Figure 1: logo uca

TP

# Algorithmique Numérique

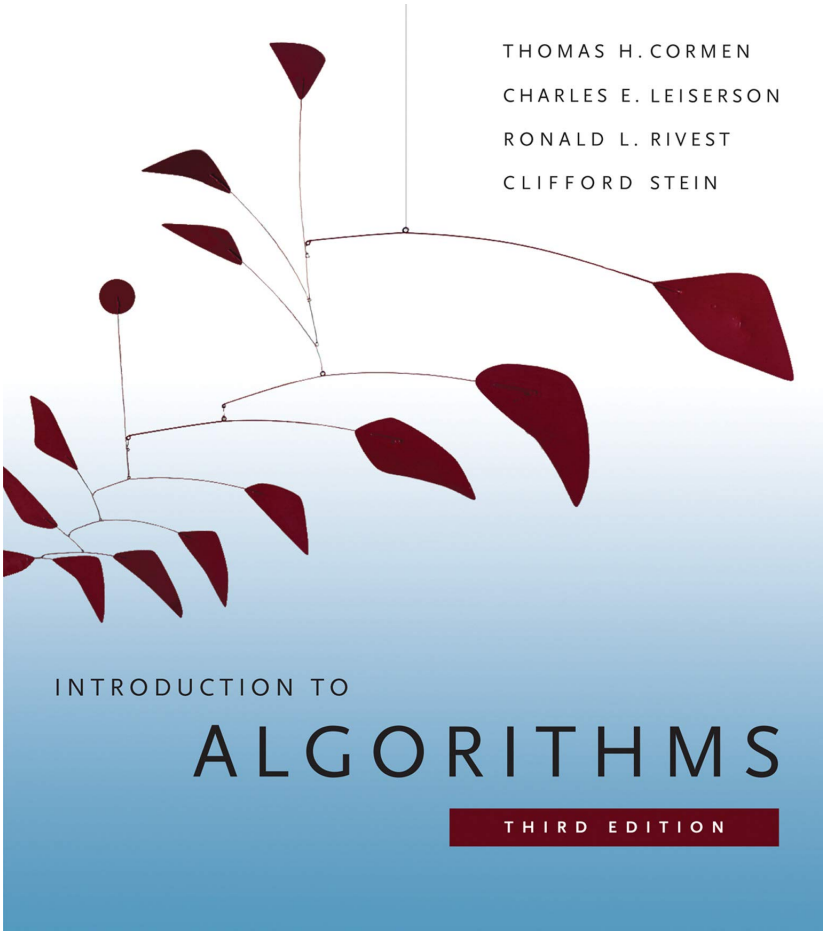


Figure 2: Cormen

SABIER Corentin et CHASSAGNOL Rémi
année 2020-2021

*Professeur:* CHORFI Amina

24 octobre 2020

# Table des matières

# 1 Algorithmiques directes

## 1.1 Gauss

### 1.1.1 Principe

Le principe de cette méthode est de manipuler une matrice **A** de coefficients du système linéaire, de sorte à ce qu'elle devienne triangulaire supérieure. Les coefficients $a_{ii} = 1$ tels que $i \in [0, n[$ . Ensuite par substitution des inconnues, on obtient la matrice solution facilement grâce au système échelonné.

$$A \cdot X = B \text{ où } A = \begin{pmatrix} a_{11} & . & . & . & a_{1n} \\ . & . & . & . & . \\ . & . & . & . & . \\ . & . & . & . & . \\ a_{n1} & . & . & . & a_{nn} \end{pmatrix} \longrightarrow \begin{pmatrix} 1 & a_{12} & . & . & a_{1n} \\ 0 & 1 & . & . & . \\ . & . & 1 & . & . \\ . & . & . & 1 & a_{n-1n} \\ 0 & . & . & 0 & 1 \end{pmatrix}$$

### 1.1.2 L'Algorithme du pivot de Gauss

```
1  k := 0;
2  Pour j allant de 1 a p :
3      Si il existe i ∈ [k+1, n] tel que a_ij ≠ 0 alors :
4
5              k := k + 1;
6              q := choix(i ∈ [k, n]) tel que a_ij ≠ 0);
7              L_q ← (1/a_qj) L_q;
8
9              Si k ≠ q alors :
10                 L_k ↔ L_q;
11
12             Pour i allant de k+1 a n;
13                 L_i ← L_i − a_ij L_k;
14
```

Dans l'ordre l'algorithme fait :

- Ligne 1 : indice de ligne du pivot;
- Ligne 2 : indice de colonne;
- Ligne 3 : test existence du prochain pivot;
- Ligne 5 : mise à jour de l'indice de ligne du pivot;
- Ligne 6 : choix du nouveau pivot;
- Ligne 7 : normalisation;
- Ligne 10: mise du pivot à la bonne place;
- Ligne 12: élimination des des valeurs sous le pivot;

### 1.1.3 Les fonctions

Pour simplifier la lisibilité du programme certaines opérations sont gérées par des fonctions annexes telles que :

- La normalisation (fonction "*normaLigne*")
- L'échange de deux lignes (fonction "*echangeLigne*")
- La soustraction entre deux lignes (fonction "*soustractionsLignes*")
- La résolution du système, une fois la matrice échelonnée (fonction "*resolution*")

```
1       void normaLigne(float ** A, float * B, int n, int q, int j)
2       {
3           int ji;
4           float tmp = A[q][j];
5
6           "Division de chaque elements de la ligne q de matrice A et B:"
7           for(ji = q;ji < n;ji++)
8           {
9               iteration++;
10              A[q][ji] = A[q][ji] / tmp;
11          }
12          B[q] = B[q]/tmp;:
13      }
```

NormaLigne : Normalise une ligne de la matrice, ici la ligne du pivot **q**.

```
1       void echangeLigne(float ** A, float * B, int n, int q, int k)
2       {
3           float tmp;
4           int j;
5           for (j = 0; j<n ;j++)
6           {
7               iteration++;
8               tmp      = A[k][j];
9               A[k][j] = A[q][j];
10              A[q][j] = tmp;
11          }
12
13          tmp   = B[k] ;
14          B[k] = B[q];
15          B[q] = tmp;
16      }
```

EchangeLigne : Echange les valeurs de la ligne **q** et **k**.

```
1       void soustractionLignes(float ** A, float * B, int n, int i, int j, int k)
2       {
3           int ji;
4           float tmp = A[i][j];
5
6           for(ji = 0; ji < n; ji++)
7           {
8               iteration++;
9               A[i][ji] = A[i][ji] - tmp*A[k][ji];
10          }
11          B[i] -= tmp * B[k];
12      }
```

