# 4 Matemaattiset algoritmit

Tämä luku sisältää erilaisia matemaattisia ongelmanratkaisuja. Luvun aiheita voi hyödyntää myös matematiikan opetuksessa. Vaihtelu virkistää! Juuri tämän luvun myötä voi todeta tietokoneen työskentelytavan ihmiseen verrattuna: ihminen laskee valmiilla kaavalla, kun taas tietokone pyrkii tekemään runsaasti työtä ja lähestymään ratkaisua askel askeleelta.

## 4.1 Kertoma

Kertoma on kokonaislukujen tulo. Kertoman merkkinä käytetään matematiikassa yleisesti huutomerkkiä (!). Esimerkiksi kertoma 3! = 1*2*3 = 6. Luku nolla (0) lasketaan kokonaislukuihin ja sen kertoma (0!) on yksi.

### Kertoma:

```
#include <iostream.h>

void main()
{
int i, j, a;
j = 1;
cout << "Minkä luvun kertoma lasketaan? \n";
cin >> a;

for (i=1; i<=a; i++)
j = j * i;

cout << "Kertoma on " << j << "\n";
}
```

**Huomautus**
Jos kertoman arvo kasvaa niin suureksi, että tulos ei mahdu tietotyypin arvoalueelle, kiepsahtaa arvoalue uudelleen nollaan ja lisää jäljelle jääneen osan tulokseksi. (Tietenkin, jos tietotyyppi on etumerkillinen (signed), hypätään arvoalueen ylittämisen jälkeen pienimpään negatiiviseen arvoon nollan sijaan.)
Tämän takia kertomassa voisikin suuria lukuja varten käyttää double-tyyppiä.

# 4.2 Likiarvoja

Suhteellisen nopeutensa ansiosta tietokone soveltuu ihmistä paremmin erilaisten likiarvojen laskemiseen. Monet päättymättömät likiarvot voidaan laskea sarjojen avulla.

## 4.2.1 Neperin luvun (e) likiarvon laskeminen

Sovellamme tässä esimerkissä edellä kuvattua kertoman algoritmia Neperin luvun likiarvon saamiseksi. Neperin luku (e) on erittäin tärkeä esimerkiksi eksponenttifunktion kantalukuna, koska tällöin funktion derivaattafunktio on sama kuin alkuperäinen funktio.

Neperin luku saadaan päättymättömän sarjan avulla seuraavasti:

$$e = \sum_{k=0}^{\infty} (1/k!)$$

Jotta ohjelma olisi järkevä, asetetaan ylärajaksi jokin kokonaisluku, joka tallennetaan muuttujaan.
Jo ylärajalla 8 saadaan Neperin lukuun 5 oikeata desimaalia.

**Neperin luku:**

```
#include <iostream.h>
int kertoma(int a);
float neper(int n);

void main()
{
int a;
cout << "Kuinka tarkka neper lasketaan? \n";
cin >> a;
cout << "Neper on " << neper(a) << "\n";
}

int kertoma(int a)
{
int i;
int j = 1;
if (a == 0) return 1;
for (i=1; i<=a; i++)
```

```
j = j * i;
return j;
}

float neper(int n)
{
int laskuri;
float tulos = 0;
for (laskuri = n; laskuri >= 0; laskuri--)
    {
    tulos = tulos + 1/(float)kertoma(laskuri);
    cout << tulos << " " << laskuri <<  "\n";
    }
return(tulos);
}
```

Lisää ylle laskentatarkkuus esimerkiksi kahden perättäisen Neperin luvun välisen erotuksen avulla.

Neperin luvun likiarvo voitaisiin laskea myös lukujonon raja-arvosta

$$e = \lim (1 + 1/n)^n \text{ , kun } n \to \infty$$

Lausekkeen hyvä puoli on se, että n on positiivinen kokonaisluku. Laskentatarkkuus riippuu mm tietokoneen ominaisuuksista. Mikäli arvo ∞ tuntuu oudolta, voi sen sijaan käyttää raja-arvoa:

$$e = \lim (1 + 1/h)^{1/h} \text{ , kun } h \to 0$$

## 4.2.2 $e^x$, cos(x) ja sin(x) sarjakehitelmillä

Edellä kuvattua kertomaa voidaan hyödyntää vielä reaalialueen sarjakehitelmissä.

$$\mathbf{e^x} = 1 + x + x^2/2! + x^3/3! + \ldots$$

$$\mathbf{cos(x)} = 1 - x^2/2! + x^4/4! - \ldots$$

$$\mathbf{sin(x)} = 1 - x^3/3! + x^5/5! - \ldots$$

### Lasketaan $e^x$:

```
Määritetään eksponentti x
```

```
Määritetään tarkkuus e, jota verrataan kahden peräkkäisen tuloksen
erotukseen
(toinen vaihtoehto on määrittää tarpeeksi pieni yläraja summalle)
Määritetään jokin yläraja, MAX
Tulos = 0
i = 0 to MAX
Tulos = Tulos + xi/i!
```

Vastaavasti voidaan määritellä likiarvo lausekkeelle **cos(x)**. Hankaluutena on tässä tapauksessa se, että jokaisen peräkkäisen summatekijän etumerkki vaihtuu. Etumerkkiä ei voida määrittää myöskään summausindeksin parillisuuden/parittomuuden perusteella. Siksi algoritmissa käytetään kahta eri summaa, joissa toisessa ovat negatiiviset ja toisessa positiiviset tekijät. Lopuksi osasummat lasketaan yhteen.

### Lausekkeen cos(x) likiarvo:

```
Tulos1 = 0
Tulos2 = 0
i = 0 to MAX, i = i + 4
Tulos1 = Tulos1 + xi/i!
i = 2 to MAX, i = i + 4
Tulos2 = Tulos2 - xi/i!
Tulos = Tulos1 + Tulos2
```

### Lausekkeen sin(x) likiarvo:

```
Tulos1 = 0
Tulos2 = 1
i = 3 to MAX, i = i + 4
Tulos1 = Tulos1 - xi/i!
i = 5 to MAX, i = i + 4
Tulos2 = Tulos2 + xi/i!
Tulos = Tulos1 + Tulos2
```

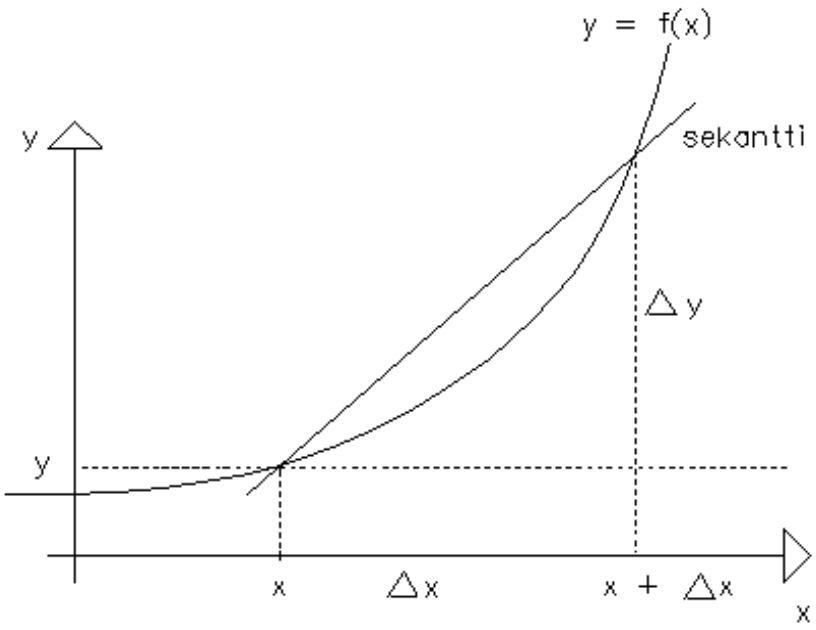
**π:n likiarvo** voidaan määritellä likiarvo hieman ‘nurinkurisesti’ esimerkiksi hyödyntäen Pascalin arcustangentti-funktiota. Kun tiedetään, että arctan(1) = π/4, saadaan π = arctan(1) * 4.

# 4.3 Derivaatta, differentiaali ja yhtälöt

Tässä aliluvussa käsitellään derivaatan ja differentiaaliyhtälön numeerista käsittelyä. Mukana on myös tavallisten yhtälöiden ratkaiseminen.

