**אלגברה לינארית נומרית**

אנחנו נעסוק כאן בתחום הצר של מציאת פתרונות של המערכת הריבועית

***A x = b***

כאשר A מטריצה ריבועית n x n הפיכה, b וקטור בגודל x 1 n נתון ואנחנו מעוניינים למצוא את x שהוא וקטור x 1 n.

כפי שמתואר לעיל, בשלב זה אנחנו יוצאים מתוך הנחה ש-A הפיכה והאלגוריתמים/תוכניות הראשונות שנראה בהמשך אינן בודקות זאת. לאחר שנסים לראות את צורות הפתרון של מטריצות הפיכות נראה איך להוסיף לאלגוריתם ולתוכניות את הבדיקה שהמטריצה היא הפיכה.

ישנם לא מעט שיטות לפתור מערכות של משוואות. לחלק מהן החשיבות שלהן היא בעיקר תיאורטית. למשל העובדה שפתרון של מערכת לינארית ניתן לייצוג/חישוב כמנה של דטרמיננטות מסוימות מובילה לאלגוריתם שהסיבוכיות הטבעית שלו הוא O(n4) כאשר קיימים פתרונות מעשיים בסיבוכיות O(n3).

קיימים גם פתרונות שבתאוריה הם מצוינים, כלומר בעלי סיבוכיות נמוכה מ- O(n3), אך בפועל בשל המורכבות הרבה של הפתרונות והמקדמים הגדולים המוסתרים תחת ההתייחסות לסדר הגודל, האלגוריתמים הללו הם טובים ביותר רק למערכות משוואות בגדלים ענקיים (ערכים גדולים מאד של n, בגדול של מליוני משתנים, נניח) שאינם נחשבים למעשיים.

אבל כאשר ניגשים לכתוב תוכניות בפועל שפותרות מערכות משוואות גדלות של מספרים המבוטאים במספרים בדיוק קבוע הנתמך ב-CPU ישנם שיקולים נוספים:

* יציבות נומרית (או חוסר דיוק הנובעות משגיאות עיגול)
* צורך בזיכרון נוסף בסדרי גודל של המערכת.

חוסר הדיוק יכול לנבוע למשל מפני שערך של מקדם במטריצה יכול להיות בסופו של דבר תוצאה של O(n2) חישובים. אם n גדול (סדר גודל של מאות או אלפים) זה יכול להיות בעיה. אבל מעבר לכך, חוסר דיוק יכול לנבוע גם בשל תכונות איכותיות של המערכת ולא רק העניין הכמותי. ישנם מטריצות ששגיאות קטנות במהלך החישוב/ייצוג של המטריצה גורמות זעזוע בפתרון. נראה את הדוגמא הבא:

*A x = b*

1. *1.0001 x1 1*

*=*

*1 1 x2 1*

למערכת הזו פתרון מדויק (x1, x2) = (1, 0).

לעומת זאת נייח שחל שינוי קטן ב-b:

*A x = b*

*1 1.0001 x1 1.0001*

*=*

*1 1 x2 1*

למערכת הזו פתרון מדויק (x1, x2) = (0, 1).

כלומר שינוי קטן במערכת גרם לזעזוע בפתרון.

היינו רוצים ששגיאות קטנות בחישוב/ייצוג המטריצה תהיה להן השפעה באותו סדר גודל בפתרון וזה נכון בדרך כלל אבל זה לא תמיד או יותר נכון זה עניין יחסי. מעיון בדוגמא שלעיל קל להבחין בכך ש-A הוא כמעט סינגולרי (בלתי הפיך) והמטריצה ההפיכה שלו היא

*-10000 10001*

*A-1 =*

*10000 -10000*

כלומר מורכב ממספרים מאד גדולים לעומת הערכים ב-A עצמו.

לסיכום ניתן לומר שדיוק של חישובים בפתרון מערכת לינארית הוא עניין חשוב שכן ההשפעה על איכות הפתרון, גם הכמותי וגם האיכותי, עשוי לעוות את התוצאה.

