Szakdolgozat

Szabó Norbert

Debrecen

2009

Debreceni Egyetem
Informatika Kar

Lineáris egyenletrendszerek iterációs megoldása

Témavezető:
Dr. Baran Ágnes
egyetemi adjunktus

Készítette
Szabó Norbert
Programtervező informatikus

Debrecen

2009

Tartalomjegyzék

1 Bevezetés **1**

2 Iterációs módszerek **2**

3 A konvergencia vizsgálata **4**

4 LDL^T felbontás **6**

4.0.1 Implementáció 6

5 Néhány alapvető iterációs eljárás **7**

5.1 Jelölések 7

5.2 Jacobi és Gauss-Seidel iteráció 8

5.2.1 Implementáció 9

5.3 SOR (Successive overrelaxation) iteráció 9

5.3.1 Implementáció 10

5.4 Egyszerű gradiens módszer 10

5.4.1 Implementáció 13

5.5 A konjugált gradiens módszer 13

5.5.1 Implementáció 16

5.6 A prekondícionált konjugált gradiens módszer 16

5.6.1 Implementáció 17

6 Inkomplett Cholesky felbontás **20**

6.0.2 Implementáció 21

7 Az implementált módszerek gyorsítása **22**

7.1 A gyorsított módszerek futási ideje 26

8 Tesztelés **29**

8.1 A tesztelésből levonható következtetések 30

9 Összefoglalás **32**

10 Köszönetnyilvánítás **33**

11 Irodalomjegyzék **34**

12 A) FÜGGELÉK **35**

1. Bevezetés

Számos gazdasági, mérnöki probléma numerikus megoldása során lineáris egyenletrendszerek megoldásával találkozhatunk.

Dolgozatomban az $Ax = b$ egyenletrendszer megoldásával foglalkozom, ahol $A \in \Re^{nxn}$, b, $x \in \Re^n$ és feltételezem, hogy a lineáris egyenletrendszer egyértelműen megoldható. Elsősorban iteratív módszereket vizsgálok, ahol egy $x^{(0)} \in \Re^n$ kezdővektorból kiindulva előállítunk egy $x^{(k)}, k = 1, 2 \ldots$ vektorsorozatot, mely alkalmas feltételek mellett konvergál az egyenletrendszer megoldásához. A dolgozat elején az iteratív és a direkt módszerek közötti különbségről lesz szó, itt megemlítek az iterációs módszerek konvergenciájával kapcsolatban néhány ismert tételt és definiciót. Ezt követően különböző módszereket mutatok be amelyek az $Ax = b$ egyenletrendszer megoldására szolgálnak. A dolgozatban tárgyalt iteratív módszerek: Jacobi-iteráció, Gauss-Seidel iteráció, SOR módszer, egyszerű gradiens módszer, konjugált gradiens módszer és a prekondicionált konjugált gradiens módszer. Röviden leírom az inkomplett Cholesky-felbontást, mert ezzel a felbontással jó prekondícionálási mátrixot tudunk létrehozni a prekondícionált konjugált gradiens számára, így a konjugált gradiens módszer konvergenciáját felgyorsíthatjuk. A módszerek mindegyikét szimmetrikus, pozitív definit mátrixokkal teszteltem. Összehasonlításképpen egy szimmetrikus mátrixok esetén használható direkt módszert, az LDL^T felbontást is vizsgáltam. Az algoritmusokat C programozási nyelven implementáltam. Mivel az iterációs módszerek használata, a direkt módszerekkel szemben elsősorban ritka mátrixok esetén hasznos, ezért a teszteléshez ritka mátrixokat használtam. A példákat a Floridai Egyetem által szerkesztett *http://www.cise.ufl.edu/research/sparse/matrices/* oldalról vettem. Az itt található ritka mátrixokat háromféle ritka mátrix reprezentáióban lehet elérni (Matlab(MAT), Matrix Market(MM) és Rutherford/Boeing (RB)), én az MM formátumot használtam.

A teszteredményeket a dolgozat végén táblázatban gyűjtöttem össze. Itt az egyes tesztmátrixok azonosító száma mellett a 2-normában becsült kondíciószámukat is megadtam.

2. Iterációs módszerek

Az $Ax = b$ lineáris egyenletrendszer megoldását két alapvető módszerrel végezhetjük: az egyik a direkt módszer, a másik az iterációs módszer. A direkt módszerek lényege, hogy a megoldást egy lépésben adják, és elméletileg a pontos megoldást szolgáltatják. Hátrányuk az, hogy nagy a memóriaigényük. A feladatokban azonban sokszor olyan nagy méretű, úgynevezett ritka mátrixokkal találkozunk, melyekben az elemek jelentős része nullával egyenlő. Ilyenek például a parciális differenciál egyenletek numerikus megoldása során keletkező lineáris egyenletrendszerek alapmátrixai.

Ha a mátrixban a nemnulla elemek bizonyos jól meghatározott struktúra szerint helyezkednek el, akkor lehetőség van arra, hogy a direkt módszereket ilyen esetekre módosítsuk, és ezzel a memória és műveletigényt csökkentsük. Ilyen a helyzet például akkor, ha az A mátrix úgynevezett sávos mátrix, azaz a nemnulla elemek a főátlóval párhuzamos átlókban állnak (például a mátrix tridiagonális). Ebben az esetben például az LU-felbontás módosítható úgy, hogy csak ezeket az értékeket tároljuk. Sokszor azonban a nemnulla elemek elhelyezkedése miatt ilyen egyszerűsítésre nincs lehetőség, ugyanis az algoritmus során a nulla elemek is nemnullává válhatnak, a mátrix feltöltődhet. Ilyenkor nem célravezető a direkt módszer használata. Az iterációs módszereknél a tárigény kisebb lehet, viszont nem biztos, hogy a pontos megoldást adják, és gondot okozhat, ha lassú a konvergencia. Ugyanakkor nem is biztos, hogy érdemes a pontos eredményt kiszámolni, mert általában a mátrix és a jobb oldal is hibás.

Az iterációs módszereket általában a következő alakban szoktuk felírni:

$$x^{(m+1)} = Bx^{(m)} + f, \qquad m = 0, 1, \ldots \tag{1}$$

ahol a B és f alkalmas mátrix és vektor, ezeket az A mátrixból és b vektorból állítjuk elő, az adott módszernek megfelelően.

Ha a B mátrix nem függ m-től, akkor az iterációt stacionárius iterációnak nevezzük. Az iteráció esetén egy iterációs lépés az egy márix-vektor szorzás, ami nxn-es mátrix esetén n^2 műveletet jelent (egy művelet alatt 1 összeadást és egy szorzást értve).

Iterációs módszereknél fontos kérdés, hogy a módszer konvergál-e, erre hatással lehet $x^{(0)}$ megválasztása. Ugyanakkor a kezdővektor megválasztásába sokszor nem érdemes túl sok munkát fektetni, általában kezdővektornak a nullvektort szokták választani. Rosszul kondícionált mátrixoknál lehet érdemes a kezdővektor kiszámítására időt és energiát fordítani, egyébként nem.

A másik fontos kérdés, az iterációs módszerek alkalmazásánál a leállási kritérium. Általában akkor szoktuk befejezni az iterációt, ha a megoldás két egymás utáni közelítése kellően közel

van egymáshoz, azaz

$$|| x^{(m)} - x^{(m+1)} || \leq \epsilon \tag{2}$$

teljesül valamilyen vektornormában, ahol az ϵ pontosságot bemenetként várja a program. Ez a leállási feltétel nem feltétlenül biztosítja, hogy a megoldás ϵ sugarú környezetében sikerül megállnunk.

Mivel az iterációs eljárás nem mindig konvergál, ezért érdemes egy maximális iterációszámot megadni, ha ezt elértük, akkor a program befejezi a működését. Ilyenkor a $|| x^{(m)} - x^{(m+1)} ||$ értékének ismeretében dönthetünk a maximális iterációszám emeléséről vagy más kezdővektor választásáról.

3. A konvergencia vizsgálata

Ebben a fejezetben összefoglalok néhány ismert eredményt az iterációs módszerek konvergenciájával kapcsolatban.

3.1. Definíció. Azt mondjuk, hogy (1) iteráció adott $x^{(0)} \in \Re^n$ mellett konvergál, ha $x^{(m)}$ sorozat konvergens az $\Re^n$ tér valamely normájában. Ha tetszőleges $x^{(0)} \in \Re^n$-re konvergál, akkor konvergensnek hívjuk az iterációs eljárást.

Az iterációs módszerek konvergenciájára egy elegendő feltételt adhatunk a Banach-féle fixpont tétel segítségével.

Tegyük fel, hogy a $B : \Re^n \to \Re^n$ leképezés minden $x,\ y \in \Re^n$ esetén eleget tesz a

$$\| Bx - By \| \leq\ q* \| x - y \|$$

feltételnek, ahol $0 \leq q < 1$ rögzített. Ekkor az $x = Bx + f$ egyenletnek létezik egyértelmű $x^* \in \Re^n$ megoldása. Tetszőleges $x^{(0)} \in \Re^n$ esetén a (1) sorozat tart x^*-hoz, továbbá

$$\| x^{(m)} - x^* \| \leq\ \frac{q^m}{1-q} \| x^{(1)} - x^{(0)} \|, \tag{3}$$

illetve

$$\| x^{(m)} - x^* \| \leq\ \frac{q}{1-q} \| x^{(m)} - x^{(m-1)} \| \tag{4}$$

teljesül.

Megjegyzés. Ha a $B \in \Re^{nxn}$ mátrix olyan, hogy $\| B \| =: q < 1$ a fenti vektornorma által indukált mátrxnormában, akkor

$$\| Bx - By \| = \| B(x-y) \| \leq \| B \| * \| x - y \| =\ q* \| x - y \|$$

teljesül minden $x,\ y \in \Re^n$ esetén, tehát ekkor az iteráció konvergens.

Megjegyzés. Mivel $\Re^n$-en bármely két norma ekvivalens, ezért a $\| B \| <\ 1$ feltételnek elegendő egyetlen, indukált mátrixnormában teljesülni.

Megjegyzés. A (3) egyenlőtlenség segítségével becslést adhatunk a szükséges iterációs lépések számára, míg a (4) egyenlőtlenség a leállási feltételnél használható.

3.1. Tétel. *(Stacionárius iteráció szükséges és elégséges konvergencia feltétele). Az (1) iteráció legyen stacionárius. Ilyenkor az iterációs eljárás pontosan akkor konvergens, ha a $\rho(B)$ spektrál sugárra teljesül, hogy*

$$\rho(B) = max \mid \lambda(B) \mid < 1.$$

Megjegyzés.