## 4.3.1 Derivaatan likiarvo

Tässä käsitellään hieman derivaatan laskemista lähinnä sen määritelmän pohjalta. Allaoleva kuva havainnollistaa derivaatan määritelmää erotusosamäärän käsitteellä:




Kun delta_x lähestyy nollaa, segmentti muuttuu tangentiksi. Tangentin kulmakerroin kertoo käyrän kasvunopeuden sivuamispisteen kohdalla. Edellä olevassa kuvassa funktion keskimääräinen kasvunopeus välillä (x ... x + delta_x) saadaan erotusosamäärän avulla.

Erotusosamäärä on delta_y/delta_x. Kyseinen erotusosamäärä kertoo suoraan lineaarisen käyrän muuttumisnopeuden eli se on samalla itse käyrän (eli suoran) kulmakerroin. Epälineaarisilla funktioilla käyrän kasvunopeus riippuu tietyllä välillä x-arvojen valinnasta.

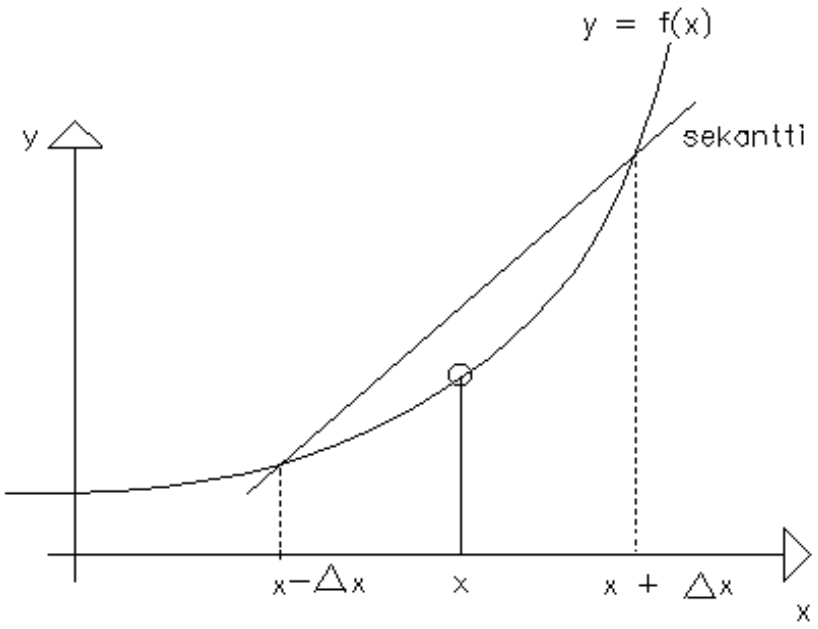
Derivaatta y'(x) = f'(x) = lim delta_y/delta_x  (kun delta_x->0)

= lim (f(x+delta_x) - f(x))/delta_x  (kun delta_x->0)

Derivaatta edustaa siis funktion paikallista kasvunopeutta eli kasvunopeutta äärettömän lyhyellä välillä.

Funktion kasvunopeus voidaan approksimoida tietokoneella muuttamalla arvoa delta_x mahdollisimman pieneksi ja määrittelemällä haluttu tarkkuus. Erityisesti approksimointi soveltuu käytettäväksi silloin, kun funktion suoraa derivointia ei voida jostain syystä tehdä. Monissa yhteyksissä riittää myös funktion keskimääräisen muuttumisnopeuden laskeminen tietyllä välillä.

Numeerinen derivointi suoritetaan yleisemmin ottamalla delta_x molemmin puolin x-arvoa. Kuvamme selvittää tilanteen:



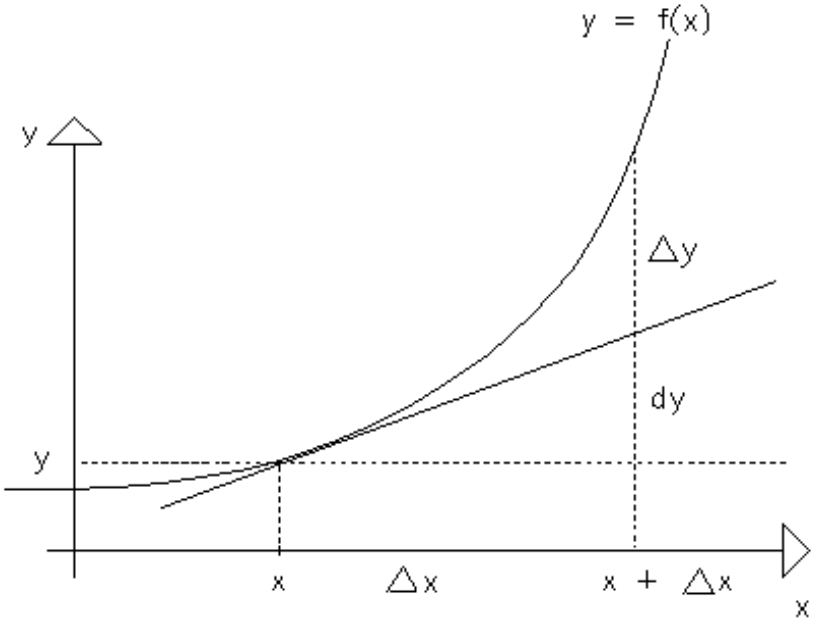

f'(x) on siis tangentin kulmakerroin ≈ sekantin kulmakerroin.

Eli kuvasta saadaan f'(x) = (f(x + delta_x) - f(x - delta_x))/ 2*delta_x

Edelleen saadaan f''(x) = (f(x + delta_x) * 2f(x) + f(x - delta_x))/ $delta\_x^2$

## 4.3.2 Differentiaali

Differentiaalin avulla kuvataan funktioissa tapahtuvia muutoksia. Alla oleva kuva selvittää differentiaalin käsitettä ja antaa samalla perusteet differentiaalin approksimointiin tietokoneella.




Kun funktio on sopiva ja delta_x lähestyy nollaa, kuvaa delta_y differentiaalia dy. Differentiaali dy on siis kasvunopeus käyrän alkupisteessä * välin pituus, eli
dy = f'(x) * delta_x

Havainnollistamme differentiaalin laskentaa esimerkillä:
Jos funktio on muotoa $1/3\ x^3$ ja halutaan tietää, paljonko funktio muuttuu, kun x kasvaa arvosta 2 arvoon 2.1.

delta_x = 0.1
$f(x) = 1/3\ x^3$ , jolloin $f'(x) = x^2$
$dy = f'(x) * delta\_x = x^2 * 0.1 = 2^2 * 0.1 = 0.4.$

Funktio kasvaa siis 0.4 yksikköä, kun x kasvaa arvosta 2 arvoon 2.1.

Differentiaalin avulla voidaan siis laskea funktiossa tapahtuva muutos (dy), kun tiedetään x:n muutos (delta_x). Approksimointi on nyt helppo tehdä, koska olemme aiemmin jo käyneet läpi derivaatan approksimoinnin. Nyt f'(x):n likiarvo tietyssä pisteessä saadaan aiemmasta algoritmista, jolloin tarvitaan vain antaa delta_x, jotta dy saadaan laskettua.

## 4.3.3 Differentiaaliyhtälön numeerinen ratkaiseminen

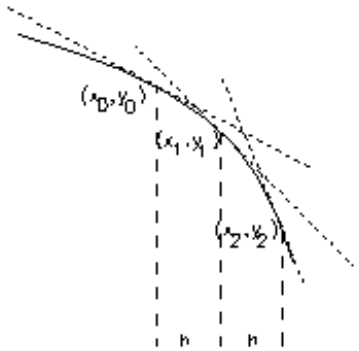
Differentiaaliyhtälön y' = r(x,y) yleistä ratkaisua edustaa käyräparvi, jossa jokaisella C:n arvolla on oma käyränsä. Jos jonkin tunnetun pisteen $(x_0,y_0)$ kautta kulkevan käyrän suunta halutaan tietää, voidaan laskea y', joka on siis kyseisen käyrän tangentin kulmakerroin pisteessä $(x_0,y_0)$.

*Eulerin menetelmällä* voidaan saada esille ratkaisukäyrän likimääräisiä pisteitä ratkaistaessa yhtälöä:

y' = r(x,y), missä ratkaisufunktion y = y(x) tulee toteuttaa alkuehto $y(x_0) = y_0$.

Tällöin edetään radan tangentin suuntaan lyhyitä askelia. Askelpituus delta_x = h.

Kuva havainnollistaa menettelyä:




Menettelyllä saadaan ratkaisukäyrän pisteitä seuraavasti:
$x_0$ ja $y_0$ siis tiedetään.

