1 Taulukot ja lukujoukot

Tässä luvussa käsitellään pääosin matriisilaskentaa. Luvussa käsitellään determinantin, käänteismatriisin sekä transponoidun matriisin laskemisalgoritmit. Matriisilaskentaa sovelletaan yhtälöryhmien ratkaisemisessa. Luvun viimeisinä aiheina ovat dynaamisen taulukon kehittäminen luokan avulla sekä turvallisen vektorin muodostaminen.

1.1 Matriisin laatiminen

Luomme ensin matriisirakenteen (esimerkiksi 2-ulotteinen).

```
Matriisi:

// Matriisin alkioiden arvot sijoitetaan sisäkkäisissä silmukoissa:

for (i = 0; i < m; i++)  // i on rivi-indeksi
    for (j = 0; j < n; j++)  // j on sarakeindeksi
        matriisi[i][j] = 0;  // nollataan alkiot
```

Seuraavana on koko ohjelma:

```
#include <iostream.h>
main()
{
int i, j;
const int m = 3;
const int n = 2;
int matriisi[m][n];
// Matriisin alkioiden arvot sijoitetaan
// sisäkkäisissä silmukoissa:
for (i = 0; i < m; i++)
        for (j = 0; j < n; j++)
            matriisi[i][j] = 0;
    // Matriisin tulostus
for (i = 0; i < m; i++)
        for (j = 0; j < n; j++)
            cout << matriisi[i][j] << "\n";
}
```

Tässä vielä matriisimuotoon jäsennelty tulostus:

```
// Matriisin tulostus
for (i = 0; i < m; i++)
      {
         for (j = 0; j < n; j++)
             cout << matriisi[i][j] << "\t";
      cout << "\n";
      }
```

1.2 Laskutoimituksia matriiseilla

Tässä aliluvussa käsittelemme matriisilaskentaa: yhteenlasku, kertominen, determinantti, transponointi ja käänteismatriisi.

1.2.1 Matriisien yhteenlasku

Kaksi matriisia, A ja B, voidaan laskea yhteen vain, jos ne ovat samaa kertalukua eli niissä on yhtä monta riviä ja saraketta. Olkoot seuraavassa molemmat matriisin esimerkiksi kertalukua m x n.

Matriisien summa on tekijämatriisien vastinalkioiden summa.

Matriisi C = A + B ja cij = aij + bij; i=1,...,m; j=1,...,n

Matriisit on tietenkin ensin perustettava ja yhteenlaskettavat matriisit myös täytettävä arvoilla.

Matriisien yhteenlasku:

```
for (i = 0; i < m; i++)
        for (j = 0; j < n; j++)
            C[i][j] = A[i][j] + B[i][j];
```

1.2.2 Matriisin kertominen vakiolla

Matriisi kerrotaan vakioluvulla siten, että jokainen alkio kerrotaan erikseen kyseisellä luvulla.

Matriisin kertominen vakiolla:

```
for (i = 0; i < m; i++)
        for (j = 0; j < n; j++)
            matriisi[i][j] = k * matriisi[i][j];
```

1.2.3 Matriisien tulo

Kahden matriisin, jotka ovat muotoa A_{mxn} ja B_{nxp}, tulomatriisi C on tyyppiä m x p.

Tulomatriisin C alkiot saadaan kertomalla matriisi B vaakariveittäin matriisin A pystyriveillä. Eli kaavana

$$C_{ij} = \sum_{k=1}^{n} a_{ik} b_{kj}$$

Matriisien tulo:

```
#include <iostream.h>
main()
{
int i, j;
const int m = 3;
const int n = 2;
const int p = 3;
int k;
int Matriisi1[m][n];
int Matriisi2[n][p];
int Tulo[m][p];

// Matriisin arvot sijoitetaan sisäkkäisissä silmukoissa:
for (i = 0; i < m; i++)
        for (j = 0; j < n; j++)
            Matriisi1[i][j] = i+j;
```

```
for (i = 0; i < n; i++)
        for (j = 0; j < p; j++)
            Matriisi2[i][j] = i * j;

for (i = 0; i < m; i++)
        for (j = 0; j < p; j++)
            Tulo[i][j] = 0;


// muotoa Am x n ja Bn x p tulomatriisi C on tyyppiä m x p.
for (i = 0; i < m; i++)
    for (j = 0; j < p; j++)
        for (k = 0; k < n; k++)
         Tulo[i][j] = Tulo[i][j] + Matriisi1[i][k] *
    Matriisi2[k][j];

// Tulostus
int a = 0;
for (i = 0; i < m; i++)
        for (j = 0; j < n; j++)
            {
            a++;
                cout << Matriisi1[i][j] << "\t";
                if (a % 2 == 0) cout << "\n";
            }

cout << "\n\n";
a = 0;
for (i = 0; i < n; i++)
    for (j = 0; j < p; j++)
    {
        a++;
        cout << Matriisi2[i][j] << "\t";
        if (a % 3 == 0) cout << "\n";
    }

cout << "\n";


for (i = 0; i < m; i++)
    for (j = 0; j < p; j++)
        cout << Tulo[i][j] << "\t";

}
```

1.2.4 Determinantti

Matriisin determinantti on hyvin tärkeä monissa eri yhteyksissä. Determinantin avulla voidaan tehdä matriisioperaatioita ja esimerkiksi ratkaista erilaisia yhtälöryhmiä. Determinantti voidaan laskea vain neliömatriisille.

Determinantti merkitään det A tai | A |.

Tarkastelemme seuraavana eri tapoja laskea matriisin determinantti. Tutustumme ensin permutaatio-käsitteeseen, koska permutaation avulla saamme määriteltyä matriisin determinantin.