1. A $\rho(B)$ általában nem norma, kivéve szimmetrikus mátrix esetén.

2. A $\rho(B) < 1$ feltétel nem mindig biztosítja a numerikus konvergenciát: az iteráció a számítógépen lehet, hogy nem konvergál, hanem túlcsordulás miatt leáll, mivel a $B^m \to 0$ konvergencia nem monoton.

4. LDL^T felbontás

Az elsőként tárgyalt algoritmus egy direkt módszer, mely szimmetrikus A esetén alkalmazható az $Ax = b$ lineáris egyenletrendszer megoldására. Az algoritmus lényege, hogy a szimmetrikus A mátrixot felbontjuk $A = LDL^T$ alakban, ahol az L alsó háromszögmátrix, melynek átlójában csupa egyes áll, míg a D mátrix diagonális.

Felhasználva, az A mátrix szimmetriáját, elegendő a mátrix alsó háromszög részét tárolni, és az L illetve D mátrixok tárolása ugyanennyi helyen megoldható. Ez körülbelül felére csökkenti a műveletigényt.

Az algoritmus az L mátrix alsó háromszögbeli elemeit oszloponként állítja elő, és ehhez A-nak csak a bal alsó háromszögbeli elemeit használja fel. Az LDL^T felbontás pszeudo kódja a következő:

1. $j := 1(1)n$
2. $\quad [i := j(1)n \quad [r_i := a_{ij}]_i, \quad x_j := b_j$
3. $\quad k := 1(1)j - 1$
4. $\quad\quad [i := j(1)n \quad [r_i := r_i - l_{ik} * r_k * l_{jk}]_i, \quad x_j := x_j - l_{jk} * x_k]_k$
5. $\quad$ Ha $r_j = 0$, akkor hibajelzés, j közlése, és kiszállás. Egyébként $d_j := 1/r_j$.
6. $\quad i := j + 1(1)n \quad [l_{ij} := r_i * d_j]_i]_j$
7. $i := n(-1)1 \quad [x_i := d_i * x_i$
8. $\quad k := i + 1(1)n \quad [x_i := x_i - l_{ki} * x_k]_k]_i$
9. stop [eredmény: x]

4.0.1. Implementáció

Az LDL^T felbontást *LDLT.c* néven implementáltam. A program n, A, b -t vár bemenetként. A kimenet pedig x, eltelt idő. A felbontás nem iterációs felbontás. A tesztek során a módszer körülbelül a Jacobi-iteráció-val és a Gauss-Seidel-el azonos idő alatt állította elő a keresett x vektort.

5. Néhány alapvető iterációs eljárás

Az $Ax = b$ lineáris egyenletrendszerből az $x^{(k+1)} = Bx^{(k)} + f$ iterációhoz például az úgynevezett szétvágással juthatunk el. Ez azt jelenti, hogy az A mátrixot felbontjuk két mátrix különbségére: $A = P - Q$, ahol P reguláris.

$$Ax = b \tag{5}$$

$$(P - Q)x = b \tag{6}$$

$$Px = Qx + b \qquad / * P^{-1} \tag{7}$$

$$x = P^{-1}Qx + P^{-1}b, \quad \text{ahol} \quad B := P^{-1}Q \quad \text{és} \quad f := P^{-1}b \tag{8}$$

A felbontást regulárisnak nevezzük, ha P reguláris és $\rho(P^{-1}Q) < 1$. Ekkor az iterációs eljárás konvergens.

5.1. Jelölések

Az egyes algoritmusok ismertetése előtt egy rövid technikai rész következik.

Az implementálás során azonos jelöléseket használtam a programok bemeneteihez és kimeneteihez. Ezek jelentését most leírom, és később már csak hivatkozom rájuk a módszerek bemutatásánál. Ezek a jelek:

– bemenetekben:

n : az A mátrix nxn-es lesz.

A : az A a lineáris egyenletrendszer alapmátrixa.

b : az $Ax = b$ egyenletrendszer jobb oldala.

x0 : a kezdővektor.

maxit : maximális iterációszám. Ha a program eléri a maximális iterációszámot, akkor befejezi a működését.

e : pontosság, ha az $x^{(k)}$ és $x^{(k+1)}$ vektor különbségének euklideszi normája ettől kisebb, akkor a program leáll.

w : iterációs paraméter, csak a SOR iterációnál fordul elő.

H : prekondicionálási mátrix. Ezt a mátrixot a prekondicionált konjugált módszer használja fel. Alkalmas prekondicionálási mátrixot választva, kevesebb lépésszámban ad a módszer eredmenyt. Az Inkomplett Cholesky felbontás is egy ilyen prekondicionálási mátrixot állít elő.

– kimenetekben:

x : az $Ax = b$ egyenletrendszer megoldás vektora.

lépésszám : a módszer hány iterációs lépés alatt talált megoldást.

eltelt idő : a program futásának ideje.

Ezeket írja ki minden program, ebben a sorrendben és külön sorokban.

5.2. Jacobi és Gauss-Seidel iteráció

Bontsuk fel az A mátrixot három mátrix összegére: $A = L+D+U$. Itt a D mátrix diagonális mátrix, amely az A mátrix diagonálisában álló elemeket tartalmazza, az L mátrix szigorú alsó háromszög mátrix, az A mátrix diagonálisa alatt álló elemeket tartalmazza, míg az U mátrix az A szigorúan vett felső háromszög része.

1. Jacobi iteráció:
   Ekkor $A = D - (-L - U)$, ahol $P := D$, és $Q := (-L - U)$. Akkor
   $x^{(k+1)} = -D^{-1}(L + U)x^{(k)} + D^{-1}b$. Így az iteráció soronként kiirva:
   $$x_i^{(k+1)} = \frac{1}{a_{ii}} \left(- \sum_{j=1, j\neq i}^{n} a_{ij} x_j^{(k)} + b_i \right) \tag{9}$$
2. Jacobi Seidel iteráció:
   Ekkor $A = (D + L) - (-U)$, ahol $P := (D + L)$, és $Q := (-U)$. Így az iteráció:
   $$x^{(k+1)} = -(D + L)^{-1} U x^{(k)} + (D + L)^{-1} b, \tag{10}$$
   vagy soronként kiirva:
   $$x_i^{(k+1)} = \frac{1}{a_{ii}} \left(- \sum_{j=1}^{i-1} a_{ij} x_j^{(k+1)} - \sum_{j=i+1}^{n} a_{ij} x_j^{(k)} + b_i \right). \tag{11}$$

A Gauss-Seidel iterációnál tulajdonképpen az történik, hogy a $(k + 1)$-edik közelítővektor i-edik koordinátájának kiszámításakor figyelembe vesszük, hogy az $1., 2., \ldots, (i - 1)$. koordinátákat már kiszámítottuk, így azokat felhasználjuk. Emiatt a Gauss-Seidel iterációnál az új közelítővektor koordinátáit csak meghatározott sorrendben számíthatjuk, míg a Jacobi iterációnál ilyen megkötés nincs, az egyes koordinátákat, akár párhuzamosan is számíthatjuk. Ugyanakkor a Seidel-iteráció esetén az $x^{(k+1)}$ vektorral azonnal felülírhatjuk az $x^{(k)}$ vektort, míg a Jacobi iteráció esetén ezt nem tehetjük meg.

A módszerek konvergenciájának vizsgálatánál a 3. fejezetben leírt tételeket alkalmazhatjuk. Néhány mátrixosztályban, például a domináns főátlójú mátrixok esetén, mindkét módszer konvergens.

5.2.1. Implementáció

A két módszert a *Jacobi_iteracio.c* és *Jacobi_Seidel.c* néven implementáltam. Mindkettő ugyanolyan bemenetet vár: $n, A, b, x0, maxit, e$. A kimenetük pedig x, lépésszám, eltelt idő. Ez a két módszer elég kevés lépésszámban, és gyorsan ad eredményt. Az eredmény vektor nagyon jól közelít a keresett x vektorhoz.

5.3. SOR (Successive overrelaxation) iteráció

Ez az iteráció a Gauss-Seidel iteráció egy módosulata.

Az $\omega Ax = \omega b$ egyenletet, (ahol ω egy alkalmasan megválasztott paraméter) alakítsuk át:

$$x \;=\; (1-\omega)x - \omega D^{-1}(L+U)x + \omega D^{-1}b,$$

ahol az L, U és D mátrixok megegyeznek az előző két iterációban leírtakkal. Ebből a következő iterációs eljáráshoz juthatunk:

$$x^{(k+1)} \;=\; (1-\omega)x^{(k)} - \omega D^{-1}(L+U)x^{(k)} + \omega D^{-1}b,$$

vagy koordinátánként kiirva:

$$x_i^{(k+1)} \;=\; (1-\omega)x_i^{(k)} - \frac{\omega}{a_{ii}}\left(\sum_{j=1}^{i-1} a_{ij}x_j^{(k+1)} + \sum_{j=i+1}^{n} a_{ij}x_j^{(k)} - b_i\right),$$

ahol $i = 1, \ldots, n$.

Ha $\omega = 1$, akkor a Gauss-Seidel eljárást kapjuk. Ha az $\omega > 1$ és az A mátrix eleget tesz bizonyos feltételeknek, akkor kevesebb lépésszámban és gyorsabban talál az iteráció megoldást, tehát felgyorsul a konvergencia. Az iterációt ekkor felső relaxációs eljárásnak hívjuk, ezért az angol neve Successive overrelaxation. Ez az eljárás kombinálja a régi és a Gauss-Seidel iteráció által javasolt új közelítést. Úgy, mint a Gauss-Seidel iterációnál, a SOR végrehajtásához is szükséges, hogy $a_{kk} \neq 0,\ k = 1, \ldots, n$, teljesüljön.

5.1. Tétel. *Legyen $D := diag(a_{kk})$ reguláris. A SOR iteráció csak $0 < \omega < 2$, esetén lesz konvergens.*

5.2. Tétel. *Legyen $0 < \omega < 2 \quad A = D + L + L^T \in \Re^{nxn}$ és $D := diag(a_{kk})$ pozitív definit. Ebben az esetben a relaxációs módszer akkor és csak akkor konvergál, ha A is pozitív definit.*

A SOR iterációval alapvetően az a probléma, hogy bizonyos paraméterek helyes megválasztásán múlik az, hogy az iteráció jó eredményt ad e.

Az iteráció pszeudo kódja a következő:

1. $for\ i = 1 : n$

2. $x_i^{(k+1)} = (1-\omega)x_i^{(k)} - \frac{\omega}{a_{ii}}\left(\sum_{j=1}^{i-1} a_{ij}x_j^{(k+1)} + \sum_{j=i+1}^{n} a_{ij}x_j^{(k)} - b_i\right)$

3. end

Hasonlóan a Gauss-Seidel iterációhoz itt is elegendő egyetlen vektort tárolni; a $(k+1)$-edik vektor éppen kiszámított koordinátáival felülírhatjuk a k-adik vektor alkalmas koordinátáit.