SoustractionLignes : Soustrait les valeurs de la ligne **k** aux valeurs de la ligne $i$ de sorte à ce que la valeur sous le pivot (à la colonne **j**) devienne nulle.
*Note : ne pas prendre en compte la variable* **iteration** *, elle sert à retourner la complexité pratique du programme.*

```
1       float * resolution(float ** A, float * B, int n)
2       {
3           float * X = (float *)malloc(n*sizeof(float));
4           float somme;
5
6           for (int i = n-1; i>=0; i--)              "indice de la ligne"
7           {
8               iteration++;
9               somme = 0;
10              for (int j = n-1; j>i; j--)           "indice de la colonne"
11              {
12                  somme += A[i][j]*X[j];
13              }
14              X[i] = (B[i]-somme);
15          }
16          return X;
17      }
```

Resolution : Renvoie la matrice solution du système linéaire. Elle parcours la matrice **A** de puis le bas de la diagonale (en $a_{nn}$). puis remonte jusqu'à l'autre extrémité. A ce stade, pour chaque ligne **i** on a l'inconnue $x_i$ telle que :

$$x_i = b_i - \sum_{k=i+1}^{n} a_{ik}x_k$$

### 1.1.4 Les tests

Tout les systèmes seront tels que :

$$A \cdot X = \begin{pmatrix} 1 \\ 1 \\ 1 \\ 1 \end{pmatrix}$$

## Ding Dong :

```
1.750000 ,1.250000 ,0.750000 ,0.250000 |   1.000000
1.250000 ,0.750000 ,0.250000 ,-0.250000 |   1.000000
0.750000 ,0.250000 ,-0.250000 ,-0.750000 |   1.000000
0.250000 ,-0.250000 ,-0.750000 ,-1.250000 |   1.000000

1.000000 ,0.714286 ,0.428571 ,0.142857 |   0.571429
0.000000 ,1.000000 ,1.999999 ,2.999999 |   -1.999999
0.000000 ,0.000000 ,1.000000 ,1.500000 |   -1.000000
0.000000 ,0.000000 ,0.000000 ,1.000000 |   0.000000

Nombres d'iterations : 34
Temps d'éxécution : 0.178000  seconde

Matrice solution
[ 1.0 ,0.0 ,-1.0 ,0.0 ]
```

Ici on a Ding Dong carré de dimension 4, et le système échelonné ci-dessous. Le programme trouve la matrice solution attendue. On remarque que la méthode a une complexité pratique de 34 opérations sur ce système, et qu'il le résout en 0.178 seconde environ.
La fonction d'erreur est nulle (on trouve les valeurs exactes). Résolvons le même système en faisant varier les valeurs dans b de $+0.005$, on obtiens:

```
Matrice solution
[ 0.343333 ,0.490000 ,0.000000 ,-0.833333 ]
```

Les valeurs sont assez proches, mais les valeurs à 1 et -1 se retrouvent plus proches de 0 et inversement, les valeurs proches de 0 en sont encore plus éloignées que ces dernières. La méthode n'est donc pas stable pour ce système linéaire.

## Moler :

```
1.000000 ,-1.000000 ,-1.000000 ,-1.000000 |   1.000000
-1.000000 ,2.000000 ,0.000000 ,0.000000 |   1.000000
-1.000000 ,0.000000 ,3.000000 ,1.000000 |   1.000000
-1.000000 ,0.000000 ,1.000000 ,4.000000 |   1.000000

1.000000 ,-1.000000 ,-1.000000 ,-1.000000 |   1.000000
0.000000 ,1.000000 ,-1.000000 ,-1.000000 |   2.000000
0.000000 ,0.000000 ,1.000000 ,-1.000000 |   4.000000
0.000000 ,0.000000 ,0.000000 ,1.000000 |   8.000000

Nombres d'iterations : 34
Temps d'éxécution : 0.349000  seconde

Matrice solution
[ 43.000000 ,22.000000 ,12.000000 ,8.000000 ]
```

Ici on a Ding Dong carré de dimension 4, et le système échelonné ci-dessous. Le programme trouve la bonne solution. La méthode a encore une complexité pratique de 34 opérations sur ce système, et qu'il le résout en 0.349 seconde environ.
La fonction d'erreur est nulle (on trouve les valeurs exactes). Résolvons le même système en faisant varier les valeurs dans b de +0.005, on obtiens:

```
Matrice solution
[ 43.215000 ,22.109999 ,12.059999 ,8.040000 ]
```

Or les valeurs restent sensiblement identiques, ce qui indique que la méthode est stable sur ce système.

## Franc :

```
1.000000 ,1.000000 ,1.000000 ,1.000000 |   1.000000
1.000000 ,2.000000 ,2.000000 ,2.000000 |   1.000000
0.000000 ,2.000000 ,3.000000 ,3.000000 |   1.000000
0.000000 ,0.000000 ,3.000000 ,4.000000 |   1.000000

1.000000 ,1.000000 ,1.000000 ,1.000000 |   1.000000
0.000000 ,1.000000 ,1.000000 ,1.000000 |   0.000000
0.000000 ,0.000000 ,1.000000 ,1.000000 |   1.000000
0.000000 ,0.000000 ,0.000000 ,1.000000 |   -2.000000

Nombres d'iterations : 34
Temps d'éxécution : 0.185000  seconde

Matrice solution
[ 1.0 ,-1.0 ,3.0 ,-2.0 ]
```

Ici on a Ding Dong carré de dimension 4, et le système échelonné ci-dessous. Le programme trouve la matrice solution attendue. On remarque que la méthode a une complexité pratique de 34 opérations sur ce système, et qu'il le résout en 0.221 seconde environ.
La fonction d'erreur est nulle (on trouve les valeurs exactes).
Résolvons le même système en faisant varier les valeurs dans b de +0.005, on obtiens:

```
Matrice solution
[ 1.005000 ,-1.005000 ,3.015000 ,-2.010000 ]
```

Or les valeurs restent sensiblement identiques, ce qui indique que la méthode est stable sur ce système linéaire également.

## 1.2 Cholesky

### 1.2.1 Principe

Le principe de cette méthode est de trouver une matrice $R$ telles que $A = R^T R$ où $R$ est une matrice triangulaire supérieure et $R^T$ est la matrice transposée de R. Le système $Ax = b$ peut donc se diviser en deux sous-systèmes $R^T y = b$ et $Rx = y$. Cette méthode est légèrement plus rapide que celle de Gauss avec une complexité de $\frac{n^3}{3}$ au lieu de $\frac{2n^3}{3}$, cependant elle nécessite que la matrice soit définie positive.

