

Algorithmes numériques – Rapport

Résolution de systèmes linéaires

Axel Delsol, Pierre-Loup Pissavy

Novembre 2013

# Table des matières

<b>1</b>	<b>Préambule</b>	<b>2</b>
1.1	Structure du programme . . . . .	2
1.2	Compilation . . . . .	3
1.3	Systèmes testés . . . . .	4
<b>2</b>	<b>Méthodes directes</b>	<b>5</b>
2.1	Méthode de Gauss . . . . .	5
2.1.1	Programme . . . . .	5
2.1.2	Résultats des tests . . . . .	7
2.2	Méthode de Cholesky . . . . .	9
2.2.1	Programme . . . . .	9
2.2.2	Résultats des tests . . . . .	11
2.3	Comparaison . . . . .	13
<b>3</b>	<b>Méthodes itératives</b>	<b>14</b>
3.1	Méthode de Jacobi . . . . .	14
3.1.1	Programme . . . . .	14
3.1.2	Résultats des tests . . . . .	16
3.2	Méthode de Gauss-Seidel . . . . .	18
3.2.1	Programme . . . . .	18
3.2.2	Résultats de tests . . . . .	19
3.3	Comparaison . . . . .	21
<b>4</b>	<b>Conclusion</b>	<b>22</b>
<b>5</b>	<b>Annexe</b>	<b>23</b>
5.1	Fonctions caractéristiques du calcul matriciel . . . . .	23
5.2	Génération de matrices aléatoires . . . . .	27
5.3	Méthode de Sur-relaxation . . . . .	34
5.3.1	Programme . . . . .	34

# 1 Préambule

## 1.1 Structure du programme

Nous avons conçu un programme principal avec menus, présenté sous la forme suivante :

```
MENU PRINCIPAL : RESOLUTION D'EQUATIONS LINEAIRES

1. Résolution par Gauss
3. Résolution par Cholesky
2. Résolution par Jacobi
4. Résolution par Gauss-Seidel
5. Résolution par Surrelaxation
6. Génération de matrices carrées
7. Jeux de test
0. Quitter
Votre choix :
```

FIGURE 1.1 – Aperçu : Menu Principal

Toutes les fonctions de résolution font appel à des fonctions adaptées aux matrices, écrites dans le fichier `matrices.c`.

Chaque méthode de résolution reçoit les matrices (à éléments réels) générées auparavant (juste après le choix de la méthode dans le menu) en arguments.

Chaque entrée du menu est codée dans un fichier source qui lui est propre (cf Figure 1.2). Les fichiers headers correspondants contiennent les prototypes. Le fichier source `main.c` contient le menu principal.

Les matrices sont générées juste après le choix de la méthode, avant d'être passées en arguments à la fonction de résolution.

```
gauss.c
gauss-seidel.c
generateur.c
jacobi.c
main.c
matrices.c
surrelaxation.c
cholesky.h
gauss.h
gauss-seidel.h
generateur.h
jacobi.h
matrices.h
surrelaxation.h
makefile
```

FIGURE 1.2 – Aperçu : Arborescence

Toutes les fonctions de résolution font appel à des fonctions intermédiaires, caractéristiques du calcul matriciel, écrites dans le fichier `matrices.c` présenté en annexe, page 23.

## 1.2 Compilation

La compilation est gérée par un `makefile`.

Le compilateur utilisé est `GCC`. Il suffit de taper `make` pour lancer la compilation.

Pour nettoyer les fichiers temporaires, il faudra taper `make clean`.

Ce `makefile` permet également de générer ce rapport ainsi que quelques fichiers qui y sont intégrés.

### 1.3 Systèmes testés

Les matrices suivantes sont toutes symétriques définies positives à diagonale dominante. Nous avons choisi de prendre 10 systèmes pouvant être résolus par toutes les méthodes pour faciliter la comparaison des méthodes directes, itératives et directes-itératives.

Systèmes :

$$\begin{pmatrix} 2 & 1 & 0 & 0 \\ 1 & 3 & 1 & 0 \\ 0 & 1 & 5 & 1 \\ 0 & 0 & 1 & 2 \end{pmatrix} \cdot \begin{pmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \end{pmatrix} = \begin{pmatrix} 1 \\ 0 \\ 0 \\ -1 \end{pmatrix} \quad (1)$$

$$\begin{pmatrix} 19 & 0 & 12 & -6 \\ 0 & 4 & 2 & 1 \\ 12 & 2 & 49 & -4 \\ -6 & 1 & -4 & 51 \end{pmatrix} \cdot \begin{pmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \end{pmatrix} = \begin{pmatrix} 1 \\ 1 \\ 1 \\ 1 \end{pmatrix} \quad (2)$$

$$\begin{pmatrix} 5 & 1 & 1 & 1 \\ 1 & 13 & 5 & 5 \\ 1 & 5 & 49 & 14 \\ 1 & 5 & 14 & 51 \end{pmatrix} \cdot \begin{pmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \end{pmatrix} = \begin{pmatrix} 0 \\ 2 \\ 4 \\ 6 \end{pmatrix} \quad (3)$$

$$\text{Matrice à bord : } \begin{pmatrix} 1 & 0.5 & 0.25 \\ 0.5 & 1 & 0 \\ 0.25 & 0 & 1 \end{pmatrix} \cdot \begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix} = \begin{pmatrix} 1 \\ 2 \\ 1 \end{pmatrix} \quad (4)$$

$$\begin{pmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 9 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ 0 & 1 & 3 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 8 & 1 & 1 & 1 & 1 & 1 & 1 \\ 0 & 1 & 0 & 1 & 4 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 1 & 0 & 7 & 1 & 1 & 1 & 1 \\ 0 & 1 & 0 & 1 & 0 & 1 & 4 & 0 & 0 & 0 \\ 0 & 1 & 0 & 1 & 0 & 1 & 0 & 6 & 1 & 1 \\ 0 & 1 & 0 & 1 & 0 & 1 & 0 & 1 & 5 & 0 \\ 0 & 1 & 0 & 1 & 0 & 1 & 0 & 1 & 0 & 5 \end{pmatrix} \cdot \begin{pmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \\ x_5 \\ x_6 \\ x_7 \\ x_8 \\ x_9 \\ x_{10} \end{pmatrix} = \begin{pmatrix} 1 \\ 2 \\ 3 \\ 4 \\ 5 \\ 6 \\ 7 \\ 8 \\ 9 \\ 10 \end{pmatrix} \quad (5)$$

$$\begin{pmatrix} 4 & 1 & 0 \\ 1 & 4 & 1 \\ 0 & 1 & 4 \end{pmatrix} \cdot \begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix} = \begin{pmatrix} -5 \\ 6 \\ 7 \end{pmatrix} \quad (6)$$

