Lineáris egyenletrendszerek megoldása

Bevezetés

Általánosan egy lineáris egyenletrendszer a következőképp írható fel:

$$
\begin{aligned}
a_{11}x_1 + a_{12}x_2 + ... + a_{1N}x_N &= b_1 \\
a_{21}x_1 + a_{22}x_2 + ... + a_{2N}x_N &= b_2 \\
&... \\
a_{M1}x_1 + a_{M2}x_2 + ... + a_{MN}x_N &= b_M
\end{aligned}
$$

Ezt felírhatjuk mátrix formában is:

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

Van N ismeretlenünk $x_j,\ j = 1, 2, ..., N$ és M egyenletünk. Az $a_{ij},\ i = 1, 2, ..., M\ j = 1, 2, ..., N$ együtthatók és a $b_i,\ i = 1, 2, ..., N$ úgynevezett *jobboldali értékek* ismertek. Cél az ismeretlenek meghatározása.

Ha $N = M$, vagyis ugyanannyi egyenletünk van mint ahány ismeretlenünk, akkor jó esélyünk van arra, hogy létezzen egyértelmű megoldás. Ez akkor nem igaz, ha a mátrix *degenerált*, vagyis ha van olyan sora, amely két másik lineáris kombinációjaként előáll (sor-degenerált) illetve ha két oszlopa megegyezik (oszlop-degenerált). Ilyenkor az egyenletrendszert *szinguláris*nak nevezzük.

A gyakorlatban a számábrázolás véges volta miatt még további két probléma adódhat.

Egyrészt, ha az egyenletrendszer nem szinguláris ugyan, de közel szinguláris, akkor a megoldás során a kerekítések miatt szingulárissá válhat. Ebben az esetben az algoritmusban hiba (pl. 0-val való osztás) lép fel. Ez a hiba kimutatható.

A másik probléma különösen olyankor jelentkezhet, ha N túl nagy és az egyenletek majdnem szingulárisak, de nem annyira mint az imént említett esetben. Ilyenkor a számábrázolás miatt halmozódó hibák olyan eredményt adnak, amik hibásak. Az algoritmus ilyenkor nem jelez hibát, a fenti probléma csak az ismeretlenek visszahelyettesítésével ismerhető fel.

A kifinomult lineáris egyenletrendszer-megoldó programcsomagok többnyire ezt a két problémát próbálják detektálni illetve kiküszöbölni, és erre már néhány (pl. 10) egyenletnél is szükség lehet. Általában ha az egyenletrendszer nincs közel a szinguláris esethez, akkor néhányszor 10 egyenlet minden különösebb gond nélkül megoldható, néhány 100 egyenlet duplapontosságú aritmetikával. Persze a konkrét problémától nagyon függ a megoldhatóság. Néhány 1000 egyenlet esetén a fenti problémák fokozottan jelentkeznek és ezenkívül a gép *sebességkorlát*ai is kezdenek jelentkezni, általánosan a lineáris egyenletrendszer megoldásához szükséges idő N^3-al skálázik, a táolókapacitás pedig N^2-el. Ennél nagyob egyenletrendszerek soros számítógépeken általában csak akkor oldhatóak meg, ha az egyenletrendszernek speciális formája van, pl. tridiagonális (lásd: spline interpoláció), Vandermonde típusú (pl.interpolált polinom együtthatói), az együtthatók túlnyomó többsége 0, stb. Másik megoldás speciális hardware, párhuzamos számítógépek alkalmazása.

A következő problémák fordulnak elő gyakran, és megoldásuk egymáshoz hasonló módszereket igényel.

- Az $\mathbf{A} \cdot \mathbf{x} = \mathbf{b}$ lineáris egyenletrendszer megoldása

- Az $\mathbf{A} \cdot \mathbf{x}_k = \mathbf{b}_k \quad k = 1, 2, \ldots$ egyenletrendszerek megoldása, ahol különböző jobboldali értékek mellett keressük a megoldásokat ugyanazon együttható mátrixnál. Ez általában egyszerűbb, mint az egyenletrendszerek független megoldása. A numerikus kerekítési hibák miatt az sem célszeű, hogy először invertáljuk a mátrixot, és annak segítségével állítjuk elő a megoldásokat.

