Algorithmes numériques – Rapport

Résolution de systèmes linéaires

Axel Delsol, Pierre-Loup Pissavy

Novembre 2013

Table des matières

1 Préambule **2**
- 1.1 Structure du programme . . . 2
- 1.2 Compilation . . . 3
- 1.3 Systèmes testés . . . 4

2 Méthodes directes **5**
- 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

3 Méthodes itératives **14**
- 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

4 Conclusion **22**

5 Annexe **23**
- 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 – Apercu : 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 – Apercu : 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 :

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

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

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

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

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

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

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

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

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

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

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 quel $N \cdot X = Y$

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

```c
#include <stdio.h>
#include <stdlib.h>
#include "matrices.h"

void gauss (double** a, double** b, int n)
{
  //initialisation
  int i, j, k;
  double pivot;

  //gauss
  for (k=0; k<n-1; k++)
  {
    for (i=k+1; i<n; i++)
    {
      pivot = a[i][k]/a[k][k];
      for (j=k+1; j<n; j++)
      {
        a[i][j]=a[i][j]-pivot*a[k][j];
      }
      b[i][0]=b[i][0]-pivot*b[k][0];
    }
    printf("\nMatrice :\n");
    afficherMatrice(a,n,n);
    printf("Vecteur :\n");
    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

```c
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include "matrices.h"

void cholesky (double ** a, double ** b, int n)
{
  int i, j, k;
  double **r = (double**) malloc(n*sizeof(double*));
  for (i=0; i<n; i++)
  {
    r[i]=(double*) malloc(n*sizeof(double));
  }

  /*Calcul de R*/
  for (k=0; k<n; k++)
  {
    for (i=k; i<n; i++)
    {
      double somme=0;
      if(k==i)
      {
        for (j=0; j<k; j++)
        {
          somme=somme+r[i][j]*r[i][j]; //somme des carrés
        }
        r[i][k]=sqrt(a[i][k]-somme);
      }
      else
      {
        for (j=0; j<k; j++)
        {
          somme=somme+r[i][j]*r[k][j];
        }
        r[i][k]=(1.0/r[k][k])*(a[i][k]-somme);
      }
    }
  }
  printf("Matrice R :\n");
  afficherMatrice(r,n,n);
  printf("Matrice Rt :\n");
  afficherMatrice(transpose(r,n),n,n);

  /*Résolution de Ry=b*/
  double** y=solveTriangulaireInf(r,b,n);

  /*Résolution de tRx=y*/
  double** x=solveTriangulaireSup(transpose(r,n),y,n);

  /*Affichage de x*/
  printf("\nVecteur résultat :\n");
  afficherMatrice(x,n,1);

  // libération mémoire
  for (i=0; i<n; i++)
  {
```

```
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

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include "matrices.h"

void jacobi (double** a, double** b, double** xInit, int n, double prec)
{
  int i, j, cpt;
  double residu=2*prec;
  double** xNext= (double**) malloc(n*sizeof(double*));
  double** ax;
  double** axb;
  cpt=0;
  for (i=0; i<n; i++) //initialisation xNext
  {
    xNext[i]= (double*) malloc(sizeof(double));
  }
  while (residu>=prec)
  {
    for (i=0; i<n; i++)
    {
      double somme1=0, somme2=0;
      for (j=0; j<i; j++) //avant l'élément diagonal
      {
        somme1=somme1+a[i][j]*xInit[j][0];
      }
      for (j=i+1; j<n; j++) //après l'élément diagonal
      {
        somme2=somme2+a[i][j]*xInit[j][0];
      }
      xNext[i][0]=(1/a[i][i])*(b[i][0]-somme1-somme2);
    }
```