### 1.2.2 L'Algorithme

L'algorithme qui permet de trouver la matrice $R$ est le suivant:

```
1  Pour i allant de 1 a n:
2      s = a_ii − Σ_{j=1}^{i−1} r_ji^2
3      Si s ⩽ 0 alors:
4          arret("la matrice n'est pas definie positive")
5      Sinon
6          r_ii = √s
7          Pour j allant de i+1 a n:
8              r_ij = (a_ij − Σ_{k=1}^{i−1}) / r_ii
```

Ensuite il suffit de résoudre les deux sous-systèmes ce qui ce fait facilement car la matrice est triangulaire.

### 1.2.3 Les Variables

Dans la fonction *main* on initialise un matrice de taille **LEN** qui est une constante pré-processeur. Les valeurs dans la matrice dépendent de la constante **EXAMPLE** qui permet de déterminer l'initialisation de la matrice via une fonction qui se situe dans le fichier *matrix.c*.

```
1   float *solus = malloc(LEN * sizeof(float));
2   if (solus == NULL)
3     exit(EXIT_FAILURE);
4   float *b = malloc(LEN * sizeof(float));
5   if (b == NULL)
6     exit(EXIT_FAILURE);
7   for (i = 0; i < LEN; i++) {
8     b[i] = 1;
9   }
10  float **matrix = create_mat(LEN);
11  mat(matrix, LEN, EXAMPLE);
12  show(matrix, LEN);
13  solver_cholesky(matrix, solus, b, LEN);
```

### 1.2.4 Calcule de la matrice R

On calcule la matrice $R$ (le tableau à deux dimensions $r$) à l'aide de la fonction nommée **make_R** qui suit l'algorithme ci-dessus. A noter que cette fonction renvoi une erreur si la matrice en entrée n'est pas définie positive ce qui stop le programme car le système n'est pas résolvable avec cette méthode (en fait si la matrice n'est pas définie positive, la matrice $R$ n'existe pas).

```c
void make_R(float **matrix, float **r, int n) {
  int i, j, k;   // loop var
  float sum = 0; // Used to calculate the sum of r_ki*r_kj
  float s;
  for (i = 0; i < n; i++) {
    s = matrix[i][i];
    for (j = 0; j < i; j++) {
      s -= r[j][i] * r[j][i];
    }
    if (s <= 0) {
      printf("La matrice n'est pas d finie positive\n");
      exit(EXIT_FAILURE);
    } else {
      r[i][i] = sqrtf(s);
      for (j = (i + 1); j < n; j++) {
        sum = 0;
        for (k = 0; k < i; k++) {
          sum += r[k][i] * r[k][j];
        }
        r[i][j] = (matrix[i][j] - sum) / r[i][i];
      }
    }
  }
}
```

### 1.2.5 Résolution des sous-systèmes

Enfin la fonction **solver_cholesky** résout les deux sous-systèmes de la même manière qu'avec *Gauss* sauf que les $a_{ii}$ ne sont pas forcement égaux à 1 d'où la division pas $a_{ii}$. De plus, dans le premier sous-système, la matrice $R^T$ est triangulaire inférieure, la résolution se fait donc en commencent par calculer la valeur de la première inconnue.

```c
void solver_cholesky(float **matrix, float *solus_x, float *b, int n) {
  int i, j;
  float tmp;
  float *solus_y = malloc(n * sizeof(float));
  /*Creation of the matrix  R:*/
  float **r = create_mat(n);
  mat_0(r, n);
  make_R(matrix, r, n);
  /*Resolution de R^T * y = b*/
  transpose(r, n);
  for (i = 0; i < n; i++) {
    tmp = b[i];
    for (j = 0; j < i; j++) {
      tmp -= solus_y[j] * r[i][j];
    }
    solus_y[i] = tmp / r[i][i];
  }
  /*Resolution R * x = y*/
  transpose(r, n);
  for (i = (n - 1); i >= 0; i--) {
    for (j = (n - 1); j > i; j--) {
      solus_y[i] -= solus_x[j] * r[i][j];
    }
    solus_x[i] = solus_y[i] / r[i][i];
  }
  /*FREE*/
  free(solus_y);
  for (i = 0; i < n; i++) {
    free(r[i]);
```

```
30    }
31    free(r);
32  }
```

Cette fonction fait appelle à plusieurs autres fonctions:

- **create_mat** qui retourne un tableau à deux dimensions de taille n
- **mat_0** qui remplit un tableau à deux dimensions de 0
- **make_R** vue ci dessus
- **transpose** qui transpose la matrice qui lui est donnée en argument

Résolution du premier sous_système en détail:

```
1     for (i = 0; i < n; i++) {
2       for (j = 0; j < i; j++) {
3         b[i] -= solus_y[j] * r[i][j];
4       }
5       solus_y[i] = b[i] / r[i][i];
6     }
```

Voici la matrice r (après avoir été transposé):

$$\begin{vmatrix} a_{11} & 0 & \cdots & \cdots & 0 \\ a_{21} & a_{22} & 0 & \cdots & 0 \\ \vdots & \cdots & \ddots & 0 & 0 \\ a_{n1} & \cdots & \cdots & \cdots & a_{nn} \end{vmatrix}$$

Nous résolvons $Ry = b$ donc:

$$y_i = \frac{b_i - \sum_{j=1}^{i}}{a_{ii}}$$

Ce qui se fait à l'aide des deux boucles ci-dessus.

A noter que ce programme alloue plus de mémoire que le précédent à cause de l'utilisation du tableau **r** ce qui pourrait être problématique si on travail avec des matrices de grandes taille.

### 1.2.6 Les Tests

- Bord

```
| 1.000000 | 0.500000 | 0.250000 | 0.125000 |
| 0.000000 | 1.000000 | 0.000000 | 0.000000 |
| 0.000000 | 0.000000 | 1.000000 | 0.000000 |
| 0.000000 | 0.000000 | 0.000000 | 1.000000 |
0.186047
0.906977
0.953488
0.976744
nb itération: 34
temps = 0.000004
```

La méthode ne converge pas car la matrice n'est pas à diagonale dominante. Les vraies solutions du système sont: $x_1 = \frac{1}{8} = 0.125, x_2 = 1, x_3 = 1, x_4 = 1$

- Ding Dong

```
| 1.750000 | 1.250000 | 0.750000 | 0.250000 |
| 1.250000 | 0.750000 | 0.250000 | -0.250000 |
| 0.750000 | 0.250000 | -0.250000 | -0.750000 |
| 0.250000 | -0.250000 | -0.750000 | -1.250000 |
La matrice n'est pas définie positive
```

Les programme retourne une erreur, en effet, la matrice n'est pas définie positive donc la méthode ne converge pas.

