j) + (4^j - 1)^{-1}[D(i, j) - D(i-1, j)]$, $1 \le j \le i - 1 \le N - 1$.

Miten tästä päästään Rombergin menetelmään?

Muodostetaan sarja lukuja $R(n, m)$, jotka ovat approksimaatioita integraalille $\int_a^b f(x)dx$.

$$\begin{array}{lllll} R(1,1) & & & & \\ R(2,1) & R(2,2) & & & \\ R(3,1) & R(3,2) & R(3,3) & & \\ \ldots & \ldots & \ldots & & \\ R(N,1) & R(N,2) & R(N,3) & \ldots & R(N,N) \end{array} \quad . \tag{5.37}$$

Ensimmäinen sarake sisältää puolisuunnikassäännöllä lasketut approksimaatiot: $R(n+1, 1)$

on laskettu käyttäen 2^n integrointiväliä. Eli ensimmäinen on

$$R(1,1) = \frac{1}{2}(b-a)[f(a)+f(b)] \ . \tag{5.38}$$

Toinen termi saadaan soveltamalla puolisuunnikassääntöä kahdesti:

$$\begin{aligned} R(2,1) &= \frac{1}{4}(b-a)\left[f(a)+f(\frac{a+b}{2})\right] + \frac{1}{4}(b-a)\left[f(\frac{a+b}{2})+f(b)\right] \\ &= \frac{1}{4}(b-a)[f(a)+f(b)] + \frac{1}{2}(b-a)f(\frac{a+b}{2}) \\ &= \frac{1}{2}R(1,1) + \frac{1}{2}(b-a)f(\frac{a+2}{2}) \end{aligned} \tag{5.39}$$

Yleisesti voidaan todistaa rekursiokaava

$$R(n+1,1) = \frac{1}{2}R(n,1) + h\sum_{k=1}^{2^{n-1}} f(a+(2k-1)h) \ , \tag{5.40}$$

missä $h = (b-a)/2^n$ ja $n \geq 1$.
Mentäessä seuraaviin sarakkeisiin saadaan luvut R rekursiokaavan avulla

$$R(n+1,m+1) = R(n+1,m) + \frac{1}{4^m-1}[R(n+1,m)-R(n,m)] \ . \tag{5.41}$$

Esimerkiksi, jos $R(5,3) = 8$ ja $R(4,3) = 1$, mitä on $R(5,4)$?

$$R(5,4) = R(5,3) + \frac{1}{63}[R(5,3)-R(4,3)] = 8 + \frac{1}{63}(8-1) = 8\frac{1}{9} \ .$$

Kaava (5.41) voidaan perustella edelläesitettyyn Richardsonin ekstrapolointiin ja puolisuunnikassäännön virheeseen (5.13) perustuen. Kaavan (5.13) voimme kirjoittaa muodossa (2^{n-1} integrointijakoväliä)

$$\int_a^b f(x)dx = R(n,1) + a_2h^2 + a_4h^4 + a_6h^6 + \ldots , \tag{5.42}$$

missä $h = (a-b)/2^{n-1}$, $R(n,1)$ puolisuunnikassäännön antama arvio integraalille. Kertoimet a_i riippuvat funktiosta f, mutta eivät jakovälistä h. Puolittamalla jakoväli (eli $h \to h/2$ ja $n \to n+1$) saadaan

$$\int_a^b f(x)dx \;=\; R(n+1,\,1) + \frac{1}{4}a_{\,2}h^2 + \frac{1}{16}a_4h^4 + \frac{1}{64}a_6h^6 + \ldots \;. \tag{5.43}$$

Vähentämällä 4×[yhtälö (5.43)] yhtälöstä (5.42) saadaan

$$\int_a^b f(x)dx \;=\; R(n+1,\,2) - \frac{1}{4}a_4h^4 - \frac{5}{16}a_6h^6 - \ldots \;, \tag{5.44}$$

missä

$$R(n+1,\,2) \;=\; R(n+1,\,1) + \frac{1}{3}[R(n+1,\,1) - R(n,\,1)] \;. \tag{5.45}$$

Tämä oli siis tapaus $m = 1$ rekursiokaavassa (5.41). Tästä voidaan jatkaa tapaukseen $m = 2$, soveltamalla kaavaa (5.44) hieman muutettuna $(n+1 \to n;\; h \to 2h)$ ja vähentämällä tämä yhtälöstä (5.44) siten, että termi h^4 häviää:

$$\int_a^b f(x)dx \;=\; R(n+1,\,3) - \frac{1}{4^3}a_6h^6 - \frac{21}{4^5}a_8h^8 - \ldots \;, \tag{5.46}$$

missä

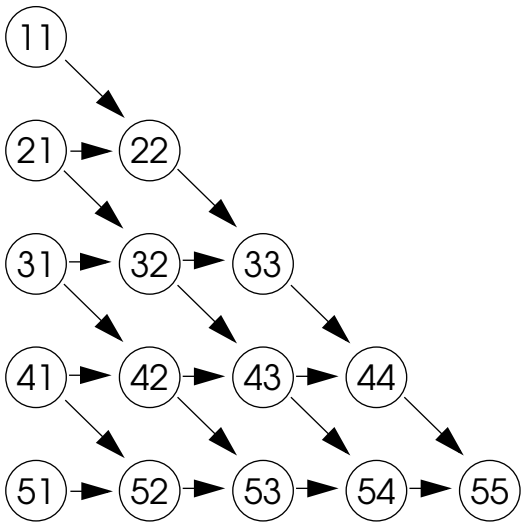
$$R(n+1,\,3) \;=\; R(n+1,\,2) + \frac{1}{15}[R(n+1,\,2) - R(n,\,2)] \;,\; n \geq 2 \;. \tag{5.47}$$

Rekursiokaava (5.41) arvolla $m = 2$ antaa tämän saman tuloksen. Eli rekursiokaava näyttää pätevän (emme sitä tässä todista...).

Näillä eväillä voimme helposti kirjoittaa algoritmin Rombergin integrointimenetelmälle.

i. Aluksi lasketaan 1. sarake. Sen ensimmäinen termi on
$R(1,\,1) \;=\; (1/2)(b-a)[\,f(a) - f(b)]$. Loput termit lasketaan kaavalla (5.40).

ii. Seuraavat sarakkeet lasketaan edellisistä käyttäen rekursiokaavaa (5.41).



Kuva 5.4: Termien $R(n, m)$ keskinäinen riippuvuus.

Termejä voi tietysti laskea myös sarake kerrallaan. Jos halutaan tulokselle jokin tietty tarkkuus, voidaan tarkastella peräkkäisten termien erotusta ja laskea termejä niin pitkälle, että erotus menee tarpeeksi pieneksi.

NR:n rutiini `qromb` ei laske integraalia täsmälleen tällä tavalla, vaan apuna käytetään integraalin polynomi-ekstrapolointia h:na arvolle nolla.

Eli kirjoitamme integraalin muodossa

$$\int_a^b f(x)dx \;=\; a_0 + a_2h^2 + a_4h^4 + a_6h^6 + \ldots \;, \tag{5.48}$$

Arvio integraalille on siis h:n funktio:

$$\int_a^b f(x)dx \equiv D(h) \tag{5.49}$$

ja haluamamme tulos on raja-arvo

$$\lim_{h \to 0} D(h) \;=\; a_0 \;. \tag{5.50}$$

Ideana on laskea integraali usealla h:n arvolla (kuten edelläkin) ja polynomi-ekstrapoloinnilla arvioida tuota raja-arvoa.

Näillä eväillä NR:n aliohjelman `qromb` periaatteen pitäisi aueta. Aliohjelman kutsu on muotoa:

```
float qromb(float (*func)(float), float a, float b)
```

5.5 Gaussin kvadratuurit

Edelläesitetyissä menetelmissä integraali laskettiin tasaisesti jakautuneista pisteiden f-arvoista painottamalla niitä sopivasti. Mitä enemmän sallimme vapausasteita näiden kertoimien määräämisessä, sitä korkeamman kertaluvun menetelmiin päästiin.

Gaussin kvadratuurin idea on sallia noiden pisteiden valinta optimaalisella tavalla. 'Vapausasteiden' lukumäärä kahdentuu ja periaatteessa menetelmän kertaluku on kaksinkertainen edellämainittujen menetelmiin samalla funktioevaluaatioiden lukumäärällä.

Täytyy tietysti aina muistaa, että korkea kertaluku ei aina takaa suurta tarkkuutta. Integroitavan funktion on oltava enemmän tai vähemmän tasainen, jotta päästään suureen tarkkuuteen.

Voidaan osoittaa, että pisteiden paikkojen vapauttaminen yleistää edelläesitettyjen menetelmien ominaisuutta, jonka mukaan tietyn kertaluvun menetelmä on tarkka saman kertaluvun polynomille.

Gaussin kvadratuurissa voimme valita pisteiden paikat ja niiden painot siten, että menetelmä antaa tarkan integraalin funktiolle (polynomi)×(jokin tunnettu funktio $W(x)$). Eli tarkemmin sanoen

$$\int_a^b W(x) f(x) dx \approx \sum_{j=1}^{N} w_j f(x_j) \tag{5.51}$$

on tarkka, jos $f(x)$ on polynomi. Funktion $W(x)$ avulla voidaan poistaa integrandista singulariteetteja.

Esimerkiksi integraalia

$$\int_{-1}^{1} \frac{\exp(-\cos^2 x)}{\sqrt{1-x^2}} dx \tag{5.52}$$

laskiessa kannattaisi valita funktio W seuraavasti

$$W(x) = \frac{1}{\sqrt{1-x^2}} \; . \tag{5.53}$$

Painofunktion ei tarvitse olla eksplisiittisesti näkyvissä: merkitään $g(x) = W(x) f(x)$ ja $v_j = w_j / W(x_j)$, jolloin (5.51) tulee muotoon

$$\int_a^b g(x)dx = \sum_{j=1}^{N} v_j g(x_j) \ . \tag{5.54}$$

Ongelmana on tietysti pisteiden paikkojen ja painojen valinta. Yksi tapa käyttää menetelmää on etsiä taulukkokirjoista tietyille painofunktioille lasketut pisteiden paikat ja painot ja käyttää yhtälöä (5.51) integraalin laskemiseen.

Koska Gaussin kvadratuuria käyttämällä voidaan päästä paljon vähemmällä laskemisella[1], käydään hieman tarkemmin läpi tämä menetelmä.

Otetaan yksinkertainen esimerkki: integraali välillä $[0, 1]$, kun painofunktio $W(x) = 1$.

$$\int_{-1}^{1} f(x)dx \approx \sum_{j=1}^{N} w_j f(x_j) \ . \tag{5.55}$$

On määrättävä painot w_i ja solmupisteet x_i valitaan siten, että virhe

$$E_N(f) = \int_{-1}^{1} f(x)dx - \sum_{j=1}^{N} w_j f(x_j) \tag{5.56}$$

on nolla mahdollisimman suuriasteiselle polynomille. Huomaa, että painokertoimet ja solmupisteet riippuvat myös luvusta N: $w_{j,N}$, $x_{j,N}$. Jätetään tuo toinen indeksi kuitenkin merkitsemättä. Ensinnäkin pätee

$$\begin{aligned} &E_N(a_0 + a_1 x + \ldots + a_m x^m) \\ &= a_0 E_N(1) + a_1 E_N(x) + \ldots + a_m E_N(x^m) \end{aligned} \ . \tag{5.57}$$

Siten $E_N(f) = 0$ jokaiselle alle $m + 1$ -asteiselle polynomille ainoastaan, jos

$$E_N(x^i) = 0 \ , \ i = 0, 1, \ldots, m \ . \tag{5.58}$$

Olkoon $N = 1$. Tällöin meillä on kaksi parametria w_1 ja x_1, joilta vaaditaan

$$E_N(1) = 0 \, , \ E_N(x) = 0 \ .$$

Eli toisin sanoen

1. Siis tietokone voi päästä....

$$\int_{-1}^{1} 1\,dx - w_1 = 0 \ , \ \int_{-1}^{1} x\,dx - w_1 x_1 = 0$$

$$\rightarrow w_1 = 2 \ , \ x_1 = 0$$

Eli

$$\int_{-1}^{1} f(x)\,dx \approx 2\,f(0) \ .$$

Olkoon $N = 2$. Parametreja on nyt neljä: w_1, w_2, x_1 ja x_2. Nämä voidaan ratkaista, kun kirjoitetaan ehto (5.58) i:n arvoon 3 saakka:

$$E_N(x^i) = \int_{-1}^{1} x^i dx - [w_1 x_1^i + w_2 x_2^i] = 0, \ i = 0, 1, 2, 3$$

eli

$$\begin{cases} w_1 + w_2 = 2 \\ w_1 x_1 + w_2 x_2 = 0 \\ w_1 x_1^2 + w_2 x_2^2 = \frac{2}{3} \\ w_1 x_1^3 + w_2 x_2^3 = 0 \end{cases} .$$

Ratkaisu on

$$\begin{cases} w_1 = 1 \\ w_2 = 1 \\ x_1 = -\frac{\sqrt{3}}{3} \\ x_2 = \frac{\sqrt{3}}{3} \end{cases} .$$

Eli integraalin approksimaatio on

$$\int_{-1}^{1} f(x)dx \approx f(-\frac{\sqrt{3}}{3}) + f(\frac{\sqrt{3}}{3}) \ .$$

Tämän approksimaation kertaluku on kolme. Simpsonin säännöllä saadaan sama tarkkuus käyttäen kolmea pistettä.

Mielivaltaisen N:n tapauksessa parametreja on $2N$, ja ne voidaan ratkaista vaatimalla, että (5.55) antaa tarkan arvon integraalille polynomeille, joiden asteluku on $2N-1$. Yhtälöt ovat siis

$$E_N(x^i) = 0 \ , \ i = 0, 1, 2, \ldots, 2N-1 \tag{5.59}$$

tai

$$\sum_{j=1}^{N} w_j x_j^i = \begin{cases} 0, i = 1, 3, 5, \ldots, 2N-1 \\ \frac{2}{i+1}, i = 0, 2, 4, \ldots, 2N-2 \end{cases} . \tag{5.60}$$

Näiden yhtälöiden ratkaiseminen ei tässä muodossa ole helppo tehtävä, vaan parametrien määrittämiseen käytetään yleensä muita menetelmiä.

Useimmiten ne perustuvat ortogonaalisiin polynomeihin. Niistä todennäköisesti puhumme myöhemmin enemmän, mutta käydään nyt lyhyesti läpi tämä menetelmä.

Olkoon tarkasteltava väli $[a, b]$. Määritellään kahden funktion sisätulo tällä välillä ja käyttäen painofunktiota $W(x)$

$$\langle f|g\rangle \equiv \int_a^b W(x) f(x) g(x) dx \ . \tag{5.61}$$

Kaksi funktiota ovat ortogonaaliset, jos

$$\langle f|g\rangle = 0 , \tag{5.62}$$

ja funktio on normitettu, jos

$$\langle f|f\rangle = 1 . \tag{5.63}$$

Funktiojoukkoa, jonka jäsenet ovat keskenään ortogonaalisia ja jotka ovat normitettuja, kutsutaan ortonormaaliksi.

On olemassa algoritmi, jolla pystytään konstruoimaan polynomijoukko, jossa on täsmälleen

yksi polynomi asteluvulla j, $p_j(x)$, $j = 0, 1, 2, \ldots$ ja jotka ovat keskenään ortogonaalisia tietyllä painofunktiolla $W(x)$. Algoritmi on seuraavanlainen:

i. $p_{-1}(x) \equiv 0$; $p_0(x) \equiv 1$

ii. $p_{j+1}(x) = (x - a_j)p_j(x) - b_j p_{j-1}(x)$, $j = 0, 1, 2, \ldots$,
missä

$$a_j = \frac{\langle x p_j | p_j \rangle}{\langle p_j | p_j \rangle} \ , \ j = 0, 1, 2, \ldots$$

$$b_j = \frac{\langle p_j | p_j \rangle}{\langle p_{j-1} | p_{j-1} \rangle} \ , \ j = 1, 2, 3, \ldots \ .$$

Näiden polynomien korkeimman asteen termin kerroin on aina ykkönen. Polynomit saadaan kuitenkin haluttaessa normitettua jakamalla ne termillä $[\langle p_j | p_j \rangle]^{1/2}$.

Jokaisella polynomilla $p_j(x)$ on välillä $[a, b]$ täsmälleen j erillistä nollakohtaa. Lisäksi polynomien $p_j(x)$ ja $p_{j+1}(x)$ nollakohdat osuvat lomittain: kahden jälkimmäisen polynomin nollakohdan välissä on aina täsmälleen yksi edellisen nollakohta.

Gaussin kvadratuuriin nämä asiat liittyvät seuraavasti: Ortogonaalisten polynomien (painolla $W(x)$) nollakohdat ovat juuri nuo Gaussin kvadratuurin solmupisteet kaavassa (5.51) (samalla painolla $W(x)$). Todistus sivuutetaan. Halukkaat voivat katsoa sen alan kirjallisuudesta[1].

Painokertoimet w_j saadaan yhtälöistä

$$\begin{cases} w_1 p_0(x_1) + w_2 p_0(x_2) + \ldots + w_N p_0(x_N) = \int_a^b W(x) p_0(x) dx \\ w_1 p_1(x_1) + w_2 p_1(x_2) + \ldots + w_N p_1(x_N) = \int_a^b W(x) p_1(x) dx \\ \ldots \\ w_1 p_{N-1}(x_1) + w_2 p_{N-1}(x_2) + \ldots + w_N p_{N-1}(x_N) = \int_a^b W(x) p_{N-1}(x) dx \end{cases} \tag{5.64}$$

Huomaa, että $\int_a^b W(x) p_j(x) dx = 0$, kun $j > 0$, koska $p_0(x)$ on vakio ja polynomit $p_j(x)$ ovat ortogonaalisia.

Voidaan vielä osoittaa, että näillä solmupisteillä ja painokertoimilla myös seuraavat polynomit $p_k(x)$, $k = N, N+1, \ldots, 2N-1$ integroidaan tarkasti kaavalla (5.51).

Useinkaan ei käytetä yhtälöryhmää (5.64) kertoimien määrittämiseen, vaan ne lasketaan kaavasta (ks. viite 1., s. 69)

1. Esimerkiksi K. E. Atkinson, *An Introduction to Numerical Analysis*, (John-Wiley & Sons, 1989), 2. laitos, kappale 5.3,

$$w_j = \frac{\langle p_{N-1}|p_{N-1}\rangle}{p_{N-1}(x_j)p'_N(x_j)} \ . \tag{5.65}$$

Gaussin kvadratuurin laskemisessa on siis kolme vaihetta:

i. Ortogonaalisten polynomien generointi sivun 69 algoritmilla.

ii. Polynomien nollakohtien määrittäminen.

iii. Painokertoimien määrittäminen.

Kirjallisuudesta löytyy joillekin usein esiintyville painofunktioille taulukoituna solmupisteitä ja painokertoimia. Näitä ovat mm.

Gauss-Legendre:

$$\begin{aligned} &W(x) = 1 \\ &-1 < x < 1 \\ &(j+1)P_{j+1} = (2j+1)xP_j - jP_{j-1} \end{aligned} \quad , \tag{5.66}$$

Gauss-Tsebysev:

$$\begin{aligned} &W(x) = \frac{1}{\sqrt{1-x^2}} \\ &-1 < x < 1 \\ &T_{j+1} = 2xT_j - T_{j-1} \end{aligned} \quad , \tag{5.67}$$

Gauss-Laguerre:

$$\begin{aligned} &W(x) = x^{\alpha}e^{-x} \\ &0 < x < \infty \\ &(j+1)L^{\alpha}_{j+1} = (-x+2j+\alpha+1)L^{\alpha}_j - (j+\alpha)L^{\alpha}_{j-1} \end{aligned} \quad , \tag{5.68}$$

Gauss-Hermite:

$$\begin{aligned} &W(x) = e^{-x^2} \\ &-\infty < x < \infty \qquad\qquad \text{j} \\ &H_{j+1} = 2xH_j - 2jH_{j-1} \end{aligned} \tag{5.69}$$

Gauss-Jacobi:

$$\begin{aligned} W(x) &= (1-x)^{\alpha}(1+x)^{\beta} \\ -1 &< x < 1 \\ c_j P_{j+1}^{(\alpha,\beta)} &= (d_j + e_j x) P_j^{(\alpha,\beta)} - f_j P_{j-1}^{(\alpha,\beta)} \end{aligned} \quad , \tag{5.70}$$

missä kertoimet c_j, d_j, e_j ja f_j saadaan kaavoilla

$$\begin{aligned} c_j &= 2(j+1)(j+\alpha+\beta+1)(2j+\alpha+\beta) \\ d_j &= (2j+\alpha+\beta+1)(\alpha^2-\beta^2) \\ e_j &= (2j+\alpha+\beta)(2j+\alpha+\beta+1)(2j+\alpha+\beta+2) \\ f_j &= 2(j+\alpha)(j+\beta)(2j+\alpha+\beta+2) \end{aligned} \quad . \tag{5.71}$$

Vastaavat NR:n rutiinit ovat `gauleg`, `gaulag`, `gauher`, `gaujac`.

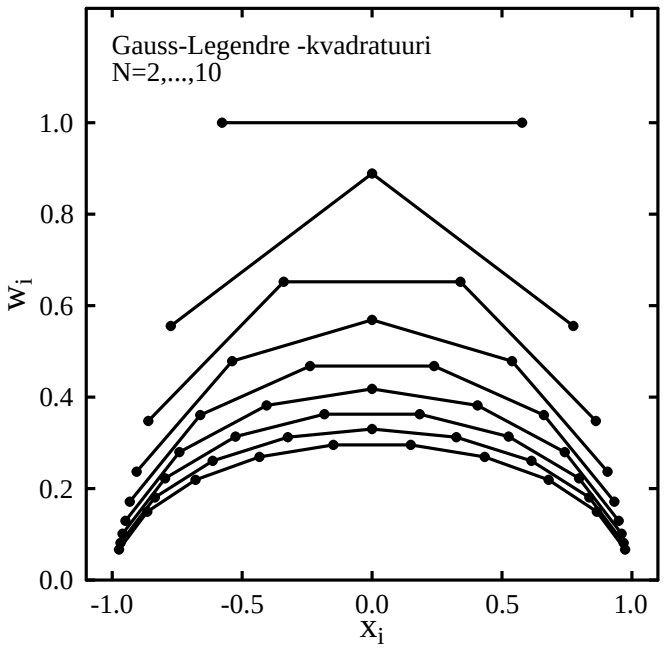
Tsebysevin polynomeille ei erillistä rutiinia tarvita, sillä solmupisteet ja painot saadaan yksinkertaisista kaavoista:

$$\begin{aligned} x_j &= \cos\left[\frac{\pi\left(j-\frac{1}{2}\right)}{N}\right] \\ w_j &= \frac{\pi}{N} \end{aligned} \quad . \tag{5.72}$$

Kun solmupisteet ja painokertoimet ovat tiedossa, itse integraalin arvo sitten lasketaan yksinkertaisesti

$$\int_a^b f(x)dx = \sum_{i=1}^{N} w_i f(x_i) \ . \tag{5.73}$$

Integrointirajat voidaan tietysti muuttaa muuttujan vaihdolla (esim. $[-1, 1] \rightarrow [a, b]$).
Alla esimerkkinä NR:n rutiinilla `gauleg` lasketut Gauss-Legendren kvadratuurin solmupisteet ja painot.



Kuva 5.5: Gauss-Legendre -kvadratuurin solmupisteet ja painot kertaluvun arvoilla $N = 2, 3, \ldots, 10$.

Gaussin kvadratuurin virhettä voidaan karkeasti arvioida laskemalla peräkkäisten (solmupisteitä N ja $N + n$, missä $n = 2$ esimerkiksi) summien erotusta.

Gaussin kvadratuurin miinuspuoli on se, että seuraavalla iteraatiokierroksella (solmupisteitä yksi enemmän) ei voida käyttää hyväksi edellisen kierroksen tuloksia. On kehitetty menetelmiä, joissa tät haittaa ei ole. Toisaalta, useimmiten Gaussin kvadratuuri on niin tehokas menetelmä, että tästä ei ole käytännössä haittaa.

Seuraavalla sivulla on lopuksi verrattu eri menetelmillä laskettuna integraalia

$$\int_0^{\pi} e^x \cos x \, dx = -\frac{(e^{\pi} + 1)}{2} \approx -12.070346$$

Table 5.1 Trapezoidal rule for evaluating (5.1.11)

n	I_n	E_n	Ratio	$\tilde{E}_n$
2	−17.389259	5.32		4.96
			4.20	
4	−13.336023	1.27		1.24
			4.06	
8	−12.382162	3.12E − 1		3.10E − 1
			4.02	
16	−12.148004	7.77E − 2		7.76E − 2
			4.00	
32	−12.089742	1.94E − 2		1.94E − 2
			4.00	
64	−12.075194	4.85E − 3		4.85E − 3
			4.00	
128	−12.071558	1.21E − 3		1.21E − 3
			4.00	
256	−12.070649	3.03E − 4		3.03E − 4
			4.00	
512	−12.070422	7.57E − 5		7.57E − 5

Table 5.3 Simpson's rule for evaluating (5.1.11)

n	I_n	E_n	Ratio	$\tilde{E}_n$
2	−11.5928395534	−4.78E − 1		−1.63
			5.59	
4	−11.9849440198	−8.54E − 2		−1.02E − 1
			14.9	
8	−12.0642089572	−6.14E − 3		−6.38E − 3
			15.5	
16	−12.0699513233	−3.95E − 4		−3.99E − 4
			15.9	
32	−12.0703214561	−2.49E − 5		−2.49E − 5
			16.0	
64	−12.0703447599	−1.56E − 6		−1.56E − 6
			16.0	
128	−12.0703462191	−9.73E − 8		−9.73E − 8

Table 5.17 Example of Romberg integration

k	Nodes	$J_k(f)$	Error
0	2	−34.77851866026	2.27E + 1
1	3	−11.59283955342	−4.78E − 1
2	5	−12.01108431754	−5.93E − 2
3	9	−12.07042041287	7.41E − 5
4	17	−12.07034720873	8.92E − 7
5	33	−12.07034631632	−6.82E − 11
6	65	−12.07034631639	< 5.00E − 12

Table 5.11 Gaussian quadrature for (5.1.11)

n	I_n	$I - I_n$
2	−12.33621046570	2.66E − 1
3	−12.12742045017	5.71E − 2
4	−12.07018949029	−1.57E − 4
5	−12.07032853589	−1.78E − 5
6	−12.07034633110	1.47E − 8
7	−12.07034631753	1.14E − 9
8	−12.07034631639	−4.25E − 13

Edellä olemme puhuneet vain yleisesti funktion integroinnista. Usein funktio on esimerkiksi mittaus- tai simulaatiodataa, jolle ei tiedetä funktionaalista muotoa.

Tällöin integrointi on parasta tehdä laskemalla datajoukolle interpolointifunktio (polynomi tai kuutiosplini) ja integroimalla tämä funktio.

Jos taas datassa on kohinaa, ei kannata sovittaa funktiota menemään pisteiden kautta, vaan on käytettävä jonkinlaista datan tasoitusta. Tämä taas datan sovitukseen liittyvä tehtävä, josta puhumme tuonnempana.

5.6 Moniulotteiset integraalit

Moniulotteisten integraalien laskeminen on hieman hankala tehtävä. Ensinnäkin funktion arvon laskujen lukumäärä voi nopeasti nousta hyvin suureksi. Toiseksi integrointialue voi olla hyvinkin monimutkainen.

Jotkut moniulotteiset integraalit pystytään redusoimaan yksiulotteisiksi. Esimerkiksi ns. iteroidut integraalit voidaan lausua yksiulotteisina:

$$\int_0^x dt_n \int_0^{t_n} dt_{n-1} \cdots \int_0^{t_3} dt_2 \int_0^{t_2} f(t_1)\,dt_1 = \frac{1}{(n-1)!}\int_0^x (x-t)^{n-1} f(t)\,dt \quad . \tag{5.74}$$

Jos integrointialue on monimutkainen ja integrandi taas tasainen on Monte Carlo -menetelmä varteenotettava vaihtoehto. Siitä puhumme myöhemmin.

Jos alue on suhteellisen yksinkertainen ja integrandi tasainen, voidaan käyttää peräkkäisiä yksiulotteisia integraaleja. Kolmiulotteinen integraali tulee nyt muotoon:

$$\begin{aligned} I &= \iiint dx\,dy\,dx\, f(x,y,z) \qquad . \\ &= \int_{x_1}^{x_2} dx \int_{y_1(x)}^{y_2(x)} dy \int_{z_1(x,y)}^{z_2(x,y)} dz\, f(x,y,z) \end{aligned} \tag{5.75}$$

Esimerkiksi funktion f integrointi yli yksikköympyrän on

$$\int_{-1}^{1} dx \int_{-\sqrt{1-x^2}}^{\sqrt{1-x^2}} dy\, f(x,y) \quad . \tag{5.76}$$

Merkitään kaavan (5.75) sisintä integraalia $G(x, y)$:llä

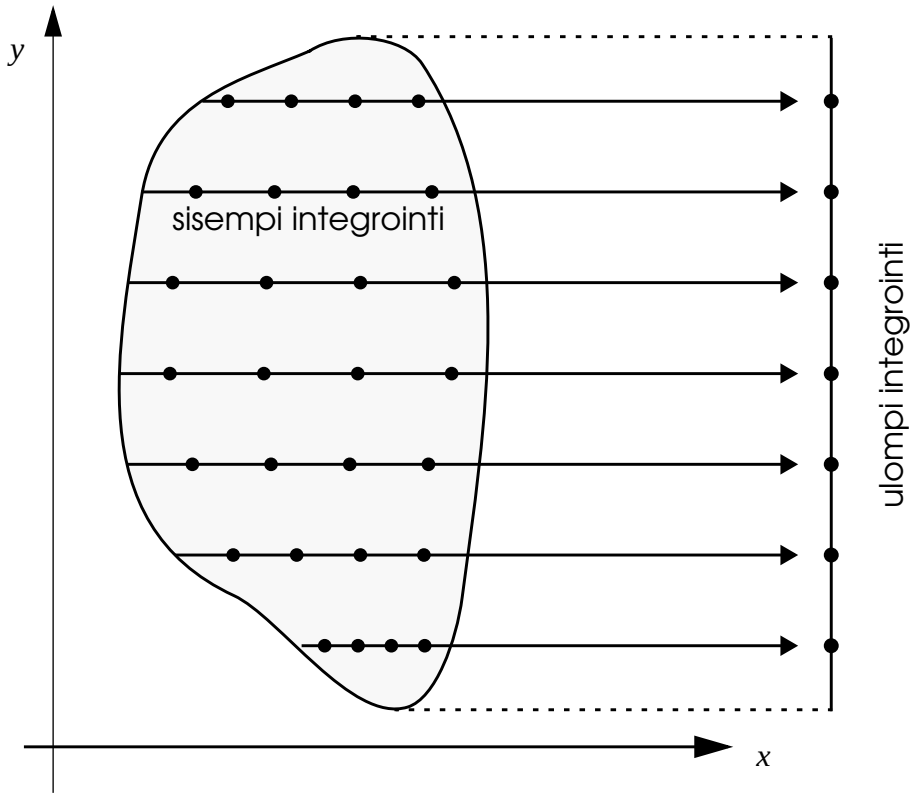
$$G(x, y) = \int_{z_1(x, y)}^{z_2(x, y)} dz\, f(x, y, z) \tag{5.77}$$

ja sen integraalia $H(x)$:llä

$$H(x) = \int_{y_1(x)}^{y_2(x)} G(x, y) dy \ . \tag{5.78}$$

Lopullinen tulos on tämän integraali yli välin $[x_1, x_2]$

$$I = \int_{x_1}^{x_2} H(x) dx \ . \tag{5.79}$$



Kuva 5.6: Kaksiulotteinen integrointi.

C-kielellä tämän voisi toteuttaa seuraavasti. Olkoon aliohjelma `qgaus` rutiini, joka laskee yhden muuttujan funktion integraalin halutulla välillä. Tässä esimerkissä käytetään kymmenenpisteen Gauss-Legendere-kvadratuuria:

```
float qgaus(float (*func)(float), float a, float b)
{
 int j;
 float xr,xm,dx,s;
 static float x[]={0.0,0.1488743389,0.4333953941,
     0.6794095682,0.8650633666,0.9739065285};
 static float w[]={0.0,0.2955242247,0.2692667193,
     0.2190863625,0.1494513491,0.0666713443};

 xm=0.5*(b+a);
 xr=0.5*(b-a);
 s=0;
 for (j=1;j<=5;j++) {
     dx=xr*x[j];
     s += w[j]*((*func)(xm+dx)+(*func)(xm-dx));
 }
 return s *= xr;
}
```

Itse 3d-integrointirutiini on `quad3d` (integroitava funktio on `func`):

```
static float xsav,ysav;
static float (*nrfunc)(float,float,float);

float quad3d(float (*func)(float, float, float), float x1,
float x2)
{
     float qgaus(float (*func)(float), float a, float b);
     float f1(float x);

     nrfunc=func;
     return qgaus(f1,x1,x2);
}

float f1(float x)
{
     float qgaus(float (*func)(float), float a, float b);
     float f2(float y);
     float yy1(float),yy2(float);

     xsav=x;
     return qgaus(f2,yy1(x),yy2(x));
}

float f2(float y)
{
     float qgaus(float (*func)(float), float a, float b);
     float f3(float z);
     float z1(float,float),z2(float,float);

     ysav=y;
     return qgaus(f3,z1(xsav,y),z2(xsav,y));
}

float f3(float z)
{
     return (*nrfunc)(xsav,ysav,z);
}
```

Ja alla pääohjelma, joka käyttää tätä rutiinia:

```
/* Driver for routine quad3d */
#include <stdio.h>
#include <math.h>
#define NRANSI
#include “nr.h”
#define PI 3.1415927
#define NVAL 10

static float xmax;
float func(float x,float y,float z)
{
     return x*x+y*y+z*z;
}
float z1(float x,float y)
{
     return (float) -sqrt(xmax*xmax-x*x-y*y);
}
float z2(float x,float y)
{
     return (float) sqrt(xmax*xmax-x*x-y*y);
}
float yy1(float x)
{
     return (float) -sqrt(xmax*xmax-x*x);
}
float yy2(float x)
{
     return (float) sqrt(xmax*xmax-x*x);
}

int main(void)
{
 int i;
 float xmin,s;

 printf(“Integral of r^2 over a spherical volume\n\n”);
 printf(“%13s %10s %11s\n”,”radius”,”QUAD3D”,”Actual”);
 for (i=1;i<=NVAL;i++) {
     xmax=0.1*i;
     xmin = -xmax;
     s=quad3d(func,xmin,xmax);
     printf(“%12.2f %12.6f %11.6f\n”,
     xmax,s,4.0*PI*pow(xmax,5.0)/5.0);
 }
 return 0;
}
```

Tämä ohjelma laskee funktion $f(x, y, z) = r^2$ ($r = \sqrt{x^2 + y^2 + z^2}$) integraalin yli kolmiulotteisen R-säteisen pallon. Tarkka arvo on $(4/5)\pi R^5$.

Käännös ja ajo:

```
3d> cc -o xquad3d xquad3d.c quad3d.c qgaus.c -lm
xquad3d.c:
cc: Error: nr.h, line 157: In this declaration, the type of "erff" is not
compatible with the type of a previous declaration of "erff" at line
number 272 in file /usr/include/math.h.
float erff(float x);
------^
cc: Error: nr.h, line 423: In this declaration, the type of "select" is
not compatible with the type of a previous declaration of "select" at line
number 231 in file /usr/include/sys/select.h.
float select(unsigned long k, unsigned long n, float arr[]);
------^
quad3d.c:
qgaus.c:
3d>
```

Käännös tehtiin Digitalin Alphassa. Ongelmana oli se, että symbolit `erff` ja `select` on jo systeemin puolesta määritelty. Ratkaisu on kommentoida näiden symboleiden määrittelyt pois tiedostosta `nr.h`.

```
3d> cc -o xquad3d xquad3d.c quad3d.c qgaus.c -lm
xquad3d.c:
quad3d.c:
qgaus.c:
3d> xquad3d
Integral of r^2 over a spherical volume

       radius     QUAD3D      Actual
        0.10     0.000025    0.000025
        0.20     0.000805    0.000804
        0.30     0.006112    0.006107
        0.40     0.025756    0.025736
        0.50     0.078601    0.078540
        0.60     0.195583    0.195432
        0.70     0.422733    0.422406
        0.80     0.824187    0.823550
        0.90     1.485211    1.484063
        1.00     2.515218    2.513274
3d>
```

Huomataan, että vaikka käytettiin peräti kymmenen pisteen Gauss-Legendre-kvadratuuria, on lopputuloksen virhe huomattava.

Toinen menetelmä useampiulotteisten integraalien laskemiseksi on ns. tulosääntö. Tarkastellaan kaksiulotteista integraalia

$$I = \iint_D f(x,y)\,dA\,. \tag{5.80}$$

Yksiulotteisen integraalin tapaan voisi yrittää etsiä muotoa

$$I = \sum_{i=1}^{N} w_i f(x_i, y_i) + R_N \tag{5.81}$$

olevaa approksimaatiota (R_N on approksimaation virhe), joka integroi tarkasti kaksimuuttujaiset polynomit, joiden asteluku on enintään d, missä polynomi on lineaarikombinaatio termeistä $x^p y^q$, ja $p+q \le d$. Esimerkiksi, kun $d = 2$, on polynomi muotoa

$$a_{00} + a_{10}x + a_{01}y + a_{20}x^2 + a_{11}xy + a_{02}y^2\ .$$

Yksi tapa (vaikkakaan ei välttämättä optimaalinen) tehdä tämä on ns. tulosääntö. Merkitään

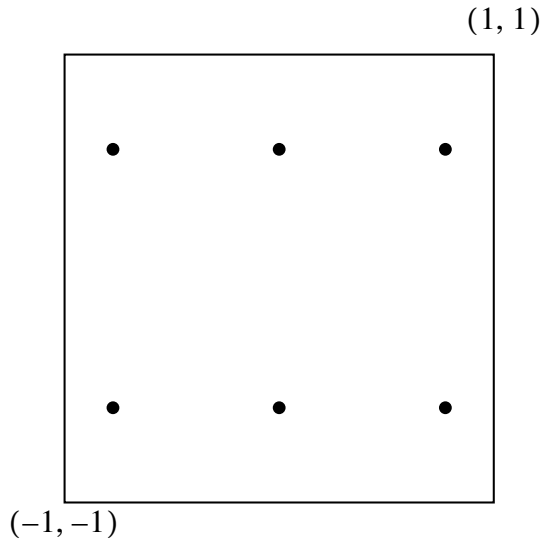
$$\int_\alpha^\beta f(x)dx = \sum_{i=1}^{N} w_i f(x_i) + R_1 \tag{5.82}$$

$$\int_a^b g(y)dy = \sum_{i=1}^{M} p_i g(y_i) + R_2\ . \tag{5.83}$$

Tällöin kaksiulotteinen integraali voidaan kirjoittaa muodossa

$$\begin{aligned}
\int_a^b\int_\alpha^\beta f(x,y)dxdy &= \int_a^b \left[\sum_{i=1}^{N} w_i f(x_i,y) + R_1\right]dy \quad . \\
&= \sum_{i=1}^{N}\left[w_i\int_a^b f(x_i,y)dy\right] + \int_a^b R_1 dx \\
&= \sum_{i=1}^{N} w_i\left[\sum_{j=1}^{M} p_j f(x_i,y_j) + R_2\right] + \int_a^b R_1 dx \\
&= \sum_{i=1}^{N}\sum_{j=1}^{M} w_i p_j f(x_i,y_j) + \sum_{i=1}^{N} w_i R_2 + \int_a^b R_1 dx \\
&= \sum_{i=1}^{N}\sum_{j=1}^{M} w_i p_j f(x_i,y_j) + R
\end{aligned} \tag{5.84}$$

Tämä on ns. tulosääntö. Siinä käytetään NM pistettä integraalin laskemiseen. Kuvassa 5.7 on esitetty yksinkertainen esimerkki.



Kuva 5.7: Kaksi- ja kolmisolmuisista Gauss-Legendre-kvadratuurista koostuva tulosääntö.

Voidaan osoittaa, että jos yksiulotteiset yhtälöt (5.82) ja (5.83) ovat tarkkoja polynomeille, joiden asteluku on enintään d_1 ja d_2, tulosääntökaava integroi tarkasti polynomit $x^p y^q$, jossa $p \leq d_1$ ja $q \leq d_2$.

Otetaan esimerkiksi funktion

$$f(x, y) = xye^{-x^2 y} \tag{5.85}$$

integraali yli neliön $x, y \in [0, 1]$. Tarkka arvo on

$$\int_0^1 \int_0^1 xye^{-x^2 y} dxdy = \frac{1}{2e} \approx 0.18393972 \ . \tag{5.86}$$

Käytetään Gauss-Legendre-kvadratuuria ja NR:n aliohjelmaa `gauleg`, joka laskee solmupisteet ja painokertoimet. Pääohjelma on listattu seuraavalla sivulla (lisäksi tarvitaan tiedostot `nr.h`, `nrutil.h`, `nrutil.c` ja `gauleg.c`:

```
#include <stdio.h>
#include <math.h>
#include <stdlib.h>
#define NRANSI
#include “nr.h”
#include “nrutil.h”

float f(float x, float y)
{
  return x*y*exp(-x*x*y);
  /* return sin(5.0*x*x)*cos(y*x); */
}

int main(int argc, char **argv)
{
  int nx,ny,i,j;
  float x1,x2,y1,y2;
  float s;
  float *xx,*wx,*xy,*wy;

  if (argc!=7 ) {
      fprintf(stderr,
         ”Komentorivi: %s nx ny x1 x2 y1 y2\n”,argv[0]);
      return (1);
  }
  nx=atoi(*++argv); ny=atoi(*++argv);
  x1=atof(*++argv); x2=atof(*++argv);
  y1=atof(*++argv); y2=atof(*++argv);

  xx=vector(1,nx);
  wx=vector(1,nx);
  xy=vector(1,ny);
  wy=vector(1,ny);

  gauleg(x1,x2,xx,wx,nx);
  gauleg(y1,y2,xy,wy,ny);
  s=0.0;
  for (i=1;i<=nx;i++)
      for (j=1;j<=ny;j++)
          s+=wx[i]*wy[j]*f(xx[i],xy[j]);
  printf(“GL-kvadratuuri %20.12g\n”,s);
  return (0);
 }
 #undef NRANSI
```

Käännös ja ajo:

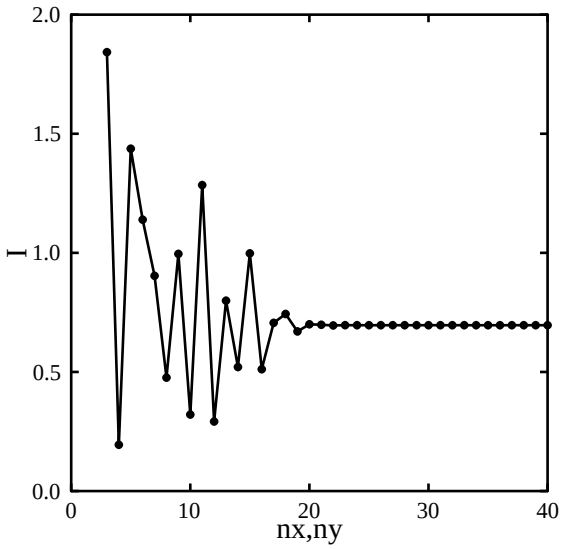
```
2d> cc -o xgauleg xgauleg.c gauleg.c nrutil.c -lm
xgauleg.c:
gauleg.c:
nrutil.c:
2d> xgauleg 3 3 0 1 0 1
GL-kvadratuuri       0.183959037066
Tarkka arvo          0.183939725161
2d> xgauleg 5 5 0 1 0 1
GL-kvadratuuri       0.183939725161
Tarkka arvo          0.183939725161
2d>
```

Eli GL suoriutuu varsin hyvin tästä integraalista. Toisaalta, funktio on hyvin tasainen ja integrointialue säännöllinen. Lisäksi on muistettava, että integroitava funktio lasketaan nx×ny kertaa.

Hankalampi tapaus on on integraali

$$I = \int_0^3 \int_0^3 \sin(5x^2)\cos(xy)\,dy\,dx \ . \tag{5.87}$$

Koska integrandi oskilloi integrointialueella melkoisesti, tarvitaan paljon solmupisteitä, jotta saadaan luotettava arvio integraalille.



Kuva 5.8: Integraali (5.87) laskettuna tulosäännöllä käyttäen Gauss-Legendre-kvadratuuria. Integraali on piirretty solmupisteiden lukumäärän funktiona.

Usein integrointialue on jotain muuta kuin suorakaide. Monissa tapauksissa alue voidaan sopivalla muuttujanvaihdolla saattaa suorakaiteeksi. Esimerkiksi kaksiulotteinen integraali yli kolmion

$$I = \iint_{\Delta} f(x,y)dA \quad , \qquad \Delta = \{(x,y)\colon 0 \le x \le 1, 0 \le y \le x\} \quad . \tag{5.88}$$

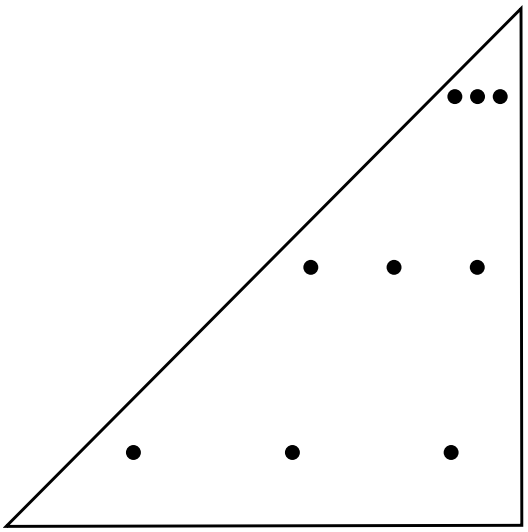
Tehdään muuttujanvaihdos

$$u = x \ , \ v = \frac{y}{x} \ \rightarrow du = dx, \, dv = \frac{dy}{x}, \tag{5.89}$$

josta saadaan integraali muotoon

$$I = \int_0^1 dx \int_0^x dy \, f(x,y) = \int_0^1 du \int_0^1 dv \, f(x,y)u \ . \tag{5.90}$$

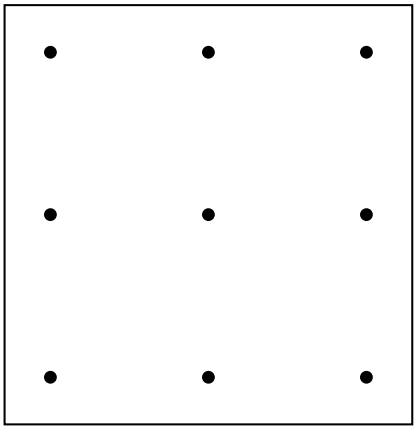
Nyt voidaan helposti soveltaa tulosääntöä tähän integraaliin. Ongelmia voi tuottaa se, että integrointialuetta ‘sämplätään’ epätasaisesti.



Kuva 5.9: Solmupisteiden sijainti, kun integrointialue on kolmio, joka muuttujanvaihdoksella saatetaan neliöksi.

On myös kehitetty menetelmiä, joilla integrointipisteet valitaan optimaalisemmin kuin tulosäännössä.

Esimerkiksi Gauss-Legendre-kvadratuurissa käyttäen 3×3 pistettä valitaan pisteet oheisen kuvan mukaan. Menetelmä on kertalukua 5 ja funktioevaluaatioita on 9 kappaletta.

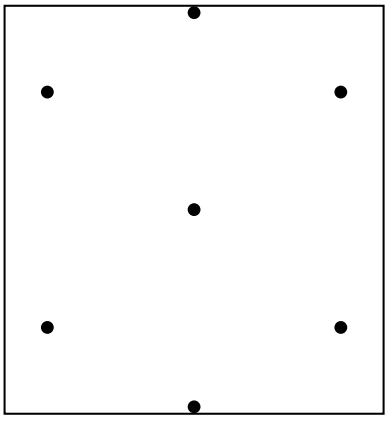


Kuva 5.10: Yksikköneliön integrointipisteet Gauss-Legendre kvadratuurissa käyttäen tulosääntöä ja 3×3 solmupistettä.

Ns. Radonin kaava antaa paremmin optimoidut solmupisteiden paikat. Esimerkiksi 7 pisteen menetelmässä

$$\int_{-1}^{1}\int_{-1}^{1} f(x,y)dxdy \approx \frac{5}{9}[f(r,s)+f(r,-s)+f(-r,s)+f(-r,-s)] + \frac{20}{63}[f(0,t)+f(0,-t)] + \frac{8}{7}f(0,0) \quad , \qquad (5.91)$$

missä $r = \sqrt{3/5}$, $s = \sqrt{1/3}$ ja $t = \sqrt{14/15}$. Tässä tapauksessa funktio lasketaan 7 pisteessä ja kertaluku on 5. Solmupisteet sijoittuvat allaolevan kuvan mukaisesti yksikköneliön sisälle.



Kuva 5.11: Yksikköneliön integrointipisteet seitsemänpisteisessä Radonin kaavassa.

6 Epälineaariset yhtälöt ja yhtälöryhmät

6.1 Yleistä

Luvussa 3 ratkaistiin lineaarisia yhtälöitä. Todellisessa elämässä ongelmat ovat usein epälineaarisia. Esimerkiksi biologiassa tietyn lajin populaation P aikakäyttäytyminen voi olla muotoa

$$P(t) = 2e^{0.1t} \ . \tag{6.1}$$

Jos haluamme tietää, millä ajanhetkellä populaatio on esimerkiksi 1000 on ratkaistava t epälineaarisesta yhtälöstä

$$1000 - 2e^{0.1t} = 0 \ . \tag{6.2}$$

Tämä tietysti ratkeaa analyyttisesti. Malliin voi joutua lisäämään termejä, jotta se olisi realistisempi. Esimerkiksi ylläolevan ongelman ratkaisu mallilla

$$P(t) = 2e^{0.1t} - 0.05t^2 - 0.001t^{1.3} \tag{6.3}$$

on analyyttisesti mahdotonta.

Epälineaariset yhtälöt ja yhtälöryhmät esiintyvät hyvin usein laskennallisessa tieteessä joko itsenäisinä tehtävinä tai (kuten laskuharjoituksissa 6 huomaamme) osana suurempaa ongelmaa.

Yleisesti epälineaarinen yhtälö voidaan kirjoittaa muotoon

$$f(x) = 0 \ . \tag{6.4}$$

Jos muuttujia on useampia, saadaan yhtälöryhmä

$$\begin{aligned} f_1(x_1, x_2, \ldots, x_N) &= 0 \\ f_2(x_1, x_2, \ldots, x_N) &= 0 \\ &\ldots \\ f_N(x_1, x_2, \ldots, x_N) &= 0 \end{aligned} \ . \tag{6.5}$$

Merkitsemällä $\mathbf{x} = (x_1, x_2, \ldots, x_N)^T$ ja $\mathbf{f}(\mathbf{x}) = (f_1(\mathbf{x}), f_2(\mathbf{x}), \ldots, f_N(\mathbf{x}))^T$, tulee yhtälöryhmä muotoon

$$\mathbf{f}(\mathbf{x}) = 0 \ . \tag{6.6}$$

Päinvastoin kuin ei-singulaarisella lineaarisella yhtälöllä (tai -ryhmällä) epälineaarisella yhtälöllä ei välttämättä ole reaalista ratkaisua ollenkaan (esim. $\sin x + 2 = 0$) tai ratkaisuja on useampia ($\sin x = 0$).

Lineaarinen yhtälö voidaan ratkaista äärellisellä määrällä aritmeettisia operaatioita. Vain harvat epälineaariset yhtälöt voidaan tällä tavoin ratkaista. Esimerkiksi astelukuun 4 saakka voidaan polynomin nollakohdat ratkaista analyyttisesti. Siitä eteenpäin ei ole olemassa yleistä ratkaisukaavaa, jolla nollakohdat pystyttäisiin laskemaan käyttäen äärellinen määrä aritmeettisia operaatioita ja juurenottoa.

Koska jokainen äärellinen sekvenssi tietokoneen laskentaoperaatioita vastaa jotain tiettyä kaavaa, seuraa tästä se, että yleensä epälineaarisia yhtälöitä ei voi tarkasti ratkaista tietokoneella (vaikka olisi käytössä äärettömän tarkat liukuluvut).

Olkoon meillä epälineaarinen yhtälö (6.4), jonka tarkka ratkaisu on x^*: $f(x^*) = 0$. Etsimämme approksimaatio tälle on $\bar{x}$, jolle pätee

$$|f(\bar{x})| \approx 0 \tag{6.7}$$

tai

$$|\bar{x} - x^*| \approx 0 \quad . \tag{6.8}$$

Jälkimmäinen ehto näyttää käyttökelvottomalta, koska emme tiedä oikeaa ratkaisua x^*. Jos funktio f on jatkuva, ei oikeaa ratkaisua tarvitse tietää. Olkoon x:n arvot a ja b sellaisia, että funktion arvo on niissä erimerkkinen: $f(a)f(b) < 0$. Jatkuvuudesta seuraa, että väliltä $[a, b]$ löytyy piste x^*, jolle pätee $f(x^*) = 0$. Ja joka pisteelle x tällä välillä pätee $|x - x^*| \leq |a - b|$. Jos a on lähellä b:tä, pätee ehto (6.8)

Useimmiten yllämainitut ehdot ovat ekvivalentteja. Oletetaan, että funktion f derivaatta f' on jatkuva ja rajoitettu:

$$|f'(x)| \leq L \quad . \tag{6.9}$$

Lausumme funktion arvon pisteessä $\bar{x}$ Taylorin sarjana pisteen x^* ympäristössä:

$$|f(\bar{x})| = |f(x^*) + f'(\zeta)(\bar{x} - x^*)| = |f'(\zeta)| \cdot |\bar{x} - x^*| \leq L|\bar{x} - x^*| \quad , \tag{6.10}$$

missä ζ on jokin piste välillä $[\bar{x}, x^*]$. Eli, jos $|\bar{x} - x^*| \approx 0$, niin siitä seuraa, että $|f(\bar{x})| \approx 0$.

Jos edelleen oletamme, että

$$|f'(x)| \geq c > 0 \quad , \tag{6.11}$$

seuraa tästä se, että

$$|\bar{x} - x^*| \leq \frac{|f(\bar{x})|}{c} \quad . \tag{6.12}$$

Eli implikaatio toiseen suuntaan.

Epälineaarisen yhtälön ratkaisu lasketaan yleensä iteratiivisesti. Tuloksena saadaan sarja lukuja $x_1, x_2, \ldots, x_n, x_{n+1}, \ldots$, jotka ovat (toivon mukaan) aina parempia approksimaatioita yhtälön $f(x) = 0$ ratkaisulle.

Menetelmät vaativat yleensä jonkin alkuarvauksen ratkaisulle tai alueelle, jolla se sijaitsee. Hyvin paljon riippuu tästä alkuarvosta.