$x_1 = x_0 + h$
$x_2 = x_1 + h$
jne

$y_1 = y(x_0 + h) \approx y_0 + r(x_0, y_0)h$
$y_2 \approx y_1 + r(x_1, y_1)h$
jne

Algoritmissa annetaan alkuarvopiste ja r(x,y):n lauseke ja määritetään askelpituus sekä väli, jolta pisteitä halutaan.

*Runge-Kuttan menetelmää* pidetään yleisesti Euleria tarkempana menetelmänä. Runge-Kutta sisältää Eulerin periaatteen, mutta uusi piste lasketaan (ekstrapoloidaan) paremmalla menetelmällä.

Runge-Kutta-menetelmässä käytetään kaavaa

$$y_{n+1} = y[x_0 + (n+1)h] \approx y(x_{n+1}) \approx y_n + h/6(k_1 + 2k_2 + 2k_3 + k_4), \text{ missä}$$
$$k_1 = r(x_n, y_n)$$
$$k_2 = r(x_n + h/2, y_n + h/2\, k_1)$$
$$k_3 = r(x_n + h/2, y_n + h/2\, k_2)$$
$$k_4 = r(x_n + h, y_n + h\, k_3)$$

Kyseiset kertoimet kuuluvat neljännen kertaluvun Runge-Kutta-menettelyyn. Toisen kertaluvun Runge-Kutta-menettely on melko lähellä Euleria, mutta siinä haetaan pisteet kohdasta, jossa askel on h/2.

Algoritmi on Euleria työläämpi, mutta sinänsä yksinkertainen laadittava.

## 4.3.4 Toisen asteen yhtälö

Yhtälö kirjoitetaan yleismuodossa seuraavasti:

$$ax^2 + bx + c = 0$$

Yhtälön juuret saadaan muodosta:

$$x = (-b \pm \sqrt{(b^2 - 4ac)})/2a$$

Yhtälöllä voi olla kaksi reaalijuurta $x_1$ ja $x_2$ tai kaksoisjuuri $x_{1,2}$ tai sillä ei ole reaalijuuria (vaan imaginäärijuuret), jolloin funktion kuvaaja ei leikkaa tai sivua x-akselia.

### 2. asteen yhtälö:

Yhtälön juuret voidaan tarkistaa lauseketta (b2 - 4ac) tutkimalla.

Jos lausekkeen arvo > 0, yhtälöllä on kaksi reaalijuurta.
Jos lausekkeen arvo = 0, yhtälöllä on reaalinen kaksoisjuuri.
Jos lausekkeen arvo < 0, yhtälöllä on kaksi imaginäärijuurta.

Juurten muotojen ja yhtälön ratkaisun algoritmi on

$x1 = (-b + \sqrt{(b2 - 4ac)})/2a$
$x2 = (-b - \sqrt{(b2 - 4ac)})/2a$

### Huomautus

Otsikkotiedostoa complex.h käyttäen voit laskea myös imaginääriset juuret. Myös muita mahdollisuuksia on kompleksiluvuille eri työkaluissa.
Muutoin negatiivisten lukujen neliöjuurien syntymisen tulisi heittää poikkeus, joka ohjelmassa käsitellään. Heitettävänä poikkeusluokkana on yleensä runtime_error ja sen aliluokka numeric_error.

## 4.3.5 Muut yhtälöt

Usein korkeampiasteisia yhtälöitä ei saada hajotettua osiin, jolloin niitä voitaisiin sieventämisen jälkeen ratkaista vähemmällä työllä. Yksi mahdollisuus on luonnostella kuvaajaa ja määrittää siitä X-akselin leikkauspisteitä. Kuvaajasta voidaan saada myös alkuarvo tarkemmalle iteroinnille, jolloin juuren arvosta saadaan tarkempi.

Tietokone on kuin luotu tekemään raskaita iterointitehtäviä. Seuraavassa esitellääään yksi malli kolmannen asteen yhtälön ratkaisemiseen iteroinnin kautta. Mikäli yhtälöllä on kokonaislukujuuri, murtolukujuuri tai päättyvä desimaalilukujuuri, voidaan sen avulla yhtälöä sieventää edelleen, jolloin muiden juurien ratkaisu helpottuu.

Olettakaamme, että yhtälö olisi muotoa $X^3 + 2X^2 - 3X + 4 = 0$.

Ensin tarvitaan alkuluku, jota kasvatetaan sopivalla lisäyksellä, kunnes päästään joko täysin oikeaan juuriarvoon tai hyväksyttävään likiarvoon.

Iterointi voidaan suorittaa seuraavasti:

### 3-asteisen yhtälön juuren hakeminen:

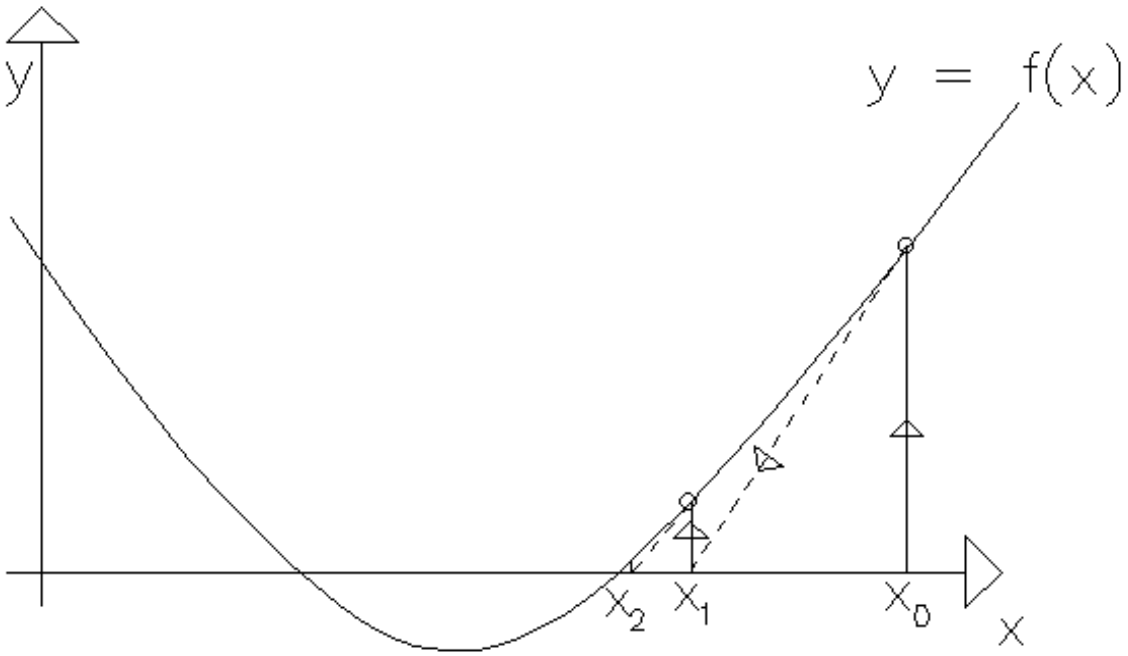
```
#include <iostream.h>

void main()
{
float x;
float y, tulos;
x = -3;
do
{
    x = x + 0.1;
    y = x*x*x + 2*x*x + 3*x + 6;
    tulos = y;
    tulos = (tulos < 0 ? (-1)* tulos: tulos);
    if (tulos < 0.001) {cout << x; break; };
}
while(x < 2);
}
```

Juurta kasvatetaan siis kymmenyksellä jokaisen kierroksen jälkeen. Algoritmiin tulee lisätä negatiivisen alkuluvun käyttömahdollisuus sekä riittävän likiarvon hakutoiminto.
Esimerkiksi tuloksen vaihtaessa merkkiä voitaisiin palata askel taaksepäin ja pienentää askellusarvoa sadasosiin. Myös kahden peräkkäisen tuloksen erotusta voidaan tarkkailla.

Edellä käytetty menettely vaatii runsaasti laskentaa ja etsintää, jotta juurta päästäisiin lähestymään. Niinpä yleisesti käytetään toista menetelmää, jolla etsintä nopeutuu, koska kohdetta (juuren likiarvo) päästään lähestymään huomattavasti edellistä menettelyä nopeammin. Kyseinen menettely juuren etsintään on *Newton-Raphson-menetelmä*, joka hakee juuren seuraavasti:

Kuvamme esittää Newton-Raphson-menetelmän etenemistä:




Menetelmässä piirretään annettua käyrää tietyssä pisteessä sivuava tangentti. Kyseinen alkupiste (kuvassamme $x_0$) on yhtälön juuren ensimmäinen approksimaatio. Tangentti leikkaa x-akselia tietyssä pisteessä, josta saadaan uusi käyrän piste, josta piirretyn tangentin ja x-akselin leikkauspiste on aiempaa lähempänä oleva juuren likiarvo. Näin tangenttien avulla jatketaan, kunnes on saatu riittävä tarkkuus juurelle.