- Matrice à diagonale dominante

```
| 64.000000 | 40.000000 | 24.000000 |
| 40.000000 | 29.000000 | 17.000000 |
| 24.000000 | 17.000000 | 19.000000 |
-0.046007
0.069444
0.048611
nb itération: 16
temps = 0.000007
```

Cette matrice est a diagonale dominante donc la méthode converge. La solution en valeur exacte est : $x_1 = \frac{-53}{1152} \approx -0.4600694444$, $x - 2 = \frac{5}{72} \approx 0.06944444444$ $x_3 = \frac{7}{144} = 0.048611111111$

Ce qui est très poche de ce que trouve le programme soit une précision de 0.18% calculée avec la formule suivante:

$$\frac{\| x_c \| - \| x_p \|}{\| x_c \|}$$

On constate en observant ces test qu'il y a une incohérence dans le nombre d'itérations, en effet, pour une matrice de même taille, cholesky devrait mettre approximativement deux fois mois d'itérations pour trouver la solution (si on se réfère à la complexité théorique). Si on observe le code en annexe, on voit que cela reste correcte car pour **gauss** on compte les opérations uniquement sur les lignes et avec **cholesky** on compte les opérations sur les valeurs donc gauss fait bien deux fois plus d'itérations que cholesky.

De plus, le temps d'exécution de gauss est nettement plus important que celui de cholesky ce qui est incohérence et qui traduit certainement une erreur d'optimisation dans gauss.

### 1.3 Conclusion sur les méthodes de résolution directes

Pour conclure, Gauss permet de trouver une solution exacte, excepté si les coefficients sont des valeurs extrêmes (les approximations de la machine fausse le résultat de cette méthode) et est stable aux écart de valeurs en général. Cependant sa complexité est un problème, en effet on ne peut pas utiliser efficacement cette méthode sur des matrices à très grande dimensions.

La méthode de Choleski a une complexité deux fois plus basse que celle de Gauss (donc deux fois plus rapide en théorie), se plus elle est stable aux écart de valeurs. Cependant elle est limité par la nature des matrices qu'on lui soumet, si une matrice n'est pas définie positive, on ne peut pas appliquer cette méthode au système linéaire (on ferait donc des opérations inutiles en l'utilisant ainsi).

Ainsi pour les matrices définies positives, il vaut mieux utiliser la méthode de Choleski plus efficace, sinon la méthode du pivot de Gauss est utile pour ces autres cas.

## 2 algorithmes itératifs

Les algorithmes itératifs sont des algorithmes qui vont trouver une solution de plus en plus précise à chaque itération. Ils partent d'un point de départ arbitraire (un vecteur arbitraire) et vont converger vers la solution du système.

A noter que les formules utilisées par ces deux méthodes sont obtenues en divisant la matrice $A$ du système en trois sous-matrices $D$, $-E$ et $-F$ où $D$ est une matrice diagonale composée uniquement de la diagonale de $A$. $-E$ est une matrice triangulaire inférieure composée des éléments inférieures de $A$ et $-F$ est une matrice triangulaire supérieure composée des éléments supérieurs de $A$.

Exemple

$$A = \begin{pmatrix} 1 & 1 & 1 \\ 1 & 2 & 2 \\ 1 & 2 & 3 \end{pmatrix} \Rightarrow D = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 2 & 0 \\ 0 & 0 & 3 \end{pmatrix}, -E = \begin{pmatrix} 0 & 0 & 0 \\ 1 & 0 & 0 \\ 1 & 2 & 0 \end{pmatrix} et - F = \begin{pmatrix} 0 & 1 & 1 \\ 0 & 0 & 2 \\ 0 & 0 & 0 \end{pmatrix}$$

## 2.1 Jacobi

### 2.1.1 Principe

Cet algorithme détermine les solutions du système $Ax = b$ en utilisant la formule suivante:

$$x^{(k+1)} = D^{-1}(E + F)x^{(k)} + D^{-1}b$$

En utilisant les matrices $D$, $-E$ et $-F$ définies ci-dessus, ce qui donne:

$$x_i^{(k+1)} = \frac{b_i - \sum_{j=1}^{i-1} a_{ij}x_j^{(k)} - \sum_{j=i+1}^{n} a_{ij}x_j^{(k)}}{a_{ii}}$$

A chaque tours de boucle, l'algorithme calcule le nouveau vecteur $x^{(k+1)}$ en utilisant la solution $x^{(k)}$ qu'il a calculé à l'itération précédente qui au premier tour est un vecteur choisit arbitrairement qui sert de point de départ. L'algorithme s'arrête quand la solution trouvée est convenable $\| x^{(k+1)} - x^{(k)} \| \leqslant \varepsilon$ (ou $\varepsilon$ est la précision choisit) ou quand il a fait un nombre de tours pré-déterminer (pour éviter les boucles infinie si la méthode ne converge pas).

### 2.1.2 L'algorithme

---

1 Tant que $\varepsilon^{(k)} \geqslant \varepsilon$ :
2 $x_i^{(k+1)} = \frac{b_i - \sum_{j=1}^{i-1} a_{ij}x_j^{(k)} - \sum_{j=i+1}^{n} a_{ij}x_j^{(k)}}{a_{ii}}$
3 $\varepsilon^{(k+1)} = \| x^{(k+1)} - x^{(k)} \|$

---

### 2.1.3 Implémentation en C

Les variables sont initialisées de la même façon que dans les méthodes directes vues précédemment et le point de départ est **solus_k** qui vaut $(1, \ldots, 1)$. La fonction qui calcule les solutions est la suivante:

---

```c
void jacobi(float **matrix, float *b, float *solus_k, int n) {
  int i, j;
  int compt = 0;
  int bool = 1;
  float *solus_k1 = malloc(n * sizeof(float));
  while (bool && compt < MAX) {
    for (i = 0; i < n; i++) {
      solus_k1[i] = b[i];
      for (j = 0; j < n; j++) {
        if (j != i)
          solus_k1[i] -= matrix[i][j] * solus_k[j];
      }
      solus_k1[i] /= matrix[i][i];
    }
    bool = test(solus_k1, solus_k, n);
    for (i = 0; i < n; i++) {
      solus_k[i] = solus_k1[i];
    }
    compt++;
  }
  printf("Nb tours: %d\n", compt);
  // FREE
  free(solus_k1);
}
```

---

Cette fonction utilise le même principe que l'algorithme ci dessus, elle fait appel à une fonction *test* qui retourne $\| x^{(k+1)} - x^{(k)} \| \leqslant \varepsilon$ qui détermine l'arrêt (où sinon la fonction s'arrête d'itérer à un nombre de tours maximum). A la fin, on affiche le nombre de tours effectués pour avoir un idée de la vitesse de convergence.