Yksiulotteisessa tapauksessa yleensä annetaan rajat $[a, b]$, joiden sisäpuolelta ratkaisu löytyy. Eli tulee päteä

$$f(a)\,f(b) < 0 \quad . \tag{6.13}$$

Yleensä tehtävästä ja funktiosta f tiedetään niin paljon, että nämä rajat pystytään asettamaan.

Tässä kappaleessa käymme läpi menetelmiä, joilla pyritään löytämään yksi yhtälön (6.4) tai yhtälöryhmän (6.6) reaalinen ratkaisu. Kaikkien juurien etsiminen on varsin hankala tehtävä (ellei mahdoton). Lisäksi sivuutamme kompleksijuuret. Aluksi käsittelemme yksittäistä yhtälöä.

6.2 Puolitusmenetelmä

Tämä on yksinkertaisin ja ehkä luotettavin menetelmä yhtälön $f(x) = 0$ ratkaisuun. Syöttötietoina annetaan väli $[a, b]$ ($a < b$), jonka päätepisteissä funktio f on erimerkkinen:

$$f(a) \cdot f(b) < 0 \quad . \tag{6.14}$$

Jokaisella iteraatiokierroksella väli jaetaan kahtia, ja otetaan uudeksi väliksi se puolikas, jonka alueella ratkaisu on. Tätä jatketaan, kunnes on saavutettu tarkkuus. Eli alogoritmina

i. Jos $b - a \le \mathrm{tol}_1$, lopeta.

ii. Aseta $m = \frac{1}{2}(a + b)$, eli välin $[a, b]$ keskipiste. Laske $f(m)$. Jos $|f(m)| \le \mathrm{tol}_2$, lopeta.

iii. Jos $f(a) \cdot f(m) < 0$, aseta $b \leftarrow m$, muuten aseta $a \leftarrow m$. Mene askeleeseen i.

Tässä on siis kaksi lopetusehtoa, jos ollaan tarpeeksi lähellä juurta tai jos funktion arvo on tarpeeksi pieni. Näitä kontrolloidaan parametreilla tol_1 ja tol_2 . Käytännön ohjelmassa on tietysti oltava jokin maksimi iteraatiokierrosten määrälle.

Otetaan esimerkiksi funktion $f(x) = x^3$ juuren etsintä väliltä $[a, b] = [-1, 2]$.

Alkutilanne: $a = -1$, $b = 2$, $f(a) = -1$, $f(b) = 8$.

1. iteraatio: $m = (-1 + 2)/2 = 1/2$, $f(m) = 1/8$,
$b \leftarrow 1/2$, $[a, b] = [-1, 1/2]$.

2. iteraatio: $m = (-1 + 1/2)/2 = -1/4$, $f(m) = -1/64$,
$a \leftarrow -1/4$, $[a, b] = [-1/4, 1/2]$.

3. iteraatio: $m = (-1/4 + 1/2)/2 = 1/8$, $f(m) = 1/512$,
$b \leftarrow 1/8$, $[a, b] = [-1/4, 1/8]$.

jne.

Aliohjelmana puolitusmenetelmä ei montaa riviä vie. NR:n vastaava rutiini on `rtbis`.

Jos funktiosta ei ole kovin paljon tietoa, voidaan alunperin annettua väliä $[a, b]$ laajentaa, jos se ei sisällä nollakohtaa. NR:n rutiini `zbrac` tekee tämän.

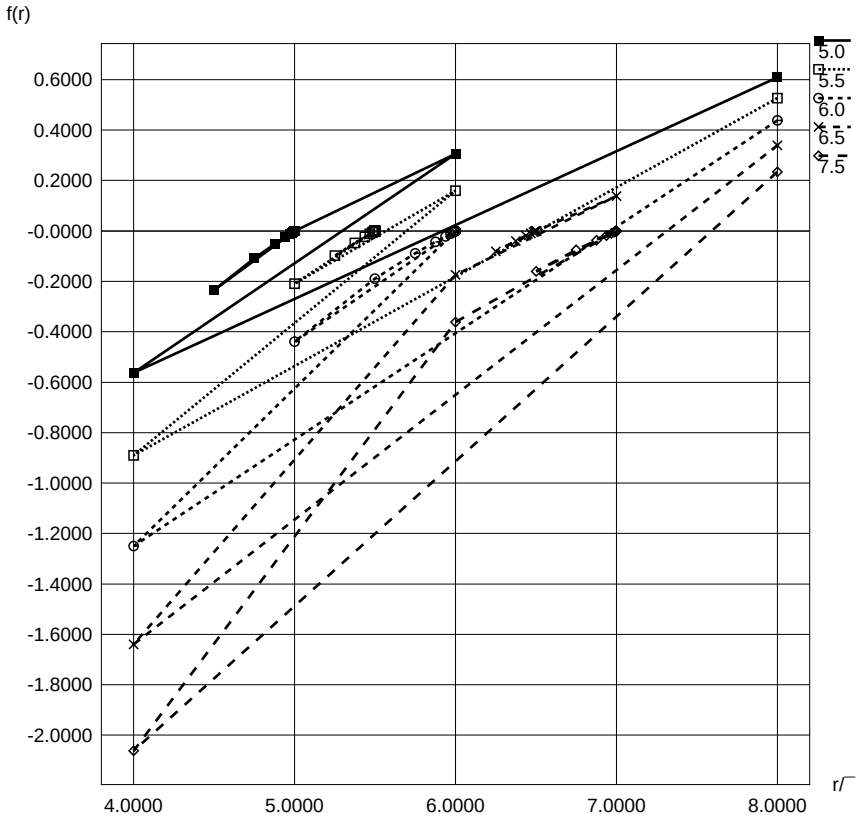
Esimerkkinä funktion

$$f(r) = 1 - \frac{b^2}{r^2} - \frac{V(r)}{E} \tag{6.15}$$

nollakohdan etsintä. Tässä funktio $V(r)$ on

$$V(r) = 4\varepsilon\left[\left(\frac{\sigma}{r}\right)^{12} - \left(\frac{\sigma}{r}\right)^{6}\right] \quad , \tag{6.16}$$

missä $\varepsilon = 3.121\times10^{-3}$ eV ja $\sigma = 2.74$ Å . Yhtälö $f(r) = 0$ ratkaistiin parametrien arvoilla $E = 5$ eV ja $b = 5.0, 5.5, 6.0, 6.5, 7.0$ Å . Tarkkuusvaatimus `xacc` oli 1×10^{-6} . Alla kuva iteraation kulusta eri b : arvoilla.



Kuva 6.1: Yhtälön $f(r) = 0$ ratkaisu rutiinilla `rtbis`.

Menetelmän tarkkuuden arviointi on varsin yksinkertaista. Jokaisella iteraatiokierroksella etsintäväli puolittuu. Jos ajattelemme, että yhtälön ratkaisun x^* arvio on välin keskipiste m, puolittuu myös ratkaisun virhe $|m - x^*|$.

Eli jokaisella iteraatiokierroksella saadaan yksi merkitsevä binäärinumero lisää ratkaisuun. Jos liukuluvussa on 24-bittinen mantissa, saavutetaan suurin mahdollinen tarkkuus 24 iteraatiossa.

Otetaan esimerkiksi funktion $f(x) = x^2 - 2$ nollakohdan etsintä. Alla ohjelmanpätkä, joka tämän tekee:

```
program root
  implicit none
  integer, parameter :: sp=4
  real(sp) :: x1,x2,dx,f,fmid,xmid,rtbis
  integer, parameter :: MAXIT=60
  integer :: j

  x1=-1.0_sp
  x2=2.0_sp
  fmid=func(x2)
  f=func(x1)
  if (f < 0.0) then
     rtbis=x1
     dx=x2-x1
  else
```

```
      rtbis=x2
      dx=x1-x2
    end if
    do j=1,MAXIT
      dx=dx*0.5_sp
      xmid=rtbis+dx
      fmid=func(xmid)
      if (fmid <= 0.0) rtbis=xmid
      write(6,'(i5,2g24.14)') j,fmid,xmid
    end do

  contains
    function func(x)
      implicit none
      real (sp) :: func,x
      func=x**2-2.0_sp
      return
    end function func

  end program root
```

Seuraavalla sivulla ohjelman tulostus yksinkertaisen ja kaksinkertaisen tarkkuuden lukuja käytettäessä.

1	-1.75000	0.50000000
2	-0.437500	1.25000000
3	0.640625	1.62500000
4	0.664063E-01	1.43750000
5	-0.194336	1.34375000
6	-0.661621E-01	1.39062500
7	-0.427246E-03	1.41406250
8	0.328522E-01	1.42578125
9	0.161781E-01	1.41992188
10	0.786686E-02	1.41699219
11	0.371766E-02	1.41552734
12	0.164461E-02	1.41479492
13	0.608683E-03	1.41442871
14	0.905991E-04	1.41424561
15	-0.168324E-03	1.41415405
16	-0.388622E-04	1.41419983
17	0.259876E-04	1.41422272
18	-0.643730E-05	1.41421127
19	0.977516E-05	1.41421700
20	0.166893E-05	1.41421413
21	-0.238419E-05	1.41421270
22	-0.357628E-06	1.41421342
23	0.715256E-06	1.41421378
24	0.238419E-06	1.41421366
25	-0.119209E-06	1.41421354
26	-0.119209E-06	1.41421354
27	-0.119209E-06	1.41421354

1	-1.75000	0.500000000000000000
2	-0.437500	1.250000000000000000
3	0.640625	1.625000000000000000
4	0.664063E-01	1.437500000000000000
5	-0.194336	1.343750000000000000
6	-0.661621E-01	1.390625000000000000
7	-0.427246E-03	1.414062500000000000
8	0.328522E-01	1.425781250000000000
9	0.161781E-01	1.419921875000000000
10	0.786686E-02	1.416992187500000000
11	0.371766E-02	1.415527343750000000
12	0.164467E-02	1.414794921875000000
13	0.608578E-03	1.414428710937500000
14	0.906326E-04	1.414245605468750000
15	-0.168315E-03	1.414154052734375000
16	-0.388434E-04	1.414199829101562500
17	0.258941E-04	1.414222717285156250
18	-0.647477E-05	1.414211273193359375
19	0.970962E-05	1.414216995239257813
20	0.161742E-05	1.414214134216308594
21	-0.242868E-05	1.414212703704833984
22	-0.405632E-06	1.414213418960571289
23	0.605893E-06	1.414213776588439941
24	0.100130E-06	1.414213597774505615
25	-0.152751E-06	1.414213508367538452
26	-0.263102E-07	1.414213553071022034
27	0.369100E-07	1.414213575422763824
28	0.529990E-08	1.414213564246892929
29	-0.105052E-07	1.414213558658957481
30	-0.260263E-08	1.414213561452925205
31	0.134863E-08	1.414213562849909067
32	-0.627000E-09	1.414213562151417136
33	0.360817E-09	1.414213562500663102
34	-0.133092E-09	1.414213562326040119
35	0.113863E-09	1.414213562413351610
36	-0.961431E-11	1.414213562369695865
37	0.521241E-10	1.414213562391523737
38	0.212550E-10	1.414213562380609801
39	0.582023E-11	1.414213562375152833
40	-0.189693E-11	1.414213562372424349
41	0.196154E-11	1.414213562373788591
42	0.324185E-13	1.414213562373106470
43	-0.932365E-12	1.414213562372765409
44	-0.450084E-12	1.414213562372935939
45	-0.208944E-12	1.414213562373021205
46	-0.883738E-13	1.414213562373063837
47	-0.279776E-13	1.414213562373085153
48	0.222045E-14	1.414213562373095812
49	-0.128786E-13	1.414213562373090483
50	-0.532907E-14	1.414213562373093147
51	-0.155431E-14	1.414213562373094479
52	0.444089E-15	1.414213562373095145
53	-0.444089E-15	1.414213562373094923
54	0.444089E-15	1.414213562373095145
55	-0.444089E-15	1.414213562373094923
56	-0.444089E-15	1.414213562373094923
57	-0.444089E-15	1.414213562373094923

Jos merkitsemme virhettä iteraatiokierroksella i symbolilla $e_i = m - x^*$, saadaan puolitussäännölle

$$\frac{|e_{i+1}|}{|e_i|} \approx \frac{1}{2} \quad . \tag{6.17}$$

Yleisesti sanomme, että iteraation suppenemisen kertaluku on r , jos pätee

$$\lim_{i \to \infty} \frac{|e_{i+1}|}{|e_i|^r} = C \quad , \tag{6.18}$$

missä C on jokin äärellinen nollasta poikkeava vakio. Toinen tapa kirjoittaa (6.18) on

$$|e_{i+1}| = O(|e_i|^r) \quad . \tag{6.19}$$

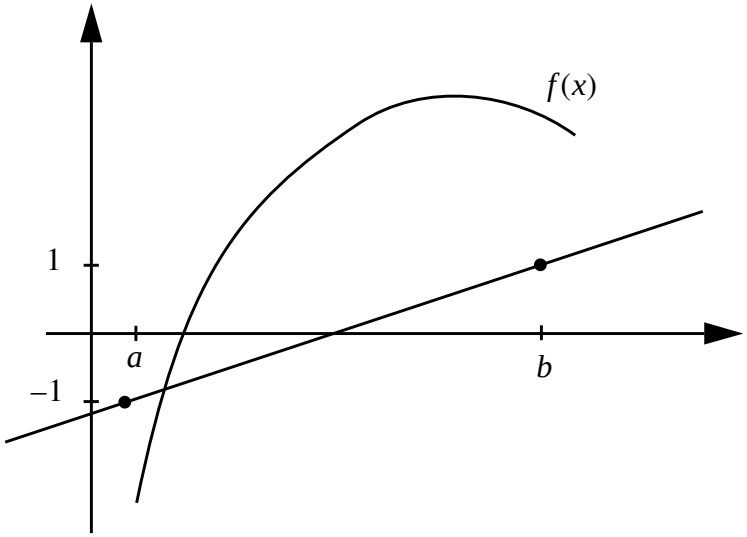
Puolitussäännölle $r = 1$ ja $C = 1/2$. Puhutaan lineaarisesta suppenemisesta. Yleensä pyritään mahdollisimman suureen suppenemisen kertalukuun.

Puolitussäännön voi myös hahmottaa funktion approksimoinnin mielessä. Joka iteraatiokierroksella approksimoidaan funktiota jollain lineaarisella funktiolla ja lasketaan tämän nollakohta.

Puolitussäännössä tämä funktio sattuu olemaan yksinkertaisesti suora viiva pisteestä $(a, \text{sign}(f(a)))$ ja $(b, \text{sign}(f(b)))$ (kuva 6.2). Jos oletetaan, että $f(a) < 0$ ja $f(b) > 0$, on tuon viivan yhtälö

$$y = -1 + 2\frac{x-a}{b-a} \quad . \tag{6.20}$$

Nyt $y = 0$, jos $x = m = (1/2)(a+b)$.

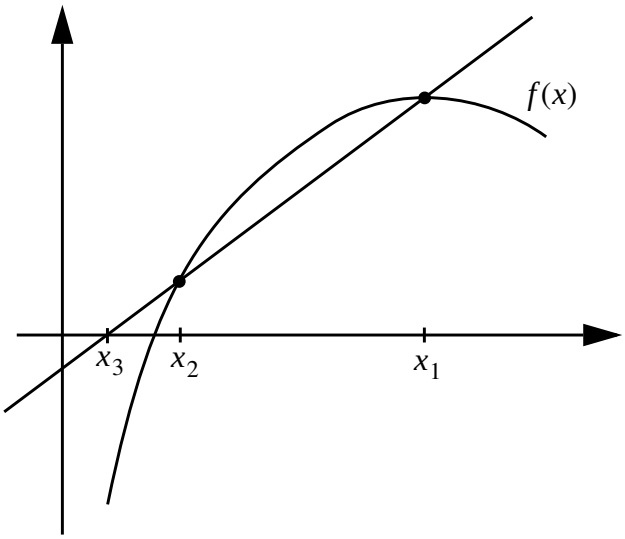


Kuva 6.2: Puolitussääntö.

Approksimoimalla funktiota paremmin, päästään suurempiin suppenemisen kertalukuihin r.

6.3 Sekanttimenetelmä

Sekanttimenetelmässä kahden edellisen iteraatiopisteen $(x_1, f(x_1))$ ja $(x_2, f(x_2))$ kautta piirretään suora, jonka leikkauspiste x-akselin kanssa otetaan uudeksi approksimaatioksi ja toinen edellisistä pisteistä jätetään pois.



Kuva 6.3: Sekanttimenetelmä.

Merkitään edellisiä iteraatiotuloksia $(x_i, f(x_i))$ ja $(x_{i-1}, f(x_{i-1}))$. Näiden kautta kulkevan suoran yhtälöhän on

$$y = f(x_i) + (x - x_i)\frac{f(x_i) - f(x_{i-1})}{x_i - x_{i-1}} \quad . \tag{6.21}$$

Leikkauspiste x-akselin kanssa saadaan yhtälöstä $y = 0$ eli (merkitään ratkaisua x_{i+1} :llä)

$$x_{i+1} = x_i - f(x_i)\frac{x_i - x_{i-1}}{f(x_i) - f(x_{i-1})} \quad . \tag{6.22}$$

Esimerkkinä muutama iteraatiokierros funktion $f(x) = x^2 - 2$ nollakohdan etsinnässä:

Alkutilanne: $x_0 = 1$, $x_1 = 2$.

1. iteraatio: $x_2 = x_1 - f(x_1)((x_1 - x_0)/(f(x_1) - f(x_0)))$
$= 2 - (2)((2-1)/(2-(-1))) = 4/3$

2. iteraatio: $x_3 = 4/3 - (-2/9)(4/3 - 2)/(-2/9 - 2) = 7/5$

3. iteraatio: $x_4 = 7/5 - (-1/25)(7/5 - 4/3)/(-1/25 - (-2/9))$.
$= 58/41 \approx 1.414634$

Oikea vastaushan on $x^* = \sqrt{2} \approx 1.414214$.

Puolitussäännöllä 3. iteraatiokierroksen jälkeen tulos olisi $x_4 = 1.625$ (ks. sivu 92).

Sekanttimenetelmä suppenee ylilineaarisesti: $r = (1/2)(1 + \sqrt{5})$ eli

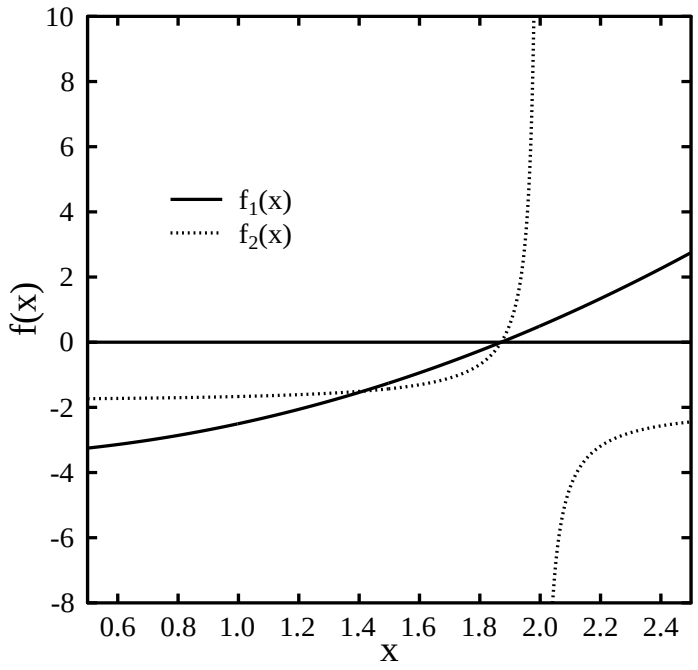
$$|e_{i+1}| = O(|e_i|^{1.6180}) \quad . \tag{6.23}$$

Kuten odotettavissa on, löytyy tilanteita, joissa sekanttimenetelmä toimii huonosti. Vertaillaan kahta funktiota:

$$\begin{aligned} f_1(x) &= x^2 - 7/2 \\ f_2(x) &= -\frac{1}{x^2 - 4} - 2 \end{aligned} \quad . \tag{6.24}$$

Molempien nollakohta on $x = \sqrt{7/2} \approx 1.8708287$.

Alla funktioiden kuvaajat.



Kuva 6.4: Funktiot $f_1(x) = x^2 - 7/2$ ja $f_2(x) = -1/(x^2 - 4) - 2$.

Juuret $f_1(x) = 0$ ja $f_2(x) = 0$ etsittiin käyttäen NR:n rutiinia `rtsec` ja parametrien arvoja `x1=1.00` , `x2=1.91` ja `xacc=1.0e-6`. Iteraation aika tulostettiin juuren arvo ja funktiona arvo tässä pisteessä. Oleelinen osa rutiinista on alla:

```
fl=(*func)(x1);
f=(*func)(x2);
if (fabs(fl) < fabs(f)) {
     rts=x1; xl=x2; swap=fl; fl=f; f=swap;
} else {
     xl=x1; rts=x2;
}
for (j=1;j<=MAXIT;j++) {
     dx=(xl-rts)*f/(f-fl);
     xl=rts;
     fl=f;
     rts += dx;
     f=(*func)(rts);
     if (fabs(dx) < xacc || f == 0.0) return rts;
}
```

Ensin funktio f_1 :

```
I    xi                      f(xi)

1   1.85910652920962      -4.372291305015263E-002
2   1.87070686809931      -4.558136460848239E-004
3   1.87082907626297       1.432590948535761E-006
4   1.87082869337450      -4.664535424581118E-011
```

Funktio f_2 :

```
I    xi                      f(xi)

1   1.60463917525773       -1.29831116271794
2   1.78989537239138        -0.744151759464560
3   2.03866625715734         -8.40368409286354
4   1.76572636525828        -0.866483554758109
5   1.73434894249072        -0.991969774865652
6   1.98238777075349         12.2574587791504
7   1.75291932985699        -0.921569901811837
8   1.76896536888348        -0.851579941473249
9   1.96420029842120         5.04636285662813
10   1.79715454527271       -0.701695848164392
11   1.81754670155672       -0.564299266477370
12   1.90129899503933        0.596983495110348
13   1.85824424684140       -0.171606835644274
14   1.86785728619677       -4.347078591337628E-002
15   1.87111855714021        4.348051768997330E-003
16   1.87082201762604       -9.990847146412740E-005
17   1.87082867838917       -2.244664789596840E-007
18   1.87082869338775        1.161382101599884E-011
```

Laskut tehtiin kaksoistarkkuuden luvuilla.

Funktiolla f_2 joudutaan iteroimaan varsin pitkään ennen kuin saavutetaan haluttu tarkkuus. Tämä johtuu osittain siitä, että sekanttimenetelmässä (NR:n versiossa) käytetään aina kahta edellistä iteraatiotulosta riippumatta siitä, onko juuri niiden välissä. Lisäksi f_2 :lla on nimittäjän nollakohta, kun $x = 2$ ja iteraation alkuvaiheessa käydään funktion toisessa haarassa.

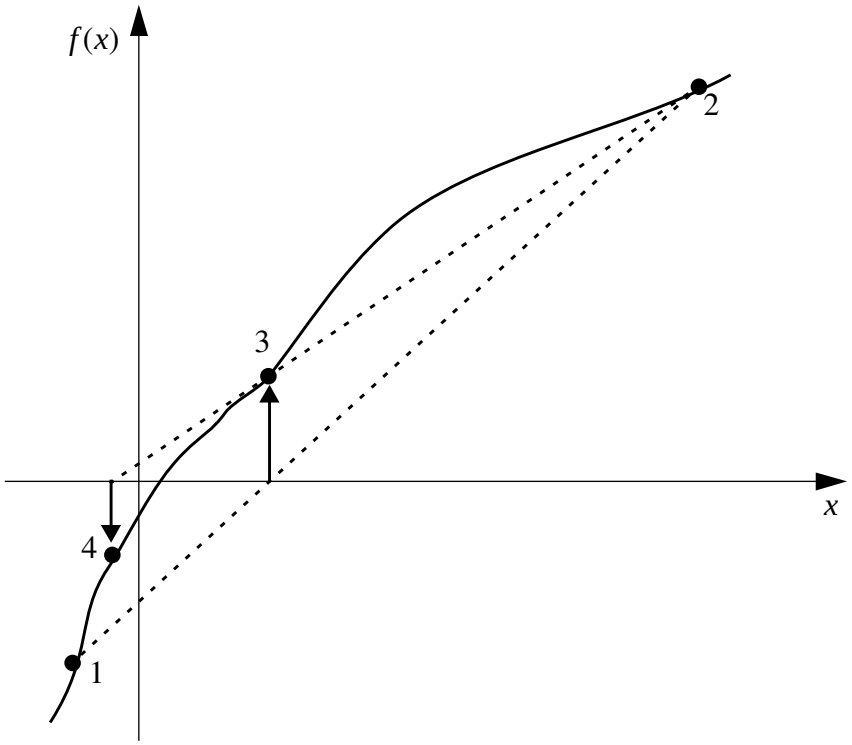
Sekanttimenetelmässä voi myös valita seuraavan itreraatioparin toisin kuin `rtsec`-rutiinin, eli siten, että nollakohta säilyy aina iteraatioparin välissä. NR:n rutiini on `rtflsp`. Alla samaisen funktion f_2 nollakohdan etsintä tällä rutiinilla:

```
I    xi                      f(xi)

1   1.60463917525773       -1.29831116271794
2   1.78989537239138      -0.744151759464560
3   1.84625319104558      -0.308951670432048
4   1.86336905936041      -0.105543034024736
5   1.86856464865188      -3.330075657966880E-002
6   1.87014156465159      -1.022952210069694E-002
```

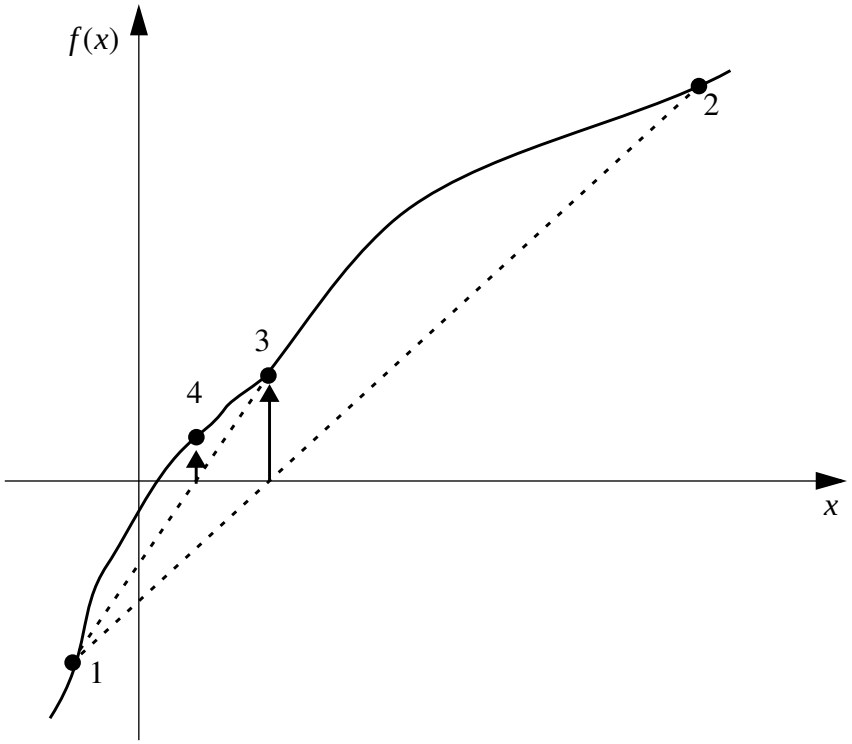
```
7   1.87062015459057      -3.116086384895134E-003
8   1.87076540350391      -9.467718107392109E-004
9   1.87080948542342      -2.874356834887681E-004
10   1.87082286392825      -8.724340726740110E-005
11   1.87082692419442      -2.647848642811645E-005
12   1.87082815645166      -8.036078376294498E-006
13   1.87082853043155      -2.438890370992652E-006
```

Tämän menetelmän suppeneminen on kuitenkin hieman hitaampaa kuin rutiinin `rtsec` menetelmän.

Allaolevat kuvat selittävät rutiinien `rtsec` ja `rtflsp` eron.



Kuva 6.5: Rutiinin `rtsec` toimintaperiaate.



Kuva 6.6: Rutiinin `rtflsp` toimintaperiaate.

No, tämä f_2 on hankala muillekin menetelmille. Esimerkiksi puolitussäännöllä saadaan tulos

```
I xi                    f(xi)

1 1.455000000000000071  -1.46893
2 1.682500000000000107  -1.14471
3 1.796250000000000124 -0.707152
4 1.853125000000000133 -0.232990
5 1.881562500000000027  0.175225
6 1.867343750000000080 -0.507861E-01
7 1.874453125000000053  0.558133E-01
8 1.870898437500000178  0.104440E-02
9 1.869121093750000018 -0.252232E-01
10 1.870009765625000098 -0.121793E-01
11 1.870454101562500027 -0.559015E-02
12 1.870676269531249991 -0.227858E-02
13 1.870787353515624973 -0.618520E-03
14 1.870842895507812464  0.212581E-03
15 1.870815124511718830 -0.203059E-03
16 1.870829010009765758  0.473879E-05
17 1.870822067260742294 -0.991657E-04
18 1.870825538635253915 -0.472148E-04
19 1.870827274322509837 -0.212384E-04
20 1.870828142166137686 -0.824988E-05
```

Tässä tapauksessa funktion nimittäjän nollakohta ei tuota vaikeuksia, jos tämä kohta ei ole alkuperäisen välin $[a, b]$ sisällä.

Sekanttimenetelmä joutuu myös vaikeuksiin, jos $f(x_i) \approx f(x_{i-1})$. Tällöin korjaus iteraatiokaavassa (6.22) voi kasvaa hyvinkin suureksi, ja seuraava iteraatio x_{i+1} voi joutua kauas todellisen nollakohdan läheltä.

Sekanttimenetelmää voi yleistää sovittamalla suoran sijaan toisen asteen käyrän kolmeen edelliseen iteraatiopisteeseen:

$$
\begin{aligned}
g(y) = {} & x_{i-2}\frac{(y-f_{i-1})(y-f_i)}{(f_{i-2}-f_{i-1})(f_{i-2}-f_i)} \\
& + x_{i-1}\frac{(y-f_{i-2})(y-f_i)}{(f_{i-1}-f_{i-2})(f_{i-1}-f_i)} \\
& + x_i\frac{(y-f_{i-2})(y-f_{i-1})}{(f_i-f_{i-2})(f_i-f_{i-1})}
\end{aligned}
\quad . \tag{6.25}
$$

Tässä on siis sovitettu paraabeli $g(y)$ pistejoukkoon (f_{i-2}, x_{i-2}) , (f_{i-1}, x_{i-1}) , (f_i, x_i) . Arvio nollakohdalle saadaan asettamalla paraabelin arvona $g(0)$. Voidaan osoittaa, että tämän menetelmän suppenemisen kertaluku on 1.839.

Edellä olemme huomanneet, että jokaisella menetelmällä on hyvät ja huonot puolensa:

> Puolitussääntö on melko varma menetelmä, jos alkuarvot eli väli $[a, b]$ on järkevä, mutta se suppenee hitaasti.
>
> Sekanttimenetelmä ja ylläesitetty paraabelisovitus suppenevat nopeammin, mutta niiden kanssa voi helposti joutua vaikeuksiin, jos funktio ei ole hyvin tasainen.

Hyvä strategia on yhdistää nämä menetelmät. Jos käytettäessä nopeammin suppenevia menetelmiä huomataan, että suppeneminen ei ole tarpeeksi nopeaa, tehdään muutama iteraatio puolitussäännöllä.

Näin on tehty NR:n rutiinissa `zbrent`, jossa käytetään paraabelisovitusta (6.25). Jos suppeneminen ei ole toivottua, tai kaavan (6.25) mukainen iteraatiotulos asettuisi niiden rajojen ulkopuolelle, joiden sisällä juuren tiedetään olevan, käytetään paraabelisovituksen sijasta puolitussääntöä.

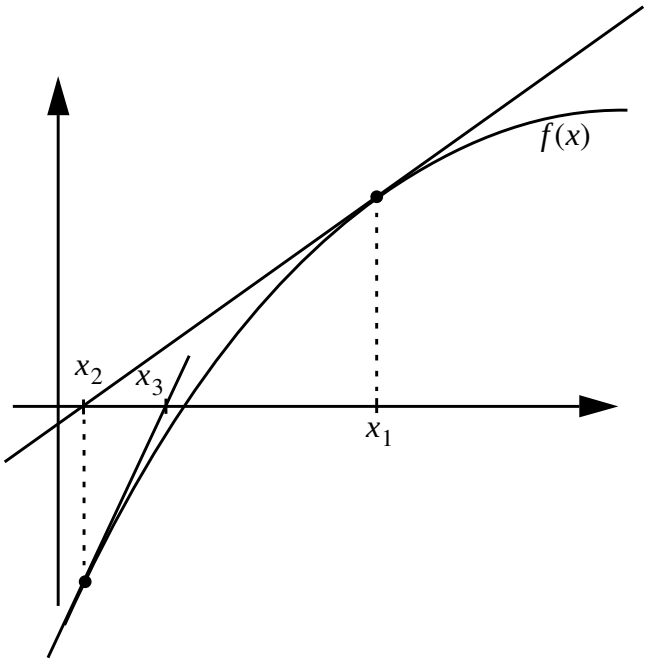
Alla esimerkkinä käytetyn funktion f_2 [kaava (6.24)] nollakohdan iterointi parametrien arvoilla $[a, b] = [1.0, 1.91]$ ja `xacc` $= 1\times10^{-6}$.

```
I    xi                    f(xi)

1   1.91000000000000       0.841716396703608
2   1.91000000000000       0.841716396703608
3   1.78989537239138      -0.744151759464560
4   1.84625319104558      -0.308951670432048
5   1.87551973726703       7.285799378975089E-002
6   1.86993501261844      -1.328338036293353E-002
7   1.87079620182582      -4.861667285616100E-004
8   1.87082870157870       1.226026520306789E-007
9   1.87082870157870       1.226026520306789E-007
```

Eli varsin nopea suppeneminen.

6.4 Newtonin menetelmä

Jos käytössä on funktion derivaatta $f'(x)$, on Newtonin (tai Newtonin ja Raphsonin) menetelmä nopeasti suppenevana varteenotettava vaihtoehto. Alla oleva kuva havainnollistaa menetelmää.



Kuva 6.7: Newtonin menetelmä.

Funktiota approksimoidaan tangentillaan pisteessä x_i . Uusi arvio nollakohdalle saadaan tangentin nollakohdasta.

Esitetään funktio $f(x)$ Taylorin sarjana pisteen x_i ympäristössä:

$$f(x) = f(x_i) + f'(x_i)(x - x_i) + \ldots \quad . \tag{6.26}$$

Tangentin yhtälö on nuo kaksi ensimmäistä termiä sarjasta

$$y = f(x_i) + f'(x_i)(x - x_i) \quad . \tag{6.27}$$

Asettamalla $y = 0$ saadaan iteraatio kaava Newtonin menetelmälle:

$$x_{i+1} = x_i - \frac{f(x_i)}{f'(x_i)} \quad . \tag{6.28}$$

Otetaan esimerkiksi funktio $f(x) = x^2 - 2$. Derivaatta on $f'(x) = 2x$.

Alkutilanne: $x_0 = 1$

1. iteraatio: $x_1 = x_0 - (x_0^2 - 2)/(2x_0) = 1 - (1-2)/2 = 3/2$

2. iteraatio: $x_2 = 3/2 - (9/4 - 2)/3 = 17/12$

3. iteraatio: $x_3 = 17/12 - (289/144 - 2)/(17/6)$
$= 577/408 \approx 1.414216$

Oikea arvo on $x \approx 1.414214$. Eli kolmen iteraation jälkeen 6 merkitsevää numeroa oikein.

Neliöjuuren laskemista varten aikoinaan[1] koulussa opetettu iteraatiokaava perustuu Newtonin menetelmään. Eli luvun a neliöjuuri saadaan iteroimalla funktion $f(x) = x^2 - a$ nollakohtaa. Iteraatiokaava (6.26) tulee muotoon

$$x_{i+1} = \frac{1}{2}\left(x_i + \frac{a}{x_i}\right) \quad . \tag{6.29}$$

Alla lyhyt C-ohjelma, joka laskee neliöjuuren iteroimalla:

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
main (int argc, char **argv)
{
  double a,x0,x,y;
  int n,i;
  if (argc!=4) {
    fprintf(stderr,"Usage: %s x0 a n\n",argv[0]); return (1);
  }
  x0=atof(*++argv); a=atof(*++argv); n=atoi(*++argv);
  y=sqrt(a); x=x0;
  printf(" %5d %25.18g %25.18g\n",0,x,(x-y)*(x-y));
  for (i=1;i<=n;i++) {
    x=0.5*(x+a/x);
    printf(" %5d %25.18g %25.18g\n",i,x,(x-y)*(x-y));
  }
  return(0);
}
```

1. Vielä 70-luvun lopulla.

Käännös ja ajo:

```
rootfinding> cc -o sqrt_c sqrt.c -lm
rootfinding> sqrt_c 1 9 10
     0      1.000000000000000000
     1      5.000000000000000000
     2      3.399999999999999900
     3      3.023529411764705800
     4      3.000091554131380200
     5      3.000000001396983900
     6      3.000000000000000000
     7      3.000000000000000000
     8      3.000000000000000000
     9      3.000000000000000000
    10      3.000000000000000000
rootfinding> sqrt_c -1 9 10
     0     -1.000000000000000000
     1     -5.000000000000000000
     2     -3.399999999999999900
     3     -3.023529411764705800
     4     -3.000091554131380200
     5     -3.000000001396983900
     6     -3.000000000000000000
     7     -3.000000000000000000
     8     -3.000000000000000000
     9     -3.000000000000000000
    10     -3.000000000000000000
rootfinding>
```

Newtonin menetelmän suppenemisen kertaluku on kaksi. Tämä voidaan osoittaa seuraavasti.

Kehitetään $f(x)$ Taylorin sarjaksi pisteen x_i ympäristössä:

$$f(x) = f(x_i) + f'(x_i)(x - x_i) + \frac{1}{2} f''(\zeta)(x - x_i)^2 \quad , \tag{6.30}$$

missä ζ on jokin välin $[x, x_i]$ piste. Asetetaan $x = x^*$ eli yhtälön $f(x) = 0$ ratkaisu. Nyt saadaan

$$0 = f(x^*) = f(x_i) + f'(x_i)(x^* - x_i) + \frac{1}{2} f''(\zeta)(x^* - x_i)^2 \quad . \tag{6.31}$$

Jakamalla tämä yhtälö $f'(x_i)$:llä saamme edelleen

$$x^* - \left(x_i - \frac{f(x_i)}{f'(x_i)} \right) = \frac{f''(\zeta)}{2 f'(x_i)} (x^* - x_i)^2 \ . \tag{6.32}$$

Yhtälön vasen puoli on (6.28):n mukaan $x^* - x_{i+1}$. Jos määrittelemme tulosten virheeksi $e_i = x^* - x_i$ ja $e_{i+1} = x^* - x_{i+1}$, saamme

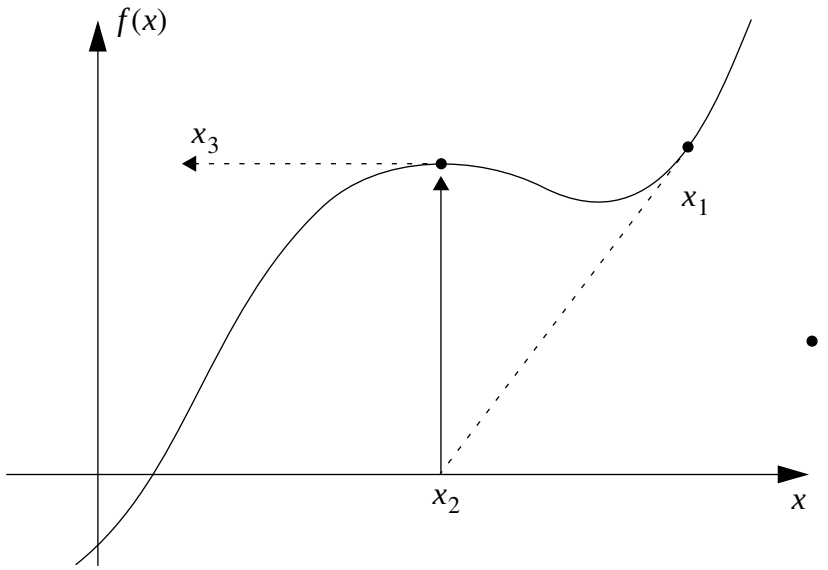
$$e_{i+1} = C_i e_i^2 \quad , \tag{6.33}$$

missä $C_i = f''(\zeta)/(2f'(x_i))$. Jos iteraatio suppenee, niin

$$\lim_{i \to \infty} \frac{|e_{i+1}|}{|e_i^2|} = C \ , \tag{6.34}$$

missä $C = |f''(x^*)/(2f'(x^*))|$.

Newtonin menetelmä voi tietysti myös epäonnistua. Esimerkiksi allaolevan kuvan mukaisessa tilanteessa, kun $f'(x_i) \approx 0$, seuraava iteraatiotulos ajautuu hyvin kauas oikeasta ratkaisusta.



Kuva 6.8: Newtonin menetelmän epäonnistuminen.

Kuten edellä on tullut esille, paras menetelmä on useamman kuin yhden menetelmän yhdistelmä. Tällainen on NR:n rutiini `rtsafe`, jossa sovelletaan puolitussääntöä, jos Newtonin menetelmä vie iteraation ennaltamäärättyjen rajojen ulkopuolelle tai jos se ei suppene riittävän nopeasti.

Alla esimerkkinä käytetyn funktion f_2 [kaava (6.24)] nollakohdan iterointi parametrien arvoilla $[a, b] = [1.0, 1.91]$ ja `xacc` $= 1 \times 10^{-6}$.

```
I    xi                    f(xi)

1   1.45500000000000       -1.46892550352501
2   1.68250000000000       -1.14470976260350
3   1.79625000000000      -0.707151673329548
4   1.85312500000000      -0.232989939776709
5   1.87325873806264       3.706770509442148E-002
6   1.87087445941745       6.852062755804411E-004
7   1.87082870962077       2.429652523616710E-007
```

Paras suppeneminen tähän mennessä.

Newtonin menetelmällä ja fraktaaleilla on tekemistä keskenään. Tutkitaan yhtälöä

$$z^3 - 1 = 0 \quad , \tag{6.35}$$

missä z on kompleksiluku. Yhtälön ratkaisut ovat $z = 1$, ja $z = \exp(\pm 2\pi i/3)$. Newtonin menetelmä antaa iteraatiokaavan

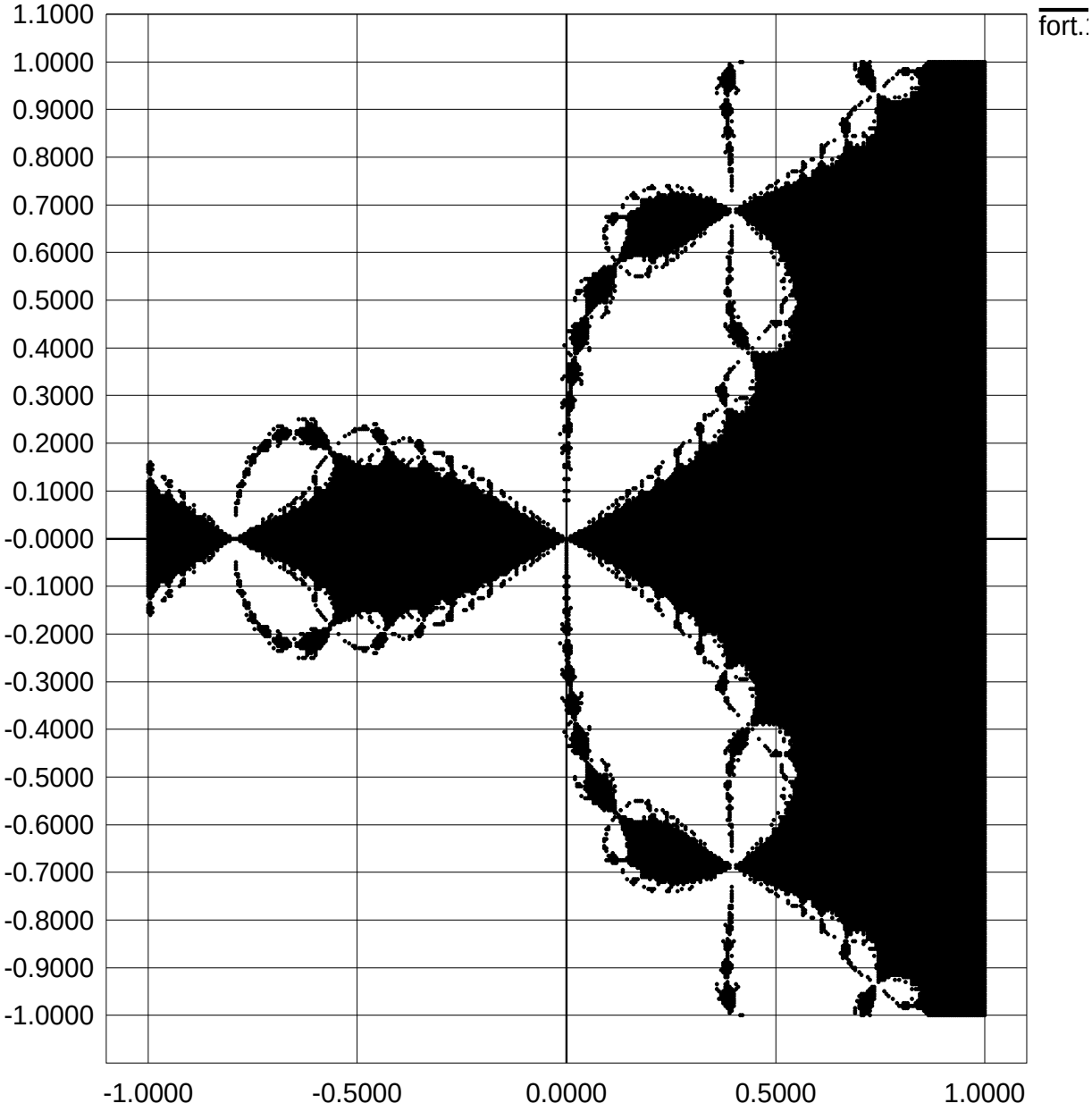
$$z_{j+1} = z_j - \frac{z_j^3 - 1}{3z_j^2} \quad . \tag{6.36}$$

Tutkittaessa sitä kompleksitason aluetta, josta iteraatioon lähdettäessä päädytään juureen $z = 1$, havaitaan sen muodostavan fraktaalimaisen kuvion.

Alla lyhyt f90-ohjelma, jolla voidaan asiaa havainnollistaa:

```
program iter
  implicit none
  integer, parameter :: rr=4,IMAX=100,IRMAX=500,IIMAX=500
  real (rr), parameter :: DR=0.001,DI=0.001,EPSI=1.0E-6
  complex (rr), parameter :: zroot=(1.0,0.0)
  real (rr) :: d,zr,zi
  complex (rr) :: z0,z1
  character (len=80) :: line
  integer :: ir,ii,i

  do ir=-IRMAX,IRMAX
     zr=DR*ir
     do ii=-IIMAX,IIMAX
     if (ir==0 .and. ii==0) exit
     zi=DI*ii
     z0=cmplx(zr,zi)
     do i=1,IMAX
        z1=z0-(z0**3-1.0_rr)/(3.0_rr*z0**2)
        d=abs(z1-zroot)
        if (d<EPSI) exit
        z0=z1
     enddo
     if (i<IMAX) write(10,*) zr,zi
     enddo
  enddo