$$\begin{pmatrix} 7 & -2 & 0 \\ -2 & 9 & -6 \\ 0 & -6 & 25 \end{pmatrix} \cdot \begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix} = \begin{pmatrix} 3/2 \\ 3/5 \\ 3/4 \end{pmatrix} \quad (7)$$

$$\begin{pmatrix} 4 & 1 & 1 & 0 \\ 1 & 4 & 0 & 1 \\ 1 & 0 & 4 & 1 \\ 0 & 1 & 1 & 4 \end{pmatrix} \cdot \begin{pmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \end{pmatrix} = \begin{pmatrix} 0 \\ 0 \\ 0 \\ 24 \end{pmatrix} \quad (8)$$

$$\begin{pmatrix} 65 & 40 & 24 \\ 40 & 68 & 17 \\ 24 & 17 & 81 \end{pmatrix} \cdot \begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix} = \begin{pmatrix} 1 \\ 1 \\ 1 \end{pmatrix} \quad (9)$$

$$\text{Matrice KMS } (p = 0.25) : \begin{pmatrix} 1 & 0.25 & 0.0625 & 0.015625 \\ 0.25 & 1 & 0.25 & 0.0625 \\ 0.0625 & 0.25 & 1 & 0.25 \\ 0.015625 & 0.0625 & 0.25 & 1 \end{pmatrix} \cdot \begin{pmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \end{pmatrix} = \begin{pmatrix} 1 \\ 2 \\ 3 \\ 4 \end{pmatrix} \quad (10)$$

## 2 Méthodes directes

Les méthodes directes ont pour but d'obtenir des solutions exactes en simplifiant, étape par étape, le système donné sous la forme  $A_{n,n} \cdot X_{n,1} = B_{n,1}$  où :

- $A$  est une matrice donnée
- $X$  est le vecteur solution
- $B$  est un vecteur colonne donné

La résolution se fait en 2 étapes :

1. On décompose la matrice  $A_{n,n}$  tel que  $A = M \cdot N$  où  $M_{n,n}$  est facile à inverser et  $N_{n,n}$  est triangulaire.
2. On résout les systèmes suivants :
  - On trouve  $Y_{n,1}$  tel que  $M \cdot Y = B$
  - On trouve enfin  $X_{n,1}$  tel que  $N \cdot X = Y$

### 2.1 Méthode de Gauss

La méthode de Gauss permet de calculer une solution exacte en un nombre fini d'étapes.

On cherche la matrice  $N$  triangulaire supérieure telle que  $A = M \cdot N$  avec  $M$  la matrice identité.

Remarque : La résolution du système  $M \cdot Y = B$  est évidente puisque la matrice  $M$  est la matrice identité.

Critère d'application de l'algorithme :

- Les éléments diagonaux ne peuvent être nuls,
- Le déterminant ne doit pas être nul.

#### 2.1.1 Programme

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include "matrices.h"
4
5  void gauss (double** a, double** b, int n)
6  {
7      //initialisation
8      int i, j, k;
9      double pivot;
10
11     //gauss
12     for (k=0; k<n-1; k++)
13     {
14         for (i=k+1; i<n; i++)
15         {
16             pivot = a[i][k]/a[k][k];
17             for (j=k+1; j<n; j++)
18             {
19                 a[i][j]=a[i][j]-pivot*a[k][j];
20             }
21             b[i][0]=b[i][0]-pivot*b[k][0];
22         }
23         printf("\nMatrice :\n");
24         afficherMatrice(a,n,n);
25         printf("Vecteur :\n");
26         afficherMatrice(b,n,1);
```

```

27 | }
28 |
29 | //calcul et affichage resultat
30 | double** x=solveTriangulaireSup(a, b, n);
31 | printf("\nRésultat :\n");
32 | afficherMatrice(x, n, 1);
33 |
34 | //libération mémoire
35 | for (i=0;i<n;i++)
36 | {
37 |     free(x[i]);
38 | }
39 | free(x);
40 | }

```

FIGURE 2.1 – Code : gauss.c

### 2.1.2 Résultats des tests

Système	Résultat obtenu	Résultat théorique	Ecart
(1)	$\begin{pmatrix} 0.63415 \\ -0.26829 \\ 0.17073 \\ -0.58537 \end{pmatrix}$	$\begin{pmatrix} \frac{26}{41} \\ -\frac{11}{41} \\ \frac{7}{41} \\ -\frac{24}{41} \end{pmatrix}$	0.0008213058895
(2)	$\begin{pmatrix} 0.06132 \\ 0.24599 \\ -0.00287 \\ 0.02177 \end{pmatrix}$	$\begin{pmatrix} \frac{3067}{50015} \\ \frac{7382}{30009} \\ -\frac{431}{150045} \\ \frac{1089}{50015} \end{pmatrix}$	0.02643817358
(3)	$\begin{pmatrix} -0.04885 \\ 0.10345 \\ 0.04458 \\ 0.09623 \end{pmatrix}$	$\begin{pmatrix} -\frac{1181}{20928} \\ \frac{563}{6976} \\ \frac{121}{1308} \\ \frac{95}{872} \end{pmatrix}$	26.27445408
(4)	$\begin{pmatrix} -0.36364 \\ 2.18182 \\ 1.09091 \end{pmatrix}$	$\begin{pmatrix} -\frac{4}{11} \\ \frac{24}{11} \\ \frac{12}{11} \end{pmatrix}$	0.0003888866667
(5)	$\begin{pmatrix} 1.00000 \\ -0.81145 \\ 1.27048 \\ -0.46016 \\ 1.56790 \\ 0.05780 \\ 2.05345 \\ 0.88005 \\ 1.86675 \\ 2.06675 \end{pmatrix}$	$\begin{pmatrix} 1 \\ -\frac{241}{297} \\ \frac{1132}{891} \\ -\frac{410}{891} \\ \frac{127}{81} \\ \frac{103}{1782} \\ \frac{4879}{2376} \\ \frac{697}{792} \\ \frac{22177}{11880} \\ \frac{24553}{11880} \end{pmatrix}$	0.0001766790236



Système	Résultat obtenu	Résultat théorique	Ecart
(6)	$\begin{pmatrix} -1.64286 \\ 1.57143 \\ 1.35714 \end{pmatrix}$	$\begin{pmatrix} -\frac{23}{14} \\ \frac{11}{7} \\ \frac{19}{14} \end{pmatrix}$	0.0001584521670
(7)	$\begin{pmatrix} 0.26370 \\ 0.17294 \\ 0.07150 \end{pmatrix}$	$\begin{pmatrix} \frac{645}{2446} \\ \frac{423}{2446} \\ \frac{1749}{24460} \end{pmatrix}$	0.003509317007
(8)	$\begin{pmatrix} 1.00000 \\ -2.00000 \\ -2.00000 \\ 7.00000 \end{pmatrix}$	$\begin{pmatrix} 1 \\ -2 \\ -2 \\ 7 \end{pmatrix}$	0
(9)	$\begin{pmatrix} 0.00707 \\ 0.00843 \\ 0.00848 \end{pmatrix}$	$\begin{pmatrix} \frac{1435}{203107} \\ \frac{1712}{203107} \\ \frac{1723}{203107} \end{pmatrix}$	0.03881495963
(10)	$\begin{pmatrix} 0.53333 \\ 1.20000 \\ 1.80000 \\ 3.46667 \end{pmatrix}$	$\begin{pmatrix} \frac{8}{15} \\ \frac{6}{5} \\ \frac{9}{5} \\ \frac{52}{15} \end{pmatrix}$	0.0001802844952

## 2.2 Méthode de Cholesky

La méthode de Cholesky est une décomposition  $LU$  de  $A$ . On décompose la matrice  $A$  en  $R^T \cdot R$  où  $R$  est triangulaire supérieure et  $R^T$  sa transposée.

Cette méthode permet également d'obtenir des solutions exactes en un nombre fini d'itérations. Cependant, elle est plus restrictive que Gauss pour les raisons suivantes :

- $A$  doit être symétrique,
- $A$  doit être diagonale dominante.

### 2.2.1 Programme

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <math.h>
4  #include "matrices.h"
5
6  void cholesky (double ** a, double ** b, int n)
7  {
8      int i, j, k;
9      double **r = (double**) malloc(n*sizeof(double*));
10     for (i=0; i<n; i++)
11     {
12         r[i]=(double*) malloc(n*sizeof(double));
13     }
14
15     /*Calcul de R*/
16     for (k=0; k<n; k++)
17     {
18         for (i=k; i<n; i++)
19         {
20             double somme=0;
21             if(k==i)
22             {
23                 for (j=0; j<k; j++)
24                 {
25                     somme=somme+r[i][j]*r[i][j]; //somme des carrés
26                 }
27                 r[i][k]=sqrt(a[i][k]-somme);
28             }
29             else
30             {
31                 for (j=0; j<k; j++)
32                 {
33                     somme=somme+r[i][j]*r[k][j];
34                 }
35                 r[i][k]=(1.0/r[k][k])*(a[i][k]-somme);
36             }
37         }
38     }
39     printf("Matrice R :\n");
40     afficherMatrice(r,n,n);
41     printf("Matrice Rt :\n");
42     afficherMatrice(transpose(r,n),n,n);
43
44     /*Résolution de Ry=b*/
45     double** y=solveTriangulaireInf(r,b,n);
46
47     /*Résolution de tRx=y*/
48     double** x=solveTriangulaireSup(transpose(r,n),y,n);
49
50     /*Affichage de x*/
51     printf("\nVecteur résultat :\n");
52     afficherMatrice(x,n,1);
53
54     // libération mémoire
55     for (i=0; i<n; i++)
56     {
```

```
57 |     free(r[i]);  
58 | }  
59 | free(r);  
60 | }
```

