Toteutusdokumentti

Lasse Lybeck

2. maaliskuuta 2013

Sisältö

1 Ohjelman yleisrakenne

Ohjelman keskeisin luokka on luokka Matrix, joka toimii kirjaston rajapintana ulospäin. Kaikki matriisioperaatiot voidaan suorittaa kutsumalla niitä suoraan matriisi-oliosta. Kaikki operaatiot palauttavat kutsusta matriisin, lukuunottamatta determinatti funktiota, joka palauttaa double-arvon, ja LU-hajotelmaa, joka palauttaa tyyppiä LU olevan olion, johon on sisällytetty hajotelman matriisit.

Operaatiot on toteutettu Operations-luokassa staattisina metodeina. Tämän luokan näkyvyys on kuitenkin rajoitettu vain matrix-kansion luokille. Näin kaikki operaatiot ovat helposti kutsuttavissa suoraan matriisiolioista, mutta toiminnallisuus on siirretty toiseen luokkaan, ettei Matrix-luokka paisuisi aivan liian suureksi.

Käyttöliittymä on toteutettu kansiossa ui. Tätä emme kuitenkaan suuremmin tarkastele, sillä harjoitustyön tarkoituksena oli toteuttaa tehokas matriisikirjasto. Käyttöliittymä on lisätty vain operaatioiden visualisoimista ja testaamista varten. Näin ulkopuolisen käyttäjän on helppo saada tuntuma kirjaston toiminnallisuudesta.

2 Saavutetut aikavaativuudet

2.1 Yhteenlasku

```
add(Matrix a, Matrix b)
    for i = 1 .. a.rows
        for j = 1 .. a.columns
            result(i,j) = a(i,j) + b(i,j)
    return result
```

Kahden $m \times n$ matriisin yhteenlaskun aikavaatimukseksi saadaan selvästi $O(mn)$. Jos kyseessä on kahden $n \times n$ neliömatriisin yhteenlasku saadaan siis aikavaatimukseksi $O\left(n^2\right)$.

Vähennyslasku on täysin ekvivalentti yhteenlaskun kanssa, joten vaativuudet ovat samat.

2.2 Kertolasku

2.2.1 Skalaarilla kertominen

```
scale(Matrix m, float k)
    for i = 1 .. m.rows
```

```
        for j = 1 .. m.columns
            m(i,j) *= k
```

$m \times n$ matriisin skalaarilla kertominen on selvästi aikavaativuudeltaan $O(mn)$. $n \times n$ neliömatriisille tämä siis on $O\left(n^2\right)$.

2.2.2 Matriisikertolasku

```
mul(Matrix a, Matrix b)
    for i = 1 .. a.rows
        for j = 1 .. b.columns
            result(i,j) = 0
            for k = 1 .. a.columns
                result(i,j) += a(i,k) * b(k,j)
    return result
```

Matriisikertolaskussa ensimmäisen matriisin sarakkeiden määrä tulee olla sama kuin toisen matriisin rivien määrä. Matriisien $A \in \mathbb{R}^{m \times n}$ ja $B \in \mathbb{R}^{n \times p}$ tuloksena saadaan siis matriisi $AB \in \mathbb{R}^{m \times p}$. Naiivia algoritmia (yllä) seuraamalla aikavaativuudeksi saadaan $O(mnp)$, joka neliömatriisien $A, B \in \mathbb{R}^{n \times n}$ tapauksessa on $O\left(n^3\right)$.

Kappaleessa 3 sivulla 7 verrataan naiivia algoritmia Strassenin asymptoottisesti nopeampaan algoritmiin.

2.3 Potenssiin korottaminen

```
pow(Matrix m, int e)
    if e == 1
        return m
    else if e mod 2 == 0
        return pow(m * m, e/2)
    else
        m * pow(m * m, (e-1)/2)
```

Potenssiin korottaminen on toteutettu toistuvalla neliöinnillä. Tämä perustuu seuraavaan ideaan. Olkoon $M \in \mathbb{R}^{n \times n}$ neliömatriisi ja $e \in \mathbb{Z}_+$, $e \geq 2$. Jos e on parillinen pätee

$$M^e = \left(M^2\right)^{e/2} = (M \cdot M)^{e/2},$$

jos taas e on pariton on

$$M^e = M \cdot M^{e-1} = M \cdot (M \cdot M)^{(e-1)/2}.$$

Pahimmassa tapauksessa jokaisen exponentin jaon jälkeen on exponentti jälleen pariton, jolloin yhden kutsun aikana joudutaan tekemään kaksi kertolaskutoimitusta (jotka kappaleen 2.2.2 mukaan ovat luokan $O(n^3)$ operaatioita). Tämä tapaus saadaan tilanteessa, jossa $e = 2^k - 1$ jollakin $k \in \mathbb{Z}_+$. Tällöin exponentin jaon jälkeen saadaan uudeksi exponentiksi

$$e' = \frac{e-1}{2} = \frac{\left(2^k - 1\right) - 1}{2} = \frac{2\left(2^{k-1} - 1\right)}{2} = 2^{k-1} - 1.$$