end program iter
```

Alla ohjelman tulostus graafisesti.


1.1000
1.0000
0.9000
0.8000
0.7000
0.6000
0.5000
0.4000
0.3000
0.2000
0.1000
-0.0000
-0.1000
-0.2000
-0.3000
-0.4000
-0.5000
-0.6000
-0.7000
-0.8000
-0.9000
-1.0000
-1.0000
-0.5000
0.0000
0.5000
1.0000

6.5 Moninkertaiset juuret

Funktion $f(x)$ monikertaisten juurien paikantaminen on jossain määrin vaikeampaa kuin yksinkertaisten juurien. Juuren x^* kertaluku m määritellään siten, että juuren ympäristössä funktio voidaan kirjoittaa muodossa

$$f(x) = (x - x^*)^m g(x) \quad , \tag{6.37}$$

missä $g(x)$ on jatkuva x^* :n ympäristössä ja $g(x^*) \neq 0$. Jos $m = 1$ on juuri yksinkertainen, muulloin moninkertainen. Kertaluvun ei välttämättä tarvitse olla kokonaisluku. Esimerkiksi tapauksessa

$$f(x) = x\sqrt{x-1}$$

juuren $x^* = 1$ kertaluku on $1/2$. Jos funktio on tarpeeksi tasainen, juurien kertaluku on positiivinen kokonaisluku.

Jos funktion $f(x)$ m ensimmäistä derivaattaa ovat jatkuvia jollain välillä, jolle juuri x^* sisältyy, ja lisäksi pätee seuraava

$$\begin{cases} f(x^*) = 0 \\ f'(x^*) = f''(x^*) = \ldots = f^{(m-1)}(x^*) = 0 \\ f^{(m)}(x^*) \neq 0 \end{cases} \quad , \tag{6.38}$$

on juuren x^* kertaluku m . Tämä on helppo huomata, kun kehitetään funktio sarjaksi juuren ympäristössä (Taylorin teoreema):

$$\begin{aligned} f(x) &= f(x^*) + (x - x^*) f'(x^*) + \frac{(x-x^*)^2}{2} f''(x^*) \\ &+ \ldots + \frac{(x-x^*)^{m-1}}{(m-1)!} f^{(m-1)}(x^*) + \frac{(x-x^*)^m}{m!} f^{(m)}(\zeta_x) \end{aligned} \quad , \tag{6.39}$$

missä tuttuun tapaan $\zeta_x \in [x, x^*]$.

Kaavan (6.38) tämä sievenee muotoon

$$f(x) = \frac{(x-x^*)^m}{m!} f^{(m)}(\zeta_x) \quad . \tag{6.40}$$

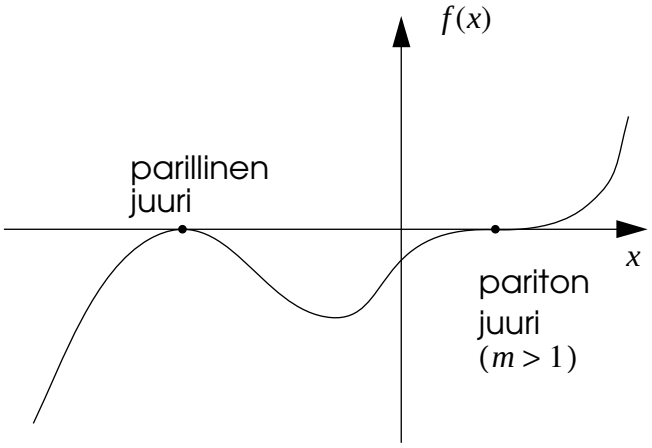
Jos siis merkitsemme $g(x) = f^{(m)}(\zeta_x)/m!$, saamme tuloksen (6.37), sillä $g(x^*) = f^{(m)}(x^*)/m! \neq 0$.

Yhtälöstä (6.40) voimme päätellä, että jos m on pariton, vaihtaa funktio merkkiä juuren kohdalla. Parillisilla juurilla funktio ainoastaan sivuaa x-akselia.

Lisäksi tietokoneen liukulukujen äärellinen tarkkuus voi aiheuttaa sen, että kahta erillistä, mutta toisiaan lähellä olevaa juurta (x_1^* ja x_2^*) luullaan yhdeksi monikertaiseksi juureksi:

$$\begin{aligned} f(x) &= (x - x_1^*)(x - x_2^*)g(x) \\ &= (x - x_1^*)[x - x_1^* - (x_2^* - x_1^*)]g(x) \\ &\approx (x - x_1^*)^2 g(x) \end{aligned}$$

jos $\left|x - x_1^*\right| \gg \left|x_2^* - x_1^*\right|$.



Kuva 6.9: Funktion $f(x)$ juuren kertaluku

Miten juuren kertaluku vaikuttaa tulosten tarkkuuteen?

Merkitään tietokoneen palauttamaa funktion arvoa juuren $x*$ lähellä symbolilla $\bar{f}$. Se on siis approksimaatio oikealle funktion arvolle:

$$\left| f(x) - \bar{f}(x) \right| \le \varepsilon \quad , \tag{6.41}$$

missä ε on jokin vakio. Oletetaan, että kone palauttaa funktiolle arvon nolla argumentin arvolla z:

$$\bar{f}(z) = 0 \quad . \tag{6.42}$$

Kuinka paljon z:ssa on virhettä? Jos juuren x^* kertaluku on m, voidaan käyttää kaavaa (6.40):

$$f(z) \approx (z - x^*)^m \, \frac{f^{(m)}(x^*)}{m!} \quad . \tag{6.43}$$

Yhtälön (6.41) mukaan $f(z)$ voi olla enintään ε eli

$$\mp\varepsilon \approx (z - x^*)^m \, \frac{f^{(m)}(x^*)}{m!} \quad , \tag{6.44}$$

josta saamme edelleen

$$\left|z - x^*\right| \approx \varepsilon^{1/m} \left| \frac{m!}{f^{(m)}(x^*)} \right|^{1/m} \quad . \tag{6.45}$$

Jos nyt ε on pieni ja m suuri, voi termi $\varepsilon^{1/m}$ olla hyvinkin suuri, ja seurauksena on tarkkuuden väheneminen. Eli monikertaisen juuren läheisyydessä pienikin virhe funktion arvossa voi aiheuttaa suuren virheen juuren paikassa. Tämän voi hahmottaa siten, että monikertaisen juuren läheisyydessä funktion kulkee lähes vaakasuoraan, ja pienikin siirros ylös tai alas muuttaa funktion ja x-akselin leikkauspisteen paikkaa paljon.

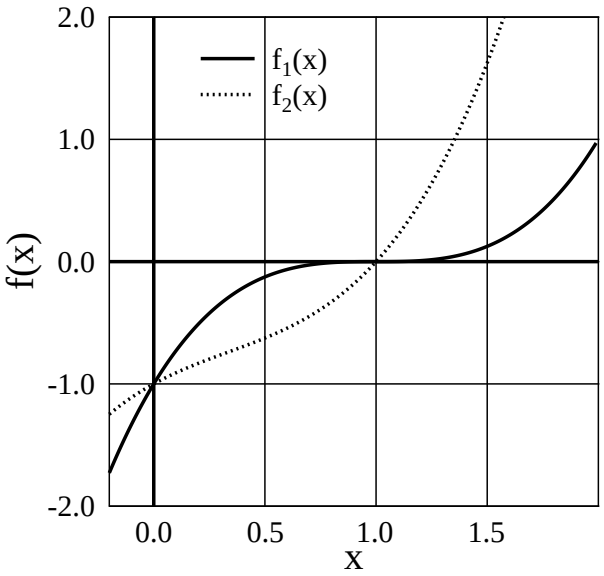
Myös yksinkertaisen juuren tapauksessa voi tulla ongelmia, jos ensimmäinen derivaatta on pieni:

$$|z - x^*| \approx \left| \frac{\varepsilon}{f'(x^*)} \right| \quad . \tag{6.46}$$

Otetaan esimerkkinä kahden funktion

$$\begin{aligned} f_1(x) &= (x-1)^3 \\ f_2(x) &= x^3 - x^2 + x - 1 \end{aligned} \quad , \tag{6.47}$$

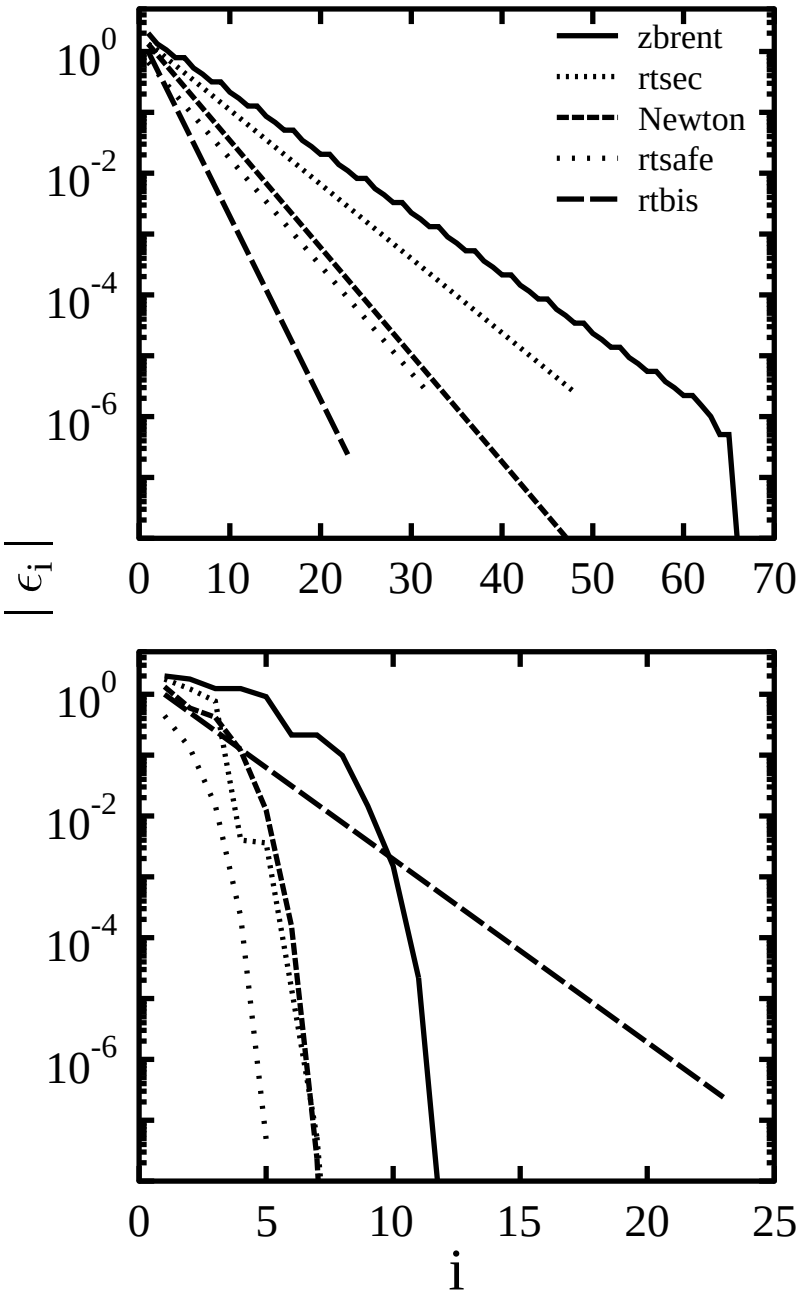
juuren etsintä NR:n rutiineilla `zbrent`, `rtsec`, `rtsafe`, `rtbis` sekä puhtaalla Newtonin menetelmällä.



Kuva 6.10: Kaavan (6.47) funktiot.

Molemmilla on yksi reaalijuuri $x^* = 1$. Funktion f_1 tapauksessa se on kolminkertainen ja f_2:lla yksinkertainen.

Alla on esitetty iteraatioaskeleen i funktiona juuren paikan virheen itseisarvo eli $|\varepsilon_i| = |x_i - 1|$.



Kuva 6.11: Kaavan (6.47) funktioiden f_1 (ylempi kuva) ja f_2 (alempi kuva) nollakohdan $x^* = 1$ etsintä.

Puolitussäännön suppenemisnopeuteen ei juuren pariton kertaluku vaikuta. Muut menetelmät taas hidastuvat, kun juuren kertaluku on ykköstä suurempi.

Toisaalta puolitussäännöllä ei parillisen kertaluvun juuria voi laskea ollenkaan, koska funktio ei vaihda merkkiä niissä.

Jos funktion derivaatta $f'(x)$ on laskettavissa, voidaan tehtävä muokata sellaiseksi, että puolitussääntöä voidaan käyttää. Tuttuun tapaan juuren x^* (kertaluku m) lähellä [vrt. (6.40)]

$$f(x) = (x - x^*)^m g(x) \tag{6.48}$$

ja

$$f'(x) = (x - x^*)^{m-1} G(x) \quad , \tag{6.49}$$

missä

$$G(x) = mg(x) + (x - x^*)g'(x) \quad . \tag{6.50}$$

ja

$$G(x^*) = mg(x^*) \neq 0 \quad . \tag{6.51}$$

Siis $f(x)$:n parillisen kertaluvun juuret ovat $f'(x)$:n parittoman kertaluvun juuria. Voidaan siis käyttää puolitussääntöä.

Jos merkitsemme

$$u(x) = \frac{f(x)}{f'(x)} \ , \tag{6.52}$$

niin lähellä juurta x^* pätee

$$u(x) = (x - x^*)H(x) \quad , \tag{6.53}$$

missä

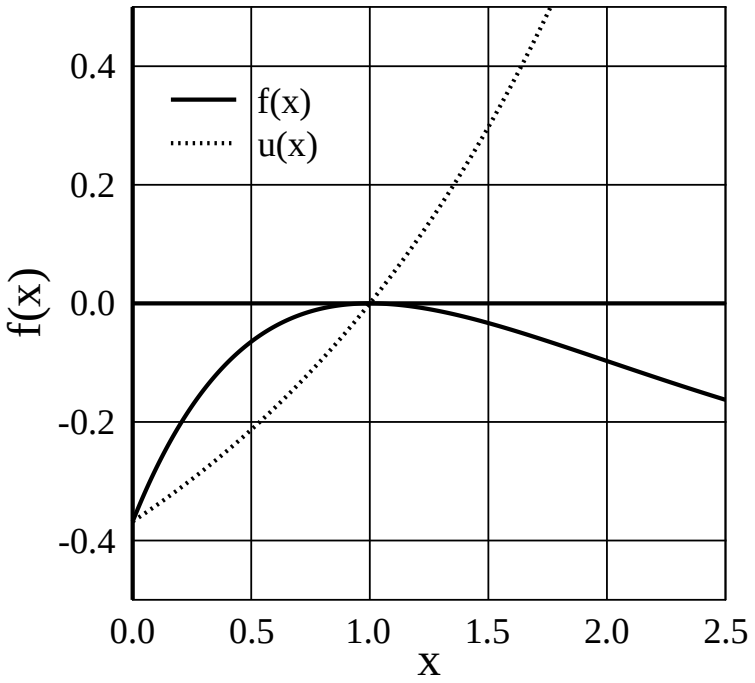
$$H(x) = \frac{g(x)}{G(x)} \quad \text{ja} \quad H(x^*) = \frac{1}{m} \ , \ m \neq 0 \ . \tag{6.54}$$

Eli monikertaisen juuren etsintä on korvattu yksinkertaisen juuren etsinnällä. On tietysti huomattava, että uudella funktiolla $u(x)$ on singulariteetti derivaatan $f'(x)$ nollakohdassa.

Esimerkkinä funktio

$$f(x) = xe^{-x} - e^{-1} \ . \tag{6.55}$$

Sillä on nollakohta $x^* = 1$. Sen kertaluku on 2, minkä voi todeta derivoimalla (6.55):n ja sijoittamalla $x = 1$. Tällöin $f'(1) = 0$, ja $f^{(m)}(1) \neq 0$, kun $m > 1$.



Kuva 6.12: Funktiot (6.55) ja (6.56).

Tässä muodossa (käyttäen edelläesiteltyjä rutiineja) ei funktion nollakohtaa pysty löytämään. Muodostamalla funktio $u(x)$ voidaan menetelmiä soveltaa:

$$u(x) = \frac{f(x)}{f'(x)} = \frac{xe^{-x} - e^{-1}}{e^{-x}(1-x)} \ . \tag{6.56}$$

Derivoimalla saadaan Newtonin menetelmässä tarvittava u' :

$$u'(x) = \frac{1 - e^{x-1}}{1-x} + \frac{x - e^{x-1}}{(1-x)^2} \quad . \tag{6.57}$$

Newtonin menetelmän iteraatiotulokset alla (alkuarvona $x = 2$):

```
i      xi               u(xi)              abs(ei)

1   1.281718172      0.7182818285      0.2817181715
2   1.025236738      0.1550732730      0.2523673838E-01
3   1.000211406      0.1272519113E-01  0.2114062102E-03
4   1.000000015      0.1057105538E-03  0.1489881019E-07
5   1.000000000      0.1490351257E-07  0.4702349621E-11
6   1.000000000       0.000000000      0.4702349621E-11
```

Myös sekanttimenetelmällä juuri saadaan muutamalla iteraatiolla käyttäen funktiota $u(x)$:

i	xi	u(xi)	abs(ei)
1	1.513440361	0.3069293249	0.5134403609
2	1.150395478	0.7911365175E-01	0.1503954779
3	1.024320717	0.1225954387E-01	0.2432071733E-01
4	1.001201434	0.6009576680E-03	0.1201434043E-02
5	1.000009719	0.4859604479E-05	0.9719171624E-05
6	1.000000004	0.000000000	0.3886029276E-08

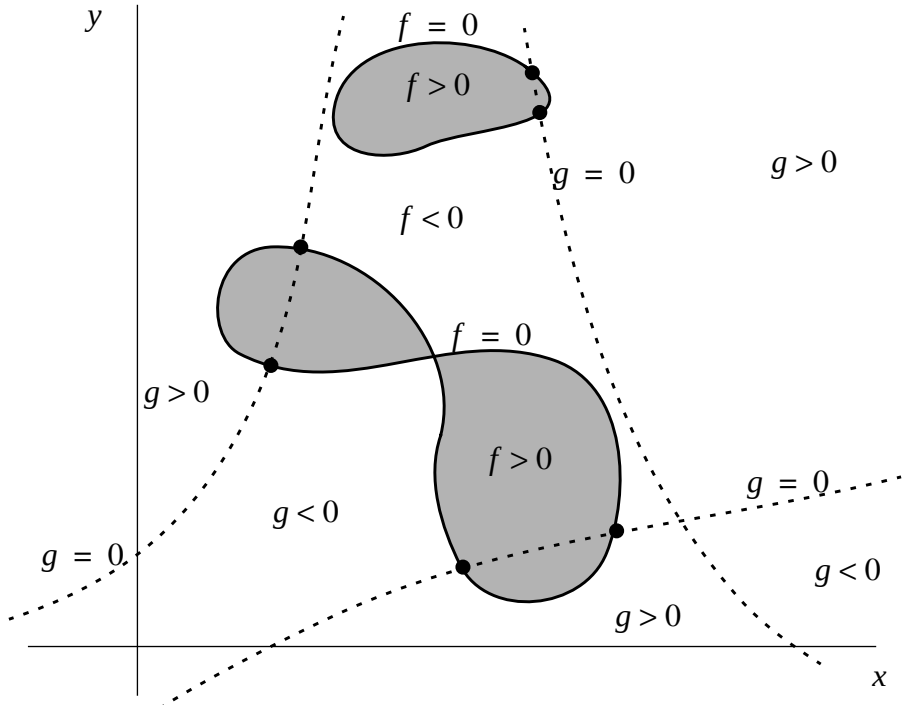
6.6 Yhtälöryhmät

Epälineaaristen yhtälöryhmien ratkaisu on huomattavasti vaikeampaa kuin yksittäisen yhtälön. Lähtökohtana voisi olla yksiulotteisten menetelmien yleistys. Puolitussääntöä ei ole mahdollista yleistää, mutta Newtonin menetelmä voidaan.

Tarkastellaan kaksiulotteista yhtälöryhmää

$$\begin{cases} f(x, y) = 0 \\ g(x, y) = 0 \end{cases} . \tag{6.58}$$

Funktiot jakavat tason alueisiin, joissa ne ovat negatiivisia tai positiivisia. Funktioiden nollakohdat muodostavat joukon käyriä tasossa ja näiden käyrien leikkauspisteet ovat yhtälöryhmän (6.58) ratkaisut. Koska yleensä funktioilla f ja g ei ole mitään yhteistä, eivät nämä yhteiset pisteet nollakohtakäyrillä ole millään lailla erityisiä funktioiden kannalta. Eli joudumme enemmän tai vähemmän käymään läpi kaikki nollakohtakäyrät, jos haluamme löytää kaikki ratkaisut.



Kuva 6.13: Kahden epälineaarisen yhtälön ratkaisut.

Yksinkertaisin epälineaaristen yhtälöryhmien ratkaisumenetelmä on yksiulotteisen Newtonin menetelmän yleistys.

Olkoon meillä N kappaletta N:n muuttujan funktioita, joiden nollakohdat tulee etsiä

$$F_i(x_1, x_2, \ldots, x_N) = 0 \quad , \quad i = 1, 2, \ldots, N \quad . \tag{6.59}$$

Vektorimuodossa tämä on

$$\mathbf{F}(\mathbf{x}) = \mathbf{0} \quad . \tag{6.60}$$

Kehitetään funktiot F_i Taylorin sarjaksi pisteen $\mathbf{x}$ ympärillä:

$$F_i(\mathbf{x} + \delta\mathbf{x}) = F_i(\mathbf{x}) + \sum_{j=1}^{N} \frac{\partial F_i}{\partial x_j} \delta x_j + O(\delta\mathbf{x}^2) \quad . \tag{6.61}$$

Otetaan käyttöön ns. Jacobin matriisi $\mathbf{J}$

$$J_{ij} = \frac{\partial F_i}{\partial x_j} \quad . \tag{6.62}$$

Tällöin (6.61) tulee muotoon

$$\mathbf{F}(\mathbf{x} + \delta\mathbf{x}) = \mathbf{F}(\mathbf{x}) + \mathbf{J} \cdot \delta\mathbf{x} + O(\delta\mathbf{x}^2) \quad . \tag{6.63}$$

Pudotamme korkeammat termit $O(\delta\mathbf{x}^2)$ ja asetamme $\mathbf{F}(\mathbf{x} + \delta\mathbf{x}) = 0$, jolloin saamme yhtälön korjaukselle $\delta\mathbf{x}$, joka vie kaikkia funktioita yhtäaikaa lähemmäs nollakohtaa:

$$\mathbf{J} \cdot \delta\mathbf{x} = -\mathbf{F}(\mathbf{x}) \quad . \tag{6.64}$$

Tämä yhtälö voidaan ratkaista esimerkiksi LU-hajotelmalla. Jokaisella iteraatiokierroksella siis lisätään korjaus vektoriin $\mathbf{x}$:

$$\mathbf{x}_{i+1} = \mathbf{x}_i + \delta\mathbf{x} \quad . \tag{6.65}$$

Iteraation aikana (kuten yksiulotteisessa tapauksessa) seurataan sekä funktioiden arvojen että nollakohtien suppenemista. Jos jompikumpi saavuttaa halutun tarkkuuden, ei iteraatiota kannata enää jatkaa.

NR:n rutiini `mnewt` tekee parametrin `ntrial` kertoman määrän iteraatioita (6.65) `N`-dimensioiselle yhtälöryhmälle, kun syöttötietoina annetaan aloituspiste `x` ja `tolx` ja `tolf`, jotka kertovat halutut nollakohtien ja funktioiden arvojen tarkkuudet. Kutsu C:ssä on muotoa

```
mnewt(ntrial,x,N,tolx,tolf);
```

Lisäksi tulee olla määritelty itse funktio **F** , jota kutsutaan nimellä `usrfun`:

```
usrfun(x,N,fvec,fjac);
```

Tässä vektoriin `fvec` palautetaan funktion arvo ja matriisiin `fjac` Jacobin matriisi.

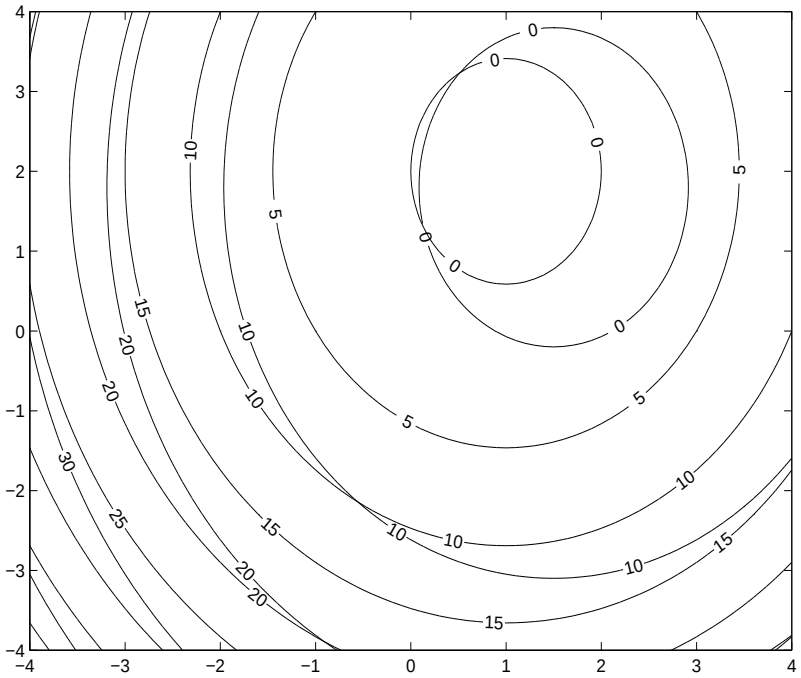
Otetaan esimerkiksi yhtälöryhmän

$$\begin{cases} F_1(x_1, x_2) = (x_1 - 1)^2 + \frac{1}{2}(x_2 - 2)^2 - 1 = 0 \\ F_2(x_1, x_2) = \left(x_1 - \frac{3}{2}\right)^2 + \frac{1}{2}\left(x_2 - \frac{9}{5}\right)^2 - 2 = 0 \end{cases} \tag{6.66}$$

ratkaisu.
Jacobin matriisin on nyt seuraavanlainen

$$\begin{aligned} J_{11} &= \frac{\partial F_1}{\partial x_1} = 2(x_1 - 1) & J_{12} &= \frac{\partial F_1}{\partial x_2} = x_2 - 1 \\ J_{21} &= \frac{\partial F_2}{\partial x_1} = 2\left(x_1 - \frac{3}{2}\right) & J_{22} &= \frac{\partial F_2}{\partial 2} = x_2 - \frac{9}{5} \end{aligned} \quad . \tag{6.67}$$

Alla funktioiden kuvaajat tasa-arvokäyrinä.



Kuva 6.14: Funktioiden (6.66) kuvaajat.

Nähdään, että löytyy kaksi juurta $\mathbf{x}_1^* \approx (0.1, 1.3)$ ja $\mathbf{x}_2^* \approx (0.5, 3.2)$.

Juuret etsittiin seuraavalla pääohjelmalla:

```
  #include <stdio.h>
  #include <math.h>
  #define NRANSI
  #include “nr.h”
  #include “nrutil.h”

  void usrfun(float *x,int n,float *fvec,float **fjac)
  {
    fvec[1] = (x[1]-1.0)*(x[1]-1.0) + 0.5*(x[2]-2.0)*(x[2]-2.0) - 1.0 ;
    fvec[2] = (x[1]-1.5)*(x[1]-1.5) + 0.5*(x[2]-1.8)*(x[2]-1.8) - 2.0 ;
    fjac[1][1] =  2.0*(x[1]-1.0);
    fjac[1][2] =  (x[2]-2.0);
    fjac[2][1] =  2.0*(x[1]-1.5);
    fjac[2][2] =  (x[2]-1.8);
  }

  #define NTRIAL 20
  #define TOLX 1.0e-6
  #define TOLF 1.0e-6
  #define N 2
  #define KMAX 3
  #define KKMAX 3
  #define SCALE 1.0

  int main(void)
  {
    int i,j,k,kk;
    float *x,*fvec,**fjac;

    fjac=matrix(1,N,1,N);
    fvec=vector(1,N);
    x=vector(1,N);
    printf(“\n   x10    x20          x1*       x2*           F1
F2\n\n”);
    for (kk=-KKMAX;kk<=KKMAX;kk++) {
      for (k=-KMAX;k<=KMAX;k++) {
        x[1]=SCALE*(kk+0.1);
        x[2]=SCALE*k;
        printf(“%6.2f %6.2f     “,x[1],x[2]);
        mnewt(NTRIAL,x,N,TOLX,TOLF);
        usrfun(x,N,fvec,fjac);
        printf(“%10.6f %10.6f  %15.6g %15.6g\n”,x[1],x[2],fvec[1],fvec[2]);
      }
    }
    printf(“\n”);
    return 0;
  }
  #undef NRANSI
```

Käännös ja ajo:

```
systems> cc -o newton2d newton2d.c mnewt.c nrutil.c lubksb.c ludcmp.c -lm
systems> newton2d
```

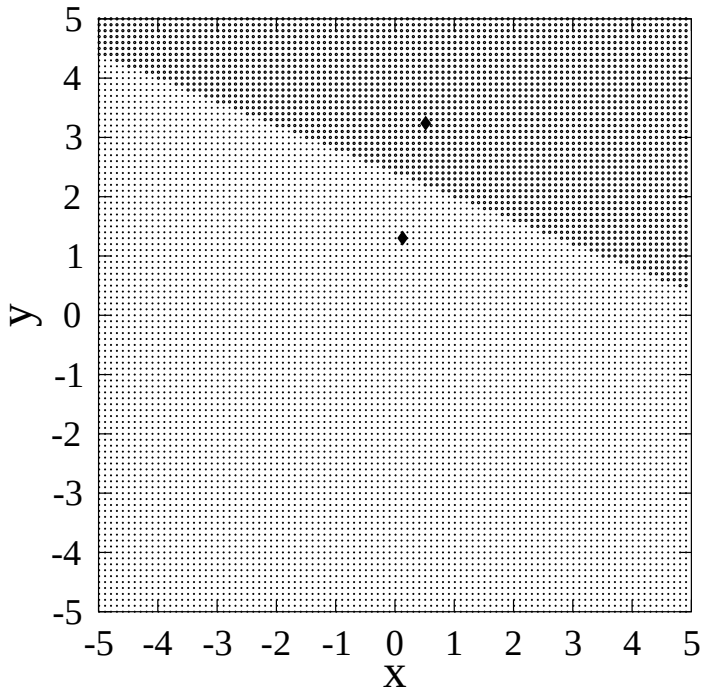
x10	x20	x1*	x2*	F1	F2
-2.90	-3.00	0.130363	1.301815	1.20086e-07	1.18298e-07
-2.90	-2.00	0.130363	1.301815	1.09384e-08	1.80909e-08
-2.90	-1.00	0.130363	1.301815	1.09384e-08	1.80909e-08
-2.90	0.00	0.130363	1.301815	1.20086e-07	1.18298e-07
-2.90	1.00	0.130363	1.301815	1.09384e-08	1.80909e-08
-2.90	2.00	0.130363	1.301815	1.09384e-08	1.80909e-08
-2.90	3.00	0.130363	1.301815	1.09384e-08	1.80909e-08
-1.90	-3.00	0.130363	1.301815	1.09384e-08	1.80909e-08
-1.90	-2.00	0.130363	1.301815	1.09384e-08	1.80909e-08
-1.90	-1.00	0.130363	1.301815	1.09384e-08	1.80909e-08
-1.90	0.00	0.130363	1.301815	1.09384e-08	1.80909e-08
-1.90	1.00	0.130363	1.301815	1.09384e-08	1.80909e-08
-1.90	2.00	0.130363	1.301815	1.09384e-08	1.80909e-08
-1.90	3.00	0.130363	1.301815	1.09384e-08	1.80909e-08
-0.90	-3.00	0.130363	1.301815	1.09384e-08	1.80909e-08
-0.90	-2.00	0.130363	1.301815	1.09384e-08	1.80909e-08
-0.90	-1.00	0.130363	1.301815	1.09384e-08	1.80909e-08
-0.90	0.00	0.130363	1.301815	1.09384e-08	1.80909e-08
-0.90	1.00	0.130363	1.301815	1.09384e-08	1.80909e-08
-0.90	2.00	0.130363	1.301815	1.09384e-08	1.80909e-08
-0.90	3.00	0.517785	3.238926	1.51653e-07	1.94568e-07
0.10	-3.00	0.130363	1.301815	1.09384e-08	1.80909e-08
0.10	-2.00	0.130363	1.301815	1.09384e-08	1.80909e-08
0.10	-1.00	0.130363	1.301815	1.09384e-08	1.80909e-08
0.10	0.00	0.130363	1.301815	1.09384e-08	1.80909e-08
0.10	1.00	0.130363	1.301815	1.46003e-07	1.59116e-07
0.10	2.00	0.130363	1.301815	1.09384e-08	1.80909e-08
0.10	3.00	0.517785	3.238926	3.89552e-07	4.20546e-07
1.10	-3.00	0.130363	1.301815	1.09384e-08	1.80909e-08
1.10	-2.00	0.130363	1.301815	1.09384e-08	1.80909e-08
1.10	-1.00	0.130363	1.301814	4.99362e-07	5.00555e-07
1.10	0.00	0.130363	1.301815	1.09384e-08	1.80909e-08
1.10	1.00	0.130363	1.301815	1.20086e-07	1.18298e-07
1.10	2.00	0.517785	3.238926	9.41686e-08	7.74793e-08
1.10	3.00	0.517785	3.238926	1.51653e-07	1.94568e-07
2.10	-3.00	0.130363	1.301815	1.20086e-07	1.18298e-07
2.10	-2.00	0.130363	1.301815	1.09384e-08	1.80909e-08
2.10	-1.00	0.130363	1.301815	1.09384e-08	1.80909e-08
2.10	0.00	0.130363	1.301814	2.5515e-07	2.59323e-07
2.10	1.00	0.130363	1.301814	3.64298e-07	3.59529e-07
2.10	2.00	0.517785	3.238926	1.51653e-07	1.94568e-07
2.10	3.00	0.517785	3.238926	1.51653e-07	1.94568e-07
3.10	-3.00	0.130363	1.301815	1.09384e-08	1.80909e-08
3.10	-2.00	0.130363	1.301815	1.09384e-08	1.80909e-08
3.10	-1.00	0.130363	1.301815	1.09384e-08	1.80909e-08
3.10	0.00	0.130363	1.301815	1.09384e-08	1.80909e-08
3.10	1.00	0.130363	1.301815	1.09384e-08	1.80909e-08
3.10	2.00	0.517785	3.238926	1.51653e-07	1.94568e-07
3.10	3.00	0.517785	3.238926	1.51653e-07	1.94568e-07

```
systems>
```

Huomaamme, että molemmat juuret löytyivät. Lisäksi tarkastelemalla nollakohtien ja funktioiden arvoja iteraatiokierrosten funktiona, havaitaan, että iteraatio suppenee jo muutaman (<10) kierroksen jälkeen. Tämän voi helposti todeta käskemällä rutiinin `mnewt` suorittaa kerralla vain yksi iteraatio:

```
...

x[1]=SCALE*(kk+0.1);
x[2]=SCALE*k;
printf("%8.4f %8.4f ",x[1],x[2]);
for (j=1;j<=NTRIAL;j++) {
     mnewt(1,x,N,TOLX,TOLF);
     usrfun(x,N,fvec,fjac);
     printf("%15.6f %15.6f %15.6f %15.6f\n",
               x[1],x[2],fvec[1],fvec[2]);
}

...
```

Käymällä läpi (x_1, x_2) -tasoa tarkemmin voimme tutkia, kumpaan juureen tietystä pisteestä lähtenyt iteraatio suppenee.



Kuva 6.15: Yhtälöparin (6.66) iteraation suppeneminen. Tummemmalta alueelta aloitettu iteraatio suppenee ylempänä sijaitsevaan juureen vaaleammalta aloitettu alempaan.

Tulemme huomaamaan, että skalaarifunktion minimoinnille on löydettävissä melko tehokkaita algoritmeja. Eikö näitä voisi käyttää epälineaaristen yhtälöryhmien ratkaisuun? Minimoinnissahan etsitään funktion gradientin nollakohtaa.

Erona epälineaarisiin yhtälöryhmiin on se, että minimoinnissa on luonnollinen suunta (‘alamäkeen’), johon iteraation kuluessa kuljetaan. Nollakohtien etsinnässä ei tällaista suuntaa ole.

Toinen vaihtoehto voisi olla nollakohtien etsimisen muuttaminen minimointitehtäväksi. Yhtälöryhmän (6.60) ratkaisu $\mathbf{x}^*$ minimoi summan

$$F(\mathbf{x}) = \sum_{i=1}^{N} [F_i(\mathbf{x})]^2 \quad , \tag{6.68}$$

jolloin voisimme käyttää epälineaarisen pienimmän neliösumman ongelmaan tarkoitettuja menetelmiä. Ongelmana on se, että funktiolle (6.68) voi löytyä paikallinen minimi, joka ei ole yhtälöryhmän ratkaisu.

Tehokkaampia menetelmiä kuin suora Newtonin menetelmä saadaan yhdistämällä tämä yllämainittuun funktion F minimointiin. Tällöin käytetään hyväksi Newtonin menetelmän nopeaa suppenemista lähellä juurta ja minimointitehtävän parempaa kykyä löytää minimi (‘lähes’) riippumatta alkupisteestä.

Newtonin menetelmässä iteraatioaskel yhtälön

$$\mathbf{F}(\mathbf{x}) = 0 \tag{6.69}$$

ratkaisemiseksi tehdään seuraavasti

$$\mathbf{x}_{i+1} = \mathbf{x}_i + \delta\mathbf{x} \quad , \tag{6.70}$$

missä korjaus $\delta\mathbf{x}$ on

$$\delta\mathbf{x} = -\mathbf{J}^{-1} \cdot \mathbf{F} \quad . \tag{6.71}$$

Yhdistelmämenetelmässä tehdään aluksi Newtonin iteraatio ja se hyväksytään tietyllä kriteerillä. Kriteerinä on se, että $|\mathbf{F}|^2 = \mathbf{F} \cdot \mathbf{F}$ pienenee. Tämähän on sama vaatimus kuin funktion

$$f = \frac{1}{2}\mathbf{F} \cdot \mathbf{F} \tag{6.72}$$

minimoinnissa. Jokainen (6.69):n ratkaisu minimoi (6.72):n, mutta f:llä voi olla lokaaleja minimejä, jotka eivät ole $\mathbf{F}$:n nollakohtia. Eli pelkkää minimointalgoritmia ei voi käyttää.

Newton-askel (6.71) on funktion f kannalta alamäkeen, koska

$$\nabla f \cdot \delta\mathbf{x} = (\mathbf{F} \cdot \mathbf{J}) \cdot (-\mathbf{J}^{-1} \cdot \mathbf{F}) = -\mathbf{F} \cdot \mathbf{F} < 0 \quad . \tag{6.73}$$

Nyt jokaisella iteraatioaskeleella lasketaan $\delta\mathbf{x}$ yhtälöstä (6.1). Jos askel ei pienennä funktiota f, mennään Newtonin askelta taaksepäin niin pitkälle, että f pienenee. Tämä tulee varmasti

jossain vaiheessa vastaan yhtälön (6.73) perusteella.

Eli uusi askel on

$$\mathbf{x}_{i+1} = \mathbf{x}_i + \lambda\delta\mathbf{x} \quad , \quad 0 < \lambda \leq 1 \quad , \tag{6.74}$$

ja λ valittu siten, että $f(\mathbf{x}_i + \lambda\delta\mathbf{x})$ pienenee riittävästi.

Sopiva kriteeri riittävälle pienenemiselle on esimerkiksi, että nykyisellä askeleella f pienenee keskimäärin vähintään jonkun osuuden askeleen alun pienenemisnopeudesta:

$$f(\mathbf{x}_{i+1}) \leq f(\mathbf{x}_i) + \alpha\nabla f \cdot (\mathbf{x}_{i+1} - \mathbf{x}_i) \quad , \tag{6.75}$$

missä $0 < \alpha < 1$.

NR:n ohjelmista löytyy rutiini `newt`, joka ratkaisee yhtälöryhmän edelläesitetyllä tavalla. Sen kutsu on

```
newt(x,N,&check,funcv);
```

missä `x` on alkuarvaus juurille (`N`-alkioinen vektori), `check` on lippu, joka asetetaan, jos aliohjelmassa ilmeni jotain ongelmia ja `funcv` on haluttu funktio $\mathbf{F}$. Jacobin matriisi lasketaan rutiinissa `fdjac` numeerisesti erotusosamääränä.

Seuraavalla sivulla esimerkkinä yhtälöryhmän

$$\begin{cases} x_1^2 + x_2^2 - 2 = 0 \\ e^{x_1 - 1} + x_2^3 - 2 = 0 \end{cases}$$

ratkaiseminen (yksi juuri on $x_1 = x_2 = 1$)

Ohjelmakoodi:

```
#include <stdio.h>
#include <math.h>
#define NRANSI
#include “nr.h”
#include “nrutil.h”

void funcv(int n,float x[],float f[])
{
  f[1]=SQR(x[1])+SQR(x[2])-2.0;
  f[2]=exp(x[1]-1.0)+x[2]*SQR(x[2])-2.0;
}
#define N 2
#define KMIN 2
#define KKMIN 0.5
#define KMAX 4
#define KKMAX 4
#define SCALE 0.5
int main(void)
{
  int i,check,k,kk;
  float *x,*f;
  x=vector(1,N);
  f=vector(1,N);
  for (kk=KKMIN;kk<=KKMAX;kk++) {
    for (k=KMIN;k<=KMAX;k++) {
      x[1]=SCALE*kk;
      x[2]=SCALE*k;
      printf(“%8.5f %8.5f     “,x[1],x[2]);
      newt(x,N,&check,funcv);
      funcv(N,x,f);
      printf(“%10.6f %10.6f  %15.6g %15.6g\n”,x[1],x[2],f[1],f[2]);
      if (check) printf(“Convergence problems.\n”);
    }
  }
  return 0;
}
```

Käännös ja ajo:

```
  nrnewt> cc -o xnewt xnewt.c newt.c ludcmp.c lubksb.c lnsrch.c fdjac.c nru-
til.c fmin.c -lm
  nrnewt> xnewt
  0.00000  1.00000      -0.713748   1.220887      3.09046e-07    -1.83345e-08
  0.00000  1.50000      -0.713757   1.220893       2.9386e-05     2.70813e-05
  0.00000  2.00000      -0.713747   1.220890      6.34566e-06     1.40348e-05
  0.50000  1.00000       1.000000   1.000000      2.38419e-07     1.19209e-07
  0.50000  1.50000       1.000000   1.000001      8.34466e-07     2.08616e-06
  0.50000  2.00000       1.000000   1.000000      2.38419e-07     5.96047e-07
  1.00000  1.00000       1.000000   1.000000                0               0
  1.00000  1.50000       0.999999   1.000001      1.07289e-06     2.68221e-06
  1.00000  2.00000       1.000000   1.000000      3.57628e-07      8.9407e-07
  1.50000  1.00000       1.000002   1.000000      4.52996e-06     3.21865e-06
  1.50000  1.50000       0.999999   1.000002      1.43051e-06      3.8147e-06
  1.50000  2.00000       1.000000   1.000000      4.76837e-07     1.19209e-06
  2.00000  1.00000       0.999998   1.000003      3.09945e-06     7.74863e-06
  2.00000  1.50000       0.999994   1.000013      1.32324e-05     3.28427e-05
  2.00000  2.00000       0.999999   1.000001      1.07289e-06     2.68221e-06
```

Ylläesitetyn Newtonin menetelmän haitta on se, että laskennassa tarvitaan Jacobin matriisia. Aina ei kuitenkaan ole saatavilla analyyttista lauseketta derivaatoille $\partial F_i / \partial x_j$ ja funktion laskeminen voi olla niin raskasta, että ei kannata laskea derivaattoja numeerisesti.

Tällaisessa tapauksessa on edullista käyttää sekanttimenetelmän yleistystä useampaan dimensioon. NR:n vastaava rutiini on `broydn`.

7 Funktion minimointi

7.1 Yleistä

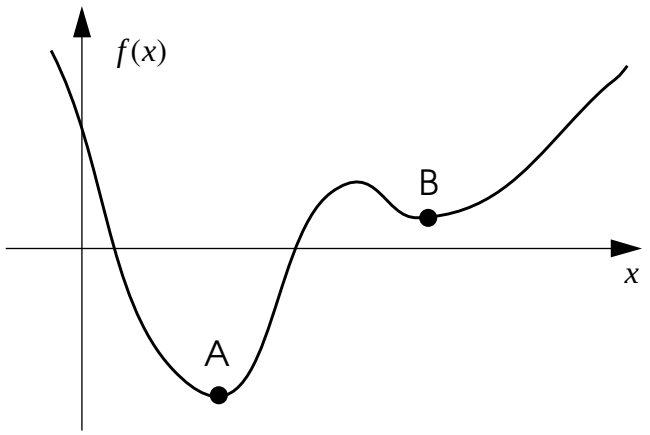
Olkoon meillä joukko atomeja paikoissa $\{\mathbf{r}_i\}$, $i = 1, 2, \ldots, N$. Atomit vuorovaikuttavat potentiaalienergian välityksellä. Koko systeemin potentiaalienergia voidaan lausua muodossa $V(\{\mathbf{r}_i\})$.

Nyt on löydettävä sellainen atomikonfiguraatio $\{\mathbf{r}_i\}$, joka minimoi potentiaalienergian $V(\{\mathbf{r}_i\})$. Eli kysymys on: Millaisen aineen atomitason rakenteen potentiaalienergiafunktio V ennustaa olevan stabiilein?[1]

Funktion minimoinnin ongelma on siis: Etsi funktion f sellainen argumentti, jolla funktiolla on pienin mahdollinen arvo. Funktiolla voi tietysti olla yksi tai useampi muuttujia. Lisäksi funktio ja muuttujat voivat saada joko jatkuvia tai diskreettejä arvoja. Funktion f maksimointi voidaan luonnollisesti palauttaa minimointiin käyttämällä funktiota $-f$.

Tehtävä on tietysti ratkaistava mahdollisimman tehokkaasti, mikä yleensä tarkoittaa mahdollisimman pientä määrää funktion arvojen ja mahdollisesti sen osittaisderivaattojen laskua.

Yleensä ollaan kiinnostuttu globaalista minimistä. Funktiolla voi olla myös lokaaleja minimejä.



Kuva 7.1: Funktion globaali (A) ja lokaali (B) minimi.

Lokaali minimi useinkin löytyy helposti, mutta globaalin minimin löytäminen on melko vaikea tehtävä. Yksi tapa on käyttää erilaisia lähtöarvoja funktion muuttujille ja valita näistä saaduista (lokaaleista) minimeistä pienin.

Myös viimeaikoina kehitettyä ns. simuloitua jäähdytysmenetelmää (simulated annealing) on menestyksellisesti käytetty globaalin minimin etsimisessä. Sen etuna on lisäksi se, että sitä voidaan soveltaa lähes yhtä helposti sekä diskreetteihin että jatkuviin tehtäviin. Toisaalta, koska kyseessä on stokastinen menetelmä, löydetty ratkaisu on ainoastaan todennäköisesti oikea.

1. Tarkkaan ottaen pitäisi lisätä määre *nollalämpötilassa*.

Seuraavassa esitellään muutamia jatkuvien funktioiden ja muuttujien minimointimenetelmiä.

Menetelmän valinnassa usein merkittävää on se, ovatko funktion osittaisderivaatat helposti laskettavissa ja kompensoiko niiden aikaansaama nopeutus niiden laskemiseen käytetyn ajan.

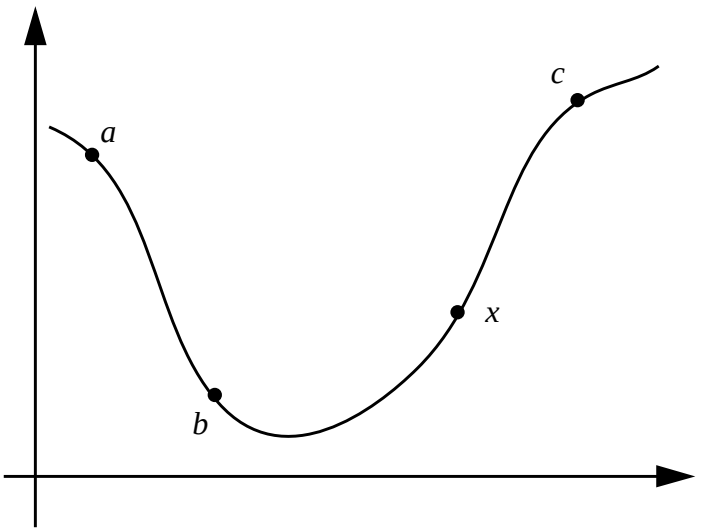
7.2 Yksidimensioiset minimointialgoritmit

Tehtävänä on siis etsiä funktion $f(x)$ minimi. Käytössä olevat algoritmit useimmiten toimivat siten, että aluksi valitaan hakusuunta, jonka jälkeen käytetään jotain menetelmää minimin löytämiseksi kuljettaessa tähän suuntaan.

7.2.1 Kultainen leikkaus

Funktion nollakohdan etsimisessä puolitussäännöllä nollakohta ensin eristettiin tietylle välille $[a, b]$ ja väliä pienennettiin jakamalla se kolmannella pisteellä x ja ottamalla uudeksi väliksi se väleistä $[a, x]$ ja $[x, b]$, jonka sisään nollakohta jäi. Keskimäärin optimaalinen pisteen x paikka on välin $[a, b]$ puolivälissä.

Kultainen leikkaus on tämän idean sovellus minimin etsimisessä. Nollakohdan eristämiseen tietylle välille riittää kaksi pistettä, joissa funktio on erimerkkinen. Minimin eristäminen taas vaatii kolme pistettä $a < b < c$ siten, että $f(b) < f(a)$ ja $f(b) < f(c)$. Puolitussäännön kanssa analogisesti valitaan nyt uusi piste x joko väliltä $[a, b]$ tai $[b, c]$. Oletetaan, että valinta tehdään väliltä $[b, c]$. Jos nyt $f(b) < f(x)$, uusi lukukolmikko on (a, b, x) (kuvan 7.2 tapaus) muutoin se on (b, x, c).



Kuva 7.2: Minimin etsintä kultaisen leikkauksen menetelmällä.

Aina siis valitaan se lukukolmikko, jonka keskimmäisellä luvulla on pienin funktion f arvo. Tätä iteraatiota jatketaan, kunnes päästään haluttuun tarkkuuteen, eli väli $[a, c]$ on riittävän pieni.

Mikä sitten on riittävän pieni?

Jos minimi sijaitsee paikassa $x = b$, voisi ajatella, että sen pystyy koneella paikantamaan välille $[(1-\varepsilon)b, (1+\varepsilon)b]$, missä ε on liukulukujen tarkkuus (ks. kappale 3.2). Asia ei kuitenkaan ole näin.

Lähellä minimiään funktio voidaan lausua muodossa (1. derivaatta on lähellä nollaa)

$$f(x) \approx f(b) + \frac{1}{2} f''(b)(x-b)^2 \ . \tag{7.1}$$

Yhtälön oikean puolen toinen termi on merkityksetön ensimmäiseen verrattuna, kun

$$|x-b| < \sqrt{\varepsilon}|b| \sqrt{\frac{2|f(b)|}{b^2 f''(b)}} \ . \tag{7.2}$$

Jälkimmäisen neliöjuurilausekkeen voidaan olettaa olevan suuruusluokkaa 1 useimmille funktioille. Eli minimin paikka b saadaan enimmillään tarkkuudella $\sqrt{\varepsilon} \cdot b$. Usein miniminetsintärutiineissa annetaan parametrina tarkkuus, jolla minimin paikka etsitään. Ylläolevan perusteella on yleensä hyödytöntä antaa parametrille arvo joka on pienempi kuin $\sqrt{\varepsilon}$.

Vielä on kehitettävä algoritmi, jolla valitaan uusi piste x, kun tiedossa on kolme pistettä a, b ja c.

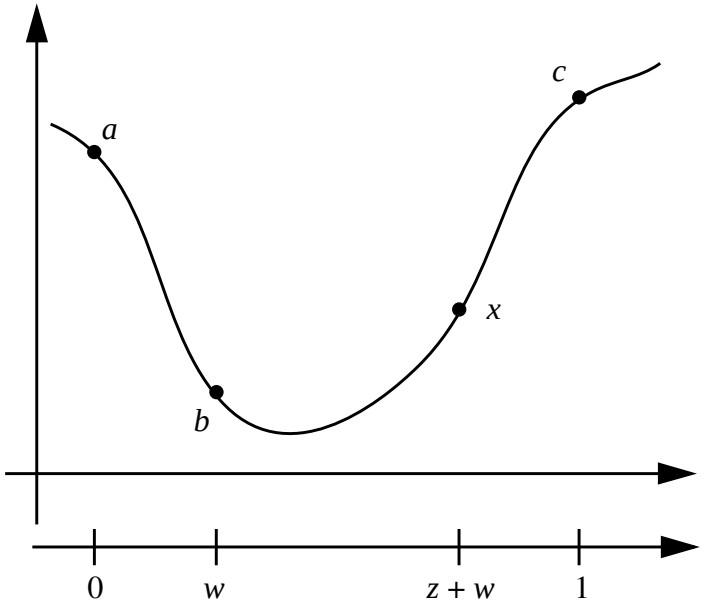
Merkitään w:llä b:n suhteellista paikkaa välillä $[a, c]$

$$w = \frac{b-a}{c-a} \quad , \quad 1-w = \frac{c-b}{c-a} \ . \tag{7.3}$$

Samalla tavalla olkoon uuden pisteen x suhteellinen paikka b:stä mitattuna z:

$$z = \frac{x-b}{c-a} \ . \tag{7.4}$$

Nyt seuraavan välin pituus on (edelleen suhteellisissa yksiköissä) joko $z + w$ (a, b, x) tai $1 - w$ (b, x, c).



Kuva 7.3: Seuraavan pisteen valinta kultaisen leikkauksen menetelmässä.

Optimaalisen (tai vähiten pessimistisen valinnan) saavutamme, kun valitsemme pisteen x siten, että nämä kaksi vaihtoehtoa ovat yhtäsuuret

$$z + w = 1 - w \Rightarrow z = 1 - 2w \ . \tag{7.5}$$

Nähdään, että uusi piste x on symmetrinen pisteen b kanssa, eli $|b - a| = |x - c|$. Tästä seuraa, että x sijaitsee suuremmassa väleistä $[a, b]$ ja $[b, c]$.

Pisteen x paikka kyseisen välin sisällä saadaan seuraavasti.

Oletamme, että aikaisemmilla iteraatiokierroksilla olemme käyttäneet samaa algoritmia. Tästä seuraa se, että x:n etäisyys b:stä verrattuna väliin $[b, c]$ (jos se on nyt suurempi väli) on sama kuin b:n etäisyys a:sta verrattuna väliin $[a, b]$. Eli suhteellisissa yksiköissä saamme yhtälön

$$\frac{z}{1 - w} = \frac{w}{1} \ . \tag{7.6}$$

Yhdistämällä (7.5) ja (7.6) saamme toisen asteen yhtälön

$$w^2 - 3w + 1 = 0 \ , \tag{7.7}$$

jonka ratkaisu on

$$w = \frac{3 - \sqrt{5}}{2} \approx 0.381966 \ . \tag{7.8}$$

(Plusmerkistä saatava juuri on suurempi kuin 1.)

Siis optimaalinen pistekolmikko (a, b, c) on sellainen, jossa b on etäisyydellä $0.381966(c - a)$ välin $[a, c]$ toisesta päätepisteestä ja etäisyydellä $0.618034(c - a)$ toisesta.

Algoritmi on siis seuraava:

Olkoon meillä luvut a, b ja c edelliseltä iteraatiokierrokselta. Valitaan uusi piste x siten, että se on suhteellisen etäisyyden 0.381966 päässä pisteestä b ja suuremman välin (joko $[a, b]$ tai $[b, c]$) suuntaan. Uudeksi pistekolmikoksi valitaan se, jonka keskipisteessä funktiolla on pienempi arvo.

Jokaisella iteraatiokierroksella tarkasteltava väli pienenee siis 0.618034:een osaan edellisestä. Suppenemiskertaluku on yksi eli suppeneminen on lineaarista.

NR:n ohjelmista löytyy funktio `golden`, jolla etsitään funktion minimi edelläkuvatulla menetelmällä. Sen kutsu on muotoa (C:ksi)

```
fmin=golden(ax,bx,cx,func,TOL,&xmin);
```

Funktio palauttaa arvonaan funktion `func` minimiarvon ja parametrissa `xmin` minimin sijainnin. Syöttötietoina tarvitaan aloitusväli `ax` ,`bx`, `cx`.

Alla esimerkkinä Besselin funktion $J_0(x)$ minimin etsintä alkaen väliltä $[0, 6]$.
Pääohjelma on seuraavanlainen

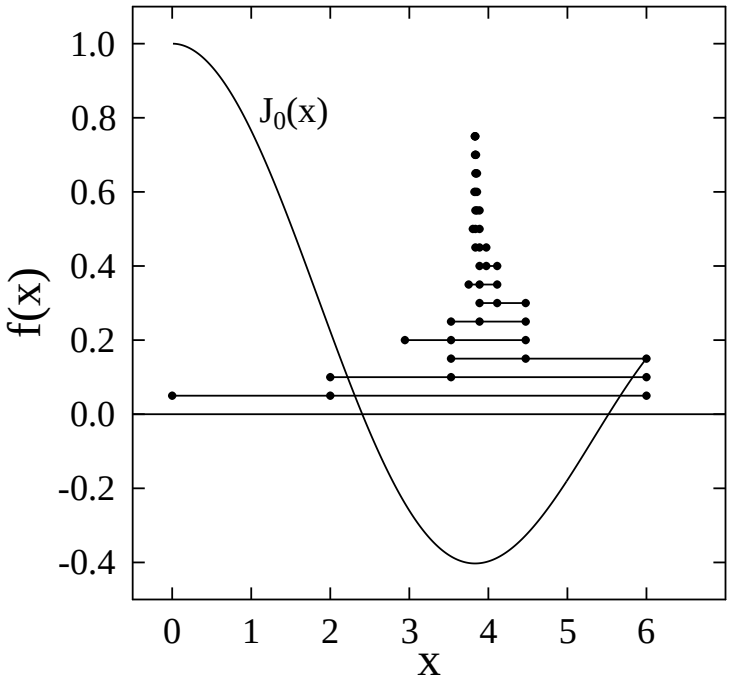
```
#include <stdio.h>
#include <math.h>
#define NRANSI
#include "nr.h"

float goldenx(float ax, float bx, float cx, float (*f)(float),
              float tol, float *xmin, int *j);

float func(float x) {return bessj0(x);}

int main(int argc, char **argv)
{
  int i,j;
  float ax,bx,cx,xmin,gold,tol;

  if (argc!=5 ) {
    fprintf(stderr,"Usage: %s tol a b c\n",argv[0]);
    return (1);
  }
  tol=atof(*++argv);
  ax=atof(*++argv);
  bx=atof(*++argv);
  cx=atof(*++argv);
  gold=goldenx(ax,bx,cx,func,tol,&xmin,&j);
  printf("Minimum with tolerance %8.1e after %d iterations is %f, f=
            %f\n",tol,j,xmin,gold);
  return 0;
}
```

(NR:n rutiinin nimi muutettu nimeksi `goldenx`, koska siihen lisättiin ulostuloparametriksi tehtyjen iteraatioiden lukumäärä.)

Käännös ja ajo