Permutaatio

Permutointi tarkoittaa jonon asettamista eri järjestykseen. Esimerkiksi luvut 1, 2 ja 3 voidaan esittää seuraavina permutaatioina:

p(1, 2, 3),
p(2, 3, 1),
p(3, 1, 2),
p(1, 3, 2),
p(3, 2, 1) ja
p(2, 1, 3).

Eri järjestyksessä olevia jonoja tuli siis kuusi kappaletta, joka saadaan myös käyttämällä kertomaa, eli määrä on 3! (=1 * 2 * 3 = 6). Permutaatio on keskeisessä asemassa laskettaessa neliömatriisin determinanttia.

Determinantti voidaan laskea permutaatioita hyödyntäen seuraavasti:

$$\det A = \sum p((j_1, ..., j_n)a_{1j1} * a_{2j2} * ...* a_{njn}),$$
jossa $(j_1, ..., j_n)$ kulkee kaikki N_n:n permutaatiot.

Indeksi j kuvaa matriisien pystyrivejä.

Katsomme laskentatapaa esimerkin valossa: 2 x 2 -matriisin determinantti lasketaan seuraavasti:

Olkoon matriisi A kuvattuna seuraavasti:

a_{11}	a_{12}
a_{21}	a_{22}

det A = a_{11}* a_{22} - a_{21} * a_{12}

Ylläoleva tapa laskea 2x2-matriisin determinantti on helposti muistettava menettely. Jos katsomme laskentatapaa ylempänä olevan permutaatiokaavan valossa, saamme j:n arvoksi 2.

Jono muodostuu siis luvuista 1 ja 2, jotka voidaan asettaa järjestykseen
p(1, 2) ja p(2, 1).

Tällöin siis permutaatio p(1,2) (pystyrivien arvot j_1 ja j_2) näkyy 2 x 2 -matriisin determinantin laskussa alkioiden a_{11} ja a_{22} pystyrivin indeksinä. Toinen mahdollinen permutaatio eli p(2,1) näkyy seuraavan summattavan tulon (a_{21} * a_{12}) alkioiden pystyrivin indeksinä. Mutta mistä tulikaan miinusmerkki?

Se, onko ylläolevan determinantin laskukaavan summattavien tekijöiden etumerkki positiivinen tai negatiivinen, riippuu ns. parillisuusindeksistä eli siitä, onko permutaation inversioiden määrä parillinen tai pariton (permutaatiossa (j_1, j_2, ..., j_n) oleva pari muodostaa inversion, jos $j_s > j_t$ ja s<t)). Jos inversioiden määrä on parillinen, on etumerkki + ja määrän ollessa pariton, etumerkki on -. Esimerkiksi permutaatioiden p(1, 2) ja p(2, 1) kohdalla nähdään, että permutaation p(1, 2) inversioiden määrä on nolla ja permutaation p(2, 1) taas 1 eli pariton määrä, joka selittää 2 x 2 -matriisin determinantin laskennassa käytetyn negatiivisen etumerkin.

Otamme vielä kaavan sovellusesimerkiksi 3 x 3 -matriisin determinantin laskemisen, johon on kehitetty myös näppärä muistisääntö.

Jos matriisi A on muotoa

a_{11}	a_{12}	a_{13}
a_{21}	a_{22}	a_{23}
a_{31}	a_{32}	a_{33}

on det A = a_{11} a_{22} a_{33} + a_{12} a_{23} a_{31} + a_{13} a_{21} a_{32} - a_{11} a_{23} a_{32} - a_{13} a_{22} a_{31} - a_{12} a_{21} a_{33}

Perusteluina lausekkeelle haemme ensin kaikki permutaatiot. Pystyrivejä ovat siis rivit 1,2 ja 3.

Nämä rivit voidaan asettaa 3! eli kuuteen eri järjestykseen, jotka ovat:

1	2	3
2	3	1
3	1	2
1	3	2
3	2	1
2	1	3

Taulukosta nähdäänkin nyt determinantin lausekkeessa esiintyvien tulojen alkioiden pystyrivien indeksit. Indeksit ovat nyt samassa järjestyksessä kuin determinantin laskulausekkeen alkioissa. Kolmen viimeisen tulotekijän etumerkki on negatiivinen, mikä johtuu siitä, että permutaatioiden p(1, 3, 2), p(3, 2, 1) ja p(2, 1, 3) inversioiden määrä on pariton. Esimerkiksi permutaation p(1, 3, 2) inversioita on yksi kappale eli (3, 2) ja permutaation p(3, 2, 1) taas 3 kappaletta eli (3, 2), (3, 1) ja (2, 1).

Kun matriisin koko kasvaa, on muistisääntöjen tai manuaalisen laskennan suorittaminen determinantin hakemiseksi liian työlästä.

Seuraava algoritmi kuvaa determinantin laskentaa edellä selvitetyllä menettelyllä:

Matriisin n x n determinantin laskeminen:

```
Oletetaan, että matriisi (n x n) on jo muodostettu.
Annetaan neliömatriisin koko n
Varataan tilaa permutaatioille ja parillisuusindeksille (voi olla
myös true/false)
Muodostetaan permutaatiot
Tutkitaan parillisuusindeksi eli inversioiden määrä
Muodostetaan determinantin kukin summatekijä kehittämällä alkioiden
tulo siten, että alkion vaakarivin indeksi on 1...n ja pystyrivin
indeksi vuorotellen kukin permutaatio. Tulontekijän etumerkki
määräytyy parillisuusindeksistä, ja etumerkki voidaan ottaa samasta
taulukosta, jossa permutaatiot ovat (yksi sarake ko. taulukossa).
Suoritetaan laskenta ja tulostetaan determinantti.
```

Ennen laskentaa tai tietojen syöttämistä matriisiin kannattaa myös tarkistaa, onko matriisissa kahta identtistä riviä, jolloin determinantin arvo on nolla.