- Az együtthatómátrix inverzének ($\mathbf{A}^{-1}$; $\mathbf{A} \cdot \mathbf{A}^{-1} = \mathbf{A}^{-1} \cdot \mathbf{A} = \mathbf{1}$; $\mathbf{1}$ az egységmátrix) kiszámolása. Ez a feladat megegyezik az előző pontival, ha pont N db. N ismeretlenes N egyenletből álló egyenletrendszerünk van, ahol a jobboldali értékek az N lehetséges egységvektor (minden elem 0 kivéve egyet ami 1). A megfelelő $\mathbf{x}$-ek az inverz mátrix oszlopai lesznek.

- Az $\mathbf{A}$ mátrix determinánsának ($\det \mathbf{A} = \sum_{k,l,\ldots s} \epsilon_{kl\ldots s} A_{1k} A_{2l} \ldots A_{Ns}$), kiszámítása.

- Ha $M < N$ vagy pedig degeneráltak az egyenletek (kevesebb egyenlet van mint ismeretlen) akkor több megoldása van az egyenletrendszernek (vagy nincs megoldása). Ezek a megoldások előállíthatóak egy adott $\mathbf{x_p}$ és egy $M - N$ dimenziós altér (ez a *mátrix nulltere*) lineáris kombinációjával. Ezt hívjuk az egyenlet megoldásterének. Ennek megtalálása az ún. *szinguláris érték dekompozíció*val (singular value decomposition, SVD) lehetséges.

- Ha $M > N$ akkor az egyenletrendszer *túlhatározott*. Ilyenkor általában nincs megoldás, ilyenkor értelmes lehet annak a megoldásnak megtalálása, amely leginkább kielégíti az egyenleteket. Ha ezt a legkisebb négyzetek módszerével keressük (a jobb és baloldalak közötti különbségnégyzetek összegének minimumát keressük) akkor a következő N ismeretlenes lineáris problémára vezethetjük vissza:

$$(\mathbf{A}^T \cdot \mathbf{A}) \cdot \mathbf{x} = (\mathbf{A}^T \cdot \mathbf{b})$$

  Ahol $\mathbf{A}^T$ a mátrix transzponáltja. Ezt a legkisebb négyzetek *normálegyenlet*ének nevezik. Általában azonban nem ezek direkt megoldása a legoptimálisabb, hanem célszerű itt is az SVD-t használni (lásd ott).

Ezek a numerikus lineáris algebra tipikus feladatai. Ez egy nagyon nagy terület, rengeteg kutató dolgozik rajta, sok speciális módszer ismert, sok kész programcsomag van, itt csak az alapelveket és a típusokat nézzük végig. Az ismertebb programcsomagok: LINPACK, LAPACK (Argonne National Laboratories, ingyenes), IMSL (fortran ?), NAG. Ezek optimalizálva vannak nagy egyenletrendszerre a futási idő és tárolókapacitás minimalizálásával, a probléma mátrixának (tridiagonális, Vandermonde, stb.) megfelelően különböző verziókban.

A Gauss-Jordan elimináció

A Gauss-Jordan elimináció az egyik legrégebbi, viszonylag könnyen megérthető módszer. Abban az esetben, ha nem csak az egyenletrendszer megoldása a kérdés, hanem az együttható mátrix inverze is, akkor hatékonysága se marad el sokkal a többi általános módszertől. Azonban ha a mátrix inverzre nincs szükség, akkor háromszor lassabb mint a legjobb alternatív eljárás (pl. LU-dekompozíció). A Gauss-Jordan elimináció algoritmusával tehát egyszerre oldjuk meg az alábbi problémákat:

$$\mathbf{A} \cdot \mathbf{x}_k = \mathbf{b}_k$$

$$\mathbf{A} \cdot \mathbf{Y} = 1$$

A következő átrendezések nem változtatják meg az egyenletrendszer megoldását:

- $\mathbf{A}$ két *sor*ának megcserélése, feltéve ha $\mathbf{b}$ illetve 1 megfelelő sorait is megcseréljük. Ez csupán azt jelenti, hogy más sorrendben írtuk fel ugyanazt a lineáris egyenletrendszert.
- Hasonlóan ha $\mathbf{A}$ egy sorát egy nem nulla számmal megszorozzuk, és ahhoz teszőleges más sorok tetszőleges

számmal megszorozva hozzáadunk (*lineáris kombináció*). Természetesen a jobboldalon is ugyanígy kell eljárni.