```
golden> cc -o xgolden xgolden.c golden.c nrutil.c bessj0.c -lm
xgolden.c:
golden.c:
nrutil.c:
bessj0.c:
golden> xgolden 1e-6 0 1 6
Minimum with tolerance  1.0e-06 after 30 iterations is 3.831540,
f= -0.402759
golden> xgolden 1e-8 0 1 6
Minimum with tolerance  1.0e-08 after 100 iterations is 3.831540,
f= -0.402759
golden>
```

Alla kuva iteraation kulusta rutiinin `golden` sisällä.



Kuva 7.4: Besselin funktion J_0 minimin etsintä kultaisen leikkauksen menetelmällä. Pallot kuvaavat iteraatiovälin muuttumista.

Kultaisen leikkauksen menetelmä on kuten puolitusmenetelmä nollakohtien etsinnässä: hitaasti suppeneva, mutta suppenee melko varmasti.

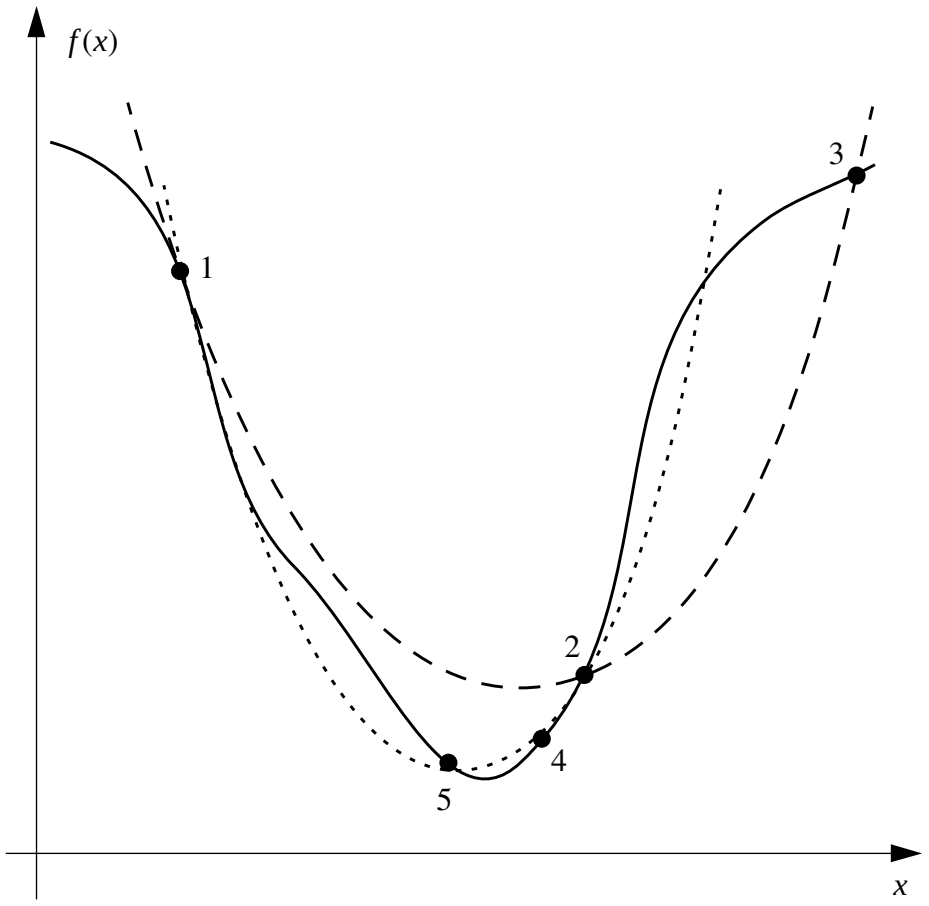
7.2.2 Parabolinen interpolointi

Yhtälön (7.1) mukaan funktio $f(x)$ voidaan tarpeeksi lähellä minimiään approksimoida paraabeliksi. Tätä voidaan käyttää hyväksi minimin etsinnässä.

Olkoon meillä taas kolme pistettä a, b ja c $(a < b < c)$. Sovitetaan näiden pisteiden kautta paraabeli ja etsitään sen minimi. Tulokseksi minimin paikalle x-akselilla saadaan

$$x = b - \frac{1}{2}\frac{(b-a)^2[f(b)-f(c)] - (b-c)^2[f(b)-f(a)]}{(b-a)[f(b)-f(c)] - (b-c)[f(b)-f(a)]} \quad . \tag{7.9}$$

Yhtälö ei toimi, jos pisteet ovat samalla suoralla. Tällöin nimittäjä menee nollaksi. Lisäksi yhtälö ei erottele minimin ja maksimin välillä.



Kuva 7.5: Minimin etsintä paraabelisovituksella. Katkoviiva on pisteiden 1, 2 ja 3 kautta sovitettu paraabeli. Tämän minimi on pisteessä 4. Pisteviiva on pisteiden 1, 4 ja 2 kautta sovitettu paraabeli, jonka minimi on pisteessä 5.

Paras strategia on tälläkin kertaa yhdistää kultaisen leikkauksen menetelmä paraboliseen interpolointiin.

Brentin menetelmässä yritetään ensin paraabelisovitusta. Jos sen antama minimi ei osu kyseisen iteraatioaskeleen väliin tai, jos iteraatio ei konvergoi tarpeeksi nopeasti, sovelletaan kultai-

sen leikkauksen menetelmää. Menetelmässä tarvitaan kohtuullinen määrä kirjanpitoa, joten jätämme sen toteutuksen yksityiskohtien tarkastelun. Halukkaat voivat tietysti katsoa NR:n rutiinin `brent` listausta. Sen kutsurakenne on samanlainen kuin funktiolla `golden`.

Alla on ohjelma, joka vertailee näiden kahden menetelmän suppenemista (rutiinin nimi on `brentx`, koska siihen on lisätty ulostuloparametriksi suoritettujen iteraatioiden lukumäärä):

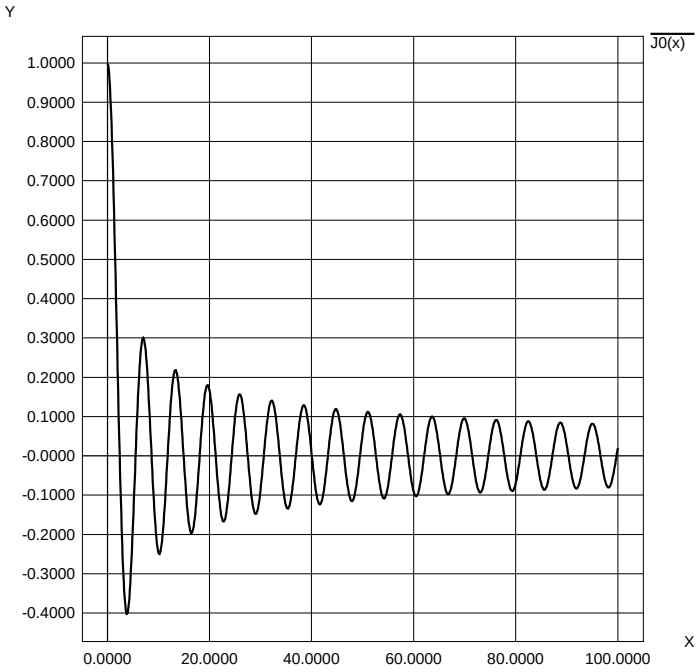
```
#include <stdio.h>
#include <math.h>
#define NRANSI
#include "nr.h"

float goldenx(float ax, float bx, float cx, float (*f)(float),
              float tol, float *xmin, int *j);
float brentx(float ax, float bx, float cx, float (*f)(float), float tol,
         float *xmin, int *j);

float func(float x) {return bessj0(x);}

int main(int argc, char **argv)
{
  int i,j;
  float ax,bx,cx,xmin,gold,brent,tol;

  if (argc!=5 ) {
    fprintf(stderr,"Usage: %s tol a b c\n",argv[0]);
    return (1);
  }
  tol=atof(*++argv);
  ax=atof(*++argv);
  bx=atof(*++argv);
  cx=atof(*++argv);
  gold=goldenx(ax,bx,cx,func,tol,&xmin,&j);
  printf("Minimum with tolerance %8.1e after %d iterations is %f,
          f= %f\n",tol,j,xmin,gold);
  brent=brentx(ax,bx,cx,func,tol,&xmin,&j);
  printf("Minimum with tolerance %8.1e after %d iterations is %f,
      = %f\n",tol,j,xmin,brent);
  return 0;
}
```

Ja seuraavassa ajotulokset:

```
golden> cc -o xgolden xgolden.c golden.c brent.c nrutil.c bessj0.c -lm
golden> xgolden 1e-4 0 1 6
Minimum with tolerance  1.0e-04 after 21 iterations is 3.831589,
f= -0.402759
Minimum with tolerance  1.0e-04 after 9 iterations is 3.831650,
f= -0.402759
golden> xgolden 1e-6 0 1 6
Minimum with tolerance  1.0e-06 after 30 iterations is 3.831540,
f= -0.402759
Minimum with tolerance  1.0e-06 after 12 iterations is 3.831658,
f= -0.402759
golden> xgolden 1e-7 0 1 6
Minimum with tolerance  1.0e-07 after 35 iterations is 3.831540,
f= -0.402759
Minimum with tolerance  1.0e-07 after 17 iterations is 3.831651,
f= -0.402759
golden>
```

Kuten odotettavissa on, Brentin menetelmä suppenee nopeammin.

Besselin funktiolla $J_0(x)$ on monia lokaaleja minimejä, joihin voidaan päätyä globaalin minimin sijasta.



Kuva 7.6: Besselin funktio $J_0(x)$.

Alla muutama ajo:

```
golden> xgolden 1e-7 0  50 100
Minimum with tolerance  1.0e-07 after 36 iterations is 35.332024,
     f= -0.134211
Minimum with tolerance  1.0e-07 after 18 iterations is 35.332317,
     f= -0.134211
golden> xgolden 1e-7 0 10 100
Minimum with tolerance  1.0e-07 after 39 iterations is 10.173343,
     f= -0.249705
Minimum with tolerance  1.0e-07 after 22 iterations is 10.173368,
     f= -0.249705
golden> xgolden 1e-7 0 90 100
Minimum with tolerance  1.0e-07 after 36 iterations is 47.901138,
     f= -0.115274
Minimum with tolerance  1.0e-07 after 17 iterations is 35.332199,
     f= -0.134211
golden> xgolden 1e-7 50 75 100
Minimum with tolerance  1.0e-07 after 33 iterations is 60.469193,
     f= -0.102601
Minimum with tolerance  1.0e-07 after 21 iterations is 60.469463,
     f= -0.102601
```

7.2.3 Newtonin menetelmä

Edelläesitetty menetelmät käyttivät vain funktion $f(x)$ arvoja minimin etsinnässä. Jos myös funktion derivaattoja $f'(x)$ on laskettavissa tai muuten saatavilla, voidaan niitä käyttää hyväksi.

Ideana on siis approksimoida funktiota $f(x)$, poynomilla ja laskea tämän minimi, joka otetaan uudeksi arvioksi funktion minimille. Koska suoralla ei ole minimiä, approksimoidaan funktiota paraabelilla. Kirjoitetaan funktio Taylorin sarjana iteraatiokierroksen k tuloksen x_k ympärillä

$$f(x_k + p) = f(x_k) + p\,f'(x_k) + \frac{1}{2}p^2 f''(x_k) + \ldots \quad . \tag{7.10}$$

Alkuperäistä minimointitehtävää approksimoidaan Taylorin sarjan minimoinnilla:

$$\begin{aligned} f(x^*) &= \min_x[f(x)] \quad . \\ &= \min_p[f(x_k + p)] \\ &= \min_p\left[f(x_k) + p\,f'(x_k) + \frac{1}{2}p^2 f''(x_k) + \ldots\right] \\ &\approx \min_p\left[f(x_k) + p\,f'(x_k) + \frac{1}{2}p^2 f''(x_k)\right] \end{aligned} \tag{7.11}$$

Paraabelin minimi löydetään (toivon mukaan) asettamalla sen derivaatta p:n suhteen nollaksi. Tästä saadaan

$$p = -\frac{f'(x_k)}{f''(x_k)} \quad . \tag{7.12}$$

Eli jokaisella iteraatiokierroksella päivitetään minimin paikkaa:

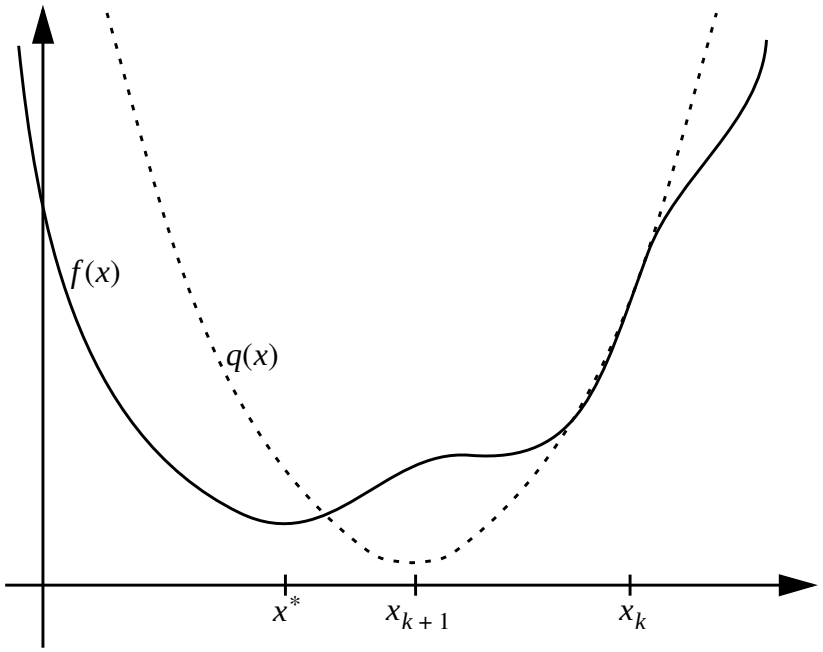
$$x_{k+1} = x_k - \frac{f'(x_k)}{f''(x_k)} \quad . \tag{7.13}$$

Tämä on täsmälleen Newtonin menetelmä [vrt. yhtälö (6.28)] funktion $f'(x)$ nollakohdan etsimiseksi.

Graafisesti Newtonin menetelmän voi hahmottaa seuraavasti. Paraabeli, jolla interpoloidaan funktiota $f(x)$ olkoon $q(x)$. Sille pätee

$$\begin{aligned} q(x_k) &= f(x_k) \\ q'(x_k) &= f'(x_k) \quad . \\ q''(x_k) &= f''(x_k) \end{aligned} \tag{7.14}$$

Piste, joka minimoi $q(x)$:n on seuraava iteraatiopiste x_{k+1}.



Kuva 7.7: Newtonin menetelmä funktion minimin määrittämiseksi.

Edellisestä luvusta muistamme, että Newtonin menetelmän suppenemisen kertaluku on 2. Eli

$$\left|x_{k+1} - x^*\right| \leq C\left|x_k - x^*\right|^2 \ . \tag{7.15}$$

Otetaan esimerkiksi funktio

$$f(x) \ = \ \sin x - \cos x \tag{7.16}$$

ja $x_0 \ = \ -0.5$. Derivaatat ovat

$$f'(x) \ = \ \cos x + \sin x$$
$$f''(x) \ = \ -\sin x + \cos x \ .$$

Tarkka tulos on $x^* \ = \ -\frac{\pi}{4} \approx -0.78539816$.

Ensimmäinen iteraatiokierros:

$$f'(x_0) \ = \ \cos(-0.5) + \sin(-0.5) \ = \ 0.39815702$$
$$f''(x_0) \ = \ -\sin(-0.5) + \cos(-0.5) \ = \ 1.35700810$$
$$p \ = \ -\frac{f'(x_0)}{f''(x_0)} \ = \ -\frac{0.39815702}{1.35700810} \ = \ -0.29340799$$
$$x_1 \ = \ x_0 + p \ = \ -0.5 - 0.29340799 \ = \ -0.79340799$$

Kolmen iteraatiokierroksen tulokset ovat seuraavassa:

Taulukko 7.1: Funktion (7.16) Newtonin iteraatio.

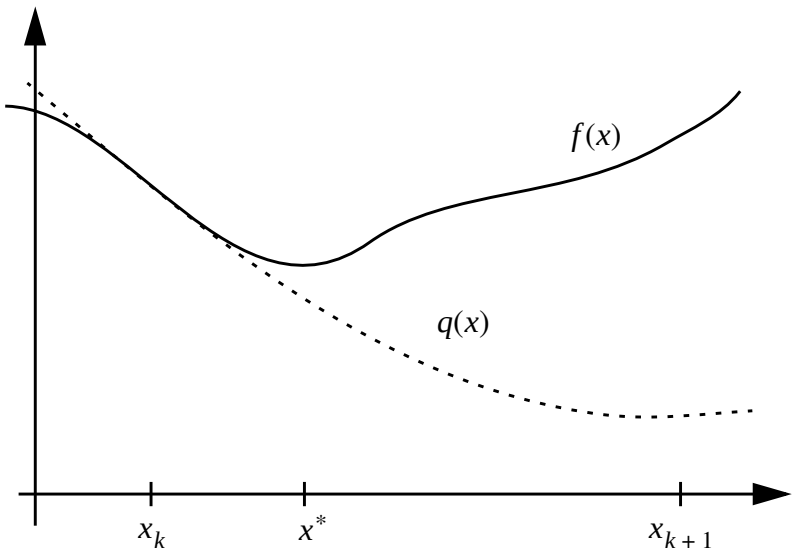
k	x_k	$\lvert x_k - x^* \rvert$
0	–0.50000000	2.8×10^{-1}
1	–0.79340799	8.0×10^{-3}
2	–0.78539799	1.7×10^{-7}
3	–0.78539816	1.3×10^{-17}

Näemme, että suppeneminen on nopeaa.

Jos $f''(x^*) < 0$, on x^* funktion maksimin paikka. Jos taas $f''(x^*) = 0$, on funktion derivaatalla moninkertainen juuri x^*:ssä ja menetelmä suppenee lineaarisesti.

Newtonin menetelmällä on huonot puolensa, joista olemme jotkut jo aikaisemmin todenneet:

i. Jos paraabeli ei olekaan hyvä approksimaatio funktiolle $f(x)$, ei ole mitään takeita siitä, että piste x_{k+1} on lähempänä minimiä kuin x_k. (ks. kuva 7.8).
ii. Newtonin menetelmä voi supeta myös kohti maksimia.
iii. Funktion 1. ja 2. derivaatan tulee olla laskettavissa.



Kuva 7.8: Newtonin menetelmän epäonnistuminen.

Newtonin menetelmä tuleekin yhdistää varmempaan menetelmään (esimerkiksi kultaisen leikkauksen menetelmään) samaan tyyliin kuten epälineaarisen yhtälön ratkaisun yhteydessä.

7.3 Useampimuuttujaisen funktion minimointi

Tehtävänä on minimoida funktio $f(x_1, x_2, \ldots, x_N) \equiv f(\mathbf{x})$ eli etsiä vektori $\mathbf{x}^*$, jolle pätee

$$f(\mathbf{x}^*) \leq f(\mathbf{x}) \quad \forall \mathbf{x} \in \mathbf{R}^N \ . \tag{7.17}$$

Useimpien algoritmien periaate on seuraava:

i. Valitaan alkupiste (tai -pisteet)
ii. Valitaan suunta $\mathbf{d}_k$, josta minimiä ryhdytään etsimään.
iii. Haetaan askelpituus $\lambda_k > 0$ siten, että $f(\mathbf{x}_k + \lambda_k \mathbf{d}_k)$ saa minimiarvon. Tähän ns. viivahakuun käytetään usein yhden muuttujan funktion minimointiin tarkoitettuja algoritmeja.
iv. Asetetaan $\mathbf{x}_{k+1} = \mathbf{x}_k + \lambda_k \mathbf{d}_k$.
v. Jos minimiä ei ole saavutettu, mennään askeleeseen ii.

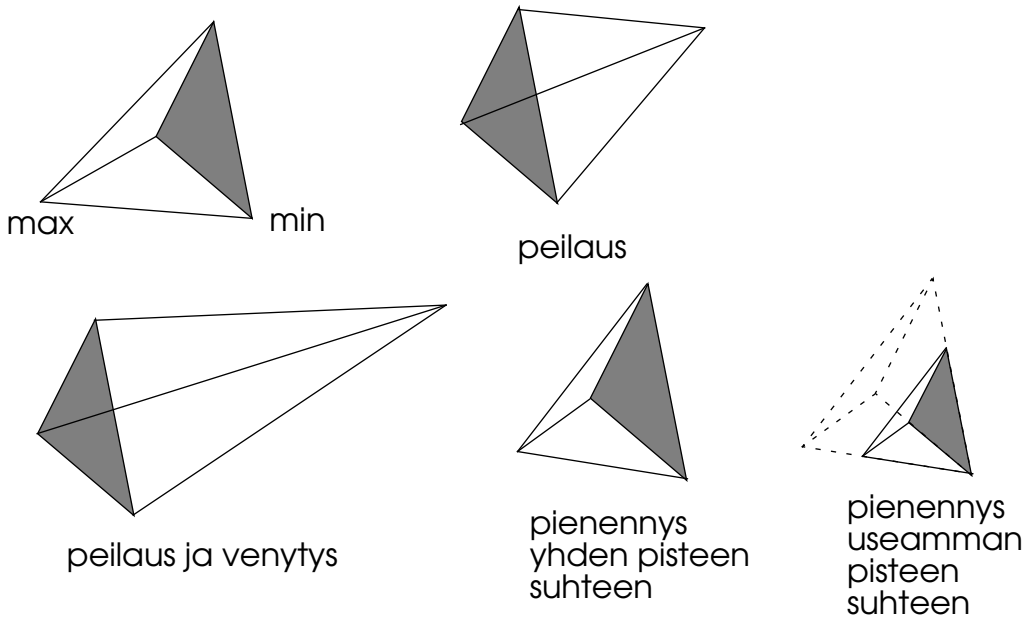
Useimmat algoritmit käyttävät hyväksi funktion Taylorin sarjaa iteraatiopisteen ympäristössä:

$$\begin{aligned} f(\mathbf{x}+\mathbf{h}) &= f(\mathbf{x}) + \sum_{i=1}^{N} \frac{\partial f(\mathbf{x})}{\partial x_i} h_i + \frac{1}{2} \sum_{i=1}^{N} \sum_{j=1}^{N} h_i \frac{\partial^2 f(\mathbf{x})}{\partial x_i \partial x_j} h_j + \ldots \quad , \\ &= f(\mathbf{x}) + [\nabla f(\mathbf{x})]^T \cdot \mathbf{h} + \frac{1}{2} \mathbf{h}^T \mathbf{H}(\mathbf{x}) \mathbf{h} + \ldots \end{aligned} \tag{7.18}$$

missä $\mathbf{H}(\mathbf{x})$ on ns. Hessen matriisi.

7.3.1 Polytooppihaku

Esitellään aluksi lyhyesti algoritmi, joka ei käytä hyväkseen funktion f derivaattoja. Alkuarvona tarvitaan $N+1$ pistettä, jotka muodostavat N-ulotteisen monitahokkaan. Siis jos $N = 2$ on monitahokas kolmio, ja jos $N = 3$ on se tetraedri. Jokaisella iteraatiokierroksella monitahokkaalle tehdään jokin kuvan 7.9 operaatioista (esimerkkinä tapaus $N = 3$).



Kuva 7.9: Polytooppihaun operaatiot.

Useimmat askeleet ovat sen pisteen, jossa funktiolla on suurin arvo, peilaus monitahokkaan vastakkaisen tahon suhteen. Lisäksi yritetään muita kuvan operaatioita. Tällä tavalla lähestytään funktion minimä. Iteraation voi lopettaa esimerkiksi, kun monitahokkaan pisteiden suurimman ja pienimmän funktionarvon erotus on tarpeeksi pieni.

Menetelmä suppenee varsin hitaasti, mutta se varsin luotettava ja etuna tietysti se, että funktion derivaattoja ei tarvita.

Polytooppihaku on toteutettu NR:n rutiinissa `amoeba`.

7.3.2 Jyrkimmän suunnan menetelmä

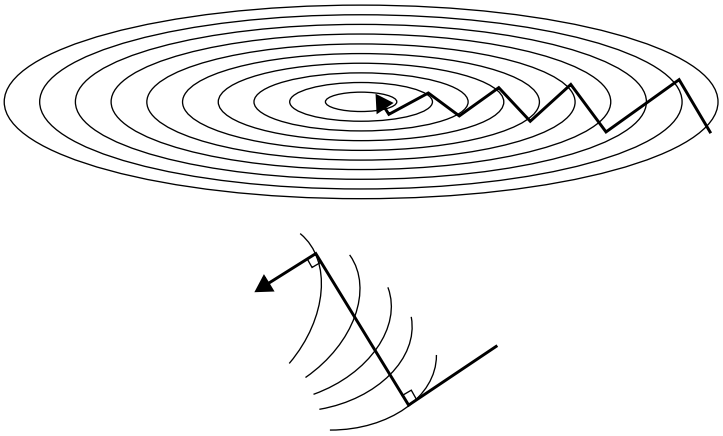
Olkoon meillä aloituspiste $\mathbf{P}$ N-ulotteisessa avaruudessa. Etenemme tästä pisteestä suuntaan $\mathbf{n}$. Yksimuuttujaisten funktioiden minimointialgoritmeja käyttäen voimme etsiä funktion $f(\mathbf{x})$ minimin tässä suunnassa. Useita erilaisia monen muuttujan funktion minimointialgoritmeja voidaan konstruoida valitsemalla erilaisia tapoja määrätä tuo suunta $\mathbf{n}$ iteraation kussakin vaiheessa.

Jyrkimmän suunnan menetelmässä edetään yksinkertaisesti siihen suuntaan, jossa funktio (paikallisesti) vähenee eniten eli $-\nabla f(\mathbf{x}_k)$. Algoritmi on seuraava:

i. Haetaan minimipiste $\mathbf{y}_k = \mathbf{x}_k + \lambda_k(-\nabla f(\mathbf{x}_k))$, siten että $f(\mathbf{y}_k)$ saa minimiarvon askelpituudella $\lambda_k > 0$.

ii. Asetetaan $\mathbf{x}_{k+1} = \mathbf{x}_k + \mathbf{y}_k$. Mennään askeleeseen i, jos ei olla saavutettu haluttua tarkkuutta.

Menetelmä suppenee ainoastaan lineaarisesti, ja sitä ei kovin paljon käytetä, koska kehittyneempiäkin menetelmiä on. Joskus sitä voidaan käyttää muiden menetelmien yhteydessä, esimerkiksi jos ollaan kovin kaukana minimistä ja kehittyneempien menetelmien suppeneminen on hidasta.



Kuva 7.10: Jyrkimmän suunnan menetelmä.

Ylläoleva kuva esittää menetelmän käyttäytymistä kapeassa ‘laaksossa’. Koska joka askeleella minimoidaan funktion gradientin suunnassa, tehdään seuraava askel kohtisuoraan edelliseen nähden.

7.3.3 Newtonin menetelmä

Oletetaan ensin, että funktio f on kvadraattinen eli

$$f(\mathbf{x}) = \frac{1}{2}\mathbf{x}^T \cdot \mathbf{A} \cdot \mathbf{x} + \mathbf{b}^T \cdot \mathbf{x} + \mathbf{c} \ . \tag{7.19}$$

Funktion gradientiksi saadaan

$$\nabla f(\mathbf{x}) = \mathbf{A} \cdot \mathbf{x} + \mathbf{b} \tag{7.20}$$

ja Hessen matriisiksi

$$\mathbf{H}(\mathbf{x}) = \nabla^2 f(\mathbf{x}) = \mathbf{A} \ . \tag{7.21}$$

Funktion gradientti pisteessä $\mathbf{x} + \mathbf{s}$ on

$$\begin{aligned} \nabla f(\mathbf{x}+\mathbf{s}) &= \mathbf{A} \cdot (\mathbf{x}+\mathbf{s}) + \mathbf{b} \\ &= (\mathbf{A} \cdot \mathbf{x} + \mathbf{b}) + \mathbf{A} \cdot \mathbf{s} \\ &= \nabla f(\mathbf{x}) + \mathbf{H}(\mathbf{x}) \cdot \mathbf{s} \end{aligned} \ . \tag{7.22}$$

Yhtälöstä

$$\nabla f(\mathbf{x}+\mathbf{s}) = 0 \tag{7.23}$$

saadaan ehto kvadraattisen funktion minimiin vievälle askeleelle $\mathbf{s}$:

$$\mathbf{H}(\mathbf{x}) \cdot \mathbf{s} = -\nabla f(\mathbf{x}) \ . \tag{7.24}$$

Newtonin iteraatio toimii myös muille kuin kvadraattisille funktioille, sillä niitä voidaan pisteen $\mathbf{x}$ ympäristössä arvioida Taylorin sarjalla

$$f(\mathbf{x}+\mathbf{d}) \approx f(\mathbf{x}) + [\nabla f(\mathbf{x})]^T \cdot \mathbf{d} + \frac{1}{2}\mathbf{d}^T \cdot \mathbf{H}(\mathbf{x}) \cdot \mathbf{d} \ . \tag{7.25}$$

Jokaisella iteraatioaskeleella k tehdään siis seuraavat operaatiot:

i. Ratkaistaan askel $\mathbf{s}_k$ yhtälöstä

$$\mathbf{H}(\mathbf{x}_k) \cdot \mathbf{s}_k = -\nabla f(\mathbf{x}_k) \ . \tag{7.26}$$

ii. Päivitetään iteraatiopistettä: $\mathbf{x}_{k+1} = \mathbf{x}_k + \mathbf{s}_k$.

Otetaan esimerkiksi funktio $f(\mathbf{x}) = \frac{1}{2}x_1^2 + \frac{9}{2}x_2^2$. Funktion Hessen matriisi on

$$\mathbf{H}(\mathbf{x}) = \begin{bmatrix} 1 & 0 \\ 0 & 9 \end{bmatrix}$$

ja gradientti

$$\nabla f(\mathbf{x}) = \begin{bmatrix} x_1 \\ 9x_2 \end{bmatrix} \ .$$

Newtonin menetelmä löytää funktion minimin yhdellä askeleella yhtälöstä (7.24). Olkoon alkuarvaus $\mathbf{x}_1 = (9, 1)$. Saadaan yhtälöryhmä

$$\begin{bmatrix} 1 & 0 \\ 0 & 9 \end{bmatrix} \begin{bmatrix} s_1 \\ s_2 \end{bmatrix} = -\begin{bmatrix} 9 \\ 9 \end{bmatrix} \ ,$$

jonka ratkaisu on $\mathbf{s} = (-9, -1)$, ja $\mathbf{x}_2 = (0, 0)$, joka on funktion minimi.

7.3.4 Kvasi-Newtonin menetelmä

Puhdas Newtonin menetelmä ei välttämättä joka askeleella tuo meitä lähemmäs funktion minimiä.

Jotta lähestyisimme funktion minimiä, tulee $f(\mathbf{x})$:n arvon vähetä iteraatioaskeleella. Tämä pätee, jos

$$\mathbf{s}_k^T \cdot \nabla f(\mathbf{x}_k) < 0 \tag{7.27}$$

eli [ks. yhtälö (7.26)]

$$\mathbf{s}_k^T \cdot \mathbf{H}(\mathbf{x}_k) \cdot \mathbf{s}_k > 0 \quad . \tag{7.28}$$

Tämä tarkoittaa, että Hessen matriisi on ns. positiivisesti definiitti. Ollessamme kaukana minimistä ei ole takeita, että tämä pätee. Siinä tapauksessa Newtonin askel voi viedä kauemmas minimistä.

Kvasi-Newtonin menetelmässä iteraation alussa ei käytetä Hessen matriisia, vaan jotain muuta symmetristä ja positiivisesti definiittä matriisia, jota iteraation kuluessa päivitetään siten, että se lähestyy tarkkaa Hessen matriisia

$$\lim_{i \to \infty} \mathbf{H}_i = \mathbf{H} \ . \tag{7.29}$$

Aluksi voidaan käyttää esimerkiksi yksikkömatriisia $\mathbf{H}_1 = \mathbf{1}$.

Kvasi-Newton-algoritmit eroavatkin tuon matriisin $\mathbf{H}_i$ päivityksen suhteen.

Laskuharjoituksissa tutustumme NR:n kvasi-Newton-rutiiniin `dfpmin`. Seuraavassa vilä hieman asiaa tuosta rutiinista ja sen käyttämästä menetelmästä.

Olessamme kaukana minimistä, Hessen matriisi on positiivisesti definiitti ja se takaa, että funktion arvo pienenee otettavan askeleen suunnassa. Lähellä minimiä Hessen matriisin $\mathbf{H}_i$ päivitys takaa sen, että se lähestyy tarkkaa arvoaan ja saavutamme Newtonin menetelmän kvadraattisen suppenemisen. On tietenkin käytännöllistä käsitellä Hessen matriisin käänteismatriisia $\mathbf{H}^{-1}$, koska $\mathbf{H}$:ta ei missään vaiheessa lasketa osittaisderivaatoista[1].

Miten sitten matriisia $\mathbf{H}_i^{-1}$ päivitetään, että se lähestyy Hessen matriisia? Asiaa voi perustella seuraavasti.

Yhtälöstä (7.24) saamme

$$\mathbf{x}_{i+1} - \mathbf{x}_i = \mathbf{H}^{-1} \cdot [\nabla f(\mathbf{x}_{i+1}) - \nabla f(\mathbf{x}_i)] \ . \tag{7.30}$$

---

1. Muistanet, että $H_{ij}(\mathbf{x}) = \dfrac{\partial^2 f(\mathbf{x})}{\partial x_i \partial x_j}$.

Tässä $\mathbf{H}$ on tarkka Hessen matriisi. On luontevaa, että myös $\mathbf{H}_i$:lle pätee ylläoleva, eli

$$\mathbf{x}_{i+1} - \mathbf{x}_i = \mathbf{H}_{i+1}^{-1} \cdot [\nabla f(\mathbf{x}_{i+1}) - \nabla f(\mathbf{x}_{i+1})] \ . \tag{7.31}$$

Voimme myös ajatella, että päivitys on muotoa $\mathbf{H}_{i+1}^{-1} = \mathbf{H}_i^{-1} + \text{korjaus}$. Materiaali, josta tuo korjaus koostuu on $\mathbf{x}_{i+1} - \mathbf{x}_i$ ja $\nabla f(\mathbf{x}_{i+1}) - \nabla f(\mathbf{x}_{i+1})$.

Ns. Davidon-Fletcher-Powell-algoritmi (DFP) tekee tämän päivityksen seuraavasti

$$\begin{aligned}\mathbf{H}_{i+1}^{-1} = \mathbf{H}_i^{-1} &+ \frac{(\mathbf{x}_{i+1} - \mathbf{x}_i) \bullet (\mathbf{x}_{i+1} - \mathbf{x}_i)}{(\mathbf{x}_{i+1} - \mathbf{x}_i) \cdot [\nabla f(\mathbf{x}_{i+1}) - \nabla f(\mathbf{x}_{i+1})]} \\ &- \frac{\{\mathbf{H}_i^{-1} \cdot [\nabla f(\mathbf{x}_{i+1}) - \nabla f(\mathbf{x}_{i+1})]\} \bullet \{\mathbf{H}_i^{-1} \cdot [\nabla f(\mathbf{x}_{i+1}) - \nabla f(\mathbf{x}_{i+1})]\}}{[\nabla f(\mathbf{x}_{i+1}) - \nabla f(\mathbf{x}_{i+1})] \cdot \mathbf{H}_i^{-1} \cdot [\nabla f(\mathbf{x}_{i+1}) - \nabla f(\mathbf{x}_{i+1})]}\end{aligned} \tag{7.32}$$

Tässä operaatio $\mathbf{u} \bullet \mathbf{v}$ tarkoittaa vektoreista muodostettua matriisia: $(\mathbf{u} \bullet \mathbf{v})_{ij} = u_i v_j$.

Ns. Broyden-Fletcher-Goldfarb-Shanno-algoritmi (BFGS) on lähes samanlainen kun yllämainittu DFP, mutta siinä on lisätermi

$$\begin{aligned}&\ldots + [\nabla f(\mathbf{x}_{i+1}) - \nabla f(\mathbf{x}_{i+1})] \cdot \mathbf{H}_i^{-1} \cdot [\nabla f(\mathbf{x}_{i+1}) - \nabla f(\mathbf{x}_{i+1})] \quad , \\ &\times (\mathbf{u} \bullet \mathbf{u})\end{aligned} \tag{7.33}$$

missä $\mathbf{u}$ on vektori

$$\begin{aligned}\mathbf{u} = &\frac{\mathbf{x}_{i+1} - \mathbf{x}_i}{(\mathbf{x}_{i+1} - \mathbf{x}_i) \cdot [\nabla f(\mathbf{x}_{i+1}) - \nabla f(\mathbf{x}_{i+1})]} \\ &- \frac{\mathbf{H}_i^{-1} \cdot [\nabla f(\mathbf{x}_{i+1}) - \nabla f(\mathbf{x}_{i+1})]}{[\nabla f(\mathbf{x}_{i+1}) - \nabla f(\mathbf{x}_{i+1})] \cdot \mathbf{H}_i^{-1} \cdot [\nabla f(\mathbf{x}_{i+1}) - \nabla f(\mathbf{x}_{i+1})]}\end{aligned} \quad . \tag{7.34}$$

NR:n rutiini `dfpmin` on toteutettu käyttäen BFGS-algoritmia.

Esimerkkinä otamme funktion

$$f(x, y) = 1 - e^{-x^2/4} \sin^2 y \ . \tag{7.35}$$

Funktion osittaisderivaatat ovat

$$\begin{aligned}\frac{\partial f}{\partial x} &= \frac{1}{2} e^{-x^2/4} \sin^2 y \\ \frac{\partial f}{\partial y} &= -2 e^{-x^2/4} \cos y \ \sin y\end{aligned} \quad . \tag{7.36}$$

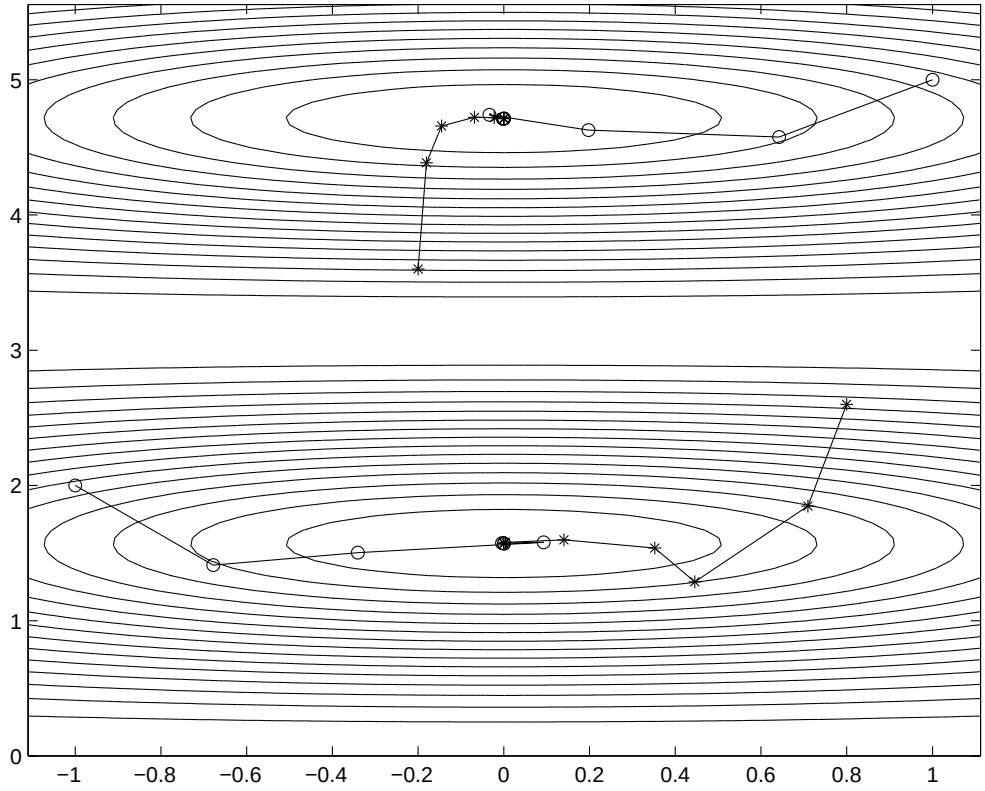
Pääohjelma voisi näyttää seuraavalta:

```
#include <stdio.h>
#include <math.h>
#define NRANSI
#include “nr.h”
#include “nrutil.h”
#define A 4.0
#define NDIM 2
#define GTOL 1.0e-4
static int nfunc,ndfunc;
float func(float x[])
{ nfunc++; return 1.0-exp(-x[1]*x[1]/A)*sin(x[2])*sin(x[2]); }
void dfunc(float x[],float df[])
{
  float t;
  ndfunc++;
  t=exp(-x[1]*x[1]/A);
  df[1]=2.0/A*x[1]*t*sin(x[2])*sin(x[2]);
  df[2]=-2.0*t*cos(x[2])*sin(x[2]);
}

int main(int argc, char **argv)
{
  int iter;
  float *p,fret;
  if (argc!=3) {
    fprintf(stderr,”Usage: %s x0 y0 \n”,argv[0]);
    return (1);
  }
  p=vector(1,NDIM);
  p[1]=atof(*++argv); p[2]=atof(*++argv);
  nfunc=ndfunc=0;
  printf(“Starting vector: (%7.4f,%7.4f)\n”,p[1],p[2]);
  dfpmin(p,NDIM,GTOL,&iter,&fret,func,dfunc);
  printf(“Iterations: %3d\n”,iter);
  printf(“Solution vector: (%9.6f,%9.6f)\n”,p[1],p[2]);
  printf(“Func. value at solution %14.6g\n”,fret);
  return 0;
}
```

Käännös ja ajo (itse rutiinia `dfpmin` on muutettu siten, että se tulostaa iteraatiopisteen iteraation kuluessa):

```
dfpmin> cc -o xdfpmin xdfpmin.c dfpmin.c lnsrch.c nrutil.c -lm
xdfpmin.c:
dfpmin.c:
lnsrch.c:
nrutil.c:
dfpmin> xdfpmin -0.2 3.6
Starting vector: (-0.2000, 3.6000)
ITE: 0 -0.2 3.6 0.806124
ITE: 1 -0.180612 4.38577 0.110225
ITE: 2 -0.145471 4.65777 0.00824103
ITE: 3 -0.0685644 4.72128 0.00125362
ITE: 4 -0.0227184 4.71944 0.000178732
ITE: 5 0.000113919 4.71283 2.02031e-07
ITE: 6 2.49532e-05 4.71241 6.16609e-10
ITE: 7 1.98841e-07 4.71239 1.0103e-14
Iterations:   7
Func. evals:  10
Deriv. evals:   8
Solution vector: ( 0.000000, 4.712389)
Func. value at solution     1.0103e-14
dfpmin>
```

Ohjelma on varsin herkkä alkupisteelle. Alla muutama iteraation kulku graafisesti.



Kuva 7.11: Funktion $f(x, y) = 1 - e^{-x^2/4}\sin^2 y$ minimointi rutiinilla `dfpmin`.

7.3.5 Liittogradienttimenetelmät

Kvasi-Newtonin menetelmässä on laskun aikana pidettävä muistissa Hessen matriisi $\mathbf{H}$. Jos on kovin moniulotteisista optimointitehtävistä kyse, voi tämä olla ongelma.

Tällöin voi liittogradienttimenetelmä olla käyttökelpoinen.

Kvadraattista muotoa (7.19) olevan funktion minimointi voidaan tehdä N:ssä askeleessa siten, että mikään askeleella k suuntaan $\mathbf{s}_k$ tehty minimointi ei ‘pilaa’ aikaisempien suuntien minimointia. Ongelmana on vain näiden N:n ns. liittosuunnan löytäminen. Lisäksi funktiot ovat vain approksimatiivisesti kvadraattisia, joten iteraatiota joutuu tekemään pidempään kuin N kierrosta.

Liittogradienttimenetelmässä generoidaan iteraation kuluessa vektorit $\mathbf{g}_i$ ja $\mathbf{h}_i$. Alussa asetetaan $\mathbf{g}_1 = -\nabla f(\mathbf{x}_1)$ ja $\mathbf{h}_1 = \mathbf{g}_1$, missä $\mathbf{x}_1$ on alkuarvaus minimipisteelle.

Iteraatio sujuu seuraavasti

i. Aseta $i = 1$

ii. Minimoi funktio $f(\mathbf{x})$ suuntaan $\mathbf{h}_i$. Saadaan piste $\mathbf{x}_{i+1}$.

iii. Aseta $\mathbf{g}_{i+1} = -\nabla f(\mathbf{x}_{i+1})$.

iv. Laske[1] $\gamma_i = \dfrac{(\mathbf{g}_{i+1} - \mathbf{g}_i) \cdot \mathbf{g}_{i+1}}{\mathbf{g}_i \cdot \mathbf{g}_i}$.

v. Päivitä suuntaa: $\mathbf{h}_{i+1} = \mathbf{g}_{i+1} + \gamma_i \mathbf{h}_i$.

vi. Jos minimiä ei löydetty, mene askeleeseen ii.

NR:n rutiini `frprmn` minimoi funktion ylläesitetyn alogritmin tavoin.

Algoritmista on olemassa myös versioita, joissa se aloitetaan uudelleen tietyissä tapauksissa, eli resetoidaan minimointisuunnaksi gradientin vastainen suunta.

1. Fletcher-Reeves -menetelmän Polak-Ribiere -variantti.

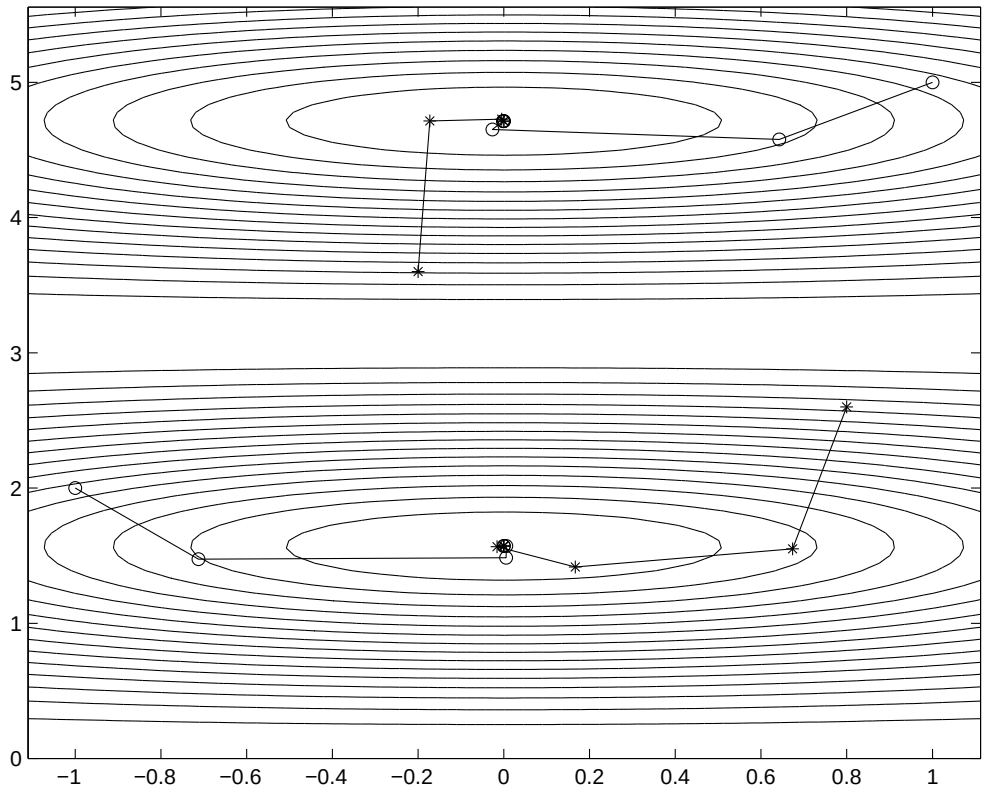
Seuraavassa rutiinia `frprmn` käyttävä pääohjelma, joka minimoi funktion (7.35).

```
#include <stdio.h>
#include <math.h>
#define NRANSI
#include “nr.h”
#include “nrutil.h”
#define A 4.0
#define NDIM 2
#define FTOL 1.0e-6

static int nfunc,ndfunc;

float func(float x[])
{
  nfunc++;
  return 1.0-exp(-x[1]*x[1]/A)*sin(x[2])*sin(x[2]);
}

void dfunc(float x[],float df[])
{
  float t;
  ndfunc++;
  t=exp(-x[1]*x[1]/A);
  df[1]=2.0/A*x[1]*t*sin(x[2])*sin(x[2]);
  df[2]=-2.0*t*cos(x[2])*sin(x[2]);
}

int main(int argc, char **argv)
{
  int iter;
  float fret,*p;

  p=vector(1,NDIM);
  if (argc!=3) {
    fprintf(stderr,”Usage: %s x0 y0 \n”,argv[0]);
    return (1);
  }
  p=vector(1,NDIM);
  p[1]=atof(*++argv);
  p[2]=atof(*++argv);
  printf(“Starting vector: (%7.4f,%7.4f)\n”,p[1],p[2]);
  frprmn(p,NDIM,FTOL,&iter,&fret,func,dfunc);
  printf(“Iterations: %3d\n”,iter);
  printf(“Func. evals: %3d\n”,nfunc);
  printf(“Deriv. evals: %3d\n”,ndfunc);
  printf(“Solution vector: (%9.6f,%9.6f)\n”,p[1],p[2]);
  printf(“Func. value at solution %14.6g\n”,fret);
  free_vector(p,1,NDIM);
  return 0;
}
```

Käännös ja ajo:

```
frprmn> cc -o xfrprmn xfrprmn.c frprmn.c mnbrak.c linmin.c nrutil.c
frprmn> xfrprmn -0.2 3.6
Starting vector: (-0.2000, 3.6000)
ITE: 0 -0.2 3.6 0.806124
ITE: 1 -0.172527 4.71345 0.00741493
ITE: 2 -0.00504783 4.72753 0.000235587
ITE: 3 8.32696e-05 4.7124 1.78478e-09
ITE: 4 1.24055e-08 4.71239 2.22045e-16
Iterations:   5
Func. evals:  68
Deriv. evals:   5
Solution vector: (-0.000000, 4.712389)
Func. value at solution    2.22045e-16
frprmn>
```

Rutiiniin `frprmn` lisättiin iteraation kulusta kertova tulostus. Alla graafisesti iteraation kulku joillain alkuarvoilla (vertaa kuvaan 7.11).



Kuva 7.12: Funktion $f(x, y) = 1 - e^{-x^2/4}\sin^2 y$ minimointi rutiinilla `frprmn`.

7.3.6 Simuloitu jäähdytys

Simuloitu jäähdytys on stokastinen minimointimenetelmä, jota on sovellettu lähinnä kombinatorisiin ongelmiin, eli ongelmiin, joissa etsittävä avaruus ei ole yksinkertaisesti $\mathbf{R}^N$, vaan diskreetti, mutta suuri joukko mahdollisia konfiguraatioita. Menetelmää on mahdollista soveltaa myös jatkuvia muuttujia sisältävään tehtävään.

Kuten jo menetelmän nimi sanoo, se on jossain analoginen aineen hitaalle jäähdyttämiselle. Lämpökäsittelyssä aine ensin sulatetaan ja sen jälkeen hitaasti jäähdytetään. Lämpötilan laskiessa systeemi alkaa hakeutua energiaminimiinsä eli kiteiseen muotoon. Jos jäähdytys tehdään nopeasti, ei systeemi ehdi löytää globaalia energiaminimiä, vaan joutuu metastabiiliin tilaan (esim. lasi).

Menetelmä perustuu Boltzmannin todennäköisyysjakaumaan

$$P(E) \propto e^{-E/T} \ , \tag{7.37}$$

missä P on todennäköisyys, että systeemi on tilassa, jonka energia on E, lämpötilassa T[1]. Systeemi voi korkeassa lämpötilassa olla myös suurenergisissä tiloissa. Lämpötilan laskiessa jakauma alkaa yhä enemmän piikittyä energiaminimin kohdalle.

Simuloidussa jäähdytyksessä minimoitavan funktion $f(\mathbf{x})$ voidaan ajatella vastaavan energiaa. Lisäksi tarvitaan kontrolliparametri c, joka vastaa lämpötilaa. Systeemin tilan kertoo muuttuja $\mathbf{x}$, joka tässä tapauksessa voi olla myös diskreetti.

Periaatteena on generoida tiloja, jotka noudattavat jakaumaa

$$P(f) \propto e^{-f(\mathbf{x})/c} \ . \tag{7.38}$$

Voidaan osoittaa, että lämpötilan c lähestyessä nollaa, lähestyy jakauma (7.38) minimikonfiguraatiota[2]

$$P(f) \propto \delta(\mathbf{x} - \mathbf{x}_{\min}) \ , \tag{7.39}$$

missä $\mathbf{x}_{\min}$ on konfiguraatio, jossa funktiolla f on minimiarvo.

Miten sitten generoimme jakauman (7.39) mukaisia tiloja? Ratkaisu on ns. Metropolis-algoritmi, joka alunperin kehitettiin Monte Carlo -simulaatiomenetelmää varten. Olkoon meillä tila $\mathbf{x}_i$. Lisäämme tähän pienen häiriön $\delta\mathbf{x}$. Tämä uusi tila $\mathbf{x}_j = \mathbf{x}_i + \delta\mathbf{x}$ hyväksytään seuraavanlaisella todennäköisyydellä:

---

1. Boltzmannin vakio k_B on asetettu ykköseksi, eli lämpötilan yksikkö on sama kuin energian.
2. E. Aarts, J. Korst, *Simulated Annealing and Boltzmann Machines*, Wiley Interscience Series in Discrete Mathematics and Optimization, (John Wiley & Sons, 1989, New York).

$$P_{\mathrm{acc}}(i \to j) = \begin{cases} 1, & \delta f_{ij} \le 0 \\ \exp\left(-\dfrac{\delta f_{ij}}{c}\right), & \delta f_{ij} > 0 \end{cases}, \tag{7.40}$$

missä

$$\delta f_{ij} = f(\mathbf{x}_j) - f(\mathbf{x}_i) \ . \tag{7.41}$$

Jos siis menemme funktion f kannalta alamäkeen hyväksymme uuden tilan varmasti. Voimme myös hyväksyä siirtoja, jotka vievät funktion f arvoa ylöspäin. Tähän perustuu menetelmän kyky löytää globaaleja minimejä. Aikaisemmin tässä luvussa esiteltyjä menetelmiä voisi verrata karkaisuun. Kappale jäähdytetään nopeasti, jolloin se hakeutuu lähimpänä olevaan lokaaliin energiaminimiin.

Siitä, että (7.40) generoi jakauman (7.38) mukaisia tiloja voidaan vakuuttua seuraavasti.

Merkitään systeemin tilaa todennäköisyysvektorilla $\mathbf{P}^{(i)}$. Sen alkiot kertovat tietyn tilan todennäköisyyden 'ajanhetkellä' i. Olkoon meillä Markovin prosessi, jonka generoi stokastinen matriisi $\mathbf{Q}$. Alkio Q_{ij} kertoo todennäköisyyden siirtymälle $i \to j$. Stokastiselle matriisille pätee

$$\sum_j Q_{ij} = 1 \ , \quad Q_{ij} > 0 \ . \tag{7.42}$$

Markov-ketjussa päästään eteenpäin operoimalla tilavektoriin $\mathbf{P}^{(i)}$ matriisilla $\mathbf{Q}$:

$$\mathbf{P}^{(i+1)} = \mathbf{P}^{(i)}\mathbf{Q} \ . \tag{7.43}$$

Pitkällä aikavälillä todennäköisyysjakauma saadaan raja-arvosta

$$\mathbf{P} = \lim_{\tau \to \infty} \mathbf{P}^{(1)}\mathbf{Q}^{\tau} \ . \tag{7.44}$$

Tämän tasapainojakauman $\mathbf{P}$ tulee toteuttaa ominaisarvoyhtälön

$$\mathbf{P} = \mathbf{P}\mathbf{Q} \ , \tag{7.45}$$

eli todennäköisyysvektori ei muutu; tilanne on stationäärinen. Auki kirjoitettuna tämä on

$$\sum_m P_m Q_{mn} = P_n \ . \tag{7.46}$$

Rajoittavampi kuin ehto (7.46) on ns. detaljibalanssi

$$P_i Q_{ij} = P_j Q_{ji} \ . \tag{7.47}$$

On helppo osoittaa, että (7.47):stä seuraa myös (7.46). Lisäksi täytyy olla mahdollista päästä mistä tahansa tilasta i mihin tahansa toiseen tilaan j. Eli prosessin on oltava ergodinen.

Voidaan osoittaa, että (7.40) toteuttaa nämä ehdot. On vain muistettava, että (7.40) on vain hyväksymistodennäköisyys, se on lisäksi kerrottava yritesiirtojen todennäköisyydellä.

Itse algoritmi on seuraavanlainen

i. Aseta kontrolliparametri (lämpötila) $c = c_0$.

ii. Aseta $i = 1$. Aseta systeemi alkutilaan $\mathbf{x}_i = \mathbf{x}_0$.

iii. Muuta systeemin tilaa: $\mathbf{x}_i \rightarrow \mathbf{x}_i + \delta\mathbf{x}$

iv. Laske funktion f muutos: $\delta f = f(\mathbf{x}_i + \delta\mathbf{x}) - f(\mathbf{x}_i)$.

v. Generoi tasaisesti välille $[0, 1)$ jakautunut satunnaisluku ξ.

vi. Jos $\xi < \exp(-\delta f / c)$, hyväksytään uusi tila: $\mathbf{x}_{i+1} = \mathbf{x}_i + \delta\mathbf{x}$.

vii. Aseta $i = i + 1$. Jos $i \leq i_{\max}$, mene askeleeseen iii. Muuten mene seuraavaan askeleeseen.

viii. Laske lämpötilaa: esimerkiksi $c = c\alpha$, $0 < \alpha < 1$.

ix. Jos $c < c_{\min}$, lopeta.

x. Aseta $i = 1$ ja mene askeleeseen iii.

Otetaan esimerkiksi aikaisemminkin käytetty funktio

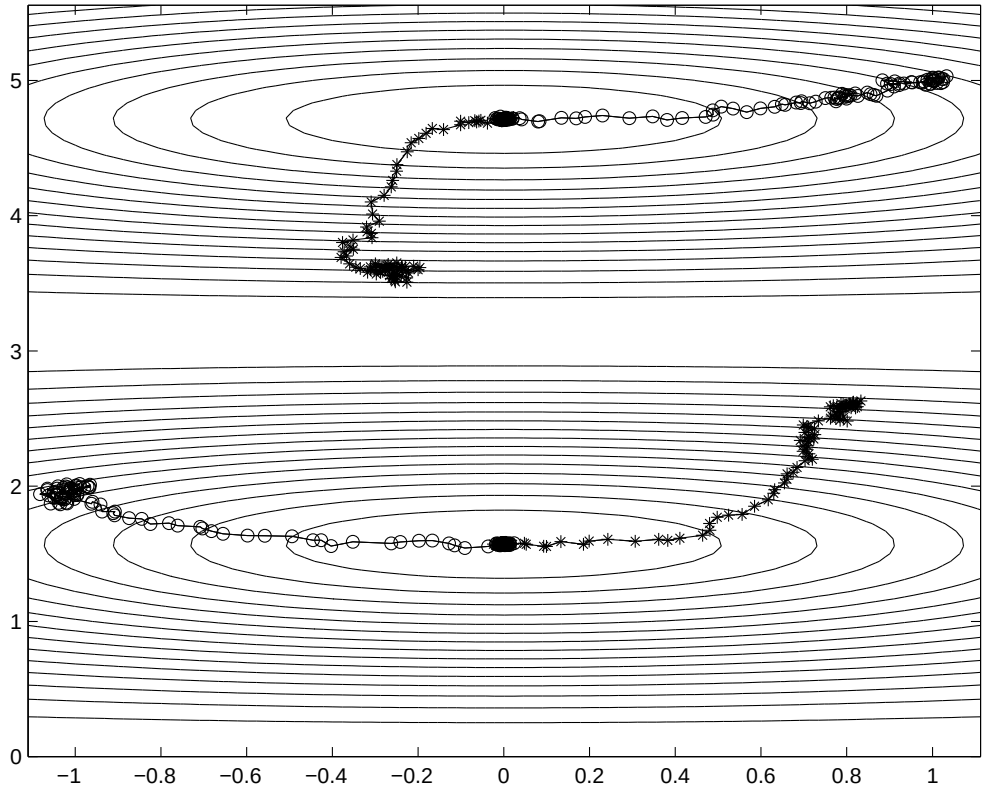
$$f(x, y) = 1 - e^{-x^2/4} \sin^2 y \quad . \tag{7.48}$$

C-ohjelma, joka etsii minimin on alla[1]