## 2.2 Gauss-seidel

### 2.2.1 Le principe

Cette méthode repose sur un principe similaire à celle de *Jacobi*, cependant, elle utilises les solutions $x^{(k+1)}$ au fur et à mesure qu'elle les calcules, ce qui lui permet de converger plus vite et donc de trouver une solution plus rapidement. La formule utilisée par la méthode est la suivante:

$$x^{(k+1)} = D^{-1}(Ex^{(k+1)} + fx^{(k)} + b)$$

Ce qui donne:

$$x_i^{(k+1)} = \frac{b_i - \sum_{j=1}^{i-1} a_{ij}x_j^{(\mathbf{k+1})} - \sum_{j=i+1}^{n} a_{ij}x_j^{(k)}}{a_{ii}}$$

Cette méthode converge lorsque la matrice est définie positive et symétrique, ou si elle est à diagonale dominante.

### 2.2.2 L'algorithme

L'algorithme est le même qu'avec **Jacobi** sauf qu'il utilise la nouvelle formule:

```
1  Tant que ε^(k) ⩾ ε:
2      x_i^(k+1) = (b_i − Σ_{j=1}^{i−1} a_ij x_j^(k+1) − Σ_{j=i+1}^{n} a_ij x_j^(k)) / a_ii
3      ε^(k+1) = ‖ x^(k+1) − x^(k) ‖
```

### 2.2.3 Le programme en C

la fonction utilisée dans ce programme est très similaire à celle utilisée avec **Jacobi** mis à part qu'elle utilise l'autre formule ce qui ce traduit part un changement de variable.

```
1  void gauss_seidel(float **matrix, float *b, float *solus_k, int n) {
2    int i, j;
3    int compt = 0;
4    int bool = 1;
5    float *solus_k1 = malloc(n * sizeof(float));
6    while (bool && compt < MAX) {
7      for (i = 0; i < n; i++) {
8        solus_k1[i] = b[i];
9        for (j = 0; j < i; j++) {
10         solus_k1[i] -= matrix[i][j] * solus_k1[j];
11       }
12       for (j = i+1; j < n; j++) {
13     solus_k1[i] -= matrix[i][j] * solus_k[j];
14       }
15       solus_k1[i] /= matrix[i][i];
16     }
17     bool = test(solus_k1, solus_k, n);
18     for (i = 0; i < n; i++) {
19       solus_k[i] = solus_k1[i];
20     }
21     compt++;
22   }
23   printf("Nb tours: %d\n", compt);
24   // FREE
25   free(solus_k1);
26 }
```

La modification est au niveau de la ligne **10** où on utilise maintenant deux boucles la première calcule $b - \sum_{j=1}^{i-1} a_{ij}x_j^{(\mathbf{k+1})}$ et la deuxième $b - \sum_{j=i+1}^{n} a_{ij}x_j^{(k)}$ puis on divise par $a_{ii}$.

## 2.3 Comparaison des résultats

### 2.3.1 Matrice symétrique à diagonale dominante

Dans cette exemple nous allons résoudre deux systèmes avec la même matrice symétrique à diagonale supérieure:

Résolution de:

$$\begin{pmatrix} 4 & 1 & 1 & 0 \\ 1 & 4 & 0 & 1 \\ 1 & 0 & 4 & 1 \\ 0 & 1 & 1 & 4 \end{pmatrix} X = \begin{pmatrix} 15 \\ 15 \\ 19 \\ 11 \end{pmatrix}$$

Voici la solution trouvée par **Jacobi**:

```
| 4.000000 | 1.000000 | 1.000000 | 0.000000 |
| 1.000000 | 4.000000 | 0.000000 | 1.000000 |
| 1.000000 | 0.000000 | 4.000000 | 1.000000 |
| 0.000000 | 1.000000 | 1.000000 | 4.000000 |
Nb tours: 24
2.000000
3.000000
4.000000
1.000000
temps = 0.000030
```

Voici la solution trouvée par **Gauss Seidel**:

```
| 4.000000 | 1.000000 | 1.000000 | 0.000000 |
| 1.000000 | 4.000000 | 0.000000 | 1.000000 |
| 1.000000 | 0.000000 | 4.000000 | 1.000000 |
| 0.000000 | 1.000000 | 1.000000 | 4.000000 |
Nb tours: 13
2.000000
3.000000
4.000000
1.000000
temps = 0.000025
```

On constate que la solution trouvée avec les deux méthodes et la même et si on regarde le nombre de tours effectués, on voit que **gauss-seidel** converge presque deux fois plus vite que **jacobi**. Cependant, on constate tout de même que le temps d'exécution du programme est presque le même. La solution n'est pas arrondie, elle est exacte.

Résolvons le même système en faisant varier les valeurs dans b de +0.005:

Voici la solution trouvée par **Jacobi**:

```
| 4.000000 | 1.000000 | 1.000000 | 0.000000 |
| 1.000000 | 4.000000 | 0.000000 | 1.000000 |
| 1.000000 | 0.000000 | 4.000000 | 1.000000 |
| 0.000000 | 1.000000 | 1.000000 | 4.000000 |
Nb tours: 25
2.000834
3.000833
4.000833
1.000833
temps = 0.000025
```

Voici la solution trouvée par **Gauss Seidel**:

```
| 4.000000 | 1.000000 | 1.000000 | 0.000000 |
| 1.000000 | 4.000000 | 0.000000 | 1.000000 |
| 1.000000 | 0.000000 | 4.000000 | 1.000000 |
| 0.000000 | 1.000000 | 1.000000 | 4.000000 |
Nb tours: 13
2.000834
3.000833
4.000833
1.000833
temps = 0.000019
```

On constate que les résultats ainsi que les temps d'exécution (légère variation pour gauss seidel) et le nombre d'itérations n'ont pas beaucoup variés, donc la modification effectuée sur b sont bien gérées par le programme donc les méthodes sont stables pour ce système.