**Yhtälöinä menetelmä on seuraavanlainen:**
Jos $x_0$ on ensimmäinen yhtälön juuren approksimaatio, niin pisteeseen $(x_0, y_0)$ piirretty tangentti leikkaa x-akselin pisteessä:

$$x_1 = x_0 - f(x_0)/f'(x_0)$$

Uuteen pisteeseen $(x_1, y_1)$ piirretty tangentti leikkaa x-akselin pisteessä

$$x_2 = x_1 - f(x_1)/f'(x_1)$$

Eli yleismuoto on:

$$x_{n+1} = x_n - f(x_n)/f'(x_n)$$

SIIS (yleismuodosta saatu) muutos $\Delta x = x_{n+1} - x_n = - f(x_n)/f'(x_n)$

Ohjelmallisesti voidaan uusi approksimaatio laskea joko laskemalla ensin Δx ja lisäämällä vanhaan approksimaatioon tai laskemalla suoraan uusi arvo yleismuodon mukaan. Ehkä riittävän tarkkuuden määrittelemisessä on helpompi seurata Δx:n pienenemistä suoraan kuin laskea juurten likiarvojen erotus.

**Newton-Raphson-menetelmä:**

```
Määritellään ensin f(x) ja f'(x)
Annetaan alkuarvot x0 ja mahdollisesti tarkkuus, johon tulosta
verrataan
Lasketaan uusi likiarvo x1 = x0 - f(x0)/f'(x0)
Verrataan vanhan ja uuden likiarvon erotusta tarkkuuteen
Jos tarkkuus ei riitä, lasketaan taas uusi likiarvo x2... xn kunnes
tulos on tyydyttävä.
```

## 4.3.6 Moniasteisen yhtälön kaikkien reaalijuurten etsintä

Yhtälöä kuvaava käyrä y = f(x) vaihtaa merkkiään sellaisella osavälillä ($x_i < x < x_{i+1}$), jonka sisällä käyrä leikkaa x-akselia eli kyseisen osavälin sisällä sijaitsee yksi yhtälön juurista. Tällöin on siis ensin haettava osavälejä, joissa yhtälön etumerkin vaihtuminen tapahtuu ja sen jälkeen haettava kyseinen juuri esimerkiksi edellä kuvatulla menetelmällä. Etumerkki saadaan selville esimerkiksi kertomalla aiempaa x-arvoa vastaava käyrän (y:n) arvo uutta x:n arvoa (x + askel) vastaavalla käyrän arvolla: jos tulo on negatiivinen, ovat tulon tekijät siis erimerkkiset. Mikäli tulo on nolla, on 'vahingossa' löydetty yksi juuri.

## 4.3.7 Neliöjuuren approksimointi

Neliöjuuren likiarvon saamiseksi on olemassa useita erilaisia matemaattisia menettelyjä. Eräs käytetyimmistä tavoista on hyödyntää edellä kuvattua Newton-Raphsonin approksimointimenettelyä.

Lähtökäyrä on nyt muotoa:

$$f(x) = y - x^2 = 0$$

joten

$$f'(x) = -2x$$

ja

$$\Delta x = (y - x^2)/2x$$

Neliöjuurta laskettaessa on huolehdittava siitä, että neliöjuuren sisällä oleva arvo on positiivinen.

## Neliöjuuren approksimointi:

```
#include <iostream.h>

float juuri(float arvo);

void main()
{
float luku;
// Lasketaan neliöjuuri luvusta
cout << "Anna luku \n";
cin >> luku;
cout << "Neliöjuuri on " << juuri(luku) << " \n";
}

float juuri(float arvo)
{
    float x = arvo/10;   /*ensimmäinen approksimaatio juuresta*/
    float dx;
    float tarkkuus = 0.001;
    float muutos;  /*approksimaation ja oikean tuloksen erotus*/
    if (arvo < 0) arvo = arvo * (-1);
    if (arvo == 0) return (0);   /*juurta ei oteta negatiivisesta
                          luvusta*/
    do
        {
        dx = (arvo - x * x) / (2.0 * x);
        x = x + dx;
        cout << x << "\n";
        muutos = arvo - x* x;
        }
    while ( (muutos < 0 ? (-1)*muutos : muutos) > tarkkuus);
/*tarpeeksi lähellä?*/
    return (x);
}
```

# 4.4 Kompleksiluvut ja vektorit

Tässä aliluvussa käsitellään kompleksilukujen eri muotojen sekä tasovektorien laskemista. Myös kirjastotiedosto tarjoaa funktioita kompleksilukujen käsittelyyn.

## 4.4.1 Kompleksiluvut

Kompleksiluku voidaan esittää summa-, polaari-, eksponentti- tai osoitinmuodossa. Tässä tarkastellaan hieman summa- ja osoitinmuotoa.
Summamuotoinen kompleksiluku on tyyppiä Z = a + bi, jossa a kuvaa reaaliosaa ja bi imaginääriosaa.

Kompleksilukua kuvaavan vektorin pituus R Re-Im-koordinaatistossa on kompleksiluvun itseisarvo ja vektorin kulma (Re-akseliin nähden myötäpäivään) saadaan trigonometrisesti laskemalla kulma = arcussin (a/R).

Arcus-funktioissa voi olla eroja riippuen kääntäjästä. Tulos on kuitenkin yleisesti radiaanimuodossa. π:n likiarvo (jos se joudutaan laskemaan) saadaan kirjassa toisaalla olevalla algoritmilla.

Kompleksiluku on siis muotoa z = a + bi. Tämä kompleksiluvun perusmuoto on *summamuoto*. Kompleksiluvun z itseisarvo (vektorin pituus), $|z| = (a^2 + b^2)^{1/2}$. Kompleksiluvuilla lasketaan kuten tavallisilla kirjainmerkeillä, ainoastaan sievennysvaiheessa otetaan huomioon, että $i^2 = -1$.

Kompleksiluvun *polaarinen muoto* on $z = r(\cos\varphi + i\sin\varphi)$. Komponentit a ja b saadaan seuraavasti: $a = r\cos\varphi$ ja $b = r\cos\varphi$, jossa r on vektorin pituus eli kompleksiluvun itseisarvo. Tässä muodossa kompleksiluvut kerrotaan keskenään siten, että pituudet kerrotaan keskenään ja vaihekulmat lasketaan yhteen, eli $z_1z_2 = r_1\, r_2\, [\cos(\varphi_1 + \varphi_2) + i\sin(\varphi_1 + \varphi_2)$. Jakolaskussa vastaavasti pituudet jaetaan keskenään ja vaihekulmat vähennetään toisistaan.

Kompleksiluvun *osoitinmuoto* on yleinen sähkötekniikassa. Tällöin polaarisesta muodosta $z = r(\cos\varphi + i\sin\varphi)$ käytetään lyhyempää muotoa $r\,\angle\varphi$ (luetaan 'r kulmassa φ').

Kompleksiluvusta on vielä *eksponenttimuoto*, jossa hyödynnetään Eulerin kaavaa $e^{i\varphi} = \cos\varphi + i\sin\varphi$. Nyt kompleksiluku $z = r(\cos\varphi + i\sin\varphi)$ tai $r \angle\varphi$ voidaan esittää muodossa $z = r\, e^{i\varphi}$ .

Muodostetaan seuraavassa algoritmi summamuotoisen kompleksiluvun muuntamiseksi osoitinmuotoon:

### Summamuotoinen kompleksiluku osoitinmuotoon:

```
Annetaan reaali- ja imaginääriosat
Lasketaan pituus r (algoritmi edellä)
Kulma_rad = arctan(b/a)
Kulma_ast = Kulma_rad*360/π
Tulosta esimerkiksi muodossa 'Annettu kompleksiluku z = a + bi on
osoitinmuodossa
r kulmassa Kulma_ast.'
```

Algoritmiin on helppo lisätä myös polaarimuodon tulostus, koska mitään lisälaskentoja ei tarvita. Sama koskee eksponenttimuotoa.
Summamuotoon muuttaminen tehdään muista muodoista (eli tiedetään r ja φ) muodostamalla ensin $z = r(\cos\varphi + i\sin\varphi)$ ja laskemalla erikseen $r\cos\varphi$ ja $r\sin\varphi$ sekä lisäämällä tulostuksessa viimeksimainittuun arvoon imaginääriyksikkö i.