```
#include <stdio.h>
#include <math.h>
#define NRANSI
#include “nr.h”
#include “nrutil.h”
#define NDIM 2
#define MAXSTEP 1000
#define A 4.0
static int nfunc=0;
int metropx(float de, float t, long *seed);
float func(float x[])
{
  nfunc++;
  return 1.0-exp(-x[1]*x[1]/A)*sin(x[2])*sin(x[2]);
}

int main(int argc, char **argv)
{
  int niter,iter,i,j,ans;
  float fret,*x,Tfrac,Tini,Tmin,T,dx,ddx,f1,f2,df;
  long seed;
  x=vector(1,NDIM);
  if (argc!=9) {
    fprintf(stderr,”Usage: %s x0 y0 niter Tfrac Tini Tmin seed dx\n”,
            argv[0]); return (1);
  }
  x=vector(1,NDIM);
  x[1]=atof(*++argv); x[2]=atof(*++argv);
  niter=atoi(*++argv); Tfrac=atof(*++argv);
  Tini=atof(*++argv); Tmin=atof(*++argv);
  seed=atoi(*++argv); dx=atof(*++argv);
  T=Tini;
  printf(“i: %d %g %g %g %g\n”,0,T,x[1],x[2],func(x));
  for (i=1;i<=MAXSTEP;i++) {
    for (iter=1;iter<=niter;iter++) {
      for (j=1;j<=NDIM;j++) {
        f1=func(x);
        ddx=dx*(2.0*ran3(&seed)-1.0);
        x[j]+=ddx;
        f2=func(x);
        df=f2-f1;
        ans=metropx(df,T,&seed);
        if (!ans) x[j]-=ddx;
        }
    }
    T*=Tfrac;
    if (T<Tmin) {
      printf(“Iteration ended at step %d: T = %g\n”,i,T);
      printf(“Minimun = %g %g\nF(xmin) = %g\n”,x[1],x[2],func(x));
      printf(“Function evaluations =  %d\n”,nfunc);
      return;
    }
  }
}
```

1. Löytyy myös URL:stä *http://www.lce.hut.fi/teaching/S-114.100/sa/* .

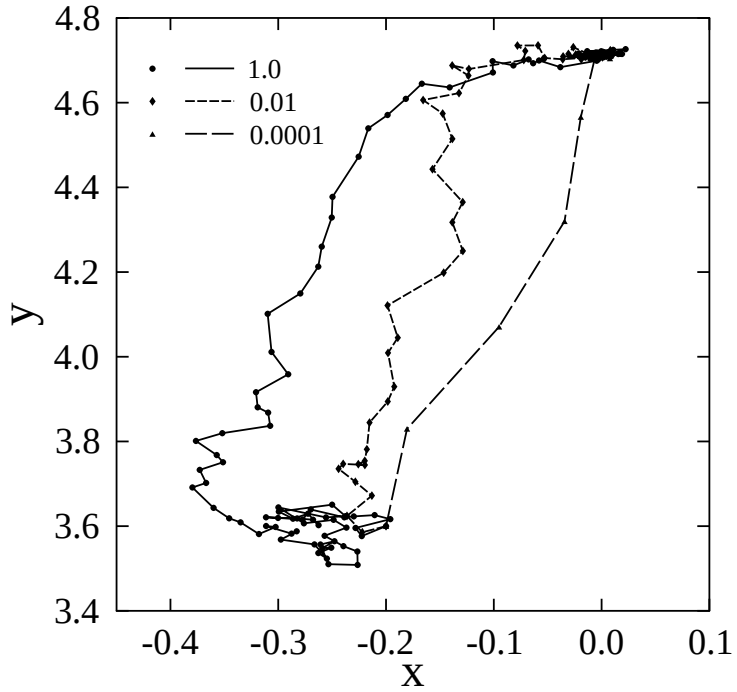
Alla kuva iteraation etenemisestä. Alkuarvoina oli `niter=1000`, `Tini=1.0`, `Tfrac=0.95`, ja `Tmin=0.00001`.



Kuva 7.13: Funktion $f(x, y) = 1 - e^{-x^2/4} \sin^2 y$ minimin etsintä simuloidulla jäähdytyksellä.

Kuvasta nähdään, että iteraatio suppenee varsin hitaasti. Parametreja viilaamalla päästään kuitenkin parempiin tuloksiin. Esimerkikisi laskemalla alkulämpötilaa iteraatio suppenee jonkin verran nopeammin.

.



Kuva 7.14: Kuvan 7.13 iteraation kulku alkulämpötilan arvoilla 1.0, 0.01 ja 0.0001.

Tämä oli ainoastaan havainnollistava esimerkki. Kyseisen funktion minimin etsintä simuloidulla jäähdytyksellä ei ole ollenkaan tehokasta.

Parhaiten menetelmä sopii tapauksiin, joissa on lokaaleja minimejä, joihin muut menetelmät voivat päätyä ja tapauksiin, joissa on kyseessä diskreettejä muuttujia (=ratkaisuavaruus on numeroituva).

Menetelmän etuna on myös se, että erilaisia rajoituksia on helppo lisätä algoritmiin.

NR:n rutiinien joukossa on kauppamatkustajan ongelman ratkaiseva simuloituun jäähdytykseen perustuva ohjelma. Siihen tutustutaan laskuharjoituksissa.

8 Satunnaislukujen tuottaminen

8.1 Tasaisesti jakautuneet satunnaisluvut

Satunnaisluvut voidaan periaatteessa jakaa kolmeen luokkaan

i. *Aidot satunnaisluvut* tuotetaan jonkin fysikaalisen prosessin pohjalta (esim. radioaktiivinen hajoaminen tai sähköinen kohina). Tämä on kuitenkin melko epäkäytännöllinen tapa.

ii. Pseudosatunnaisluvut tuotetaan deterministisen algoritmin avulla tietokoneella eli ne eivät ole aidosti satunnaisia, mutta voivat approksimoida aitoja satunnaislukuja hyvin.

iii. *Valesatunnaisluvut* tuotetaan myös algoritmin avulla. Niillä ei kuitenkaan pyritä aitoon satunnaisuuteen vaan ne on yleensä räätälöity tiettyyn sovellutukseen. Esimerkiksi Monte Carlo -integroinnissa voidaan käyttää valesatunnaislukuja peittämään tietty avaruuden osa yhä tiheämmällä ja tiheämmällä pistejoukolla. Näin voidaan saada tuloksen statistinen virhe pienemmäksi. Näistä on asiaa mm. NR:n kappaleessa 7.7.

Vattulainen ym. TFT:llä (nykyisin HIP) ovat tutkineet satunnaislukugeneraattoreiden ominaisuuksia. Hyviä referenssejä löytyy heidän WWW-sivulta

http://www.physics.helsinki.fi/~vattulai/rngs.html .

Lisäksi voisi mainita seuraavat julkaisut:

I. Vattulainen, ym. *CSC Research Reports,* **R05/92.**
T. E. Hull, A. R. Dobell, *SIAM Review* **4** (1962) 230.
S. L. Anderson, *SIAM Review* **32** (1990) 221.
P. L'Ecuyer, *Ann. Oper. Res.* **53** (1994) 77.
A. Compagner, *Am. J. Phys.* **59** (1991) 700.
F. James, *Comput. Phys. Commun.* **60** (1990) 329.

Julkaistuja satunnaislukutaulukoita käytettiin aikaisemmin satunnaislukujen lähteenä. Niihin luvut oli saatu esim. fysikaalisista prosesseista, puhelin- ja väestöluetteloista jne.

Käytännössä kaikki tieteellisessä laskennallisessa käytettävät satunnaisluvut ovat pseudosatunnaislukuja eli ne tuotetaan satunnaislukugeneraattorilla tietokoneohjelmassa.

Hyvältä satunnaislukugeneraattorilta vaaditaan mm. seuraavia ominaisuuksia

i. Luvuilla tulee olla hyvä jakauma eli niiden tulee olla lähellä satunnaista. Simulaatioissa tärkeä ominaisuus on etteivät satunnaislukujonon luvut eivät saa olla millään tavalla korreloituneita. Saman tulee päteä myös n :n peräkkäisen luvun jonoille. Numeerisessa integroinnissa taas tärkein ominaisuus on jakauman tasaisuus.

ii. Lukujonolla tulee olla pitkä periodi. Generaattoreiden toteutustavasta johtuen ne alka-

vat tietyn jakson jälkeen toistamaan itseään.

iii. Lukujonojen tulisi olla toistettavia. Usein halutaan tutkia tulosten riippuvuutta tietyistä parametreista, jolloin halutaan ajaa useita ajoja täsmälleen samalla lukujonolla. Joissain tapauksissa halutaan aloittaa simulaatioajo tietystä kohdasta, jolloin generaattorin tila on tallennettava. Tila voi koostua yhdestä tai useammasta kokonaisluvusta.

iv. Generaattorin tulisi olla helposti siirrettävissä koneympäristöstä toiseen. Algoritmi tulee siten laatia jollain korkean tason kielellä.

v. Generaattorin tulee olla luonnollisesti riittävän nopea. Tämän ominaisuuden merkitys on ehkä tietokoneiden ja simulointialgoritmien kehittyessä hieman vähentynyt. (Yhtä satunnaislukua kohti tehdään yhä enemmän asioita.)

vi. Algoritmin tulisi olla rinnakkaistettavissa.

Satunnaislukugeneraattorit toteutetaan useimmiten kokonaislukuaritmetiikalla, ja haluttu välille $[0, 1)$ sjoittuva reaaliluku saadaan skaalaamalla. Seuraavassa esitellään lyhyesti yleisimmät algoritmit.

8.1.1 Lineaarinen kongruentiaaligeneraattori

Tämä on vanhin generaattorityyppi. Se on muotoa

$$x_{i+1} = (ax_i + c) \bmod m \quad , \tag{8.1}$$

missä luvut x_i muodostavat halutun satunnaislukujonon. Oleelliset parametrit ovat a, c ja m ja siitä käytetään merkintää $\mathrm{LCG}(a, c, m)$.

Parametri m on yleensä tietokoneen suurin mahdollinen kokonaisluku tai sitä hiukan pienempi ja se määrää generaattorin jakson pituuden P. Sille pätee $P \leq m$. Esimerkiksi 32-bittisellä koneella yleinen valinta on $2^{31} - 1$ tai 2^{32}. Tulos saadaan välille $[0, 1)$ skaalaamalla: x_i / m .

Vakio c voidaan myös asettaa nollaksi, jolloin käytetään merkintää $\mathrm{MLCG}(a, m)$.

Lukujonon aloittamiseen tarvitaan siemenluku x_0 . Se voidaan antaa ohjelmalle syöttötietona tai sen voi laskea esimerkiksi päiväyksen ja kellonajan perusteella. Esimerkiksi seuraavasti

$$x_0 = y + 100(m - 1 + 12(d - 1 + 31(h + 24(n + 60s)))) \quad , \tag{8.2}$$

missä y on vuosi (kahdella numerolla), m kuukausi, d päivä, h tunnnit, n minuutit ja s sekunnit. Toinen vaihtoehto, jolla saadaan suurempi vaihtelu x_0 :n vähiten merkitseviin bitteihin on

$$x_0 = s + 60(n + 60(h + 24(d - 1 + 31(m - 1 + 12y)))) \quad . \tag{8.3}$$

Fortran90-koodina nämä voisivat näyttää seuraavilta:

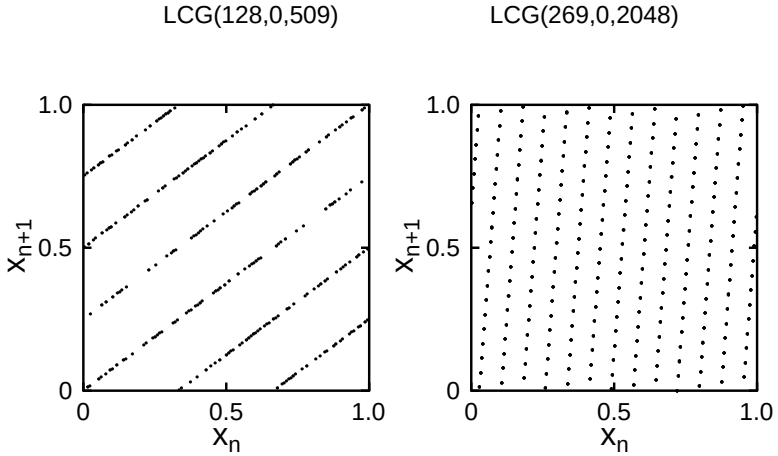
```
program seed

 implicit none
 integer (kind=8) i,j,k,iseed1,iseed2
 integer (kind=8) iy,im,id,iho,imi,ise
 character(len=10) deit,taim

 call date_and_time(deit,taim)
 read(deit(1:4),*) iy;  read(deit(5:6),*) im;
 read(deit(7:8),*) id
 read(taim(1:2),*) iho; read(taim(3:4),*) imi;
 read(taim(5:6),*) ise
 iy=iy-1900
 iseed1=iy+100*(im-1+12*(id1+31*(iho+24*(imi+60*ise))))
 iseed2=ise+60*(imi+60*(iho+24*(id-1+31*(im-1+12*iy))))

 iseed1=ior(iseed1,1) ! Pariton siemen takaa
 iseed2=ior(iseed2,1) ! mahdollisimman pitkän jakson

 write(6,'(//a,i12)') 'seed1: ',iseed1
 write(6,'(a,i12//)') 'seed2: ',iseed2

stop
end program seed
```

Ja itse generaattori [esimerkkinä LCG($69069, 1, 2^{32}$)]:

```
real function lcg(iseed)
 implicit none
 integer (kind=8) iseed
 integer (kind=8), parameter :: a=69069,c=1,m=2**32
 real, parameter :: rm=real(m)

 iseed=mod(iseed*a+c,m)
 lcg=real(iseed)/rm

end function lcg
```

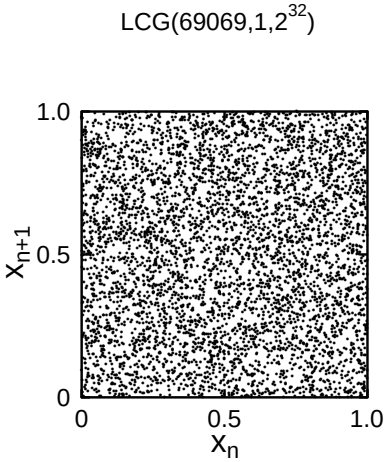
LCG-generaattoreiden ominaisuudet tunnetaan kohtalaisen hyvin.

Niiden ehkä suurin heikkous on se, että satunnaislukujen d-kertojen $\{x_{i+1}, x_{i+2}, \ldots, x_{i+d}\}$ muodostamat pisteet asettuvat d-ulotteisessa avaruudessa lukuisiin samansuuntaisiin hypertasoihin, joiden keskimääräinen etäisyys toisistaan on vakio. Tasojen lukumäärän teoreettinen maksimi d-ulotteisessa hyperkuutiossa $[0, 1)^d$ on $[d!m]^{1/d}$. Jos lukumäärä on tätä oleellisesti pienempi, on pisteiden jakauman tasaisuus heikko. Tavoitteena on etsiä parametrit (a, m), joiden avulla maksimoidaan hypertasojen lukumäärä.
Alla esimerkkinä tapaus $d = 2$ kahdella eri parametrijoukolla.



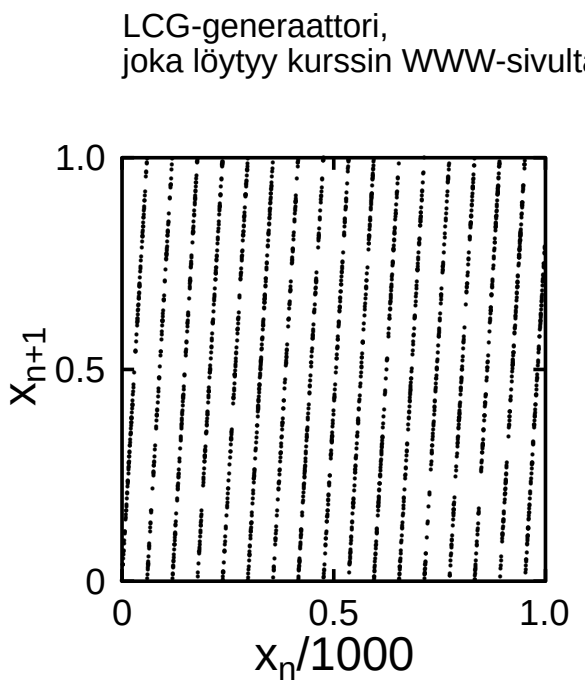
Kuva 8.1: Satunnaislukugeneraattorit LCG(128,0,509) ja LCG(269,0,2048)

Esimerkki käytännön satunnaislukugeneraattorista on allaoleva LCG(69069, 1, 2^{32})



Kuva 8.2: Satunnaislukugeneraattori LCG(69069,1,2^{32}).

Tasot löytyvät hyvänlaatuisestakin generaattorista, kunhan zoomataan tarpeeksi.



Kuva 8.3: Genraattorin LCG(69069,1,2^{32}) korrelaatiot.

Generaattorin lukujonon laatu riippuu siis parametreista. Niiden valinta on melko hankalaa, joten parasta lienee tukeutua generaatoreihin, joita löytyy kirjallisuudesta.

Esimerkkejä käytetyistä LCG-generaattoreista[1] ovat GGL, joka on MLCG(16807, $2^{31} - 1$) ja jota käytetään esimerkiksi IMSL-kirjastossa ja MATLABin versiossa 3.5g,

RAND (LCG(69069, 1, $2^{31} - 1$) , joka löytyi mm. CSC edesmenneeltä Convexilta,

G05FAF, (MLCG(13^{13}, 2^{59})), joka on yksi NAG-kirjaston (versio Mark 14) satunnaislukugeneraattoreista.

8.1.2 Viiveistetyt Fibonacci-generaattorit

Näiden generaattoreiden toiminta perustuu Fibonacci-lukujonon $x_i = x_{i-1} + x_{i-2}$ yleistykseen

$$x_i = (x_{i-q} \otimes x_{i-r}) \bmod m \quad , \tag{8.4}$$

missä parametrit q ja r ovat viiveitä ($q > r$) ja $\otimes$ on jokin binäärioperaatio seuraavista vaihtoehdoista

1. Viitteett löydät julkaisusta I. Vattulainen ym., *CSC Research Reports* **R05/92**.

$+$	yhteenlasku
$-$	erotus
$\times$	tulo
$\oplus$	XOR (exclusive OR), jos m on kahden potenssi.

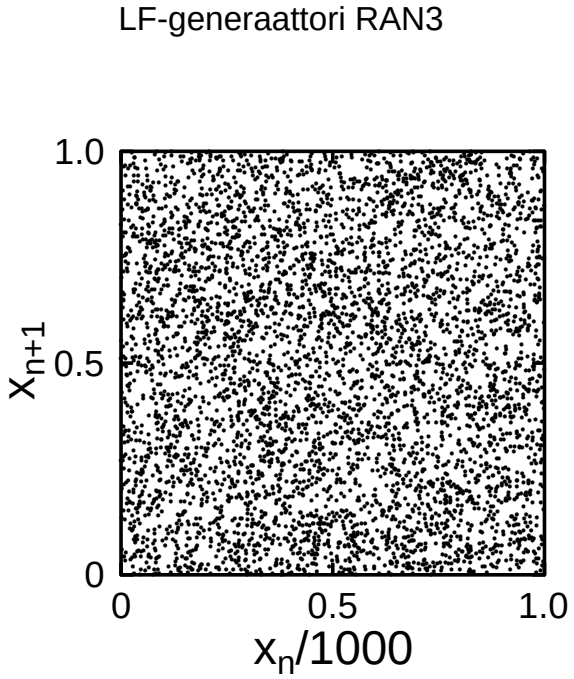
Generaattorista käytetään merkintää LF(q, r, $\otimes$) . Sen alustaminen vaatii q kappaletta siemenlukuja, jotka voidaan esimerkiksi tuottaa jollain toisella generaattorilla.

LF-generaattoreiden käyttö on melko uutta, joten niiden teoreettisista ominaisuuksista ei kovin paljon tiedetä. Niiden jakso pystytään laskemaan ja niiden etuna onkin pitkä jakso. Tietyillä ehdoilla[1] jakson pituudeksi tulee peräti $(2^p - 1)(2^m - 1)$.

Kirjallisuudessa esiintyy suosituksia, että operaatiota $\oplus$ ei kannata käyttää, koska siinä ei tapahdu tarpeeksi lukujen bittien sekoittumista.

Esimerkki LF-generaattorista on kirjan *Numerical Recipes* generaattori `RAN3`, joka on muotoa LF(55, 24, $-$) .

Samantyyppisiä korrelaatioita kuin LCG-generaattoreilla ei esimerkiksi `RAN3`:lla esiinny.



Kuva 8.4: Satunnaislukugeneraattori `ran3`.

Eräs variaatio LF-generaattorista `RCARRY`, joka perustuu ns. ‘add-and-carry’ -operaatioon. Sen algoritmi muistuttaa generaattoria LF(24, 10, $-$) , mutta siihen lisätään carry-bitti, jos hetkellinen Fibonacci-summa on suurempi kuin m . Eli yleisessä muodossa se on

1. D. E. Knuth, *Semi-Numerical Algorithms, vol. 2: The Art of Computer Programming*, 2nd ed. (Addison-Wesley, Redding, MA, 1981).

$$x_i = (x_{i-q} \pm x_{i-r} \pm g) \bmod 2 \quad , \tag{8.5}$$

missä $q > r$ ovat viiveitä ja g toimii carry-bittinä. Jos esimerkiksi $x_{i-q} + x_{i-r} < m$, on $g = 0$. Muuten se on yksi vähiten merkitsevänä bittinä. RCARRYssä $m = 2^{24}$. Käytännön implementointi on (reaaliluvuille)

$$\begin{aligned} x_i &= x_{i-q} - x_{i-r} + g_1 - g_2 \\ g_1 &= 1.0, \text{jos } x_{i-q} - x_{i-r} - g_2 < 0, \text{muuten } g_1 = 0 \\ g_2 &= 1/m, \text{jos } x_{i-q} - x_{i-r} - g_2 < 0, \text{muuten } g_1 = 0 \end{aligned} \quad . \tag{8.6}$$

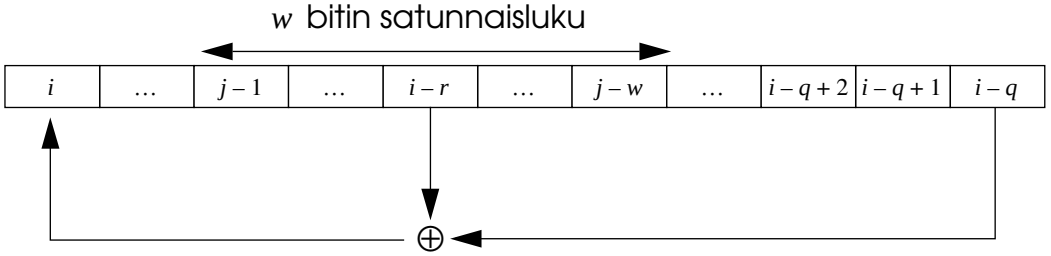
8.1.3 Siirtorekisterit

Kolmas yleinen tapa on **siirtorekisterit**. Ne voi hahmottaa LF-generaattorin erikoistapauksena $m = 2$. Ne perustuvat bittijonoon $b = \{b_i\}$, jonka termin määritellään rekursiivisesti:

$$b_i = (c_1 b_{i-1} + c_2 b_{i-2} + \ldots + c_w b_{i-w}) \bmod 2 \quad . \tag{8.7}$$

Kertoimet c_j $(j = 1, \ldots, w-1)$ ovat joko nollia tai ykkösiä ja $c_w = 1$, jolloin binäärijonon w-kertojen $\{b_{i-1}, b_{i-2}, \ldots, b_{i-w}\}$ periodi on maksimissaan $2^w - 1$. Satunnaiset kokonaisluvut saadaan näistä w-kerroista.

Käytännössä jonon $\{b_i\}$ kertoimet c_j valitaan yhtälön (8.4) mukaisesti. Kertoimet c_q ja c_r ovat siten ykkösiä ja muut nollia. Valitsemalla binäärioperaatioksi $\oplus$ ja asettamalla $m = 2$, jonon muodostuminen on kuvan mukainen.

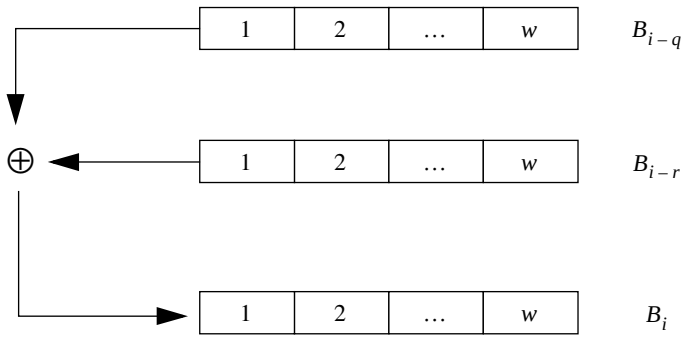


Kuva 8.5: Siirtorekisteri.

Tämä generaattorin ongelma on periodin lyhyys, joten siitä on kehitetty yleistetty versio: yleistetty takaisinkytketty siirtorekisteri (GFSR). Idean on käsitellä kokonaislukuja vastaavia bittijonoja $B_i = \{b_{j+1}, b_{j+2}, \ldots, b_{j+w}\}$ ja suorittaa binäärioperaatio $\oplus$ näiden jonojen välillä yhtälöä (8.7) vastaavalla tavalla:

$$B_i = (c_1 B_{i-1} + c_2 B_{i-2} + \ldots + c_q B_{q-1}) \bmod 2 \quad . \tag{8.8}$$

Kertoimet valitaan yleensä yhtälön (8.4) mukaan. Eli kuvana



Kuva 8.6: Takaisinkytketty siirtorekisteri.

GFSR-generaattorin etuna on periodin pituus, jonka määrää takaisinkytkennän pituus q. Generaattoreista on kuitenkin löytynyt korrelaatioita, jotka hyvin vaativissa simulaatioissa voivat vaikuttaa tuloksiin[1].

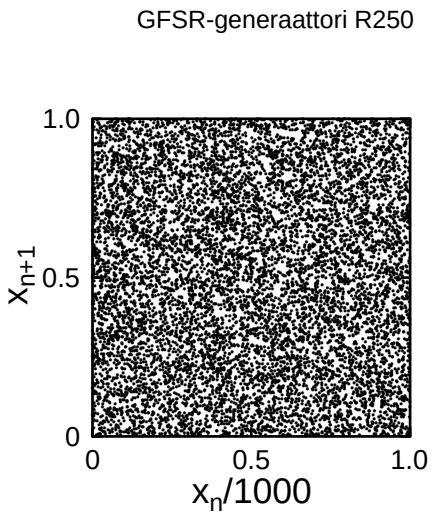
1. I. Vattulainen ym., *Phys. Rev. Lett.* **73** (1994) 2513.

Esimerkkinä GFSR-generaattorista olkoon `R250`:

```
!-----------------------------------------------------
! r250          random number generator
! A past shift-register sequence random number generator
! with maximum cycle lenght 2**250. The algoritrm is based
! on the formula represented by S.Kirkpatrick and E.Stoll
! (see Journal of Computational Physics 40, 527 (1981),
! pages 517-526).
!-----------------------------------------------------
!
!      parameters:
!      x    integer input-output array of dimension n + 250.
!           before first call to the subroutine the elements
!           1-250 of the array must be filled with seeds.
!      r -  floating point output array of dimension n. r contains
!           uniformy distributed random numbers between [0,1[.
! n-  integer input scalar, the dimension of r.
!
!        example call:        integer    a(10250)
!                             real       b(10000)
!                             .
!                             .
!                             do i=1,250
!                                a(i)=int(2147483647.0*ran(is))
!                             enddo
!                             call R250  (A, B, 10000)
!-----------------------------------------------------
```

```
subroutine r250 (x,r,n)
 implicit none

 integer,parameter :: q=103,p=250
 real (kind=8),parameter :: rmaxin=2147483648.d0
 integer n,i,k
 integer x(n+250)
 real r(n)

 do k=1,n
     x(k+p)=ieor(x(k+p-q),x(k))
     r(k)=dfloat(x(k+p))/rmaxin
 enddo

 do i=1,p
     x(i)=x(n+i)
 enddo

 return
end subroutine r250
```

`R250`:ssä ei ole yhtä pahoja 2-kertakorrelaatioita kuin LCG-generaattoreissa.



Kuva 8.7: Satunnaislukugeneraattori `R250`.

Eri generaattoreita ja menetelmiä voidaan tietysti myös yhdistellä satunnaisuuden parantamiseksi.

8.1.4 Sekoitusmenetelmä

Yksinkertaisessa sekoitusmenetelmässä käytetään kahta satunnaislukugeneraattoria, jotka tuottavat tasaisesti välille $[0, 1)$ jakautuneet lukujonot $\{x'_i\}$ ja $\{y'_i\}$. Menetelmässä poimitaan lukujonon $\{y'_i\}$ avulla näytteitä jonosta $\{x'_i\}$.

Kokeellisesti on havaittu, että yhdistämällä kahden tai useamman satunnaislukugeneraattorin tulokset voidaan saada lukujono, jonka satunnaisuus voi olla sen osia parempi.

Olkoot kombinaation osina lukujonot $\{x_i\}$ ja $\{y_i\}$,. Näiden maksimiarvot ovat X ja Y ja $m = \max(X, Y)$. Näiden kombinaatio on

$$z_i = (x_i \otimes y_i) \bmod m \quad , \tag{8.9}$$

jossa $\otimes$ on jokin operaatioista $(+, -, \times, \oplus)$. Luvut z_i normalisoidaan välille $[0, 1) : z_i/m$.

Jos lukujonot $\{x_i\}$ ja $\{y_i\}$ on jo normitettu välille $[0, 1)$, voidaan kombinaatio muodostaa esimerkiksi seuraavasti

$$z_i = \begin{cases} x_i + y_i - |x_i + y_i|, \text{ jos operaatio on } + \\ |x_i - y_i|, \text{ jos operaatio on } - \\ x_i \times y_i, \text{ jos operaatio on } \times \end{cases} . \tag{8.10}$$

Kombinaatiogeneraattorien maksimiperiodi on varsin pitkä.

Esimerkkinä kombinaatiogeneraattorista on RANMAR. Ensimmäinen komponentti on LF(97, 33, •) , missä operaatio • kahden reaalisen satunnaisluvun välillä on

$$x_i \bullet x_j = \begin{cases} x_i - x_j, \text{jos } x_i \geq x_j \\ x_i - x_j + 1, \text{jos } x_i < x_j \end{cases} \tag{8.11}$$

Toisena generaattorina on yksinkertainen aritmeettinen jono moduluksen ollessa alkuluku $m = 2^{24} - 3 = 16777213$ .24-bittiselle realliluvuille tämä voidaan toteuttaa binäärioperaatiolla $y_i \cdot g$:

$$y_i \cdot g = \begin{cases} y_i - g, \text{jos } y_i \geq g \\ y_i - g + 16777213 / 16777216, \text{jos } y_i < g \end{cases} , \tag{8.12}$$

missä

$$g = 7654321 / 16777216 \quad . \tag{8.13}$$

Kombinaation tulos saadaan näiden kahden operaation avulla:

$$\begin{aligned} x_i &= x_{i-97} \bullet x_{i-33} \\ y_i &= y_{i-1} \cdot g \qquad . \\ z_i &= x_i \bullet y_i \end{aligned} \tag{8.14}$$

RANMARin jakso on pitkä $P \approx 2^{144} = 2 \times 10^{43}$, ja sillä voidaan toteuttaa lukuisa määrä erillisiä osajonoja. Viimeksi mainittu piirre on tärkeä simulointiohjelmia rinnakkaistettaessa. Lisäksi generaattori on täysin siirrettävissä koneesta toiseen.

Toisaalta generaattorin tilan tallettamiseen tarvitaan 103 luvun talletus.

8.1.5 Generaattorien testaus

Generaattoreiden testaukseen ei ole olemassa kovin hyviä teoreettisia keinoja, joten testaus on tehtävä ‘empiirisesti’ käyttäen erilaisia mm. statistisia menetelmiä.

i. Tilastollisia testejä ovat mm.

a) χ^2 -testi, Kolmogorov-Smirnov-testi
b) run-testi, jossa tutkitaan monotonisesti kasvavien lukujonojen pituuksia;
c) spektraalitesti, jossa tutkitaan d kertojen sjoittuminen d -ulotteiseen avaruuteen jne.

ii. Lisäksi voidaan tutkia lukujonojen satunnaisuutta bittitasolla, jos generaattori on toteutettu kokonaislukuaritmetiikalla.

iii. Lopulta voidaan tehdä visuaalisia testejä. Eräs yksinkertaisimmista ja ilmeisimmistä on jo aikaisemmin mainittu satunnaislukuparien sijoituminen yksikköneliöön. Näin voidaan löytää sekä globaaleja että lokaaleja epäsatunnaisuuksia. Myös bittitasolla voidaan tehdä visuaalisia testejä.

iv. Viime aikoina on testaukseen käytetty myös tiettyjä fysikaalisia suureita[1]. Esimerkkinä autokorrelaatioajat Ising-mallin MC-simulaatiossa. Näin on löydetty mm. lyhyen kantaman korrelaatioita siirtorekisterigeneraattoreista.

Edellämainitussa julkaisussa[2] testattiin kahdeksaa satunnaislukugeneraattoria tilastollisilla, bittitason ja visuaalisilla testeillä.

Tulosten yhteenveto voidaan esittää seuraavana taulukkona (+ =hyvä ominaisuus, − =huono ominaisuus, • jättää tulkinnan avoimeksi).

	GGL	RAND	RANF	G05FAF	R250	RAN3	RANMAR	RCARRY
standarditestit	•	−	+	•	+	•	+	−
bittitestit	+	−	•	+	+	−	+	+
nopeustestit	+	•	+	+	+	•	•	•

Alla vielä testattujen generaattorien tyypit:

GGL:	MLCG
RAND:	LCG
RANF:	2×MLCG
G05FAF:	MLCG
R250:	GFSR
RAN3:	LF
RANMAR:	LF+aritmeettinen jono
RCARRY:	LF (add-and-carry)

Huomataan, että generaattoriluokan valinta ei ole olennaisen tärkeää etsittäessä hyvää generaattoria.

Parhaiten testeissä menestyivät R250 ja RANMAR. R250 on kuitenkin alustettava kunnollisesti (ei bittitason korrelaatioita).

---

1. I. Vattulainen ym., *Phys. Rev. Lett.* **73** (1994) 2513.
2. I. Vattulainen ym., *CSC Research Reports* **R05/92**

Lisäksi on huomattava, että tämän tutkimuksen jälkeen on GFSR-generaattoreista löydetty korrelaatioita, jotka voivat joissain tapauksissa vaikuttaa simulaatiotuloksiin.

Johtopäätöksenä voisi todeta G. Marsaglian sanojen mukaan

A random number generator is much like sex:
when it is good it is wonderful,
and when it is bad it is still pretty good.

Eli enää ei tarvitse välittää J. von Neumannin huomautuksesta

Anyone who considers arithmetic methods of
producing random digits
is, of course, in a state of sin.

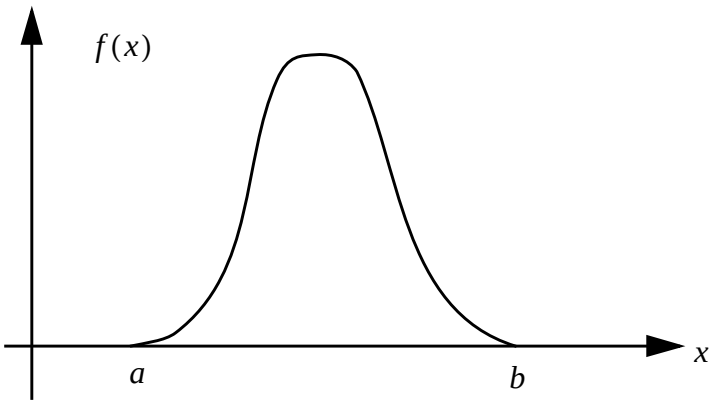
8.2 Erilaisten jakaumien tuottaminen

Laskennalisessa tieteessä tarvitaan suuria määriä eri tavoin jakautuneita satunnaislukuja. Miten saamme niitä aikaan olettaen, että käytössä on satunnaislukugeneraattori, joka antaa tasaisesti välillä $0 \rightarrow 1$ jakautuneita lukuja? Satunnaislukugeneraattoreista puhutaan hieman myöhemmin.

Olkoon haluttu todennäköisyysjakauma $f(x)$ välillä $[a, b]$. Sille tietysti pätee

$$f(x) \geq 0, \forall x \in [a, b] \tag{8.15}$$

$$\int_a^b f(x)dx = 1 \quad . \tag{8.16}$$

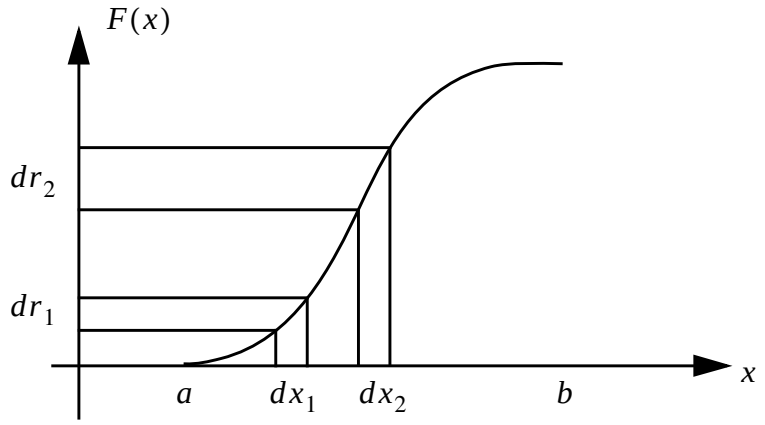


Kuva 8.8: Todennäköisyysjakauma.

Olkoon $F(x)$ kumulatiivinen jakauma

$$F(x) = \int_a^x f(x')dx' \equiv r \quad . \tag{8.17}$$

Määritelmän mukaan voimme siis kuvata $F(x)$:n satunnaismuuttujalle $r \in [0, 1]$. Tarkastellaan kuvan mukaisia yhtäsuuria alueita dx_1 ja dx_2 paikoissa x_1 ja x_2 .



Kuva 8.9: Kumulatiivinen todennäköisyysjakauma.

Helposti näemme, että

$$\frac{dr_1}{dr_2} = \frac{(dF(x)/dx)|_{x=x_1}}{(dF(x)/dx)|_{x=x_2}} = \frac{f(x_1)}{f(x\bar{2})} \quad . \tag{8.18}$$

Eli valitsemalla tasaisesti jakautuneita satunnaislukuja, niiden lukujen lukumäärä, jotka osuvat välille dr_1 suhteessa välille dr_2 osuneisiin on näiden pisteiden todennäköisyystiheyksien suhde.

Halutulla tavalla jakautuneita satunnaislukuja x saadaan tasaisesti jakautuneista luvuista r kumulatiivisen jakauman käänteisfunktion avulla

$$x = F^{-1}(r) \quad . \tag{8.19}$$

Kaikilla laillisilla todennäköisyystiheysfunktioilla on käänteisfunktio.

Ensimmäisenä esimerkkinä olkoon fotonin kulkema matka z väliaineessa (yksiköissä keskimääräinen vapaa matka). Todennäköisyysjakauma on muotoa

$$f(z) = e^{-z} \quad , \tag{8.20}$$

josta saadaan kumulatiivinen jakauma

$$F(z) = \int_0^z e^{-z'} dz' = 1 - e^{-z} \quad . \tag{8.21}$$

Tästä edelleen

$$z = -\ln(1 - r) \quad . \tag{8.22}$$

Koska $1 - r$ ja r ovat samalla tavalla jakautuneet voidaan käyttää muotoa

$$z = -\ln r \quad . \tag{8.23}$$

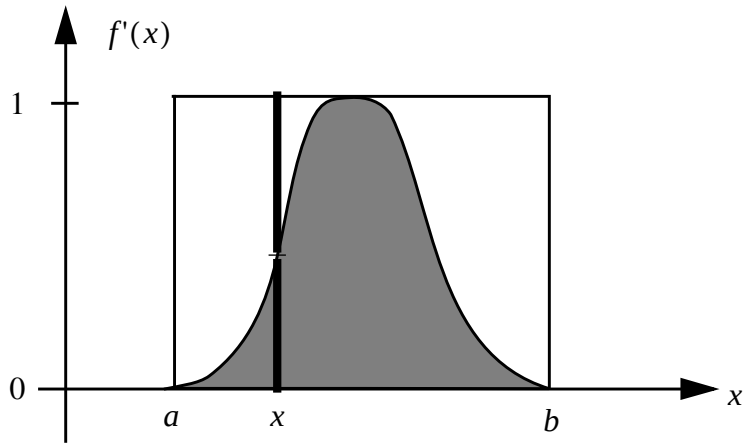
Aina ei ole mahdollista muodostaa funktiota F^{-1} . Tällöin voidaan käyttää hylkäysmenetelmää:

i. Muodostetaan todennäköisyysjakaumasta uusi funktio

$$f'(x) = \frac{f(x)}{f_{\max}} \quad . \tag{8.24}$$

ii. Generoidaan tasaisesti jakautunut satunnaisluku $r_1 \in [0, 1]$. Normitetaan tämä välille $[a, b]$: $x = a + (b - a)r_1$. Jos väli $[a, b]$ ei ole äärellinen on tehtävä muunnos, jolla se palautetaan äärelliseksi. Esim. $x \in [a, \infty]$ voidaan palauttaa muunnoksella $x = a[1 - \ln(1 - y)]$ välille $y \in [0, 1]$.

iii. Generoidaan satunnaisluku $r_2 \in [0, 1]$. Jos $r_2 < f'(x)$, hyväksytään x , muuten palataan askeleeseen 2.



Kuva 8.10: Hylkäysmenetelmä.

Menetelmässä tietysti tuhlataan satunnaislukuja; etenkin, jos $f(x)$ on kovin piikittynyt.

Näiden kahden menetelmän yhdistelmää voidaan myös käyttää, jos jakauma on piikittynyt ja vaikeasti käännettävissä ja integroitavissa. Jaamme funktion osiin

$$f(x) = g(x)h(x) \quad , \tag{8.25}$$

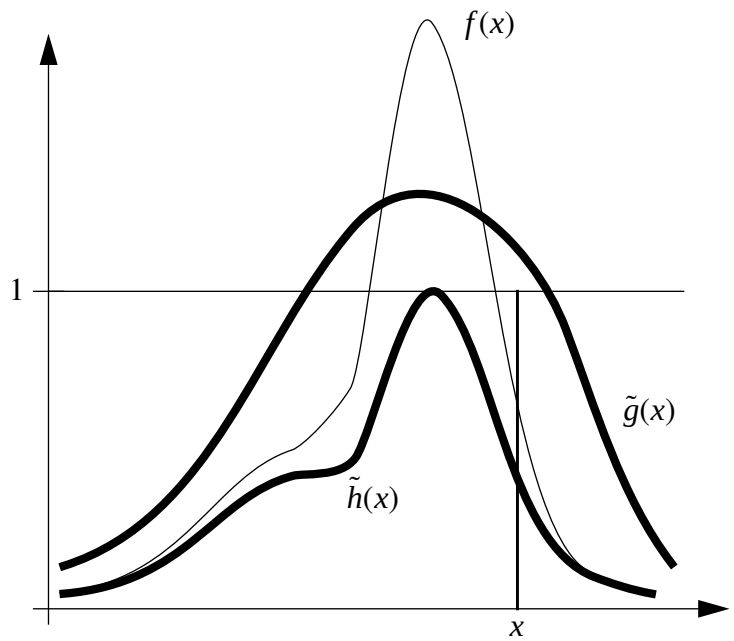
missä $g(x)$ on helposti käännettävissä ja sisältää enimmän piikittyneisyyden ja $h(x)$ on melko tasainen, mutta matemaattisesti monimutkainen. Satunnaisluvut voidaan generoida seuraavan reseptin mukaan.

i. Skaalataan $g(x) \to \tilde{g}(x)$ siten, että $\int_a^b \tilde{g}(x)dx = 1$.

ii. Skaalataan $h(x) \to \tilde{h}(x)$ siten, että $\tilde{h}(x) \le 1 \quad \forall x \in [a, b]$.

iii. Käyttäen edellämainittua suoraa menetelmää generoidaan satunnaisluku x, joka noudattaa jakaumaa $\tilde{g}(x)$.

iv. Käyttäen satunnaislukua x sovelletaan hylkäysmenetelmää jakaumafunktiona nyt $\tilde{h}(x)$: Generoidaan satunnaisluku $r \in [0, 1]$; jos $\tilde{h}(x) \le r$, hyväksy x, muuten mene askeleeseen iii.



Kuva 8.11: Suoran ja hylkäysmenetelmän yhdistelmä.

9 Datan tilastollinen kuvailu

9.1 Yleistä

Tässä luvussa tarkastelemme datajoukkojen ominaisuuksia. Data muodostuu joukosta numeroita, jotka on saatu esimerkiksi mittauksista tai simulaatioista. Käymme läpi dataa kuvaavia tunnuslukuja sekä menetelmiä, joilla arvioidaan kahden datajoukon samuutta tai erilaisuutta.

9.2 Todennäköisyysjakautumat

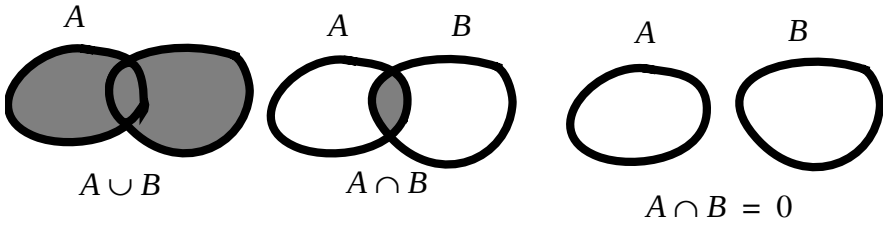
9.2.1 Yleistä

Todennäköisyyden avulla voimme kuvata enemmän tai vähemmän kvantitatiivisesti jonkin tapahtuman tai 'kokeen' odotettavissa olevaa tulosta.

Jos tapahtuman A todennäköisyys on $P(A)$, voimme odottaa, että N:n identtisen kokeen tuloksena saamme $NP(A)$ kappaletta tapahtumia A. Rajalla $N \to \infty$, tapahtumien A osuus lähestyy arvoa $P(A)$.

Kokeen *otosavaruus* (*sample space*) S on kokeen mahdollisten tulosten joukko. Siis jokainen kokeen tulos vastaa yhtä tai useampaa joukon alkiota (otosavaruuden pistettä). *Koe* tai *tapahtuma* on S:n osajoukko.

Kahden tapahtuman unioni $A \cup B$ koostuu kaikista pisteistä, jotka kuuluvat joko tapahtumaan A tai B. Leikkaus $A \cap B$ koostuu pisteistä, jotka kuuluvat molempiin tapahtumiin. Jos $A \cap B = \varnothing$ tapahtumat ovat toisensa poissulkevia.



Kuva 9.1: Tapahtumat A ja B.

Todennäköisyys, että saadaan joko tulos A tai B on

$$P(A \cup B) = P(A) + P(B) - P(A \cap B) \quad , \tag{9.1}$$

missä $P(A \cap B)$ on todennäköisyys, että saadaan sekä A että B.

Jos tapahtumat $A_1, A_2, \ldots, A_m$ ovat toisensa poissulkevia ja lisäksi A_i:t jakavat S:n osiin:

$$A_1 \cup A_2 \cup \ldots \cup A_m = S \quad , \tag{9.2}$$

niin pätee

$$P(A_1) + P(A_2) + \ldots + P(A_m) = 1 \quad . \tag{9.3}$$

Tapahtumat A ja B ovat riippumattomia, jos

$$P(A \cap B) = P(A)P(B) \quad . \tag{9.4}$$

Ehdollinen todennäköisyys on todennäköisyys, että tapahtuma A toteutuu ehdolla, että myös tapahtuma B toteutuu. Se määritellään

$$P(B|A) = \frac{P(A \cap B)}{P(B)} \quad . \tag{9.5}$$

Koska $P(A \cap B) = P(B \cap A)$ pätee myös

$$P(A)P(A|B) = P(B)P(B|A) \quad . \tag{9.6}$$

Jos A ja B ovat riippumattomia

$$P(B|A) = P(A) \quad . \tag{9.7}$$

Suure, jonka arvon määrä edelläesitellyn kokeen tulos on satunnaismuuttuja tai stokastinen muuttuja. Otosavaruuden S satunnaismuuttuja X on funktio, joka kuvaa S:n alkiot reaalilukujoukolle.

Jokaisessa kokeessa muuttuja X voi saada jonkin arvon joukosta $\{x_i\}$. Pari esimerkkiä X:stä:

i. kruunujen lukumäärä kolmen kolikon heiton jälkeen

ii. noppien silmälukujen maksimi, neljän nopan heiton jälkeen.

Olkoon X stokastinen muuttuja avaruudessa S. Olkoot sallitut arvot $X(S) = \{x_1, x_2, \ldots\}$. Voimme tehdä $X(S)$:stä otosavaruuden antamalla jokaiselle x_i:lle todennäköisyyden. Nämä todennäköisyydet $f(x_i)$ määrittelevät S:n todennäköisyysjakauman ja niille pätee

$$f(x_i) \geq 0 \tag{9.8}$$

$$\sum_i f(x_i) = 1 \quad . \tag{9.9}$$

Usein meillä on tietoa vain jakauman f momenteista:

$$\langle X^n \rangle = \sum_i x_i^n f(x_i) \quad . \tag{9.10}$$

Ensimmäinen momentti $\langle X \rangle$ on keskiarvo ja jakauman standardipoikkema on

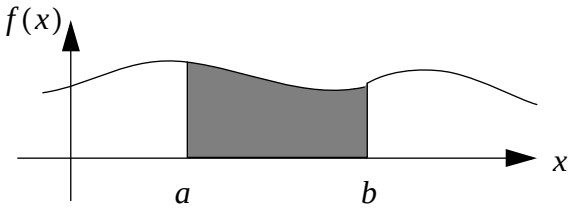
$$\sigma_X \equiv (\langle X^2 \rangle - \langle X \rangle^2)^{1/2} \quad . \tag{9.11}$$

Stokastinen muuttuja X voi tietysti saada myös jatkuvia arvoja. Esimerkiksi reaalilukuakselin väli $a \le X \le b$ voi vastata yhtä tapahtumaa. Todennäköisyysjakauma on sellainen paloittain jatkuva funktio, että tapahtuman $a \le X \le b$ todennäköisyys on

$$P(a \le X \le b) = \int_a^b f_X(x)dx \quad . \tag{9.12}$$

Lisäksi jakaumafunktio toteuttaa ehdot

$$f_X(x) \ge 0 \tag{9.13}$$



Kuva 9.2: Todennäköisyysjakauma

ja

$$\int f_X(x)dx = 1 \quad . \tag{9.14}$$

Vastaavasti momentit määritellään

$$\langle X^n \rangle = \int x^n f_X(x)dx \quad . \tag{9.15}$$

Jos tunnemme f_X :n kaikki momentit, tunnemme jakauman täysin. Tämä voidaan osoittaa ns. karakteristisen funktion $\phi_X(k)$ avulla:

$$\phi_X(k) = \langle e^{ikX} \rangle = \int e^{ikx} f_X(x)dx = \sum_{n=0}^{\infty} \frac{(ik)^n \langle X^n \rangle}{n!} \quad . \tag{9.16}$$

Todennäköisyystiheys on karakteristisen funktion Fourier-muunnos:

$$f_X(x) = \frac{1}{2\pi} \int e^{-ikx} \phi_X(k)dk \quad . \tag{9.17}$$

Vastaavasti jakauman momentit saadaan karakteristisen funktion derivaattoina:

$$\langle X^n \rangle = \frac{1}{i^n} \left[\frac{d^n}{dk^n} \phi_X(k) \right]_{k=0} \quad . \tag{9.18}$$

Joskus käytetään myös ns. kumulanttikehitelmää

$$\phi_X(k) = \exp\left[\sum_{n=1}^{\infty} \frac{(ik)^n}{n!} C_N(X)\right]$$

$$C_1(X) = \langle X \rangle \quad , \quad C_2(X) = \langle X^2 \rangle - \langle X \rangle^2 \quad ,$$

$$C_3(X) = \langle X^3 \rangle - 3\langle X \rangle \langle X^2 \rangle + 2\langle X \rangle^3 \quad , \text{jne.} \tag{9.19}$$

Stokastisia muuttujia voi olla useampiakin. Olkoon meillä muuttujat $X(S) = \{x_1, x_2, \ldots\}$ ja $Y(S) = \{y_1, y_2, \ldots\}$. Näiden tulojoukko $X(S) \times Y(S) = \{(x_1, y_1), (x_1, y_2), \ldots, (x_i, y_j), \ldots\}$ muodostaa nyt otosavaruuden, kun määrittelemme parin $\{x_i, y_j\}$ todennäköisyydeksi

$$P(X = x_i, Y = y_j) = f(x_i, y_j) \quad . \tag{9.20}$$

Jatkuvien muuttujien tapauksessa todennäköisyystiheys on $f(x, y)$. Myös tälle pätee

$$f(x, y) \geq 0 \tag{9.21}$$

$$\int f(x, y) dx dy = 1 \quad . \tag{9.22}$$

Muuttujien kovarianssi määritellään

$$\begin{aligned} \operatorname{cov}(X, Y) &= \int (x - \langle X \rangle)(y - \langle Y \rangle) f(x, y) dx dy \\ &= \int xy f(x, y) dx dy - \langle X \rangle \langle Y \rangle \\ &= \langle XY \rangle - \langle X \rangle \langle Y \rangle \end{aligned} \quad . \tag{9.23}$$

ja korrelaatio

$$\operatorname{cor}(X, Y) = \frac{\operatorname{cov}(X, Y)}{\sigma_X \sigma_Y} \quad . \tag{9.24}$$

Korrelaatiolla on seuraavat ominaisuudet

(i) $\operatorname{cor}(X, Y) = \operatorname{cor}(Y, X)$,

(ii) $-1 \leq \operatorname{cor}(X, Y) \leq 1$,

$$\text{(iii)} \quad \mathrm{cor}(X, X) = 1, \mathrm{cor}(X, -X) = -1 \quad ,$$
$$\text{(iv)} \quad \mathrm{cor}(aX + b, cY + d) = \mathrm{cor}(X, Y) \quad \text{, jos } a, c \neq 0 \quad . \tag{9.25}$$

Jos muuttujat X ja Y ovat riippumattomia pätevät seuraavat relaatiot

$$\text{(i')} \quad f(x, y) = f_X(x) f_Y(y) \quad ,$$
$$\text{(ii')} \quad \langle XY \rangle = \langle X \rangle \langle Y \rangle \quad ,$$
$$\text{(iii')} \quad \langle (X + Y)^2 \rangle - \langle X + Y \rangle^2 = \langle X^2 \rangle - \langle X \rangle^2 + \langle Y^2 \rangle - \langle Y \rangle^2 \quad ,$$
$$\text{(iv')} \quad \mathrm{cov}(X, Y) = 0 \quad . \tag{9.26}$$

Huomaa, että ehto (iv') ei välttämättä implikoi sitä, että muuttujat ovat riippumattomia.

Ylläolevat asiat voidaan helposti yleistää useammalle kuin kahdelle satunnaismuuttujalle.

9.2.2 Tärkeimpiä jakaumia

Usein meillä on kyseessä tilanne, jossa on suuri määrä N kokeita, joilla jokaisella on kaksi mahdollista tulosta ($+1$ ja -1). Esimerkiksi hiukkanen joko siroaa tietyllä matkalla tai sitten ei.

Olkoot todennäköisyydet tuloksille p ja q. Selvästikin $p + q = 1$.

Todennäköisyys, että N:n kokeen tuloksena on n_1 kertaa $+1$ ja n_2 kertaa -1 on

$$P_N(n_1) = \frac{N!}{n_1! n_2!} p^{n_1} q^{n_2} \quad . \tag{9.27}$$

Jakauman keskiarvo ja standardipoikkeama ovat

$$\langle n_1 \rangle = pN \quad , \tag{9.28}$$

$$\sigma_N^2 = Npq \quad . \tag{9.29}$$

Binomijakaumasta saadaan rajalla $N \to \infty$ ja $pN \to \infty$ (siis p ei ole kovin pieni) Gaussin jakauma

$$P_N(n_1) = \frac{1}{\sigma_N \sqrt{2\pi}} \exp\left[-\frac{1}{2} \frac{(n_1 - \langle n_1 \rangle)^2}{\sigma_N^2} \right] \quad , \tag{9.30}$$

missä

$$\sigma_N = \sqrt{Npq} \quad . \tag{9.31}$$

Gaussin jakauman määräävät kaksi ensimmäistä momenttia $\langle n_1 \rangle$ ja σ_N .

Taasen rajalla $N \to \infty$ ja $p \to 0$ siten, että $Np = a \ll N$ (missä a on äärellinen vakio) binomijakaumaa voidaan approksimoida Poissonin jakaumalla

$$P_N(n_1) = \frac{a^{n_1} e^{-a}}{n_1!} \quad . \tag{9.32}$$

Poissonin jakauman määrää ensimmäinen momentti

$$\langle n_1 \rangle = a \quad . \tag{9.33}$$

Esimerkkinä binomijakauman käytöstä olkoon yksiulotteinen satunnaiskävelijä. Todennäköisyys, että kävelijä ottaa askeleen vasemmalle on $p = 1/2$ ja oikealle $q = 1/2$. Askelten lukumäärä on N ja vasemmalle suuntautuneiden askelten lukumäärä on n_1 ja oikealle n_2 . Poikkeama origosta on siis $m = n_1 - n_2$. Koska $N = n_1 + n_2$ on $m = 2n_1 - N$. Suurilla N :n arvoilla poikkeaman tn.jakauma saadaan sijoittamalla $n_1 = (m + N)/2$ Gaussin jakaumaan (9.30):

$$P_N(m) = \left(\frac{2}{\pi N}\right)^{1/2} e^{-m^2/2N} \quad . \tag{9.34}$$

Jos askeleen pituus on l on poikkeama $x = ml$. Todennäköisyys, että kävelijä on välillä $[x, x + \Delta x]$ N :n askeleen jälkeen on $P_N(x) = P_N(m)(\Delta x/2l)$ eli

$$P_N(x) = \frac{1}{(2\pi N l^2)^{1/2}} e^{-x^2/2Nl^2} \quad . \tag{9.35}$$

Jos kävelijä ottaa n askelta aikayksikössä, saadaan todenäköisyystiheydeksi ajanhetkellä t

$$P(x, t) = \frac{1}{2(\pi D t)^{1/2}} e^{-x^2/4Dt} \quad . \tag{9.36}$$

Gaussin jakauman tärkeys perustuu ns. *keskeisraja-arvoteoreemaan* (*central limit theorem*).

Haluamme tietää, satunnaismuuttujan Y jakaumasta. Y :n arvot vastaavat N :ää muuttujan X mittausta:

$$y_N = \frac{x_1 + x_2 + \ldots + x_N}{N} \quad . \tag{9.37}$$

Tutkitaan todennäköisyysjakaumaa $f_Y(y_n - \langle X\rangle)$. Sen karakteristinen funktio voidaan kirjoittaa muodossa

$$\begin{aligned}\Phi(k) &= \int e^{ik(y_N - \langle X\rangle)} f_Y(y_N - \langle X\rangle) dy_N \\ &= \int e^{i(k/N)((x_1 - \langle X\rangle) + \ldots + (x_N - \langle X\rangle))} f_X(x_1)\ldots f_X(x_N) dx_1 \ldots dx_N \\ &= \left[\phi\left(\frac{k}{N}\right)\right]^N\end{aligned}$$

. (9.38)

Jos $\sigma^2 = \langle X^2\rangle - \langle X\rangle^2$, niin

$$\phi\left(\frac{k}{N}\right) = \int e^{i(k/N)(x_1 - \langle X\rangle)} f_X(x_1) dx_1 = 1 - \frac{1}{2}\frac{k^2}{N^2}\sigma^2 + \ldots \quad (9.39)$$

Tämä funktio pienenee nopeasti k:n kasvaessa, johtuen oskillaatiotermistä $e^{i(k/N)x_1}$, ja $[\phi(k/N)]^N$ pienenee tätäkin nopeammin. Olettaen, että jakauman $f_X(x_1)$ momentit ovat äärellisiä, saadaan

$$\Phi(k) = \left[1 - \frac{1}{2}\frac{k^2}{N^2}\sigma^2 + O\left(\frac{k^3}{N^3}\right)\right]^N \to e^{-k^2\sigma^2/2N} \qquad \text{,kun } N \to \infty \text{ .} \quad (9.40)$$

Tästä saamme lopulta todennäköisyystiheydeksi

$$\begin{aligned}f_Y(y_N - \langle X\rangle) &= \frac{1}{2\pi}\int_{-\infty}^{\infty} e^{-ik(y_N - \langle X\rangle)}\Phi(k) dk \qquad . \quad (9.41)\\ &= \frac{1}{2\pi}\int_{-\infty}^{\infty} e^{-ik(y_N - \langle X\rangle)} e^{-k^2\sigma^2/2N} dk \\ &= \sqrt{\frac{N}{2\pi}}\frac{1}{\sigma}\exp\left[-\frac{N(y_N - \langle X\rangle)^2}{2\sigma^2}\right]\end{aligned}$$

Tämä on tärkeä tulos, sillä sen mukaan jakaumasta $f_X(x)$ riippumatta tulos on gaussinen. Ehtona on $f_X(x)$:n momenttien olemassaolo.

Toinen tärkeä teoreema on *suurten lukujen laki* (*law of large numbers*). Sen mukaan tapahtumien A (todennäköisyys p) osuus N:stä riippumattomasta kokeesta lähestyy arvoa p, kun $N \to \infty$.

Olkoon meillä taas keskiarvo $y_N = (x_1 + x_2 + \ldots + x_N)/N$. Suurten lukujen laki sanoo, että todennäköisyys, että y_N poikkeaa $\langle X \rangle$:stä lähestyy nollaa, kun $N \to \infty$. Eli

$$\lim_{N \to \infty} P(|y_N - \langle X \rangle| \geq \varepsilon) = 0 \quad , \tag{9.42}$$

missä ε positiivinen luku.

Lopuksi voisi mainita vielä pari muuta tärkeää jakaumaa.

Eksponenttijakauma

$$f(x) = \lambda e^{-\lambda x} \quad ; \quad x \geq 0 \ , \ \lambda > 0 \tag{9.43}$$

kuvaa esimerkiksi radioaktiivista hajoamista tai fotonin kulkemaa vapaata matkaa väliaineessa.

Lorentzin jakauma

$$f(x) \propto \frac{a}{a^2 + x^2} \ ; \quad -\infty < x < \infty \tag{9.44}$$

on siitä outo jakauma, että sen toinen momentti divergoi. Lisäksi keskiarvon laskeminen ei suoraan onnistu, mutta jakauman symmetrian perusteella sen voi sanoa olevan 0. Lorentzin jakauman tyyppinen jakauma syntyy esimerkiksi kahden gaussisesti jakautuneen satunnaismuuttujan osamäärästä.

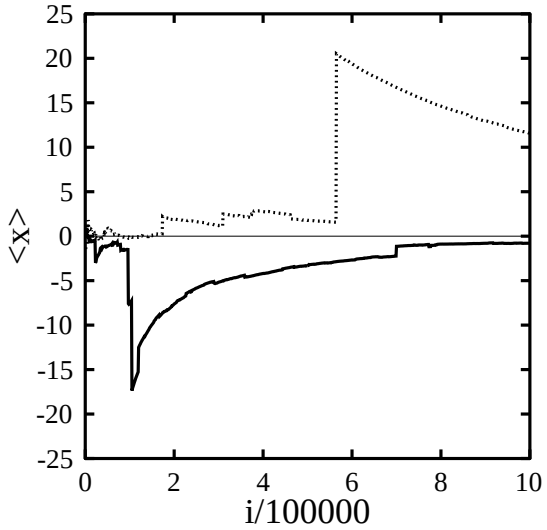
Seuraavalla sivulla on esimerkkinä demonstroitu, miten hitaasti Lorentzin jakauman keskiarvo suppenee verrrattuna Gaussin jakaumaan.

Lyhyt C-ohjelma, joka laskee keskiarvoa ja jakaumia:

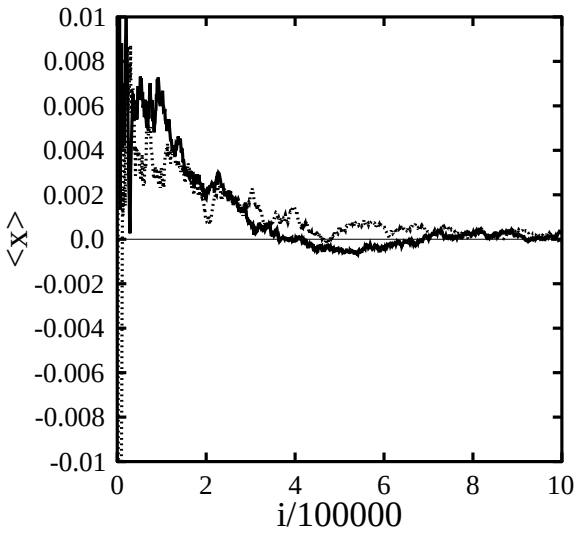
```
#include <stdio.h>
#include <math.h>
#define NRANSI
#include "nr.h"
#include "nrutil.h"
#define NMAX  500
#define SCALE 0.05
#define PI 3.141592654