### 2.3.2 Autre système

Résolution du système suivant:

$$\begin{pmatrix} 4 & 1 & -2 \\ -1 & 3 & 0 \\ -2 & -5 & 8 \end{pmatrix} X = \begin{pmatrix} 8 \\ 3 \\ 8 \end{pmatrix}$$

Solutions trouvée par **Jacobi**:

```
| 4.000000 | 1.000000 | -2.000000 |
| -1.000000 | 3.000000 | 0.000000 |
| -2.000000 | -5.000000 | 8.000000 |
Nb tours: 26
3.000000
2.000000
3.000000
temps = 0.000010
```

Solutions trouvée par **Gauss Seidel**:

```
| 4.000000 | 1.000000 | -2.000000 |
| -1.000000 | 3.000000 | 0.000000 |
| -2.000000 | -5.000000 | 8.000000 |
Nb tours: 10
3.000000
2.000000
3.000000
temps = 0.000007
```

On constate encore une fois que **Jacobi** itère plus de fois que **gauss seidel** mais que les temps d'exécution des deux programmes sont toujours très proches. De plus, le programme trouve toujours une solution précise.

Résolvons le même système en faisant varier les valeurs dans b de +0.005:

Voici la solution trouvée par **Jacobi**:

```
| 4.000000 | 1.000000 | 1.000000 | 0.000000 |
| 1.000000 | 4.000000 | 0.000000 | 1.000000 |
| 1.000000 | 0.000000 | 4.000000 | 1.000000 |
| 0.000000 | 1.000000 | 1.000000 | 4.000000 |
Nb tours: 25
2.000834
3.000833
4.000833
1.000833
temps = 0.000025
```

Voici la solution trouvée par **Gauss Seidel**:

```
| 4.000000 | 1.000000 | 1.000000 | 0.000000 |
| 1.000000 | 4.000000 | 0.000000 | 1.000000 |
| 1.000000 | 0.000000 | 4.000000 | 1.000000 |
| 0.000000 | 1.000000 | 1.000000 | 4.000000 |
Nb tours: 13
2.000834
3.000833
4.000833
1.000833
temps = 0.000019
```

On constate à nouveau la stabilité des méthodes sur cet exemple.

### 2.3.3 Une Erreur

Avec ce test, on voit que si la matrice ne respecte pas les contrainte requises par les méthodes, celles-ci tournes à l'infini sans jamais converger vers la solution du système (le programme s'arrête car la fonction

est limité à 100 tours).

Résolution de:

$$\begin{pmatrix} 1 & 2 & 1 \\ 2 & 2 & 3 \\ 5 & 1 & 8 \end{pmatrix} X = \begin{pmatrix} 2 \\ -1 \\ 3 \end{pmatrix}$$

Voici la solution trouvée par **jacobi**:

```
| 1.000000 | 2.000000 | 1.000000 |
| 2.000000 | 2.000000 | 3.000000 |
| 5.000000 | 1.000000 | 8.000000 |
Nb tours: 100
1566439752197329627469756497 92.000000
1243872464955820438002377687 04.000000
5794099055897242029582922547 2.000000
temps = 0.000019
```

Voici la solution trouvée par **gauss seidel**:

```
| 1.000000 | 2.000000 | 1.000000 |
| 2.000000 | 2.000000 | 3.000000 |
| 5.000000 | 1.000000 | 8.000000 |
Nb tours: 100
-18743254056960.000000
-181350950764544.000000
34383402631168.000000
temps = 0.000022
```

Les solutions trouvées sont incohérentes, les vraies solutions du système sont: $x_1 = \frac{76}{3} \approx 25,333333333$, $x_2 = \frac{-8}{3} \approx -2,666666667$ et $x_3 = \frac{-31}{3} \approx 10,333333333$

## 2.4 Conclusion sur les méthodes itératives

Le principal défaut de ces méthodes est qu'elles ne converges pas pour tous les systèmes, en effet, si la matrice n'est pas à diagonale dominante ou symétrique et définie positive (pour Gauss Seidel), ces méthodes ne trouverons pas de solutions aux système et bouclerons à l'infini si le programme n'est pas prévu pour s'arrêter à un nombre maximum d'itérations.

Cependant, ces deux méthodes sont bien plus précises que les méthodes directes et sont donc efficaces pour palier au problème du manque de précision et l'accumulation d'erreurs de calcules dues à l'utilisation des flottants dans les ordinateurs du fait que la solution est améliorée à chaque itération. La précision de la solution dépend de la précision demandé dans le programme qui est limité par la précision des flottants et la mémoire de l'ordinateur. A noter que plus la précision demandé est élevée, plus le temps d'exécution sera grand, cependant, en théorie, sans prendre en compte les faiblesses de la machine, si les méthodes converge, on peut obtenir une précision infinie.

# Gauss

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

int iteration; //variables compteur du nombres d'op rations sur la matrice.
//
    ////////////////////////////////////////////////////////////////////////////////////////////////////////////////

void normaLigne         (float ** A, float * B, int n, int q, int j);
void echangeLigne       (float ** A, float * B, int n, int q, int k);
void soustractionLignes(float ** A, float * B, int n, int i, int j, int k);
void gauss              (float ** A, float * B, int n);
float * resolution      (float ** A, float * B, int n);
//
    ////////////////////////////////////////////////////////////////////////////////////////////////////////////////

int main ()
{
    iteration = 0;
    int i,j,n;

    printf("(GaussPivot)Taille matrices \n");
    scanf("%d",&n);

    float Tdep = clock();
    float *  B = (float * ) malloc(n*sizeof(float  ));

    //Initialisation de la matrice B.
    for (i = 0; i < n; i++)
    {
        B[i] = 1;
    }

    float ** A = create_mat(n);
    mat(A, n, 2);

    //affichage matrice
    for (i = 0; i<n; i++)
    {
        for(j = 0; j<n;j++)
        {
            printf("%f ,", A[i][j]);
        }
        printf("|   %f", B[i]);
        printf("\n");
    }

    gauss(A,B,n);
    printf("\n");

    //Affichage matrice fin gauss
    for (i = 0; i<n; i++)
    {
        for(j = 0; j<n;j++)
        {
            printf("%f ,", A[i][j]);
        }

        printf("|   %f", B[i]);
        printf("\n");
    }

    //affichage du temps d'execution et de la complexit  pratique.
    float Tfin = clock();
    printf("\nNombres d'iterations : %d\n", iteration );
    printf("Temps d'   xcution    : %f", (Tfin - Tdep)/1000);