- $\mathbf{A}$ két *oszlop*át megcseréljük és ezzel együtt $\mathbf{x}$ illetve $\mathbf{Y}$ megfelelő sorait (!) is. Az algoritmus során nyilván kell tartani a cseréket, hogy tudjuk melyik helyen melyik ismeretlen szerepel.

A Gauss-Jordan módszer a fenti átrendezéseket használja úgy, hogy közben $\mathbf{A}$ egységmátrix alakot vegyen fel. Ekkor az átrendezések következtében a jobb oldalon nyilván $\mathbf{x}$ illetve $\mathbf{A}^{-1}$, vagyis a kívánt eredmény adódik.

A legegyszerűbb eljárás csak a fenti második pontban szereplő lépések sgítségével éri ezt el. Először az első sort leosztjuk a_{11}-el, aztán minden további i-ik sorból kivonjuk az így kapott sor a_{i1} szeresét. Természetesen ez a jobboldalra is vonatkozik. Ezzel az első együtthatókat 0-vá tesszük, kivéve az első sorét, ami 1lesz. Így tehát az első oszlop már megegyezik egy egységmátrix első oszlopával. Ezek után hasonlóan járunk el, a második sort leosztjuk a_{22}-vel, és az így kapott sor a_{i2}-szeresét kivonjuk minden $i \neq 2$ sorból. Ezzel a második oszlop is az egységmátrix megfelelő oszlopával azonos lett. Az eljárást folytatva a bal oldal átalakul az egységmátrixszá a jobb oldal pedig a megoldássá.

Nyilvánvalóan probléma lehet, ha az aktuális sorban a diagonális elem 0 és ezzel kellene leosztani. Kevésbé nyilvánvaló, de bebizonyítható, hogy ha nem is fordul ez elő, a módszer numerikusan instabil a kerekítési hibák miatt. A másik két átrendezési módszer használatával ezek a problémák kiküszöbölhetőek. Vagyis - hogy ne rontsuk el a már kész részt - az aktuális sor allatti (illetve oszloptól jobb oldalra lévő) sorok (oszlopok) cserélgetésével elérjük hogy egy olyan elem kerüljön arra a helyre amivel osztanunk kell, ami a lehető legjobban csökkenti a numerikus hibákat. Nincs elméleti válasz arra, hogy melyik a legjobb elem erre a célra, de a legnagyobb elem elég jó választásnak tekinthető. Természetesen ha egy elég nagy számmal megszorzunk egy sort, akkor nyilván abban lesz az aktuális elem. Ezért a kiválasztás előtt (vagy a legelején) célszrű minden sort normálni, pl. olyan számmal beszorozni, mely a sor legnagyobb együtthatóját 1-gyé teszi. A gyakorlat azt mutatja, hogy többnyire elegendő csak a sorokat cserélgetni, és így az algoritmusnak nem kell nyilvántartani az ismeretlenek cserélgetését.

Az algoritmus memória igénye csökkenthető, ha az inverz mátrixot az $\mathbf{A}$ mátrix azon elemeinek helyén tároljuk, amikről úgyis tudjuk, hogy 0 illetve 1, hasonlóan az $\mathbf{x}$ elemeit a $\mathbf{b}$ megfelelő helyeire tehetjük.