```
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 àl'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

```c
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include "matrices.h"

void gaussseidel (double** a, double** b, double** xInit, int n, double prec)
{
  int i, j, cpt;
  double residu=2*prec;
  double** ax;
  double** axb;
  cpt=0;
  while (residu>=prec)
  {
    for (i=0; i<n; i++)
    {
      double somme1=0, somme2=0;
      for (j=0; j<i; j++)
      {
        somme1=somme1+a[i][j]*xInit[j][0];
      }
      for (j=i+1; j<n; j++)
      {
        somme2=somme2+a[i][j]*xInit[j][0];
      }
      xInit[i][0]=(1/a[i][i])*(b[i][0]-somme1-somme2);
    }
    ax=produitMatriciel(a, xInit, n, n, 1);
    axb=difference(ax, b, n, 1);
    residu=norme(axb, n);
    cpt++;

    //affichage
    printf("\nVecteur àl'itération %d :\n", cpt);
    afficherMatrice(xInit, n, 1);
  }
  //libération mémoire
  for (i=0;i<n;i++)
  {
    free(ax[i]);
    free(axb[i]);
  }
  free(ax);
  free(axb);
}
```

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

```
#ifndef MATRICES__H
#define MATRICES__H

void remplirMatrice(double** matrice, int n, int m);
void afficherMatrice(double** matrice, int n, int m);
double ** creerRemplirMatrice(int n, int m);
double ** solveTriangulaireSup (double** mat, double** vec, int n);
double ** solveTriangulaireInf (double** mat, double** vec, int n);
double ** transpose(double** mat, int n);
double ** produitMatriciel(double** a, double** b, int n, int m, int o);
double ** difference (double** a, double** b, int n, int m);
double norme (double** x, int n);
double det (double** mat, int n);

#endif
```

FIGURE 5.1 – Code : matrices.h

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>

void remplirMatrice(double** matrice, int n, int m)
{
  int i,j;
  double v; //valeur
  for(i=0;i<n;i++)
  {
    for(j=0;j<m;j++)
    {
      printf("Entrez l'element [%d] [%d]: ", i+1,j+1);
      scanf("%lf",&v);
      matrice[i][j] = v;
    }
  }
}

void afficherMatrice(double** matrice, int n, int m)
{
  int i,j;
  for(i=0;i<n;i++)
  {
    for(j=0;j<m;j++)
    {
      printf("%+.5lf\t",matrice[i][j]);
    }
    printf("\n");
  }
}

double ** creerRemplirMatrice(int n, int m)
{
  int i;
```

```c
  int rep;

  //création matrice
  double **matrice = (double**) malloc (n*sizeof(double*));
  for(i=0;i<n;i++)
  {
    matrice[i] = (double*) malloc (m*sizeof(double));
  }
  remplirMatrice(matrice,n,m);

  //affichage
  printf("Voulez-vous afficher votre matrice ? 0-non 1-oui : ");
  scanf("%d",&rep);
  if (rep)
  {
    afficherMatrice(matrice,n,m);
  }

  return matrice;
}

double ** solveTriangulaireSup (double** mat, double** vec, int n)
{
  int i, j;

  //vecteur résultat
  double **x = (double**) malloc(n*sizeof(double*));
  for (i=0; i<n; i++)
  {
    x[i]=(double*) malloc(sizeof(double));
  }

  //calculs
  for (i=n-1; i>=0; i--)
  {
    double somme=0;
    for (j=i+1; j<n; j++)
    {
      somme=somme+(mat[i][j]*x[j][0]);
    }
    x[i][0]=(1.0/mat[i][i])*(vec[i][0]-somme);
  }
  return x;
}

double ** solveTriangulaireInf (double** mat, double** vec, int n)
{
  int i, j;

  //vecteur résultat
  double **x = (double**) malloc(n*sizeof(double*));
  for (i=0; i<n; i++)
  {
    x[i]=(double*) malloc(sizeof(double));
  }

  //calculs
  for (i=0; i<n; i++)
  {
    double somme=0;
    for (j=0; j<i; j++)
    {
      somme=somme+(mat[i][j]*x[j][0]);
    }
    x[i][0]=(1.0/mat[i][i])*(vec[i][0]-somme);
  }
  return x;
}

double ** transpose(double** mat, int n)
{
```

```
  int i, j;
  double **tr = (double**) malloc(n*sizeof(double*));
  for (i=0; i<n; i++)
  {
    tr[i]=(double*) malloc(n*sizeof(double));
  }

