1 Lineáris egyenlet rendszerek megoldása

Egy lineáris egyenlet rendszer:

$$\begin{array}{rcccl} a_{11}x_1 + a_{12}x_2 + & \dots & +a_{1n}x_n & = & b_1 \\ a_{21}x_1 + a_{22}x_2 + & \dots & +a_{2n}x_n & = & b_2 \\ \vdots & \ddots & \vdots & = & \vdots \\ a_{m1}x_1 + a_{m2}x_2 + & \dots & +a_{mn}x_n & = & b_m \end{array} \tag{1}$$

n darab ismeretlen $\{x_1, x_2, \dots, x_n\}$ és m darab egyenlet. Az a_{ij} együtthatók $i = 1, 2, \dots, m$ és $j = 1, 2, \dots, n$ ismert számok és az egyenletek jobb oldalán álló b_i, $i = 1, 2, \dots, m$ számok is azok. Ha $n = m$, akkor pont annyi ismeretlenünk van, mint egyenletünk. Ekkor van rá reményünk, hogy meg tudjuk oldani az egyenlet rendszert. Előfordulhat, hogy az m egyenletből néhány előállítható a többi lineár kombinációjaként, vagy az együtthatók néhány oszlopa állítható elő a többi lineáris kombinációjaként. (Négyzetes mátrixoknál, azaz, ha $n = m$ a két állítás egyenértékű.) Ebben az esetben azt mondjuk, hogy az egyenletrendszer szinguláris, és ekkor nem létezik egyértelmű megoldása. Ha numerikusan próbáljuk megoldani az egyenlet rendszert, akkor legalább két problémával kell szembenéznünk:

- Előfordulhat, hogy bár az egyenletrendszer nem szinguláris, mégis valamely, vagy néhány sor annyira közel van a lineáris függéshez, hogy a kerekítési hiba miatt a programunk szingulárisnak fogja találni az egyenletrendszert. Ebben az esetben a program nem tudja megtalálni a megoldást, de ezt detektálja, és jelezni is tudja.
- A másik hibalehetőség, hogy az összeadódó kerekítési hibák a megoldást használhatatlanná teszik. Ennek a hibának a valószínűsége annál nagyobb, minél közelebb áll az egyenletrendszer a szingulárishoz. Ekkor ugyanis nagyszámú olyan kivonás történik, ahol a két operandus közel van egymáshoz, és ilyenkor a művelet során értékes jegyek vesznek el. Ez a hiba olyan módon detektálható, hogy a kapott eredményt visszahelyettesítjük az eredeti egyenletrendszerbe, és megnézzük, hogy mennyire felel meg annak.

1.1 Gauss–Jordan elimináció

Egy mátrix invertálásához a Gauss–Jordan elimináció legalább olyan jó, mint bármely más módszer. Ha egyenlet rendszert kívánunk megoldani, akkor a Gauss–Jordan elimináció nem csak az ismeretlenek megoldás vektorát adja eredményül, hanem az együttható mátrix inverzét is. A gyengesége a módszernek, i hogy ha több konstans oszlopunk van, akkor ezeket mind egyidőben tárolni kell, és számolni mindegyikkel a megoldás során,

ii hogy ha az inverz mátrixra nincs szükségünk, akkor a Gauss–Jordan eliminációhoz háromszor annyi idő kell, mint az LU dekompozícióhoz.

Az i problémára adhatnánk azt a választ, hogy ha valamilyen új konstans vektorral kívánjuk megoldani az egyenlet rendszert, akkor azt csak egyszerűen meg kell szorozni az együtthatómátrix amúgy is adódó inverzével. Ez valóban megtehető, azonban a kerekítési hibák halmozódása miatt ez a megoldás már messze nem lesz olyan pontos, mint azok amelyeket az elimináció során kaptunk meg.

Örjunk fel egy négy ismeretlenes egyenlet rendszert:

$$
\begin{aligned}
a_{11}x_1 + a_{12}x_2 + a_{13}x_3 + a_{14}x_4 &= b_1 && (2)\\
a_{21}x_1 + a_{22}x_2 + a_{23}x_3 + a_{24}x_4 &= b_2 && (3)\\
a_{31}x_1 + a_{32}x_2 + a_{33}x_3 + a_{34}x_4 &= b_3 && (4)\\
a_{41}x_1 + a_{42}x_2 + a_{43}x_3 + a_{44}x_4 &= b_4 && (5)
\end{aligned}
$$

Az ismeretlenek x_i, $i = 1,2,3,4$. A konstans tagok b_i, $i = 1,2,3,4$. Az együtthatók a_{ij}, $i = 1,2,3,4$, $j = 1,2,3,4$. Ugyanaz a megoldás fogja kielégíteni ezt az egyenlet rendszert, mint azt, amelyiket ebből úgy kapunk, hogy

- az egyik egyenletet megszorozzuk egy konstans értékkel,
- az egyik egyenlethez hozáadunk egy másikat.

Használjuk az egyenletrendszer megoldásához ezt a két műveletet. Osszuk el az első egyenletet a a_{11}-el! Ekkor (2) helyett kapjuk:

$$x_1 + \frac{a_{12}}{a_{11}}x_2 + \frac{a_{13}}{a_{11}}x_3 + \frac{a_{14}}{a_{11}}x_4 = \frac{b_1}{a_{11}} \tag{6}$$

Ennek az egyenletnek a $-a_{21}$ szeresét adjuk hozzá a (3) egyenlethez, a $-a_{31}$ szeresét a (4) egyenlethez, és a $-a_{41}$ szeresét a (5) egyenlethez! Ekkor az új egyenlet rendszerben az első sort kivéve az x_1 együtthatója mindenütt nulla lesz. Ezután végezzük el ugyan ezt a műveletet a második egyenlettel, de már az új együtthatókkal. Ekkor az egyenlet rendszerben a második sor kivételével x_2 együtthatói nullák. Folytatva az eljárást, azaz $i = 1,2,3,4$-re az i-edik sort elosztjuk a_{ii}-vel, és minden $j \neq i$-re, a j-edik egyenlethez hozzáadva az i-edik egyenlet $-a_{ji}$ szeresét, az i-edik sort kivéve x_i együtthatói nullák lesznek. Mire elvégezzük a műveletet az összes sorral, minden egyes egyenletben pontosan egy darab együttható lesz nem nulla, hanem egy. Ekkor az egyenlet rendszerünk így néz ki:

$$\begin{aligned} 1 \cdot x_1 + 0 \cdot x_2 + 0 \cdot x_3 + 0 \cdot x_4 &= b_1^* \\ 0 \cdot x_1 + 1 \cdot x_2 + 0 \cdot x_3 + 0 \cdot x_4 &= b_2^* \\ 0 \cdot x_1 + 0 \cdot x_2 + 1 \cdot x_3 + 0 \cdot x_4 &= b_3^* \\ 0 \cdot x_1 + 0 \cdot x_2 + 0 \cdot x_3 + 1 \cdot x_4 &= b_4^* \end{aligned} \tag{7}$$

$$\tag{8}$$

Ahol b_i^* az i-edik egyenlet jobb oldalán álló konstans tag a műveletsor végén. Látszik, hogy az egyenlet rendszer megoldása:

$$x_i = b_i^* \tag{9}$$

Ha nem csak egy konstans vektorunk van, akkor minden egyes műveletet el kell végezni minden egyes, az éppen adott sorban lévő konstans tagokkal.

1.2 Inverz mátrix meghatározása Gauss–Jordan eliminációval

Ha az egyenlet rendszer megoldásakor használt együttható mátrix $\mathbf{A}$, az ismeretlenek vektora $\vec{x}$ és a konstans tagok vektora $\vec{b}$, akkor az egyenlet rendszer az

$$\mathbf{A}\vec{x} = \vec{b} \tag{10}$$

alakba írható. (Mind $\vec{x}$, mind pedig $\vec{b}$ oszlop vektorok.) Ha nem egy ilyen egyenletünk van, hanem n darab, és $\mathbf{A}$ mindegyikben ugyanaz, akkor ezek:

$$\begin{aligned} \mathbf{A}\vec{x}_1 &= \vec{b}_1 \\ \mathbf{A}\vec{x}_2 &= \vec{b}_2 \\ \vdots \; &\;\vdots \;\; \vdots \\ \mathbf{A}\vec{x}_n &= \vec{b}_n \end{aligned} \tag{11}$$

A rövidebb írásmód kedvéért írjuk fel ezt az egyenlet rendszert

$$\mathbf{A}[\vec{x}_1\vec{x}_2\ldots\vec{x}_n] = [\vec{b}_1\vec{b}_2\ldots\vec{b}_n] \tag{12}$$

alakban. Tulajdonképpen nem teszünk mást, mint ugyan ahhoz az $\mathbf{A}$ együttható mátrixhoz, és különböző $\vec{b}$ konstans vektorokhoz keressük a megoldást. Ha az egymás mellé írt n darab $\vec{x}$ vektor helyett egy $\mathbf{X}$ mátrixot írunk, amelynek x_{ij} eleme éppen a j-edik $\vec{x}$ vektor i-edik komponense, és hasonló módon a $\vec{b}$ vektorok helyett egy $\mathbf{B}$ mátrixot írunk, akkor ismerve a mátrix szorzás szabályait könnyen belátható, hogy

$$\mathbf{A} \cdot \mathbf{X} = \mathbf{B} \tag{13}$$

egyenlőség igaz, ha az egyes $\vec{x}$ vektorok megoldásai a megfelelő (10) egyenletnek. Ha a **B** mátrix speciális módon pont az egységmátrix, akkor az **X** mátrix az **A** mátrix inverze. Nem kell tennünk tehát semmi mást, mint követve a Gauss–Jordan eliminációt az n darab egyenlet rendszert egyszerre megoldjuk, és az eljárás végén az egységmátrix helyén megjelenik az **A** mátrix inverze, $\mathbf{A}^{-1}$. A Gauss–Jordan elimináció során, akár egyenlet rendszer megoldására használjuk, akár inverz mátrix előállítására, előfordulhat, hogy a főátló beli elem, amellyel éppen osztani akarunk nulla. Ez nem jelenti feltétlenül azt, hogy az egyenlet rendszer szinguláris. Példáil az

$$\begin{array}{ccccccc} x_1 & + & 2x_2 & + & x_3 & = & 4 \\ x_1 & + & 2x_2 & + & 2x_3 & = & 5 \\ x_1 & + & 3x_2 & + & 2x_3 & = & 6 \end{array} \tag{14}$$

egyenletrendszer megoldása során, ha a második egyenlethez hozzáadjuk az első sor -1 szeresét, akkor az

$$\begin{array}{ccccccc} x_1 & + & 2x_2 & + & x_3 & = & 4 \\ & & & & x_3 & = & 1 \\ & & x_2 & + & x_3 & = & 6 \end{array} \tag{15}$$

egyenlet rendszert kapjuk. (a_{11}-gyel nem kellett osztanunk az első sort, mert $a_{11} = 1$ volt.) Itt a második sorban x_2 együtthatója nulla, nem lehet vele osztani. Az egyenlet rendszernek mégis van egyértelmű megoldása, azaz nem szinguláris. Mivel ez a probléma csak a főelem kiválasztás nélküli Gauss–Jordan eliminációnál fordulhat elő, és mivel a Gauss–Jordan elimináció főelem kiválasztás nélkül nagyon könnyen nem stabilissá válhat, ezért ennek a problémának a megoldásával nem érdemes foglalkozni.

1.3 Főelem kiválasztás

Az elimináció során főelemnek, (angolul pivotnak) nevezzük azt a főátlóbeli elemet amelyikkel éppen elosztjuk az aktuális sort. Az elimináció akkor lesz leginkább stabilis, ha ez az elem abszolút értékben a lehető legnagyobb. Ha a főelem kiválasztás során vagy csak az aktuális sorból, vagy csak az aktuális oszlopból választjuk ki a legnagyobb abszolút értékűt, akkor részleges főelem kiválasztást, ha pedig az egész mátrixban keressük a legnagyobb elemet, akkor teljes főelem kiválasztást végzünk. Ahhoz, hogy a kiválasztott elem a főátlóba kerüljön sor és oszlopcseréket hajtunk végre a mátrixban, arra azonban ügyelni kell, hogy ezeket a cseréket a program megfelelően adminisztrálja. Ellenkező esetben ugyanis oszlop cserék után előfordulhat, hogy a konstans tagok helyén megkapjuk ugyan az ismeretlenek értékét, csak azt nem fogjuk tudni, hogy melyik érték melyik ismeretlenhez tartozik. Általában mind a részleges, mind pedig a teljes főelem kiválasztásnál el lehet kerülni az oszlopok cseréjét. Mindkétfajta főelem kiválasztásnál csak olyan elemeket vehetünk figyelembe, amelyek olyan oszlopban és sorban állnak, amelyeket az elimináció még nem normált le, vagy nullázott ki. Furcsának tűnhet a részleges főelem kiválasztásnál, hogy a legnagyobb abszolút értékű elemet választjuk ki. Az egyenlet rendszerünk megoldása ugyanis nem változik attól, ha az egyik egyenletünket egy konstans számmal megszorozzuk. Egy megfelelően nagy számot választva szorzónak bármelyik sor eleméből csinálhatunk főelemet. Emiatt néha alkalmazzák az úgynevezett implicit főelem kiválasztást, amelyik nem a legnagyobb elemet választja ki az adott oszlopból, hanem azt, amelyik a legnagyobb lenne akkor, ha minden sort végig osztottunk volna a legnagyobb elemével. A főelem kiválasztásnál megtehetjük, hogy nem cseréljük meg a sorok elemeit, hanem egy `INTEGER` tömbben tartjuk nyilván a cseréket. Ez a módszer nyilván gyorsabb, mint a sorok fizikai megcserélése, de ha az eredeti mátrix helyén elő akarjuk állítani az inverz mátrixot, akkor nem tudjuk a sorok valódi megcserélését kikerülni.

1.4 Gauss elimináció visszahelyettesítéssel

Változtassunk a Gauss–Jordan elimináción úgy, hogy az egyes sorokat csak a nagyobb sorszámú sorokhoz adjuk hozzá, azaz ha éppen az i-edik sort adjuk hozzá a többihez, akkor minden $j > i$-re a j-edik sorhoz hozzáadjuk az i-edik sor $-\frac{a_{ji}}{a_{ii}}$ szeresét. Ekkor az elimináció végére az $\mathbf{A}$ mátrix főátló alatti része lesz kinullázva.

$$\begin{pmatrix} a_{11} & a_{12} & a_{13} & \dots & a_{1n} \\ 0 & a_{22} & a_{23} & \dots & a_{2n} \\ 0 & 0 & a_{33} & \dots & a_{3n} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & 0 & \dots & a_{nn} \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \\ \vdots \\ x_n \end{pmatrix} = \begin{pmatrix} b_1 \\ b_2 \\ b_3 \\ \vdots \\ b_n \end{pmatrix} \tag{16}$$

Az eljárást eddig a pontig hívják Gauss eliminációnak. Ezután jön a visszahelyettesítés. Az egyenlet rendszer nem kapott olyan triviális formát, mint a Gauss Jordan elimináció után, de x_n értékét azért könnyen megkapjuk:

$$x_n = \frac{b_n}{a_{nn}} \tag{17}$$

Ha pedig ismerjük x_n-et, akkor x_{n-1}:

$$x_{n-1} = \frac{b_{n-1} - x_n \cdot a_{n-1,n}}{a_{n-1,n-1}} \tag{18}$$

Általánosságban azt mondhatjuk, hogy ha ismerjük x_j értékét $j = i+1, .., n$-re, akkor

$$x_i = \frac{1}{a_{ii}} \left[b_i - \sum_{j=i+1}^{n} a_{ij} x_j \right] \tag{19}$$

Az az eljárás, amelyik ennek a képletnek az alapján határozza meg az ismeretleneket visszahelyettesítés.

Ennek a módszernek az az előnye a Gauss–Jordan eliminációval szemben, hogy kevesebb művelet elvégzésére van szükség. Míg a Gauss–Jordan elimináció legbelső ciklusa, amely egy szorzást és egy kivonást tartalmaz n^3-szor hajtódik végre, addig itt, mivel nem tisztítjuk ki az egész mátrixot, csak a felét, $\frac{1}{3}n^3$-szor.

Az a hátrány, hogy a konstans vektorokat mind ismernünk kell az elimináció megkezdésekor itt is jelen van.

1.5 A Gauss–Jordan elimináció programja

A programot lépésenként fejlesztjük ki. Nézzük meg az első változatot!

```
procedure gaussj1( var a:glnpbynp; n:integer;
                  var b:glnpbymp; m:integer );
(*
TYPE glnpbynp=ARRAY[1..np,1..np] of REAL;
     glnpbymp=ARRAY[1..np,1..mp] of REAL;
                                         *)
VAR i,ll,l:integer;
    inv,dum:real;
BEGIN
  for i:=1 to n do
   BEGIN
    inv := 1.0 / a[i,i];
    for l:=1 to n do
     a[i,l] := a[i,l] * inv;
    for l:=1 to m do
     b[i,l] := b[i,l] * inv;

    for ll:=1 to n do
     if ll<>i then
      BEGIN
       dum := a[ll,i];
       for l:=1 to n do
        a[ll,l] := a[ll,l] - a[i,l] * dum;
```

```
        for l:=1 to m do
         b[ll,l]:= b[ll,l] - b[i,l] * dum ;
       END
    END
END;
```

Ez a program nem végez főelem kiválasztást, és nem is állítja elő az `a` mátrix inverzét, csak kinullázza a főátló alatti és fölötti elemeket. A legkülső ciklus `i` végigmegy minden soron. A cikluson belül először elosztja az éppen aktuális sor minden elemét (a konstans tagokat is) a főátló beli elemmel. Ezzel tulajdonképpen ezt a sort normáltuk, és a főátlóba `1.0` került. A következő ciklus `ll` végigmegy minden soron, kivéve az `i`-edik sort, és minden sorhoz hozzáadja az `i`-edik sor annyiszorosát, hogy az `i`-edik oszlopba nulla kerüljön. Ez a ciklus kinullázza az éppen aktuális sor főátlójában lévő elem alatti és feletti elemeket. Az `i`-edik sorból természetesen nem szabad kivonni önnmagát, hiszen ezzel információt veszítenénk, kinulláznánk az egész sort, a konstans tagokat is, és nem tudnánk megoldani az egyenlet rendszert.

Az eljárás lefutása után az `a` mátrix helyén megkapjuk az egység mátrixot, a `b` elemeiben pedig az ismeretlenek értékeit.

Hogyan állítsuk elő az **A** mátrix inverzét?

Ha a **B** mátrix mérete $n \times n$, és eredeti tartalma $\mathtt{b} = \underline{\underline{\mathbf{1}}}$, azaz `b` egységmátrix, akkor a `gaussj1` eljárás lefutása után `b`-ben megkapjuk `a` inverzét. Ehhez persze fel kell töltenünk `b` megfelelő elemeit nullákkal, és egyesekkel. Ehelyett most próbáljuk meg úgy megírni az eljárásunkat, hogy ne az elimináció megkezdése előtt, hanem az elimináció során töltse fel a `b` mátrixot. Gondoljuk végig, hogy az elimináció során hogyan alakul ki a `b`-ben az inverz mátrix. Az `i` ciklus első lefutásakor $\mathtt{b}_{11} = \mathtt{a}_{11}^{-1}$ lesz, a sor többi eleme pedig marad nulla, hiszen nullát bármivel osztva nullát kapunk. Az `ll` ciklusban `b` minden sorához, kivéve az elsőt, hozzáadjuk az első `dum`-szorosát. Mivel azonban az első sorban az első elemet kivéve minden elem nulla, ezért a `b` mátrixnak csak az első oszlopa fog megváltozni. Az `i` ciklus második lefutásakor a `b` mátrix második sorában, amelyet elosztottunk $\mathtt{a}_{11}$-gyel csak az első és második, a ciklus `i`-edik lefutásakor pedig az 1,2,...,i-edik elemek különböznek nullától, így a sor hátralevő része marad a végigosztás után is nulla. Hasonlóan az `ll` ciklusban a `b` mátrixnak csupán az 1,2,...,i-edik oszlopában lévő elemei változnak meg. Úgy fogalmazhatunk tehát, hogy a legkülső ciklus lefutása után a `b` mátrixban csak az 1,2,...,i oszlopokban lesznek az eredetitől eltérő elemek. Elegendő tehát, ha menet közben mindíg csak az `i`-edik oszlopot nullázzuk ki és írunk a főátlóba egyet, majd végrehajtjuk az elimináció `i`-edik ciklusát. Álljunk most meg az elimináció `i`-edik lefutása után, és képzeljük el mind az eredetileg az egységmátrixot tartalmazó `b` és az eredetileg az egyenletrendszer együtthatóit tartalmazó `a` mátrixot. A `b` mátrix minden `i`-nél nagyobb sorszámú oszlopában még őrzi az egység mátrix megfelelő részét, az `a` mátrixban pedig az `i`-nél nem nagyobb sorszámú oszlopokban alakultak ki az egységmátrix megfelelő oszlopai. Felmerül a kérdés: Minek tároljuk a nullákat és egyeseket? Helyezzük a két mátrixot egymás fölé, és tároljuk egyetlen mátrixban az elimináció során lassan elfogyó `a` és az ezzel együtt felépülő `b` mátrixot!

```
procedure gaussj2( var a:glnpbynp; n:integer );
(*
TYPE glnpbynp=ARRAY[1..np,1..np] of REAL;
                                         *)
VAR i,ll,l:integer;
    inv,dum:real;
BEGIN
  for i:=1 to n do
   BEGIN
    inv := 1.0 / a[i,i];
    a[i,i] := 1.0;
    for l:=1 to n do
     a[i,l] := a[i,l] * inv;

    for ll:=1 to n do
     if ll<>i then
      BEGIN
       dum := a[ll,i];
       a[ll,i] := 0.0;
       for l:=1 to n do
        a[ll,l] := a[ll,l] - a[i,l] * dum;
      END
   END
END;
```

Az eljárás új változatában nem kell tehát törődnünk a `b` mátrixszal külön, hiszen a megfelelő részeit mind az `a`, mind pedig a `b` mátrixnak az `a` tömbben tároljuk. A legkülső ciklus `i`-edik lefutásakor, miután a főátlóbeli elemet kivettük a mátrixból, és a reciprokát az `inv` változóban helyeztük, a főátlóba egyet írunk, így a sor végig osztása után ide pont a főátló beli elem reciproka kerül, vagyis az, ami az inverz mátrix kialakítása során a $\mathtt{b}_i$ pillanatnyi értéke kell, hogy legyen. Az `ll` ciklusban, amikor `a[ll,i]`-t kinulláztuk, éppen az inverz mátrix `[ll,i]` pozícióban levő értékét állította elő az `a` tömb `[ll,i]` elemében a program. Amikor $\mathtt{l} \leq \mathtt{i}$, akkor mindkét `l` ciklus az inverz mátrixot állítja elő, amikor pedig $\mathtt{i} < \mathtt{l}$, akkor az eliminációt végzi. Természetesen miközben előállítjuk az a tömbben az inverz mátrixot a `b` tömbben az egyenlet rendszer megoldásait is létrehozhatjuk.

```
procedure gaussj3( var a:glnpbynp;
                       n:integer;
                   var b:glnpbymp;
                       m:integer );
(*
TYPE glnpbynp=ARRAY[1..np,1..np] of REAL;
     glnpbymp=ARRAY[1..np,1..mp] of REAL;
                                           *)
VAR i,ll,l:integer;
    inv,dum:real;
BEGIN
  for i:=1 to n do
   BEGIN
    inv := 1.0 / a[i,i];
    a[i,i] := 1.0;
    for l:=1 to n do
     a[i,l] := a[i,l] * inv;
    for l:=1 to m do
     b[i,l] := b[i,l] * inv;

    for ll:=1 to n do
     if ll<>i then
      BEGIN
       dum := a[ll,i];
       a[ll,i] := 0.0;
       for l:=1 to n do
        a[ll,l] := a[ll,l] - a[i,l] * dum;
       for l:=1 to m do
        b[ll,l]:= b[ll,l] - b[i,l] * dum ;
      END
   END
END;
```

Meg is lennénk az eliminációval, ha ezek a programok nem lennének veszélyesek. A veszély abban rejlik, hogy nem végeztünk főelem kiválasztást, és ezért a halmozódó hibák miatt nagyon könnyen a valódi megoldástól távoli eső számokat kaphatunk. Mielőtt azonban nekilátnánk a főelem kiválasztásnak próbáljuk meg végig gondolni, hogy vajon működnének-e a programjaink, ha a legkülső ciklusban `i` értéke nem egytől menne egyesével n-ig, hanem az $1, 2, \ldots, n$ számokat valamely más permutáció szerint venné sorra. Természetesen a programok ekkor is ugyanazt az eredményt adják. Meg kell még jegyezni azt is, hogy egyik program sem vizsgálja, hogy a főátlóbeli elem, amivel a sort végig osztjuk nem nulla-e. Ha ez az elem nulla, az még nem jelenti azt, hogy a mátrix szinguláris. (Ahogy azt korábban már egy példán keresztül bemutattuk.)

A főelem kiválasztást alkalmazó programokat is lépésenként fejlesztjük ki.

Készítsük el először azt a programot, amelyik részleges főelem kiválasztást végez, és nem állítja elő az inverz mátrixot!

A főelemet az éppen aktuális elem oszlopában keressük. Azért az oszlopban, és nem a sorban, mert mert így ahhoz, hogy a kiválasztott elem arra a főátló beli helyre kerüljön ahol éppen tartunk, sorokat és nem oszlopokat kell megcserélni. Ha oszlopokat cserélnénk meg, akkor nyilván kellene tartani a cseréket, egy egyenlet rendszer megoldását azonban az egyenletek sorrendje nem befolyásolja. A sorcseréket természetesen a **B** mátrixban is végre kell hajtani, hiszen a konstans tagok is hozzá tartoznak az egyenletekhez. Ha az oszlopban a legnagyobb abszolút értékű elem nulla, akkor minden elem nulla, és ez azt jelenti, hogy az együttható mátrix szinguláris, az egyenlet rendszernek nincs egyértelmű megoldása. Az egyes sorok cseréje után a program hátralevő része ugyanaz, mint a főelem kiválasztás nélküli esetnél.

```
procedure gaussj4( var a:glnpbynp; n:integer;
                   var b:glnpbymp; m:integer );
(*
TYPE glnpbynp=ARRAY[1..np,1..np] of REAL;
     glnpbymp=ARRAY[1..np,1..mp] of REAL;
                                          *)
VAR i,ll,l,row:integer;
    inv,dum,big:real;
BEGIN
  for i:=1 to n do
   BEGIN
    row := 0;
    big := 0.0;
    for l:=1 to n do
     if big <= abs(a[l,i]) then
      BEGIN
       row := l;
       big := abs(a[l,i]);
      END;
    if big = 0.0 then
     BEGIN
      writeln('The matrix is singular.');
      halt;
     END;
    if row <> i then
```

```
     BEGIN
      for l:=1 to n do
       BEGIN
        dum := a[i,l];
        a[i,l] := a[row,l];
        a[row,l] := dum;
       END;
      for l:=1 to m do
       BEGIN
        dum := b[i,l];
        b[i,l] := b[row,l];
        b[row,l] := dum;
       END;
     END;
    inv := 1.0 / a[i,i];
    for l:=1 to n do
     a[i,l] := a[i,l] * inv;
    for l:=1 to m do
     b[i,l] := b[i,l] * inv;
    for ll:=1 to n do
     if ll<>i then
      BEGIN
       dum := a[ll,i];
       for l:=1 to n do
        a[ll,l] := a[ll,l] - a[i,l] * dum;
       for l:=1 to m do
        b[ll,l]:= b[ll,l] - b[i,l] * dum ;
      END;
   END;
END;
```

Mi a helyzet akkor, ha az inverz mátrixot is elő akarjuk állítani? Amikor az

$$\mathbf{A} \cdot \mathbf{X} = \mathbf{I} \tag{20}$$

egyenletet oldjuk meg, akkor minden egyes sorművelet amit az $\mathbf{A}$ és az $\mathbf{I}$ mátrixon egyidőben végzünk megfelel valamilyen $\mathbf{P}$ mátrixszal való szorzásnak. Az egész elimináció egyenlete:

$$\mathbf{P}_k \mathbf{P}_{k-1} \mathbf{P}_{k-2} \ldots \mathbf{P}_2 \mathbf{P}_1 \mathbf{A} \cdot \mathbf{X} = \mathbf{P}_k \mathbf{P}_{k-1} \mathbf{P}_{k-2} \ldots \mathbf{P}_2 \mathbf{P}_1 \mathbf{I} \tag{21}$$

ahol $\mathbf{P}_i$ az i-edik sorműveletnek megfelelő mátrix. A műveletek során a mátrix szorzásokat jobbról balra haladva végezzük el, de mivel a mátrix szorzás asszociatív semmi sem tiltja, hogy gondolatban most balról jobbra végezzük el, és a

$$\mathbf{P} = \mathbf{P}_k \mathbf{P}_{k-1} \dots \mathbf{P}_1 \tag{22}$$

jelölést használva:

$$\mathbf{P} \cdot \mathbf{A} \cdot \mathbf{X} = \mathbf{P} \cdot \mathbf{I}. \tag{23}$$

A főelem kiválasztás sorcserei ugyan olyan mátrix műveletek, mint az elimináció többi sorművelete, így igaznak kell lennie, hogy minden további sorművelet nélkül

$$\mathbf{X} = \mathbf{P} = \mathbf{A}^{-1} \tag{24}$$

Van azonban egy kis probléma. Nevezetesen az, hogy mi nem állítjuk elő az **I** mátrixot a kezdet kezdetén, hanem csak menet közben az elimináció során. Ha pedig most bővítve a `gaussj4` programunkat a `040` sor utánra beszúrjuk az

```
a[i,i]:=1.0;
```

sort, s a `048` sor utánra pedig az

```
a[ll,i]:=0.0;
```

sort, akkor az egység mátrix `i`-edik oszlopának az előállítása már a sorok cseréje után történt meg, és így az ebben az oszlopban álló elemet nem cseréltük meg, és mivel minden `j` $>$ `i` `j`-edik oszlopot is később állítunk elő, a két sor minden `j` $\geq$ `i` `j`-edik elemét nem cseréltük meg. Szerencsére ezeknek az elemeknek a zöme nulla, csupán az `i`-edik sor `i`-edik eleme, és ha az a sor amelyikkel az `i`-edikel megcseréltük a `j`-edik, akkor ennek a `j`-edik eleme nem nulla, hanem 1. Így, hogyha az `i`-edik, és `j`-edik sort megcseréljük, majd pedig az `i`-edik és `j`-edik oszlopot, akkor pont azt a mátrixot kapjuk, amelyikben az `i`-edik és `j`-edik sor első `i-1` eleme lett megcserélve. Ezzel tehát, hogy az 1-est a főátlóba, és a nullákat az éppen aktuális oszlop helyére a sor csere után írtuk be tulajdonképpen egy oszlop cserét hajtottunk végre az egység mátrixból az inverz mátrixszá átalakuló mátrix azon részén amelyik még érintetlenül tartalmazza az egységmátrix elemeit. Mivel azonban ezt az oszlopcserét nem hajtottuk végre az egyenlet bal oldalán, ezért még egyszer végre kell hajtanunk, hogy a hatását megszüntessük. Az egyes oszlopcserék a mátrix egyenletbe az egységmátrix jobbról való **Q** mátrixszokkal való szorzásával írhatók be.

$$\mathbf{PAX} = \mathbf{PI} \cdot \mathbf{Q}_1 \mathbf{Q}_1^{-1} \mathbf{Q}_2 \mathbf{Q}_2^{-1} \dots \mathbf{Q}_k \mathbf{Q}_k^{-1} \tag{25}$$

1.6 LU dekompozíció

Tegyük fel, hogy az **A** mátrixot sikerült

$$\mathbf{LU} = \mathbf{A} \tag{26}$$

alakban felírni, ahol L alsó háromszög mátrix, azaz csak a főátlóban és az alatt tartalmaz nem nulla elemeket, U pedig felső háromszög mátrix, azaz csak a főátlóban és a felett tartalmaz nem nulla elemeket. Egy $\mathbf{A}$ 4×4-es mátrixra ez így néz ki:

$$\begin{bmatrix} \alpha_{11} & 0 & 0 & 0 \\ \alpha_{21} & \alpha_{22} & 0 & 0 \\ \alpha_{31} & \alpha_{32} & \alpha_{33} & 0 \\ \alpha_{41} & \alpha_{42} & \alpha_{43} & \alpha_{44} \end{bmatrix} \cdot \begin{bmatrix} \beta_{11} & \beta_{12} & \beta_{13} & \beta_{14} \\ 0 & \beta_{22} & \beta_{23} & \beta_{24} \\ 0 & 0 & \beta_{33} & \beta_{34} \\ 0 & 0 & 0 & \beta_{44} \end{bmatrix} = \begin{bmatrix} a_{11} & a_{12} & a_{13} & a_{14} \\ a_{21} & a_{22} & a_{23} & a_{24} \\ a_{31} & a_{32} & a_{33} & a_{34} \\ a_{41} & a_{42} & a_{43} & a_{44} \end{bmatrix} \tag{27}$$

Ezt a felbontást felhasználhatjuk egy egyenletrendszer megoldására:

$$\mathbf{A}\vec{x} = (\mathbf{L} \cdot \mathbf{U})\vec{x} = \mathbf{L}(\mathbf{U}\vec{x}) = \vec{b} \tag{28}$$

úgy, hogy először megoldjuk az

$$\mathbf{L}\vec{y} = \vec{b} \tag{29}$$

egyenlet rendszert, majd pedig az

$$\mathbf{U}\vec{x} = \vec{y} \tag{30}$$

egyenletrendszert. Mi az értelme annak, hogy egy egyenletrendszer helyett kettőt oldunk meg? Az, hogy mivel mind az L, mind pedig az U háromszög mátrixok, ezért (29) és (30) megoldása triviális. A (29) egyenlet megoldható előre haladó behelyettesítéssel:

$$y_1 = \frac{b_1}{\alpha_{11}} \tag{31}$$

és $i = 2, 3, \ldots, n$-re

$$y_i = \frac{1}{\alpha_{ii}} \left[b_i - \sum_{j=1}^{i-1} \alpha_{ij} y_j \right] \tag{32}$$

A (30) pedig visszafelé haladó behelyettesítéssel:

$$x_n = \frac{y_n}{\beta_{nn}} \tag{33}$$

és $i = n-1, n-2, \ldots, 2, 1$-re

$$x_i = \frac{1}{\beta_{ii}} \left[y_i - \sum_{j=i+1}^{n} \beta_{ij} x_j \right] \tag{34}$$

Ez a megoldás n^2-tel arányos mennyiségű számítást igényel, és valahányszor egy újabb $\vec{b}$ vektorral kívánjuk megoldani az egyenletet csak a (31)...(34) képleteket kell alkalmazni. Most már csak az a kérdés, hogy egy adott **A** mátrixhoz hogyan keressük meg a megfelelő **L** és **U** mátrixokat. Ha az (26) mátrix egyenletben elvégezzük a beszorzást, akkor n^2 darab skalár egyenletet kapunk. Írjunk fel ebből egyet!

$$\alpha_{i1}\beta_{1j} + \ldots = a_{ij} \tag{35}$$

A három pont itt nem azt jelenti, hogy az összegzést vég nélkül kell folytatni, hanem azt, hogy i és j értékétől függően másképpen kell befejezni. Ha ugyanis $i < j$, akkor:

$$\alpha_{i1}\beta_{1j} + \alpha_{i2}\beta_{2j} + \ldots + \alpha_{ii}\beta_{ij} = a_{ij} \tag{36}$$

ugyanis minden $\beta_{kj} = 0$ ha $k > j$. Ha $i = j$:

$$\alpha_{i1}\beta_{1j} + \alpha_{i2}\beta_{2j} + \ldots + \alpha_{ii}\beta_{jj} = a_{ij} \tag{37}$$

és ha $i > j$, akkor:

$$\alpha_{i1}\beta_{1j} + \alpha_{i2}\beta_{2j} + \ldots + \alpha_{ij}\beta_{jj} = a_{ij}. \tag{38}$$

Ez pontosan n darab egyenletet jelent, ismeretlenünk pedig $n^2 + n$ darab van. (Az α és β értékek teljesen kitöltik az $n \times n$-es mátrixot, de a főátló kétszer szerepel, innen van a $+n$.) Hogy meglegyen a kellő számú egyenletünk, írjunk fel még n darabot! Legyen $i = 1, 2, \ldots, n$-re $\alpha = 1.0$ Most tehát van n^2+n darab egyenletünk, és ugyanennyi ismeretlenünk. Persze elég furcsának tűnhet, hogy egy n ismeretlenes egyenletrendszer helyett egy n^2+n ismeretlenes, tehát jóval nagyobb egyenletrendszert akarunk megoldani. A magyarázat az, hogy bár ez az egyenletrendszer jóval nagyobb, mégis könnyen megoldható, mert speciális. A megoldás Crout algoritmusa:

- Legyen $\alpha_{11} = 1$ és $\beta_{11} = a_{11}$. (Mivel $i = j = 1$-re $\alpha_{i1}\beta_{1j} = \alpha_{11}\beta_{11} = a_{ij} = a_{11} \Longrightarrow \beta_{11} = a_{11}$)
- Ezek után minden $j = 2, 3, \ldots, n$ értékre:
- Először $i = 1, 2, \ldots, j$-re keressük meg a β_{ij}-t az egyenleteink alapján!

$$\beta_{ij} = a_{ij} - \sum_{k=1}^{i-1} \alpha_{ik}\beta_{kj} \tag{39}$$

  (Ha $i = 1$, akkor definiáljuk a $\sum$ értékét nullának.)

- Másodszor pedig $i = j+1, j+2, \ldots, n$-re keressük meg α_{ij}-t:

$$\beta_{ij} = \frac{1}{\beta_{jj}}\left[a_{ij} - \sum_{k=1}^{i-1}\alpha_{ik}\beta_{kj}\right] \tag{40}$$

(Figyeljük meg, hogy a (39) és (40) képletek csak a β_{jj} való osztásban térnek el egymástól.) Könnyen látható, hogy követve az eljárást a következő α és β értékek kiszámításához a korábbi α és β értékek rendelkezésre állnak. Az is látszik, hogy amikor egy α_{ij}-t vagy β_{ij}-t kiszámítottunk, akkor ahhoz felhasználjuk a_{ij} értékét, de arra a továbbiakban nincsen szükség a dekompozícióhoz, ezért az eredeti mátrixban tárolhatjuk a kiszámított α és β értékeket. (Ez alól kivételek az α_{ii} értékek, ezeket azonban egyáltalán nem tároljuk, tudván, hogy ezek értéke 1.) A főelem kiválasztást a stabilitás érdekében itt is alkalmaznunk kell, hatékonyan azonban csak részleges főelem kiválasztást tudunk megvalósítani. (A részleges főelem kiválasztás a stabilitáshoz legtöbb esetben ugyan olyan jó, mint a teljes főelem kiválasztás.) A részleges főelem kiválasztás alkalmazása itt azt jelenti, hogy nem magát az $\mathbf{A}$ mátrixot bontjuk fel $\mathbf{L}$ és $\mathbf{U}$ mátrixok szorzatára, hanem egy, az $\mathbf{A}$ mátrixból sorcserékkel előállított mátrixét. Természetesen ezeket a sorcseréket a felbontás későbbi felhasználása érdekében nyilván kell tartanunk. Itt sajnos nem tehetjük meg, hogy az eljárás végén helyrehozzuk a cseréket. (Ugyanis, míg egy $\mathbf{A}$ mátrixnak mindig van $\mathbf{A}^{-1}$ inverze, ha nem szinguláris, LU felbontása nem biztos, hogy van.) A főelem kiválasztás megoldja azt a problémát is, ami akkor áll elő, ha az eredeti $\mathbf{A}$ mátrixnak nincs LU felbontása, bár nem szinguláris. A részleges főelem kiválasztást úgy végezzük el, hogy az a β_{jj} elem, amivel osztunk az α_{ij}-t előállító képletben, a lehető legnagyobb legyen. Ehhez először kiszámítjuk a β_{ij} értékeket, majd az α_{ij} értékek β_{jj}-szeresét. (Nem osztunk β_{jj}-vel az (40) képletben.) Ezután végezzük el a főelem kiválasztást. Kiválasztjuk a legnagyobb elemet a j-edik oszlopban az $i \geq j$ elemek közül, és az ilyen módon kiválasztott i-edik sort megcseréljük a j-edikkel. Ez a sorcsere a mátrix $l = 1, 2, \ldots, j-1$ oszlopában a már felbontott rész α_{jl} elemeit cseréli meg az α_{il} elemekkel, és ez ugyanaz, mintha az eredeti mátrix elemeit cseréltük volna meg még a felbontás megkezdése előtt. A mátrix $l = j+1, \ldots, n$ oszlopaiban az eredeti mátrix elemei még sértetlenek. A j-edik oszlop megcserélésével szintén ugyanazt a hatást érjük el, mintha még a felbontás megkezdése előtt cseréltük volna meg a két sort. A két sor megcserélése után pedig a j-edik oszlop $j+1, \ldots, n$ elemeit elosztjuk a β_{jj} kiválasztott főelemmel.

1.7 Az LU dekompozíció programja

Az itt közölt program implicit főelem kiválasztást végez. Ezért még mielőtt hozzálátna a dekompozícióhoz, minden sorból kikeresi a legnagyobb abszolút értékű elemet, és az abszolút érték reciprokát elhelyezi a `vv` vektor megfelelő elemében. Ezután lát neki a dekompozíciónak. Végig megy minden oszlopon. Minden oszlopban először kiszámítja a β_{ij} értéket $i = 1, 2, \ldots, j-1$-re, majd az új α_{ij} értékeket $i = j, \ldots, n$-re, de nem végzi el a β_{jj}-vel való osztást, hanem előbb kiválasztja az oszlop alsó részéből a legnagyobb abszolút értékűt (szorozva mindig a sor normájával), és megcseréli a sorokat. `imax` tartalmazza annak a sornak a számát, amelyet meg kell cserélni a `j`-edikkel. Ezután a `vv` vektor elemeit is meg kell cserélni, mivel azonban a `j`-edikre többet nem lesz szükség, ezért a csere műveletének a kétharmada elhagyható. Ezután a program elvégzi a kiválasztott főelemmel való osztást. Az `ludcmp` eljárásnak be, és kimenő paramátere az `a` mátrix. Ennek a méretét határozza meg az `n` egész változó. Az `indx` egészeket tartalmazó vektor kimenő változó, míg `d` értéke az eljárás lefutása után +1, vagy -1 attól függően, hogy páros, vagy páratlan számú sorcserét végeztünk.

```
PROCEDURE ludcmp(VAR a: glnpbynp; n: integer;
       VAR indx: glindx; VAR d: real);
CONST
   tiny=1.0e-20;
VAR
   k,j,imax,i: integer;
   sum,dum,big: real;
   vv: glnarray;
BEGIN
   d := 1.0;
   FOR i := 1 to n DO
   BEGIN
      big := 0.0;
      FOR j := 1 to n DO
       IF abs(a[i,j]) > big THEN
                      big := abs(a[i,j]);
      IF big = 0.0 THEN
      BEGIN
      writeln('pause in LUDCMP - singular matrix');
      halt;
      END;
      vv[i] := 1.0/big;
   END;
   FOR j := 1 to n DO
   BEGIN
      IF j > 1 THEN
      BEGIN
         FOR i := 1 to j-1 DO
         BEGIN
            sum := a[i,j];
```

```
      IF i > 1 THEN
      BEGIN
         FOR k := 1 to i-1 DO
         BEGIN
            sum := sum-a[i,k]*a[k,j];
         END;
         a[i,j] := sum;
      END;
   END;
END;
big := 0.0;
FOR i := j to n DO
BEGIN
   sum := a[i,j];
   IF j > 1 THEN
   BEGIN
      FOR k := 1 to j-1 DO
      BEGIN
         sum := sum-a[i,k]*a[k,j];
      END;
      a[i,j] := sum;
   END;
   dum := vv[i]*abs(sum);
   IF dum > big THEN
   BEGIN
      big := dum;
      imax := i;
   END;
END;
IF j <> imax THEN
BEGIN
   FOR k := 1 to n DO
   BEGIN
      dum := a[imax,k];
      a[imax,k] := a[j,k];
      a[j,k] := dum;
   END;
   d := -d;
   vv[imax] := vv[j];
END;
indx[j] := imax;
IF j <> n THEN
BEGIN
```

```
         IF a[j,j] = 0.0 THEN a[j,j] := tiny;
         dum := 1.0/a[j,j];
         FOR i := j+1 to n DO
         BEGIN
            a[i,j] := a[i,j]*dum;
         END;
      END;
   END;
   IF a[n,n] = 0.0 THEN a[n,n] := tiny
END;
```

Az `indx` vektort kell, hogy használja az a program, amely a felbontott mátrix segítségével megoldja az egyenletrendszert. A `d` változó az `a` determinánsának a kiszámításához lehet szükséges. Ha az **A** mátrixot felbontottuk az **L** és **U** mátrixok szorzatára, akkor az eredeti egyenletrendszert nagyon egyszerűen, a (29) és (30) képletek segítségével lehet megoldani. Ezt végzi el az `lubksb` eljárás.

```
PROCEDURE lubksb(     a: glnpbynp;
                      n: integer;
                   indx: glindx;
                  VAR b: glnarray);
VAR
   j,ip,ii,i: integer;
   sum: real;
BEGIN
   ii := 0;
   FOR i := 1 to n DO
   BEGIN
      ip := indx[i];
      sum := b[ip];
      b[ip] := b[i];
      IF  ii <> 0 THEN
         FOR j := ii to i-1 DO
                sum := sum-a[i,j]*b[j]
     ELSE
      IF sum <> 0.0 THEN
                    ii := i;
   b[i] := sum;
   END;
   FOR i := n DOWNTO 1 DO
   BEGIN
      sum := b[i];
```

```
      IF i < n THEN
         FOR j := i+1 to n DO
            sum := sum-a[i,j]*b[j];
      b[i] := sum/a[i,i];
   END;
END;
```

Ha ez a két eljárás a rendelkezésünkre áll, akkor egy egyenletrendszer megoldása két sorból áll:

```
ludcmp(a,n,indx,d);
lubksb(a,n,indx,b)
```

Ha pedig a későbbiek folyamán ugyanezekkel az együtthatókkal, de egy új $\vec{c}$ konstans vektorral kívánjuk megoldani az egyenletet, akkor már csak a második eljárást kell meghívnunk:

```
lubksb(A,N,indx,c)
```

(Az elsőt nem is szabad!) Hogyan számítsuk ki a mátrix determinánsát, ha ismert az LU felbontás? A determinánsra igaz, hogy tetszőleges, de azonos méretű négyzetes mátrixok esetén:

$$\det \mathbf{L} \cdot \det \mathbf{U} = \det \mathbf{LU} \tag{41}$$

A mi esetünkben is igaz, most azonban a szorzat nem az $\mathbf{A}$ mátrixot adja meg, hanem egy abból sorcserékkel előállított mátrixot. Igy az $\mathbf{L} \cdot \mathbf{U}$ mátrix determinánsa plusz, vagy mínusz egyszerese lesz az $\mathbf{A}$ mátrix determinánsának, attól függően, hogy páros, vagy páratlan sok sor cserét végeztünk. Ezt az értéket adtuk vissza a `d` változóban az `ludcmp` eljárás végén. Az $\mathbf{L}$ és $\mathbf{U}$ mátrixok determinánsai nagyon egyszerűen kiszámíthatók, mivel háromszög mátrixok. Az ilyen mátrixok determináns értéke nem más mint a főátlóbeli elemek szorzata. Hogy a mi esetünkben ez még egyszerűbb legyen, az $\mathbf{L}$ mátrixra ez 1, mert minden α_{ii}-t egynek választottunk. A feladatunk csupán az, hogy az $\mathbf{U}$ mátrix főátlójában levő elemeket összeszorozzuk. Ezek az elemek pedig a felbontott mátrix főátlójában vannak.

1.8 Egyéb módszerek

A lineáris egyenletrendszerek numerikus megoldása igen fontos probléma. A legtöbb numerikusan megoldandó feladat végső soron erre a feladatra redukálható. Sajnos azonban a megoldás igen idő, és tárigényes. Mint láttuk a felvázolt mószerek idő és tárigénye az ismeretlenek számának négyzetével arányos. Nem is létezik olyan módszer, amely ennél kisebb időigényű lenne, és a gyakorlatban jól alkalmazható általános esetben.

A gyakorlatban előforduló esetek legtöbbször tartalmaznak valamilyen speciális megkötést, feltételt, ami lehetőséget nyújt arra, hogy gyorsabb, könyebben alkalmazható módszereket használjunk.

Egy egydimenziós differenciálegyenlet numerikus megoldása során a diszkretizálás után kapott lineáris egyenletrendszer például tridiagonális. Ez annyit jelent, hogy az együttható mátrix főátlója, valamint az ez alatt és felett elhelyezkedő elemek kivételével minden elem nulla. Egy ilyen speciális feladatnál idő és tár pocsékolás lenne akár a Gauss módszert, akár az LU dekompozíciót használni. Itt sokkal egyszerűbb megoldás is lehetséges.

Sok esetben az egyenletrendszer egy közelítő megoldását már ismerjük. Például egy időfüggő folyamatot számolunk, és ismert egy időpillanatbani eredmény, és egy nem sokkal későbbi időpillanatra kívánjuk meghatározni az egyenletrendszer megoldását. Ilyenkor tudjuk egy a korábbi

$$\mathbf{A}\vec{x} = \vec{b} \tag{42}$$

egyenlet eredményét, és tudjuk, hogy a most megoldandó

$$\mathbf{A}^{*}\vec{x}^{*} = \vec{b}^{*} \tag{43}$$

egyenletben az együttható mátrix és konstans vektor nem tér el sokkal a korábban már megoldott egyenletbelitől. Ilyen esetekben iterációs módszerek sokkal gyorsabb eredményt adhatnak, mint a hagyományos együttható eliminációs módszerek.

1.8.1 Tridiagonális rendszer

Egy olyan lineáris egyenlet rendszernél, ahol az együttható mátrix tridiagonális nagyon hatékonyan alkalmazható a Gauss elimináció. Természetesen nem abban az általános formában, amit a korábbiakban tárgyaltunk.

$$\begin{array}{ccccccc}
 & & a_{11}x_1 & + & a_{12}x_2 & = & b_1 \\
a_{21}x_1 & + & a_{22}x_2 & + & a_{23}x_3 & = & b_2 \\
a_{32}x_2 & + & a_{33}x_3 & + & a_{34}x_4 & = & b_3 \\
a_{43}x_3 & + & a_{44}x_4 & + & a_{45}x_5 & = & b_4 \\
\vdots & & \vdots & & \vdots & & \vdots \\
a_{n-1,n-2}x_{n-2} & + & a_{n-1,n-1}x_{n-1} & + & a_{n-1,n}x_n & = & b_{n-1} \\
a_{n,n-1}x_{n-1} & + & a_{n,n}x_n & & & = & b_n
\end{array} \tag{44}$$

Egy ilyen egyenlet rendszer megoldásánál a sorok megfelelelő elemeinek kinullázása megállhat egy-egy sor után, hiszen a tridiagonalitás miatt a többi együttható már amúgy is nulla.

1.8.2 Iterációs módszerek

Legyen a lineáris egyenlet rendszerünk

$$\mathbf{A}\vec{x} = \vec{b} \tag{45}$$

Ha az **A** mátrix determinánsa nem nulla, akkor az egyenlet rendszer megoldható. Alakítsuk át az egyenlet rendszert a

$$\vec{x} = \boldsymbol{\beta} + \boldsymbol{\alpha}\vec{x} \tag{46}$$

alakra. Ha ezt valamilyen módon megtettük, akkor választva valamilyen $\vec{x}_0$ kiindulási ismeretlen vektort, és alkalmazva a

$$\vec{x}_1 = \boldsymbol{\beta} + \boldsymbol{\alpha}\vec{x}_0 \tag{47}$$

egyenletet, egy újabb közelítést kapunk az ismeretlenekre. Ha ezt használjuk fel a következő iterációhoz, akkor ismét közelebb juthatunk a megoldáshoz. Általánosan felírva, az n-edik iterációból a

$$\vec{x}_{n+1} = \boldsymbol{\beta} + \boldsymbol{\alpha}\vec{x}_n \tag{48}$$

egyenlettel kaphatjuk meg a következő közelítést. Ez a megoldás persze nem mindig konvergens, és ezért nem szabad minden kritika nélkül alkalmazni.