5.3.1. Implementáció

A SOR iterációt *SOR_iteracio.c* néven valósítottam meg. A program bemenetként n, A, b, $x0$, $maxit$, e, w -t vár. A kimenet: x, lépésszám, eltelt idő.

5.4. Egyszerű gradiens módszer

A SOR módszer esetén felmerülhet az a probléma, hogy hogyan válasszunk egy alkalmas iterációs paramétert. A következő módszer ilyen megválasztandó paramétert nem tartalmaz. Továbbra is feltételezzük, hogy az A mátrix szimmetrikus és pozitív definit.

Az $Ax = b$ lineáris egyenletrendszer megoldását egy alkalmasan megválasztott funkcionál minimalizására fogjuk visszavezetni. Ehhez először a következő lemmára lesz szükségünk:

5.1. Lemma. *Ha $A \in \Re^{nxn}$ szimmetrikus és pozitív definit, akkor az*

$$\| x \|_* = < Ax, x >^{\frac{1}{2}} \tag{12}$$

függvény normát definiál $\Re^n$-en.

Bizonyítás. Ha A szimmetrikus és pozitív definit, akkor az A mátrixnak létezik a Cholesky felbontása: $A = LL^T$, ahol L reguláris alsóháromszög mátrix. Ekkor

$$\| x \|_*^2 = < Ax, x > = < LL^T x, x > = < L^T x, L^T x > = \| L^T x \|_2^2 \tag{13}$$

teljesül minden $x \in \Re^n$ esetén. Tehát $\| x \|_* = \| L^T x \|_2$, ahol L^T egy reguláris mátrix, így az $\| \cdot \|_*$ függvény normát definiál $\Re^n$-en.

Megjegyzés. Ha $A \in \Re^{nxn}$ szimmetrikus és pozitív definit, akkor A^{-1} is szimmetrikus és pozitív definit, így $\| x \|_* := < A^{-1}x, x >$ is normát definiál.

Mivel A reguláris, ezért az $Ax = b$ lineáris egyenletrendszer megoldása pontosan a

$$J(x) = \| Ax - b \|^2 \tag{14}$$

funkcionál minimumhelye lesz, ahol a normát tetszőlegesen választhatjuk. Legyen a norma most az előbb definiált norma $\frac{1}{2}$ szerese:

$$\| x \|_* \quad = \quad \frac{1}{2} < A^{-1}x, x >^{\frac{1}{2}} . \tag{15}$$

Ekkor

$$J(x) = \| Ax - b \|_*^2 = \frac{1}{2} < A^{-1}(Ax - b), Ax - b > = \frac{1}{2} < x - A^{-1}b, Ax - b > =$$

$$= \frac{1}{2} \left(< x, Ax > - < x, b > - < A^{-1}b, Ax > + < A^{-1}b, b > \right) . \tag{16}$$

Felhasználjuk, hogy $A \in \Re^{nxn}$ szimmetrikus, így

$$< A^{-1}b, Ax > = < A^T A^{-1}b, x > = < AA^{-1}b, x > = < b, x > . \tag{17}$$

A valós euklideszi belső szorzat szimmetriája miatt

$$< b, x > = < x, b >,$$

$$J(x) \; = \; \frac{1}{2} < x, Ax > - < x, b > + \frac{1}{2} < A^{-1}b, b > = \; \frac{1}{2} x^T Ax - x^T b + \frac{1}{2} b^T A^{-1} b. \tag{18}$$

Ennek a $J(x)$ funkcionálnak pontosan ott van minimuma, ahol a

$$\phi(x) = \frac{1}{2} x^T Ax - x^T b \tag{19}$$

funkcionálnak. Így az $Ax = b$ lineáris egyenletrendszer megoldását (ahol A szimmetrikus és pozitív definit) visszavezettük a

$$\phi(x) = \frac{1}{2} x^T Ax - x^T b$$

funkcionál minimumhelyének megkeresésére.

Az A mátrix szimmetrikus, így adott x_c esetén

$$\phi(x_c + \delta x_c) \; = \; \frac{1}{2} (x_c + \delta x_c)^T A (x_c + \delta x_c) - (x_c + \delta x_c)^T b \; =$$

$$= \; \frac{1}{2} x_c^T Ax_c - x_c^T b + \frac{1}{2} \delta x_c^T A \delta x_c + \delta x_c^T Ax_c - \delta x_c^T b \; = \; \phi(x_c) + < Ax_c - b, \delta x_c > + \frac{1}{2} \delta x_c^T A \delta x_c \tag{20}$$

teljesül.

Kis $\delta x_c > 0$ esetén a Cauchy-Schwarz egyenlőtlenségből látszik, hogy a ϕ függvény a $\delta x = -(Ax_c - b) = b - Ax_c$ irányban csökken a legjobban. Ez pontosan a negatív gradiens iránya:

$$-\nabla \phi(x_c) \; = \; b - Ax_c. \tag{21}$$

Ezután a negatív gradiens irányában, azaz az r_c irányban megkeressük a ϕ függvény minimumhelyét. A

$$\phi(x_c + \alpha r_c) \;=\; \phi(x_c) - \alpha r_c^T r_c + \frac{1}{2}\alpha r_c^T A r_c \tag{22}$$

függvényt α szerint deriválva azt kapjuk, hogy a minimumát

$$\alpha \;=\; \frac{r_c^T r_c}{r_c^T A r_c} \tag{23}$$

esetén veszi fel.

A $\phi(x)$ minimalizálására a legegyszerűbb módszer, az egyszerű gradiens módszer. Ezt a módszert úgy kell elképzelni, hogy felületen vagyunk, s elindulunk a legmeredekebb irányba. A legmeredekebb irány a negatív gradiens iránya lesz. A módszer pszeudo kódja a következő:

1. x_0 =kezdővektor
2. $r_0 = b - Ax_0$
3. $k = 0$
4. while $r_k \neq 0$
5. $\quad k = k + 1$
6. $\quad \alpha_k = r_{k-1}^T r_{k-1} / r_{k-1}^T A r_{k-1}$
7. $\quad x_k = x_{k-1} + \alpha_k r_{k-1}$
8. $\quad r_k = b - Ax_k$
9. end

Belátható, hogy a

$$\left(\phi(x_k) + \frac{1}{2} b^T A^{-1} b\right) \leq \left(1 - \frac{1}{\kappa_2(A)}\right)\left(\phi(x_{k-1}) + \frac{1}{2} b^T A^{-1} b\right) \tag{24}$$

összefüggés teljesül, ahol $\kappa_2(A)$ az A mátrix kettőnormában vett kondíciószáma. Ez az összefüggés a globális konvergenciát jelenti. Mivel A szimmetrikus és pozitív definit, ezért $\kappa_2(A) = \frac{\lambda_1(A)}{\lambda_n(A)}$, ahol λ_1, illetve λ_n az A mátrix legnagyobb és legkisebb sajátértékét jelöli. Ha ez a hányados nagy, akkor a konvergencia lassú lehet. Geometriailag ez azt jelenti, hogy a ϕ függvény szintvonalai nagyon elnyújtott hiperellipszoidok lesznek, és a minimalizálás azt jelenti, hogy meg kell találni a völgyek között a legkisebb pontot. Az egyszerű gradiens módszernél arra kényszerülünk, hogy keresztül menjünk a völgyön.

5.4.1. Implementáció

Az egyszerű gradiens módszert *Steespest_Descent.c* néven implementáltam. A program $n, A, b, x0, maxit$ -t vár bemenetként. A program kimenete: x, lépések száma, eltelt idő.

5.5. A konjugált gradiens módszer

Az előző részben leírt eljárásnál gyorsabban konvergáló eljárást kapunk, ha a keresési irányt másképpen választjuk meg.

A konjugált gradiens módszer minden iterációs lépésében a d^k keresési irányt keressük, annak érdekében, hogy a $(k+1)$.-edik iterációban minél jobban közelítsük az $Ax = b$ lineáris egyenletrendszer pontos megoldását. A d^{k+1} keresési irány merőleges lesz az előző d^k keresési irányra.

Tegyük fel, hogy már meghatároztuk az x^k értékét és a d^k keresési irányt. Az x^{k+1} közelítő értéket ezek segítségével állítjuk elő: az x^k pontból a d^k irányba lépünk úgy, hogy a ϕ függvény értéke eközben csökkenjen. Ez azt jelenti, hogy x^{k+1}-et a következő alakban keressük:
$x^{k+1} = x^k + \lambda^k d^k$, ahol a λ-t, az előző részben leírtakhoz hasonlóan úgy keressük meg, hogy a ϕ függvényt minimalizáljuk a d^k irányban.

$$\phi(x^{k+1}) \;=\; \phi(x^k + \lambda d^k) \;=\; \phi(x^k) + \lambda < r^k, d^k > + \frac{1}{2}\lambda^2 < Ad^k, d^k > \tag{25}$$

λ szerint deriválva azt kapjuk, hogy

$$\lambda \;=\; \lambda^k \;=\; -\frac{< r^k, d^k >}{< Ad^k, d^k >}, \tag{26}$$

tehát $x^{k+1} = x^k + \lambda^k d^k$.

Mivel

$$r^{k+1} = Ax^{k+1} - b = A(x^k + \lambda^k d^k) - b = Ax^k - b + \lambda^k Ad^k = r^k + \lambda^k Ad^k \tag{27}$$

teljesül, ezért

$$< r^{k+1}, d^k > = < r^k + \lambda^k Ad^k, d^k > = < r^k, d^k > + \lambda^k < Ad^k, d^k > . \tag{28}$$

Ha felhasználjuk, hogy $\lambda^k = -\frac{<r^k, d^k>}{<d^k, Ad^k>}$, akkor látható, hogy $< r^{k+1}, d^k > = 0$, tehát az x^{k+1}-beli maradék vektor (a gradiens) és az x^k-beli keresési irány merőlegesek.

Ahhoz, hogy az iterációt folytathassuk, meg kell határozni az x^{k+1}-beli keresési irányt, azaz d^{k+1}-et. Az új keresési irányt úgy fogjuk meghatározni, hogy

$$< d^{k+1}, Ad^j > = 0 \tag{29}$$

teljesüljön minden $0 \leq j \leq k$ esetén.

A fenti összefüggéseknek több keresési irány is megfelel. Ez azt jelenti, hogy a lehetséges d^{k+1} keresési irányok közül kell választanunk a legjobbat. Ezt a választást segíti a következőekben taglalt módszer.