  //transposition
  for (i=0; i<n ; i++)
  {
    for (j=0 ; j<n ; j++)
    {
      tr[i][j]=mat[j][i];
    }
  }
  return tr;
}

double ** produitMatriciel(double** a, double** b, int n, int m, int o)
{
  int i, j, k; //indices de boucles

  //matrice resultat
  double** c=(double**) malloc(n*sizeof(double*));
  for (i=0; i<n; i++)
  {
    c[i]=(double*) malloc(o*sizeof(double));
  }

  //Calcul de la matrice résultat c=ab
  for (i=0; i<n; i++)
  {
    for (j=0; j<o; j++)
    {
      c[i][j]=0;
      for (k=0; k<n; k++)
      {
        c[i][j]=c[i][j]+a[i][k]*b[k][j];
      }
    }
  }

  return c;
}

double ** difference (double** a, double** b, int n, int m)
{
  int i, j;

  //matrice resultat
  double** c= (double**) malloc(n*sizeof(double*));
  for (i=0; i<n; i++)
  {
    c[i]= (double*) malloc(m*sizeof(double));
  }

  //calcul de c=a-b
  for (i=0; i<n; i++)
  {
    for (j=0; j<m; j++)
    {
      c[i][j]=(a[i][j])-(b[i][j]);
    }
  }
  return c;
}

double norme (double** x, int n)
{
  double somme=0; //somme des carrés
  int i;
```

```c
  for (i=0; i<n; i++)
  {
    somme=somme+(x[i][0]*x[i][0]);
  }
  return sqrt(somme);
}

double det (double** mat, int n)
{
  int i, j, k;
  double pivot;
  double determinant=1;

  //matrice pour calcul
  double** a= (double**) malloc(n*sizeof(double*));
  for (i=0; i<n; i++)
  {
    a[i]=(double*) malloc(n*sizeof(double));
  }

  //copie de matrice
  for (k=0; k<n; k++)
  {
    for (i=0; i<n; i++)
    {
      a[k][i]=mat[k][i];
    }
  }

  //echelonnage
  for (k=0; k<n-1; k++)
  {
    for (i=k+1; i<n; i++)
    {
      pivot = a[i][k]/a[k][k];
      for (j=k+1; j<n; j++)
      {
        a[i][j]=a[i][j]-pivot*a[k][j];
      }
    }
  }

  //calcul determinant
  for (i=0; i<n; i++)
  {
    determinant=determinant*a[i][i];
  }

  //libération mémoire
  for (i=0;i<n;i++)
  {
    free(a[i]);
  }
  free(a);

  return determinant;
}
```

FIGURE 5.2 – Code : matrices.c

5.2 Génération de matrices aléatoires

```h
#ifndef GENERATEUR__H
#define GENERATEUR__H

void generatorMenu();
void usewithMenu(double **a, int n);
double** creuse70(int n);
double** bord(int n);
double** dingDong(int n);
double** franc(int n);
double** hilbert(int n);
double** kms(int n, double p);
double** lehmer(int n);
double** lotkin(int n);
double** moler(int n);

#endif
```

FIGURE 5.3 – Code : generateur.h

```c
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include "matrices.h"
#include "gauss.h"
#include "cholesky.h"
#include "jacobi.h"
#include "gauss-seidel.h"
#include "surrelaxation.h"
#include "generateur.h"

void generatorMenu()
{
  int choice=1; //choix menu generation
  int use, i; //choix reutilisation matrice, indice boucle
  int n; //dimension matrice
  double p; //precision
  double** generatedMat; //matrice generee
  while (choice != 0)
  {
    printf("\n\nMENU : GENERATION DE MATRICES\n\n");
    printf("Choisir un type de matrice àgénérer :\n");
    printf("1. Creuse à70%%\n");
    printf("2. A bord\n");
    printf("3. Ding-Dong\n");
    printf("4. Franc\n");
    printf("5. Hilbert\n");
    printf("6. KMS\n");
    printf("7. Lehmer\n");
    printf("8. Lotkin\n");
    printf("9. Moler\n");
    printf("0. Retour au menu principal\n");
    printf("Votre choix : ");
    scanf("%d", &choice);
    if (choice!=0)
    {
      printf("Dimension : ");
      scanf("%d", &n);
      switch (choice)
      {
        case 1:
          generatedMat=creuse70(n);
          break;
        case 2:
          generatedMat=bord(n);
          break;
        case 3:
          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
```