אנחנו לא נתייחס באופן מיוחד איך מטפלים במטריצות בעיתיות מהסוג הזה אבל אנחנו כן נראה באיזה שיטות ניתן לנקות בכדי לחסוך שגיאות עיגול שאינן מחויבות המציאות.

השיטה הקלאסית של פתרון מערכת משוואות לינארית היא האלימינציה של גאוס, ועבור מערכות לא גדולות מדי היא אכן השיטה שבא משתמשים.

**אלימינציה של גאוס**

האלימנציה של גאוס הוא אלגוריתם הנלמד בכל קורס של אלגברה לינארית ולא נתייחס כמעט לאספקטים התאורטיים שלה. בעקרון היא מסתמכת על העובדה שפעולות על השורות כמו חלוקה של כל אברי שורה של מערכת בקבוע, הוספה/החסרה של כפולה כלשהיא של שורה אחת בשורה אחרת , החלפת שורה אחת של המטריצה באחרת כל אחת מהפעולות הללו יוצרת מערכת משוואות שקולה (קרי: הפתרון של הקודם הוא גם פתרון של החדש). אם כך ניתן לעשות סדרה מסוימת מאד של פעולות שורות אשר מאפשר לנו להביא את המערכת להיות מערת **משולשת**  (עליונה או תחתונה) דבר המאפשר לנו למצוא את הפתרון בהליך פשוט יחסית של **נסיגה**. xn נתון מיידית כמנה של שני מקדמים, xn-1 תלוי רק במקדמים הנתונים ובערך של xn וכו'. כל איבר חדש בוקטור הפתרון תלוי רק במקדמים שחושבו וערכי וקטור הפתרון מהאינדקס שלו והלאה.

האלימינציה של גאוס היא דוגמא קלאסית של אלגוריתם נומרי שהוא אלגוריתם **סגור** ולא איטרטיבי כמו שרואים באנליזה נומרית רוב הזמן.

לדוגמא, נראה את המערכת הבאה:

*A x = b*

*2 1 1 x1 1*

1. *0 5 x2 = 3*

*1 2 0 x3 2*

אפשר להחסיר מהשורה השנייה את הורה הראשונה פי 2, ומהשורה השלישית את השורה הראשונה פי ½ נקבל:

*2 1 1 x1 1*

*0 -2 3 x2 = 1*

*0 1.5 -0.5 x3 1.5*

עכשיו נוכל להוסיף לשורה השלישית את השורה השנייה פי ¾ ואז

נקבל

*2 1 1 x1 1*

*0 -2 3 x2 = 1*

*0 0 1.75 x3 2.25*

עכשיו אנחנו יכולים לחשב לחשב את הוקטור x.

x3 = 2.25/1.75 = 1.285714

x2 = -(1/2)(1 – 3x3) = -(1/2)(1-3\*1.285714) = 1.42857

x1 = (1/2)(1 – x2 –x1) = (1/2)(1 – 1.285714 - 1.42857 ) =

-0.857143

יחד עם זאת, יש כאן כמה בעיות. קודם כל שום דבר לא מבטיח לנו שהאיבר ai,i , איבר האיטרציה של שלב ה-i, שונה מאפס. אם הוא תוצאה של חישוב, הוא יכול להיות תאורטית אפס אבל בפועל קטן מאד (לפחות יחסית למקדמים האחרים) ויותר גרוע – חסר משמעות.

פתרון אפשרי לבעיה הזו היא שבכל שלב i נסרוק את יתרת העמודה ה-i –יית, נמצא את השורה שהערך שלה בעמודה ה-i-יית הוא הגדול ביותר בערכו המוחלט ונהפוך אותו לשורה ה-i-יית של המטריצה. הפעולה הזו של החלפת סדר המשוואות במערך לא משפיעה על הפתרון, לא על הערכים ולא על סדר המשתנים. הפעולה הזו נקראית partial pivoting.

בעיה נוספת היא שאנחנו משנים דרסטית את ערכי המקדמים של המטריצה, ואם צריכים את הערכים המקוריים צריך לשכפל את המטריצה.