Legyen $V_k = span\{d^0, d^1, \ldots, d^{k-1}, d^k\}$ halmaz, minden iterációs lépés után a d^{k+1} aktuális keresési irány vektorával bővül. Tegyük fel, hogy a k-adik gradiens merőleges az összes többi korábbi keresési irányra, azaz

$$< r^k, d^j > = 0,\ 0 \leq j \leq k-1, \tag{30}$$

ahol $k \geq 1$. Most a $(k+1)$-edik gradienst szeretnénk vizsgálni. A már említett $r^{k+1} = r^k + \lambda^k A d^k$ egyenlőség miatt

$$< r^{k+1}, d^j > = < r^k + \lambda^k A d^k > = < r^k, d^j > + \lambda^k < A d^k, d^j > = \tag{31}$$

$$= < r^k, d^j > + \lambda^k < d^k, A d^j > . \tag{32}$$

Mivel feltételünk szerint $< r^k, d^j > = 0$, ha $0 \leq j \leq k-1$, ezért ha $< d^k, A d^j > = 0$ teljesül $0 \leq j \leq k$ esetén, akkor

$$< r^{k+1}, d^j > = 0,\ 0 \leq j \leq k-1. \tag{33}$$

Ha figyelembe vesszük azt is, hogy korábban beláttuk az $< r^{k+1}, d^k > = 0$ egyenlőséget is, akkor látható, hogy a $(k+1)$-edik gradiens is merőleges az összes korábbi keresési irányra. Ezért a $(k+1)$-edik keresési irányt is úgy fogjuk meghatározni, hogy

$$< d^{k+1}, A d^k > = 0 \tag{34}$$

teljesüljön.

Az új keresési irányt a gradiensvektor és az előző keresési irány lineáris kombinációjaként keressük:

$$d^{k+1} = -r^{k+1} + \mu_k d^k \tag{35}$$

(Feltételezzük, hogy $d^0 = -r^0$)

A (34) feltétel akkor fog teljesülni, ha

$$0 = < -r^{k+1} + \mu_k d^k, A d^k > = \mu_k < d^k, A d^k > - < r^{k+1}, A d^k > \tag{36}$$

azaz

$$\mu_k = \frac{< r^{k+1}, A d^k >}{< d^k, A d^k >}. \tag{37}$$

Be lehet látni, hogy ekkor

$$< d^{k+1}, Ad^j >= 0, \quad 0 \leq j \leq k \tag{38}$$

is teljesül.

Szintén be lehet látni, hogy

$$< r^{k+1}, r^j >= 0, \quad 0 \leq j \leq k \tag{39}$$

teljesül: ugyanis abból, hogy

$$< r^{k+1}, d^j >= 0, \quad 0 \leq j \leq k, \tag{40}$$

és abból, hogy $d^j = -r^j + \mu_{j-1} d^{j-1}$ igaz következik, hogy

$$0 =< r^{k+1}, -r^j + \mu_{j-1} d^{j-1} >= - < r^{k+1}, r^j > + \mu_{j-1} < r^{k+1}, d^{j-1} >= - < r^{k+1}, r^j > . \tag{41}$$

Ez azt jelenti, hogy a gradiensvektorok ortogonálisak, így ha nincsennek keresési hibák, akkor az algoritmus legfeljebb n lépés után megáll. Ez azért van, mert $\Re^n$-en maximum n db egymásra kölcsönösen merőleges vektort tudunk előállítani.

A gradiensvektorok ortogonalitásából következik az is, hogy a λ^k és μ_k együtthatókat másképpen is kiszámolhatjuk. Mivel $d^k = -r^k + \mu_{k-1} d^{k-1}$, ezért

$$- < r^k, d^k >= < r^k, r^k > \tag{42}$$

teljesül, továbbá így

$$\lambda^k = \frac{< r^k, r^k >}{< Ad^k, d^k >}. \tag{43}$$

Hasonlóan, felhasználva, hogy $r^{k+1} = r^k + \lambda^k A d^k$ következik

$$< r^{k+1}, Ad^k >= \frac{1}{\lambda_k} < r^{k+1}, r^{k+1} - r^k >= \frac{1}{\lambda_k} < r^{k+1}, r^{k+1} > \tag{44}$$

azaz

$$\mu_k = \frac{< r^{k+1}, Ad^k >}{< d^k, Ad^k >} = \frac{1}{\lambda_k} \frac{< r^{k+1}, r^{k+1} >}{< d^k, Ad^k >} = \frac{< r^{k+1}, r^{k+1} >}{< r^k, r^k >}. \tag{45}$$

Az egyszerű gradiens módszer a konjugált gradiens módszer egyszrűsített változata, ám kevésbé hatékony. Az egyszerű gradiens módszernél μ_k nullával egyenlő.

Mivel $d^0 = -r^0$, ezért $V_k = span\{r^0, r^1, \ldots, r^k\}$.

A konjugált gradiens pszeudó kódja a következő:

1. $r^0 := Ax^0 - b,\ d^0 := -r^0$

2. $k := 0(1) maxit$

3. $[? \parallel r^k \parallel \leq \epsilon \quad ? \quad [\text{stop eredmény} : x^k]$

4. $\lambda_k := \parallel r^k \parallel^2 / (Ad^k, d^k),\ x^{k+1} := x^k + \lambda_k d^k,\ r^{k+1} := r^k + \lambda_k A d^k$

5. $\mu_k := \parallel r^{k+1} \parallel^2 / \parallel r^k \parallel^2,\ d^{k+1} := -r^{k+1} + \mu_k d^k]_k$

Fontos megjegyezni, hogy az algoritmus minden lépése jól definiált. Ugyanis, ha $d^k = 0$, akkor

$$0 = < r^k, d^k > = - < r^k, r^k >, \tag{46}$$

azaz $r^k = 0$, így a harmadik lépésben a leállási feltétel teljesül. Ekkor $Ax^k = b$, tehát x^k az egyenletrendszer megoldása.

A harmadik lépésben választhatunk tetszőleges normát, nem kell, hogy euklideszi normát alkalmazzuk. Az algoritmus minden lépésben egy mátrix-vektor szorzatot: (Ad^k), két skalárszorzatot: $\parallel r^{k+1} \parallel^2$, (Ad^k, d^k), és három $\lambda d + x = w$ alakú vektorkombinációt kell kiszámítanunk. A konjugált gradiens módszert elsősorban nagyméretű, ritka mátrixok esetén érdemes használni.

5.5.1. Implementáció

A konjugált gradiens módszert *konjugalt_gradiens.c* néven implementáltam. A program $n, A, b, x0, maxit, e$ -t vár bemenetként. A kimenet x, lépésszám, eltelt idő. Ez a módszer közel azonos lépésszámban és közel annyi idő alatt talál megoldást, mint a Jacobi-iteráció, vagy a Gauss-Seidel iteráció.

5.6. A prekondícionált konjugált gradiens módszer

A prekondícionált konjugált gradiens módszer azért fontos, mert a rosszul kondícionált mátrixú egyenletrendszerek megoldáshoz a konjugált gradiens módszer lassan konvergál. Ezért célszerű a perkondicionálás alkalmazása. Ez azt jelenti, hogy az eredeti $Ax = b$ rendszer helyett a $P^{-1}Ax = P^{-1}b$ rendszert vizsgáljuk, ahol $P^{-1}A$ közel kell, hogy essen az egységmátrixhoz $P \approx A$. Ha a mátrix szimmetrikus, akkor olyan H mátrixot keresünk, melyre $A \approx HH^T$ teljesül, ahol H reguláris mátrix. Ekkor az $Ax = b$ rendszer helyett a

$$(H^{-1}AH^T)(H^T x) = H^{-1}b, \tag{47}$$

azaz a

$$\tilde{A}x = \tilde{b}, \tag{48}$$

rendszert tekintjük, ahol $\tilde{A} = H^{-1}AH^T$, $\tilde{x} = H^T x$, $\tilde{b} = H^{-1}b$.

A $P \approx A$, illetve $A \approx HH^T$ teljesülését olyan értelemben várjuk, hogy

$$c = cond(P^{-\frac{1}{2}}AP^{-\frac{1}{2}}),$$

illetve

$$c = cond(H^{-1}AH^{-T})$$

kicsi legyen $cond(A)$-hoz képest (nem arról van szó, hogy A és P illetve HH^T elemei egymáshoz közeliek legyenek).

A prekondícionált konjugált gradiens módszer előnye abban rejlik, hogy ha az $Ax = b$ egyenletrendszerben A olyan mátrix, aminek nagy a kondíciószáma, akkor is megoldást kaphatunk. Persze a módszer sikere a prekondícionálási mátrix megválasztásán múlik. Prekondícionálási mátrixot elő lehet állítani Inkomplett Cholesky felbontás, Jacobi iteráció, szimmetrikus Gauss-Seidel iteráicó vagy Successive overrelaxation (SOR) módszerek segítségével. Én az Inkomplett Cholesky felbontást választottam, amit a következő fejezetben mutatok be részletesen. A legegyszerűbb prekondícionálási mátrix az olyan mátrix aminek csak a diagonális elemei nem nullák, a többi elem nulla.

A prekondícionált konjugált gradiens módszer pszeudó kódja a következő:

1. $g^0 := Ax^0 - b$, $\underline{HH^T e^0 = g^0}$, $h^0 := -e^0$
2. $k := 0(1)maxit$
3. $[?\ \|\ g^k\ \| \leq \epsilon ?\quad [\text{stop}\quad \text{eredmény:} x^k]$
4. $\lambda_k := (e^k, g^k)/(Ah^k, h^k)$, $x^{k+1} := x^k + \lambda_k h^k$, $g^{k+1} := g^k \lambda_k Ah^k$
5. $\underline{HH^T e^{k+1} = g^{k+1}}$
6. $\mu_k := (e^{k+1}, g^{k+1})/(e^k, g^k), h^{k+1} := -e^{k+1} + \mu_k h^k]_k$

A kódban az aláhúzott kifejezések egy-egy lineáris egyenletrenszer megoldását jelentik.

5.6.1. Implementáció

A prekondicionált konjugált gradiens módszert *prekondicionalt_konjugalt_gradiens.c* néven implementáltam. A program $n, A, H, b, x0, maxit, e$ -t vár bemenetként. A kimenet x, lépésszám, eltelt idő. A program $HH^T e^0 = g^0$-t és $HH^T e^{k+1} = g^{k+1}$-t a *lin_e_r* nevű függvény meghívásával számolja ki. Itt HH^T egy mátrix, e^0 és e^{k+1} jelenti a keresett x oszlopvektort és g^0 és g^{k+1} jelenti az egyenletrendszer jobb oldalán álló b oszlopvektort. A *lin_e_r* függvény az

H mátrixot, a b vektort, az n-t és egy int típusú hiba paramétereket várja. Az eredményvektornak dinamikusan foglalok helyet a hívó *Prekondicionalt_Konjugalt_gradiens* függvényben és a *lin_e_r* függvény ebbe fogja visszadni a megoldást. Memóriatakarékosság miatt ez az eredményvektor a $HH^Te^0 = g^0$ lineáris egyenletrendszer g^0 vektora lesz.