int main(int argc, char **argv)
{
  int imax,i,*dg,*dl,ind,iout;
  double rg,rl,rga,rla;
  long seed1,seed2;

  if (argc!=5) {
    fprintf(stderr,"Usage: %s nmax seed1 seed2 iout\n",argv[0]);
    return (1);
  }
  imax=atoi(*++argv);
  seed1=atoi(*++argv);
  seed2=atoi(*++argv);
  iout=atoi(*++argv);

  dg=ivector(0,NMAX);
  dl=ivector(0,NMAX);

  for (i=0;i<=NMAX;i++) {
    dg[i]=0;
    dl[i]=0;
  }

  rga=0.0;
  rla=0.0;
  for (i=1;i<=imax;i++) {
    rg=gasdev(&seed1);
    rl=tan(PI*ran3(&seed2));
    rga+=rg;
    rla+=rl;
    ind=(int) (sqrt(rg*rg)*SCALE*NMAX);
    if (ind>=0 && ind<=NMAX) dg[ind]++;
    ind=(int) (sqrt(rl*rl)*SCALE*NMAX);
    if (ind>=0 && ind<=NMAX) dl[ind]++;
    if (i%iout==0) printf("AVE: %d %g %g\n",i,rga/i,rla/i);
  }
  for (i=0;i<=NMAX;i++) {
    printf("DIST: %g %d %d\n",i/SCALE/NMAX,dg[i],dl[i]);
  }
}
```

Ohjelma käyttää NR:n aliohjelmia `ran3` ja `gasdev`, joista jälkimmäinen muutettiin käyttämään `ran3`:sta `ran1`:n sijaan. Tästä syystä sille piti tehdä oma kopio `ran3`:sta (nimellä `ran3x`).

Alla Lorentzin ja Gaussin jakauman mukaisten satunnaislukujen keskiarvo lukujen lukumäärän funktiona.

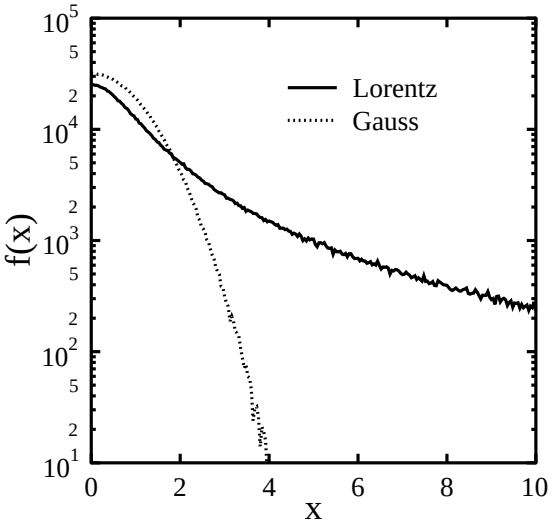


Kuva 9.3: Lorentzin jakaumaa (9.44) ($a = 0$) noudattavien satunnaislukujen keskiarvo.



Kuva 9.4: Gaussin jakaumaa $\exp(-x^2/2)$ noudattavien satunnaislukujen keskiarvo.

Lisäksi alla arpomisen tuloksena saadut jakaumat.



Kuva 9.5: Lorentzin ja Gaussin jakaumat.

9.3 Datajoukkojen vertailu

Usein joudumme tilanteeseen, jossa on vertailtava kahta datajoukkoa, ja tehtävä johtopäätöksiä siitä, voisivatko ne olla otoksia samasta todennäköisyysjakaumasta. Olkoot datajoukot

$$\begin{matrix} \{x_{\mathrm{A}1}, x_{\mathrm{A}2}, \ldots, x_{\mathrm{A}N_{\mathrm{A}}}\} \\ \{x_{\mathrm{B}1}, x_{\mathrm{B}2}, \ldots, x_{\mathrm{B}N_{\mathrm{B}}}\} \end{matrix} . \tag{9.45}$$

Näiden keskiarvot ovat

$$\begin{aligned} \bar{x}_{\mathrm{A}} &= \frac{1}{N_{\mathrm{A}}} \sum_{i=1}^{N_{\mathrm{A}}} x_{\mathrm{A}i} \\ \bar{x}_{\mathrm{B}} &= \frac{1}{N_{\mathrm{B}}} \sum_{i=1}^{N_{\mathrm{B}}} x_{\mathrm{B}i} \end{aligned} , \tag{9.46}$$

ja varianssit

$$\begin{aligned} \mathrm{Var(A)} &= \frac{1}{N_\mathrm{A}-1}\sum_{i=1}^{N_\mathrm{A}}(x_{\mathrm{A}i}-\bar{x}_\mathrm{A})^2 \\ \mathrm{Var(B)} &= \frac{1}{N_\mathrm{B}-1}\sum_{i=1}^{N_\mathrm{B}}(x_{\mathrm{B}i}-\bar{x}_\mathrm{B})^2 \end{aligned} \quad . \tag{9.47}$$

Standardipoikkeamat ovat näiden neliöjuuria

$$\begin{aligned} \sigma_\mathrm{A} &= \sqrt{\mathrm{Var(A)}} \\ \sigma_\mathrm{B} &= \sqrt{\mathrm{Var(B)}} \end{aligned} \quad . \tag{9.48}$$

Datajoukkojen vertailussa ensimmäiseksi mieleentuleva ajatus on tutkia joukkojen keskiarvojen etäisyyttä $\left|\bar{x}_\mathrm{A}-\bar{x}_\mathrm{B}\right|$ ja verrata sitä standardipoikkeamiin. Toinen asia on, onko tämä etäisyys statistisesti merkittävä. Etäisyys voi olla hyvinkin pieni verrrattuna standardipoikkeamiin, mutta jos dataa on suuri määrä tämä ero voi olla merkittävä.

Järkevämpi vertailukohta on keskiarvojen keskivirhe $\sigma_{\bar{x}_\mathrm{A}}$ tai $\sigma_{\bar{x}_\mathrm{B}}$. Se on otoksesta lasketun keskiarvon standardipoikkeama. Se kertoo siitä, miten erilaisten otosten keskiarvot ovat jakautuneet. Voidaan osoittaa (ks. seuraava sivu), että

$$\sigma_{\bar{x}_\mathrm{A}} = \frac{\sigma_\mathrm{A}}{\sqrt{N_\mathrm{A}}} \quad . \tag{9.49}$$

Otoksesta laskettu keskiarvo on sitä tarkempi, mitä enemmän dataa on käytetty sen laskemiseen.

Kaavan (9.49) todistus on melko suoraviivainen. Otoksen keskiarvo voidaan kirjoittaa muotoon

$$\bar{x} = \sum_{i=1}^{N}\frac{1}{N}x_i \quad . \tag{9.50}$$

Varianssin määritelmästä voidaan johtaa kaava

$$\begin{aligned} &\mathrm{Var}(a_1x_1 + a_2x_2 + \ldots + a_Nx_N) \\ &= a_1^2\mathrm{Var}(x_1) + a_2^2\mathrm{Var}(x_2) + \ldots + a_N^2\mathrm{Var}(x_N) \end{aligned} \quad . \tag{9.51}$$

Tämä pätee, jos muuttujat x_i ovat riippumattomia. Koska otoksen alkiot ovat riippumattomia, saamme

$$\mathrm{Var}(\bar{x}) = \sum_{i=1}^{N}\frac{1}{N^2}\mathrm{Var}(x_i) \quad . \tag{9.52}$$

Koska otoksen alkiot ovat kaikki samasta populaatiosta, jonka varianssi on $\mathrm{Var}(x)$, saadaan

$$\mathrm{Var}(\bar{x}) = \sum_{i=1}^{N} \frac{1}{N^2}\mathrm{Var}(x) = \frac{1}{N^2}\sum_{i=1}^{N} \mathrm{Var}(x) = \frac{\mathrm{Var}(x)}{N} \; . \tag{9.53}$$

Tästä saadaankin yhteys standardipoikkeamille

$$\sigma(\bar{x}) = \frac{\sigma(x)}{\sqrt{N}} \; . \tag{9.54}$$

9.3.1 Studentin *t*-testi

Studentin t-testiä voidaan käyttää arvioimaan kahden otoksen keskiarvojen samuutta. Oletamme, että otosten jakaumien varianssit ovat samat. Ensin arvioidaan keskiarvojen erotuksen keskivirhettä s_D:

$$s_D = \sqrt{\frac{\sum_{i=1}^{N_\mathrm{A}} (x_{\mathrm{A}i} - \bar{x}_\mathrm{A})^2 + \sum_{i=1}^{N_\mathrm{B}} (x_{\mathrm{B}i} - \bar{x}_\mathrm{B})^2}{N_\mathrm{A} + N_\mathrm{B} - 2}\left(\frac{1}{N_\mathrm{A}} + \frac{1}{N_\mathrm{B}}\right)} \; . \tag{9.55}$$

Sitten lasketaan suure t

$$t = \frac{\bar{x}_\mathrm{A} - \bar{x}_\mathrm{B}}{s_D} \; . \tag{9.56}$$

Suure t noudattaa ns. Studentin jakaumaa $A(t|\nu)$, missä ν on vapausasteiden lukumäärä (tässä tapauksessa $\nu = N_\mathrm{A} + N_\mathrm{B} - 2$). $A(t_0|\nu)$ kertoo todennäköisyyden, jolla kahdelle samakeskiarvoiselle otokselle laskettu t on enintään t_0. Keskiarvot ovat oleellisesti erilaiset, jos (esimerkiksi) $A(t|\nu) > 0.99$. Merkitsevyystasoksi voimme kutsua suuretta $1 - A(t|\nu)$. Pieni arvo kertoo, että keskiarvot ovat merkittävästi erilaisia.

NR:n rutiini `ttest` laskee tämän merkitsevyystason. Lisäksi tarvitaan rutiinit `avevar` ja `betai`.

Jos otosten jakaumien varianssit ovat erilaiset voidaan edelleen käyttää *t*-testiä, mutta hieman modifioituna:

$$t = \frac{\bar{x}_\mathrm{A} - \bar{x}_\mathrm{B}}{[\mathrm{Var(A)}/N_\mathrm{A} + \mathrm{Var(B)}/N_\mathrm{B}]^{1/2}} \; , \tag{9.57}$$

$$\nu = \frac{[\mathrm{Var(A)}/N_\mathrm{A} + \mathrm{Var(B)}/N_\mathrm{B}]^2}{[\mathrm{Var(A)}/N_\mathrm{A}]^2/(N_\mathrm{A} - 1) + [\mathrm{Var(B)}/N_\mathrm{B}]^2/(N_\mathrm{B} - 1)} \; . \tag{9.58}$$

Tätä testiä varten on NR:n rutiini `tutest`.

Otosten jakaumien varianssien erilaisuuden testaukseen käytetään ns. F-testiä. Vastaava NR:n rutiini on `ftest`.

9.3.2 χ^2-testi

Yleistämme edellisen kappaleen ongelman kysymällä, ovatko kaksi datajoukkoa samasta jakaumasta peräisin. Eli täsmällisemmin sanottuna: voimmeko todistaa epätodeksi (tietyllä merkitsevyystasolla) nollahypoteesin, että kaksi otosta ovat peräisin samaa todennäköisyysjakaumaa noudattavasta populaatiosta.

χ^2-testillä voidaan tutkia histogrammiin jaoteltua dataa. Meillä on siis luvut N_i, jotka kertovat lokerossa i olevien tapahtumien lukumäärän. Jatkuvan datan (N kappaletta reaalilukuja) voi myösmuuttaa histogrammimuotoon. Tässä kuitenkin häviää informaatiota.

Verrataan histogrammia $\{N_i\}$ johonkin tunnettuun jakaumaan $\{n_i\}$. Tälle datalle laskemme χ^2:n

$$\chi^2 = \sum_{i=1}^{N_B} \frac{(N_i - n_i)^2}{n_i} \ . \tag{9.59}$$

Suuri χ^2:n arvo kertoo, että on epätodennäköistä, että data noudattaa tunnettua jakaumaa. Tuo suure χ^2 noudattaa ns. χ^2-jakaumaa (yllättäen). Jakauma $Q(\chi^2|\nu)$, missä ν on vapausasteiden lukumäärä kertoo todennäköisyyden, että kaavan (9.59) summa on suurempi kuin χ^2. Edellytyksenä on, että termit summassa (9.59) ovat gaussisesti jakautuneita ja niiden keskiarvo on nolla ja varianssi yksi. Suure (9.59) noudattaa χ^2-jakaumaa, jos lokeroja on suuri määrä tai jokaisessa lokerossa on paljon tapahtumia. Vapausasteiden lukumäärä on lokerojen lukumäärä N_B. Jos n_i:t on normitettu samaan kuin otoksen koko, on vapausasteiden lukumäärä $N_B - 1$.

Tarkastellaan hieman χ^2-jakaumaa. Merkitään

$$\chi^2 = \sum_{i=1}^{N} z_i^2 \ , \tag{9.60}$$

missä z_i ovat gaussisesti jakautuneita, ja niiden keskiarvo on nolla ja varianssi yksi. Jos näin ei ole, voimme aina skaalata

$$z_i \to \frac{(z_i - \bar{z}_i)^2}{\sigma_i^2} \ . \tag{9.61}$$

Haluamme tietää χ^2:n todennäköisyystiheysfunktion $f(\chi^2)$. Olkoon $\mathbf{z} = (z_1, z_2, \ldots, z_N)$. Nyt tiheysfunktio $\mathbf{z}$:n avulla on

$$f(\mathbf{z})d^N\mathbf{z} \propto e^{-\chi^2/2}d^N\mathbf{z} \ . \tag{9.62}$$

Tässä voimme ajatella χ^2:n olevan N-ulotteisen avaruuden vektorin neliön. Tilavuusalkio $d^N\mathbf{z}$ on verrannollinen suureeseen

$$\chi^{N-1}d\chi \ = \ \frac{1}{2}(\chi^2)^{(N-2)/2}d\chi^2 \ . \tag{9.63}$$

Eli tiheysfunktio χ^2:n funktiona on

$$f(\chi^2)d\chi^2 \ = \ C \cdot (\chi^2)^{N/2-1}e^{-\chi^2/2}d\chi^2 \ , \tag{9.64}$$

missä C on normitusvakio. Se saadaan yhtälöstä

$$\int_0^\infty f(\chi^2)d\chi^2 \ = \ 1 \quad . \tag{9.65}$$

Eli lopullinen tulos on

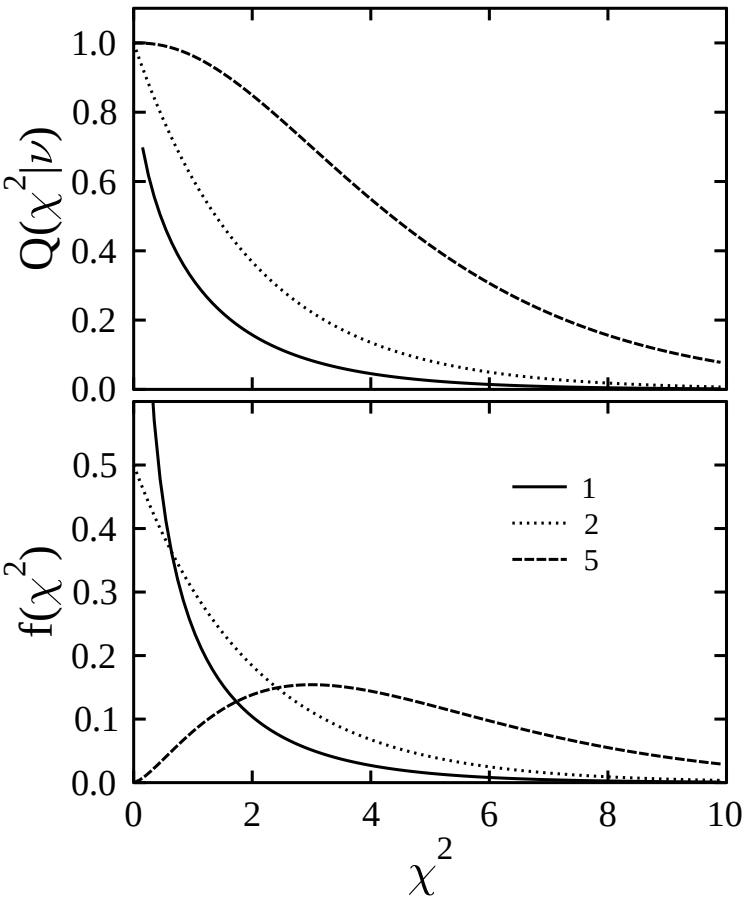
$$f(\chi^2)d\chi^2 \ = \ \frac{1}{\Gamma(N/2)}\left(\frac{\chi^2}{2}\right)^{N/2-1}e^{-\chi^2/2}d\left(\frac{\chi^2}{2}\right) \qquad , \tag{9.66}$$

missä Γ on gammafunktio. Tämä kertymäfunktio on siis jakauma $Q(\chi^2|N)$. NR:n erikois-funktiorutiinien avulla voimme laskea jakauman arvoja.

Alla esimerkkinä lyhyt ohjelmanpätkä:

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <math.h>
#define NRANSI
#include “nr.h”
#include “nrutil.h”

#define MAXSTR 80

int main(int argc, char **argv)
{
  float cmin,cmax,dof;
  float c,dc;
  int n,np;

  if (argc!=5) {
    fprintf(stderr,”Usage: %s cmin cmax np dof\n”,argv[0]);
    return (1);
  }
  cmin=atof(*++argv);
  cmax=atof(*++argv);
  np=atof(*++argv);
  dof=atoi(*++argv);

  dof=dof/2.0;
  dc=(cmax-cmin)/( (float) np);

  for (n=0;n<np;n++) {
    c=cmin+dc*n;
    printf(“%g %g \n”,c,gammq(dof,c/2.0));
  }
  return 0;
}
#undef NRANSI
```

Alla kuvat muutamalla eri vapausasteiden lukumäärän arvolla.



Kuva 9.6: χ^2 -jakaumat kolmelle eri vapausasteiden lukumäärälle.

χ^2 -testi voidaan tehdä NR:n rutiinilla `chsone`.

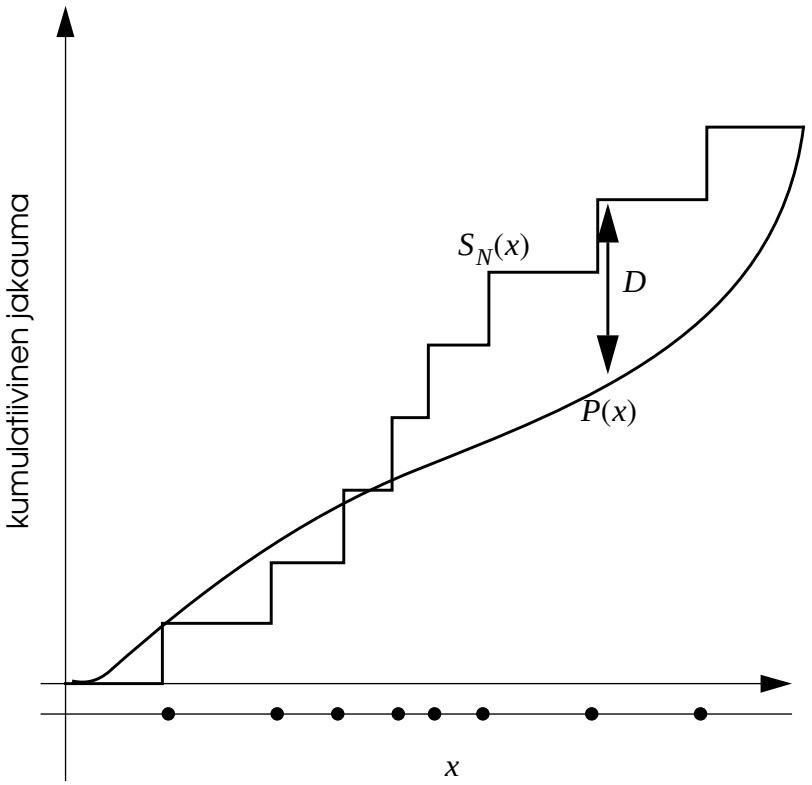
Jos vertailemme kahta lokeroitua datajoukkoa $\{R_i\}$ ja $\{S_i\}$, saadaan tarvittava suure kaavalla

$$\chi^2 = \sum_{i=1}^{N_B} \frac{(R_i - S_i)^2}{R_i + S_i} \ . \tag{9.67}$$

Erona yhtälöön (9.59) on ainoastaan summatermien nimittäjä. Se on siis suureen $R_i - S_i$ varianssi. Vapausasteiden lukumäärä taas on N_B, jos joukkoja ei ole normitettu, $N_B - 1$, jos summat yli lokerojen on normitettu samaksi. NR:n rutiini `chstwo` tekee tämän vertailun.

9.3.3 Kolmogorov-Smirnov-testi

Kolmogorov-Smirnov-testi (KS-testi) soveltuu lajittelemattoman datan ja jonkin tunnetun jakauman vertailuun. Olkoon datajoukko reaalilukuja $(x_i, i = 1, 2, \ldots N)$. Tästä on helppo laskea estimaatti kumulatiiviselle jakaumalle $P(x)$, merkitään sitä symbolilla $S_N(x)$. Se antaa niiden pisteiden x_i suhteellisen osuuden, joille pätee $x_i < x$.



Kuva 9.7: Kolmogorov-Smirnov-testi.

Jakaumien erilaisuutta voidaan kuvata jollain suureella, joka kertoo jotain oheisen kuvan käyrien väliin jäävästä pinta-alasta. KS-testissä täksi suureeksi otetaan käyrien erotuksen itseisarvon maksimi D:

$$D = \max_x \left| S_N(x) - P(x) \right| \ . \tag{9.68}$$

Vastaavasti, jos vertailemme kahta otosta:

$$D = \max_x \left| S_{N_1}(x) - S_{N_2}(x) \right| \ . \tag{9.69}$$

KS-testi hyödyllinen ominaisuus on, että se on invariantti x uudelleenparametrisoinnin suhteen. Esimerkiksi x:n ja $\ln x$:n merkitsevyystaso on sama.

KS-testin merkitsevyystaso voidaan approksimatiivisesti laskea seuraavalla kaavalla:

$$\begin{aligned}&\text{Todennäköisyys}(D > \text{havaittu}) \\ &= Q_{\mathrm{KS}}([\sqrt{N_e} + 0.12 + 0.11/\sqrt{N_e}]D)\end{aligned} \quad , \tag{9.70}$$

missä N_e on efektiivinen pisteiden lukumäärä ($N_e = N$ tai $N_e = (N_1 N_2)/(N_1 + N_2)$) ja

$$Q_{KS}(\lambda) = 2 \sum_{j=1}^{\infty} (-1)^{j-1} e^{-2j^2\lambda^2} \ . \tag{9.71}$$

NR:n rutiineista löytyvät `ksone` ja `kstwo` tekevät KS-testit.

9.4 Korrelaatioista

Monissa laskennallisen tieteen ongelmissa joudumme laskemaan datajoukosta keskiarvoja ja näiden standardipoikkeamia. Esimerkkinä Monte Carlo -simulaatiot, joista saadaan systeemin potentiaalienergia tai paine simulaatioaskeleen funktiona. Jos näiden datajoukkojen perättäiset alkiot olisivat täysin toisistaan riippumattomia, voisimme laskea suureen keskiarvon keskivirheen normaalilla tavalla.

Yleensä peräkkäiset datapisteet ovat kuitenkin korreloituneita, mikä mutkistaa asiaa hieman. Otetaan esimerkiksi jonkin simulointi, joka tuottaa tietoa systeemin suureen x arvosta simuloinnin kuluessa. Merkitään x:n arvoa simulointiaskeleella i x_i. Päämääränä on arvioida suureen x keskiarvoa ja sen keskivirhettä simulointiajon datasta: $\bar{x}$ ja $\sigma^2(\bar{x})$.

Oletetaan, että datajonon pituus on n_t. Suureen x keskiarvo $\bar{x}$ (tai sen estimaatti) saadaan tietysti seuraavasti

$$\bar{x} = \frac{1}{n_\mathrm{t}} \sum_{i=1}^{n_\mathrm{t}} x_i \ . \tag{9.72}$$

missä siis x_i on suureen x arvo simulaatioaskeleella i.
Jos suureen x arvot eri askeleilla (x_i) olisivat täysin statistisesti riippumattomia keskiarvon (9.72) keskivirhe olisi yksinkertaisesti

$$\sigma^2(\bar{x}) = \frac{\sigma^2(x)}{n_\mathrm{t}} \ , \tag{9.73}$$

missä

$$\sigma^2(x) = \overline{\delta x^2} = \frac{1}{n_\mathrm{t}} \sum_{i=1}^{n_\mathrm{t}} [x_i - \bar{x}]^2 \ . \tag{9.74}$$

Oletus x_i:n statistisesta riippumattomuudesta ei kuitenkaan aina päde. Esimerkiksi Monte Carlo-algoritmin siirtymien toteutuksesta peräkkäiset tilat ovat korreloituneita. Tämä korrelaatio tulisi jollain tavalla ottaa huomioon laskettaessa keskiarvon keskivirhettä kaavasta (9.73).

Voidaan osoittaa, että jos datapisteet ovat korreloituneita, muuttuu (9.73) muotoon

$$\sigma^2(\bar{x}) = \frac{1 + 2\tau_x}{n_\mathrm{t}} \sigma^2(x) , \tag{9.75}$$

missä τ_x on x:n ns. korrelaatiopituus (tai -aika):

$$\tau_x = \sum_{n=1}^{\infty} \phi_x(n) , \tag{9.76}$$

missä ϕ_x on x:n autokorrelaatiofunktio

$$\phi_x(n) = \frac{\overline{x_1 x_n} - \bar{x}^2}{\sigma(x)} . \tag{9.77}$$

Jos korrelaatiopituus $\tau_x \ll 1$ eli pisteitä on alunperin otettu niin harvaan, että niiden välillä ei ole korrelaatioita, saadaan yhtälö (9.73). Jos taas $\tau_x \gg 1$, saadaan muoto

$$\sigma^2(\bar{x}) = \frac{2\tau_x}{n_\mathrm{t}} \sigma^2(x) . \tag{9.78}$$

Keskiarvon keskivirhe voidaan siis laskea, jos tiedämme datan korrelaatiopituuden. Käytännössä näin ei ole, vaan on käytettävä jotain muuta menetelmää. Eräs käyttökelpoinen on ns. blocking-menetelmä.
Jaetaan data blokkeihin, joiden pituus on k. Blokkien kokonaismäärä on n_b, eli

$$n_\mathrm{b} k = n_\mathrm{t} . \tag{9.79}$$

Keskiarvo lasketaan jokaiselle blokille

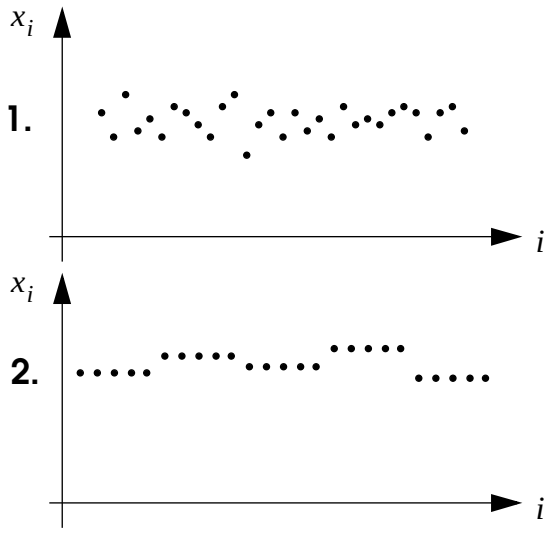
$$\bar{x}_b = \frac{1}{k} \sum_{i=1}^{k} x_i . \tag{9.80}$$

Näiden avulla voimme arvioida varianssin

$$\sigma^2(\bar{x}_\mathrm{b}) = \frac{1}{n_\mathrm{b}} \sum_{b=1}^{n_\mathrm{b}} [\bar{x}_b - \bar{x}]^2 . \tag{9.81}$$

Otetaan kaksi ääriesimerkkiä.
(Merkitään $E[X]$:llä suureen X keskiarvoa; siis $E[\bar{x}_\mathrm{b}]$ on blokkikeskiarvon odotusarvo.)

1. Oletetaan, että x:n arvot ovat täysin korreloimattomia, jolloin pätee $E[\bar{x}_\mathrm{b}] = E[x]$ ja $E[\bar{x}_{\mathrm{b}i}\bar{x}_{\mathrm{b}j}] = (E[x])^2$, kun $i \neq j$.



Kuva 9.8: Blokkikeskiarvot.

Tällöin saamme

$$\sigma^2(\bar{x}_\mathrm{b}) = E[\bar{x}_\mathrm{b}^2] - E[\bar{x}_\mathrm{b}]^2 \qquad (9.82)$$

$$= \frac{1}{k^2}E\left[\sum_{i=1}^{k} x_i^2\right] + \frac{1}{k^2}E\left[\sum_{i \neq j}^{k} x_i x_j\right] - E[x]^2$$

$$= \frac{1}{k}\{E[x^2] - (E[x])^2\} = \frac{1}{k}\sigma^2(x)$$

Eli se, mikä oli odotettavissakin.

2. Oletetaan nyt, että data koostuu pätkistä (pituus m) identtisiä arvoja x_j, missä $j = 1, \ldots, n_\mathrm{t}/m$. Oletetaan lisäksi, että arvot x_j ovat jakautuneet kuten x_i edellä. Tällöin saamme

$$\bar{x}_\mathrm{b} = \frac{1}{k}\sum_{i=1}^{k} x_i = \frac{1}{k}\sum_{j=1}^{k/m} m x_j \quad \text{ja} \qquad (9.83)$$

$$\bar{x}_\mathrm{b} = \frac{m^2}{k^2}\left\{\sum_{j=1}^{k/m} x_j^2 + \sum_{j \neq l}^{k/m} x_j x_l\right\} . \qquad (9.84)$$

Sijoittamalla nämä varianssin lausekkeeseen saamme

$$\sigma^2(\bar{x}_\mathrm{b}) = \frac{m}{k}\sigma^2(x) . \qquad (9.85)$$

Eli korrelaatioista johtuen (9.73):n antama varianssi on liian pieni.

Blokkikeskiarvojen varianssin (9.81) voidaan olettaa olevan kääntäen verrannollinen blokin kokoon k, kun koko kasvaa niin suureksi, että blokkien voidaan olettaa olevan tilastollisesti korreloimattomia:

$$\sigma^2(\bar{x}_\mathrm{b}) \propto \frac{1}{k} \ . \tag{9.86}$$

Päämääränä on löytää verrannollisuuskerroin, jolloin voimme arvioida blokkikeskiarvon varianssia blokille, jonka muodostaa koko simulointidata.

Tämä kerroin on ns. statistinen tehottomuus s, joka on raja-arvo, kun blokkien koko kasvaa rajatta:

$$s = \lim_{k \to \infty} \frac{k\sigma^2(\bar{x}_\mathrm{b})}{\sigma^2(x)} \ . \tag{9.87}$$

Se kertoo havaitun keskiarvon varianssin suhteen siihen varianssiin, joka saataisiin olettaen, että datassa ei ole ollenkaan korrelaatioita.

Jos tiedämme s:n saamme simulointidatan keskiarvon varianssin sjoittamalla yhtälöön (9.87) $k = n_\mathrm{t}$:

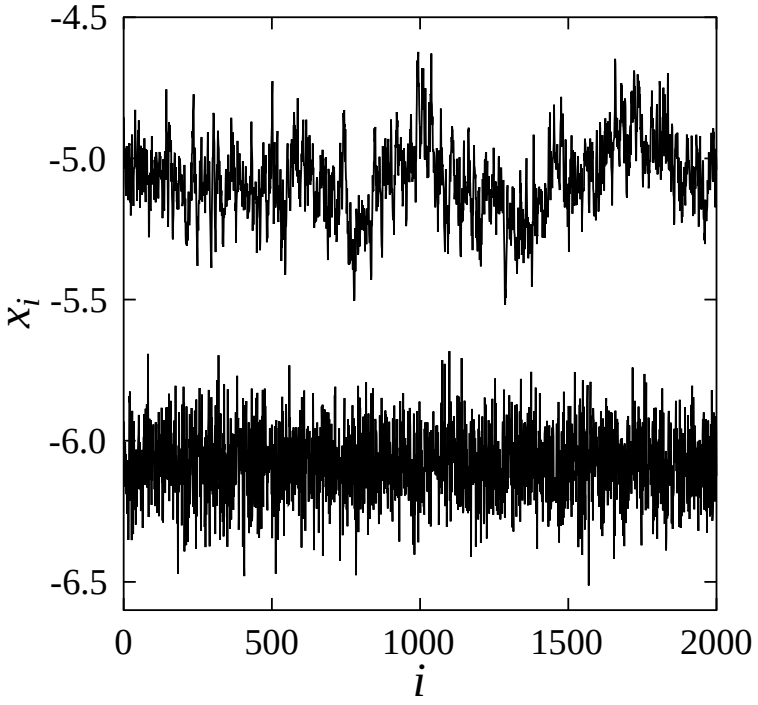
$$\sigma^2(\bar{x}) = \frac{s}{n_\mathrm{t}}\sigma^2(x) \ . \tag{9.88}$$

Statistinen tehottomuus pystytään arvioimaan simulointidatasta laskemalla

$$s(k) = \frac{k\sigma^2(\bar{x}_\mathrm{b})}{\sigma^2(x)} \tag{9.89}$$

eri blokin koon arvoilla k ja tarkastelemalla sen saturoitumisarvoa.

Alla on esimerkkinä Monte Carlo -simulaatiosta saatu 256 atomin Lennard-Jones-systeemin potentiaalienergia. Ajon pituus oli 370000 askelta. Lämpötila (LJ-yksiköissä) $T^* = 2.0$, jossa systeemi on nestemäinen. Dataa on verrattu satunnaislukugeneraattorilla tuotettuun.



Kuva 9.9: Monte Carlo -simulaatiosta saatu Lennard-Jones -systeemin potentiaalienergia simulointiaskeleen funktiona (ylempi käyrä). 256 atomista koostuvan systeemin lämpötila oli $T^* = 2.0$, jossa Lennard-Jones -aine on nestemäinen. Alempi käyrä on tuotettu satunnaislukugeneraattorilla. Lukujen jakauma oli gaussinen ja varianssi sama kuin simulointidatan.

Kuvan 9.9 simulointidatan keskiarvo on

$$\bar{x} = -5.0783$$

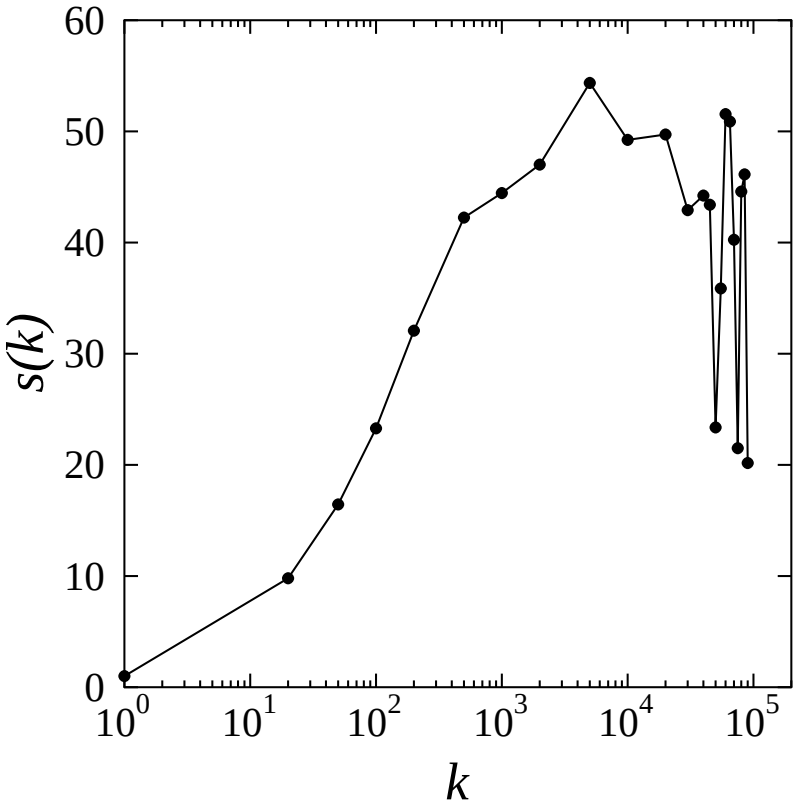
ja standardipoikkeama

$$\sigma(x) = 0.1266 .$$

Olettaen, että datapisteet ovat korreloimattomia saadaan keskiarvon keskivirheeksi:

$$\sigma(\bar{x}) = \frac{0.1266}{\sqrt{370000}} = 2.0820 \times 10^{-4} .$$

Seuraavassa kuvassa on esitetty s:n riippuvuus blokin koosta k.



Kuva 9.10: Statistisen tehottomuuden riippuvuus blokin koosta.

Suure $s(k)$ saturoituu arvoon $s \approx 38$. Tämä tarkoittaa sitä, että ainoastaan joka 38. datapisteet ovat toisistaan riippumattomia. Tästä saamme arvioksi keskiarvon keskivirheelle

$$\sigma(\bar{x}) = \frac{0.1266}{\sqrt{370000}}\sqrt{38} = 1.2834\times10^{-3} \ .$$

Eli lopullinen tulos on

$$\bar{x} = -5.0783 \pm 0.0013 \ .$$

10 Datan mallinnus

10.1 Yleistä

Karkeasti ottaen datan mallinnuksessa meillä on malli, joka kertoo, miten data käyttäytyy. Malliin liittyy parametreja, joiden arvoja ei välttämättä tunneta. Datan mallinnuksella tai sovituksella pyritään saamaan selville nuo parametrit ja arvio niiden luotettavuudelle.

Malli voi olla peräisin jostain teoriasta, jolloin sovituksesta saatavilla parametreilla voi olla jokin (fysikaalinen) merkitys. Toisaalta tarkoituksena voi olla kohinan suodatus pois datasta, jolloin parametreilla ei ole mitään erityistä merkitystä.

Peruslähestymistapa on seuraava:

Valitaan ns. ansiofunktio, joka kertoo datan ja mallin tietyillä parametrien arvoilla antaman tulosten välisen eron. Yleensä mitä pienempi ansiofunktion arvo on sitä parempi on datan ja mallin vastaavuus. Varioimalla parametrien arvoja pyritään ansiofunktio minimoimaan. Kyseessä on siis moniulotteinen minimointitehtävä.

Lisäksi on pyritään arvioimaan sovituksen hyvyyttä sekä parametrien epätarkkuutta.

Olkoot meillä N kappaletta datapisteitä (x_i, y_i), $i = 1, 2, \ldots, N$. Malli taas on M-parametrinen

$$y(x) = y(x;a_1, a_2, \ldots, a_M) \ . \tag{10.1}$$

Kysymys kuuluu siis: Jos meillä on tietty parametrijoukko $(a_1, a_2, \ldots, a_M)$, mikä on todennäköisyys, että juuri kyseessäoleva datajoukko syntyy? (On tietysti oletettava jokin vaihteluväli Δy, jolle pisteet osuvat.)

Tämän todennäköisyyden voidaan arvioida olevan

$$P \propto \prod_{i=1}^{N} \left\{ \exp\left[-\frac{1}{2}\left(\frac{y_i - y(x_i)}{\sigma_i} \right)^2 \right] \Delta y \right\} , \tag{10.2}$$

missä datapisteiden oletetaan noudattavan gaussista jakaumaa keskiarvonsa ympärillä standardipoikkeaman ollessa σ_i. Tämän maksimointi on ekvivalenttia todennäköisyyden logaritmin vastaluvun minimoinnin kanssa:

$$\sum_{i=1}^{N} \left(\frac{y_i - y(x_i)}{\sigma_i} \right)^2 - N \ln \Delta y \ . \tag{10.3}$$

Koska N ja Δy ovat vakioita on tehtävään suureen

$$\chi^2 = \sum_{i=1}^{N} \left(\frac{y_i - y(x_i;a_1, a_2, \ldots, a_M)}{\sigma_i} \right)^2 \tag{10.4}$$

minimointi varioimalla parametreja a_j. Voidaan osoittaa, että (10.4):n mukainen χ^2 noudattaa likimain kappaleessa 9.3.2 esitettyä χ^2-jakaumaa, kun vapausasteiden lukumäärä on $\nu = N - M$. Tämän perusteella voimme arvioida sovituksen hyvyyttä laskemalla todennäköisyyden saadulle parametrijoukolle.

Kuten aikaisemmin olemme todenneet antaa

$$Q(\chi^2|\nu) = \texttt{gammq}(\frac{\nu}{2}, \frac{\chi^2}{2}) \tag{10.5}$$

todennäköisyyden, että χ^2 ylittää tietyn arvon sattumalta. Jos tämä todennäköisyys on hyvin pieni, erot datan ja mallin välillä eivät ole satunnaisia fluktuaatioita tai datapisteiden virheet on arvioitu liian optimistisiksi. Jos taas todennäköisyys on hyvin lähellä ykköstä voi syynä olla liian suureksi arvioidut virheet σ_i. Nyrkkisääntönä voisi todeta, että kohtuullisen hyvällä sovituksella $\chi^2 \approx \nu$.

10.2 Suorasovitus

Aloitetaan yksinkertaisimmasta mallista eli suorasta:

$$y(x) = y(x;a, b) = a + bx \ . \tag{10.6}$$

Suorasovituksesta käytetään myös nimeä lineaarinen regressio.