FIGURE 2.2 – Code : cholesky.c

### 2.2.2 Résultats des tests

Système	Résultat obtenu	Résultat théorique	Ecart
(1)	$\begin{pmatrix} 0.63415 \\ -0.26829 \\ 0.17073 \\ -0.58537 \end{pmatrix}$	$\begin{pmatrix} \frac{26}{41} \\ -\frac{11}{41} \\ \frac{7}{41} \\ -\frac{24}{41} \end{pmatrix}$	0.0008213058895
(2)	$\begin{pmatrix} 0.06132 \\ 0.24599 \\ -0.00287 \\ 0.02177 \end{pmatrix}$	$\begin{pmatrix} \frac{3067}{50015} \\ \frac{7382}{30009} \\ -\frac{431}{150045} \\ \frac{1089}{50015} \end{pmatrix}$	0.02643817358
(3)	$\begin{pmatrix} -0.04885 \\ 0.10345 \\ 0.04458 \\ 0.09623 \end{pmatrix}$	$\begin{pmatrix} -\frac{1181}{20928} \\ \frac{563}{6976} \\ \frac{121}{1308} \\ \frac{95}{872} \end{pmatrix}$	26.27445408
(4)	$\begin{pmatrix} -0.36364 \\ 2.18182 \\ 1.09091 \end{pmatrix}$	$\begin{pmatrix} -\frac{4}{11} \\ \frac{24}{11} \\ \frac{12}{11} \end{pmatrix}$	0.0003888866667
(5)	$\begin{pmatrix} 1.00000 \\ -0.81145 \\ 1.27048 \\ -0.46016 \\ 1.56790 \\ 0.05780 \\ 2.05345 \\ 0.88005 \\ 1.86675 \\ 2.06675 \end{pmatrix}$	$\begin{pmatrix} 1 \\ -\frac{241}{297} \\ \frac{1132}{891} \\ -\frac{410}{891} \\ \frac{127}{81} \\ \frac{103}{1782} \\ \frac{4879}{2376} \\ \frac{697}{792} \\ \frac{22177}{11880} \\ \frac{24553}{11880} \end{pmatrix}$	0.0001766790236

Système	Résultat obtenu	Résultat théorique	Ecart
(6)	$\begin{pmatrix} -1.64286 \\ 1.57143 \\ 1.35714 \end{pmatrix}$	$\begin{pmatrix} -\frac{23}{14} \\ \frac{11}{7} \\ \frac{19}{14} \end{pmatrix}$	0.0001584521670
(7)	$\begin{pmatrix} 0.26370 \\ 0.17294 \\ 0.07150 \end{pmatrix}$	$\begin{pmatrix} \frac{645}{2446} \\ \frac{423}{2446} \\ \frac{1749}{24460} \end{pmatrix}$	0.003509317007
(8)	$\begin{pmatrix} 1.00000 \\ -2.00000 \\ -2.00000 \\ 7.00000 \end{pmatrix}$	$\begin{pmatrix} 1 \\ -2 \\ -2 \\ 7 \end{pmatrix}$	0
(9)	$\begin{pmatrix} 0.00707 \\ 0.00843 \\ 0.00848 \end{pmatrix}$	$\begin{pmatrix} \frac{1435}{203107} \\ \frac{1712}{203107} \\ \frac{1723}{203107} \end{pmatrix}$	0.03881495963
(10)	$\begin{pmatrix} 0.53333 \\ 1.20000 \\ 1.80000 \\ 3.46667 \end{pmatrix}$	$\begin{pmatrix} \frac{8}{15} \\ \frac{6}{5} \\ \frac{9}{5} \\ \frac{52}{15} \end{pmatrix}$	0.0001802844952

## 2.3 Comparaison

## 3 Méthodes itératives

On cherche à décomposer la matrice  $A$  en  $M$  et  $N$  telles que  $A = M - N$ .

Le but est d'obtenir une solution approchée du système avec une certaine précision.