A *lin_e_r* függvény pszeudó kódja a következő:

1. /*$Hy = b$*/
2. for i=1:n
3. for j=1:i-1
4. $b_i = b_i - h_{i,j} * b_j$
5. end
6. $b_i = b_i / h_{ii}$
7. end
8. /*$H^Tx = y$*/
9. for i=n:1
10. $b_j = b_j / h_{jj}$
11. for j=1:j-1
12. $b_i = b_i - h_{j,i} * b_j$
13. end
14. end

ahol $h_{i,j}$ a H mátrix i. sorának j. eleme. Itt a H mátrix elemeit sorfolytonosan járjuk be, amire azért volt szükség, mert ezt a módszert is a későbbiekben tárgyalt gyorsítással implementáltam. A módszerek gyorsításánál a ritkamátrixot sorfolytonosan tárolom és ahhoz, hogy oszlopfolytonosan járjam be, ahhoz a ritka mátrix elemeit egyessével kellett volna lekérdeznem. Ez megnövelte volna az algoritmus futási idejét. Ám a *lin_e_r* függvény használatával gyorsítani lehetett a kódot, mivel abban a mátrixot sorfolyotonosan kell bejárni. A *lin_e_r* függvény csak akkor használható, ha H alsó háromszög mátrix, ami a prekondícionált konjugált gradiens esetében igaz, mivel H prekondícionáló mátrix alsó háromszög mátrix.

A pszeudó kód működése a következő. A $HH^Tx = b$ lineáris egyenletrendszer megoldását két részre bontjuk: $Hy = b$ és $H^Tx = y$. A két lineáris egyenletrendszert könnyen megoldható, mert a H alsó háromszög mátrix, illetve H^T felső háromszög mátrix. A *lin_e_r* függvényben is takarékoskodom a memóriával, mert az y vektort is a b vektorban tárolom, így a megoldás is a b vektorban áll elő.

A teszteléseknél kétféle prekondícionálási mátrixra teszteltem: az egyik amikor a főátló az A mátrix főátlójával egyezik meg, a többi elem nulla, illetve a másik, amikor az inkomplett Cholesky felbontással határozom meg a H mátrixot.

6. Inkomplett Cholesky felbontás

A prekondicionált konjugált gradiens módszernél, ha alkalmas prekondicionálási mátrixot választunk, akkor kevesebb lépésszámban és kevesebb idő alatt kapunk eredményt.

A prekondicionálási mátrix meghatározására az egyik lehetőség az inkomplett Cholesky felbontás. A felbontás lényege, hogy számoljunk ki egy H háromszögmátrixot amely az A mátrix eredeti Cholesky felbontásában szereplő L mátrixhoz közelít. Ezután a prekondicionáló M mátrix legyen $M = HH^T$. Az inkomplett felbontást úgy készítjük el, hogy a Cholesky felbontás lépéseit csak az A mátrix nem nulla elemeire hajtjuk végre. Az Inkomplett Cholesky felbontás pszeudo kódja a következő:

1. $for\ k = 1 : n$
2. $\quad A(k,k) = \sqrt{A(k,k)}$
3. $\quad for\ i = k+1 : n$
4. $\quad\quad if\ A(i,k) \neq 0$
5. $\quad\quad\quad A(i,k) = A(i,k)/A(k,k)$
6. $\quad\quad end$
7. $\quad end$
8. $\quad for\ j = k+1 : n$
9. $\quad\quad for\ i = j : n$
10. $\quad\quad\quad if\ A(i,j) \neq 0$
11. $\quad\quad\quad\quad A(i,j) = A(i,j) - A(i,k) * A(j,k)$
12. $\quad\quad\quad end$
13. $\quad\quad end$
14. $\quad end$
15. end

Ennél a felbontásnál csak a nem nulla elemekkel számolunk. Az előállított mátrix alsó háromszögmátrix lesz. Így értékes elem a főátlóban és az alsó háromszögmátrixban lesz. A többi elem nulla lesz.

6.0.2. Implementáció

Az inkomplett Cholesky felbontást *Inkomlett_Cholesky.c* néven implementáltam. A program n, A -t vár bemenetként. A program kimenete egy nxn-es alsó háromszög mátrix, mely a prekondicionált konjugált gradiens módszernek lesz az alkalmas prekondicionálási mátrixa. A program a kapott A mátrixot letárolja, és az eredményt is ebben a mátrixban állítja elő. Az A mátrixban értékes elemek az alsó háromszögrészében, és a főátlóban vannak. A többi elemet kinullázza a program.

7. Az implementált módszerek gyorsítása

Ebben a fejezetben szeretnék bemutatni egy olyan módszert, melyet ha adaptálunk egy iteratív módszer implementációjába, akkor a programunk futási ideje, és memória igénye egyaránt csökken. Ezt a módszer Noszály Csaba tanár úr javasolta számomra. A módszer neve compressed row format melyet lancolt lista segitsegevel valositottam meg. A módszert C programozási nyelven implementáltam, majd adaptáltam az előzőleg már bemutatott iteratív módszerek implementációjába.

Igazán akkor látszik, hogy az ezzel a módszerrel implementált program gyorsabb, mint egy átlagos módon implementált program, amikor az nxn-es mátrixunk elég nagy és kevés nem nulla elemet tartalmaz. Az átlagos módon implementált program alatt az olyan programot értem, ahol nxn-es mátrix minden elemét letárolom, annak ellenére, hogy a mátrix nagy része nulla elemeket tartalmaz.

A módszer lényege, hogy ne tároljunk nulla elemeket és ne számoljunk velük, mert csak feleslegesen pazarolják a processzor időt és memóriát. Illetve ha összeszeretnénk szorozni egy mátrixot egy vektorral, ne kérdezzük le folyamatosan a mátrix i, j-edik elemét, mert ez mind egy-egy keresést eredményez a memóriában.

Ezek helyett tegyük a következőket. Tároljuk le a mátrix minden sorának nem nulla elemeit egy vektorba, így egy ilyen vektor a mátrix egy sorát fogja helyettesíteni. A mátrixot ilyen formában letárolva, n db vektort kapunk, melyeket ha sorrendben egymás alatt képzelünk el, akkor fésű alakú lesz, mivel csak a nem nulla elemeket tároltuk. Ám ekkor egy elemről nem tudjuk, hogy az eredeti mátrixnak mely eleme volt, csak azt tudjuk, hogy mely sorában szerepelt. Ehhez kellene egy plusz információ, hogy mely oszlopban volt eredetileg. Ezt én úgy oldottam meg, hogy a következő struktúrát definiáltam a programok elején:

```
typedef struct sor
{
        int oszlop;
        double ertek;
        struct sor *kov;
} SOR;
```

Egy ilyen struktúra egy ritka mátrixbeli elemet ír le. Egy ritka elem tárolására így huszonnégy byte szükséges, mert az *int* típus négy byte, a *double* típusnak nyolc byte, a *struct sor* * pedig négy byte, így összesen tizenhat byte memóriára van szükség. Ez csak akkor igaz, ha a számítógépünknek 32 bites processzora van.

Itt szeretném megemlíteni, hogy milyen környezetben implementáltam és futtattam a programokat. Erre azért van szükség, mert az a számítógép, amelyen én teszteltem annak 64 bites

processzora van és ez befolyásolja a struktúra méretét.

Processzor : AMD Athlon(tm)64 X2 Dual-Core Processor TK-55 1.8GHz,

Memória : 2686 MB,

Operációs rendszer : Windows Vista Home Basic 32 bites.

Ekkor a struktúra tárolására huszonnégy byte-ra van szükség.

A programok fordítására és futtatására a Cygwin programot használtam, a gcc verziója 3.4.4. Fordításhoz a következő parancsot használtam: gcc -ansi -Wall -o nev nev.c

Minden sorhoz készítettem egy egyirányban láncolt listát, melynek minden eleme egy ilyen struktúra lesz. Deklaráltam egy A_sorai nevű n elemű vektort, melyben a listák fej mutatóit tároltam. Ha például szeretnénk elvégezni egy mátrix-vektor szorzást, akkor ahelyett, hogy sorfolytonosan bejárnánk a mátrixot, és a megfelelő elemet a megfelelő elemmel összeszoroznánk, egy for ciklusban $k = 0, \ldots, n$-ig bejártam az összes listát.Ezen a for cikluson belül egy újabb for ciklusban $j = 0, \ldots, n$-ig bejártam a vektort és néztem hogy ha j egyenlő e a lista aktuális elemének oszlop értékével. Ha egyenlő, akkor elvégezhető a szorzás, amit külön kell tárolnom egy változóba, majd léphetek a lista következő elemére, illetve j-t is növelhetem eggyel. Ha nem egyenlő, akkor csak a j-t kell növelni eggyel. A j változó szerinti ciklussal addig megyek, míg $j < n$, illetve nem értem el a lista végét.

Azért választottam vektorok helyett a láncolt listákat, mert előre nem tudom, hogy a mátrix egy sorában hány db nem nulla elem van, s vektorok esetében ekkor vagy túl nagy méretet foglalok le, vagy mindig újra helyet kell foglalni a realloc segítségével. Ez így nem biztos, hogy elég hatékony lesz memória használat terén. Viszont láncolt listák esetében valóban csak annyi helyet foglalok, amennyi ritka elem van a mátrixban.

Ha nem csupán egy mátrix vektor szorzásról van szó, hanem például a következő vektort szeretnénk kiszámolni:

$$g_k = g_0 + l * Ad_0,$$

ahol g_k, g_0, d_0 azok vektorok, l az egy szám, és A az egy mátrix. Ekkor az előző mátrix vektor szorzásba beépíthetjük azt is, hogy amikor az A mátrix i-edik sorát megszoroztam a d_k vektorral, akkor kapok egy számot, ezt megszorzom l-el, majd hozzáadom a g_0 vektor i-edik elemét, s ezt érétkül adom a g_k vektor i-edik elemének. Míg hogyha külön vektorba számolnám ki Ad_k mátrix-vektor szorzatot, majd ezt a vektort megszoroznám l-el, aztán összeadnám g_0 vektorral, akkor ez sokkal több processzor időt igényelne, mint az előző esetben. Az elozó példát a konjugált gradiens módszerből vettem, s ezt C nyelven egy eljárásban implementáltam:

```
void G_k( SOR **sorok, double *g0, double lk, double *d0,
```

```
    int n, double *gk ){

  int i, j;
  for(i=0; i<n; i++){
    SOR *x=*(sorok+i);
    double sum=0.0;
    for ( j=0; j<n && x!=NULL; j++ ){
      if ( x->oszlop==j && d0[j]!=0.0 ){
        sum += x->ertek * d0[j];
        x=x->kov;
      }
      else if ( x->oszlop==j )
        x=x->kov;
    }
    gk[i] = g0[i]+lk*sum;
  }
}
```

Ebben az eljárásban *SOR **A_sorai* lesz az a vektor ami a láncolt listák fej mutatóit tartalmazza, *g0*, *d0* és l bemenetek. Illetve a *gk* vektorba adom vissza a számolás eredményét.