Huomaa myös, että C++ sisältää standardikirjaston complex.h, jonka avulla voidaan käsitellä kompleksilukuja. Math.h-tiedoston funktiot hyväksyvät myös usein tietyssä muodossa olevan kompleksiluvun (esimerkiksi abs() ).

## 4.4.2 Vektorit

Tason vektori voidaan esittää muodossa $a = xi + yj$ eli $a = [x,y]$ ja vastaavasti avaruusvektori muodossa $a = xi + yj + zk$ eli $a = [x,y,z]$.

Tarkastellaan seuraavassa kahta tason vektoria a ja b, jotka esittävät vektoreita

$a = x_1 i + y_1 j$
ja
$b = x_2 i + y_2 j$.

Vektoreiden *summa* lasketaan siten, että vektoreiden komponentit lasketaan yhteen.

$$a + b = (x_1 + x_2)i + (y_1 + y_2)j.$$

*Erotus* lasketaan vastaavalla tavalla.

Vektorin a *pituus*, |a|, lasketaan summaamalla vektorin komponenttien neliöt ja ottamalla summasta neliöjuuri.

Vektorin a pituus = $|a| = (x_1^2 + y_1^2)^{1/2}$

Vektoreilla on kaksi erilaista tuloa: *pistetulo* (eli skalaaritulo) ja *ristitulo* (eli vektoritulo). Pistetuloa merkitään notaatiolla piste **.** (esimerkiksi a **.** b) ja ristituloa notaatiolla x (esimerkiksi a x b). Pistetulo lasketaan kertomalla vektoreiden pituuksien tulo vektoreiden välisen kulman kosinilla.

Kahden vektorin a ja b *pistetulo*, a **.** b = |a| |b| cos(a,b). Pistetulon tulos on skalaari. Nyt vektorin a pituus saadaan myös pistetulosta eli $|a| = (a \,.\, a)^{1/2}$ . Tason (ja avaruuden) vektoreiden pistetulo saadaan myös kertomalla vektoreiden vastinkomponentit keskenään ja summaamalla tulot eli vektoreiden a ja b *pistetulo*,

$$a \,.\, b = x_1 y_1 + x_2 y_2.$$

*Ristitulo* saadaan ottamalla determinantti matriisista, jonka alkioina ovat i ja j sekä vektoreiden komponentit.

Kahden tason vektorin a ja b ristitulo,

$$a \times b = |a| \, |b| \sin(a,b) \, e,$$

missä e on yksikkövektori.

Avaruusvektoreiden a ja b ristitulo voidaan esittää näppärässä matriisimuodossa seuraavasti:

$$a \times b = \det [ (i,j,k), (x_1, y_1, z_{1)}, (x_2, y_{2,} z_{2)} ]^T .$$

Vektorin a projektio vektorilla b eli *projektiovektori* $a_b$ saadaan seuraavasti:

Vektorin a projektio vektorilla b , $a_b = (a \,.\, b)/(b \,.\, b) * b$.

Kahden vektorin a ja b *välinen kulma*, (a,b) saadaan seuraavasti:
Vektoreiden a ja b välinen kulma,

$$(a,b) = \arccos((a \cdot b)/(|a|\,|b|)).$$

Tulos on radiaaneissa ja se muutetaan asteiksi kertomalla luvulla $360/\pi$.

Vektorin a *yksikkövektori*, $a_0$, saadaan jakamalla vektori a sen pituudella |a|. vektorina yksikkövektori:

$$a_0 = a/|a|.$$

Lopuksi tarkistetaan kahden vektorin a ja b yhtäsuuruus. Kaksi vektoria ovat yhtäsuuria, jos ne ovat samansuuntaisia ( eli a | | b ) ja niiden pituudet ovat samat (eli |a| = |b|).

Samansuuntaisuus saadaan selville siten, että lasketaan determinantti matriisista, jonka alkioina ovat a:n ja b:n komponentit eli det $[(x_1\ y_1), (x_2\ y_2)]^T$ . Jos determinantti = 0, ovat vektorit samansuuntaisia.

## Algoritmi vektorioperaatioille:

**Suoritetaan ensin operaatioita tason vektoreille a ja b eli a = x1i + y1j ja b = x2i + y2j.**
**Annetaan ensin vektoreiden a ja b komponentit (x1, y1, x2, y2).**
**Helpointa lienee antaa komponentit muodossa [komponentti1, komponentti2].**
**Lasketaan pituudet |a| ja |b|  eli |a| = (x12 + y12)1/2ja |b| = (x22 + y22)1/2**
**Lasketaan vektoreiden a ja b pistetulo a . b = x1 y1  + x2 y2.**
**Käytetään nyt tätä kaavaa, koska toisessa kaavassa (a . b = |a| |b| cos(a,b)) tarvittaisiin vektoreiden välistä kulmaa, jota emme toistaiseksi saa kaavaa varten esille.**
**Lasketaan vektoreiden a ja b välinen kulma (a, b) hyödyntämällä edellä laskettua pistetuloa sekä aiemmin laskettuja vektoreiden pituuksia: (a,b) = arcos ((a . b)/(|a| |b|)).**
**Lasketaan tämän jälkeen myös pistetulo (b . b.)**
**Muutetaan saatu kulma-arvo (radiaaneissa) asteiksi:**
**Kulma_ast = Kulma_rad*360/ π**
**Lasketaan vektorin a projektio vektorilla b: ab=  (a . b)/(b . b) * b.**
**Lopuksi tarkistetaan vektoreiden yhtäsuuruus:**
**Vektorit a ja b ovat yhtäsuuria, jos det [(x1 y1), (x2 y2)] T = 0 AND |a| = |b|.**

Suoritetaan vielä joitakin operaatioita *avaruusvektoreille* a, b, c, jotka ovat muotoa

$$a = x_1 i + y_1 j + z_1 k$$
$$b = x_2 i + y_2 j + z_2$$
$$c = x_3 i + y_3 j + z_3 k$$

Vektoreiden a ja b *ristitulo* saatiin siis seuraavasti:

$$a \times b = |\ (i,j,k),\ (x_1,\ y_1,\ z_1),(x_2,y_2,\ z_2)|^T\ ,\ \text{josta saadaan}$$
$$a \times b = |\ (y_1,\ z_1)\ (y_2,\ z_2)|^T\ i - |\ (x_1,\ z_1)\ (x_2,\ z_2)|^T\ j + |\ (x_1,\ y_1)\ (x_2,\ y_2)|^T\ k$$

Avaruusvektoreiden a, b, c *skalaarikolmitulo* muodostetaan vielä lopuksi:

$$a \times b \cdot c = |\ (x_1,\ y_1,\ z_1),(x_2,y_2,\ z_2),\ (x_3,y_3,\ z_3)|^T$$

Molemmat determinanttimuodot (2x2) ja (3x3) voidaan laskea tässä teoksessa toisaalla olevilla algoritmeilla.

# 4.5 Alkuluvut, jakojäännös, Pascalin kolmio, potenssi ja syt

Tähän lukuun on koottu muutamia kiinnostavia ja lähes klassisia aiheita, jotka kuitenkin ovat hyvin hyödyllisiä monessakin eri sovelluskohteessa.

## 4.5.1 Alkuluvut

Alkuluvuiksi kutsutaan kokonaislukuja, joita ei voida jakaa muilla luvuilla kuin omalla itsellään. Alkulukuja ovat siis kokonaisluvut: 2, 3, 5, 7, 9, 11, ... jne (yleensä luku 1 käsitetään myös alkuluvuksi). Alkulukuja hyödynnetään monissa yhteyksissä kuten esimerkiksi salausalgoritmeissa. Kun luku on suuri (esimerkiksi yli 100 numeroa), joudutaan luvun alkuluvuksi todentamiseen tekemään runsaasti työtä. Se, onko jokin luku alkuluku, voidaan tarkistaa useallakin eri menetelmällä. Seuraavassa on kaksi erilaista menettelytapaa:

**Tapa 1**: Ensiksikin, koska luku 2 on ainoa parillinen alkuluku, riittää, kun tarkastellaan parittomia lukuja. Seuraavaksi tutkitaan jaollisuutta: onko luku jaollinen luvuilla 3, 5, 7, 9, ..., n-2, jossa n on siis tarkastettava luku. Tarkastusalgoritmissa lähdetään luvusta 3, jota lisätään silmukassa aina kahdella yksiköllä. Tarkastus voidaan tehdä katsomalla, onko jakojäännös nolla tai onko tarkastettavan luvun ja jakajan osamäärä kokonaisluku.