A chaque itération, un nouveau vecteur résultat est calculé en fonction de celui de l'itération précédente, c'est la raison pour laquelle on doit définir un vecteur  $x^{(0)}$  comme point initial.

On peut donc calculer un résidu (i.e. un écart) à chaque itération. Lorsque cet écart devient inférieur à la précision demandée, on peut arrêter l'algorithme.

Ces méthodes s'appliquent à des matrices dont les éléments diagonaux sont non-nuls.

### 3.1 Méthode de Jacobi

Pour cette méthode, on doit avoir  $M = D$  où  $D$  est la diagonale de  $A$ , et  $N = E + F$  où  $-E$  et  $-F$  sont respectivement les matrices composées des éléments du dessous et du dessus de la diagonale de  $A$ .

Les critères d'application de la méthode de Jacobi sont les suivants :

- $A$  doit être définie positive,
- $A$  doit être à diagonale dominante.

#### 3.1.1 Programme

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <math.h>
4  #include "matrices.h"
5
6  void jacobi (double** a, double** b, double** xInit, int n, double prec)
7  {
8      int i, j, cpt;
9      double residu=2*prec;
10     double** xNext= (double**) malloc(n*sizeof(double*));
11     double** ax;
12     double** axb;
13     cpt=0;
14     for (i=0; i<n; i++) //initialisation xNext
15     {
16         xNext[i]= (double*) malloc(sizeof(double));
17     }
18     while (residu>=prec)
19     {
20         for (i=0; i<n; i++)
21         {
22             double somme1=0, somme2=0;
23             for (j=0; j<i; j++) //avant l'élément diagonal
24             {
25                 somme1=somme1+a[i][j]*xInit[j][0];
26             }
27             for (j=i+1; j<n; j++) //après l'élément diagonal
28             {
29                 somme2=somme2+a[i][j]*xInit[j][0];
30             }
31             xNext[i][0]=(1/a[i][i])*(b[i][0]-somme1-somme2);
32         }
33     }
```

```

33     for(i=0; i<n; i++)
34     {
35         xInit[i][0] = xNext[i][0] ; //copie de xNext dans xInit pour la prochaine itération
36     }
37     ax=produitMatriciel(a, xNext, n, n, 1); //ax
38     axb=difference(ax, b, n, 1); //ax - b
39     residu=norme(axb, n); //norme de (ax - b)
40     cpt++;
41
42     //affichage
43     printf("\nVecteur à l'itération %d :\n", cpt);
44     afficherMatrice(xNext, n, 1);
45 }
46 //libération mémoire
47 for (i=0;i<n;i++)
48 {
49     free(ax[i]); free(axb[i]); free(xNext[i]);
50 }
51 free(ax); free(axb); free(xNext);
52 }

```

FIGURE 3.1 – Code : jacobi.c



### 3.1.2 Résultats des tests

Système	Résultat obtenu	Résultat théorique	Ecart	Convergence
(1)	$\begin{pmatrix} 0.63414 \\ -0.26829 \\ 0.17073 \\ -0.58536 \end{pmatrix}$	$\begin{pmatrix} \frac{26}{41} \\ -\frac{11}{41} \\ \frac{7}{41} \\ -\frac{24}{41} \end{pmatrix}$	0.0009999982130	18 itérations
(2)	$\begin{pmatrix} 0.06132 \\ 0.24599 \\ -0.00287 \\ 0.02177 \end{pmatrix}$	$\begin{pmatrix} \frac{3067}{50015} \\ \frac{7382}{30009} \\ -\frac{431}{150045} \\ \frac{1089}{50015} \end{pmatrix}$	0.02643817358	17 itérations
(3)	$\begin{pmatrix} -0.05643 \\ 0.08071 \\ 0.09251 \\ 0.10894 \end{pmatrix}$	$\begin{pmatrix} -\frac{1181}{20928} \\ \frac{563}{6976} \\ \frac{121}{1308} \\ \frac{95}{872} \end{pmatrix}$	0.003934499152	38 itérations
(4)	$\begin{pmatrix} -0.36364 \\ 2.18181 \\ 1.09091 \end{pmatrix}$	$\begin{pmatrix} -\frac{4}{11} \\ \frac{24}{11} \\ \frac{12}{11} \end{pmatrix}$	0.0004861144443	22 itérations
(5)	$\begin{pmatrix} 1.00000 \\ -0.81145 \\ 1.27048 \\ -0.46016 \\ 1.56790 \\ 0.05780 \\ 2.05345 \\ 0.88005 \\ 1.86675 \\ 2.06675 \end{pmatrix}$	$\begin{pmatrix} 1 \\ -\frac{241}{297} \\ \frac{1132}{891} \\ -\frac{410}{891} \\ \frac{127}{81} \\ \frac{103}{1782} \\ \frac{4879}{2376} \\ \frac{697}{792} \\ \frac{22177}{11880} \\ \frac{24553}{11880} \end{pmatrix}$	0.0001766790236	70 itérations

Système	Résultat obtenu	Résultat théorique	Ecart	Convergence
(6)	$\begin{pmatrix} -1.64286 \\ 1.57143 \\ 1.35714 \end{pmatrix}$	$\begin{pmatrix} -\frac{23}{14} \\ \frac{11}{7} \\ \frac{19}{14} \end{pmatrix}$	0.0001584521670	13 itérations
(7)	$\begin{pmatrix} 0.26370 \\ 0.17293 \\ 0.07150 \end{pmatrix}$	$\begin{pmatrix} \frac{645}{2446} \\ \frac{423}{2446} \\ \frac{1749}{24460} \end{pmatrix}$	0.003665329017	17 itérations
(8)	$\begin{pmatrix} 1.00000 \\ -2.00000 \\ -2.00000 \\ 7.00000 \end{pmatrix}$	$\begin{pmatrix} 1 \\ -2 \\ -2 \\ 7 \end{pmatrix}$	0	21 itérations
(9)	$\begin{pmatrix} 0.00707 \\ 0.00843 \\ 0.00848 \end{pmatrix}$	$\begin{pmatrix} \frac{1435}{203107} \\ \frac{1712}{203107} \\ \frac{1723}{203107} \end{pmatrix}$	0.03881495963	54 itérations
(10)	$\begin{pmatrix} 0.53333 \\ 1.20000 \\ 1.80000 \\ 3.46667 \end{pmatrix}$	$\begin{pmatrix} \frac{8}{15} \\ \frac{6}{5} \\ \frac{9}{5} \\ \frac{52}{15} \end{pmatrix}$	0.0001802844952	18 itérations

## 3.2 Méthode de Gauss-Seidel

Pour cette méthode, on doit avoir  $M = D - E$  où  $D$  est la diagonale de  $A$  et  $-E$  la matrice composée des éléments dessous, et  $N = F$  où  $-F$  est la matrice composée des éléments du dessus de la diagonale de  $A$ . Les critères d'application de la méthode de Gauss-Seidel sont les suivants :

- $A$  doit être définie positive,
- $A$  doit être à diagonale dominante,
- $A$  doit être définie symétrique.

Cette méthode est plus restrictive que Jacobi, mais elle converge plus vite pour les matrices correspondant à ces critères d'application.

En effet, lors de la génération d'un vecteur résultat à une itération  $k$ , l'algorithme n'utilisera les valeurs de l'itération  $k - 1$  que dans le cas où il ne connaît pas encore celles de l'itération en cours. C'est-à-dire qu'il utilisera les valeurs de l'itération en cours lorsqu'il cherchera à déterminer les valeurs d'indices inférieurs à l'indice de l'élément diagonal de la ligne itérée.

### 3.2.1 Programme