A már említett Jacobi-iteráció, konjugált gradiens és prekondícionált konjugált gradiens módszerek közül csak az utólsó két módszert tudtam gyorsítani. Ez azért volt, mert a Jacobi-iterciónál csak egyetlen helyen lehetett ezt a gyorsítást implementálni. A másik két módszernél viszont több olyan hely is volt, ahol a fenti mátrix-vektor szorzáshoz gyorsítást tudtam elvégezni. Ezért a továbbiakban a gyorsított módszerek alatt ezt a két módszert értem.

Az előbbi két módszerrel kapcsolatban az a tapasztalatom, hogy bár maga az algoritmus gyors, de a beolvasási idő túl hosszú. Ez a probléma főleg nagy méretű mátrixoknál jelentkezik. Ezért a módszerek bemeneteit átalakítottam úgy, hogy az A illetve H mátrixok ritka mátrixformában legyenek úgy, mint a Matrix Market(MM) ritka mátrix reprezentációban, azzal a különbséggel, hogy az indexek nem $1, \ldots, n$, hanem $0, \ldots, n-1$ közöttiek lehetnek. Így a módszerek bemenetei és ezzel a beolvasási idejük is drasztikusan lecsökkent. Ám ehhez a módszerek beolvasását is meg kellett változtatni, tehát mostmár nem egy mátrixot kell beolvasni, hanem adott számú sort ahol egy sorban egy elemhármas reprezentál egy ritka elemet. A beolvasott sorokat a megfelelő módon le kellett tárolnom az egyirányban láncolt listákba, úgy, hogy egy i, j, m elemhármas azt jelenti, hogy az i-edik sor j-edik oszlopában, illetve ha $i! = j$, akkor j-edik sor i-edik oszlopában is m érték szerepel. De ahhoz, hogy mindig a megfelelő lista

végére tudjak beszúrni, mindig be kellett járnom a teljes listát, hogy eljussak a legégére. Ez még mindig lassúnak bizonyult, ezért ehelyett mindig a lista elejére szúrtam be, az új elemet, majd rendeztem a sorokat az elemek oszlop értéke alapján. Mivel a memóriafoglalás és felszabadítás időigényes folyamat, ezért minden listát ideiglenesen két vektorban tároltam: külön az elemek oszlop értékét, és külön az értékét. Ezután ezt a két vektort párhuzamosan rendeztem a beszúró rendezés algoritmusával, majd a két vektort visszaillesztettem a listába, felülírva a benne lévő elemeket. Ezáltal a lista által lefoglalt memória területet nem kellett felszabadítanom, majd újra lefoglalnom és egyben feltöltenem. A rendezést két eljárással valósítottam meg, amelyeknek a C implementációja a következő:

```
void rendez( SOR ** f, int n){
  int i;

  for( i=0; i<n; i++){
    SOR *x=f[i];
    int sorhossz;
    for( sorhossz=0; x!=NULL; sorhossz++ ) x=x->kov;
    if ( sorhossz > 2 && sorhossz>0){
      int *rendezett_oszlop = (int *) calloc
        ( sorhossz, ( sizeof ( int ) ) );
      double *rendezett_ertek = (double *)
        calloc( sorhossz, ( sizeof ( double ) ) );
      for( x=f[i], sorhossz=0; x!=NULL; sorhossz++, x=x->kov ){
        rendezett_ertek[sorhossz]=x->ertek;
        rendezett_oszlop[sorhossz]=x->oszlop;
      }
      Beszuro_rendezes(rendezett_ertek,
       rendezett_oszlop, sorhossz);
      for( x=f[i], sorhossz=0; x!=NULL; sorhossz++, x=x->kov ){
        x->ertek=rendezett_ertek[sorhossz];
        x->oszlop=rendezett_oszlop[sorhossz];
      }
      free(rendezett_ertek);
      free(rendezett_oszlop);
    }
  }
```

```
}
void Beszuro_rendezes(double *A, int *oszlop, int hossz){
  if(hossz<2) return;
  int i,j;

  double kulcs_e;
  int kulcs_o;
  for( j=1; j<hossz; j++ ){
    kulcs_e=A[j];
    kulcs_o=oszlop[j];
    i=j-1;
    while ( i>=0 && oszlop[i]>kulcs_o ){
      A[i+1]=A[i];
      oszlop[i+1]=oszlop[i];
      i=i-1;
    }
    A[i+1]=kulcs_e;
    oszlop[i+1]=kulcs_o;
  }
}
```

7.1. A gyorsított módszerek futási ideje

Ebben az alfejezetben összehasonlítottam a konjugált gradiens módszer és a prekondícionált konjugált gradiens módszer régi és gyorsítottott változatát, néhány tesztesetre mutatott futási idejét, illetve a bemenetek méretét. A gyorsított módszerek bemenetei ritka mátrix formában vannak.

A mátrixokat a Bevezetésben említett *http://www.cise.ufl.edu/research/sparse/matrices/* oldalról vettem és az ott megadott hivatkozási számokat használtam.

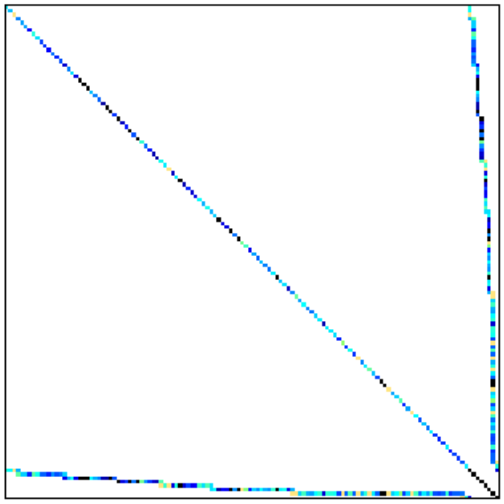
Konjugált gradiens módszer

1401-es mátrix	A bemenet mérete	futási idő(sec)	beolvasási idő(sec)	alg. futási idő(sec)
régi módszer	110,8M	82,673	16,352	66,321
gyorsított módszer	171k	31,903	0,055	31,848

2208-as mátrix	A bemenet mérete	futási idő(sec)	beolvasási idő(sec)	alg. futási idő(sec)
régi módszer	1,5M	2,124	0,262	1,862
gyorsított módszer	66,5k	1,3	0,033	1,267

Látható, hogy a gyorsított módszer ritka mátrix bemenettel az két és félszer gyorsabb, mint a régi módszer a 1401-es mátrixra, illetve másfélszer gyorsabb az 2208-es mátrixra.

Az 1401-es mátrix mérete 2541x2541 és 7361 db nem nulla elemet tartalmaz. A mátrix nem nulla elemeinek elhelyezkedését ábrázolja az 1. ábra.



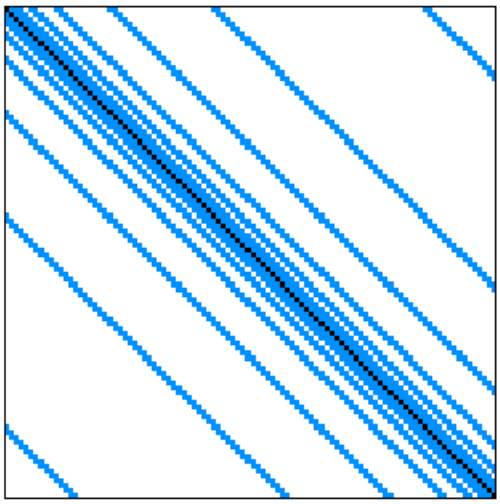
1. ábra.

A 2208-as mátrix mérete 300x300 és 4678 db nem nulla elemet tartalmaz. A mátrix nem nulla elemeinek elhelyezkedését ábrázolja a 2. ábra.

Prekondícionált konjugált gradiens módszer

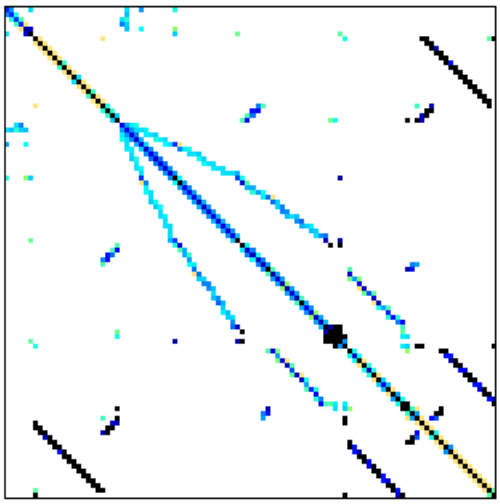
875-ös mátrix	A bemenet mérete	futási idő(sec)	beolvasási idő(sec)	alg. futási idő(sec)
régi módszer	3,2M	643,836	0,499	643,337
gyorsított módszer	40,9k	2,924	0,029	2,895
221-es mátrix	A bemenet mérete	futási idő(sec)	beolvasási idő(sec)	alg. futási idő(sec)
régi módszer	7,5M	2121,044	1,229	2119,815
gyorsított módszer	101,5k	5,257	0,04	5,217

Itt is látható a sebességnövekedés. A gyorsított módszer ritka mátrix bemenettel a 875-ös mátrixra körülbelül kétszázhúszszor gyorsabb, illetve a 221-es mátrixra körülbelül négyszázhá-romszor gyorsabb, mint a régi módszer.



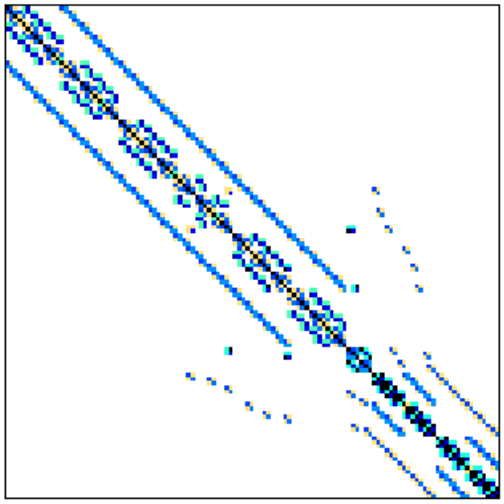
2. ábra.

A 875-ös mátrix mérete 306x306 és 2018 db nem nulla elemet tartalmaz. A mátrix nem nulla elemeinek elhelyezkedését ábrázolja a 3. ábra.



3. ábra.