Matriisin determinantti voidaan laskea myös *alideterminanttien* avulla, mikä saattaa olla kätevä keino erityisesti silloin, kun päämatriisi on 4 x 4 -matriisi. Tällöin nimittäin alideterminantti on muotoa 3 x 3 ja se on vielä järkevää laskea manuaalisesti. Emme tässä ota tätä menettelyä sen tarkemmin esille, koska kaavan

mukaan on joka tapauksessa pystyttävä laskemaan (tässä tapauksessa alemman kertaluvun matriisin) determinantti erikseen.

Samantapainen menettely kuin edellä on myös determinantin laskeminen *matriisin komplementin* avulla. Determinantti voidaan lisäksi laskea myös ottamalla matriisin mikä tahansa rivi ja kertomalla rivin alkiot vastaavilla komplementeilla ja summaamalla tulot. Eli det $A = \sum a_{ik}A_{ik}$. Lauseke vaikuttaa yksinkertaiselta, mutta erityisesti suuremman kertaluvun matriisien komplementtien hakeminen on työlästä.

Lineaarisen yhtälöparin ratkaisu determinantin avulla

Puhuimme aiemmin determinanttien laajasta käyttöalueesta. Voimmekin nyt ottaa esille lineaarisen yhtälöparin ratkaisemisen determinantin avulla.

Olettakaamme, että yhtälöpari on muotoa

$$a_1x + b_1y = c_1$$
$$a_2x + b_2y = c_2$$

Yhtälöparin ratkaisussa sijoitetaan yhtälöparin yhtälöiden x:n ja y:n kertoimet sekä vakiot matriiseihin, joiden determinantteja hyödynnetään sitten yhtälöparin ratkaisussa.

Merkitsemme tässä determinanttia pelkällä D-kirjaimella, jolloin D_x, D_y ja D määrittelevät seuraavat yhtälöparista saatavien matriisien determinantit:

D on determinantti matriisista, joka sisältää seuraavat yhtälöparista saatavat arvot:

a_1	b_1
a_2	b_2

D_x on determinantti matriisista:

c_1	b_1
c_2	b_2

D_y on determinantti matriisista:

a_1	c_1
a_2	c_2

Jos determinantti D<>0, saadaan yhtälöparin ratkaisu determinanteista seuraavasti:
$x = D_x/D$ ja $y = D_y/D$

Yhtälöparin ratkaiseminen:

```
Muodostetaan matriisit A, B ja C
Tarkistetaan, onko matriisin A determinantti D <> 0.
Jos kyllä, niin
    Lasketaan matriisien B ja C vastaavat determinantit Dx ja Dy
    Lasketaan x ja y
```

1.2.5 Matriisin transponointi ja käänteismatriisi

Matriisin transponointi

Matriisin transponointi tapahtuu siten, että matriisin pystyrivit sijoitetaan transponoidun matriisin vaakariveiksi. Jos siis alkuperäinen matriisi A on muotoa m x n, sen transpoosi on muotoa n x m ja merkitään A^T.

Esimerkiksi matriisin

1	2
3	4

transpoosi on

1	3
2	4

Matriisin transponoinnissa on alkuperäisen matriisin (muotoa m x n) luomisen jälkeen esiteltävä toinen matriisi, joka on muotoa n x m. Tämän jälkeen alkuperäisen matriisin pystyrivien alkiot siirretään transponoidun matriisin vaakarivien alkioiksi.

Matriisin transponointi:

```
for (i = 0; i < m; i++)
    for (j= 0; j < n; j++)
        B[i][j] = A[j][i];
```

Tässä yhteydessä voimme ottaa myös esille matriisin symmetrisyyskäsitteen. Matriisi on symmetrinen, jos $A = A^T$. Tämä voidaan tarkistaa vertailemalla näiden kahden matriisin alkioita keskenään.

Käänteismatriisi

Jos kahden matriisin (A ja B) tulona (A*B ja B*A) syntyy yksikkömatriisi (I), sanotaan matriisia B matriisin A käänteismatriisiksi ja merkitään $A^{(-1)}$. Yksikkömatriisi on matriisi, jonka lävistäjäalkiot ovat ykkösiä ja kaikki muut alkiot nollia. Mikäli johonkin tarkoitukseen tarvitaan vain käänteismatriisin determinantti, on hyvä ennen käänteismatriisin laskemista tietää, että käänteismatriisien determinantit ovat toistensa käänteislukuja. Eli riittää, kun laskee alkuperäisen matriisin determinantin ja ottaa siitä käänteisluvun.

Käänteismatriisin muodostaminen on (ainakin manuaalisesti) melko työlästä, koska kukin käänteismatriisin alkio on laskettava erikseen.

Jos neliömatriisi A on muotoa n x n ja A_{ji} taas se (n-1)-rivinen matriisi, joka saadaan A:sta poistamalla siitä j:s rivi ja i:s sarake, niin A:n käänteismatriisin alkio saadaan kaavalla:

$$A^{(-1)}_{ij} = ((-1)^{(i+j)} * |A_{ji}|)/|A|$$

Kaavan jaettavassa oleva $|A_{ji}|$ on nimeltään alkuperäisen matriisin A komplementin transpoosi. Komplementtialkio A_{ij} saadaan siis ottamalla A:sta pois i:s vaakarivi ja j:s pystyrivi ja laskemalla syntyneen matriisin determinantti. Voidaan myös nähdä, että etumerkkitekijä $(-1)^{(i+j)}$ on positiivinen silloin, kun summa i + j on parillinen.