Oletaan, että datapisteiden x-arvot tiedetään tarkasti. Ansiofunktio tulee nyt muotoon

$$\chi^2(a, b) = \sum_{i=1}^{N} \left(\frac{y_i - a - bx_i}{\sigma_i} \right)^2 \ . \tag{10.7}$$

Minimissä χ^2:n derivaatat parametrien suhteen häviävät eli

$$\frac{\partial \chi^2}{\partial a} = -2 \sum_{i=1}^{N} \frac{y_i - a - bx_i}{\sigma_i^2} = 0 \tag{10.8}$$

$$\frac{\partial \chi^2}{\partial b} = -2 \sum_{i=1}^{N} \frac{x_i(y_i - a - bx_i)}{\sigma_i^2} = 0 \ . \tag{10.9}$$

Määritellään seuraavat summat

$$S \equiv \sum_{i=1}^{N} \frac{1}{\sigma_i^2}\ , \quad S_x \equiv \sum_{i=1}^{N} \frac{x_i}{\sigma_i^2}\ , \quad S_y \equiv \sum_{i=1}^{N} \frac{y_i}{\sigma_i^2}$$
$$S_{xx} \equiv \sum_{i=1}^{N} \frac{x_i^2}{\sigma_i^2}\ , \quad S_{xy} \equiv \sum_{i=1}^{N} \frac{x_i y_i}{\sigma_i^2}\ . \tag{10.10}$$

Ratkaisu saadaan muotoon

$$\begin{aligned} \Delta &= SS_{xx} - S_x^2 \\ a &= \frac{S_{xx}S_y - S_x S_{xy}}{\Delta}\ . \\ b &= \frac{SS_{xy} - S_x S_y}{\Delta} \end{aligned} \tag{10.11}$$

Jonkinlainen arvio parametrien a ja b tarkkuudelle saadaan virheen kasautumislain avulla:

$$\sigma_a^2 = \sum_{i=1}^{N} \sigma_i^2 \left(\frac{\partial a}{\partial y_i}\right)^2 = \sum_{i=1}^{N} \sigma_i^2 \left(\frac{S_{xx} - S_x x_i}{\sigma_i^2 \Delta}\right)^2 = \frac{S_{xx}}{\Delta} \tag{10.12}$$

$$\sigma_b^2 = \sum_{i=1}^{N} \sigma_i^2 \left(\frac{\partial b}{\partial y_i}\right)^2 = \sum_{i=1}^{N} \sigma_i^2 \left(\frac{S x_i - S_x}{\sigma_i^2 \Delta}\right)^2 = \frac{S}{\Delta}\ . \tag{10.13}$$

(Tässä ei ole otettu huomioon parametrien korrelaatioita.)

Lopuksi sitten lasketaan todennäköisyys $Q(\chi^2|\nu)$.

NR:n rutiini, joka tekee kaikki nämä asiat on nimeltään `fit`.

Tässä on oletettu, että vain datan y-arvoissa voi olla virhettä. Jos myös x-arvoissa on epätarkkuutta, monimutkaistuu asia hieman. Halukkaat voivat käydä läpi NR:n luvun 15.3.

10.3 Yleinen lineaarinen sovitus

Tässä lineaarisuus tarkoittaa mallin lineaarisuutta parametrien a_j suhteen. Tällainen tapaus voidaan lausua muodossa

$$y(x) = \sum_{k=1}^{M} a_k X_k(x)\ , \tag{10.14}$$

missä $X_k(x)$ ovat mielivaltaiset funktiot. Niitä kutsutaan kantafunktioiksi. Huomaa, että niiden ei tarvitse olla lineaarisia x:n suhteen.

Ansiofunktio on nyt

$$\chi^2 = \sum_{i=1}^{N} \left[\frac{y_i - \sum_{k=1}^{M} a_k X_k(x_i)}{\sigma_i} \right]^2 . \tag{10.15}$$

Otamme käyttöön matriisinotaation. $\mathbf{A}$ on $N \times M$-matriisi, jonka alkiot ovat

$$A_{ij} = \frac{X_j(x_i)}{\sigma_i} . \tag{10.16}$$

Yleensä $\mathbf{A}$:ssa on enemmän rivejä kuin sarakkeita:

kantafunktiot →

datapisteet ↓

$$\begin{bmatrix} \frac{X_1(x_1)}{\sigma_1} & \frac{X_2(x_1)}{\sigma_1} & \frac{X_3(x_1)}{\sigma_1} & \cdots & \frac{X_M(x_1)}{\sigma_1} \\ \frac{X_1(x_2)}{\sigma_2} & \frac{X_2(x_2)}{\sigma_2} & \frac{X_3(x_2)}{\sigma_2} & \cdots & \frac{X_M(x_2)}{\sigma_2} \\ \frac{X_1(x_3)}{\sigma_3} & \frac{X_2(x_3)}{\sigma_3} & \frac{X_3(x_3)}{\sigma_3} & \cdots & \frac{X_M(x_3)}{\sigma_3} \\ \cdots & \cdots & \cdots & \cdots & \cdots \\ \frac{X_1(x_N)}{\sigma_N} & \frac{X_2(x_N)}{\sigma_N} & \frac{X_3(x_N)}{\sigma_N} & \cdots & \frac{X_M(x_N)}{\sigma_N} \end{bmatrix}$$

N-alkioinen vektori $\mathbf{b}$ määritellään

$$b_i = \frac{y_i}{\sigma_i} \tag{10.17}$$

ja parametrit a_i kuvataan M-alkioisella vektorilla $\mathbf{a}$:

$$\mathbf{a} = (a_1, a_2, \ldots, a_M) . \tag{10.18}$$

Asettamalla χ^2:n osittaisderivaatat parametrien suhteen nolliksi saamme yhtälöt

$$\sum_{i=1}^{N} \frac{1}{\sigma_i^2} \left[y_i - \sum_{j=1}^{M} a_j X_j(x_i) \right] X_k(x_i) = 0 , \quad k = 1, \ldots, M . \tag{10.19}$$

Tämä voidaan kirjoittaa matriisimuodossa

$$\sum_{j=1}^{M} \alpha_{kj} a_j = \beta_k \ , \tag{10.20}$$

missä

$$\alpha_{kj} = \sum_{i=1}^{N} \frac{X_j(x_i) X_k(x_i)}{\sigma_i^2} \quad (\text{tai} \quad [\alpha] = \mathbf{A}^T \cdot \mathbf{A}) \tag{10.21}$$

on $M \times M$-matriisi ja

$$\beta_k = \sum_{i=1}^{N} \frac{y_i X_k(x_i)}{\sigma_i^2} \quad (\text{tai } [\beta] = \mathbf{A}^T \cdot \mathbf{b}) \tag{10.22}$$

on M-alkioinen vektori.

Eli matriismuodossa ns. normaaliyhtälöt ovat

$$[\alpha] \cdot \mathbf{a} = [\beta] \tag{10.23}$$

tai

$$(\mathbf{A}^T \cdot \mathbf{A}) \cdot \mathbf{a} = \mathbf{A}^T \cdot \mathbf{b} \ . \tag{10.24}$$

Voidaan osoittaa, että parametrien virheille saadaan arviot

$$\sigma^2(a_j) = C_{jj} \ , \tag{10.25}$$

missä matriisi $\mathbf{C}$ on $[\alpha]$:n käänteismatriisi:

$$C_{jk} = [\alpha]^{-1}_{jk} \ . \tag{10.26}$$

Matriisiyhtälö (10.23) voidaan ratkaista lineaarialgebran standardimenetelmillä (Gauss-Jordan tai LU-hajotelma). NR:n rutiini `lfit` tekee sovituksen käyttäen Gauss-Jordan-menetelmää.

Usein normaaliyhtälöiden ratkaisu on altis numeerisille virheille. Tällöin käyttökelpoinen menetelmä on singulaariarvohajotelman käyttö. Emme kuitenkaan käy sitä tässä läpi. Halukkaat voivat tutustua NR:n lukuihin 2.6 ja 15.4.

10.4 Epälineaarisen mallin sovitus

10.4.1 Yleistä

Epälineaarisessa tapauksessa on turvauduttava iteratiivisiin menetelmiin. Annamme alkuarvot parametreille $a_1, a_2, \ldots, a_M$ ja jollain algoritmilla parannamme ratkaisua, kunnes χ^2:n arvo ei enää pienene.

Tehtävä on lähellä epälineaarista usean muuttujan funktion minimointia (ks. kappale 7.3). Eli tarpeeksi lähellä minimiään χ^2 voidaan approksimoida kvadraattiseksi

$$\chi^2(\mathbf{a}) \approx \gamma - \mathbf{d} \cdot \mathbf{a} + \frac{1}{2}(\mathbf{a} \cdot \mathbf{D} \cdot \mathbf{a}) \ , \tag{10.27}$$

missä $\mathbf{d}$ on M-alkioinen vektori ja $\mathbf{D}$ on $M \times M$-matriisi. Tämän minimi löytyy yhdellä askeleella

$$\mathbf{a}^{(i+1)} = \mathbf{a}^{(i)} + \mathbf{D}^{-1} \cdot [-\nabla\chi^2(\mathbf{a}^{(i)})] \ . \tag{10.28}$$

Jos taas (10.27) on huono approksimaatio (olemme kaukana minimistä) voimme käyttää jyrkimmän suunnan menetelmää:

$$\mathbf{a}^{(i+1)} = \mathbf{a}^{(i)} - \mu \times \nabla\chi^2(\mathbf{a}^{(i)}) \ , \tag{10.29}$$

missä μ on jokin vakio. Jotta pystymme käyttämään näitä algoritmeja on pystyttävä laskemaan χ^2:n gradientti $\mathbf{a}$:n suhteen sekä Hessen matriisi $\mathbf{D}$, joka sisältää χ^2:n toisia derivaattoja $\mathbf{a}$:n suhteen. Nämä ovat nyt tiedossa, koska olemme itse spesifioineet mallin

$$y = y(x;\mathbf{a}) \, . \tag{10.30}$$

Ansiofunktio on siis

$$\chi^2(\mathbf{a}) = \sum_{i=1}^{N} \left[\frac{y_i - y(x_i;\mathbf{a})}{\sigma_i}\right]^2 \ . \tag{10.31}$$

Gradientin $\nabla\chi^2(\mathbf{a})$ komponentit ovat

$$\frac{\partial\chi^2}{\partial a_k} = -2 \sum_{i=1}^{N} \left[\frac{y_i - y(x_i;\mathbf{a})}{\sigma_i^2}\right]\left[\frac{\partial y(x_i;\mathbf{a})}{\partial a_k}\right] , \ k = 1, 2, \ldots, M \ . \tag{10.32}$$

Hessen matriisin alkio D_{kl} taas on

$$\frac{\partial^2 \chi^2}{\partial a_k \partial a_l} \qquad . \tag{10.33}$$
$$= 2 \sum_{i=1}^{N} \frac{1}{\sigma_i^2} \left[\left(\frac{\partial y(x_i;\mathbf{a})}{\partial a_k} \right) \left(\frac{\partial y(x_i;\mathbf{a})}{\partial a_l} \right) - [y_i - y(x_i;\mathbf{a})] \left(\frac{\partial^2 y(x_i;\mathbf{a})}{\partial a_k \partial a_l} \right) \right]$$

Hankkiudutaan eroon noista kakkosista eli määritellään

$$\beta_k = -\frac{1}{2} \frac{\partial \chi^2}{\partial a_k} \ , \tag{10.34}$$

$$\alpha_{kl} = \frac{1}{2} \frac{\partial^2 \chi^2}{\partial a_k \partial a_l} \ . \tag{10.35}$$

Eli matriisi $[\alpha]$ on puolet Hessen matriisista: $[\alpha] = \frac{1}{2}\mathbf{D}$.

Nyt yhtälö (10.28) voidaan kirjoittaa muotoon

$$\sum_{l=1}^{M} \alpha_{kl} \delta a_l = \beta_k \ , \tag{10.36}$$

missä

$$\delta a_l = a_l^{(i+1)} - a_l^{(i)} \tag{10.37}$$

on iteraatiokierroksella tehtävä siirros.

Yhtälöryhmästä (10.36) ratkaistaan siirros $\delta\mathbf{a}$ ja se lisätään nykyiseen vektoriin ja näin saadaan uusi piste.

Jyrkimmän suunnan menetelmä taas tulee muotoon

$$\delta a_l = \mu \beta_l \ . \tag{10.38}$$

Hessen matriisi (10.33) sisältää χ^2:n toisia derivaattoja $\mathbf{a}$:n suhteen. Useimmiten nämä termit unohdetaan. Voimme perustella sitä sillä, että $\partial^2 y / \partial a_k \partial a_l$:n kerroin (10.33):ssa on $[y_i - y(x_i)]$, joka itseasiassa (onnistuneelle mallille) on satunnaisesti jakautunut datapisteiden virhe. Näinollen nämä termit enemmän tai vähemmän kumoavat toisensa. Jos datassa on jotain patologisia pisteitä, jotka ovat kaukana muista (outliers) voi tuon toisen derivaatan sisältämän termin mukaanottaminen tehdä tehtävästä epästabiilimman. Eli uudelleenmäärittelemme matriisin $[\alpha]$:

$$\alpha_{kl} = 2\sum_{i=1}^{N}\frac{1}{\sigma_i^2}\left(\frac{\partial y(x_i;\mathbf{a})}{\partial a_k}\right)\left(\frac{\partial y(x_i;\mathbf{a})}{\partial a_l}\right) . \tag{10.39}$$

Matriisin $[\alpha]$ modifiointi ei muuta lopullista ratkaisua, ainoastaan reitti $\mathbf{a}$-avaruudessa alkupisteestä minimiin muuttuu (toivon mukaan lyhyemmäksi).

10.4.2 Levenbergin ja Marquardtin menetelmä

Levenbergin ja Marquardtin (LM) menetelmässä siirrytään pehmeästi alunperin käänteisen Hessen matriisin menetelmästä jyrkimmän suunnan menetelmään, kun lähestytään χ^2:n minimiä.

Lisäksi menetelmässä arvioidaan vakion μ suuruutta yhtälössä (10.38) Hessen matriisin avulla. Vakion μ dimensio on a_k^2. Ainoa alkio Hessen matriisissa $[\alpha]$, jolla on tämä dimensio on $1/\alpha_{kk}$. Oletetaan, että tämä alkio kertoo tehtävän 'mittakaavan'. Ja jotta emme tekisi liian pitkiä hyppyjä jaetaan se jollain luvulla $\lambda \gg 1$. Yhtälö (10.38) korvataan siis yhtälöllä

$$\delta a_l = \frac{1}{\lambda\alpha_{ll}}\beta_l . \tag{10.40}$$

Käänteisen Hessen matriisin (10.36) ja jyrkimmän suunnan (10.40) menetelmien yhdistäminen käy seuraavasti. Otetaan käyttöön matriisi $[\alpha']$:

$$\begin{aligned} \alpha'_{jj} &= \alpha_{jj}(1+\lambda) \\ \alpha'_{jk} &= \alpha_{jk} \qquad (j \neq k) \end{aligned} , \tag{10.41}$$

ja korvataan yhtälöt (10.36) ja (10.40) yhdellä yhtälöllä

$$\sum_{l=1}^{M}\alpha'_{kl}\delta a_l = \beta_k . \tag{10.42}$$

Jos λ on hyvin suuri, yhtälö (10.42) lähestyy yhtälöä (10.40) ja jos taas λ on hyvin pieni saadaan Hessen matriisin menetelmä (10.36).

LM-menetelmän algoritmi on seuraavanlainen (parametrien alkuarvo on $\mathbf{a}$):

i. Laske $\chi^2(\mathbf{a})$.

ii. Aseta λ:lle maltillinen alkuarvo esimerkiksi $\lambda = 0.001$.

iii. Ratkaise yhtälöstä (10.42) $\delta\mathbf{a}$ ja laske $\chi^2(\mathbf{a}+\delta\mathbf{a})$.

iv. Jos $\chi^2(\mathbf{a}+\delta\mathbf{a}) \geq \chi^2(\mathbf{a})$, kasvata vakiota $\lambda \leftarrow 10\cdot\lambda$ ja mene askeleeseen iii.

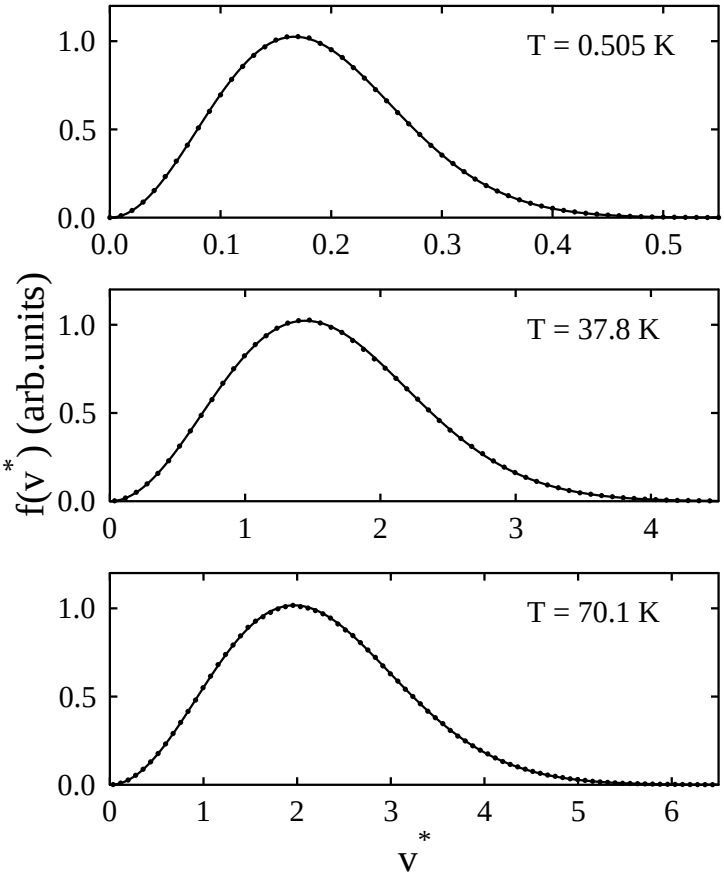
v. Jos $\chi^2(\mathbf{a} + \delta\mathbf{a}) < \chi^2(\mathbf{a})$, pienennä vakiota $\lambda \leftarrow \lambda / 10$, päivitä vektoria $\mathbf{a} \leftarrow \mathbf{a} + \delta\mathbf{a}$ ja mene askeleeseen iii.

Lisäksi tietysti tarvitaan jokin lopetuskriteeri. Käytännössä toimivaksi on osoittautunut ehto, että lopetetaan, kun χ^2 ensimmäisen tai toisen kerran vähenee vain hieman. Tuo hieman voi olla esimerkiksi absoluuttinen (0.01) tai suhteellinen arvo (0.001).

Käytännön esimerkkinä olkoon Maxwell-Boltzmann -jakauman

$$y = ax^2 e^{-x^2/(2b)} \tag{10.43}$$

sovitus molekyylidynamiikkasimulaatioista saatuihin Lennard-Jones-atomien nopeusjakaumiin.



Kuva 10.1: Maxwell-Boltzmann-jakauman sovitus molekyylidynamiikkasimulaatioista saatuihin Lennard-Jones-atomien nopeusjakaumiin. Simulaatio tehtiin neonin potentiaaliparametreilla. Nopeus SI-yksiköissä saadaan redusoidusta nopeudesta v^* kertomalla tämä vakiolla 122.2 m/s.

NR:n rutiini `mrqmin` tekee yhden LM-iteraatiokierroksen. Kutsu on seuraavannäköinen

```
void mrqmin(float x[], float y[], float sig[], int ndata, float a[],
            int ia[], int ma, float **covar, float **alpha, float *chisq,
            void (*funcs)(float, float [], float *, float [], int),
             float *alamda)
```

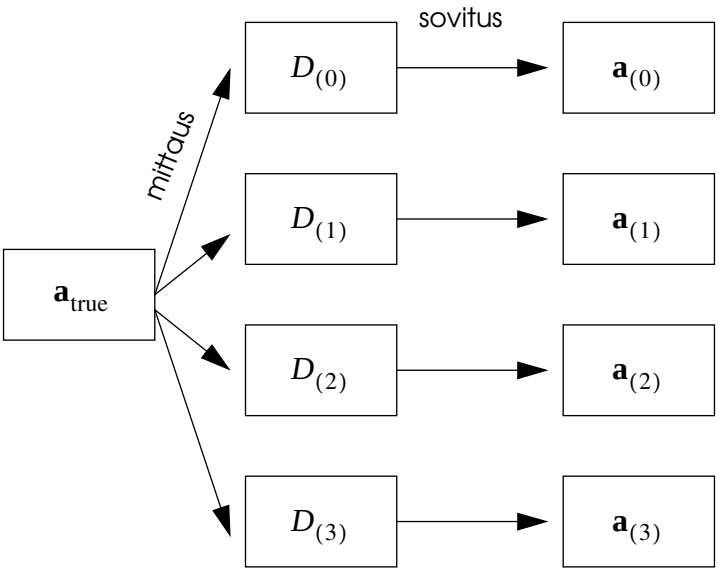
Datajoukko (`ndata` kappaletta pisteitä) välitetään taulukoissa *x* ja `y`. Parametrivektori **a** on taulukossa `a`. Taulukon `ia` alkio kertoo varioidaanko vastaavaa parametria taulukossa `a`, vai pidetäänkö se alkuperäisessä arvossaan. Lisäksi välitetään funktio `funcs(x,a,yfit,dyda,ma)`, joka laskee mallifunktion arvon ja derivaatat parametrien suhteen. Ulostuloparametri `chisq` kertoo χ^2:n arvon iteraation jälkeen.

Ennen ensimmäistä kutsua parametri `alamda` asetetaan negatiiviseksi. Tämän jälkeen rutiinia kutsutaan, kunnes parametrin `chisq` arvon perusteella päätellään iteraation supenneen. Tämän jälkeen kutsutaan rutiinia `alamda`n arvolla nolla, jolloin saadaan ulos kovarianssimatriisi `covar` ja matriisi `alpha`.

10.5 Parametrien virhearvio

Tässä kappaleessa käsittelemme hieman sovituksesta saatavien parametrien virheitä.

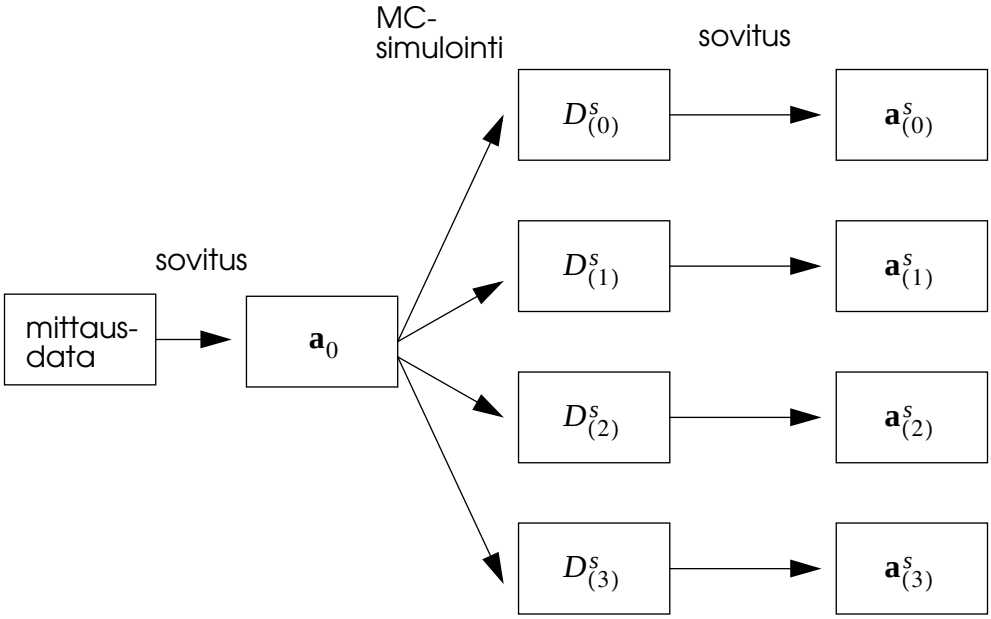
Oheisen kuvan mukaan voimme ajatella, että on olemassa jokin todellinen mallin parametrivektori $\mathbf{a}_{\text{true}}$. Tähän perustuen voidaan realisoida useita datajoukkoja, jotka noudattavat mallia, mutta joissa on satunnaisia virheitä.



Kuva 10.2: Mallin $\mathbf{a}_{\text{true}}$ realisoituneet datajoukot.

Tästä seuraa, että myös sovitusparametrit $\mathbf{a}_{(i)}$ ovat erilaisia. Periaatteessa haluamme tietää parametrien todennäköisyysjakauman.

Eräs moderni menetelmä on synteettisten datajoukkojen generointi Monte Carlo -menetelmällä. Menetelmän periaate on esitetty seuraavassa kuvassa.



Kuva 10.3: Sovitusparametrien virhearvio Monte Carlo -menetelmällä.

Menetelmä edellyttää, että tunnemme datapisteisiin syntyvien satunnaisvirheiden jakauman. Jos on kyseessä kokeellinen data, nuo virheet syntyvät mittalaitteissa, ja ne ovat usein tunnettuja.

Generoimalla tarpeeksi monta datajoukkoa, saamme joukon parametrivektoreita, joista voimme arvioida parametrien varianssia ym. Voimme myös tutkia, minkälaista jakaumaa parametrit noudattavat. Tämä voi tietysti vaatia melko suuria laskentaresursseja.

Parametrien virhearviosta ja luottamusvälien arvioinnista muilla menetelmillä on asiaa NR:n luvussa 15.6.

11 Fourier-muunnos

11.1 Yleistä

Hyvin monissa laskennallisen tieteen ongelmissa tarvitaan datan Fourier-muunnosta. Esimerkiksi joidenkin differentiaaliyhtälöiden ratkaisu voi olla yksinkertaisempaa Fourier-muunnoksen avulla. Toisaalta Fourier-muunnosta voidaan tarvita sellaisenaan esimerkiksi datan tehospektrin arvioimiseen.

Funktion $h(t)$ Fourier -muunnos $H(f)$ määritellään seuraavasti

$$H(f) = \int_{-\infty}^{\infty} h(t)e^{2\pi ift}dt \tag{11.1}$$

Vastaavasti käänteismuunnoksella päästään takaisin funktioon $h(t)$

$$h(t) = \int_{-\infty}^{\infty} H(f)e^{-2\pi ift}df \ . \tag{11.2}$$

Ylläolevat merkinnät antavat ymmärtää, että usein alkuperäinen data on jokin suure ajan funktiona ja muunnettu data taajuuden funktiona. Alkuperäinen data voi olla myös esimerkiksi paikan funktio, jolloin muunnettu data on aallonpituuden käänteisluvun funktion (ns. aaltoluku).

Joskus käytetään taajuuden f sijasta ns. kulmataajuutta ω. Tällöin muunnokset tulevat muotoon

$$\begin{aligned} H(\omega) &= \int_{-\infty}^{\infty} h(t)e^{i\omega t}dt \\ h(t) &= \frac{1}{2\pi}\int_{-\infty}^{\infty} H(\omega)e^{-i\omega t}d\omega \end{aligned} \ . \tag{11.3}$$

Funktion $h(t)$ symmetria- ja reaalisuusominaisuudet heijastuvat myös sen muunnokseen $H(f)$. Näitä ominaisuuksia voidaan käyttää hyväksi laskettaessa muunnoksia. Alla on taulukko muutamista ominaisuuksista:

i. $h(t)$ reaalinen $\Rightarrow$	$H(-f) = [H(f)]^*$
ii. $h(t)$ imaginaarinen $\Rightarrow$	$H(-f) = -[H(f)]^*$
iii. $h(t)$ parillinen $\Rightarrow$	$H(f)$ parillinen
iv. $h(t)$ pariton $\Rightarrow$	$H(f)$ pariton
v. $h(t)$ reaalinen, parillinen $\Rightarrow$	$H(f)$ reaalinen, parillinen
vi. $h(t)$ reaalinen, pariton $\Rightarrow$	$H(f)$ imag., pariton

vii. $h(t)$ imaginaarinen, parillinen $\Rightarrow$ $H(f)$ imagin., parillinen

viii. $h(t)$ imaginaarinen, pariton $\Rightarrow$ $H(f)$ reaalinen, pariton

Lisäksi Fourier-muunnoksella on joitain käteviä ominaisuuksia. Seuraavassa muutamia niistä[1]:

$$h(at) \Leftrightarrow \frac{1}{|a|} H(\frac{f}{a}) \tag{11.4}$$

$$\frac{1}{|b|} h(\frac{t}{b}) \Leftrightarrow H(bf) \tag{11.5}$$

$$h(t - t_0) \Leftrightarrow H(f) e^{2\pi i f t_0} \tag{11.6}$$

$$h(t) e^{-2\pi i f_0 t} \Leftrightarrow H(f - f_0) \tag{11.7}$$

Kahden funktion $g(t)$ ja $h(t)$ konvoluutio $g*h$ määritellään

$$g*h \; = \; \int_{-\infty}^{\infty} g(\tau) h(t - \tau) d\tau \; . \tag{11.8}$$

Voidaan helposti osoittaa, että tämän Fourier-muunnos on funktioiden Fourier-muunnosten $G(f)$ ja $H(f)$ tulo:

$$g*h \Leftrightarrow G(f) H(f) \; . \tag{11.9}$$

Korrelaatiofunktio taas määritellään

$$\mathrm{Corr}(g, h) \; = \; \int_{-\infty}^{\infty} g(\tau + t) h(\tau) d\tau \; . \tag{11.10}$$

Voidaan taas helposti osoittaa, että pätee seuraava

$$\mathrm{Corr}(g, h) \Leftrightarrow G(f) H^*(f) \; , \tag{11.11}$$

jos funktiot g ja h ovat reaalisia. Funktion korrelaatio itsenäs kanssa on ns. autokorrelaatio-funktio:

$$\mathrm{Corr}(g, g) \Leftrightarrow |G(f)|^2 \; . \tag{11.12}$$

Parsevalin teoreeman mukaan signaalin kokonaisteho on sama laskimmepa sen aika- tai taa-

1. Merkki $\Leftrightarrow$ tarkoittaa muunnosparia eli: $h(t) \Leftrightarrow H(f)$.

juusavaruudessa:

$$P_{\text{tot}} \equiv \int_{-\infty}^{\infty} |h(t)|^2 dt \;=\; \int_{-\infty}^{\infty} |H(f)|^2 df \;. \tag{11.13}$$

Teho taajuusvälillä $[\,f,\; f + df\,]$ määritellään (useimmiten ei erotella positiivisia ja negatiivisia taajuuksia)

$$P_h(f) \equiv |H(f)|^2 + |H(-f)|^2 \quad (0 \le f < \infty). \tag{11.14}$$

Reaalisen funktion h tapauksessa tämä supistuu muotoon

$$P_h(f) \equiv 2|H(f)|^2 \;. \tag{11.15}$$

11.2 Diskreetti Fourier-muunnos

Useimmiten data ei ole jatkuvan funktion muodossa, vaan meillä on joukko tasaisesti aika-akselille sijoittuneita pisteitä h_n:

$$h_n \;=\; h(\Delta n) \;,\;\; n \;=\; \ldots, -3, -2, -1, 0, 1, 2, 3, \ldots \;. \tag{11.16}$$

Aikavälin Δ käänteislukua kutsutaan näyttenottotaajuudeksi (*sampling rate*). Nyquistin kriittiseksi taajuudeksi kutsutaan taajuutta

$$f_{\text{c}} \equiv \frac{1}{2\Delta} \;. \tag{11.17}$$

Otetaan jatkuvasta funktiosta $h(t)$ näytteitä taajuudella $1/\Delta$. Jos funktion kaistanleveys on rajoitettu siten, että $H(f) \;=\; 0$ kaikille taajuuksille $|f| \ge f_{\text{c}}$, tällöin otetut näytteet h_n määräävät funktion $h(t)$ täydellisesti. Funktio voidaan nyt lausua muodossa

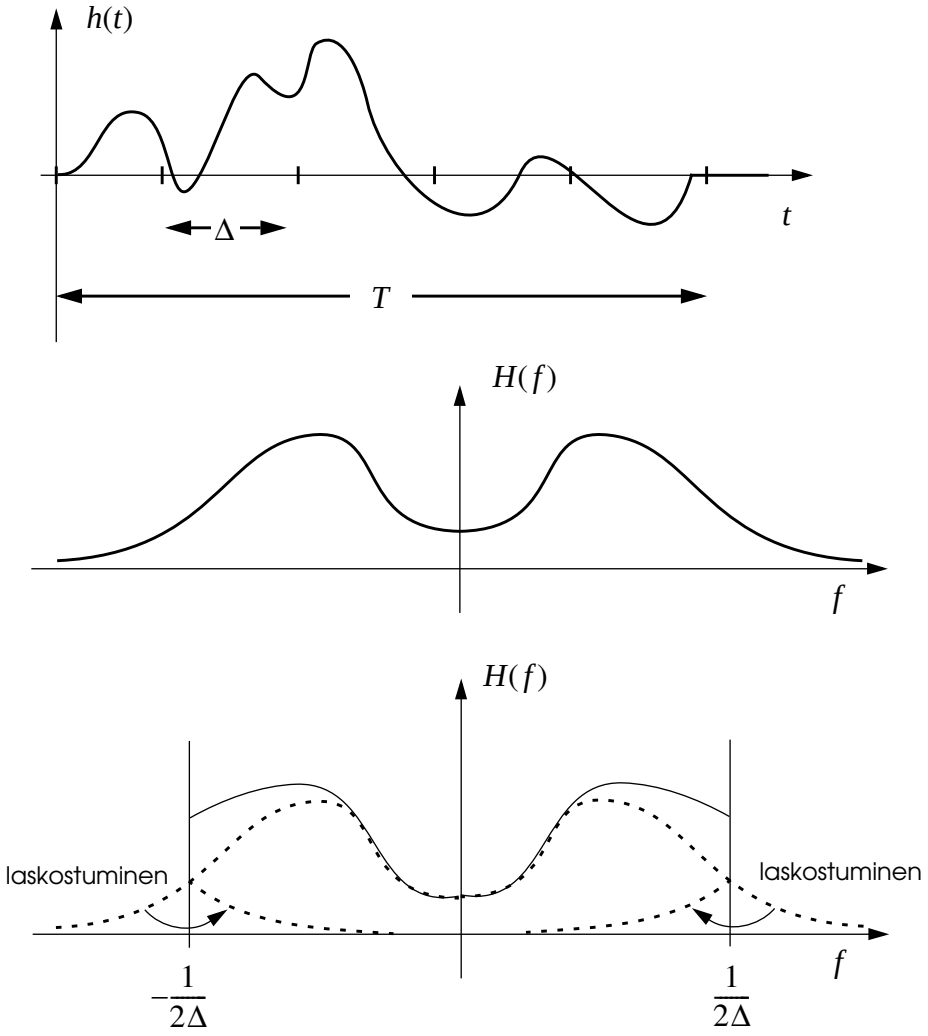
$$h(t) \;=\; \Delta \sum_{n\,=\,-\infty}^{\infty} h_n \frac{\sin[2\pi f_{\text{c}}(t - \Delta n)]}{\pi(t - \Delta n)} \;. \tag{11.18}$$

Usein tiedämme mitatusta datasta, että sen kaistanleveys on rajoitettu (signaali on kulkenut alipäästösuodattimen läpi tms.). Tällöin riittää ottaa singnaalista näytteitä taajuudella, joka on kaksinkertainen signaalista tavattavaan maksimitaajuuteen verrattuna.

Jos taas otamme näytteitä liian alhaisella taajuudella, osoittautuu, että kaikki taajuusalueen $-f_{\text{c}} < f < f_{\text{c}}$ ulkopuolella oleva teho laskostuu (*aliasing*) tälle alueelle.

Esimerkiksi signaaleista $\exp(2\pi i f_1 t)$ ja $\exp(2\pi i f_2 t)$ saadaan samat näytteet, kun taajuuksien ero on $1/\Delta$:n monikerta. Ja $1/\Delta$ on taajuusvälin $[-f_{\text{c}},\, f_{\text{c}}]$ pituus.

Laskostumista on havainnollistettu kuvassa 11.1.



Kuva 11.1: Fourier-muunnoksen laskostuminen.

Diskreetillä Fourier-muunnoksella (DFM) approksimoidaan jatkuvan funktion Fourier-muunnosta siitä otettujen näytteiden perusteella. Olkoon meillä N kappaletta näytteitä:

$$h_k \equiv h(t_k) \ , \ t_k \equiv k\Delta \ , \ k = 0, 1, 2, \ldots, N-1 \ . \tag{11.19}$$

Oletetaan lisäksi, että N on parillinen.

Koska meillä on N kappaletta syöttötietoja Fourier-muunnoksen tekoon, voimme odottaa ainoastaan N kappaletta riippumatonta tulostietoa. Tästä syystä DFM:a ei lasketa kaikkialla alueella $[-f_c, f]$ vaan ainoastaan pisteissä

$$f_n \equiv \frac{n}{N\Delta} \ , \ n = -\frac{N}{2}, \ldots, \frac{N}{2} \ . \tag{11.20}$$

Rajat $f_{-N/2}$ ja $f_{N/2}$ vastaavat Nyquistin kriittistä taajuutta (11.17).

DFM lasketaan approksimoimalla (11.1):n integraalia summalla

$$H(f_n) = \int_{-\infty}^{\infty} h(t)e^{2\pi i f_n t}dt \quad . \tag{11.21}$$
$$\approx \sum_{k=0}^{N-1} h_k e^{2\pi i f_n t_k}\Delta = \Delta \sum_{k=0}^{N-1} h_k e^{2\pi i k n/N}$$

Ylläolevan yhtälön viimeistä summaa kutsutaan varsinaiseksi DFM:ksi. Eli

$$H_n \equiv \sum_{k=0}^{N-1} h_k e^{2\pi i k n/N} \quad . \tag{11.22}$$

Eli DFM kuvaa N kompleksilukua (h_k) N:ksi kompleksiluvuksi (H_n). Huomaa, että DFM ei riipu mistään dimensiollisesta parametrista kuten esimerkiksi näytteenottovälistä Δ. Yhteys jatkuvan Fourier-muunoksen ja DFM:n välilä on siis

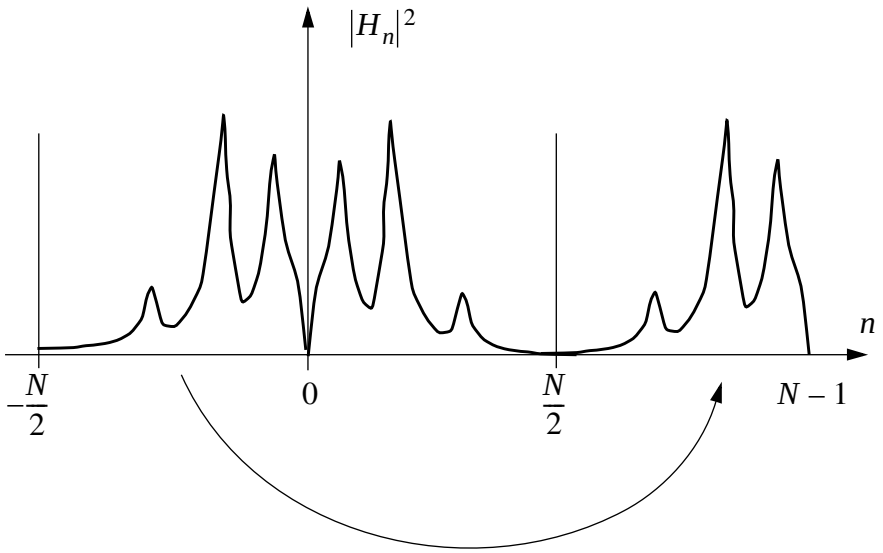
$$H(f_n) \approx \Delta H_n \ , \tag{11.23}$$

missä taajuus f_n saadaan kaavasta (11.20).

Yhtälön (11.20) Mukaan oletimme yllä, että indeksi n on välillä $[-N/2, N/2]$. Tarkasteltaessa yhtälöä (11.22) huomaamme kuitenkin, että H_n on periodinen ja sen jakso on N. Tämä tarkoittaa mm. siät, että

$$H_{-n} = H_{N-n} \ . \tag{11.24}$$

Tästä syystä yleensä n asetetaan käymään läpi arvot $n = 0, 1, 2, \ldots, N-1$ yhtälössä (11.20). Tällöin positiiviset taajuudet löytyvät arvoilla $n = 1, 2, \ldots, N/2 - 1$ ($n = 0$ vastaa 'tasavirtaa') ja negatiiviset arvoilla $n = N/2, \ldots, N-1$. Arvo $n = N/2$ vastaa sekä taajuuksia f_c ja $-f_c$.



Kuva 11.2: Indeksointi diskreetissä Fourier-muunnoksessa.

Ylläesitetyllä tavalla indeksoidussa DFM:lle on samat symmetriaominaisuudet kuin jatkuvallakin (kappale 11.1). Argumentin etumerkin vaihto vastaa DFM:ssä siirtoa N:n verran: $-f \Leftrightarrow N - n$.

Käänteinen DFM hoituu kuten jatkuvakin:

$$h_k = \frac{1}{N} \sum_{n=0}^{N-1} H_n e^{-2\pi i k n / N} \ . \tag{11.25}$$

11.3 Nopea Fourier-muunnos

Diskreetin Fourier-muunnoksen laskemiseen kuluva näyttäisi olevan $O(N^2)$. Tätä voi perustella seuraavasti. Määritellään kompleksiluku

$$W \equiv e^{2\pi i / N} \ . \tag{11.26}$$

Yhtälö (11.22) voidaan kirjoittaa muotoon

$$H_n = \sum_{k=0}^{N-1} h_k W^{nk} \ . \tag{11.27}$$

Eli N-alkioinen vektori h_k kerrotaan $N \times N$ matriisilla, jonka alkiot ovat kompleksiluvun W potensseja. Matriisikertolasku vaatii $O(N^2)$ operaatiota.

60-luvun puolivälissä John Tukey ja John Cooley julkaisivat artikkelin, jossa kuvattiin algoritmi, joka suoriutuu DFM:n laskemisesta ajassa $O(N \log_2 N)$.[1] Algoritmia kutsutaan yleisesti nopeaksi Fourier-muunnokseksi (*Fast Fourier transform*, FFT).

FFT-algoritmin perusteena on se, että N;n näytteen f_j DFM voidaan lausua kahden $N/2$ pituisen jonon DFM:na, joista toinen koostuu parillisten indeksien näytteistä ja toinen parittomien[1]:

$$\begin{aligned} F_k &= \sum_{j=0}^{N-1} f_j e^{2\pi ijk/N} \\ &= \sum_{j=0}^{N/2-1} f_{2j} e^{2\pi i(2j)k/N} + \sum_{j=0}^{N/2-1} f_{2j+1} e^{2\pi i(2j+1)k/N} \\ &= \sum_{j=0}^{N/2-1} f_{2j} e^{2\pi ijk/(N/2)} + W^k \sum_{j=0}^{N/2-1} f_{2j+1} e^{2\pi ijk/(N/2)} \\ &= F_k^{\mathrm{e}} + W^k F_k^{\mathrm{o}} \end{aligned} \tag{11.28}$$

Tässä vakio W on sama kuin yhtälössä (11.26). F_k^{e} on alkuperäisen datajonon parillisista alkioista lasketun DFM:n k:s komponentti. Vastaavasti F_k^{o} on laskettu parittomista alkioista. FFT:n ideana on soveltaa ylläesiteltyä jakoa rekursiivisesti. Eli hajotetaan F_k^{e} kahteen osaan F_k^{ee} ja F_k^{eo}, jotka molemmat ovat $N/4$:n pituisia DFM:a.

Jos oletamme, että N on kahden potenssi, voimme jatkaa jakoa, kunnes tuloksena on yhden mittainen DFM. Sehän ei ole mitään muuta kuin identiteettioperaatio. Eli jokaiselle erilaiselle $\log_2(N)$-pituiselle kombinaatiolle symboleja e ja o löytyy 1-alkioinen DFM:

$$F_k^{\mathrm{eooeoeoeeeeoo...oeoee}} = f_n \ . \tag{11.29}$$

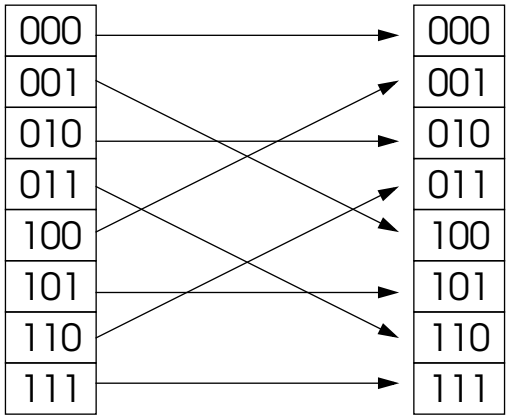
Ongelmana on löytää tuo indeksin n arvo, joka vastaa tiettyä kombinaatiota. Ensimmäisessä jaossa tutkitaan alkion n vähiten merkitsevää bittiä,. Jos se on 0 eli alkio on parillinen, kuuluu alkio termiin F_k^{e}, ja jos se on 1, kuuluu alkio termiin F_k^{o}. Seuraavassa jaossa tutkitaan alkion toiseksi merkitsevintä bittiä ja niin edespäin.
Eli alkion indeksin saamme binäärilukuna, kun käännämme merkkijonon eoo...oee järjestyksen, teemme korvaukset $\mathrm{e} \to 0$ ja $\mathrm{o} \to 1$.

Jos lajittelemme alkuperäisen datajoukon siten, että se on bittikäännetyn indeksin järjestyksessä (ks. kuva), ovat alkiot nyt yksialkioisen DFM:n tuloksia. Yhdistämällä kaksi vierekkäistä alkiota kaavalla (11.28) saadaan kahden pisteen DFM. Näin jatketaan kunnes saadaan N:n pisteen DFM. Jokainen yhdistäminen vie aikaa $O(N)$ ja näitä yhdistämisiä on $O(\log_2 N)$ kappaletta, joten algoritmin ajankäyttö on $O(N \log_2 N)$. Oletamme tietysti, että datan lajittelu bittikäännettyyn järjestykseen ei vie enempää aikaa.

---

1. FFT:n historiasta on mielenkiintoinen luku kirjassa: D. Kahaner, C. Moler, S. Nash: *Numerical Methods and Software*, Prentice-Hall, Lontoo, 1989.

1. f_j ei nyt ole taajuus vaan datajonon näyte j.



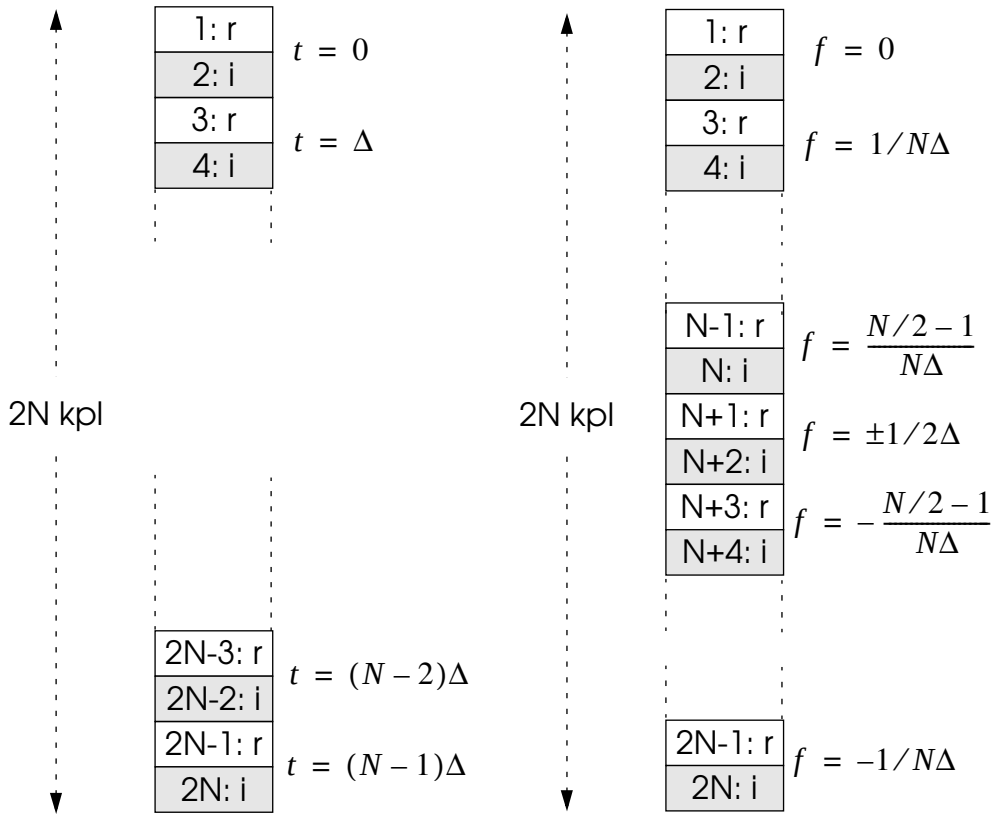
Kuva 11.3: Käännetty bittijärjestys 8 alkion tapauksessa.

FFT-algoritmi koostuu ensinnäkin datan lajittelusta käännettyyn bittijärjestykseen. Itse muunnos tehdään kahdessa silmukassa , joista ulompi käy läpi muunnokset, joiden pituus on 2, 4, 8, jne. Sisemmässä silmukassa taas käydään läpi jo tehdyt osamuunnokset.

NR:n rutiini `four1` tekee FFT:n datajoukolle. Sen kutsu on seuraavanlainen

```
void four1(float data[], unsigned long nn, int isign)
```

Datajoukko, jolle muunnos tehdään on taulukossa `data[1..2*nn]`. Se koostuu `nn` kappaleesta kompeksilukuja. Parametri `isign` kertoo, kumpaan suuntaa muunnos tehdään (-1=käänteismuunnos). Kaavassa (11.25) olevaa tekijää $1/N$ ei tuloksessa ole kuitenkaan mukana. Aliohjelma palauttaa taulukossa `data` muunnoksen tuloksen. Syöttö- ja tulostiedon tallennusformaatti on esitetty kuvassa 11.4. Formaatti on hyvin yleisessä käytössä FFT-rutiineissa.



Kuva 11.4: FFT-rutiinin `four1` datan tallennusformaatti.