```
1 | #include <stdio.h>
2 | #include <stdlib.h>
3 | #include <math.h>
4 | #include "matrices.h"
5 |
6 | void gaussseidel (double** a, double** b, double** xInit, int n, double prec)
7 | {
8 |     int i, j, cpt;
9 |     double residu=2*prec;
10 |    double** ax;
11 |    double** axb;
12 |    cpt=0;
13 |    while (residu>=prec)
14 |    {
15 |        for (i=0; i<n; i++)
16 |        {
17 |            double somme1=0, somme2=0;
18 |            for (j=0; j<i; j++)
19 |            {
20 |                somme1=somme1+a[i][j]*xInit[j][0];
21 |            }
22 |            for (j=i+1; j<n; j++)
23 |            {
24 |                somme2=somme2+a[i][j]*xInit[j][0];
25 |            }
26 |            xInit[i][0]=(1/a[i][i])*(b[i][0]-somme1-somme2);
27 |        }
28 |        ax=produitMatriciel(a, xInit, n, n, 1);
29 |        axb=difference(ax, b, n, 1);
30 |        residu=norme(axb, n);
31 |        cpt++;
32 |
33 |        //affichage
34 |        printf("\nVecteur à l'itération %d :\n", cpt);
35 |        afficherMatrice(xInit, n, 1);
36 |    }
37 |    //libération mémoire
38 |    for (i=0; i<n; i++)
39 |    {
40 |        free(ax[i]);
41 |        free(axb[i]);
42 |    }
43 |    free(ax);
44 |    free(axb);
45 | }
```

FIGURE 3.2 – Code : gauss-seidel.c

### 3.2.2 Résultats de tests

Système	Résultat obtenu	Résultat théorique	Ecart	Convergence
(1)	$\begin{pmatrix} 0.63414 \\ -0.26829 \\ 0.17073 \\ -0.58537 \end{pmatrix}$	$\begin{pmatrix} \frac{26}{41} \\ -\frac{11}{41} \\ \frac{7}{41} \\ -\frac{24}{41} \end{pmatrix}$	0.0009270780048	10 itérations
(2)	$\begin{pmatrix} 0.06132 \\ 0.24599 \\ -0.00287 \\ 0.02177 \end{pmatrix}$	$\begin{pmatrix} \frac{3067}{50015} \\ \frac{7382}{30009} \\ -\frac{431}{150045} \\ \frac{1089}{50015} \end{pmatrix}$	0.02643817358	5 itérations
(3)	$\begin{pmatrix} -0.05643 \\ 0.08070 \\ 0.09251 \\ 0.10894 \end{pmatrix}$	$\begin{pmatrix} -\frac{1181}{20928} \\ \frac{563}{6976} \\ \frac{121}{1308} \\ \frac{95}{872} \end{pmatrix}$	0.004105014648	8 itérations
(4)	$\begin{pmatrix} -0.36362 \\ 2.18181 \\ 1.09091 \end{pmatrix}$	$\begin{pmatrix} -\frac{4}{11} \\ \frac{24}{11} \\ \frac{12}{11} \end{pmatrix}$	0.001652774444	11 itérations
(5)	$\begin{pmatrix} 1.00000 \\ -0.81145 \\ 1.27048 \\ -0.46016 \\ 1.56790 \\ 0.05780 \\ 2.05345 \\ 0.88005 \\ 1.86675 \\ 2.06675 \end{pmatrix}$	$\begin{pmatrix} 1 \\ -\frac{241}{297} \\ \frac{1132}{891} \\ -\frac{410}{891} \\ \frac{127}{81} \\ \frac{103}{1782} \\ \frac{4879}{2376} \\ \frac{697}{792} \\ \frac{22177}{11880} \\ \frac{24553}{11880} \end{pmatrix}$	0.0001766790236	10 itérations

Système	Résultat obtenu	Résultat théorique	Ecart	Convergence
(6)	$\begin{pmatrix} -1.64286 \\ 1.57143 \\ 1.35714 \end{pmatrix}$	$\begin{pmatrix} -\frac{23}{14} \\ \frac{11}{7} \\ \frac{19}{14} \end{pmatrix}$	0.0001584521670	7 itérations
(7)	$\begin{pmatrix} 0.26370 \\ 0.17294 \\ 0.07150 \end{pmatrix}$	$\begin{pmatrix} \frac{645}{2446} \\ \frac{423}{2446} \\ \frac{1749}{24460} \end{pmatrix}$	0.003509317007	9 itérations
(8)	$\begin{pmatrix} 1.00000 \\ -2.00000 \\ -2.00000 \\ 7.00000 \end{pmatrix}$	$\begin{pmatrix} 1 \\ -2 \\ -2 \\ 7 \end{pmatrix}$	0	12 itérations
(9)	$\begin{pmatrix} 0.00707 \\ 0.00843 \\ 0.00848 \end{pmatrix}$	$\begin{pmatrix} \frac{1435}{203107} \\ \frac{1712}{203107} \\ \frac{1723}{203107} \end{pmatrix}$	0.03881495963	11 itérations
(10)	$\begin{pmatrix} 0.53334 \\ 1.20000 \\ 1.80000 \\ 3.46667 \end{pmatrix}$	$\begin{pmatrix} \frac{8}{15} \\ \frac{6}{5} \\ \frac{9}{5} \\ \frac{52}{15} \end{pmatrix}$	0.0003365376202	7 itérations

### 3.3 Comparaison

## 4 Conclusion

## 5 Annexe

### 5.1 Fonctions caractéristiques du calcul matriciel