Edellä oleva kaava voidaan kirjoittaa muotoon:

$$A^{(-1)} = 1/\det A * (\text{A:n komplementti})^T$$

Eli kaavan mukaan lasketaan ensin matriisin A alkioiden komplementit ja transponoidaan kyseinen matriisi sekä jaetaan lopuksi det A:lla.
Katsotaan ensin tarkemmin matriisin alkioiden komplementtien laskemista esimerkin valossa:

Olkoon meillä matriisi A_{3x3} seuraavasti:

$A_{ij} =$

a_{11}	a_{12}	a_{13}
a_{21}	a_{22}	a_{23}
a_{31}	a_{32}	a_{33}

Jos matriisi A:n alkiot ovat seuraavat:

A=

3	2	2
1	3	1
1	2	2

niin komplementtialkiot (A_{ij}) saadaan seuraavasti:

A_{11} saadaan poistamalla matriisin A 1. pystyrivi ja 1. vaakarivi ja laskemalla syntyneen 2 x 2 -matriisin determinantti (D_{11}), jonka etumerkki määräytyy lauseesta $(-1)^{i+j}$ eli tässä tapauksessa lause on $(-1)^2 = +1$ (determinantin etumerkki on siis positiivinen).

Komplementtialkio A_{11} saadaan siis seuraavasti:

$A_{11} = +1 * D_{11}$, jossa

D_{11}= determinantti matriisista

3	1
2	2

eli nyt $D_{11} = 3 * 3 - 2 * 1 = 7$. Näin saatiin ensimmäinen komplementtialkio.

Vastaavasti lasketaan muut komplementtialkiot determinanttien D_{21}, D_{31}, D_{12}, D_{22}, D_{23}, D_{13}, D_{23} ja D_{33} kautta.
D_{ij}:n etumerkki on siis negatiivinen (-) silloin, kun indeksien i+j summa on pariton.

Käänteismatriisin muodostaminen:

Oletetaan, että matriisi A on muotoa n x n.

Lasketaan det A; jos det A = 0, käänteismatriisia ei ole; loppu; muuten lasketaan matriisin A komplementtialkiot seuraavasti:
Esitellään apumatriisi A_supp, joka on muotoa (n-1 x n-1) sekä n x n-matriisi, A_komp.
Sijoitetaan A_supp-matriisiin matriisin A ne pysty- ja vaakarivit, jotka eivät ole komplementtialkion indekseinä.

```
q = 1
p = 1 to n
    k = 1 to n
        r = 0
        i = 1 to n
            j = 1 to n
            if (j<>k)  r = r+1
                if (i<>p) or  )j <> k)
                    A_supp[q,r] = A[i,j]
                next j
                q = q+1
        next i
        Laske det A_supp
        Kerro det A_supp kertoimella (-1)p+k
        Sijoita tulos matriisiin A_komp alkioksi A_komp[p,k]
    next k
next p
```

Kyseistä menettelyä ei ainakaan manuaalisessa ratkaisemisessa ole järkevää käyttää kuin vain pienille matriiseille. Käsin laskentaa varten on suuremmille matriiseille kehitetty ns. ositusmenetelmä, jossa yksikkömatriisin (I) avulla ja käyttäen Gaussin periaatteita saadaan käänteismatriisi esille vähemmällä työllä.
Tietokoneella kuitenkin on ehkä helpompi kehittää algoritmi nimenomaan esitellylle menettelylle.

Seuraavana on vielä 'vippaskonsti' 2 x 2 -matriisin käänteismatriisin laskemiseen.

Olkoon meillä matriisi A, jonka alkiot ovat a...d seuraavasti:

a	b
c	d

Käänteismatriisi on:

1/det A *

d	-b
-c	a

Saamme siis yksinkertaisen algoritmin 2 x 2 -matriisin käänteismatriisin laskentaan:

2 x 2 -matriisin käänteismatriisi:

```
Esittele ensin matriisit A ja B  (molemmat 2x2-matriiseja)
Sijoita arvot matriisiin A
Laske dete = a11*a22 - a12*a21
Laske ja sijoita alkiot käänteismatriisiin B:
B(1,1) = 1/dete * a22
B(1,2) = -1/dete * a12
B(2,1) = -1/dete * a21
B(2,2) = 1/dete * a11
Tulosta käänteismatriisi B
```

Jos haluat tarkistaa 'vippaskonstin', laske A*B. Tuloksen on oltava 2 x 2 -yksikkömatriisi I, joka on siis tässä tapauksessa muotoa:

1	0
0	1

Käänteismatriisin tarkistus:

```
Esittele ensin 2 x 2 -matriisi I ja sijoita siihen kertolaskun
alkiot:
I(1,1) = A(1,1)*B(1,1) + A(1,2)*B(2,1)
I(1,2) = A(1,1)*B(1,2) + A(1,2)*B(2,2)
I(2,1) = A(2,1)*B(1,1) + A(2,2)*B(2,1)
I(1,1) = A(2,2)*B(1,2) + A(2,2)*B(2,2)
```

Koska 2 x 2 -matriisi on melkoisen yleisesti tarvittava muoto, simuloimme proseduuria vielä esimerkillä.