Gauss elimináció visszahelyettesítéssel

Harmadára csökkenthetjük a szükséges operációk számát ha a fenti eljárást úgy végezzük, hogy a leosztás után, csak az aktuális sor alatti sorokon hajtjuk végre a fentieket. Így nem egy egységmátrixhoz jutunk, hanem egy háromszögmátrixhoz. (Csak az elemek felét nulláztuk ki, ez felére csökkenti a műveletek számát, és az algoritmus során előrejelezhetően megjelennek 0-k, ez tovább csökkenti a műveletigényt.) Ezután alulról indulva:

$$x_N = b'_N / a'_{NN}$$

$$x_{N-1} = \frac{1}{a'_{(N-1)(N-1)}}[b'_{N-1} - x_N a'_{(N-1)N}]$$

$$\ldots$$

Ezt nevezzük visszahelyettesítésnek.

A visszahelyettesítéses Gauss-eliminációnál optimálisabb az alábbi dekompozíciós eljárás, mert itt a mátrix redukcióját a jobb oldal nélkül tudjuk elvégezni, és az így kapott mátrixxszal több különböző jobboldali vektorra megoldhatjuk a problémát, a műveletigény peddig ugyanennyi.

Alsó-felső háromszög dekompozíció

(LU decomposition)

Tegyük fel, hogy fel tudjuk írni a mátrixot két mátrix szorzataként: $\mathbf{L} \cdot \mathbf{U} = \mathbf{A}$, ahol $\mathbf{L}$ és $\mathbf{U}$ olyan háromszög mátrixok, ameleyknek csak az átló és az az alatti ($\mathbf{L}$) illetve feletti ($\mathbf{U}$) elemek nem 0-k.

Ekkor mivel

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

két lépésben megoldhatjuk a problémát:

$$\mathbf{L} \cdot \mathbf{y} = \mathbf{b}$$

$$\mathbf{U} \cdot \mathbf{x} = \mathbf{y}$$

Ez azért előnyös, mert mindkét mátrix háromszögmátrix, így az előbb bevezetett visszahelyettesítéssel egyszerűen megoldhatóak.

$$y_1 = \frac{b_1}{l_{11}}$$

$$y_i = \frac{1}{l_{ii}} \left[b_i - \sum_{j=1}^{i-1} l_{ij} y_j \right]$$

$$x_N = \frac{y_N}{u_{NN}}$$

$$x_i = \frac{1}{u_{ii}} \left[y_i - \sum_{j=i+1}^{N} u_{ij} x_j \right]$$

Az $\mathbf{A} = \mathbf{L} \cdot \mathbf{U}$ mátrixegyenletet kifejtve a 0 elemek kiírását elhagyva a következőt kapjuk:

$$i < j: \quad l_{i1} u_{1j} + l_{i2} u_{2j} + ... + l_{ii} u_{ij} = a_{ij}$$

$$i = j: \quad l_{i1} u_{1j} + l_{i2} u_{2j} + ... + l_{ii} u_{jj} = a_{ij}$$

$$i > j: \quad l_{i1} u_{1j} + l_{i2} u_{2j} + ... + l_{ij} u_{jj} = a_{ij}$$

Ez N^2 egyenlet $N^2 + N$ ismeretlennel (u_{ij}, l_{ij}), mert a diagonális elemek mindkét dekompozícióban szerepelnek. Így még N feltételre van szükség, hogy egyértelmű legyen a megoldás. Válasszuk ezt úgy, hogy $l_{ii} = 1$ minden i-re. Ez tehát egy újabb lineáris egyenletrendszer, viszont a *Crout algoritmus*sal könnyen megoldható:

$j = 1, 2, ...N$`-re {`
$i = 1, 2, ...j$`-re {`
`az első két egyenlet felhasználásával`
$u_{ij} = a_{ij} - \sum_{k=1}^{i-1} l_{ik} u_{kj}$
`}`
$i = j + 1, j + 2, ...N$`-re {`
`a harmadik egyenlet felhasználásával`
$l_{ij} = (1/u_{jj})(a_{ij} - \sum_{k=1}^{j-1} l_{ik} u_{kj})$
`}`
`}`

Az algoritmus második ciklusában ismét szükség van átrendezés alkalmazására, hogy ne legyen a nevezőben 0 elem. Itt is célszeű a Gauss-Jordan eliminációnál említett eljáráshoz hasonlóan a legnagyobb elemet bemozgatni cserélgetéssel.

A fenti eljárások, melyek háromszög illetve diagonális alakra hozták a mátrixot a determináns számolásra is kiválóan alkalmasak, hiszen a determináns nem lesz más mint a háromszög (diagonális) mátrix átlóban lévő elemeinek szorzata

$$\det \mathbf{A} = \prod_{j=1}^{N} u_{jj}$$

Mikor ezt számoljuk nagy N esetén a túlcsordulás megakadályozására célszerű (pl. 10 megfelelő hatványával) az összeszorzandó értékeket 1 körüli értékre normálni, és a végén ezt figyelembe venni, vagy az abszolútértékek logaritmusának összegét venni, és a végén a exponensre emelni és a helyes előjellel ellátni.

Az mátrix invertálásra is optimálisan használható, hiszen a dekompozíciót csak egyszer kell megcsinálni, és utána az inverz mátrix oszlopait megkaphajuk sorban úgy, hogy olyan jobboldalakra oldjuk meg az egyenletrendszert, aminek minden eleme 0 kivéve az aktuális oszlopnak megfelelő indexű, ami pedig 1.

Tridiagonális illetve sávdiagonális mátrixok esetén a fenti dekompozícióhoz hasonló elven működő algoritmusok hatékonyan alkalmazhatóak. Ilyenkor nem célszerű a 0 elemeket eltárolni (lásd Num.Rec.).

A dekompozíciós algoritmus (példaként a C programozásra, és egy algoritmus implementálására):

```
#include <math.h>
#define NRANSI
#include "nrutil.h"
#define TINY 1.0e-20;

void ludcmp(float **a, int n, int *indx, float *d)
/*
Az a[1...n][1...n] mátrix LU dekompozíciója. Átrendezi a sorokat is úgy,
hogy az átlóba a relatíve legnagyobb elem kerüljön. indx[1...n] tartalmazza az
átrendezés permutációját, d pedig attól függően ±1, hogy páros vagy páratlan
permutáció történt (a determináns számolásnál kellhet). A dekomponált mátrix az
eredeti helyére kerül.
*/
{
        int i,imax,j,k;
        float big,dum,sum,temp;
        float *vv;                        /* A skálázási faktorok soronként */

        vv=vector(1,n);
        *d=1.0;
        for (i=1;i<=n;i++) {              /* A skálázási faktor beállítása */
                big=0.0;
                for (j=1;j<=n;j++)
                        if ((temp=fabs(a[i][j])) > big) big=temp;
                if (big == 0.0) nrerror("Singular matrix in routine
ludcmp");
                vv[i]=1.0/big;
        }
        for (j=1;j<=n;j++) {
                for (i=1;i<j;i++) {                /* u_ij kiszámolása */
                        sum=a[i][j];
                        for (k=1;k<i;k++) sum -= a[i][k]*a[k][j];
                        a[i][j]=sum;
                }
                big=0.0;
```

```
        for (i=j;i<=n;i++) {                /* l_ij kiszámolása */
                sum=a[i][j];
                for (k=1;k<j;k++) sum -= a[i][k]*a[k][j];
                a[i][j]=sum;
                /* Keresi az adott oszlop legnagyobb átskálázott
                  elemét */
                if ( (dum=vv[i]*fabs(sum)) >= big) {
                        big=dum;
                        /* A legnagyobb elem indexe */
                        imax=i;
                }
        }
        /* Ha nem átlóban van a legnagyobb elem, kicseréli */
        if (j != imax) {
                for (k=1;k<=n;k++) {
                        dum=a[imax][k];
                        a[imax][k]=a[j][k];
                        a[j][k]=dum;
                }
                /* A permutációs indexet is beállítja */
                *d = -(*d);
                vv[imax]=vv[j];
        }
        indx[j]=imax;
        /* Néha szinguláris mátrix megoldása is kellhet,
          ilyenkor a 0-val való osztást kerülni kell */
        if (a[j][j] == 0.0) a[j][j]=TINY;
        if (j != n) {
        /* leosztunk az átlós elemekkel az alsó háromszög
          visszahelyettesítéshez */
                dum=1.0/(a[j][j]);
                for (i=j+1;i<=n;i++) a[i][j] *= dum;
        }
    }
    free_vector(vv,1,n);
}
#undef TINY
#undef NRANSI
```

A visszahelyettesítő megoldó program:

```
void lubksb(float **a, int n, int *indx, float b[])
/*
A visszahelyettesítő program. a és indx ludcmp lefuttatásából kapható. Opti-
malizálva van arra, hogy b sok 0 elemmel kezdődhet, és ilyenkor kevesebbet kell
számolni. A megoldás b-be kerül.
*/
{
        int i,ii=0,ip,j;
        float sum;

/* Visszahelyettesítés az alsó háromszöggel */
        for (i=1;i<=n;i++) {
                ip=indx[i];
                sum=b[ip];
                b[ip]=b[i];
                if (ii) /* Addig 0, amíg b elemei 0-k */
                        for (j=ii;j<=i-1;j++) sum -= a[i][j]*b[j];
                else if (sum) ii=i;
                b[i]=sum;
        }
/* Visszahelyettesítés a felső háromszöggel */
        for (i=n;i>=1;i--) {
                sum=b[i];
                for (j=i+1;j<=n;j++) sum -= a[i][j]*b[j];
                b[i]=sum/a[i][i];
        }
}
```

Példa alkalmazásra:

```
/* Driver for routine lubksb */

#include <stdio.h>
#include <stdlib.h>
#define NRANSI
#include "nr.h"
#include "nrutil.h"

#define NP 20
#define MAXSTR 80

int main(void)
{
        int j,k,l,m,n,*indx;
        float p,*x,**a,**b,**c;
        char dummy[MAXSTR];
        FILE *fp;

        indx=ivector(1,NP);
        x=vector(1,NP);
        a=matrix(1,NP,1,NP);
        b=matrix(1,NP,1,NP);
        c=matrix(1,NP,1,NP);
        if ((fp = fopen("matrx1.dat","r")) == NULL)
                nrerror("Data file matrx1.dat not found\n");
        while (!feof(fp)) {
                fgets(dummy,MAXSTR,fp);
                fgets(dummy,MAXSTR,fp);
                fscanf(fp,"%d %d ",&n,&m);
                fgets(dummy,MAXSTR,fp);
                for (k=1;k<=n;k++)
                        for (l=1;l<=n;l++) fscanf(fp,"%f ",&a[k][l]);
                fgets(dummy,MAXSTR,fp);
                for (l=1;l<=m;l++)
                        for (k=1;k<=n;k++) fscanf(fp,"%f ",&b[k][l]);
                /* Save matrix a for later testing */
```

```
		for (l=1;l<=n;l++)
			for (k=1;k<=n;k++) c[k][l]=a[k][l];
		/* Do LU decomposition */
		ludcmp(c,n,indx,&p);
		/* Solve equations for each right-hand vector */
		for (k=1;k<=m;k++) {
			for (l=1;l<=n;l++) x[l]=b[l][k];
			lubksb(c,n,indx,x);
			/* Test results with original matrix */
			printf("right-hand side vector:\n");
			for (l=1;l<=n;l++)
				printf("%12.6f",b[l][k]);
			printf("\n%s%s\n","result of matrix applied",
				" to sol'n vector");
			for (l=1;l<=n;l++) {
				b[l][k]=0.0;
				for (j=1;j<=n;j++)
					b[l][k] += (a[l][j]*x[j]);
			}
			for (l=1;l<=n;l++)
				printf("%12.6f",b[l][k]);
			printf("\n*******************************\n");
		}
		printf("press RETURN for next problem:\n");
		(void) getchar();
	}
	fclose(fp);
	free_matrix(c,1,NP,1,NP);
	free_matrix(b,1,NP,1,NP);
	free_matrix(a,1,NP,1,NP);
	free_vector(x,1,NP);
	free_ivector(indx,1,NP);
	return 0;
}
#undef NRANSI
```

A bemeneti file:

```
MATRICES FOR INPUT TO TEST ROUTINES
Size of matrix (NxN), Number of solutions:
3 2
Matrix A:
1.0 0.0 0.0
0.0 2.0 0.0
0.0 0.0 3.0
Solution vectors:
1.0 0.0 0.0
1.0 1.0 1.0
NEXT PROBLEM
Size of matrix (NxN), Number of solutions:
3 2
Matrix A:
1.0 2.0 3.0
2.0 2.0 3.0
3.0 3.0 3.0
Solution vectors:
1.0 1.0 1.0
1.0 2.0 3.0
NEXT PROBLEM:
Size of matrix (NxN), Number of solutions:
5 2
Matrix A:
1.0 2.0 3.0 4.0 5.0
2.0 3.0 4.0 5.0 1.0
3.0 4.0 5.0 1.0 2.0
4.0 5.0 1.0 2.0 3.0
5.0 1.0 2.0 3.0 4.0
Solution vectors:
1.0 1.0 1.0 1.0 1.0
1.0 2.0 3.0 4.0 5.0
NEXT PROBLEM:
Size of matrix (NxN), Number of solutions:
5 2
Matrix A:
```

```
1.4 2.1 2.1 7.4 9.6
1.6 1.5 1.1 0.7 5.0
3.8 8.0 9.6 5.4 8.8
4.6 8.2 8.4 0.4 8.0
2.6 2.9 0.1 9.6 7.7
Solution vectors:
1.1 1.6 4.7 9.1 0.1
4.0 9.3 8.4 0.4 4.1
```

Az eredmény:

```
right-hand side vector:
    1.000000    0.000000    0.000000
result of matrix applied to sol'n vector
    1.000000    0.000000    0.000000
*******************************
right-hand side vector:
    1.000000    1.000000    1.000000
result of matrix applied to sol'n vector
    1.000000    1.000000    1.000000
*******************************
press RETURN for next problem:
right-hand side vector:
    1.000000    1.000000    1.000000
result of matrix applied to sol'n vector
    1.000000    1.000000    1.000000
*******************************
right-hand side vector:
    1.000000    2.000000    3.000000
result of matrix applied to sol'n vector
    1.000000    2.000000    3.000000
*******************************
press RETURN for next problem:
right-hand side vector:
    1.000000    1.000000    1.000000    1.000000    1.000000
result of matrix applied to sol'n vector
    1.000000    1.000000    1.000000    1.000000    1.000000
*******************************
right-hand side vector:
    1.000000    2.000000    3.000000    4.000000    5.000000
result of matrix applied to sol'n vector
    1.000000    2.000000    3.000000    4.000000    5.000000
*******************************
press RETURN for next problem:
right-hand side vector:
    1.100000    1.600000    4.700000    9.100000    0.100000
result of matrix applied to sol'n vector
```

```
    1.100000    1.600000    4.700003    9.100000    0.100000
*********************************
right-hand side vector:
    4.000000    9.300000    8.400000    0.400000    4.100000
result of matrix applied to sol'n vector
    4.000001    9.300002    8.399989    0.400000    4.100000
*********************************
press RETURN for next problem:
```

Iteratív módszerek

Nagyon nagy illetve közel szinguláris mátrixok direkt megoldása általában nem lehetséges, illetve a számábrázolás hibái miatt hibás eredményt kaphatunk. Ekkor - annak ellenére, hogy matematikailag létezik direkt megoldás - iteratív módon kaphatjuk meg a megoldást, illetve iteratív módon javíthatjuk a megoldást. Itt csak a megoldás iteratív javítását tárgyaljuk, referenciák az [1] megfelelő fejezetében találhatóak.

A megoldás iteratív javítása:

Tegyük fel, hogy a megoldás $\delta\mathbf{x}$-el eltér az igazitól. Ekkor ezt a megoldást visszahelyettesítve a jobboldalon is eltérést tapasztalunk:

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

Kivonva az igazi $\mathbf{A} \cdot \mathbf{x} = \mathbf{b}$ egyenletet, az eltérésre kapjuk a

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

egyenletet. A jobb oldalt ismerjük, így megoldhatunk $\delta\mathbf{x}$-re. Ha dekompozíciós módszert használtunk, akkor egyszerűen be kell helyettesíteni. Több lépésben egyre közelebb kerülhetünk az igazi megoldáshoz, illetve a megoldás jóságának tesztelésére is alkalmas a módszer.

Iterációs módszerek alapelve:

Az $\mathbf{A}\cdot\mathbf{x}=\mathbf{b}$ egyenlet átírható $\mathbf{x}=(\mathbf{I}-\mathbf{A})\cdot\mathbf{x}+\mathbf{b}$ alakúra, ahol $\mathbf{I}$ az egységmátrix. Ezzel képezhető a $\mathbf{x}_{k+1}=(\mathbf{I}-\mathbf{A})\cdot\mathbf{x}_k+\mathbf{b}$ iteráció. Erről belátható, hogy ha $\mathbf{I}-\mathbf{A}$ mátrixnormája 1-nél kisebb, akkor konvergens. Ezen alapul pl. a Gauss-Seidel iterációs eljárás, és több más amelyek a mátrix dekompozícióját használják.

Speciális módszerek

Az alábbi módszerek az [1] könyvben több kevesebb részletességgel tárgyalva vannak.

SVD, singular value decomposition : Szinguláris illetve numerikusan szinguláris egyenletrendszerek esetén a szingularitást adó lineáris kombinációk kidobásával kevesebb egyenletet kapunk mint ahány ismeretlenünk van. Ezt megoldhatjuk a legkisebb négyzetek módszeréhez hasonló elven az alábbi költségfüggvény minimalizálásával:

$$|\mathbf{A} \cdot \mathbf{x} - \mathbf{b}|$$

Az így kapott megoldás numerikusan szinguláris egyenletrendszer esetén közelebb lehet az igazihoz, mint a direkt módszerrel kapható.

Ritka (sparse) mátrixok: Ha az együtthatók többsége 0 (pl. sávmátrixok) akkor általában ha az általános módszert használjuk, akkor nagyon sok 0-t kell tárolni, illetve sok fölösleges 0-val való szorzást végzünk el. Nagy N esetén ez gyakran meg is akadályozhatja az adott gépen a probléma megoldását. Bizonyos gyakran előforduló ritka mátrixra vannak kidolgozott módszerek, hogy hogyan lehet tömören eltárolni, és a Gauss-Jordan vagy LU dekompzíciós módszert optimálisan alkalmazni gyakran N-el skálázó megoldási idővel az általános N^3 helyett. Egy általános ritka mátrix esetén használhatunk egy olyan minimalizációs eljárást, ahol a

$$|\mathbf{A} \cdot \mathbf{x} - \mathbf{b}|^2$$

költségfüggvényt minimalizáljuk a [1]-ben tárgyalt konjugált gradiens módszer segítségével. Ez azért előnyös, mert itt minden lépésben csak $\mathbf{A}$ és $\mathbf{x}$ szorzatát kell kiszámolni, ami a 0-k figyelésével optimálisan megtehető lényegesen kevesebb szorzással.

Vandermonde mátrix: $a_{ij} = \alpha_i^{j-1}$, $i,j = 1...N$, alakú, vagy ennek transzponáltja. Az ilyen mátrix N értékből egyértelműen meghatározható. Ilyen mátrix fordul elő a polinom interpolációnál, vagy a transzponált típusú mátrix a momentumomokkal adott valószínűségeloszlás meghatározásánál. N^2 rendű oprációval megoldható.

Toeplitz mátrix: $a_{ij} = \alpha_{i-j}$ azaz az átlóval párhuzamos vonalban egyforma elemek vannak összesen $2N-1$ különböző. Pl. dekonvolúciós és egyéb jelfeldolgozási problémáknál fordulnak elő. N^2 rendű oprációval megoldható.