```
        }
      }
      printf("Vecteur b :\n"); //demande du vecteur b
      b=creerRemplirMatrice(n,1);
      switch (choice)
      {
        case 1: //gauss
            gauss(mat,b,n);
          break;
        case 2: //cholesky
            cholesky(mat,b,n);
          break;
        case 3: //jacobi
            printf("Vecteur initial x0 :\n");
            xInit=creerRemplirMatrice(n,1);
            printf("Précision : ");
            scanf("%lf", &prec);
            printf("\nRésolution par Jacobi...\n");
            jacobi(mat,b,xInit,n,prec);
            //libération mémoire
            for (i=0;i<n;i++)
            {
              free(xInit[i]);
            }
            free(xInit);
          break;
        case 4: //gauss-seidel
            printf("Vecteur initial x0 :\n");
            xInit=creerRemplirMatrice(n,1);
            printf("Précision : ");
            scanf("%lf", &prec);
            printf("\nRésolution par Gauss-Seidel...\n");
            gaussseidel(mat,b,xInit,n,prec);
            //libération mémoire
            for (i=0;i<n;i++)
            {
              free(xInit[i]);
            }
            free(xInit);
          break;
        case 5: //sur-relaxation
            printf("Vecteur initial x0 :\n");
            xInit=creerRemplirMatrice(n,1);
            printf("Précision : ");
            scanf("%lf", &prec);
            printf("Entrer omega : ");
            scanf("%lf", &omega);
            printf("\nRésolution par Surrelaxation...\n");
            surrelaxation(mat,b,xInit,n,prec,omega);
            //libération mémoire
            for (i=0;i<n;i++)
            {
              free(xInit[i]);
            }
            free(xInit);
          break;
      }
      //libération mémoire
      for (i=0;i<n;i++)
      {
        free(b[i]);
      }
      free(b);
    }
  }
  //libération mémoire copie matrice -> choice=0
  for (i=0; i<n; i++)
  {
    free(mat[i]);
  }
  free(mat);
```

```
}

double** creuse70(int n)
{
  int i, j;
  //initialisation matrice
  double** a= (double**) malloc(n*sizeof(double*));
  for (i=0; i<n; i++)
  {
    a[i]=(double*) malloc(n*sizeof(double));
  }
  //generation matrice
  for (i=0; i<n; i++)
  {
    for (j=0; j<n; j++)
    {
      int rand_value = rand()%100;
      if (rand_value<=70)
      {
        a[i][j]=0;
      }
      else
      {
        double a_value = (double) (rand()%10);
        a[i][j]=a_value;
      }
    }
  }
  return a;
}

double** bord(int n)
{
  int i, j;
  //initialisation matrice
  double** a= (double**) malloc(n*sizeof(double*));
  for (i=0; i<n; i++)
  {
    a[i]=(double*) malloc(n*sizeof(double));
  }
  //generation matrice
  for (i=0; i<n; i++)
  {
    for (j=i; j<n; j++)
    {
      if(i==j)
      {
        a[i][j]=1;
      }
      else if (i==0)
      {
        a[i][j]= pow(2, (double) (-j));
        a[j][i]= pow(2, (double) (-j));
      }
      else
      {
        a[i][j]=0;
        a[j][i]=0;
      }
    }
  }
  return a;
}

double** dingDong(int n)
{
  int i, j;
  //initialisation matrice
  double** a= (double**) malloc(n*sizeof(double*));
  for (i=0; i<n; i++)
  {
```