Az 221-es mátrix mérete 468x468 és 5172 db nem nulla elemet tartalmaz. A mátrix nem nulla elemeinek elhelyezkedését ábrázolja a 4. ábra.



4. ábra.

8. Tesztelés

Az iterációs módszereket több nagy méretű ritka mátrixszal is leteszteltem. A tesztesetekhez ilyen mátrixokat a *http://www.cise.ufl.edu/research/sparse/matrices/* oldalról töltöttem le. *Dr. Baran Ágnes* tanárnő segítségével kerestünk ezen mátrixok közül olyanokat, amelyeknek viszonylag kicsi a kondíciószáma, nagyméretű ritka mátrix, szimmetrikus és pozitív definit.

Az említett weboldalon a mátrixok háromféle ritka mátrix reprezentációban vannak tárolva: Matlab mat-file, Matrix Market formátum, Rutherford/Boeing formátum. Én a Matrix Market formátumot választottam. A Matrix Market formátumban megadott ritka mátrix fájl .mtx kiterjesztésű. Ebben a fájlban az elején % karakterrel kezdődnek a megjegyzések amelyek a mátrixra, és a tárolás módjára vonatkoznak. Ezután egy sorban van három szám: a mátrix sorainak száma, a mátrix oszlopainak száma és hogy a mátrix hány darab ritka elemet tartalmaz. Majd minden sorban egy számhármas reprezentál egy ritka elemet: i, j, d. Ami azt jelenti, hogy a mátrix i-dik sorában és j-edik oszlopában egy d szám szerepel, vagyis $a_{ij} = d$. Fontos megemlíteni, hogy a nem csak az $a_{ij} = d$, hanem $a_{ji} = d$. Ez azért van, mert a mátrix szimmetrikus ezért a ritka mátrix reprezentációban nem tárolódik duplán ugyanaz a szám, hogy ezzel is kevesebb helyet foglaljon az állomány.

Ahhoz, hogy az általam implementált módszerek ezekkel a mátrixokkal dolgozni tudjanak, a ritka mátrix reprezentációt át kellett konvertálni, illetve olyan bemenetet készíteni belőle, amit az egyes módszerek várnak. Ezért megírtam az $mm2matrix.c$ programot. Amely első argumentumként megkapja, azt az $.mtx$ kiterjesztésű állományt, amiben a ritka mátrix reprezentáció

van letárolva. Második argumentumként megkapja azt a fájlnevet, amibe a program le fogja tárolni a ritka mátrixot és azokat a plusz információkat, amelyeket az adott módszer vár majd bemenetként. A program futásakor ki kell választani, hogy milyen módszerhez akarunk bemenetet gyártani. Majd olyan dolgokat vár bemenetként, mint $maximálisiterációszám, w, e$, stb... A többit dolgot a program kiírja pl.: az n érékét az $.mtx$ kiterjesztésű fájlból kiolvassa, az x kezdővektor elemei egytől n-ig felsorolva, a b vektor az úgy lett megválasztva, hogy az eredmény vektor csupa egyesekből áll. A tesztelés eredményeit az A) FÜGGELÉKBEN -ben írtam le. Ebben a táblázatban jelöltem a mátrixok sorszámát, méretét, kondíciószámát és a nem nulla elemek számát. A tesztesetekhez felsoroltam, hogy a különböző módszerek milyen pontosságú eredményt adtak a tesztelésen, hány lépésben és mennyi idő alatt adtak eredményt. Azon mátrixok esetén lett pontatlan az eredmény, amelyeknek túl nagy a kondíciószáma. Ebben a függelékben szemmelláthtató a módszerek konvergenciája és gyorsasága, egy adott mátrix esetében.

8.1. A tesztelésből levonható következtetések

Ebben az alfejezetben szeretnék felsorolni néhány következtetést, amelyeket az A) FÜGGELÉKBEN -ben található teszterdményekből vontam le. Ezek a következtetések a tesztmátrixok kondíciószámával és a módszerek által adott megoldás pontosságával és az iterációs lépések számával függnek össze.

Az LDL^T felbontás az egyetlen direkt módszer, melyet a különböző teszteseteknél a pontosság összehasonlítása miatt tárgyalok.

A konjugált gradiens módszer és a prekondícionált konjugált gradiens módszer, azoknál a teszteseteknél, ahol a kondíciószám kicsi, ott közel azonos pontossággal és majdnem egyenlő iterációs lépéssel ad megoldást.

Kis kondíciószámú mátrixok esetén, mint például 872-es és 873-as mátrixok esetén a SOR módszer pontosabb eredményt ad, mint az egyszerű gradiens módszer. Illetve az egyszerű gradiens módszernek körülbelül kétszerannyi iterációs lépésre van szüksége. Nagy kondíciószámú mátrixok esetén, mint például a 876-os és 2206-os mátrixok esetén, az egyszerű gradiens módszer adja a pontosabb eredményt. Viszont az iterációs lépések száma már a SOR módszer iterációs lépéseinek számának sokszorosa.

A 872-es és 874-es mátrixok esetén, melyek kis kondíciószámúak, a SOR módszer és a konjugált gradiens módszer közel azonos pontossággal és közel azonos iterációs lépésben adnak eredményt. Ám nagy kondíciószámú mátrixoknál, mint például a 2204-es, a 2206-os és a 2208-as mátrixok, a pontosság tekintetében a konjugált gradiens módszer a pontosabb, de előfordul, hogy az iterációs lépések száma több.

A SOR iteráció minden esetben pontosabb eredményt szolgáltat, mint a Jacobi iteráció vagy a Gauss-Seidel iteráció.

Kis kondíciószám esetén a konjugált gradiens módszer, a prekondícionált konjugált gradiens módszer a maximum egy-két tizedesjeggyel adnak pontosabb megoldást, viszont nagy kondíciószám esetén már sokkal pontosabb megoldást adnak, mint a Jacobi iteráció vagy a Gauss-Seidel iteráció.

Összefoglalva azt mondhatjuk, hogy a tárgyalt iterációs módszerek közül a prekondícionált konjugált gradiens módszer az összes tesztesetre nézve a leggyorsabb és legpontosabb eredményt szolgáltató módszer. Ez az állítás kis és nagy kondíciószámú mátrixokra egyaránt igaz. De ennek az az ára, hogy amikor előállítom a szükséges bemenetet, akkor a mátrixból elő kell állítani a prekondícionáló mátrixot az inkomplett Cholesky felbontással, ami nagy mátixok esetén időigényes lehet. A módszer pontossága nagyon közelít az LDL^T direkt módszer pontosságához. De ha a mátrix méretéhez képest sok nem nulla elem található a mátrixban, mint például a 2204-es és 2206-os mátrixok esetén, akkor az LDL^T felbontás pontosabb eredményt ad.

Vegyük figyelembe azt, hogy az összes iterációs módszer esetén a pontosságot nagyban befolyásolja a kezdővektor, illetve a SOR módszernél a w helyes megválasztása. Tesztelésnél az összes módszernek ugyanazt a kezdővektort és pontosságot, illetve a SOR módszernél a w-t 1.3-nak választottam.

9. Összefoglalás

Dolgozatomban sikerült elmélyülnöm a lineáris egyenetrendszerek megoldásával kapcsolatos módszerekben. A dolgozat elején bemutattam ezen módszerek két nagy osztályát: a direkt módszereket és az iterációs módszereket azok előnyeivel és hátrányaival együtt. Egyetlen direkt módszert tárgyaltam, LDL^T felbontást. Erre azért volt szükség, hogy a tesztelésnél a futási eredményeket össze tudjam hasonlítani. Ahhoz, hogy a prekondícionált konjugált gradiens módszerhez elő tudjuk állítani a prekondícionálási mátrixot, szükségem volt az inkomplett Cholesky felbontásra, így ezt a módszert is implementáltam.

Ezt követően néhány iterációs módszert tárgyaltam részletesen. Ismertettem a különböző módszerek elméleti hátterét és pszeudó kódját. A módszerek mindegyikét pszeudó kódjuk alapján implementáltam C programozási nyelven.

Mivel nagy méretű ritka mátrixokkal foglalkoztam ezért a módszerek bemenetei esetenként túl nagyok voltak és a futási idők is hosszúak voltak. Emiatt a ritka mátrix reprezentációt kihasználva az összes iterációs módszer bemenetét átalakítottam ritka mátrix formájúra és a módszereket is átírtam. A gyorsítás módszerét *Az implementált módszerek gyorsítása* című fejezetben írtam le. Összehasonlítottam a régi és a gyorsított módszerek futási idejét és a bemenetek méretét néhány tesztesetre. Ezután teszteltem a már gyorsított módszereket különböző tesztesetekre és a futási eredményeket az A) függelékben írtam le. Majd a futási eredmények alapján az eltelt időt, a mátrix kondícószámát és az elvégzett iterációs lépések számát figyelmbe véve megfogalmaztam néhány következtetést.

Visszatekintve úgy gondolom, hogy a dolgozat írása és a módszerek implementálása közben, hasznos matematikai és programozási tapasztalatokkal gazdagodtam.

10. Köszönetnyilvánítás

Ezúton szeretnék köszönetet nyílvánítani konzulensemnek, Dr. Baran Ágnes tanárnőnek, aki egyetemi munkája és sok egyéb teendője mellett időt szakított a konzultációkra, és a felmerülő elméleti és implementálási kérdések megoldásában nyújtott segítséget. Külön köszönöm a matematikai és tartalmi hibákkal kapcsolatos észrevételeit. Köszönöm Noszály Csaba tanár úrnak, hogy implementálási javaslataival segített lecsökkenteni a módszerek futási idejét.

Természetesen köszönöm szüleimnek, akik támogatták egyetemi tanulmányaimat.