בעיה נוספת היא כאן הוא מגבלה של C:ב-C (בניגוד ללא מעט שפות אחרות) אי אפשר להקצות מערך דו מימדי דינאמי, ולפיכך נצטרך לעקוף את הבעיה בעזרת מערך דו מימדי דינאמי.

רוטינת ה-C הבאה מיישמת את האלימינציה של גאוס בעזרת partial pivoting:

void swap\_rows(double \*A[], int n, int m1, int m2)

{

int i;

double temp;

for(i=0; i <= n; i++)

{

temp = A[m1][i];

A[m1][i] = A[m2][i];

A[m2][i] = temp;

} /\* for \*/

} /\* swap\_rows \*/

void gaussian(double \*A[], double b[], int n, double x[])

{

int i, j, k, p;

double \*\*W;

double \*\*M;

double MaxValue;

M = (double \*\*)malloc(n\*sizeof(double \*));

W = (double \*\*)malloc(n\*sizeof(double \*));

for(i=0; i < n; i++)

W[i] = (double \*)malloc((n+1)\*sizeof(double));

for(i=0; i < n; i++)

M[i] = (double \*)malloc(n\*sizeof(double));

for(i=0; i < n; i++)

for(j=0; j < n; j++)

W[i][j] = A[i][j];

for(i=0; i < n; i++)

W[i][n] = b[i];

for (k=0; k < n; k++)

{

p = k;

MaxValue = fabs(W[k][k]);

for(i=k+1; i < n; i++)

if (fabs(W[i] [k]) > MaxValue)

{

p = i;

MaxValue = fabs(W[i] [k]);

}

swap\_rows(W, n, k, p);

for(i=k+1; i < n; i++)

M[i][k] = W[i][k]/W[k][k];

for(i=k+1; i < n; i++)

W[i][k] = 0;

for(i=k+1; i < n; i++)

for(j=k+1; j <= n; j++)

W[i][j] = W[i][j] - M[i][k]\*W[k][j];

} /\* for \*/

x[n-1] = W[n-1][n]/W[n-1][n-1];

for(i=n-2; i >= 0; i--)

{

double temp;

temp = W[i][n];

for(k=i+1; k < n; k++)

temp = temp - W[i][k]\*x[k];

x[i] = temp/W[i][i];

} /\* for \*/

for(i=0; i < n; i++)

{

free(M[i]);

free(W[i]);

} /\* for \*/

free(M);

free(W);

} /\* gaussian \*/

מבחינת הסיבוכיות של האלגוריתם, בשלב ליכסון המטריצה אנחנו עושים n איטרציות שכל אחד מהם מאפס את עמודה אחת. המחיר של כל איטרציה היא O(n2) שכן מציאת שורת ה-pivot הוא O(n) אבל סיבוכיות הפעולות של השורות הוא O(n2).

לפיכך סה"כ הפעולות למציאת המטריצה האלכסונית הוא O(n3). שלב הנסיגה הוא O(n2) , זהו גם המחיר של שכפול המטריה (אם עושים זאת, כמו בקוד שלעיל) לפיכך הסיבוכיות הכוללת נקבעת למעשה ע"י שלב הליכסון שהוא הדומיננטי שהוא O(n3).

**האחדת סדר הגודל Scaling**

שיפור נוסף שעשוי לעזור בצמצום ההשפעה של שגיאות העיגול טכניקה שנקראית scaling שמשמעותה להביא את ערכי המטריצה למספרים בעלי אותו סדר גודל אם הדבר אפשרי. לדוגמא, מקור ערכי המטריצה יכול להיות משוואות של איזה שהוא יחסי גומלין בין תהליכים כימיים במפעל וכל משוואה מנוסחת במימדים אחרים כמו ק"ג, ליטרים, מטרים מעוקבים וכו'. מבחינה נומרית משמעות הדבר מקדמים גדולים במשוואה אחת ומקדמים קטנים במשוואה אחרת, כאשר אנחנו מוצאים את עצמנו מחלקים או מכפלים מספרים גדולים במספרים קטנים כאשר בשלב הבא אנחנו מכפילים או מחלקים במספרים מן הסוג ההפוך. מבחינה מתמטית זה לא משנה אבל מבחינת דיוק נומרי זה יכול לשנות, אנחנו גם עשויים לחרוג מתחום הייצוג של מספרים. תמיד כשאפשר מנסים לבצע פעולות על מספרים באותו סדר גודל. הטכניקה הפשוטה ביותר במקרה שלנו יהיה פשוט לחלק כל שורה הערך המוחלט של המקדם הגדול ביותר בשורה. כלומר