```
1  #ifndef MATRICES__H
2  #define MATRICES__H
3
4  void remplirMatrice(double** matrice, int n, int m);
5  void afficherMatrice(double** matrice, int n, int m);
6  double ** creerRemplirMatrice(int n, int m);
7  double ** solveTriangulaireSup (double** mat, double** vec, int n);
8  double ** solveTriangulaireInf (double** mat, double** vec, int n);
9  double ** transpose(double** mat, int n);
10 double ** produitMatriciel(double** a, double** b, int n, int m, int o);
11 double ** difference (double** a, double** b, int n, int m);
12 double norme (double** x, int n);
13 double det (double** mat, int n);
14
15 #endif
```

FIGURE 5.1 – Code : matrices.h

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <math.h>
4
5  void remplirMatrice(double** matrice, int n, int m)
6  {
7      int i,j;
8      double v; //valeur
9      for(i=0;i<n;i++)
10     {
11         for(j=0;j<m;j++)
12         {
13             printf("Entrez l'element [%d] [%d]: ", i+1,j+1);
14             scanf("%lf",&v);
15             matrice[i][j] = v;
16         }
17     }
18 }
19
20 void afficherMatrice(double** matrice, int n, int m)
21 {
22     int i,j;
23     for(i=0;i<n;i++)
24     {
25         for(j=0;j<m;j++)
26         {
27             printf("%+.5lf\t",matrice[i][j]);
28         }
29         printf("\n");
30     }
31 }
32
33 double ** creerRemplirMatrice(int n, int m)
34 {
35     int i;
```



```

36     int rep;
37
38     //création matrice
39     double **matrice = (double**) malloc (n*sizeof(double*));
40     for(i=0;i<n;i++)
41     {
42         matrice[i] = (double*) malloc (m*sizeof(double));
43     }
44     remplirMatrice(matrice,n,m);
45
46     //affichage
47     printf("Voulez-vous afficher votre matrice ? 0-non 1-oui : ");
48     scanf("%d",&rep);
49     if (rep)
50     {
51         afficherMatrice(matrice,n,m);
52     }
53
54     return matrice;
55 }
56
57 double ** solveTriangulaireSup (double** mat, double** vec, int n)
58 {
59     int i, j;
60
61     //vecteur résultat
62     double **x = (double**) malloc(n*sizeof(double*));
63     for (i=0; i<n; i++)
64     {
65         x[i]=(double*) malloc(sizeof(double));
66     }
67
68     //calculs
69     for (i=n-1; i>=0; i--)
70     {
71         double somme=0;
72         for (j=i+1; j<n; j++)
73         {
74             somme=somme+(mat[i][j]*x[j][0]);
75         }
76         x[i][0]=(1.0/mat[i][i])*(vec[i][0]-somme);
77     }
78     return x;
79 }
80
81 double ** solveTriangulaireInf (double** mat, double** vec, int n)
82 {
83     int i, j;
84
85     //vecteur résultat
86     double **x = (double**) malloc(n*sizeof(double*));
87     for (i=0; i<n; i++)
88     {
89         x[i]=(double*) malloc(sizeof(double));
90     }
91
92     //calculs
93     for (i=0; i<n; i++)
94     {
95         double somme=0;
96         for (j=0; j<i; j++)
97         {
98             somme=somme+(mat[i][j]*x[j][0]);
99         }
100         x[i][0]=(1.0/mat[i][i])*(vec[i][0]-somme);
101     }
102     return x;
103 }
104
105 double ** transpose(double** mat, int n)
106 {

```

```

107     int i, j;
108     double **tr = (double**) malloc(n*sizeof(double*));
109     for (i=0; i<n; i++)
110     {
111         tr[i]=(double*) malloc(n*sizeof(double));
112     }
113
114     //transposition
115     for (i=0; i<n ; i++)
116     {
117         for (j=0 ; j<n ; j++)
118         {
119             tr[i][j]=mat[j][i];
120         }
121     }
122     return tr;
123 }
124
125 double ** produitMatriciel(double** a, double** b, int n, int m, int o)
126 {
127     int i, j, k; //indices de boucles
128
129     //matrice resultat
130     double** c=(double**) malloc(n*sizeof(double*));
131     for (i=0; i<n; i++)
132     {
133         c[i]=(double*) malloc(o*sizeof(double));
134     }
135
136     //Calcul de la matrice resultat c=ab
137     for (i=0; i<n; i++)
138     {
139         for (j=0; j<o; j++)
140         {
141             c[i][j]=0;
142             for (k=0; k<n; k++)
143             {
144                 c[i][j]=c[i][j]+a[i][k]*b[k][j];
145             }
146         }
147     }
148
149     return c;
150 }
151
152 double ** difference (double** a, double** b, int n, int m)
153 {
154     int i, j;
155
156     //matrice resultat
157     double** c= (double**) malloc(n*sizeof(double*));
158     for (i=0; i<n; i++)
159     {
160         c[i]= (double*) malloc(m*sizeof(double));
161     }
162
163     //calcul de c=a-b
164     for (i=0; i<n; i++)
165     {
166         for (j=0; j<m; j++)
167         {
168             c[i][j]=(a[i][j])-(b[i][j]);
169         }
170     }
171     return c;
172 }
173
174 double norme (double** x, int n)
175 {
176     double somme=0; //somme des carrés
177     int i;

```

```

178     for (i=0; i<n; i++)
179     {
180         somme=somme+(x[i][0]*x[i][0]);
181     }
182     return sqrt(somme);
183 }
184
185 double det (double** mat, int n)
186 {
187     int i, j, k;
188     double pivot;
189     double determinant=1;
190
191     //matrice pour calcul
192     double** a= (double**) malloc(n*sizeof(double*));
193     for (i=0; i<n; i++)
194     {
195         a[i]=(double*) malloc(n*sizeof(double));
196     }
197
198     //copie de matrice
199     for (k=0; k<n; k++)
200     {
201         for (i=0; i<n; i++)
202         {
203             a[k][i]=mat[k][i];
204         }
205     }
206
207     //echelonnage
208     for (k=0; k<n-1; k++)
209     {
210         for (i=k+1; i<n; i++)
211         {
212             pivot = a[i][k]/a[k][k];
213             for (j=k+1; j<n; j++)
214             {
215                 a[i][j]=a[i][j]-pivot*a[k][j];
216             }
217         }
218     }
219
220     //calcul determinant
221     for (i=0; i<n; i++)
222     {
223         determinant=determinant*a[i][i];
224     }
225
226     //libération mémoire
227     for (i=0; i<n; i++)
228     {
229         free(a[i]);
230     }
231     free(a);
232
233     return determinant;
234 }

```

FIGURE 5.2 – Code : matrices.c

## 5.2 Génération de matrices aléatoires

```
1 | #ifndef GENERATEUR__H
2 | #define GENERATEUR__H
3 |
4 | void generatorMenu();
5 | void usewithMenu(double **a, int n);
6 | double** creuse70(int n);
7 | double** bord(int n);
8 | double** dingDong(int n);
9 | double** franc(int n);
10 | double** hilbert(int n);
11 | double** kms(int n, double p);
12 | double** lehmer(int n);
13 | double** lotkin(int n);
14 | double** moler(int n);
15 |
16 | #endif
```