Koska exponentti näin saadaan (vähintään) puolitettua joka kutsulla, tarvitaan kutsuja yhteensä $\log_2(e)$ kappaletta. Kun jokaisessa kutsussa tehdään korkeintaan kaksi kertolaskutoimitusta, saadaan aikavaativuudeksi

$$O\left(\log_2(e) \cdot \left(2 \cdot n^3\right)\right) = O\left(n^3 \log(e)\right).$$

2.4 LU-hajotelma ja determinantti

LU-hajotelmassa matriisi $M \in \mathbb{R}^{n \times n}$ jaetaan kahden matriisin $L, U \in \mathbb{R}^{n \times n}$ tuloksi niin, että L on (yksikkö)alakolmio- ja U yläkolmiomatriisi (lower ja upper triangular matrix, mistä matriisien nimet tulevat). Tällainen jako ei aina välttämättä ole mahdollinen, ja yhtälö saakin yleensä muodon $P^{-1}M = LU$, missä $P \in \mathbb{R}^{n \times n}$ on permutaatiomatriisi. Tarkastelemme tässä kuitenkin vain tapausta, jossa pärjäämme ilman permutaatiomatriisia, sillä algoritmin idea ei tästä kärsi (voimmehan merkitä esimerkiksi $M' = P^{-1}M$, jolloin voimme tarkastella tapausta $M' = LU$).

Seuraavassa algoritmi pseudokoodina.

```
LU(Matrix M)
    L = eye(n,n)
    U = copy(M)
    for i = 1 .. n
        for j = i + 1 .. n
            c = -U(j,i) / U(i,i)
            addMulRow(U, i, j, c)
            L(j,i) = -c
    return L, U

addMulRow(Matrix M, i, j, c)
    for k = 1 .. n
        M(j,k) += c * M(i,k)
```

Käydään läpi algoritmin idea pikaisesti.

Aloitetaan tilanteesta, jossa U sisältää samat alkiot kuin M, ja L on yksikkömatriisi, jolloin $M = LU$. Nyt toistetaan kaikille luvuille $i = 1 \dots n-1$ seuraava:

Matriisista U poistetaan alkiot ensimmäisen diagonaalialkion alta. Tämä tehdään niin, että kaikille $i < j \leq n$ asetetaan $L'_{j,i} = \frac{U_{j,i}}{U_{i,i}}$. Tällöin $U'_{j,i} = U_{j,i} - L'_{j,i}U_{i,i} = 0$, joten kaikki alkion $U_{i,i}$ alla olevat alkiot saadaan poistettua lisäämällä matriisissa U riviin j rivi i kerrottuna luvulla $-L'_{j,i}$. Lisäksi tulon LU arvo säilyy muuttumattomana.

Algorimista nähdään helposti, että uloin silmukka suoritetaan n kertaa ja sisempi silmukka $n-(i-1)$ kertaa. Lisäksi sisemmän silmukan runko on selvästi $O(n)$, sillä rivin lisäämisessä tarvitaan silmukka, joka suoritetaan n kertaa. Näin ollen aikavaativuudeksi saadaan

$$\sum_{k=1}^{n} kn = \frac{1}{2}(n-1)n^2 = O\left(n^3\right).$$

Matriisin determinantin laskeminen seuraa suoraan matriisin LU-hajotelmasta. Tiedetään, että $\det(AB) = \det(A)\det(B)$. Lisäksi tiedetään, että kolmiomatriisin determinantti saadaan kertomalla diagonaalialkiot keskenään. Kun L on yksikkökolmiomatriisi (eli diagonaalilla on pelkkiä ykkösiä), saadaan $\det(L) = 1$, ja edelleen

$$\det(A) = \det(LU) = \det(L)\det(U) = \det(U) = \prod_{k=1}^{n} U_{k,k}.$$

Näin ollen determinantin saa laskettua ajassa $O(n)$ kun tiedetään matriisin LU-hajotelma, eli determinantin laskemisen aika vaativuus on myös $O\left(n^3\right)$.