    // trouver matrice solution
```

```
67      float * solution = (float *) malloc(n*sizeof(float));
68      solution = resolution(A,B,n);
69
70      //affichage de la solution
71      printf("[ ");
72      for(i=0 ;i<n; i++)
73      {
74          printf("%.1f ,",solution[i]);
75      }
76      printf("\b]\n");
77
78
79  //
        ////////////////////////////////////////////////////////////////////////////////////////////////////
80      free(B);free(solution);
81      for (int i = 0; i < n; i++)
82      {
83          free(A[i]);
84      }
85      free(A);
86      return 0;
87  }
88
89  //
        ////////////////////////////////////////////////////////////////////////////////////////////////////
90  void normaLigne(float ** A, float * B, int n, int q, int j)
91  {
92      int ji;
93      float tmp = A[q][j];
94
95      // Division de chaque    lments    de la ligne q de matrice A & B: "normalisation de la
         ligne"
96      for(ji = q;ji < n;ji++)
97      {
98          iteration++;
99          A[q][ji] = A[q][ji] / tmp;
100     }
101     B[q] = B[q]/tmp;
102 }
103 void echangeLigne(float ** A, float * B, int n, int q, int k)
104 {
105     float tmp;
106     int j;
107     for (j = 0; j<n ;j++)
108     {
109         iteration++;
110         tmp      = A[k][j];
111         A[k][j] = A[q][j];
112         A[q][j] = tmp;
113     }
114
115     tmp  = B[k] ;
116     B[k] = B[q];
117     B[q] = tmp;
118 }
119 void soustractionLignes(float ** A, float * B, int n, int i, int j, int k)
120 {
121     int ji;
122     float tmp = A[i][j];
123
124     for(ji = 0; ji < n; ji++)
125     {
126         iteration++;
127         A[i][ji] = A[i][ji] - tmp*A[k][ji];
128     }
129     B[i] -= tmp * B[k];
130 }
131
132 void gauss(float ** A,float * B,int n)
133 {
134     int i, j, k,test, q;
```

```
135     k = -1;                                                 //indice de ligne du pivot
136
137     for (j = 0; j < n; j++)                                 //indice de colonne
138     {
139         test = 0;
140
141         for (i = k+1; i < n; i++)                           //test de l' xistence  du
        prochain pivot
142             {
143                 if(A[i][j]!=0) {test = 1; break;}
144             }
145
146         if (test)                                           //Si teste concluant..
147         {
148             k++;                                            //MAJ de l'indice de ligne du
         pivot
149             q = i;                                          //r cup ration de la ligne
        du prochain pivot(trouv  avec le teste)
150             normaLigne(A,B,n,q,j);                          //normalisation de la ligne
151             if (q!=k) echangeLigne(A,B,n,q,k);              //mise du pivot au bon
        endroit
152
153             for (i = k+1; i < n; i++)                       //  limination  des valeurs
        sous le pivot
154             {
155                 soustractionLignes(A, B, n, i, j, k);
156             }
157         }
158     }
159 }
160
161 float * resolution(float ** A, float * B, int n)
162 {
163     float * X = (float *)malloc(n*sizeof(float));
164     float somme;
165
166     for (int i = n-1; i>=0; i--)                            //indice de la ligne
167     {
168         iteration++;
169         somme = 0;
170         for (int j = n-1; j>i; j--)                         //indice de la colonne
171         {
172             somme += A[i][j]*X[j];
173         }
174         X[i] = (B[i]-somme);
175     }
176     return X;
177 }
```

./prog/gauss.c

# Cholesky

```c
#include "matrix.h"
#include <math.h>
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#define LEN 4
#define EXAMPLE 1

int nbtours;

void make_R(float **matrix, float **r, int n) {
  int i, j, k;   // variables de boucles
  float sum = 0; // Utilise pour calculer la somme des r_ki*r_kj
  float s;
  for (i = 0; i < n; i++) {
    s = matrix[i][i];
    for (j = 0; j < i; j++) {
      s -= r[j][i] * r[j][i];
      nbtours++;
    }
    if (s <= 0) {
      printf("La matrice n'est pas d finie positive\n");
      exit(EXIT_FAILURE);
    } else {
      r[i][i] = sqrtf(s);
      for (j = (i + 1); j < n; j++) {
        sum = 0;
        for (k = 0; k < i; k++) {
          sum += r[k][i] * r[k][j];
          nbtours++;
        }
        r[i][j] = (matrix[i][j] - sum) / r[i][i];
      }
    }
  }
}

// transpose la matrice en entree
void transpose(float **matrix, int n) {
  int i, j;
  float tmp;
  for (i = 0; i < n; i++) {
    for (j = (i + 1); j < n; j++) {
      tmp = matrix[i][j];
      matrix[i][j] = matrix[j][i];
      matrix[j][i] = tmp;
      nbtours++;
    }
  }
}

void solver_cholesky(float **matrix, float *solus_x, float *b, int n) {
  int i, j;
  float tmp;
  float *solus_y = malloc(n * sizeof(float));
  /*Creation de la matrice R:*/
  float **r = create_mat(n);
  mat_0(r, n);
  make_R(matrix, r, n);
  /*Resolution de R^T * y = b*/
  transpose(r, n);
  for (i = 0; i < n; i++) {
    tmp = b[i];
    for (j = 0; j < i; j++) {
      tmp -= solus_y[j] * r[i][j];
      nbtours++;
    }
    solus_y[i] = tmp / r[i][i];
  }
  /*Resolution de R * x = y*/
```

```c
  transpose(r, n);
  for (i = (n - 1); i >= 0; i--) {
    for (j = (n - 1); j > i; j--) {
      solus_y[i] -= solus_x[j] * r[i][j];
      nbtours++;
    }
    solus_x[i] = solus_y[i] / r[i][i];
  }
  /*FREE*/
  free(solus_y);
  for (i = 0; i < n; i++) {
    free(r[i]);
  }
  free(r);
}

void show(float **matrix, int n){
  int i, j;
  for (i = 0; i < n; i++) {
    printf("|");
    for (j = 0; j < n; j++) {
      printf(" %f |", matrix[i][j]);
    }
    printf("\n");
  }
}

int main() {
  nbtours = 0;
  // Time
  float temps;
  clock_t t1, t2;
  int i;
  float *solus = malloc(LEN * sizeof(float));
  if (solus == NULL)
    exit(EXIT_FAILURE);
  float *b = malloc(LEN * sizeof(float));
  if (b == NULL)
    exit(EXIT_FAILURE);
  for (i = 0; i < LEN; i++) {
    b[i] = 1;
  }
  float **matrix = create_mat(LEN);
  mat(matrix, LEN, EXAMPLE);
  show(matrix, LEN);
  t1 = clock();
  solver_cholesky(matrix, solus, b, LEN);
  t2 = clock();
  // Affiche les solutions
  for (i = 0; i < LEN; i++) {
    printf("%f\n", solus[i]);
  }
  printf("nb it ration: %d\n", nbtours);
  temps = (float)(t2 - t1) / CLOCKS_PER_SEC;
  printf("temps = %f\n", temps);
  /*FREE*/
  free(solus);
  free(b);
  for (i = 0; i < LEN; i++) {
    free(matrix[i]);
  }
  free(matrix);
  return 0;
}
```

./prog/cholesky.c

# Jacobi