FIGURE 5.3 – Code : generateur.h

```
1 | #include <stdio.h>
2 | #include <stdlib.h>
3 | #include <math.h>
4 | #include "matrices.h"
5 | #include "gauss.h"
6 | #include "cholesky.h"
7 | #include "jacobi.h"
8 | #include "gauss-seidel.h"
9 | #include "surrelaxation.h"
10 | #include "generateur.h"
11 |
12 | void generatorMenu()
13 | {
14 |     int choice=1; //choix menu generation
15 |     int use, i; //choix reutilisation matrice, indice boucle
16 |     int n; //dimension matrice
17 |     double p; //precision
18 |     double** generatedMat; //matrice generee
19 |     while (choice != 0)
20 |     {
21 |         printf("\n\nMENU : GENERATION DE MATRICES\n\n");
22 |         printf("Choisir un type de matrice à générer :\n");
23 |         printf("1. Creuse à 70%\n");
24 |         printf("2. A bord\n");
25 |         printf("3. Ding-Dong\n");
26 |         printf("4. Franc\n");
27 |         printf("5. Hilbert\n");
28 |         printf("6. KMS\n");
29 |         printf("7. Lehmer\n");
30 |         printf("8. Lotkin\n");
31 |         printf("9. Moler\n");
32 |         printf("0. Retour au menu principal\n");
33 |         printf("Votre choix : ");
34 |         scanf("%d", &choice);
35 |         if (choice!=0)
36 |         {
37 |             printf("Dimension : ");
38 |             scanf("%d", &n);
39 |             switch (choice)
40 |             {
41 |                 case 1:
42 |                     generatedMat=creuse70(n);
43 |                     break;
44 |                 case 2:
45 |                     generatedMat=bord(n);
46 |                     break;
47 |                 case 3:
48 |                     generatedMat=dingDong(n);
```

```

49     break;
50     case 4:
51         generatedMat=franc(n);
52         break;
53     case 5:
54         generatedMat=hilbert(n);
55         break;
56     case 6:
57         printf("Parametre p : ");
58         scanf("%lf", &p);
59         generatedMat=kms(n,p);
60         break;
61     case 7:
62         generatedMat=lehmer(n);
63         break;
64     case 8:
65         generatedMat=lotkin(n);
66         break;
67     case 9:
68         generatedMat=moler(n);
69         break;
70 }
71 printf("Matrice générée :\n");
72 afficherMatrice(generatedMat, n, n);
73 printf("Utiliser cette matrice pour une résolution ? (1-oui, 0-non) : ");
74 scanf("%d",&use);
75 if (use==1)
76 {
77     usewithMenu(generatedMat, n); //menu resolution
78 }
79 //libération mémoire
80 for (i=0;i<n;i++)
81 {
82     free(generatedMat[i]);
83 }
84 free(generatedMat);
85 }
86 }
87 }
88
89 void usewithMenu(double **a, int n)
90 {
91     int choice=1, i, j; //choix, indice boucle
92     double** b; //vecteur b
93     double** xInit; //jacobi et sur-relaxation
94     double prec; //jacobi et sur-relaxation
95     double omega; //sur-relaxation
96     double** mat=(double**) malloc(n*sizeof(double*)); //matrice copie de a
97     for (i=0; i<n; i++)
98     {
99         mat[i]=(double*) malloc(n*sizeof(double));
100     }
101     while (choice != 0)
102     {
103         printf("\n\nSOUS-MENU : UTILISER LA MATRICE PRECEDEMMENT GENEREE\n\n");
104         printf("Choisir un type de résolution :\n");
105         printf("1. Gauss\n");
106         printf("2. Cholesky\n");
107         printf("3. Jacobi\n");
108         printf("4. Gauss-Seidel\n");
109         printf("5. Surrelaxation\n");
110         printf("0. Ne rien faire, retourner au menu de génération de matrices.\n");
111         printf("Votre choix : ");
112         scanf("%d", &choice);
113         if (choice!=0)
114         {
115             for (i=0; i<n; i++)
116             {
117                 for (j=0; j<n; j++)
118                 {
119                     mat[i][j]=a[i][j]; //copie matrice

```

```

120     }
121 }
122 printf("Vecteur b :\n"); //demande du vecteur b
123 b=creerRemplirMatrice(n,1);
124 switch (choice)
125 {
126     case 1: //gauss
127         gauss(mat,b,n);
128         break;
129     case 2: //cholesky
130         cholesky(mat,b,n);
131         break;
132     case 3: //jacobi
133         printf("Vecteur initial x0 :\n");
134         xInit=creerRemplirMatrice(n,1);
135         printf("Précision : ");
136         scanf("%lf", &prec);
137         printf("\nRésolution par Jacobi...\n");
138         jacobi(mat,b,xInit,n,prec);
139         //libération mémoire
140         for (i=0;i<n;i++)
141         {
142             free(xInit[i]);
143         }
144         free(xInit);
145         break;
146     case 4: //gauss-seidel
147         printf("Vecteur initial x0 :\n");
148         xInit=creerRemplirMatrice(n,1);
149         printf("Précision : ");
150         scanf("%lf", &prec);
151         printf("\nRésolution par Gauss-Seidel...\n");
152         gaussseidel(mat,b,xInit,n,prec);
153         //libération mémoire
154         for (i=0;i<n;i++)
155         {
156             free(xInit[i]);
157         }
158         free(xInit);
159         break;
160     case 5: //sur-relaxation
161         printf("Vecteur initial x0 :\n");
162         xInit=creerRemplirMatrice(n,1);
163         printf("Précision : ");
164         scanf("%lf", &prec);
165         printf("Entrer omega : ");
166         scanf("%lf", &omega);
167         printf("\nRésolution par Surrelaxation...\n");
168         surrelaxation(mat,b,xInit,n,prec,omega);
169         //libération mémoire
170         for (i=0;i<n;i++)
171         {
172             free(xInit[i]);
173         }
174         free(xInit);
175         break;
176 }
177 //libération mémoire
178 for (i=0;i<n;i++)
179 {
180     free(b[i]);
181 }
182 free(b);
183 }
184 }
185 //libération mémoire copie matrice -> choice=0
186 for (i=0; i<n; i++)
187 {
188     free(mat[i]);
189 }
190 free(mat);