2.5 Käänteismatriisin laskeminen

Matriisin $M \in \mathbb{R}^{n\times n}$ käänteismatriisin laskeminen perustuu yhtälön $MX = I$ ratkaisemiseen, missä $I \in \mathbb{R}^{n\times n}$ on yksikkömatriisi. Matriisilla M on käänteismatriisi kun $\det(M) \neq 0$. Matriisiyhtälö $AX = B$ ratkaistaan ratkaisemalla n lineaarista yhtälöryhmää. Merkitään

$$X = [x_1, x_2, \dots, x_n] \quad \text{ja} \quad B = [b_1, b_2, \dots, b_n],$$

missä $x_i, b_i \in \mathbb{R}^{n\times 1}$ ovat pystyvektoreita. Kun $1 \leq i \leq n$ saadaan x_i ratkaistua yhtälöstä $Ax_i = b_i$, mikä on normaali n:n yhtälön lineaarinen yhtälöryhmä. Tällä yhtälöllä on yksikäsitteinen ratkaisu, kun $\det(A) \neq 0$.

Tarkastellaan nyt lineaarisen yhtälöryhmän ratkaisemista. Olkoon $A \in \mathbb{R}^{n\times n}$ ja $b \in \mathbb{R}^{n\times 1}$ ja haluamme ratkaista vektorin $x \in \mathbb{R}^{n\times 1}$ yhtälöstä $Ax = b$.

Yhtälöryhmän ratkaiseminen on nopeaa kun tiedetään matriisin A LU-hajotelma. Tällöin yhtälö saa muodon $LUx = b$. Tämän ratkaiseminen on helppoa kun tiedetään, että L on alakolmio- ja U yläkolmiomatriisi. Merkitään $Ux = z$, jolloin yhtälö saadaan muotoon $Lz = b$. Nyt z saadaan ratkaistua helposti substituoimalla matriisi L eteenpäin. Nyt voimme ratkaista yhtälön $Ux = z$, joka ratkeaa substituoimalla U taaksepäin.

Esitetään nyt käänteismatriisin laskeminen pseudokoodina.

```
inv(Matrix m)
    L,U = LU(m)
    e = eye(n,n)
    Vector[] eCols = getColumns(e)
    for i = 1 .. n
        xCols[i] = solveSystem(L, U, e)
    return matrixFromColumns(xCols)

solveSystem(Matrix L, Matrix U, Matrix e)
    z = forwardSubstitute(L, e)
    x = backwardSubstitute(U, z)
    return x

forwardSubstitute(Matrix L, Matrix e)
    x = new Vector(n,1)
    x(1,1) = e(1,1)
    for i = 1 .. n
        sum = 0
        for j = 1 .. i - 1
            sum += L(i,j) * x(j,1)
        x(i,1) = e(i,1) - sum
    return x

backwardSubstitute(Matrix U, Matrix e)
    x = new Vector(n,1)
    x(n,1) = e(n,1) / U(n,n)
    for i = n - 1 .. 1
        sum = 0
        for j = n .. i + 1
            sum += U(i,j) * x(j,1)
        x(i,1) = (e(i,1) - sum) / U(i,i)
    return x
```

Tiedämme, että LU-hajotelma saadaan ratkaistua ajassa $O\left(n^3\right)$ (kts. kappale 2.4). Lineaarisen yhtälöryhmän ratkaisemisessa tarvitaan kaksi substituu-

tiokutsua. Näiden aikavaativuus on

$$\sum_{k=1}^{n-1} = \frac{1}{2}(n-1)n.$$

Kun yhtälöryhmiä ratkaistaan n kappaletta, saadaan koko algoritmin aikavaativuudeksi

$$O\left(n^3 + n \cdot 2 \cdot \frac{1}{2}(n-1)n\right) = O\left(n^3 + (n-1)n^2\right) = O\left(n^3\right).$$

3 Suorituskyky- ja O-analyysivertailu

Matriisikertolaskulle on kirjastossa toteutettu normaalin kertolaskualgoritmin lisäksi asymptoottisesti nopeampi Strassenin algoritmi. Kuten kappaleesta 2.2.2 tiedetään, on normaalin kertolaskun aikavaativuus $O\left(n^3\right)$. Strassenin algoritmin aikavaatimus on noin $O\left(N^{2.8074}\right)$ (tätä emme osoita), mutta algoritmi toimii vain matriiseilla $M \in \mathbb{R}^{N \times N}$, missä $N = 2^n$ jollakin $n \in \mathbb{Z}_+$. Jos matriisi ei ole tätä muotoa aloitetaan algoritmi suurentamalla matriisi tällaiseen muotoon. Tästä syystä algoritmeja on järkevä vertailla vain matriiseille $M \in \mathbb{R}^{2^n \times 2^n}$.

Käydään läpi Strassenin algoritmin idea pikaisesti. Strassenin algoritmi perustuu matriisin jakamiseen osiin, ja näitä osia yhdistelemällä rakentamaan matriisien tulon.