```
    a[i]=(double*) malloc(n*sizeof(double));
  }
  //generation matrice
  for (i=0; i<n; i++)
  {
    for (j=0; j<n; j++)
    {
      a[i][j]=1/(2*(n-(i+1)-(j+1)+1.5));
    }
  }
  return a;
}

double** franc(int n)
{
  int i, j;
  //initialisation matrice
  double** a= (double**) malloc(n*sizeof(double*));
  for (i=0; i<n; i++)
  {
    a[i]=(double*) malloc(n*sizeof(double));
  }
  //generation matrice
  for (i=0; i<n; i++)
  {
    for (j=0; j<n; j++)
    {
      if (i>=j+2)
      {
        a[i][j]=0;
      }
      else if (i<j)
      {
        a[i][j]=(double) i+1;
      }
      else
      {
        a[i][j]=(double) j+1;
      }
    }
  }
  return a;
}

double** hilbert(int n)
{
  int i, j;
  //initialisation matrice
  double** a= (double**) malloc(n*sizeof(double*));
  for (i=0; i<n; i++)
  {
    a[i]=(double*) malloc(n*sizeof(double));
  }
  //generation matrice
  for (i=0; i<n; i++)
  {
    for (j=0; j<n; j++)
    {
      a[i][j]=1/(double)(i+j+1);
    }
  }
  return a;
}

double** kms(int n, double p)
{
  int i, j;
  //initialisation matrice
  double** a= (double**) malloc(n*sizeof(double*));
  for (i=0; i<n; i++)
  {
```

```
    a[i]=(double*) malloc(n*sizeof(double));
  }
  //generation matrice
  for (i=0; i<n; i++)
  {
    for (j=0; j<n; j++)
    {
      a[i][j]=pow(p, fabs((double) (i-j)));
    }
  }
  return a;
}

double** lehmer(int n)
{
  int i, j;
  //initialisation matrice
  double** a= (double**) malloc(n*sizeof(double*));
  for (i=0; i<n; i++)
  {
    a[i]=(double*) malloc(n*sizeof(double));
  }
  //generation matrice
  for (i=0; i<n; i++)
  {
    for (j=0; j<n; j++)
    {
      if (i<=j)
      {
        a[i][j]=((double)(i+1))/((double)(j+1));
      }
      else
      {
        a[i][j]=((double)(j+1))/((double)(i+1));
      }
    }
  }
  return a;
}

double** lotkin(int n)
{
  int i, j;
  //initialisation matrice
  double** a= (double**) malloc(n*sizeof(double*));
  for (i=0; i<n; i++)
  {
    a[i]=(double*) malloc(n*sizeof(double));
  }
  //generation matrice
  for (i=0; i<n; i++)
  {
    for (j=0; j<n; j++)
    {
      if (i==0)
      {
        a[i][j]=1;
      }
      else
      {
        a[i][j]=1/((double) (i+j+1));
      }
    }
  }
  return a;
}

double** moler(int n)
{
  int i, j;
  //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

```c
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include "matrices.h"
#include "gauss-seidel.h"

void surrelaxation (double** a, double** b, double** xInit, int n, double prec, double ohm)
{
  if (ohm==1)
  {
    gaussseidel(a,b,xInit,n,prec);
  }
  else
  {
    int i, j, cpt;
    double residu=2*prec;
    double** ax;
    double** axb;
    cpt=0;
    while (residu>=prec)
    {
      for (i=0; i<n; i++)
      {
        double somme1=0, somme2=0;
        for (j=0; j<i; j++)
        {
          somme1=somme1+a[i][j]*xInit[j][0];
        }
        for (j=i+1; j<n; j++)
        {
          somme2=somme2+a[i][j]*xInit[j][0];
        }
        xInit[i][0]=(1-ohm)*xInit[i][0]+(ohm/a[i][i])*(b[i][0]-somme1-somme2);
      }
      ax=produitMatriciel(a, xInit, n, n, 1);
      axb=difference(ax, b, n, 1);
      residu=norme(axb, n);
      cpt++;

      //affichage
      printf("\nVecteur àl'itération %d :\n", cpt);
      afficherMatrice(xInit, n, 1);
    }
    //libération mémoire
    for (i=0;i<n;i++)
    {
      free(ax[i]);
      free(axb[i]);
    }
    free(ax);
    free(axb);
  }
}
```

FIGURE 5.5 – Code : surrelaxation.c