Olkoon matriisi A seuraavanlainen:

2	1
2	3

Det A = $a_{11}*a_{22} - a_{12}*a_{21} = 2*3 - 2*1 = 4$

Käänteismatriisin alkiot ovat:

B(1,1) = 1/dete * a_{22} = 1/4 * 3 = 3/4
B(1,2) = -1/dete * a_{12} = -1/4 * 1 = -1/4
B(2,1) = -1/dete * a_{21} = -1/4 * 2 = -1/2
B(2,2) = 1/dete * a_{11} = 1/4 * 2 = 1/2

Käänteismatriisi B, eli A^{-1} on siis:

3/4	-1/4
-1/2	1/2

Teemme vielä tarkistuksen, onko A*B = I:

I(1,1) = A(1,1)*B(1,1) + A(1,2)*B(2,1) = 2*3/4 + 1*(-1/2) = 1
I(1,2) = A(1,1)*B(1,2) + A(1,2)*B(2,2) = 2*(-1/4) + 1*1/2 = 0
I(2,1) = A(2,1)*B(1,1) + A(2,2)*B(2,1) = 2*3/4 + 3*(-1/2) = 0
I(1,1) = A(2,1)*B(1,2) + A(2,2)*B(2,2) = 2 *(-1/4) + 3 * 1/2 = 1

Kertolasku tuotti siis yksikkömatriisin

1	0
0	1

Vippaskonsti siis pätee.

Matriisin käänteismatriisi voidaan laskea myös LU-hajotelman kautta, mutta emme voi syventyä siihen tämän kirjan puitteissa, koska johdatus menettelyyn vie aivan liian paljon tilaa.

Lineaarisen yhtälöryhmän ratkaisu Cramerin säännöllä

Edellä ratkaisimme yhtälöparin determinanttien avulla. Käsittelemme tässä vielä asiaa yleisemmin, yhtälöryhmän ratkaisussa. Hyödynnämme seuraavana determinanttia lineaarisen yhtälöryhmän ratkaisussa.

Olettakaamme, että meillä on yhtälöryhmä:

$$a_{11}x_1 + a_{12}x_2 + ... + a_{1n}x_n = b_1$$
$$a_{21}x_1 + a_{22}x_2 + ... + a_{2n}x_n = b_2$$
$$...$$
$$a_{m1}x_1 + a_{m2}x_2 + ... + a_{mn}x_n = b_m$$

Yhtälöryhmä voidaan esittää matriisimuodossa:

$$Ax = b$$

Jos kerroinmatriisin determinantti det A <> 0, niin yhtälöryhmä voidaan ratkaista kaavalla:

$$x = A^{(-1)}b$$

Yhtälöryhmän ratkaisu löydettäisiin siis suoraan käänteismatriisin avulla.

Juuret saadaan myös esille kaavalla:

$x_j = D_j/detA$, jossa kaavassa

D_j = determinantti, joka saadaan, kun det A:n j:s pystyrivi korvataan b:n komponenteilla.

Yhtälöryhmän ratkaiseminen:

```
Muodostetaan yhtälöryhmän mukainen kerroinmatriisi A (muotoa n x n)
Muodostetaan pystyvektori b (muotoa n x 1)
Lasketaan det A
Korvataan silmukassa (k = 1 to n) kukin matriisin A pystyrivi (j)
vuorotellen vektorilla b ja lasketaan kunkin syntyneen matriisin
determinantti, Dj.
Lasketaan xj = Dj/detA
```

Simuloimme algoritmia seuraavalla esimerkillä:

Olkoon yhtälöryhmä seuraava:

$$2x + y + 2z = 0$$

$2x + z = -5$
$x + 2y + 2z = -3$

Kerroinmatriisi A on

2	1	2
2	0	1
1	2	2

b-arvoja kuvaava pystyvektori on:

0
-5
-3

Lasketaan det A: det A = +1
Lasketaan D_1, D_2 ja D_3
D_1 saadaan laskemalla determinantti matriisista, joka on saatu korvaamalla matriisin A 1. pystyrivi b:n komponenteilla (harmaa alue uudessa matriisissa).

D_1 on siis determinantti matriisista:

0	1	2
-5	0	1
-3	2	2

Saadaan $D_1 = -13$

Vastaavalla tavalla lasketaan D_2 ja D_3 .

$D_2 = -16$
$D_3 = 21$

Lopputulos on siis (det A oli siis +1):

$x = -13$
$y = -16$
$z = 21$

1.2.6 Dynaaminen taulukko

Taulukon rivien ja sarakkeiden määrän tulee perinteisesti olla vakioita. Kuitenkin on monesti edullista, jos ajon aikana voisimme määrittää taulukon koon. Tämä onnistuu, kun käytämme osoittimia ja struct-rakennetta; tällöin varaamme taulukolle muistia dynaamisesti.

C++-kielessä moniulotteinen taulukko on taulukoista koostuva taulukko. Esimerkiksi määrittely int a[2][4] kuvaa taulukkoa, jossa on 2 alkiota ja kukin noista 2:sta alkiosta sisältää 4-alkioisen taulukon. Edellä kerrottu selkiintyy, kun ajattelemme osoittimia ja taulukoita: taulukon nimi ilman hakasulkuja (tai jonkin hakasulkuparin puuttuessa) on osoitin taulukkoon.