Olkoon nyt $n \in \mathbb{Z}_+$ ja $A, B \in \mathbb{R}^{2^n \times 2^n}$. Haluamme laskea matriisin $C = AB$. Jaetaan ensin matriisit osiin

$$A = \begin{pmatrix} A_{1,1} & A_{1,2} \\ A_{2,1} & A_{2,2} \end{pmatrix}, \quad B = \begin{pmatrix} B_{1,1} & B_{1,2} \\ B_{2,1} & B_{2,2} \end{pmatrix}, \quad C = \begin{pmatrix} C_{1,1} & C_{1,2} \\ C_{2,1} & C_{2,2} \end{pmatrix},$$

missä $A_{i,j}, B_{i,j}, C_{i,j} \in \mathbb{R}^{2^{n-1} \times 2^{n-1}}$. Matriisi C saadaan nyt laskemalla

$$\begin{aligned} C_{1,1} &= A_{1,1}B_{1,1} + A_{1,2}B_{2,1} \\ C_{1,2} &= A_{1,1}B_{1,2} + A_{1,2}B_{2,2} \\ C_{2,1} &= A_{2,1}B_{1,1} + A_{2,2}B_{2,1} \\ C_{2,2} &= A_{2,1}B_{1,2} + A_{2,2}B_{2,2}. \end{aligned}$$

Tähän tarvitaan kahdeksan kertolaskutoimitusta.

Lasketaan nyt matriisit

$$
\begin{aligned}
M_1 &= (A_{1,1} + A_{2,2})(B_{1,1} + B_{2,2}) \\
M_2 &= (A_{2,1} + A_{2,2})B_{1,1} \\
M_3 &= A_{1,1}(B_{1,2} - B_{2,2}) \\
M_4 &= A_{2,2}(B_{2,1} - B_{1,1}) \\
M_5 &= (A_{1,1} + A_{1,2})B_{2,2} \\
M_6 &= (A_{2,1} - A_{1,1})(B_{1,1} + B_{1,2}) \\
M_7 &= (A_{1,2} - A_{2,2})(B_{2,1} + B_{2,2}).
\end{aligned}
$$

Nyt saamme muodostettua matriisin C seuraavasti.

$$
\begin{aligned}
C_{1,1} &= M_1 + M_4 - M_5 + M_7 \\
C_{1,2} &= M_3 + M_5 \\
C_{2,1} &= M_2 + M_4 \\
C_{2,2} &= M_1 - M_2 + M_3 + M_6
\end{aligned}
$$

Tällä tavalla matriisin C konstruointiin tarvittiin kahdeksan kertolaskutoimituksen sijaan vain seitsemän laskutoimitusta. Vaikka yhteenlaskujen määrä kasvoi reilusti, ei tämä haittaa suuremmin, sillä yhteenlasku on verrattain nopea toimenpide kertolaskuun verrattuna.

Taulukosta 1 sivulla 9 nähdään, että Strassenin algoritmi jää reilusti jälkeen normaalista kertolaskusta kohtuullisen pienillä matriiseilla. Huomattavaa on kuitenkin, että suhteellinen ero pienenee huomattavasti kun matriisien koko kasvaa. Vaikka Strassenin algoritmi kesti 128×128 matriiseille yli 300 kertaa niin kauan kuin normaali kertolasku, ei se kestänyt enää kuin hieman yli 20 kertaa niin kauan kuin normaali kertolasku, kun matriisien koko oli 1024×1024.

Tämä Strassenin algoritmin toteutus ei kuitenkaan ole läheskään optimoitu, kuten vertailusta voi päätellä. Java ei muutenkaan ole välttämättä paras mahdollinen kieli tällaisen operaation tekemiseen. Algoritmissa luodaan jatkuvasti uusia matriiseja, jotka nopeasti hylätään ja Javassa tämä tarkoittaa, että roskienkerääjä aktivoituu paljon. Esimerkiksi C-kielessä näiden matriisien muisti voitaisiin helposti vapauttaa, eikä tämä haittaisi suorituskykyä. Javassa tämä kuitenkaan ei ole mahdollista.

Algoritmia pystyisi luultavasti kuitenkin optimoimaan melko paljon myös Javassa niin, että se saattaisi päästä lähelle normaalin kertolaskun vauhtia jo parin tuhannen kokoisille matriiseille. Tämä ei kuitenkaan ajan puutteen vuoksi ole ollut mahdollista.

Taulukko 1: Kertolaskujen vertailu

n	Normaali (s)	Strassen (s)	Suhde
128	0.0040	1.2380	309.5000
256	0.0550	7.4720	135.8545
512	0.6960	53.2340	76.4856
1024	17.2210	380.5040	22.0953