```

```

191 }
192
193 double** creuse70(int n)
194 {
195     int i, j;
196     //initialisation matrice
197     double** a= (double**) malloc(n*sizeof(double*));
198     for (i=0; i<n; i++)
199     {
200         a[i]=(double*) malloc(n*sizeof(double));
201     }
202     //generation matrice
203     for (i=0; i<n; i++)
204     {
205         for (j=0; j<n; j++)
206         {
207             int rand_value = rand()%100;
208             if (rand_value<=70)
209             {
210                 a[i][j]=0;
211             }
212             else
213             {
214                 double a_value = (double) (rand()%10);
215                 a[i][j]=a_value;
216             }
217         }
218     }
219     return a;
220 }
221
222 double** bord(int n)
223 {
224     int i, j;
225     //initialisation matrice
226     double** a= (double**) malloc(n*sizeof(double*));
227     for (i=0; i<n; i++)
228     {
229         a[i]=(double*) malloc(n*sizeof(double));
230     }
231     //generation matrice
232     for (i=0; i<n; i++)
233     {
234         for (j=i; j<n; j++)
235         {
236             if(i==j)
237             {
238                 a[i][j]=1;
239             }
240             else if (i==0)
241             {
242                 a[i][j]= pow(2, (double) (-j));
243                 a[j][i]= pow(2, (double) (-j));
244             }
245             else
246             {
247                 a[i][j]=0;
248                 a[j][i]=0;
249             }
250         }
251     }
252     return a;
253 }
254
255 double** dingDong(int n)
256 {
257     int i, j;
258     //initialisation matrice
259     double** a= (double**) malloc(n*sizeof(double*));
260     for (i=0; i<n; i++)
261     {

```

```

262     a[i]=(double*) malloc(n*sizeof(double));
263 }
264 //generation matrice
265 for (i=0; i<n; i++)
266 {
267     for (j=0; j<n; j++)
268     {
269         a[i][j]=1/(2*(n-(i+1)-(j+1)+1.5));
270     }
271 }
272 return a;
273 }
274
275 double** franc(int n)
276 {
277     int i, j;
278     //initialisation matrice
279     double** a= (double**) malloc(n*sizeof(double*));
280     for (i=0; i<n; i++)
281     {
282         a[i]=(double*) malloc(n*sizeof(double));
283     }
284     //generation matrice
285     for (i=0; i<n; i++)
286     {
287         for (j=0; j<n; j++)
288         {
289             if (i>=j+2)
290             {
291                 a[i][j]=0;
292             }
293             else if (i<j)
294             {
295                 a[i][j]=(double) i+1;
296             }
297             else
298             {
299                 a[i][j]=(double) j+1;
300             }
301         }
302     }
303     return a;
304 }
305
306 double** hilbert(int n)
307 {
308     int i, j;
309     //initialisation matrice
310     double** a= (double**) malloc(n*sizeof(double*));
311     for (i=0; i<n; i++)
312     {
313         a[i]=(double*) malloc(n*sizeof(double));
314     }
315     //generation matrice
316     for (i=0; i<n; i++)
317     {
318         for (j=0; j<n; j++)
319         {
320             a[i][j]=1/(double) (i+j+1);
321         }
322     }
323     return a;
324 }
325
326 double** kms(int n, double p)
327 {
328     int i, j;
329     //initialisation matrice
330     double** a= (double**) malloc(n*sizeof(double*));
331     for (i=0; i<n; i++)
332     {

```



```

333     a[i]=(double*) malloc(n*sizeof(double));
334 }
335 //generation matrice
336 for (i=0; i<n; i++)
337 {
338     for (j=0; j<n; j++)
339     {
340         a[i][j]=pow(p, fabs((double) (i-j)));
341     }
342 }
343 return a;
344 }
345
346 double** lehmer(int n)
347 {
348     int i, j;
349     //initialisation matrice
350     double** a= (double**) malloc(n*sizeof(double*));
351     for (i=0; i<n; i++)
352     {
353         a[i]=(double*) malloc(n*sizeof(double));
354     }
355     //generation matrice
356     for (i=0; i<n; i++)
357     {
358         for (j=0; j<n; j++)
359         {
360             if (i<=j)
361             {
362                 a[i][j]=((double)(i+1))/((double)(j+1));
363             }
364             else
365             {
366                 a[i][j]=((double)(j+1))/((double)(i+1));
367             }
368         }
369     }
370     return a;
371 }
372
373 double** lotkin(int n)
374 {
375     int i, j;
376     //initialisation matrice
377     double** a= (double**) malloc(n*sizeof(double*));
378     for (i=0; i<n; i++)
379     {
380         a[i]=(double*) malloc(n*sizeof(double));
381     }
382     //generation matrice
383     for (i=0; i<n; i++)
384     {
385         for (j=0; j<n; j++)
386         {
387             if (i==0)
388             {
389                 a[i][j]=1;
390             }
391             else
392             {
393                 a[i][j]=1/((double) (i+j+1));
394             }
395         }
396     }
397     return a;
398 }
399
400 double** moler(int n)
401 {
402     int i, j;
403     //initialisation matrice

```

```

404 double** a= (double**) malloc(n*sizeof(double*));
405 for (i=0; i<n; i++)
406 {
407     a[i]=(double*) malloc(n*sizeof(double));
408 }
409 //generation matrice
410 for (i=0; i<n; i++)
411 {
412     for (j=0; j<n; j++)
413     {
414         if (i==j)
415         {
416             a[i][j]=i+1;
417         }
418         else if (i<j)
419         {
420             a[i][j]=(double) i-1;
421         }
422         else
423         {
424             a[i][j]=(double) j-1;
425         }
426     }
427 }
428 return a;
429 }

```

FIGURE 5.4 – Code : generateur.c

## 5.3 Méthode de Sur-relaxation

### 5.3.1 Programme

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <math.h>
4  #include "matrices.h"
5  #include "gauss-seidel.h"
6
7  void surrelaxation (double** a, double** b, double** xInit, int n, double prec, double ohm)
8  {
9      if (ohm==1)
10     {
11         gaussseidel(a,b,xInit,n,prec);
12     }
13     else
14     {
15         int i, j, cpt;
16         double residu=2*prec;
17         double** ax;
18         double** axb;
19         cpt=0;
20         while (residu>=prec)
21         {
22             for (i=0; i<n; i++)
23             {
24                 double somme1=0, somme2=0;
25                 for (j=0; j<i; j++)
26                 {
27                     somme1=somme1+a[i][j]*xInit[j][0];
28                 }
29                 for (j=i+1; j<n; j++)
30                 {
31                     somme2=somme2+a[i][j]*xInit[j][0];
32                 }
33                 xInit[i][0]=(1-ohm)*xInit[i][0]+(ohm/a[i][i])*(b[i][0]-somme1-somme2);
34             }
35             ax=produitMatriciel(a, xInit, n, n, 1);
36             axb=difference(ax, b, n, 1);
37             residu=norme(axb, n);
38             cpt++;
39
40             //affichage
41             printf("\nVecteur à l'itération %d :\n", cpt);
42             afficherMatrice(xInit, n, 1);
43         }
44         //libération mémoire
45         for (i=0;i<n;i++)
46         {
47             free(ax[i]);
48             free(axb[i]);
49         }
50         free(ax);
51         free(axb);
52     }
53 }
```

FIGURE 5.5 – Code : surrelaxation.c