11. Irodalomjegyzék

- Stoyan Gisbert és Takó Galina: Numerikus módszerek 1 Typotex Kiadó Bp. 2002.
- Owe Axelson: Iterative Solution Methods Cambrdige University Press 1996.
- Golub, van Loan: Matrix Computations John Hopkins University Press Baltimore 1989.
- http://www.cise.ufl.edu/research/sparse/matrices/

12. A) FÜGGELÉK

Id: méret: nnz:	cond:	$x_0 - x_k$:	it.:	elteltidő(ms):
57 bcsstm02.mtx 66 66				
LDL^T		0.0000400388260575	–	30
Jacobi iteráció		0.0000400388260576	1	67
Jacobi Seidel		0.0000400388260576	1	45
SOR iteráció		0.0000399825798887	17	236
Egyszerű gradiens		0.0003538542930628	55	334
Konjugalt gradiens		0.0000400388260456	12	144
Prek. konj. grad.		0.0000400388170111	12	114
Prek. konj. grad. ink. Cholesky		0.0000400388258724	1	78
60 bcsstm05.mtx 153 153	12.6981			
LDL^T		0.0000181719060634	–	118
Jacobi iteráció		0.0000181719060631	1	89
Jacobi Seidel		0.0000181719060631	1	47
SOR iteráció		0.0000180936729073	18	187
Egyszerű gradiens		0.0000974930121703	91	536
Konjugalt gradiens		0.0000182873920699	19	207
Prek. konj. grad.		0.0000181762625804	20	201
Prek. konj. grad. ink. Cholesky		0.0000181719060616	1	100

64 bcsstm09.mtx 1083 1083	10^4			
LDL^T		26.8700577266950020	–	25655
Jacobi iteráció		26.8700577266950020	1	152
Jacobi Seidel		26.8700577266950020	1	97
SOR iteráció		26.8700572193432627	20	498
Egyszerű gradiens		24.0008927327134494	3	178
Konjugalt gradiens		26.8700577266903728	2	153
Prek. konj. grad.		26.8700577059495842	2	227
Prek. konj. grad. ink. Cholesky		26.8697823446894155	1	195
71 bcsstm21.mtx 3600 3600	23.71			
LDL^T		1.1140860942340953	–	1800073
Jacobi iteráció		1.1140860942340960	1	542
Jacobi Seidel		1.1140860942340960	1	518
SOR iteráció		1.1140858856974452	22	3709
Egyszerű gradiens		0.7368551500630151	59	11653
Konjugalt gradiens		1.1140860939018637	3	957
Prek. konj. grad.		1.1140860941968171	3	1946
Prek. konj. grad. ink. Cholesky		1.1140895818719629	1	951
72 bcsstm22.mtx 138 138	941.2946			
LDL^T		0.0480190197776541	–	99
Jacobi iteráció		0.0480190197776542	1	44
Jacobi Seidel		0.0480190197776542	1	50
SOR iteráció		0.0480189826995631	18	123
Egyszerű gradiens		0.7695706259063095	1277	2335
Konjugalt gradiens		0.0480045836693620	47	488
Prek. konj. grad.		0.0480437878499945	48	523
Prek. konj. grad. ink. Cholesky		0.0480190194117330	1	95

74 bcsstm24.mtx 3562 3562	$1.8 * 10^{13}$			
LDL^T		8.1240396961258430	–	1527540
Jacobi iteráció		8.1240396961258430	1	469
Jacobi Seidel		8.1240396961258430	1	524
SOR iteráció		8.1240396115339539	22	3722
Egyszerű gradiens		73977.1562781847169	1000	1474061
Konjugalt gradiens		26954.0602332219750	5000	
Prek. konj. grad.		10997364.8564792908	4999	2153414
Prek. konj. grad. ink. Cholesky		8.1240008295542	1	1008
221 nos5.mtx 468 5172	$2.92 * 10^4$			
LDL^T		0.0000000000004103	–	3749
Jacobi iteráció		0.0004711223290056	7340	32994
Jacobi Seidel		0.0004711223290110	7340	41470
SOR iteráció		0.0002524726564626	4114	24906
Egyszerű gradiens		44.3554731340985668	20000	149470
Konjugalt gradiens		0.0000000000118332	523	3839
Prek. konj. grad.		0.0000000000533866	389	4713
Prek. konj. grad. ink. Cholesky		0.0000000003015971	59	1026
229 plat362.mtx 362 5786	$7.08 * 10^{11}$			
LDL^T		101843.7131520515104057	–	430
Jacobi iteráció		394.1550631977571584	10000	50267
Jacobi Seidel		394.1550631977584658	10000	50517
SOR iteráció		667.7055136662153245	2000	11377
Egyszerű gradiens		671.5946520921041838	1000	7171
Konjugalt gradiens		601.3684335210467680	2000	13237
Prek. konj. grad.		328.4665917339290786	4999	43607
Prek. konj. grad. ink. Cholesky		394.1550001077384554	10000	29123

872 mesh1e1.mtx 48 306	8.1992			
LDL^T		0.0000005560889071	–	20
Jacobi iteráció		0.0000013087793348	16	146
Jacobi Seidel		0.0000013087793347	16	137
SOR iteráció		0.0000007368669553	21	174
Egyszerű gradiens		0.0000033482501977	37	354
Konjugalt gradiens		0.0000005950104625	20	257
Prek. konj. grad.		0.0000005670958114	21	239
Prek. konj. grad. ink. Cholesky		0.0000005767211340	7	117
873 mesh1em1.mtx 48 306	34.0191			
LDL^T		0.0000006321051073	–	22
Jacobi iteráció		0.0000028531797430	47	342
Jacobi Seidel		0.0000028531797429	47	360
SOR iteráció		0.0000015349057353	19	219
Egyszerű gradiens		0.0000041483193911	123	474
Konjugalt gradiens		0.0000006537118863	33	299
Prek. konj. grad.		0.0000006263122963	46	505
Prek. konj. grad. ink. Cholesky		0.0000006314217294	12	119
874 mesh1em6.mtx 48 306	9.3026			
LDL^T		0.0000007281815383	–	21
Jacobi iteráció		0.0000021854972061	23	149
Jacobi Seidel		0.0000021854972061	23	201
SOR iteráció		0.0000008374455696	19	189
Egyszerű gradiens		0.0000061150312118	45	416
Konjugalt gradiens		0.0000007927677006	20	134
Prek. konj. grad.		0.0000007702646382	23	200
Prek. konj. grad. ink. Cholesky		0.0000007287566702	8	134

875 mesh2e1.mtx 306 2018	422.03			
LDL^T		0.0000014120026332	–	580
Jacobi iteració		0.0000302863301955	506	1843
Jacobi Seidel		0.0000302863301953	506	2135
SOR iteració		0.0000164871146757	284	1347
Egyszerű gradiens		0.0000057910165843	2332	8555
Konjugalt gradiens		0.0000014583534530	126	830
Prek. konj. grad.		0.0000014538301434	443	2923
Prek. konj. grad. ink. Cholesky		0.0000014070672286	23	365
876 mesh2em5.mtx 306 2018	292.64			
LDL^T		0.0000026909226351	–	589
Jacobi iteració		0.0000317652089619	535	1867
Jacobi Seidel		0.0000317652089617	535	2341
SOR iteració		0.0000170126353928	301	1345
Egyszerű gradiens		0.0000076196803869	2133	8645
Konjugalt gradiens		0.0000027051616833	95	626
Prek. konj. grad.		0.0000026771695503	293	2046
Prek. konj. grad. ink. Cholesky		0.0000026972246199	13	220
877 mesh3e1.mtx 289 1377	9			
LDL^T		0.0000000000000106	–	482
Jacobi iteració		0.0000020043767378	37	384
Jacobi Seidel		0.0000020043767378	37	374
SOR iteració		0.0000003251625001	32	292
Egyszerű gradiens		0.0000068469785998	64	506
Konjugalt gradiens		0.0000003369793715	29	376
Prek. konj. grad.		0.0000002383460516	33	465
Prek. konj. grad. ink. Cholesky		0.0000001394541236	10	224

878 mesh3em5.mtx 289 1377	5			
LDL^T		0.0000000000000052	–	478
Jacobi iteráció		0.0000008985580648	28	337
Jacobi Seidel		0.0000008985580647	28	389
SOR iteráció		0.0000004126344354	25	304
Egyszerű gradiens		0.0000056410816213	42	562
Konjugalt gradiens		0.0000002474584416	20	219
Prek. konj. grad.		0.0000002289997117	22	339
Prek. konj. grad. ink. Cholesky		0.0000000000575632	2	107
1207 t2dal_e.mtx 4257 4257	$3.77 * 10^7$			
LDL^T		50.037568027679377	–	4445058
Jacobi iteráció		50.0375680276793773	1	546
Jacobi Seidel		50.0375680276793773	1	700
SOR iteráció		50.0375678306616791	22	5908
Egyszerű gradiens		15132.1814273899872205	10000	2789349
Konjugalt gradiens		1808.2846481383242008	718	188599
Prek. konj. grad.		50.3252165744738065	9999	7023608
Prek. konj. grad. ink. Cholesky		50.0407634283183995	1	1180
1401 Chem97ZtZ.mtx 2541 7361	462.6457			
LDL^T		0.0000000000000265	–	669995
Jacobi iteráció		0.0000025347492519	48	6130
Jacobi Seidel		0.0000025347492519	48	7266
SOR iteráció		0.0000003088953639	23	3873
Egyszerű gradiens		0.0000013180491984	2896	561473
Konjugalt gradiens		0.0000000340811821	164	31270
Prek. konj. grad.		0.0000010771803105	233	69577
Prek. konj. grad. ink. Cholesky		0.0000000000802223	1	710

2203 Trefethen_20b.mtx 19 147	44.0660			
LDL^T		0.0000000000000014	–	12
Jacobi iteráció		0.0000004521992996	9	109
Jacobi Seidel		0.0000004521992996	9	87
SOR iteráció		0.0000006456278917	17	141
Egyszerű gradiens		0.0000030573377073	197	498
Konjugalt gradiens		0.0000000000011042	19	204
Prek. konj. grad.		0.0000000004203342	21	205
Prek. konj. grad. ink. Cholesky		0.0000000158225486	6	102
2204 Trefethen_20.mtx 20 158	95.8369			
LDL^T		0.0000000000000012	–	11
Jacobi iteráció		0.0000007613692245	14	124
Jacobi Seidel		0.0000007613692245	14	124
SOR iteráció		0.0000001853261975	18	217
Egyszerű gradiens		0.0000061949509804	379	843
Konjugalt gradiens		0.0000000000125510	20	189
Prek. konj. grad.		0.0000000019444911	23	250
Prek. konj. grad. ink. Cholesky		0.0000000573616902	7	123
2206 Trefethen_200b.mtx 199 2873	727.11			
LDL^T		0.0000000000000125	–	254
Jacobi iteráció		0.0000005034793111	9	140
Jacobi Seidel		0.0000005034793111	9	132
SOR iteráció		0.0000004701049934	21	301
Egyszerű gradiens		0.0000030863922617	3407	9871
Konjugalt gradiens		0.0000000066687440	129	695
Prek. konj. grad.		0.0000000019758392	99	850
Prek. konj. grad. ink. Cholesky		0.0000000032314352	7	166

2208 Trefethen_300.mtx 300 4678	$2.58 * 10^3$			
LDL^T		0.0000000000000189	–	559
Jacobi iteráció		0.0000008247915578	15	218
Jacobi Seidel		0.0000008247915578	15	200
SOR iteráció		0.0000002581772172	22	237
Egyszerű gradiens		0.0000002581772172	22	237
Konjugalt gradiens		0.0000000043199790	182	1231
Prek. konj. grad.		0.0000000020245415	139	1337
Prek. konj. grad. ink. Cholesky		0.0000000003607824	9	209