```c
#include "matrix_tp2.h"
#include <math.h>
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#define LEN 3
#define SIGMA 0.0000001
#define MAX 100
#define EXAMPLE 4

int test(float *solus_k, float *solus_k1, int n) {
  int i;
  float *tmp = malloc(n * sizeof(float));
  float s_quadra = 0;
  for (i = 0; i < n; i++) {
    tmp[i] = solus_k1[i] - solus_k[i];
    s_quadra += tmp[i] * tmp[i];
  }
  // FREE
  free(tmp);
  return (sqrtf(s_quadra) > SIGMA);
}

void jacobi(float **matrix, float *b, float *solus_k, int n) {
  int i, j;
  int compt = 0;
  int bool = 1;
  float *solus_k1 = malloc(n * sizeof(float));
  while (bool &&compt < MAX) {
    for (i = 0; i < n; i++) {
      solus_k1[i] = b[i];
      for (j = 0; j < n; j++) {
        if (j != i)
          solus_k1[i] -= matrix[i][j] * solus_k[j];
      }
      solus_k1[i] /= matrix[i][i];
    }
    bool = test(solus_k1, solus_k, n);
    for (i = 0; i < n; i++) {
      solus_k[i] = solus_k1[i];
    }
    compt++;
  }
  printf("Nb tours: %d\n", compt);
  // FREE
  free(solus_k1);
}

void show(float **mat, int n){
  int i, j;
  for (i = 0; i < n; i++) {
    printf("|");
    for (j = 0; j < n; j++) {
      printf(" %f |", mat[i][j]);
    }
    printf("\n");
  }
}

int main() {
  float temps;
  clock_t t1, t2;
  int i;
  // init b
  float *b = malloc(LEN * sizeof(float));
  if (b == NULL)
    exit(EXIT_FAILURE);
  // init solus
  float *solus = malloc(LEN * sizeof(float));
  if (solus == NULL)
```

```c
    exit(EXIT_FAILURE);
  for (i = 0; i < LEN; i++)
    solus[i] = 0;
  // init matrix
  float **matrix = create_mat(LEN);
  // Initialise matrix et b en fonction de l'exemple choisit
  init_mat(matrix, b, LEN, EXAMPLE);
  printf("\n");
  show(matrix, LEN);
  t1 = clock();
  jacobi(matrix, b, solus, LEN);
  t2 = clock();
  for (i = 0; i < LEN; i++) {
    printf("%f\n", solus[i]);
  }
  temps = (float)(t2 - t1) / CLOCKS_PER_SEC;
  printf("temps = %f\n", temps);
  // FREE
  free(solus);
  free(b);
  for (i = 0; i < LEN; i++) {
    free(matrix[i]);
  }
  free(matrix);
  return 0;
}
```

./prog/jacobi.c

# Gauss Seidel

```c
#include "matrix_tp2.h"
#include <math.h>
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#define LEN 3
#define SIGMA 0.0000001
#define MAX 100
#define EXAMPLE 4

int test(float *solus_k, float *solus_k1, int n) {
  int i;
  float *tmp = malloc(n * sizeof(float));
  float s_quadra = 0;
  for (i = 0; i < n; i++) {
    tmp[i] = solus_k1[i] - solus_k[i];
    s_quadra += tmp[i] * tmp[i];
  }
  // FREE
  free(tmp);
  return (sqrtf(s_quadra) > SIGMA);
}

void gauss_seidel(float **matrix, float *b, float *solus_k, int n) {
  int i, j;
  int compt = 0;
  int bool = 1;
  float *solus_k1 = malloc(n * sizeof(float));
  while (bool &&compt < MAX) {
    for (i = 0; i < n; i++) {
      solus_k1[i] = b[i];
      for (j = 0; j < i; j++) {
        solus_k1[i] -= matrix[i][j] * solus_k1[j];
      }
      for (j = i + 1; j < n; j++) {
        solus_k1[i] -= matrix[i][j] * solus_k[j];
      }
      solus_k1[i] /= matrix[i][i];
    }
    bool = test(solus_k1, solus_k, n);
    for (i = 0; i < n; i++) {
      solus_k[i] = solus_k1[i];
    }
    compt++;
  }
  printf("Nb tours: %d\n", compt);
  // FREE
  free(solus_k1);
}

void show(float **mat, int n){
  int i, j;
  for (i = 0; i < n; i++) {
    printf("|");
    for (j = 0; j < n; j++) {
      printf(" %f |", mat[i][j]);
    }
    printf("\n");
  }
}

int main() {
  float temps;
  clock_t t1, t2;
  int i;
  // init b
  float *b = malloc(LEN * sizeof(float));
  if (b == NULL)
    exit(EXIT_FAILURE);
  // init solus
```

```c
  float *solus = malloc(LEN * sizeof(float));
  if (solus == NULL)
    exit(EXIT_FAILURE);
  for (i = 0; i < LEN; i++)
    solus[i] = 0;
  // alloc matrix
  float **matrix = create_mat(LEN);
  // Initialise matrix en fonction de l'exemple choisit
  init_mat(matrix, b, LEN, EXAMPLE);
  printf("\n");
  show(matrix, LEN);
  t1 = clock();
  gauss_seidel(matrix, b, solus, LEN);
  t2 = clock();
  for (i = 0; i < LEN; i++) {
    printf("%f\n", solus[i]);
  }
  temps = (float)(t2 - t1) / CLOCKS_PER_SEC;
  printf("temps = %f\n", temps);
  // FREE
  free(solus);
  free(b);
  for (i = 0; i < LEN; i++) {
    free(matrix[i]);
  }
  free(matrix);
  return 0;
}
```

./prog/gauss_seidel.c