Alla pääohjelma, joka käyttää rutiinia `four1`. Se tekee signaalin

$$\cos\left(\frac{2\pi t}{P_1}\right) + \cos\left(\frac{2\pi t}{P_2}\right) + \sigma_n r \tag{11.30}$$

FFT:n ja tarkistuksen vuoksi myös käänteismuunoksen. Tässä P_1 ja P_2 ovat kosinitermien jaksoja ja r on gaussisesti jakautunut satunnaisluku ($\bar{r} = 0$, $\sigma(r) = 1$). Syöttötietoina annetaan jaksot ja gaussisen kohinan standardipoikkema σ_n.

```
#include “nr.h”
#include “nrutil.h”
#define PI 3.1415926

int main(int argc, char **argv)
{
  unsigned long i,np,np2;
  float per1,per2,std,*data;
  long seed;

  if (argc!=6) {
    fprintf(stderr,”Usage: %s np per1 per2 std seed\n”,argv[0]);
    return (1);
  }
  np=atoi(*++argv);
  per1=atof(*++argv);
  per2=atof(*++argv);
```

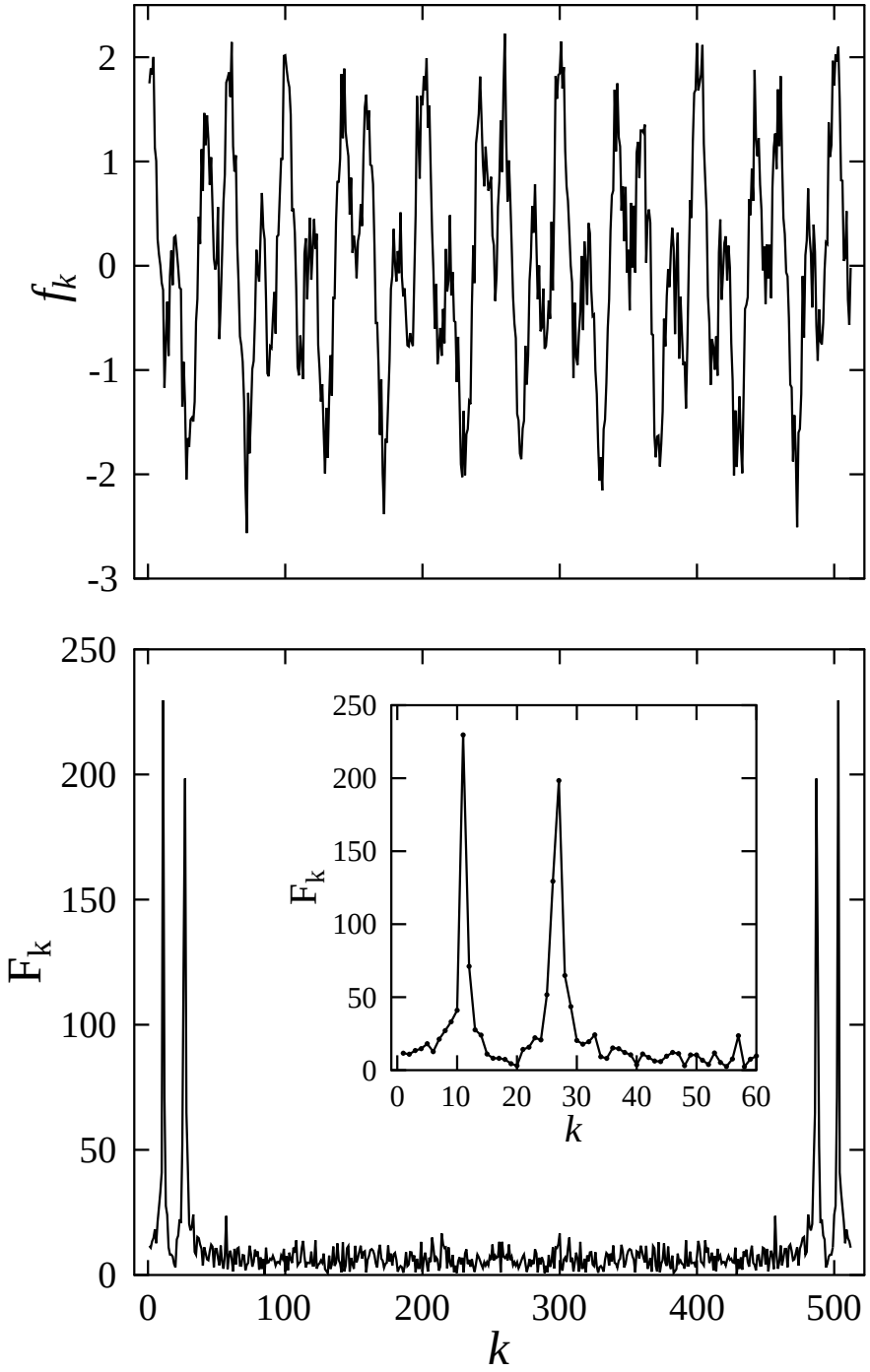
```
  std=atof(*++argv);
  seed=atol(*++argv);
  np2=2*np;

  data=vector(1,np2);

  printf("\n");
  for (i=1;i<=np;i++) {
    data[2*i-1]=cos(2.0*PI*(i-1)/per1)+cos(2.0*PI*(i-1)/per2)+
                  std*gasdev(&seed);
    data[2*i]=0.0;
  }
  for (i=1;i<=np;i++) printf("ORIG: %d %12f %12f\n",i,
                       data[2*i-1],data[2*i]); printf("\n");
  four1(data,np,1);
  for (i=1;i<=np;i++) printf("FFT: %d %12f %12f %12f\n",
         i,sqrt(SQR(data[2*i-1])+SQR(data[2*i])),data[2*i-1],data[2*i]);
  printf("\n");
  four1(data,np,-1);
  for (i=1;i<=np;i++) printf("INV: %d %12f %12f\n",i,
                       data[2*i-1]/np,data[2*i]/np); printf("\n");

}
#undef NRANSI
```

Käännös ja ajo:

```
fft> cc -o xfour1 xfour1.c four1.c gasdev.c ran1.c nrutil.c -lm
xfour1.c:
four1.c:
gasdev.c:
ran1.c:
nrutil.c:
fft>
fft> xfour1 8 3 3 0.0 111

ORIG: 1     2.000000     0.000000
ORIG: 2    -1.000000     0.000000
ORIG: 3    -1.000000     0.000000
ORIG: 4     2.000000     0.000000
ORIG: 5    -1.000000     0.000000
ORIG: 6    -1.000000     0.000000
ORIG: 7     2.000000     0.000000
ORIG: 8    -1.000000     0.000000

FFT: 1     1.000000     1.000000     0.000000
FFT: 2     1.242641     0.878680    -0.878680
FFT: 3     3.000001     0.000000    -3.000001
FFT: 4     7.242640     5.121320     5.121320
FFT: 5     3.000000     3.000000     0.000000
FFT: 6     7.242640     5.121320    -5.121320
FFT: 7     3.000001     0.000000     3.000001
FFT: 8     1.242641     0.878680     0.878680

INV: 1     2.000000     0.000000
INV: 2    -1.000000     0.000000
INV: 3    -1.000000     0.000000
INV: 4     2.000000    -0.000000
INV: 5    -1.000000     0.000000
INV: 6    -1.000000     0.000000
INV: 7     2.000000    -0.000000
INV: 8    -1.000000     0.000000
```

Alla kuvana esimerkkisignaali ja sen FFT. Komento oli

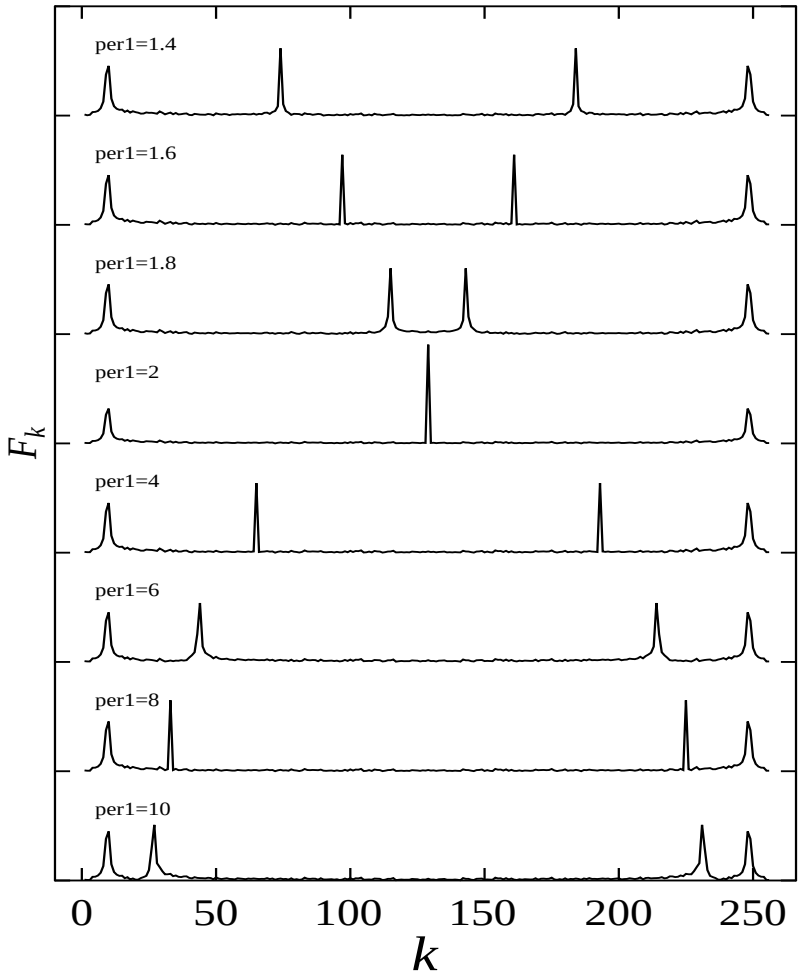
```
xfour1 512 20 50 0.3 987
```



Kuva 11.5: Yhtälön (11.30) mukainen signaali (ylempi kuvaaja) ja sen FFT (alempi kuvaaja). Parametrien arvot olivat $P_1 = 20$, $P_2 = 50$, $\sigma_n = 0.3$, ja pisteiden lukumäärä $N = 512$.

Voimme havainnollistaa laskostumista varioimalla edelläesitetyn ohjelman toisen signaalin jaksoa. Alla sellaisten datajoukkojen FFT:n kuvaajat, joissa on vaihdeltu jaksoa P_1 välillä 1.4-10. Huomaa,että tässä tilanteessa, koska $\Delta = 1$, on Nyquistin kriittinen taajuus

$$f_c = 1/2 \ . \tag{11.31}$$



Kuva 11.6: Yhtälön (11.30) mukaisen signaalin FFT. Toisen kosinitermin jaksoa P_1 on varioitu välillä 1.4-10. Muiden parametrien arvot olivat , $P_2 = 30$, $\sigma_n = 0.1$, ja pisteiden lukumäärä $N = 256$.

Kun jakso P_1 menee alle kahden, havaitaan sitä vastaavan piikin ilmestyminen väärään paikkaan.

Jos muunnettava data on reaalista (yleensä näin on), tehdään yleisessä FFT-rutiinissa turhaa työtä. Datan reaalisuutta voidaan käyttää hyväksi laskentataakan pinentämiseen.

Esimerkiksi NR:n rutiineista löytyvät aliohjelmat `twofft` ja `realft`, jotka laskevat reaalisen datan FFT:n. Rutiini `twofft` laskee yhdellä kertaa kahden reaalidatan FFT:n. `realft` taas laskee yhden reaalidatan muunnoksen käyttäen hyväksi reaalisuutta.

DFM:n ja FFT:n voi helposti yleistää useampiin dimensioihin. Halukkaat voivat lukea asiasta esimerkiksi NR:n luvusta 12.5.

11.4 FFT:n sovellutuksia

Eräs tärkeimpiä FFT:n sovellutuksia on signaalin tehospektrin arviointi. Olkoon meillä signaali $c(t)$. Sen näytteistä c_j tehty DFM on

$$C_k = \sum_{j=0}^{N-1} c_j e^{2\pi i j k/N} \ , \quad k = 0, 1, \ldots, N-1 \ . \tag{11.32}$$

Tehospektrin estimaatti $P(f)$ on määritelty $N/2+1$ pisteessä:

$$\begin{aligned} P(0) &= P(f_0) = \frac{1}{N^2}|C_0|^2 \\ P(f_k) &= \frac{1}{N^2}[|C_k|^2 + |C_{N-k}|^2] \quad , \quad k = 1, 2, \ldots, \frac{N}{2} - 1 \ , \\ P(f_c) &= P(f_{N/2}) = \frac{1}{N^2}|C_{N/2}|^2 \end{aligned} \tag{11.33}$$

missä taajuus f_k on määritelty vain positiivisille taajuuksille

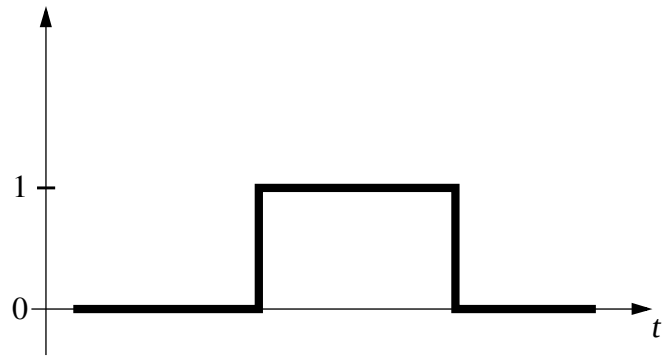
$$f_k \equiv \frac{k}{N\Delta} = 2 f_c \frac{k}{N} \ , \quad k = 0, 1, \ldots, N/2 \ . \tag{11.34}$$

Johtuen datan diskretoinnista ja näytteen äärellisestä koosta ei $P(f_k)$ vastaa täsmälleen ideaalista jatkuvaa tehospektriä samassa pisteessä. Voidaan helposti osoittaa, että $P(f_k)$ on itseasiassa n.s ikkunafunktiolla painotettu keskiarvo tehospektristä pisteen f_k ympäristössä ja tämä ikkunafunktio on

$$W(s) = \frac{1}{N^2}\left[\frac{\sin(\pi s)}{\sin(\pi s/N)}\right]^2 \ . \tag{11.35}$$

Funktiolla on häntä, joka menee nollaan kuten $(\pi s)^{-2}$ suurilla s:n arvoilla. Tämä ei ole kovin nopeaa, joten DFM:n jakovälien väliläl on ns. vuotoa.

Ikkunafunktion (11.35) muoto tulee itseasiassa siitä, että itse data on kerrottu ikkunafunktiolla, joka on allaesitettyä muotoa



Kuva 11.7: Suorakulmainen ikkunafunktio.

Tilannetta voidaan hieman parantaa käyttämällä ‘pehmeämpiä’ ikkunafunktioita. Tämä tarkoittaa sitä, että ennen muunnoksen tekoa data kerrotaan tällä funktiolla. Esimerkkejä näistä funktioista ovat

$$1 - \left|\frac{j - N/2}{N/2}\right| \ , \ \frac{1}{2}\left[1 - \cos\left(\frac{2\pi j}{N}\right)\right] \ , \ \ 1 - \left(\frac{j - N/2}{N/2}\right)^2 \ . \tag{11.36}$$

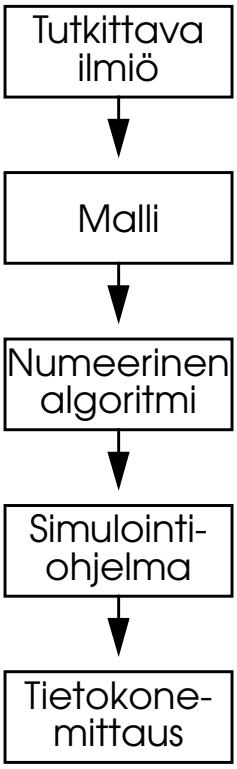
Näihin tutustutaan tarkemmin laskuharjoituksissa.

12 Monte Carlo -simulaatiomenetelmästä

12.1 Yleistä

Termiä Monte Carlo (MC) käytetään hyvinkin erilaisten menetelmien yhteydessä. Karkeasti sanottuna MC-simulaatiosta on kyse, jos menetelmässä käytetään paljon satunnaislukuja. MC-menetelmiä käytetään mm. moniulotteisten integraalien laskemiseen erilaisten (fysikaalisten) systeemien tasapaino-ominaisuuksien laskemiseen[1] ja systeemien ajasta riippuvien ominaisuuksien tutkimiseen. Kurssilla olemme jo kahdesti törmänneet MC-menetelmään: simuloitu jäähdytys (kappale 7.3.6) ja malliparametrien virhearvio MC-menetelmällä (kappale 10.5).

Simulaation suoritus yleensä voidaan jakaa kuvan mukaisiin vaiheisiin.



Kuva 12.1: Simulaation eri vaiheet.

Malli voi olla hyvinkin ilmeinen (esim. kiinteä aine=atomit, jotka vuorovaikuttavat potentiaalin välityksellä) tai sitten ei.

Numeerisen algoritmin avulla pystytään laskemaan mallin antamia tuloksia. Se voi olla esim. algoritmi, jolla kuljetetaan systeemiä ajassa eteenpäin (‘liikeyhtälöt’) tai jolla käydään läpi systeemin faasiavaruutta.

Simulaatioissa ja kokeellisissa mittauksissa on samankaltaisia piirteitä (raakadatan käsittely, statistiset virheet tuloksissa,...), joten usein simulaatioita kutsutaan tietokonemittauksiksi (computer experiments).

Simulaatiomenetelmät voidaan tietyllä tasolla[2] jakaa deterministisiin ja stokastisiin (=MC). Esimerkiksi laskettaessa monen vapausasteen fysikaalisen systeemin tasapaino-ominaisuuksia jako on selkeä.

Useimmiten haluamme laskea jollekin systeemille tietyn suureen A odotusarvon

$$\langle A \rangle = Z^{-1} \int_{\Omega} A(\mathbf{x}) f(H(\mathbf{x})) d\mathbf{x} \quad , \qquad (12.1)$$

$$Z = \int_{\Omega} f(H(\mathbf{x})) d\mathbf{x} \qquad (12.2)$$

1. Tässä on itseasiassa kyse moniulotteisen integraalin laskemisesta.
2. Ainakin fysiikassa.

missä $\mathbf{x}$ on systeemin tilan kertova vektori, $H(\mathbf{x})$ on systeemin kokonaisenergiafunktio (Hamiltonin funktio), Ω on systeemin faasiavaruus ja $f(H(\mathbf{x}))$ on todennäköisyystiheysfunktio.

Tämä on ns. ensemblekeskiarvo, jota ei suoraan voi laskea simuloimalla, koska koko faasiavaruutta ei voida käydä läpi. On tyydyttävä enemmän tai vähemmän edustavaan otokseen.

Deterministinen tapa antaa systeemin oman dynamiikan (liikeyhtälöt) kuljettaa tilavektoria $\mathbf{x}$ halki faasiavaruuden. Ensemblekeskiarvo korvataan aikakeskiarvolla $\overline{A}_t$

$$\overline{A}_t = t^{-1}\int_0^t A(\mathbf{x}(\tau))d\tau \quad . \tag{12.3}$$

Tätä menetelmää kutsutaan fysiikassa molekyylidynamiikaksi (MD).[1]

Stokastisessa eli MC-menetelmässä systeemin tiloja $\mathbf{x}^{(m)}$ generoidaan satunnaisesti. Suureen A odotusarvo on nyt

$$\overline{A} = M^{-1}\sum_{k=1}^{M} A(\mathbf{x}^{(m)}) \quad . \tag{12.4}$$

Ongelmana on kehittää tehokas algoritmi, jolla käydään läpi faasiavaruuden niitä osia, jotka ovat merkittäviä suureen A laskemisen kannalta.

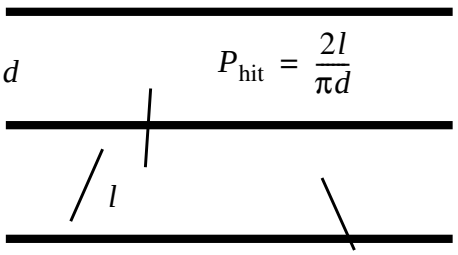
Huomaa, että ylläolevassa esimerkissä ei itse systeemissä (tai sen mallissa) ollut mitään stokastista, satunnaista. Integroitaessa faasiavaruuden yli MC-menetelmässä vain käytettiin satunnaisotantaa. Itse systeemissä voi olla satunnaiselementtejä. Esimerkiksi gammasäteilyn etenemisessä väliaineessa vuorovaikutusten välimatka on satunnainen suure, joka noudattaa tiettyä todennäköisyysjakaumaa.

Satunnaisuus voi johtua myös ‘puutteellisesta’ mallista: jotkut vapausasteet otetaan huomioon stokastisina elementteinä.

1. Helsingin yliopiston fysiikan laitoksella syksyllä 1996 järjestetyn kurssin kotisivulta *http://www.lce.hut.fi/~akuronen/MDkurssi/* löytyy menetelmään liittyvää materiaalia.

12.1.1 Historiaa

Italialainen matemaatikko Lazzerini arvioi piin likiarvoa heittämällä 3407 kertaa neulan tasavälisten suorien päälle (Buffonin neula). Likiarvoksi tuli 3.1415929 eli 7 numeron tarkkuudella oikea (sattumalta?).



Kuva 12.2: Buffonin neula.

W. S. Gossett ('Student') arvioi t-jakaumansa korrelaatiokertoimia otantakokeella.

Lordi Kelvinin assistentti generoi 5000 satunnaista rataa tutkiessaan hiukkasen ja kaarevan seinän välisiä törmäyksiä.

Varsinainen MC-menetelmä[1] kehitettiin Los Alamosissa 50-luvulla. Ensimmäinen julkaisu lienee kaksiulotteisten kovien kiekkojen tilanyhtälöä tutkiva työ.[2].

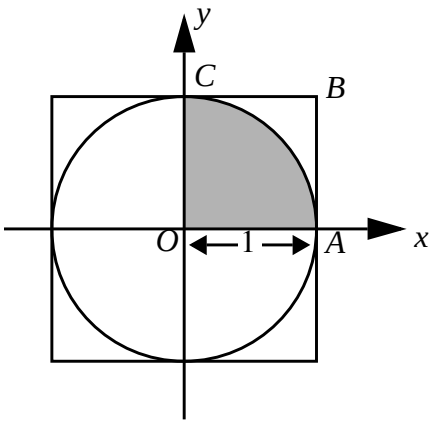
12.2 Monte Carlo -integrointi

Monissa tapauksissa Monte Carlo -simulaatiossa on kyse integraalin laskemisesta. Suoraviivaisin tapa MC-integroinnissa on ottaa painottamaton otos.

Esimerkiksi piin likiarvo saadaan arpomalla N satunnaislukuparia välillä $[0, 1)$ ja laskemalla harmaalle alueelle osuneiden pisteiden lukumäärä N_{hit} :

$$\pi = \frac{4 \times (\text{käyrän AC alla oleva ala})}{(\text{neliön OABC ala})} = \frac{4N_{\text{hit}}}{N} \ . \quad (12.5)$$

Tuloksen tarkkuus luonnollisesti riippuu N:n arvosta ja se käyttäytyy kuten $O(N^{-1/2})$.



Kuva 12.3: Piin askeminen MC-menetelmällä.

1. Ns. Metropolis - Monte Carlo
2. N. Metropolis, A. W. Rosenbluth, M. N. Rosenbluth, A. H. Teller, E. Teller, Equation of State Calculations by Fast Computing Machines, *J. Chem. Phys.*, **21** (1953) 1087.

Kuvassa 12.4 on esitetty piin likiarvon MC-laskun tulokset.

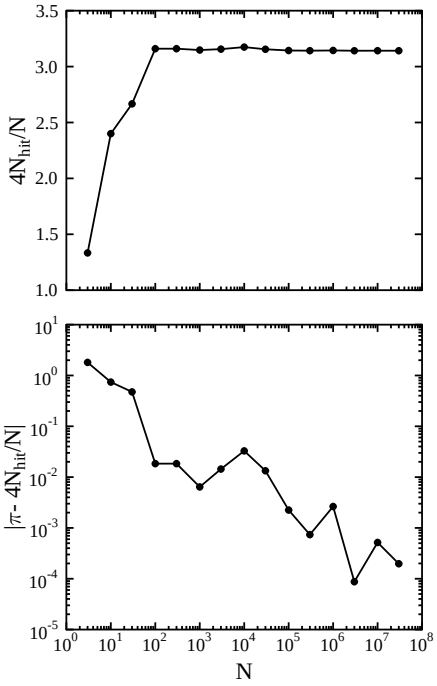
Tasainen otos ei kuitenkaan ole kovin tehokas menetelmä integraalien laskemiseksi. Parempi tulos saadaan ns. otoksen keskiarvo -menetelmällä (*sample mean*).

Olkoon meillä laskettavana integraali

$$F = \int_{x_1}^{x_2} dx\, f(x) \quad . \tag{12.6}$$

Kirjoitetaan se muotoon

$$F = \int_{x_1}^{x_2} dx \left[\frac{f(x)}{\rho(x)}\right] \rho(x) \quad , \tag{12.7}$$



Kuva 12.4: Piin likiarvon MC-laskun tulokset. Ylemmässä kuvassa on esitetty itse piin likiarvo ja alemmassa sen absoluuttinen virhe.

missä $\rho(x)$ on mielivaltainen todennäköisyystiheys [eli sille pätee

$$\rho(x) \geq 0 \qquad \text{ja} \qquad \int_{x_1}^{x_2} dx \rho(x) = 1 \quad]. \tag{12.8}$$

Otetaan N:n suuruinen otos siten, että valitaan satunnaislukuja ζ, joka noudattavat jakaumaa $\rho(x)$ välillä $[x_1, x_2]$.

Tällöin

$$F = \left\langle \frac{f(\zeta)}{\rho(\zeta)} \right\rangle \quad , \tag{12.9}$$

missä keskiarvo otetaan otoksen yli.

Yksinkertaisin valinta todennäköisyystiheydeksi ρ olisi

$$\rho(x) = \frac{1}{x_2 - x_1} \quad , \quad x_1 \leq x \leq x_2 \quad , \tag{12.10}$$

jolloin arvio integraalille on

$$F \approx \frac{x_2 - x_1}{N} \sum_{i=1}^{N} f(\zeta_i) \quad . \tag{12.11}$$

Edellämainittuun esimerkkiin (π:n arvon laskeminen) sovellettuna $f(x) = (1 - x^2)^{-1/2}$ ja $0 \le x \le 1$. Tyypillinen arvio π:n arvolle 10^7 suuruisella otoksella on 3.14169 .

Yksiulotteisessa tapauksessa MC-integrointi häviää selvästi muille (deterministisille) menetelmille esimerkiksi Simpsonin menetelmälle (10^4 funktion arvon laskulla $\pi \approx 3.141593$). Mutta otetaan moniulotteinen esimerkki. Statistisessa fysiikassa on usein laskettava konfiguraatiointegraali

$$Z = \int \rho_e(\mathbf{r}) d\mathbf{r} , \tag{12.12}$$

missä $\rho_e(\mathbf{r})$ on tiheysfunktio, joka kertoo todennäköisyyden, että systeemi on tilassa, jonka spesifioi vektori $\mathbf{r}$. Olkoon meillä 100 atomia kuutiossa $x, y, z \in [-L/2, L/2]$. Karkea Simpsonin menetelmä (10 funktion evaluointia/koordinaatti) vaatisi 10^{300} funktion evaluointia.

Konfiguraatiointegraali voidaan tietysti laskea MC-integroinnilla käyttäen painottamatonta otosta:

i. Arvo satunnaisesti kaikkien 100 atomin paikat ($\mathbf{r}^{(i)}$,).

ii. Laske tiheysfunktio $\rho_e(\mathbf{r}^{(i)})$ -

Integraali on nyt

$$Z = \frac{V^N}{n} \sum_{i=1}^{n} \rho_e(\mathbf{r}^{(i)}) \ . \tag{12.13}$$

Tämä on tehoton menetelmä, koska useimmissa käytännön tapauksissa suurin osa otoksista on sellaisia, joille $\rho_e(\mathbf{r}^{(i)})$ on hyvin pieni[1].

Samat vaikeudet koskevat jonkin suureen A odotusarvojen

$$\langle A \rangle = \frac{\int d\mathbf{r} A(\mathbf{r}) \rho_e(\mathbf{r})}{\int d\mathbf{r} \rho_e(\mathbf{r})} \approx \frac{\sum_{i=1}^{n} A(\mathbf{r}^{(i)}) \rho_e(\mathbf{r}^{(i)})}{\sum_{i=1}^{n} \rho_e(\mathbf{r}^{(i)})} \tag{12.14}$$

1. Esimerkiksi, kun lämpötila, tilavuus ja hiukkasten lukumäärä ovat vakioita, on tiheysfunktio Boltzmannin tekijä $\rho_e(\mathbf{r}^{(i)}) \propto \exp(-V(\mathbf{r}^{(i)})/(k_b T))$, missä $V(\mathbf{r}^{(i)})$ on systeemin potentiaalienergia ja k_B on Boltzmannin vakio. Tämä on ns. kanoninen ensemble.

laskemista.

Ratkaisu ongelmaan on ns. *importance sampling*, jossa otokseen otetaan mukaan enemmän näytteitä niilät alueilta, joissa integroitava funktio on suuri.

MC-integraalin otos painotetaan jollain todennäköisyysjakaumalla ρ, jolloin integraali (integrandi on nyt $f = \rho_e A$)

$$\langle A \rangle = \int d\mathbf{r} A(\mathbf{r}) \rho_e(\mathbf{r}) \tag{12.15}$$

saadaan otoksen keskiarvona (vrt. kaava (12.9))

$$\langle A \rangle = \langle \frac{A\rho_e}{\rho} \rangle_{\text{otos}} . \tag{12.16}$$

Hyvin usein funktio $A(\mathbf{r})$ on merkittävä siellä, missä $\rho_e(\mathbf{r})$:kin.

Tässä tapauksessa hyvä valinta todennäköisyystiheydelle on $\rho = \rho_e$. Tällöin integraali (12.15) tulee yksinkertaiseen muotoon

$$\langle A \rangle = \langle A \rangle_{\text{otos}} . \tag{12.17}$$

Tämä voidaan perustella seuraavasti: Integraalin (12.9) varianssi on

$$\begin{aligned} \sigma^2 &= \langle \frac{f^2}{\rho^2} \rangle - \langle \frac{f}{\rho} \rangle^2 \\ &\approx \int_a^b dx \left[\frac{f(x)}{\rho(x)} \right]^2 \rho(x) - \left\{ \int_a^b dx \left[\frac{f(x)}{\rho(x)} \right] \rho(x) \right\}^2 \end{aligned} \quad . \tag{12.18}$$

Voimme olettaa, että $f(x) > 0$ integrointivälillä. Tällöin varianssi saadaan mahdollisimman pieneksi, kun valitsemme todennäköisyystiheydeksi $\rho(x)$ funktion

$$\rho(x) \approx \frac{f(x)}{\int_a^b dx\, f(x)} \quad . \tag{12.19}$$

Ylläolevassa kaavassa on tietysti nimittäjässä itse integraali, jonka haluamme laskea. Tämän ongelman kiertämiseksi on kaksi tapaa. Ensinnäkin tiheysfunktion ei tarvitse olla täsmälleen tuo optimaalinen. Useimmiten riittää, kun se muistuttaa itse integroitavaa funktiota. Huomaa myös, että ylläolevassa esimerkissä valinta $\rho = \rho_e$ ei ole optimaalinen, koska keskiarvon (12.14) laskemisessa integrandi ei ole pelkkä ρ_e vaan $\rho_e A$.

Toisessa tavassa ei tuota normitusvakiota tarvita ollenkaan. Generoitaessa Markovin ketjulla integrointipisteitä tarvitsemme ainoastaan todennäköisyystiheyksien suhdetta eri pisteissä, ja tällöin normitustekijä (12.19):ssa kumoutuu pois.

Ongelmana on vain kehittää algoritmi, jonka avulla voimme tehdä ρ_e:n mukaisen otoksen.

Tämä onnistuu Markovin prosessin avulla[1]. Olkoon meillä systeemi, joka voi olla jossain tilassa n. Markovin prosessin määrittelevät siirtymätodennäköisyydet π_{mn} systeemin tilojen välillä ja tasapainotodennäköisyysjakauma ρ_n. Eli käytännössä otetaan systeemille jokin alkutila ja arvotaan seuraava tila siirtymätodennäköisyyksien π_{mn} mukaan. Käymällä läpi tiloja tarpeeksi kauan, lähestymme prosessin tasapainojakaumaa ρ_n, eli otokseen mukaan tulevien tilojen paino on verrannollinen jakaumaan ρ_n.

Jotta lähestyisimme tasapainojakaumaa, tulee siirtymätodennäköisyyksien toteuttaa ehdot

$$\pi_{mn} \geq 0 \tag{12.20}$$

$$\sum_n \pi_{mn} = 1 \tag{12.21}$$

$$\rho_m \pi_{mn} = \rho_n \pi_{nm} \quad . \tag{12.22}$$

(Tässä ρ on siis tasapainojakauma.)

Lisäksi prosessin tulisi olla *ergodinen* eli mistä tahansa tilasta tulee olla mahdollista siirtyä mihin tahansa toiseen tilaan. Tämä on hankalampi asia, ja oletamme, että prosessi on yleensä ergodinen.

Esimerkiksi kanonisen ensemblen tapauksessa siis tasapainojakauma on

$$\rho_e = \rho_m = \frac{1}{Z} e^{-V(m)/k_B T} \quad , \quad Z = \int d\mathbf{r}\, e^{-V(\mathbf{r})/k_B T} \, . \tag{12.23}$$

Ehdot (12.20)-(12.22) ovat varsin väljät, joten voimme melko vapaasti valita siirtymätodennäköisyydet π_{mn}. Koska yhtälössä (12.22) esiintyy vain todennäköisyystiheyksien suhde, yksi ehdotus siirtymätodennäköisyyksiksi on

$$\pi_{mn} = \begin{cases} \alpha_{mn}; & \rho_n \geq \rho_m \quad m \neq n \\ \alpha_{mn}\left(\dfrac{\rho_n}{\rho_m}\right); & \rho_n < \rho_m, \quad m \neq n \\ 1 - \displaystyle\sum_{m \neq n} \pi_{mn}, & m = n \end{cases} \quad . \tag{12.24}$$

Tässä α on symmetrinen stokastinen matriisi $\alpha_{mn} = \alpha_{nm}$. Se sisältää tavallaan yritesiirtymätodennäköisyydet. Symmetrisyyteen perustuen voidaan osoittaa, että yhtälön (12.24) siirty-

1. Tästä oli puhetta simuloidun jäähdytyksen yhteydessä kappaleessa 7.3.6.

mätodennäköisyydet toteuttavat ehdot (12.20)-(12.22).

Eräs tärkeä siirtymätodennäköisyyksien π_{mn} piirre on, että siinä esiintyy vain tilojen todennäköisyyksien suhde ρ_n/ρ_m . Tästä syystä emme tarvitse normitettuja tiheyksiä. Tämä on suunnaton helpotus, sillä normitustekijän laskeminen on suunnilleen sama tehtävä kuin koko integraalin laskeminen.

Otetaan käytännön toteutuksesta esimerkkinä fysikaalisen systeemin, jonka lämpötila, tilavuus ja hiukkasten lukumäärä ovat vakioita[1], MC-simulaatio? Systeeminä meillä olkoon N kappaletta atomia tai molekyyliä, jotka vuorovaikuttavat (pari)potentiaalin $V(r)$ välityksellä. Systeemin tilan m määräävät atomien paikat $\{\mathbf{r}_i^m\}$ ($i = 1 \ldots N$).

Keskitymme systeemin potentiaalienergiaan, koska Metropolis-MC ei sano mitään liike-energiasta[2]. Eli systeemin potentiaalienergia

$$V(m) = \sum_{i=1}^{N-1} \sum_{j=i+1}^{N} V(r_{ij}^m) \qquad ; \qquad r_{ij}^m = \left|\mathbf{r}_i^m - \mathbf{r}_j^m\right| \tag{12.25}$$

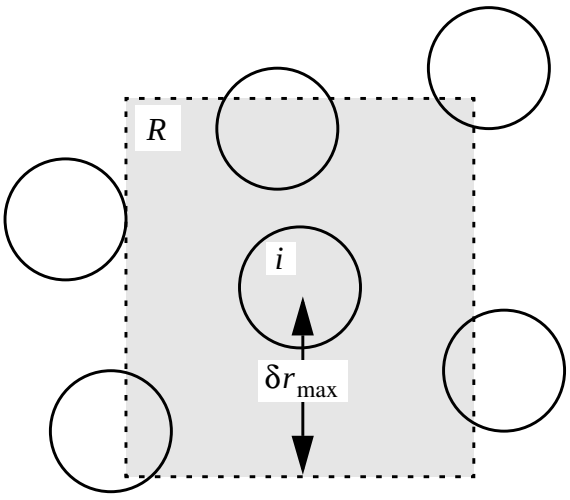
määrää siirtymätodennäköisyydet π_{mn} .

Stokastisen matriisin α avulla pääsemme tilasta m mihin tahansa sen naapuritilaan n yhtäsuurilla todennäköisyyksillä. Ainoa rajoitus oli mikroskooppinen reversiibeliys $\alpha_{mn} = \alpha_{nm}$.

Naapuritila voidaan määritellä kuvan mukaan.

Kuusi atomia ovat tilassa m . Uusi tila n muodostetaan valitsemalla satunnaisesti jokin atomeista (i) ja siirtämällä sitä paikasta $\mathbf{r}_i^m$ satunnaisesti yhtäsuurella todennäköisyydellä mihin tahansa paikkaan $\mathbf{r}_i^n$ neliön R sisällä.

Tietokoneessa on äärellinen määrä N_R paikkoja neliön alueella, joten muodollisesti matriisi α on



Kuva 12.5: Naapuritilat Monte Carlo -simulaatiossa.

1. Eli kyseessä on kanoninen ensemble.
2. Tai itseasiassa liike-energia on sisäänmenoparametri: lämpötila,

$$\alpha_{mn} = \begin{cases} 1/N_R, & \mathbf{r}_i^n \in R \\ 0, & \mathbf{r}_i^n \notin R \end{cases} \quad . \tag{12.26}$$

Maksimisiirtymä $\delta r_{\max}$ kontrolloi Markovin ketjun suppenemista. C-koodina asia näyttää seuraavanlaiselta

```
rxinew = rx[i] + (2.0*ran3(&seed)-1.0)*drmax
ryinew = ry[i] + (2.0*ran3(&seed)-1.0)*drmax
rzinew = rz[i] + (2.0*ran3(&seed)-1.0)*drmax
```

Siirtymätodennäköisyys π_{mn} riippuu tilojen m ja n todennäköisyystiheyksien ρ_n ja ρ_m suhteesta

$$\frac{\rho_n}{\rho_m} = e^{-\delta V_{nm}/k_B T} \quad , \tag{12.27}$$

missä eri tilojen potentiaalienergioiden erotus on

$$\delta V_{nm} = V(n) - V(m) \quad . \tag{12.28}$$

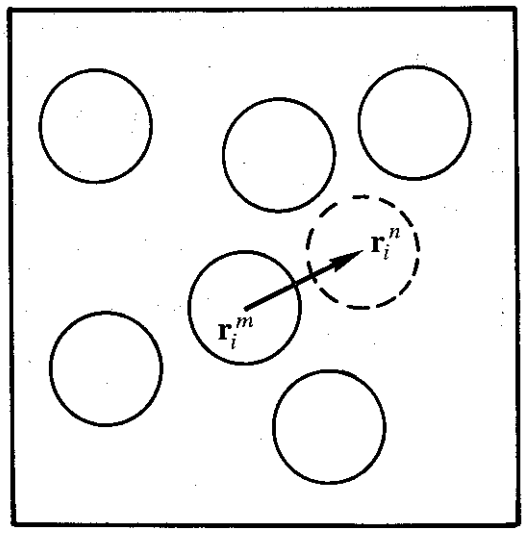
Tämän laskemiseen ei tarvitse käydä kaikkia atomipareja kuten (12.25):ssä vaan riittää laskea atomin i vuorovaikutus muiden atomien kanssa:

$$\delta V_{nm} = \sum_{j=1}^{N} V(r_{ij}^n) - \sum_{j=1}^{N} V(r_{ij}^m) \tag{12.29}$$

Jos ehdotettu siirtymä on alamäkeen energiassa ($\delta V_{nm} \leq 0$), uusi tila hyväksytään. Tämä vastaa ensimmäistä tapausta kaavassa (12.24), eli tila hyväksytään todennäköisyydellä $\pi_{mn} = \alpha_{mn}$.

Jos taas $\delta V_{nm} > 0$, uusi tila hyväksytään todennäköisyydellä ρ_n / ρ_m kaavan (12.24) toisen rivin mukaan. Tekijä α_{nm} on tässäkin tapauksessa mukana. Tiheyksien suhde on tietysti

$$\frac{\rho_n}{\rho_m} = \frac{Z_{NVT}^{-1} e^{-V(n)/k_B T}}{Z_{NVT}^{-1} e^{-V(m)/k_B T}} = \frac{e^{-V(n)/k_B T}}{e^{-V(n)/k_B T} e^{\delta V_{nm}/k_B T}} = e^{-\delta V_{mn}/k_B T} \qquad (12.30)$$



Kuva 12.6: Yritesiirto Monte Carlo -simulaatiossa.

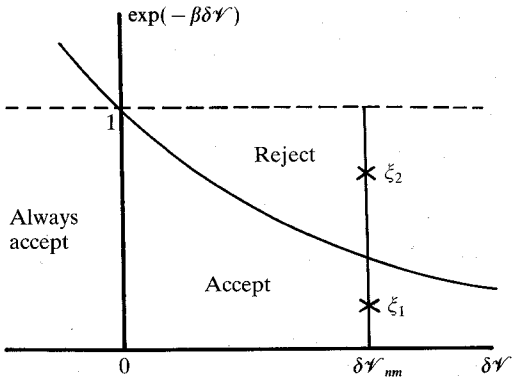
Hyväksyminen todennäköisyydellä $\exp(-\delta V_{mn}/k_B T)$ toteutetaan seuraavasti:

i. Generoidaan tasaisesti välille $[0, 1)$ jakautunut satunnaisluku ξ .

ii. Jos $\xi < \exp(-\delta V_{mn}/k_B T)$, uusi tila hyväksytään.

Jos tämä ylämäkeen tehtävä siirto hylätään, systeemi pysyy vanhassa tilassaan, ja tämä tila lasketaan uudestaan mukaan.
Siis: jokainen ehdotettu tila hyväksytään todennäköisyydellä

$$\min(1, e^{-\delta V_{nm}/k_B T}) \quad . \qquad (12.31)$$



Kuva 12.7: Yritesiirron hyväksyminen MC-simulaatiossa.

Alla on esitetty tyypillisen Metropolis-MC-koodin ydin.

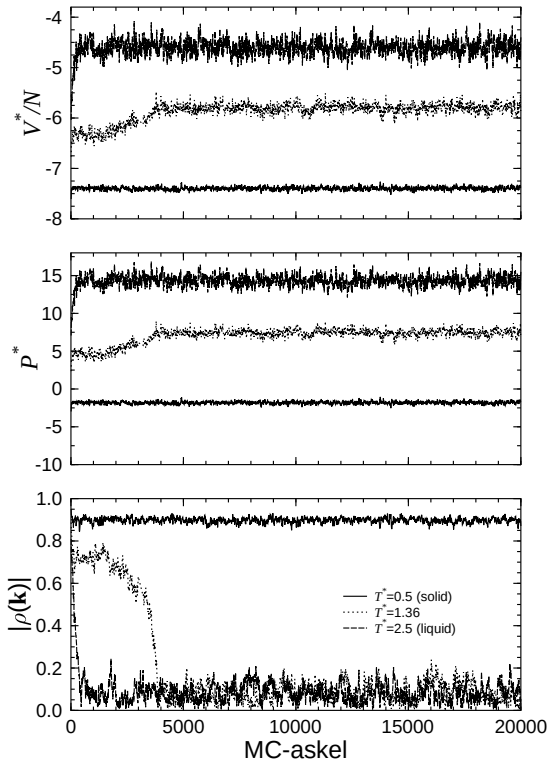
```
deltv = vnew-vold
deltvb = beta*deltv
if (deltvb <= 0.0) {
    v += deltv
    rx[i] = rxinew
    ry[i] = ryinew
    rz[i] = rzinew
    naccpt++
} else if (exp(-deltvb) > ran(seed)) {
    v += deltv
    rx[i] = rxinew
    ry[i] = ryinew
    rz[i] = rzinew
    naccpt++
}

ntrial++

... laske keskiarvoja ...
```

Tässä `deltv`= δV_{nm} ja `beta`=$\beta = 1/k_{\mathrm{B}}T$.

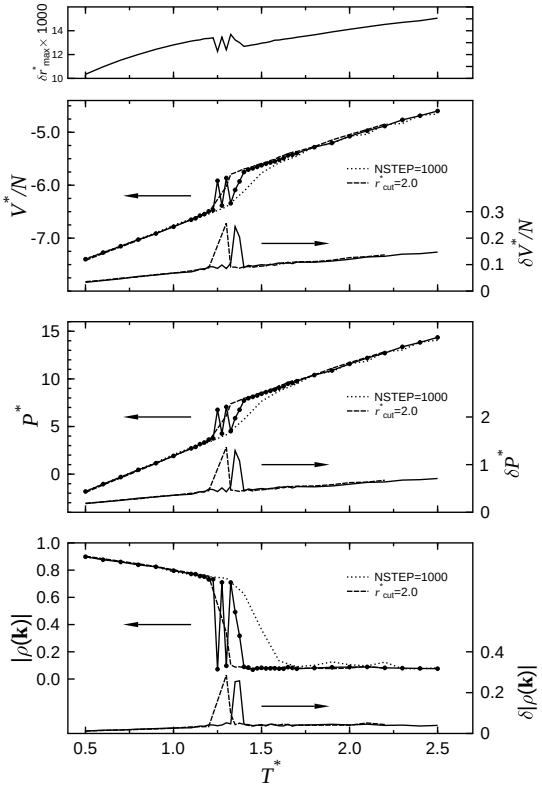
Oheisissa kuvissa on esimerkkinä Lennard-Jones -systeemin sulamisen MC-simulointi. LJ-systeemi koostuu atomeista, jotka vuorovaikuttavat potentiaalienergian

$$V(r_{ij}) = 4\varepsilon\left[\left(\frac{\sigma}{r_{ij}}\right)^{12} - \left(\frac{\sigma}{r_{ij}}\right)^{6}\right] \tag{12.32}$$

välityksellä. Tässä r_{ij} on kahden atomin välinen etäisyys[1].



Kuva 12.9: LJ-systeemin MC-simulointi. Potentiaalienergia V^*, paine P^* ja rakennetekijä ρ on piirretty simulaatioaskeleen funktiona. Kun rakennetekijä on lähellä ykköstä, on systeemi kiteinen. Suureet on esitetty ns. redusoiduissa LJ-yksiköissä.



Kuva 12.8: LJ-systeemin MC-simulointi eri lämpötiloissa. Yritesiirron suuruus $\delta r^*_{\max}$, potentiaalienergia V^*, paine P^* ja rakennetekijä ρ on piirretty lämpötilan T^* funktiona. Systeemi sulaa lämpötilassa $T^* \approx 1.3$

1. Tarkempaa tietoa kyseisistä simulaatioista saat *Laskennallisen tieteen erikoiskurssin* kotisivulta *http://www.lce.hut.fi/teaching/S-114.250/*

12.3 Ajasta riippuvien ilmiöiden Monte Carlo -simulointi

12.3.1 Kineettinen Monte Carlo

MC-menetelmää voidaan tietysti käyttää myös ajasta riippuvien ilmiöiden mallintamiseen. Usein puhutaan kineettisestä Monte Carlosta (KMC)[1].

Tärkeitä sovellutuksia ovat mm. diffuusio kiinteässä aineessa, kiteen kasvu, säteilyvaurioiden kulkeutuminen, erilaisten jonosysteemien (liikenne ym.) simulointi.

Systeemiä kuvataan masteryhtälöllä

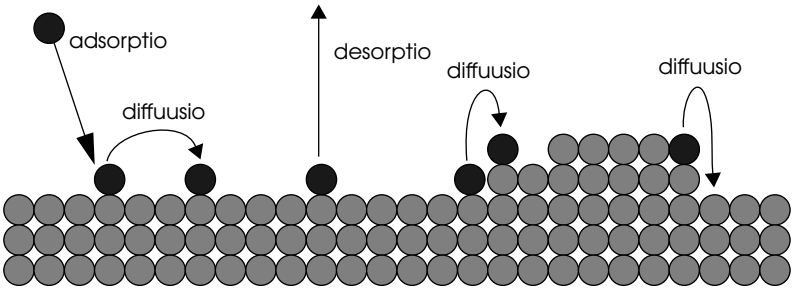
$$\frac{\partial P(C,t)}{\partial t} = \sum_{C'} [W(C' \to C)P(C',t) - W(C \to C')P(C,t)] \quad , \qquad (12.33)$$

missä C indeksoi systeemin tilaa.

Kuten staattisessa MC:ssä (SMC), KMC:ssäkin masteryhtälö ratkaistaan generoimalla jono systeemin tiloja. SMC:ssä ainoa kriteeri generoiduille jonoille oli, että mikroskooppinen reversiibeliys pätee. Tämä takasi konvergenssin tasapainojakaumaa kohti. KMC:ssä tämän lisäksi siirtymien tilasta toiseen tulee vastata todellisia systeemin siirtymiä.

KMC perustuu siihen, että kuvaamme systeemiä sellaisella tasolla, että voimme erottaa tietyn joukon diskreettejä tapahtumia $\mathbf{E} = \{e_1, e_2, \ldots, e_N\}$, joissa systeemi siirtyy tilasta toiseen. Lisäksi aikaskaala on sellainen, että mitkään tapahtumat eivät tapahdu samanaikaisesti.

Otetaan esimerkiksi kiteen kasvu kaasufaasista:



Kuva 12.10: Kiteen kasvatus.

Tapahtumia ovat: adsorptio, desorptio ja atomin hypyt pinnalla erilaisissa geometrioissa.

Olkoon tapahtuman a todennäköisyys aikayksikköä kohti (*rate*) R_a . Yleensä todennäköisyydet $\{R_a\}_{a=1}^{N}$ ja N riippuvat konfiguraatiosta C .

Määritellään kokonaistodennäköisyys siirtymälle (*total rate*) tietyssä konfiguraatiossa

1. K. Binder, *Monte Carlo Methods in Statistical Physics*, toim. K. Binder, Topics in Current Physics 7, (Springer-Verlag, Berlin, 1986), 2. painos ; K. A. Fichthorn, W. H. Weinberg, *J. Chem. Phys.* **95** (1991) 1090; A. C. Levi, M. Kotrla, *J. Phys.: Condens. Matter* **9** (1997) 299.

$$Q = Q(C) = \sum_{a=1}^{N} R_a \quad . \tag{12.34}$$

Muodollisesti siirtymätodennäköisyys W voidaan kirjoittaa muodossa

$$W(C \to C') = \sum_{a=1}^{N} R_a V^a(C \to C') \quad , \tag{12.35}$$

missä V^a on tavallaan stokastinen yritematriisi. Se kertoo, onko siirtymä $C \to C'$ mahdollinen tapahtuman a kautta.

Simulaatiossa tapahtuma a pitäisi valita todennäköisyydellä $R_a / Q(C)$ eli verrannollisesti todennäköisyyteen aikayksikkö kohti.

Suoraviivainen algoritmi tämän toteuttamiseksi olisi seuraava. Simulaation alussa etsimme tapahtuman, jolla on suurin mahdollinen todennäköisyys aikayksikköä kohti $R_{\max}$. Laskemme kaikille mahdollisille tapahtumille suhteelliset todennäköisyydet

$$P_a = \frac{R_a}{R_{\max}} \quad , \tag{12.36}$$

ja luomme listan alkukonfiguraatiossa mahdollisista tapahtumista. Algoritmin sisin silmukka askeleella k voisi olla seuraavanlainen:

i. Valitse satunnaisesti yksi konfiguraatiossa C_k mahdollinen tapahtuma. olkoon sen suhteellinen todennäköisyys P_e .

ii. Generoi tasaisesti jakautunut satunnaisluku $\xi \in [0, 1)$.

iii. Jos $\xi \leq P_e$, päivitä systeemin tila $C_{k+1} = C'$ ja mahdollisten tapahtumien lista. Muuten pysy entisessä tilassa.

Tämä algoritmi ei ole kovin käyttökelpoinen, koska kaikkia tapahtumia yritetään samalla todennäköisyydellä. Usein systeemissä on tapahtumia, joiden todennäköisyydet aikayksikköä kohti eroavat jopa kertaluvuilla. Tällöin epätodennäköisiä tapahtumia yritetään ja hylätään usein.

Esimerkiksi diffuusiosimulaatiossa atomeja, joilla on suuri tai pieni koordinaatio[1] yritetään siirtää yhtä usein, mutta atomi, jolla on pieni koordinaatio liikkuu paljon enemmän.

Usein mallista voidaan unohtaa tapahtumat, joiden todennäköisyys on kovin pieni. Aina tämä

1. Todennäköisyyksien suhde on $e^{-\beta \Delta E}$, missä ΔE on atomien sidosenergioiden ero.

ei kuitenkaan ole mahdollista. Esimerkkinä voisi mainita nukleaation kiteen kasvun simulaatiossa. Usein adatomin desorption todennäköisyys on paljon suurempi kuin useamman adatomin liityminen klusteriksi. Kuitenkin, kun klusteri on saavuttanut tietyn kriittisen koon, se alkaa kasvaa.

Nopeampi algoritmi, jossa vältetään tuloksettomat yritykset on ns. BKL-algoritmi[1]. Siinä tapahtuma alunperin valitaan oikealla todennäköisyydellä ja toteutetaan.

i. Generoi tasaisesti jakautunut satunnaisluku $\xi \in [0, Q(C_k))$.

ii. Valitse tätä vastaava tapahtuma: valitse ensimmäinen indeksi s, jolle pätee

$$\sum_{a=1}^{s} R_a(C) \geq \xi \quad .$$

iii. Etene uuteen konfiguraatioon C_{k+1} toteuttamalla tapahtuma s.

iv. Päivitä niitä todennäköisyyksiä R_a, jotka ovat muuttuneet tapahtuman s seurauksena. Päivitä Q ja muut tarvittavat tietorakenteet.

Tarkastellaan algoritmin CPU-ajankäytön riippuvuutta systeemin koosta N. Askeleiden i. ja iii. ajankäyttö ei riipu N:stä. Askel ii. vaatii ajan $O(N)$. Useimmiten askeleen iv. suoritusaika ei riipu systeemin koosta, vaikkakin se vaatii huolellista ohjelmointia.

Vielä nopeammaksi algoritmin saadaan, jos tapahtumat ryhmitellään[2]. Ryhmittely voidaan tehdä ottamalla jokaiseen ryhmään sama määrä tapahtumia (tehokkainta) tai luokittelemalla samantyyppiset *prosessit* samoihin ryhmiin (kiteenkasvatuksessa adatomin diffuusio tietyn energiavallin yli, tietyn sidosenergiaisen adatomin desoprtio, jne.).

Ylläolevissa algoritmeissa ei ollut aikaa eksplisiittisesti mukana. Joskus kuitenkin haluamme kiinnittä systeemin kehityksen aikaskaalan.

Emme voi suoraan olettaa, että MC-askel (MCS) kertoo ajan: $t \propto \mathrm{MCS}$. Jos voimme olettaa, että vain yksi tapahtuma voi tapahtua kerrallaan (eli aikaskaala, jolla systeemiä tutkitaan on tarpeeksi pieni) ja että tapahtumat ovat Poisson-jakautuneita saamme laskuihimme aikaskaalan.
Todennäköisyys, että aikavälillä $[0, t]$ on p kappaletta tapahtumia on

$$P(p) = \frac{(Qt)^p}{p!} e^{-Qt} \quad , \tag{12.37}$$

missä edelleen $Q = Q(C) = \sum_{a=1}^{N} R_a$ .Todennäköisyys, että ko. välillä ei ole yhtään tapahtumaa on siten

1. A. B. Bortz, M. H. Kalos, J. L. Lebowitz, *J. Comput. Phys.* **17** (1975) 10.
2. P. A. Maksym, *Semicond. Sci. Tech.* **3** (1988) 594.

$$P(0) = e^{-Qt} \quad . \tag{12.38}$$

Tapahtumien välisen ajan τ (odotusaika, waiting time) (oikein normitettu) todennäköisyysjakauma on siten

$$P(\tau) = Qe^{-Q\tau} \quad , \tag{12.39}$$

ja sen odotusarvo on

$$\langle \tau \rangle = \frac{1}{Q} \quad . \tag{12.40}$$

Simulaatioissa aika saadaan siis generoimalla ylimääräinen satunnaisluku ξ_3 ja laskemalla siitä aikaväli seuraavaan tapahtumaan:

$$\Delta t_k = - \frac{\ln \xi_3}{Q(C_k)} \, . \tag{12.41}$$

Erilaiset aikakeskiarvot lasketaan simulaatioissa seuraavasti:

$$\langle A \rangle = \frac{1}{t} \sum_{k=1}^{M} \Delta t_k A(C_k) \quad , \tag{12.42}$$

missä kokonaisaika on

$$t = \sum_{k=1}^{M} \Delta t_k \quad . \tag{12.43}$$

12.3.2 Yksinkertaisten kuljetusilmiöiden simulointi

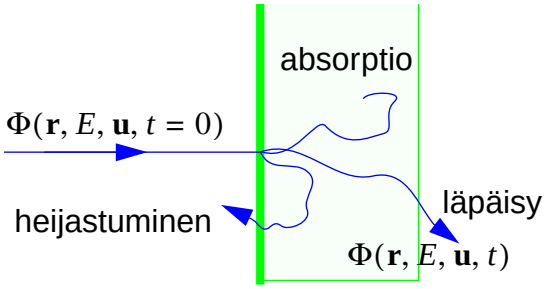
Tiettyjä kuljetusilmiöitä voidaan simuloida yksinkertaisemmilla menetelmillä kuin edelläesitetty KMC. Otetaan esimerkiksi fotoni- ja elektronisäteilyn eteneminen väliaineessa.

Analyyttisten ratkaisutapojen ongelmana on mm. sekundäärihiukkasten synty: elektroni-gamma-kaskadi.

MC-simulaation periaate on seuraava:

Seurataan hiukkasta väliaineessa vuorovaikutuksesta toiseen.

Vuorovaikutusten keskimääräinen välimatka on



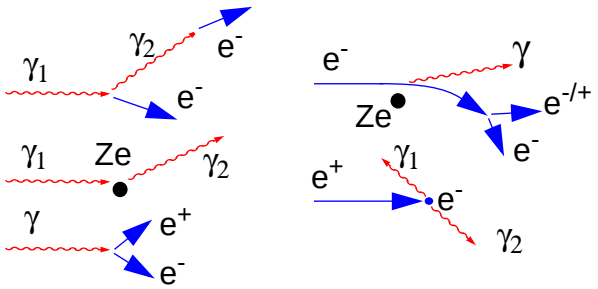
Kuva 12.11: Säteilyn ja aineen vuorovaikutus.

$$\lambda \propto \sigma_t^{-1} \ , \tag{12.44}$$

missä σ_t on kokonaisvaikutusala (todennäköisyys), ja välimatkan s todennäköisyysjakauma on

$$P(s) = e^{-s/\lambda}/\lambda \quad . \tag{12.45}$$

Simuloinnissa generoidaan välimatkoja s ja jokaisen vuorovaikutuksen kohdalla arvotaan vuorovaikutusmekanismi (ks. kuva 12.12). Hiukkasen uusi suunta ja energia saadaan ko. vuorovaikutuksen differentiaalivaikutusalasta. Myös mahdollisen sekundäärihiukkasen tiedot otetaan muistiin.



Kuva 12.12: Fotonien ja elektronien vuorovaikutusmekanismit aineessa.