Si = maxj { |ai,j| }

ai,j = ai,j/si

הרוטינה הבאה מממשת את האלימינציה של גאוס תוך שימוש ב-partial pivoting בשילוב עם scaling:

void gaussian(double \*A[], double b[], int n, double x[])

{

int i, j, k, p;

double \*\*W;

double \*\*M;

double MaxValue, ScaleValue, temp;

M = (double \*\*)malloc(n\*sizeof(double \*));

W = (double \*\*)malloc(n\*sizeof(double \*));

for(i=0; i < n; i++)

W[i] = (double \*)malloc((n+1)\*sizeof(double));

for(i=0; i < n; i++)

M[i] = (double \*)malloc(n\*sizeof(double));

for(i=0; i < n; i++)

for(j=0; j < n; j++)

W[i][j] = A[i][j];

for(i=0; i < n; i++)

W[i][n] = b[i];

for(i=0; i < n; i++)

{

ScaleValue = fabs(W[i][0]);

for(j=1; j < n; j++)

{

temp = fabs(W[i][j]);

if (temp > ScaleValue)

ScaleValue = temp;

}

for(j=0; j <= n; j++)

W[i][j] = W[i][j]/ScaleValue;

} /\* for \*/

for (k=0; k < n; k++)

{

p = k;

MaxValue = fabs(W[k][k]);

for(i=k+1; i < n; i++)

if (fabs(W[i][k]) > MaxValue)

{

p = i;

MaxValue = fabs(W[i][k]);

}

swap\_rows(W, n, k, p);

for(i=k+1; i < n; i++)

M[i][k] = W[i][k]/W[k][k];

for(i=k+1; i < n; i++)

W[i][k] = 0;

for(i=k+1; i < n; i++)

for(j=k+1; j <= n; j++)

W[i][j] = W[i][j] - M[i][k]\*W[k][j];

} /\* for \*/

x[n-1] = W[n-1][n]/W[n-1][n-1];

for(i=n-2; i >= 0; i--)

{

double temp;

temp = W[i][n];

for(k=i+1; k < n; k++)

temp = temp - W[i][k]\*x[k];

x[i] = temp/W[i][i];

} /\* for \*/

for(i=0; i < n; i++)

{

free(M[i]);

free(W[i]);

} /\* for \*/

free(M);

free(W);

} /\* gaussian \*/

**Full pivoting**

שיפור נוסף שאפשר לעשות הוא לשדרג את מציאת איבר ה-pivot. במקום לחפש אותו רק בעמודה ה-i-יית נחפש אותו בכל תת המטריצה שנותרה. אחרי שנמצא אותו נחליף הן את השורה והן את העמודה. התהליך הזה אינו משנה את הסיבוכיות הכוללת. אמנם הסיבוכיות של מציאת ה-pivot עולה מ-O(n) ל-O(n2), אבל זה מצטבר למהלך של איפוס עמודה שהוא ממילא O(n2).

ברמה הטכנית החלפת עמודה שונה מהחלפת שורה. היא משנה את הסדר של המקדמים בווקטור הפתרון x. לפיכך, אם אנחנו רוצים להחזיר את ערכי המקדמים בסדר של המטריצה המקורית, אנחנו צריכים לנהל מעקב על העמודות שהחלפנו ולשחזר אותם בסוף.

הרוטינה הבאה מישמת את האלימינציה של גאוס תוך שימוש ב-full pivoting ו-scaling:

void swap\_rows(double \*A[], int n, int m1, int m2)

{

int i;

double temp;

for(i=0; i <= n; i++)

{

temp = A[m1][i];

A[m1][i] = A[m2][i];

A[m2][i] = temp;

} /\* for \*/

} /\* swap\_rows \*/

void swap\_cols(double \*A[], int n, int m1, int m2, int xindex[])

{

int i, itemp;

double dtemp;

itemp = xindex[m1];

xindex[m1] = xindex[m2];

xindex[m2] = itemp;

for(i=0; i < n; i++)

{

dtemp = A[i][m1];

A[i][m1] = A[i][m2];

A[i][m2] = dtemp;

} /\* for \*/

} /\* swap\_cols \*/

void gaussian(double \*A[], double b[], int n, double x[])

{

int i, j, k, p, q;

double \*\*W;

double \*\*M;

double \*y;

int \*xindex;

double MaxValue, ScaleValue, temp;

M = (double \*\*)malloc(n\*sizeof(double \*));

W = (double \*\*)malloc(n\*sizeof(double \*));

for(i=0; i < n; i++)

W[i] = (double \*)malloc((n+1)\*sizeof(double));

for(i=0; i < n; i++)

M[i] = (double \*)malloc(n\*sizeof(double));

for(i=0; i < n; i++)

for(j=0; j < n; j++)

W[i][j] = A[i][j];

for(i=0; i < n; i++)

W[i][n] = b[i];

xindex = (int \*)malloc(n\*sizeof(int));

for(i=0; i < n; i++)

xindex[i] = i;

for(i=0; i < n; i++)

{

ScaleValue = fabs(W[i][0]);

for(j=1; j < n; j++)

{

temp = fabs(W[i][j]);

if (temp > ScaleValue)

ScaleValue = temp;

}

for(j=0; j <= n; j++)

W[i][j] = W[i][j]/ScaleValue;

} /\* for \*/

for (k=0; k < n; k++)

{

p = k;

q = k;

MaxValue = fabs(W[k][k]);

for(i=k; i < n; i++)

for(j=k; j < n; j++)

if (fabs(W[i][j]) > MaxValue)

{

p = i;

q = j;

MaxValue = fabs(W[i][j]);

} /\* if \*/

swap\_cols(W, n, k, q, xindex);

swap\_rows(W, n, k, p);

for(i=k+1; i < n; i++)

M[i][k] = W[i][k]/W[k][k];

for(i=k+1; i < n; i++)

W[i][k] = 0;

for(i=k+1; i < n; i++)

for(j=k+1; j <= n; j++)

W[i][j] = W[i][j] - M[i][k]\*W[k][j];

} /\* for \*/

y = (double \*)malloc(n\*sizeof(double));

y[n-1] = W[n-1][n]/W[n-1][n-1];

for(i=n-2; i >= 0; i--)

{

double temp;

temp = W[i][n];

for(k=i+1; k < n; k++)

temp = temp - W[i][k]\*y[k];

y[i] = temp/W[i][i];

} /\* for \*/

for(i=0; i < n; i++)

x[xindex[i]] = y[i];

free(xindex);

free(y);

for(i=0; i < n; i++)

{

free(M[i]);

free(W[i]);

} /\* for \*/

free(M);

free(W);

} /\* gaussian \*/

**איתור מטריצות סינגולריות**

תאורטית, אם אלימינציה של גאוס מופעלת על מטריצה סינגולרית, היא תחשב בשלב כלשהוא שורה שכולה אפס. בפועל, כאשר הדבר נעשה ע"י מחשב העובד בדיוק סופי הדבר יבוא לידי ביטוי שהאלימינציה תחשב שורה של מספרים שבערכם המוחלט קטן מאד לפחות יחסית לערכי השורה הזו בתחילת האלימינציה. הנקודה היא שהשאלה מהו מספר קטן וחסר משמעות באלימינציה הרגילה של גאוס הוא עניין יחסי שכן אם המספרים המקוריים של הערכים במטריצה הם, נניח, בסדר גודל של 1020 אזי מספרים בסדר גודל של 104 יכולים להיחשב כ"אפס", תלוי בדיוק.

אלא שכאן בא לעזרתנו ה-scaling. מאחר ואנחנו ממילא בטכניקה הזו

את כל המספרים לתחום (1.0, 1.0-) הרי זה מאפשר לנו לקבוע, באופן בלתי תלוי בערכי המטריצה המקורית , קבוע אפסילון שכל מספר קטן ממנו בערכו המוחלט יחשב כ-"אפס".

הרוטינה הבאה היא גרסה של הרוטינות הקודמת של האלימינציה של גאוס המשתמשת ב-scaling ו-partial pivoting הבודק אם המטריצה בלחתי הפיכה ומחזיר 0 במקרה הזה. אם המטריצה הפיכה, היא תחזיר 1.

void swap\_rows(double \*A[], int n, int m1, int m2)

{

int i;

double temp;

for(i=0; i <= n; i++)

{

temp = A[m1][i];

A[m1][i] = A[m2][i];

A[m2][i] = temp;

} /\* for \*/

} /\* swap\_rows \*/

int gaussian(double \*A[], double b[], int n, double x[])

{

int i, j, k, p;

double \*\*W;

double \*\*M;

double MaxValue, ScaleValue, temp;

double epsilon = 0.0000001;

M = (double \*\*)malloc(n\*sizeof(double \*));

W = (double \*\*)malloc(n\*sizeof(double \*));

for(i=0; i < n; i++)

W[i] = (double \*)malloc((n+1)\*sizeof(double));

for(i=0; i < n; i++)

M[i] = (double \*)malloc(n\*sizeof(double));

for(i=0; i < n; i++)

for(j=0; j < n; j++)

W[i][j] = A[i][j];

for(i=0; i < n; i++)

W[i][n] = b[i];

for(i=0; i < n; i++)

{

ScaleValue = fabs(W[i][0]);

for(j=1; j < n; j++)

{

temp = fabs(W[i][j]);

if (temp > ScaleValue)

ScaleValue = temp;

}

for(j=0; j <= n; j++)

W[i][j] = W[i][j]/ScaleValue;

} /\* for \*/

for (k=0; k < n; k++)

{

/\* Check if matrix is singular by

testng if the current row is zero \*/

MaxValue = 0;

for(j=0; j < n; j++)

{

temp = fabs(W[k][j]);

if (MaxValue < temp)

MaxValue = temp;

} /\* for \*/

if (MaxValue < epsilon) /\* Row of zeros? \*/

return 0;

/\* End of singular check \*/

p = k;

MaxValue = fabs(W[k][k]);

for(i=k+1; i < n; i++)

if (fabs(W[i] [k]) > MaxValue)

{

p = i;

MaxValue = fabs(W[i][k]);

}

swap\_rows(W, n, k, p);

for(i=k+1; i < n; i++)

M[i][k] = W[i][k]/W[k][k];

for(i=k+1; i < n; i++)

W[i][k] = 0;

for(i=k+1; i < n; i++)

for(j=k+1; j <= n; j++)

W[i][j] = W[i][j] - M[i][k]\*W[k][j];

} /\* for \*/

x[n-1] = W[n-1][n]/W[n-1][n-1];

for(i=n-2; i >= 0; i--)

{

double temp;

temp = W[i][n];

for(k=i+1; k < n; k++)

temp = temp - W[i][k]\*x[k];

x[i] = temp/W[i][i];

} /\* for \*/

for(i=0; i < n; i++)

{

free(M[i]);

free(W[i]);

} /\* for \*/

free(M);

free(W);

return 1;

} /\* gaussian \*/

הרוטינה הבאה היא הגרסה המעודכנת של הרוטינה של scaling עם full pivoting:

void swap\_rows(double \*A[], int n, int m1, int m2)

{

int i;

double temp;

for(i=0; i <= n; i++)

{

temp = A[m1][i];

A[m1][i] = A[m2][i];

A[m2][i] = temp;

} /\* for \*/

} /\* swap\_rows \*/

void swap\_cols(double \*A[], int n, int m1, int m2, int xindex[])

{

int i, itemp;

double dtemp;

itemp = xindex[m1];

xindex[m1] = xindex[m2];

xindex[m2] = itemp;

for(i=0; i < n; i++)

{

dtemp = A[i][m1];

A[i][m1] = A[i][m2];

A[i][m2] = dtemp;

} /\* for \*/

} /\* swap\_cols \*/

int gaussian(double \*A[], double b[], int n, double x[])

{

int i, j, k, p, q;

double \*\*W;

double \*\*M;

double \*y;

int \*xindex;

double MaxValue, ScaleValue, temp;

double epsilon = 0.0000001;

M = (double \*\*)malloc(n\*sizeof(double \*));

W = (double \*\*)malloc(n\*sizeof(double \*));

for(i=0; i < n; i++)

W[i] = (double \*)malloc((n+1)\*sizeof(double));

for(i=0; i < n; i++)

M[i] = (double \*)malloc(n\*sizeof(double));

for(i=0; i < n; i++)

for(j=0; j < n; j++)

W[i][j] = A[i][j];

for(i=0; i < n; i++)

W[i][n] = b[i];

xindex = (int \*)malloc(n\*sizeof(int));

for(i=0; i < n; i++)

xindex[i] = i;

for(i=0; i < n; i++)

{

ScaleValue = fabs(W[i][0]);

for(j=1; j < n; j++)

{

temp = fabs(W[i][j]);

if (temp > ScaleValue)

ScaleValue = temp;

}

for(j=0; j <= n; j++)

W[i][j] = W[i][j]/ScaleValue;

} /\* for \*/

for (k=0; k < n; k++)

{

/\* Check if matrix is singular by

testng if the current row is zero \*/

MaxValue = 0;

for(j=0; j < n; j++)

{

temp = fabs(W[k][j]);

if (MaxValue < temp)

MaxValue = temp;

} /\* for \*/

if (MaxValue < epsilon) /\* Row of zeros? \*/

return 0;

/\* End of singular check \*/

p = k;

q = k;

MaxValue = fabs(W[k][k]);

for(i=k; i < n; i++)

for(j=k; j < n; j++)

if (fabs(W[i][j]) > MaxValue)

{

p = i;

q = j;

MaxValue = fabs(W[i][j]);

} /\* if \*/

swap\_cols(W, n, k, q, xindex);

swap\_rows(W, n, k, p);

for(i=k+1; i < n; i++)

M[i][k] = W[i][k]/W[k][k];

for(i=k+1; i < n; i++)

W[i][k] = 0;

for(i=k+1; i < n; i++)

for(j=k+1; j <= n; j++)

W[i][j] = W[i][j] - M[i][k]\*W[k][j];

} /\* for \*/

y = (double \*)malloc(n\*sizeof(double));

y[n-1] = W[n-1][n]/W[n-1][n-1];

for(i=n-2; i >= 0; i--)

{

double temp;

temp = W[i][n];

for(k=i+1; k < n; k++)

temp = temp - W[i][k]\*y[k];

y[i] = temp/W[i][i];

} /\* for \*/

for(i=0; i < n; i++)

x[xindex[i]] = y[i];

free(xindex);

free(y);

for(i=0; i < n; i++)

{

free(M[i]);

free(W[i]);

} /\* for \*/

free(M);

free(W);

return 1;

} /\* gaussian \*/

תוכנית המעוניינת להביא בחשבון את האפשרות שהמטריצה היא סינגולרית תצטרך לשמור לכל שורה את המספר שהוא בערך מוחלט הקטן ביותר בשורה ובכל שלב לבדוק את השורה הנוכחית אם כל המספרים הנותרים בערכם המוחלט הם בגבול הדיוק הנומרי יחסית למספר הנשמר בצד לשורה. במידה וזה מתקיים הסיכויים הם שהמטריצה היא סינגולרית.