**Tapa 2**: Tässä menettelyssä tehostetaan edellistä tapaa lähtien liikkeelle käänteisestä tapauksesta: jos luku ei ole alkuluku, se voidaan esittää tulona, jonka tekijät ovat suurempia kuin yksi. Siis tarkasteltava luku n = x*y, jos se ei ole alkuluku. Jos nyt x <= y, niin $x^2$ <= n ja x <= √n. Tarkasteltavan luvun jaollisuutta ei siis kannata tutkia pitemmälle kuin lähimpään parittomaan kokonaislukuun, joka on pienempi tai yhtäsuuri kuin √n. Jos tutkimme esimerkiksi, onko luku 197 alkuluku, voimme yksinkertaisesti hakea lähimmän parittoman kokonaisluvun, jonka neliö on lähinnä lukua 197. Tällainen luku on nyt 13, koska $13^2$ = 169 (lukua 14 ei hyväksytä ja luvun 15 neliö (225) menee yli 197:n).
Nyt siis riittää, että tutkitaan, onko luku 197 jaollinen luvuilla 3, 5, 7, 9, 11, 13 vai ei. Oikeastaan lukua 13 ei tietenkään tarvitsisi enää testata. Kyseessä on siis alkuluku.

Alkulukujen testaamisessa kannattaa vielä tietää, että, mikäli luku ei ole jaollinen jollakin testiluvulla, niin se ei ole jaollinen myöskään kyseisen testiluvun monikerroilla. Edellisessä esimerkissämme ei meidän siis olisi tarvinnut tehdä testaamista luvulla 9, koska testaaminen tapahtui jo luvulla kolme. Millainen jono muodostuu siis testiluvuista? Testilukuja ovat: 3, 5, 7, 11, 13, 17, 19, 23, 31, 37, 41, 43, 47, ...

Alkulukuja voidaan tulostaa esimerkiksi seuraavalla algoritmilla.

## Alkulukujen generointi:

```
#include <iostream.h>

void main()
{
    int luku;
    int maara;
    for (luku = 1; luku <= 30; luku++)
    {
        if (luku == 1)
            cout << "Luku " << luku << " on alkuluku\n";
        for (maara = 2; maara <= luku; ++maara)
            if (maara == luku)
                  cout << "Luku " << luku << " on alkuluku\n";
```

```
            else if (luku % maara == 0)
            break;     /*ei ole alkuluku*/
        }
    }
```

## 4.5.2 Jakojäännös

Seuraavalla algoritmilla voidaan korvata jakojäännös- ja jakolaskuoperaattorit. Algoritmi suorittaa kahden kokonaisluvun välisen jakolaskun ja kertoo jakojäännöksen kokonaislukuna.

### Jakojäännös:

```
#include <iostream.h>

void main()
{
int Jaettava, Jakaja, Kokonaisia, Jakojaannos;
    Kokonaisia = 0;
    cout << "Anna jaettava \n";
    cin >> Jaettava;
    cout << "Anna jakaja \n";
    cin >> Jakaja;
    Jakojaannos = Jaettava;
    while ((Jakojaannos - Jakaja) >= 0)
    {
        Jakojaannos = Jakojaannos - Jakaja;
        Kokonaisia = Kokonaisia + 1;
    }
    cout << "Jakojaannös on " << Jakojaannos << "\n";
    cout << "Kokonaisosa on " << Kokonaisia << "\n";
}
```

## 4.5.3 Luvun numeroiden lukumäärä

Joskus eri sovelluksissa on tarkistettava numeroiden lukumäärä luvussa. Algoritmi tähän on seuraavana.

### Luvussa olevien numeroiden määrä:

```
#include <iostream.h>

void main()
{

    int Lukumaara = 0;
    int Annettu_luku;
    cout << "Anna luku ";
    cin >> Annettu_luku;
    // Annetaan luku, jonka numeroiden maara lasketaan
    do
    {
        Annettu_luku = Annettu_luku / 10;
        Lukumaara = Lukumaara + 1;
    }
    while (Annettu_luku != 0);
    cout << "Antamassasi luvussa on " << Lukumaara << " nroa \n";

}
```

## 4.5.4 Pascalin kolmio

Pascalin kolmio antaa kertoimet korotettaessa kahden tekijän summaa kokonaislukupotensseihin. Laskettava lauseke on $(a + b)^n$. Kertoimia varten on kehitetty ns. Pascalin kolmio, joka on seuraavanlainen:

$(a + b)^0$ 1
$(a + b)^1$ 1 1
$(a + b)^2$ 1 2 1
$(a + b)^3$ 1 3 3 1
$(a + b)^4$ 1 4 6 4 1
.... ...

Matemaattisella kaavalla kertoimet saadaan seuraavasti:

$$(n/k) = n!/k!(n-k)!$$

Sekä kertoimet että tekijöiden a ja b eksponentit saadaan esille kaavalla:

$$SUM((n/k)a^{n-k} b^k$$

**Pascalin kolmion kertoimet:**

```
#include <iostream.h>
//(n/k) = n!/k!(n-k)!
int kertoma(int a);
void main()
{
int n, k;

n = 4;
for (int i = 0; i <= n; i++)
{
  int kerroin = (kertoma(n))/(kertoma(i) * kertoma(n-i));
  cout << kerroin << "\t";
}
}

int kertoma(int a)
{
  if ( a == 0) return 1;
  int j = 1;
  for (int i=1; i<=a; i++)
        j = j * i;
  return j;
}
```

## 4.5.5 Potenssiin korotus

Seuraavassa algoritmissa korotetaan kokonaisluku kokonaislukupotenssiin.

### Potenssiin korotus:

```
#include <iostream.h>
int korotus(int a, int b);
void main()
{
int x, y;
cout << "Anna kantaluku \n";  // kantaluku voi olla myös liukuluku
cin >> x;
cout << "Anna eksponentti \n";
cin >> y;
cout << "Luku " << x << " korotettuna potenssiin " << y << " on " <<
korotus(x,y);
}

int korotus(int a, int b)
{
    int laskuri;
```

```
    int tulos=1;
    if (b == 0)
        return(1);
    for (laskuri = 1; laskuri <= b; laskuri++)
        tulos = tulos * a;
    return(tulos);
}
```

Potenssiin korotus voidaan toteuttaa myös standardikirjastoa math.h käyttämällä, kun tiedetään että

$$x^y = e^{(y \ln x)}$$

Nyt sekä eksponenttina että kantalukuna voi olla liukuluku.

Seuraavana on ohjelma, joka hyödyntää yllä olevaa kaavaa.

## Potenssiin korotus:

```
#include <iostream.h>
#include <math.h>

double korota(double x, double y)
{
  return exp(y * log(x));
}

main()
{
double a=2.2, b=3.3;

cout << "2.2 pot 3.3 on " << korota(a,b);

return 0;
}
```

## 4.5.6 Kokonaislukujen suurin yhteinen tekijä (SYT)

Manuaalisesti laskemalla kahden kokonaisluvun SYT saadaan jakamalla molemmat luvut tekijöihin, jonka jälkeen tekijöistä haetaan yhteiset tekijät. SYT on hyvin tärkeä monissa yhteyksissä. Esimerkiksi salausmenetelmässä RSA (jonka otamme esille myöhemmissä luvuissa) haetaan lukuja, joilla **ei** saa olla yhteisiä tekijöitä.

Esimerkiksi kokonaislukujen 27 ja 6 SYT saadaan seuraavasti:

27 = **3***3*3
6 = 2***3**
SYT on siis **3**.

### SYT

```
#include <iostream.h>
int syt(int a, int b);
void main()
{
int x, y;
cout << "Anna eka luku \n";
cin >> x;
cout << "Anna toka luku \n";
cin >> y;
cout << "SYT on " << syt(x,y);
}

int syt(int a, int b)
{
int sy;
a = ( a < 0 ? (-1)*a: a);
b = ( b < 0 ? (-1)*b: b);
    while (a != b)
    {
        if (a > b) a = a-b;
        else b = b-a;
    sy = a;
    }
    return sy;

}
```

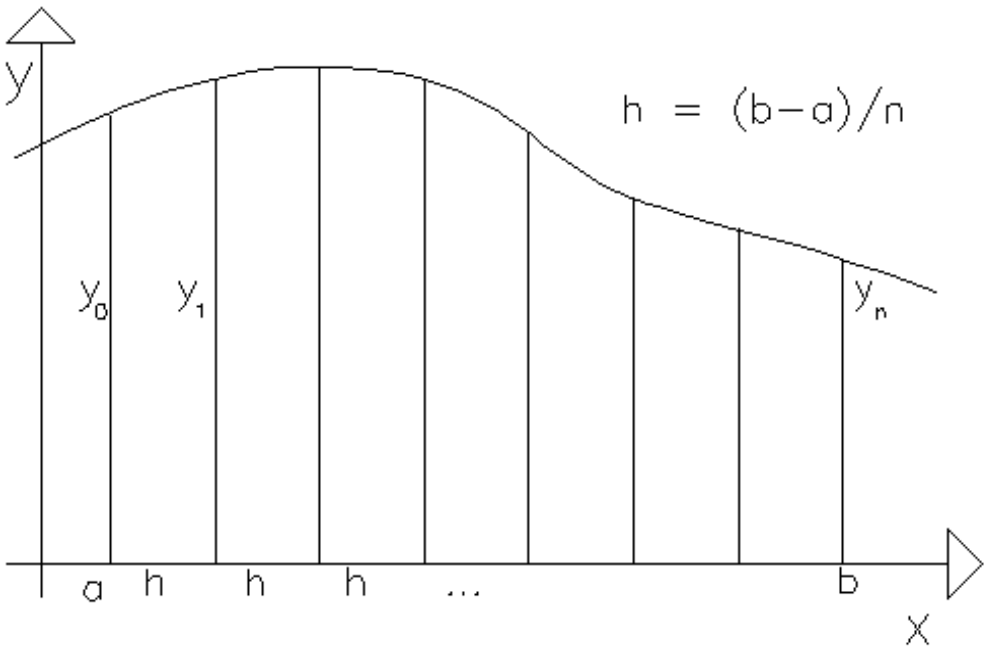
Kuten näemme, annetuista kokonaisluvuista otetaan itseisarvot, jolloin on mahdollista syöttää myös negatiivisia kokonaislukuja. SYT on tietenkin sama sekä negatiivisilla että positiivisilla kokonaisluvuilla.

**Simulaatio:**

Olkoon a = 27 ja b = 6
while 27 != 6
    if 27>6 a = 27-6;  (**a = 21**)
    if 21>6 a = 21-6;  (**a = 15**)
    if 15>6 a = 15-6;  (**a = 9**)
    if 9>6 a = 9-6;  (**a = 3**)
    if 3>6 b = 6-3;  (**b=3**)
    **a = b = 3 (while-ehto päättyy)**
**-->SYT = 3**
LOPPU

# 4.6 Tasokuvion pinta-alan likiarvo

Jos meillä on tasokuvio, jota x- ja y-akselien lisäksi rajoittaa jokin n-asteinen funktio, voimme laskea pinta-alan likiarvon usealla eri tavalla. Otamme tässä esille kaksi menettelyä: puolisuunnikaskaavan ja ns. Simpsonin kaavan. Allaolevasta nähdään sekä tilanne että käytettävät parametrit:



## 4.6.1 Puolisuunnikaskaava

Jos funktio y = f(x) rajoittaa x-y-koordinaatistossa alueen, saadaan alueen pinta-alan likiarvo puolisuunnikaskaavalla seuraavasti:

$$A(n) = h( 1/2y_0 + y_1 + y_2 + ... + y_{n-1} + 1/2y_n)$$

Yleensä (etenkin manuaalisessa laskennassa) on osavälejä kolme, jolloin kaavaa kutsutaan trapetsikaavaksi. Tällöin siis alue saadaan kaavasta:
$A(3) = h(1/2y_0 + y_1 + y_2 + 1/2y_3)$

Kehittelemme seuraavana likiarvon laskenta-algoritmia esimerkin valossa käyttäen trapetsikaavaa.
Olettakaamme, että rajoittava funktio on muotoa $y = f(x) = 2x^2 + x +1$ ja pinta-ala haluttaisiin saada aikaan väliltä $1 <= x <= 3$. Matemaattisesti tämä saataisiin integraalilla:

$$A = \int_1^3 ( x^3 + 2x^2 -1)\, dx$$

Kun integroitava funktio on positiivinen koko intergoimisvälillään, voidaan trapetsikaavalla hakea pinta-alan likiarvoa seuraavalla algoritmilla:

### Tasokuvion pinta-ala (puolisuunnikaskaava):

```
Määritellään ensin funktio: Nyt f(x) = 2x2 + x +1
Annetaan integrointirajat: siis a ja b, nyt a = 1, b = 3
Annetaan osavälien lukumäärä n (trapetsikaavassa n = 3)
Lasketaan osavälin leveys: h = (b-a)/n
Lasketaan 1/2y0 ja 1/2y3 eli funktion arvot/2 x:n arvoilla 1 ja 3 ja
lisätään summa-muuttujaan
Lasketaan silmukassa muut arvot eli i = 1 to n-1
    summa = summa + f(a + i*h)
    next i
Tulostetaan
```

Muuttamalla n:n arvoa suuremmaksi voidaan likiarvon tarkkuutta parantaa. Lukija voi verrata algoritmin tulosta matemaattisesti laskettuun tulokseen. Tietokoneella laskenta tulee sitä hyödyllisemmäksi mitä vaikeammin integroitavasta funktiosta on kysymys.

**Seuraavana on ohjelma, joka laskee integraalin:**

```
#include <iostream.h>
#include <math.h>

double arvio(float (*funktio)(float x), double a, double b, long int
n);
double integraali(float (*funktio)(float x), double a, double b,
double tarkkuus = 0.001);

float alkufunktio(float x)
{
  return x*x*x;
}


main()
{
double a = 2;
double b = 4;

cout << "Funktiosi integraali valilta " << a << " ja " << b;
cout << " on " <<  integraali(alkufunktio, a, b) << "\n"   ;
}


double integraali(float (*funktio)(float x), double a, double b,
                      double tarkkuus)
{

  long int n=1;
  double edellinen = arvio(funktio, a, b, n),
            nykyinen  = arvio(funktio, a, b, n = 2*n);

  while(fabs(nykyinen-edellinen) > tarkkuus)
  {
     edellinen = nykyinen;
     n = 2*n;
     nykyinen = arvio(funktio,a,b,n);
  }
  return nykyinen;
}

double arvio(float (*funktio)(float x), double a, double b, long int
n)
{
  double h = (b-a)/n;
  double s = (funktio(a)+funktio(b))/2;
  for (long int i=1; i<=n-1; i++)
     s = s + funktio(a+i*h);
```

```
    return s*h;
}
```

## 4.6.2 Simpsonin kaava pinta-alan likiarvon laskemiseksi

Simpsonin kaavassa on arvon n oltava parillinen.

Simpsonin kaava on muotoa:

$$A(n) = h/3(y_0 + 4y_1 + 2y_2 + 4y_3 + 2y_4 \ldots + 2y_{n-2} + 4y_{n-1} + y_n)$$

Likiarvon laskenta-algoritmi on helppo johtaa puolisuunnikaskaavan algoritmista. Menettelyjen tarkkuutta voi lukija vertailla. Selvää tietenkin on, että osavälien lähestyessä ääretöntä myös likiarvot lähestyvät samaa arvoa.

Molempia yllä esiteltyjä likiarvokaavoja voidaan käyttää myös tilavuuslaskuissa. Tällöin $y_i$:t korvataan pinta-alasiivuilla.

# 4.7 Murtoluvut ja polynomit

Murtoluku (rationaaliluku) on likiarvoa tarkempi, joten sen käsittelyn osaaminen ennen mahdollisia pyöristelyjä on hyödyllistä. Tässä aliluvussa käytetään myös hyödyksi syt-funktiota, jolla operaatioiden tulokset sievennetään.

## 4.7.1 Murtoluvut

Seuraavissa murtolukualgoritmeissa käytämme SYT-laskentaa hyödyksi. Murtoluvut voidaan tallentaa esimerkiksi luokkaan, jolloin rakenne voitaisiin määritellä seuraavasti:

**Murtoluku-luokka:**

```
class Murtoluku
{
int jakaja;
```

```
int jaettava;
public:
Murtoluku();
Murtoluku(int a, int b){jakaja = b;jaettava = a;}

friend main();
friend Murtoluku operator /(Murtoluku& c1, Murtoluku& c2);
friend Murtoluku Sievenna(Murtoluku & c1);
};
```

Kaksi murtolukua summataan (ja vähennetään toisistaan) manuaalisesti sieventämällä ensin molemmat murtoluvut, jonka jälkeen lukujen nimittäjäksi haetaan nimittäjien pienin yhteinen jaettava ja kerrotaan osoittaja. Tämän jälkeen osoittajille tehdään laskentaoperaatio ja lopuksi mahdollisesti sievennetään tulosta.

Hyödyntämällä SYT-operaatiota, voidaan ohjelmallisesti tehdä sieventäminen vasta lopputulokselle. Kahden murtoluvun yhteenlaskussa tuloksen osoittaja ja nimittäjä voidaan laskea ensin seuraavasti:

### Murtolukujen yhteenlasku:

```
tulos.osoittaja = luku1.osoittaja * luku2.nimittäjä +
luku2.osoittaja * luku1.nimittäjä
tulos.nimittäjä = luku1.nimittäjä * luku2.nimittäjä
Sievennys tapahtuu laskemalla:
tulos.osoittaja/SYT(tulos.osoittaja, tulos.nimittäjä)
tulos.nimittäjä/SYT(tulos.osoittaja, tulos.nimittäjä)
```

Kokeilemme manuaalisesti:

Olkoot murtoluvut: 1/6 ja 3/4
osoittaja = 1*4 + 3*6 = 22
nimittäjä = 6 * 4 = 24
tulos = 22/24, jota on siis sievennettävä

Sieventäminen tapahtuu hakemalla ensin SYT(22, 24), joka on siis **2**.
Osoittaja ja nimittäjä voidaan nyt jakaa luvulla 2, jolloin tulos on 11/12.

## Murtolukujen erotus

```
Erotus saadaan edellisen mukaan:
tulos.osoittaja = luku1.osoittaja * luku2.nimittäjä -
luku2.osoittaja * luku1.nimittäjä
tulos.nimittäjä = luku1.nimittäjä * luku2.nimittäjä
Lopuksi sievennetään kuten edellä.
```

## Kahden murtoluvun tulo

```
Murtolukujen tulo lasketaan kertomalla osoittajat keskenään ja
nimittäjät keskenään, eli
tulo.osoittaja = luku1.osoittaja * luku2.osoittaja
tulo.nimittäjä = luku1.nimittäjä * luku2.nimittäjä
Lopuksi sievennetään kuten edellä.
```

## Kahden murtoluvun osamäärä

```
Osamäärä saadaan seuraavasti:
osam.osoittaja = luku1.osoittaja * luku2.nimittäjä
osam.nimittäjä = luku1.nimittäjä * luku2.osoittaja
Lopuksi sievennetään kuten edellä.
```

## Seuraavana on oliopohjainen esimerkki:

```
#include <iostream.h>

class Murtoluku
{
int jakaja;
int jaettava;
public:
Murtoluku();
Murtoluku(int a, int b){jakaja = b;jaettava = a;}

friend main();
friend Murtoluku operator /(Murtoluku& c1, Murtoluku& c2);
friend Murtoluku Sievenna(Murtoluku & c1);
};

Murtoluku::Murtoluku()
{
jakaja = 0;
```

```
jaettava = 0;
}

Murtoluku Sievenna(Murtoluku & c1)
{
int syt;
int a = c1.jakaja;
int b = c1.jaettava;
      while (a != b)
            {  if (a > b) a = a-b;
              else b = b-a;
              syt = a;
            }
    c1.jakaja = c1.jakaja/syt;
    c1.jaettava = c1.jaettava/syt;
//  return c1;
}

Murtoluku operator /(Murtoluku& c1, Murtoluku& c2)
{
Murtoluku tulos;
tulos.jakaja = c1.jaettava * c2.jakaja;
tulos.jaettava = c2.jaettava * c1.jakaja;
return tulos;
}


main()
{
int p;
Murtoluku Luku1(2,3);
Murtoluku Luku2(5,6);
Murtoluku Luku3 = Luku1/Luku2;
cout << "3. murtoluvun jakaja on " << Luku3.jakaja << "\n";
cout << "3. murtoluvun jaettava on " << Luku3.jaettava << "\n";

Sievenna(Luku3);
cout << "3. murtoluvun jakaja on " << Luku3.jakaja << "\n";
cout << "3. murtoluvun jaettava on " << Luku3.jaettava << "\n";


return 0;
}

tulos.jakaja = c1.jaettava * c2.jakaja;
tulos.jaettava = c2.jaettava * c1.jakaja;
return tulos;
}
```

## 4.7.2 Polynomit

Polynomin normaalimuoto on $A_nx^n + A_{n-1}x^{n-1} + ... + A_1x^1 + A_0x^0$
Viimeinen termi $A_0x^0$ kuvaa vakiotekijää. Polynomi voi siis olla esimerkiksi lauseke $2x^3 + 3x^2 + x - 8$.

**Polynomien yhteenlasku**
Kaksi polynomia lasketaan yhteen summaamalla samanasteisten x-arvojen kertoimet. Ellei samaa astetta löydy kahdesta polynomista kuin yksi tekijä, lisätään kyseinen tekijä summapolynomiin.

Esimerkiksi: $(2x^3 + 3x^2 + x - 8) + (x^4 + 2x^2 + x) = x^4 + 2x^3 + 5x^2 + 2x - 8$

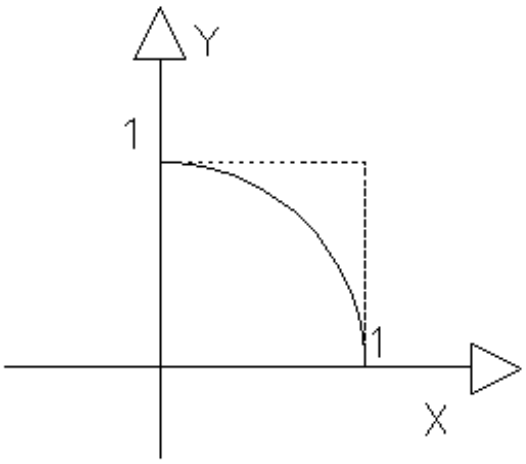
Kumpikin yhteenlaskettava polynomi tallennetaan kahteen tiedostoon tai listaan, joissa kerroin ja aste ovat alkion kenttinä. Molempia polynomeja verrataan toisiinsa järjestyksessä ja haetaan samaa astetta olevia tekijöitä. Ellei paria löydy, lisätään tekijä summapolynomiin suoraan ja siirrytään ensimmäisessä polynomissa yksi tekijä eteenpäin. Jos pari löytyy, summataan kertoimet ja lisätään summa ja aste summapolynomiin.

Edellä tulee samalla lasketuksi myös polynomien erotus, koska polynomien tekijöiden etumerkki voi olla positiivinen tai negatiivinen.

# 4.8 Likiarvo π:lle satunnaisluvuilla

Kuten tiedämme, ympyrän ala = π x $R^2$. Kun säde on yksi, puhumme yksikköympyrästä. Yksikköympyrän ala on siis = π.
Ympyrän yhtälö on $R^2 = X^2 + Y^2$. Kun tiedämme, että R = 1, voimme ratkaista yhtälöstä Y:n, eli saamme $Y = +-(1 - X^2)^{1/2}$.

Seuraavassa kuvassa yksikköympyrän se neljännes, jossa sekä X että Y saavat positiivisia arvoja, on piirretty neliön sisään. Jos nyt generoimme suuren määrän pisteitä väliltä 0...1, voimme ajatella, että tietty osa pisteistä sattuu ympyrän neljänneksen alueelle, kun taas loput osuvat neliön alueelle. Neliön ala on siis tässä tapauksessa 1 x 1 = 1 yksikköä.




Kun nyt generoimme satunnaislukuja, saamme ympyrän alan eli siis π:n selville seuraavasti:

π = generoitujen, ympyrän neljänneksen sisälle osuneiden pisteiden suhde kaikkiin pisteisiin x 4.

Jos satunnaisluku Y on suurempi tai yhtäsuuri kuin neljännesympyrää kuvaavan yhtälön oikea puoli eli $(1 - X^2)^{1/2}$ , niin Y ei ole ympyrän sisäpuolella. Muutoin taas tilanne on päinvastoin. Näin saadaan laskettua satunnaislukujen määrät ja lopulta laskettua likiarvo π:lle.