Jos meillä on siis taulukko int a[2][4], on esimerkiksi

a[0] osoitin ensimmäisen 4-alkioisen taulukon alkuun (' 1. rivi ') ja
a[1] osoitin toisen 4-alkioisen taulukon alkuun (' 2. rivi ')

Samoin

a on osoitin koko taulukon alkuun ja samalla osoitin ensimmäisen 2-alkioisen taulukon alkuun

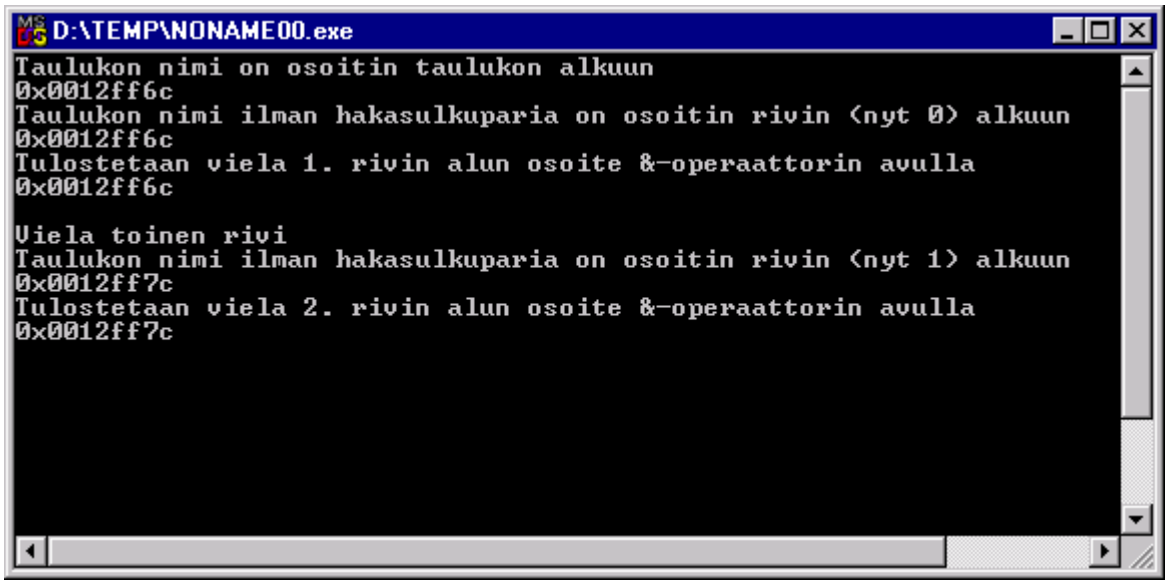
Seuraavassa on asian havainnollistamiseksi esimerkkiohjelma, joka tulostaa 2-ulotteisen taulukon rivien osoitteita sekä osoitinoperaattoria käyttäen että ilman sitä:

Taulukon nimi osoittimena:

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

main()
{
int i, j;
const int m = 2;
const int n = 4;
int matriisi[m][n];
for (i = 0; i < m; i++)
        for (j = 0; j < n; j++)
            matriisi[i][j] = 0;

cout << "Taulukon nimi on osoitin taulukon alkuun\n";
cout << matriisi << endl;
```

```
cout << "Taulukon nimi ilman hakasulkuparia on osoitin rivin (nyt 0)
alkuun\n";
cout << matriisi[0] << endl;
cout << "Tulostetaan vielä 1. rivin alun osoite &-operaattorin
avulla\n";
cout << &matriisi[0][0] << endl;
cout << "\n";
cout << "Vielä toinen rivi\n";
cout << "Taulukon nimi ilman hakasulkuparia on osoitin rivin (nyt 1)
alkuun\n";
cout << matriisi[1] << endl;
cout << "Tulostetaan vielä 2. rivin alun osoite &-operaattorin
avulla\n";
cout << &matriisi[1][0] << endl;

getch();
}
```

Tulostuksesta näkyy, kuinka osoitteet ovat rivikohtaisesti samoja:



Seuraavana on ohjelmaesimerkki, jossa varataan tila dynaamisesti 2-ulotteiselle taulukolle. Tietueen osoitin-jäsen osoittaa koko taulukon alkuun; se on kuitenkin osoittimen osoitin, koska se osoittaa osoittimiin, jotka taas osoittavat riveille.

Dynaaminen taulukko

```
#include <iostream.h>
#include <stdlib.h>
#include <conio.h>

struct dynTaulu
```

```
{
int ** kanta;
int rivimaara, sarakemaara;
};

void varaa_tila(int m, int n, dynTaulu & T);
void vapauta_tila(dynTaulu & T);
int suurin(dynTaulu & T);
void tulosta(const dynTaulu & T);

void main()
{
int i, j, m, n;
dynTaulu matr1;
cout << "Anna matriisin rivimaara\n";
cin >> m;
cout << "Anna matriisin sarakemaara\n";
cin >> n;
varaa_tila(m, n, matr1);
randomize();
for (i = 0; i < matr1.sarakemaara; ++i)
    for (j = 0; j < matr1.rivimaara; ++j)
        matr1.kanta[i][j] = rand() % 10;

tulosta(matr1);

int max = suurin(matr1);

cout << "Maksimiarvo on " << max << "\n";

getch();
}


void varaa_tila(int m, int n, dynTaulu & T)
{
T.kanta = new int* [n];
for (int i = 0; i < n; ++i)
    T.kanta[i] = new int[m];
T.rivimaara = m;
T.sarakemaara = n;
}


void vapauta_tila(dynTaulu & T)
{
for (int i = 0; i < T.sarakemaara; ++i)
    delete[] T.kanta[i];
delete[] T.kanta;
T.rivimaara = 0;
T.sarakemaara = 0;
```

```
}

int suurin(dynTaulu & T)
{
int maksimi = T.kanta[0][0];

for (int i = 0; i < T.sarakemaara; ++i)
    for (int j = 0; j < T.rivimaara; ++j)
        if (T.kanta[i][j] > maksimi)
            maksimi = T.kanta[i][j];
return (maksimi);
}

void tulosta(const dynTaulu & T)
{
for (int i = 0; i < T.sarakemaara; ++i)
    for (int j = 0; j < T.rivimaara; ++j)
        cout << "Sarakkeessa [" << i << "] rivilla [" << j << "] on
" << T.kanta[i][j] << "\n";

}
```

1.2.7 Turvallinen vektori

C-kieli ei tarkista taulukon rajojen ylittämistä. Kaiken lisäksi taulukon koko on annettava vakiona.
C++-kielessä voimme kehittää turvallisempia ja käytettävämpiä taulukoita. Tällöin sijoitamme taulukon luokan jäsenmuuttujaksi ja laadimme metodeita sekä operaattoreita, joiden avulla taulukon käyttö sujuu.

Seuraavassa on kuvattu vektoriluokka. Luokassa on käytetty assert.h-tiedoston assert-funktiota tarkistukseen. Voisimme tietysti myös käyttää virhevuota (cerr) tai muuta tapaa ilmaista virheelliset taulukon indeksit. Assert lopettaa ohjelman, jos varmistus ei anna positiivista tulosta. Vielä kätevämpää olisi käyttää poikkeuksia, jotka käsitellään hallitusti (try - throw - catch).

```
class vektori
{
public:
explicit vektori(int n = 5);
~vektori() { delete[] os; }
int & alkio(int i);
int yraja() const { return (koko - 1); }
private:
int * os;
```

```
int koko;
};
```

Muodostimen määrittely:

```
vektori::vektori(int n) : koko(n)
{
assert (n > 0);
os = new int[koko];
assert (os != NULL);
}
```

Indeksointimetodi:

```
int& vektori::alkio(int i)
{
assert (i >= 0 && i < koko);
return p[i];
}
```

Turvallinen vektori

```
#include <iostream.h>
#include <assert.h>
#include <conio.h>
#include <stdlib.h>

class vektori
{
public:
vektori(int n = 5);
~vektori() { delete[] os; }
int & alkio(int i);
int yraja() const { return (koko - 1); }
private:
int * os;
int koko;
};

//Muodostimen määrittely:

vektori::vektori(int n) : koko(n)
{
assert (n > 0);
os = new int[koko];
```

```
assert (os != NULL);
}

//Indeksointimetodi:

int& vektori::alkio(int i)
{
assert (i >= 0 && i < koko);
return os[i];
}


//Kokeilemme luokkaa:

void main()
{
vektori eka(3);
eka.alkio(0) = 100;
eka.alkio(1) = 99;
eka.alkio(2) = 1;

vektori toka(2);
toka.alkio(0) = eka.alkio(0) + eka.alkio(1);
toka.alkio(1) = eka.alkio(2) + 1;

cout << "eka.alkio(0) " << eka.alkio(0) << "\n";
cout << "toka.alkio(0) " << toka.alkio(0) << "\n";
cout << "toka.alkio(1) " << toka.alkio(1) << "\n";
cout << "\n";

// dynaaminen vektorin muodostaminen

int k;
cout << "Anna vektorin koko: "; cin >> k;

vektori kolkku(k);

for (int i = 0; i < k; i++)
    kolkku.alkio(i) = rand() % 10;

cout << "\n";

for (i = 0; i < k; i++)
  cout << kolkku.alkio(i) << "\t";


getch();
}
```

Voisimme nyt luoda esimerkiksi 5 x 5 -taulukon oletusmuodostimen avulla:

```
vektori vitoset[5];
```

Nyt esimerkiksi ilmaus

```
vitoset[0].alkio(0)
```

viittaa taulukon ensimmäisen rivin ensimmäiseen alkioon.

Taulukko-operaatioita Vektori-luokassa

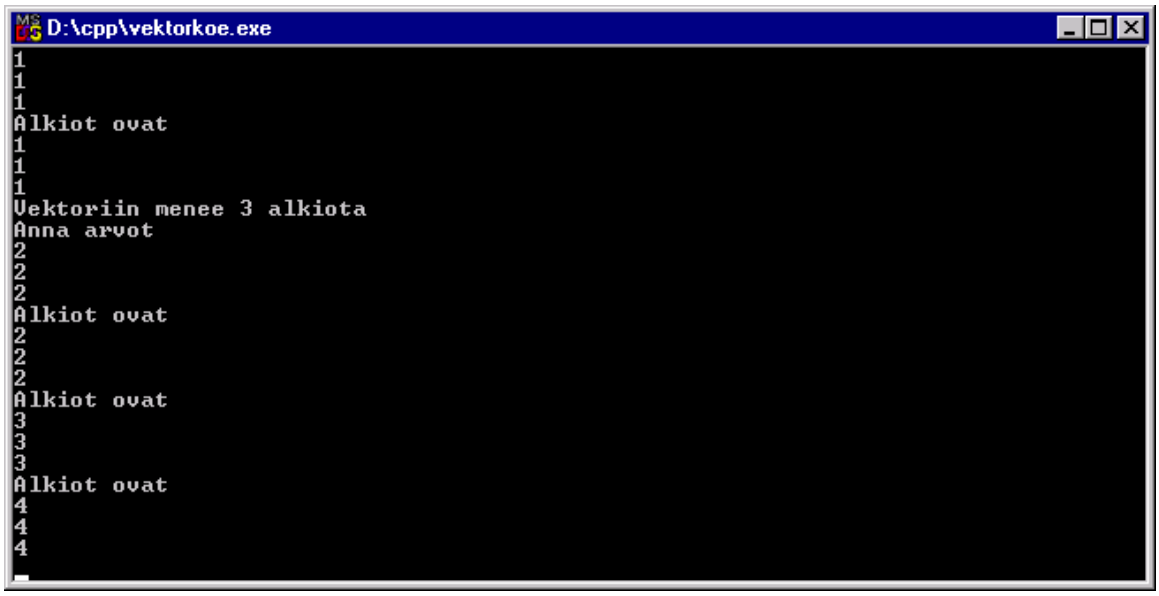
Seuraavana on esimerkki Vektori-luokan käsittelystä. Mukana ovat tulostuksen ja syötön ylimäärittelyt Vektori-luokan yhteydessä. Operaattori += on myös ylimääritelty lisäämään vektoriin toisen vektorin arvot. Sama operaattori on ylimääritelty myös ottamaan argumentikseen tavallisen tietotyypin. Tällöin esimerkissä lisätään kokonaisluku vektorin alkioihin. Tätä operaattoria hyödynnetään sitten kasvatusoperaattorin ++ ylimäärittelyssä.

Vektorin käsittely

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

class Vektori {
 public:
   Vektori(int alku, int lkm);
   Vektori(const Vektori& v);
   ~Vektori();
   const Vektori&  Vektori::operator+= (const Vektori &c);
   const Vektori& Vektori::operator+= (int x);
   const Vektori&  Vektori::operator++ ();
   friend ostream& operator<< (ostream& oo, const Vektori& c);
   friend istream& operator>> (istream& ii, const Vektori& c);
 private:
   int *v;
   const int eka;
   int maara;
 };


Vektori::Vektori(int alku, int lkm)
 : eka(alku), maara(lkm)
 {
   v = new int[maara];
```

```
 }

 Vektori::~Vektori()
 {
 delete[] v;
 }

ostream& operator<< (ostream& oo, const Vektori& c)
{
cout << "Alkiot ovat " << endl;
for (int k=0; k < c.maara; k++)
oo << c.v[k] << "\n";
return oo;
}

istream& operator>> (istream& ii, const Vektori& c)
{
   cout << "Vektoriin menee " << c.maara << " alkiota" << endl;
   cout << "Anna arvot" << endl;
    for (int k=0; k < c.maara ; k++)
   {
     ii >> c.v[k];
   }
return ii;
}

const Vektori&  Vektori::operator+= (const Vektori &c)
{
for (int i=0; i<maara; i++)
     v[i] += c.v[i];
  return *this;
}

const Vektori& Vektori::operator+= (int x)
{
    for (int i=0; i<maara; i++)
      v[i] += x;
    return *this;
}


const Vektori&  Vektori::operator++ ()
{
  return (*this) += 1;
}



void  main()
 {
 Vektori v1(0, 3);
```

```
 cin >> v1;
 cout << v1;
 Vektori v2(0, 3);
 cin >> v2;
 cout << v2;
 v1 += v2;
 cout << v1;
 v1++;
 cout << v1;


getch();
 }
```



STL-kirjaston vektorioperaatiot

Seuraavana on esimerkkejä STL-kirjaston käyttämisestä 1-ulotteisen taulukon käsittelyyn. Operaatioita on useita erilaisia. Seuraavaksi näytetään maksimin hakeminen sekä kääntäminen (reverse) ja pyöräyttäminen (rotate).

Suurimman hakeminen vektorista:

```
#include <algorithm>
template <class ForwardIterator>
InputIterator max_element(ForwardIterator first, ForwardIterator
last);


#include <algorithm>
 #include <vector>
```

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

using namespace std;

int main(void)
{
  typedef vector<int>::iterator iterator;
  int taulu[4] = {100, 200, 10, 20};

  vector<int> vekto1(taulu,taulu + 4);
  iterator suurin = max_element(vekto1.begin(), vekto1.end());

  cout << *suurin << endl;
   getch();
  return 0;

}
```

Vektorin kääntäminen:

Syntaksi:

```
#include <algorithm>
template <class BidirectionalIterator>
 void reverse (BidirectionalIterator first, BidirectionalIterator
last);
```

```
#include <algorithm>
#include <vector>
#include <iostream.h>
#include <conio.h>

 using namespace std;

 int main(void)
 {
   typedef vector<int>::iterator iterator;
   int taulu[4] = {100, 200, 10, 20};

   vector<int> vekto(taulu, taulu+4);
    cout << "Alkuperäinen vektori\n";
   for (int k = 0; k < 4; k++)
   cout << taulu[k] << "\t";

// Käännetään vektori

   reverse(vekto.begin(), vekto.end());
    cout << "\nKäännetty vektori\n";
```

```
   for (int k = 0; k < 4; k++)
   cout << vekto[k] << "\t";


   getch();

}
```

Vektorin pyöräyttäminen:

```
#include <algorithm>
template <class ForwardIterator>

void rotate (ForwardIterator first, ForwardIterator middle,
ForwardIterator last);

#include <algorithm>
 #include <vector>
 #include <iostream.h>
 #include <conio.h>

 using namespace std;

 int main(void)
 {
   typedef vector<int>::iterator iterator;
   int taulu[4] = {100, 9999, 10, 20};

   vector<int> vekto(taulu, taulu+4);
    cout << "Alkuperäinen vektori\n";
   for (int k = 0; k < 4; k++)
   cout << taulu[k] << "\t";

   rotate(vekto.begin(), vekto.begin()+2, vekto.end());

    cout << "\nPyöräytetty vektori\n";
   for (int k = 0; k < 4; k++)
   cout << vekto[k] << "\t";


   getch();

}
```

Tulos: