

Figure 1: logo uca

TP

Algorithmique Numérique

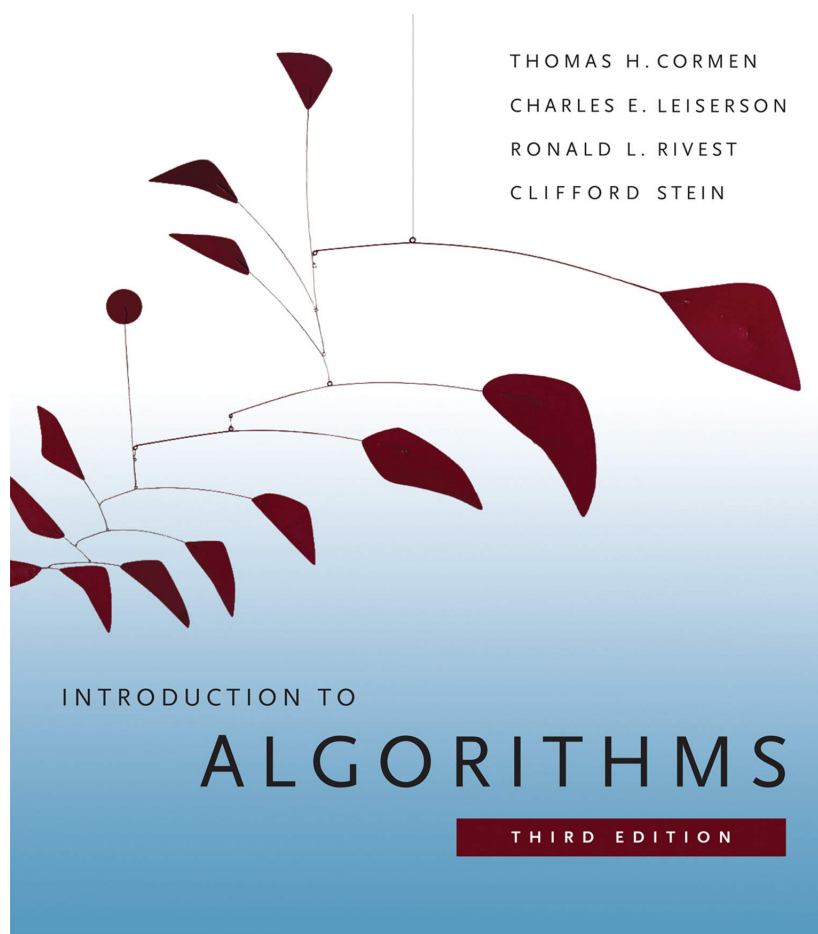


Figure 2: Cormen

SABIER Corentin et CHASSAGNOL Rémi
année 2020-2021

Professeur: CHORFI Amina

24 OCTOBRE 2020

Table des matières

1	Algorithmiques directes	2
1.1	Gauss	2
1.1.1	Principe	2
1.1.2	L'Algorithme du pivot de Gauss	2
1.1.3	Les fonctions	2
1.1.4	Les tests	4
1.2	Cholesky	6
1.2.1	Principe	6
1.2.2	L'Algorithme	6
1.2.3	Les Variables	6
1.2.4	Calcule de la matrice R	7
1.2.5	Résolution des sous-systèmes	7
1.2.6	Les Tests	8
1.3	Conclusion sur les méthodes de résolution directes	9
2	algorithmes itératifs	9
2.1	Jacobi	10
2.1.1	Principe	10
2.1.2	L'algorithme	10
2.1.3	Implémentation en C	10
2.2	Gauss-seidel	11
2.2.1	Le principe	11
2.2.2	L'algorithme	11
2.2.3	Le programme en C	11
2.3	Comparaison des résultats	12
2.3.1	Matrice symétrique à diagonale dominante	12
2.3.2	Autre système	13
2.3.3	Une Erreur	13
2.4	Conclusion sur les méthodes itératives	14

1 Algorithmes directs

1.1 Gauss

1.1.1 Principe

Le principe de cette méthode est de manipuler une matrice \mathbf{A} de coefficients du système linéaire, de sorte 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 à p :
3      Si il existe i ∈ [k+1, n] tel que aij ≠ 0 alors :
4
5          k := k + 1;
6          q := choix(i ∈ [k, n]) tel que aij ≠ 0;
7          Lq ←  $\frac{1}{a_{qj}}$  Lq;
8
9          Si k ≠ q alors :
10             Lk ↔ Lq;
11
12         Pour i allant de k+1 à n;
13             Li ← Li - aij Lk;
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 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 "*soustractionLignes*")
- La résolution du système, une fois la matrice échelonnée (fonction "*resolution*")

```

1  void normalLigne(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 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 parcourt la matrice **A** depuis 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'exécution : 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-dessus. 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'exécution : 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-dessus. 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 obtient:

```

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'exécution : 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-dessus. 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 obtient:

```

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 telle 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 à  $n$ :
2       $s = a_{ii} - \sum_{j=1}^{i-1} r_{ji}^2$ 
3      Si  $s \leq 0$  alors:
4          arret("la matrice n'est pas définie positive")
5      Sinon
6           $r_{ii} = \sqrt{s}$ 
7          Pour  $j$  allant de  $i+1$  à  $n$ :
8               $r_{ij} = \frac{a_{ij} - \sum_{k=1}^{i-1} r_{ki} r_{kj}}{r_{ii}}$ 

```

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

1.2.3 Les Variables

Dans la fonction *main* on initialise une 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 Calcul 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 renvoie une erreur si la matrice en entrée n'est pas définie positive ce qui arrête le programme car le système n'est pas résoluble avec cette méthode (en fait si la matrice n'est pas définie positive, la matrice R n'existe pas).

```

1 void make_R(float **matrix, float **r, int n) {
2     int i, j, k;    // loop var
3     float sum = 0; // Used to calculate the sum of r_ki*r_kj
4     float s;
5     for (i = 0; i < n; i++) {
6         s = matrix[i][i];
7         for (j = 0; j < i; j++) {
8             s -= r[j][i] * r[j][i];
9         }
10        if (s <= 0) {
11            printf("La matrice n'est pas d finie positive\n");
12            exit(EXIT_FAILURE);
13        } else {
14            r[i][i] = sqrtf(s);
15            for (j = (i + 1); j < n; j++) {
16                sum = 0;
17                for (k = 0; k < i; k++) {
18                    sum += r[k][i] * r[k][j];
19                }
20                r[i][j] = (matrix[i][j] - sum) / r[i][i];
21            }
22        }
23    }
24 }
```

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 forcément égaux à 1 d'où la division par a_{ii} . De plus, dans le premier sous-système, la matrice R^T est triangulaire inférieure, la résolution se fait donc en commençant par calculer la valeur de la première inconnue.

```

1 void solver_cholesky(float **matrix, float *solus_x, float *b, int n) {
2     int i, j;
3     float tmp;
4     float *solus_y = malloc(n * sizeof(float));
5     /*Creation of the matrix R:*/
6     float **r = create_mat(n);
7     mat_0(r, n);
8     make_R(matrix, r, n);
9     /*Resolution de R^T * y = b*/
10    transpose(r, n);
11    for (i = 0; i < n; i++) {
12        tmp = b[i];
13        for (j = 0; j < i; j++) {
14            tmp -= solus_y[j] * r[i][j];
15        }
16        solus_y[i] = tmp / r[i][i];
17    }
18    /*Resolution R * x = y*/
19    transpose(r, n);
20    for (i = (n - 1); i >= 0; i--) {
21        for (j = (n - 1); j > i; j--) {
22            solus_y[i] -= solus_x[j] * r[i][j];
23        }
24        solus_x[i] = solus_y[i] / r[i][i];
25    }
26    /*FREE*/
27    free(solus_y);
28    for (i = 0; i < n; i++) {
29        free(r[i]);
30    }
```

```

30     }
31     free(r);
32 }

```

Cette fonction fait appel 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 transposée):

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

Nous résolvons $Ry = b$ donc:

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

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 travaille avec des matrices de grande 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

```

Le 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 à diagonale dominante donc la méthode converge. La solution en valeur exacte est : $x_1 = \frac{-53}{1152} \approx -0.4600694444$, $x_2 = \frac{5}{72} \approx 0.0694444444$ $x_3 = \frac{7}{144} = 0.0486111111$

Ce qui est très proche 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 tests 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 moins 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 incohérent 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 faussent le résultat de cette méthode) et est stable aux écarts 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 dimension.

La méthode de Choleski a une complexité deux fois plus basse que celle de Gauss (donc deux fois plus rapide en théorie), de plus elle est stable aux écarts de valeurs. Cependant elle est limitée 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 les 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 inférieurs de A et $-F$ est une matrice triangulaire supérieure composée des éléments supérieurs de A .

Exemple

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

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 tour 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 choisi arbitrairement qui sert de point de départ. L'algorithme s'arrête quand la solution trouvée est convenable $\|x^{(k+1)} - x^{(k)}\| \leq \varepsilon$ (ou ε est la précision choisie) ou quand il a fait un nombre de tours pré-déterminé (pour éviter les boucles infinies si la méthode ne converge pas).

2.1.2 L'algorithme

```

1  Tant que  $\varepsilon^{(k)} \geq \varepsilon$ :
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 $(1, \dots, 1)$. La fonction qui calcule les solutions est la suivante:

```

1  void jacobi(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 < n; j++) {
10                 if (j != i)
11                     solus_k1[i] -= matrix[i][j] * solus_k[j];
12             }
13             solus_k1[i] /= matrix[i][i];
14         }
15         bool = test(solus_k1, solus_k, n);
16         for (i = 0; i < n; i++) {
17             solus_k[i] = solus_k1[i];
18         }
19         compt++;
20     }
21     printf("Nb tours: %d\n", compt);
22     // FREE
23     free(solus_k1);
24 }

```

Cette fonction utilise le même principe que l'algorithme ci dessus, elle fait appel à une fonction *test* qui retourne $\|x^{(k+1)} - x^{(k)}\| \leq \varepsilon$ 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 utilise les solutions $x^{(k+1)}$ au fur et à mesure qu'elle les calcule, 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:

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

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

```

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 se traduit par 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 $b - \sum_{j=1}^{i-1} a_{ij}x_j^{(k+1)}$ 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 cet 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 est 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 est bien gérée 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 contraintes requises par les méthodes, celles-ci tournent à l'infini sans jamais converger vers la solution du système (le programme s'arrête car la fonction

est limitée 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
156643975219732962746975649792.000000
124387246495582043800237768704.000000
57940990558972420295829225472.000000
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: $x_1 = \frac{76}{3} \approx 25,33333333$, $x_2 = \frac{-8}{3} \approx -2,66666667$ et $x_3 = \frac{-31}{3} \approx 10,33333333$

2.4 Conclusion sur les méthodes itératives

Le principal défaut de ces méthodes est qu'elles ne convergent 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 trouveront pas de solutions au système et boucleront 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 calcul 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 demandée dans le programme qui est limitée par la précision des flottants et la mémoire de l'ordinateur. À noter que plus la précision demandée est élevée, plus le temps d'exécution sera grand, cependant, en théorie, sans prendre en compte les faiblesses de la machine, si les méthodes convergent, on peut obtenir une précision infinie.

Gauss

```

1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <math.h>
4 #include <time.h>
5 #include "matrix.h"
6
7 int iteration; //variables compteur du nombres d'op rations sur la matrice.
8 //
9 ////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
10 void normalLigne      (float ** A, float * B, int n, int q, int j);
11 void echangeLigne     (float ** A, float * B, int n, int q, int k);
12 void soustractionLignes(float ** A, float * B, int n, int i, int j, int k);
13 void gauss            (float ** A, float * B, int n);
14 float * resolution    (float ** A, float * B, int n);
15 //
16 ////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
17
18 int main ()
19 {
20     iteration = 0;
21     int i,j,n;
22
23     printf("(GaussPivot)Taille matrices \n");
24     scanf("%d",&n);
25
26     float Tdep = clock();
27     float * B = (float * ) malloc(n*sizeof(float ));
28
29     //Initialisation de la matrice B.
30     for (i = 0; i < n; i++)
31     {
32         B[i] = 1;
33     }
34
35     float ** A = create_mat(n);
36     mat(A, n, 2);
37
38     //affichage matrice
39     for (i = 0; i<n; i++)
40     {
41         for(j = 0; j<n;j++)
42         {
43             printf("%f ", A[i][j]);
44         }
45         printf("| %f", B[i]);
46         printf("\n");
47     }
48
49     gauss(A,B,n);
50     printf("\n");
51
52     //Affichage matrice fin gauss
53     for (i = 0; i<n; i++)
54     {
55         for(j = 0; j<n;j++)
56         {
57             printf("%f ", A[i][j]);
58         }
59         printf("| %f", B[i]);
60         printf("\n");
61     }
62
63     //affichage du temps d'exécution et de la complexité pratique.
64     float Tfin = clock();
65     printf("\nNombres d'iterations : %d\n", iteration );
66     printf("Temps d' xécution : %f", (Tfin - Tdep)/1000);
67
68     // 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
81     free(B);free(solution);
82     for (int i = 0; i < n; i++)
83     {
84         free(A[i]);
85     }
86     free(A);
87     return 0;
88 }
89 //
90
91 void normalLigne(float ** A, float * B, int n, int q, int j)
92 {
93     int ji;
94     float tmp = A[q][j];
95     // Division de chaque lments de la ligne q de matrice A & B: "normalisation de la
96     // ligne"
97     for(ji = q;ji < n;ji++)
98     {
99         iteration++;
100         A[q][ji] = A[q][ji] / tmp;
101     }
102     B[q] = B[q]/tmp;
103 }
104 void echangeLigne(float ** A, float * B, int n, int q, int k)
105 {
106     float tmp;
107     int j;
108     for (j = 0; j<n ;j++)
109     {
110         iteration++;
111         tmp = A[k][j];
112         A[k][j] = A[q][j];
113         A[q][j] = tmp;
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' existence du
142         prochain pivot
143         {
144             if(A[i][j]!=0) {test = 1; break;}
145         }
146         if (test) //Si teste concluant..
147         {
148             k++; //MAJ de l'indice de ligne du
149             pivot //r cup ration de la ligne
150             q = i; //normalisation de la ligne
151             du prochain pivot(trouv avec le teste) //mise du pivot au bon
152             normalLigne(A,B,n,q,j);
153             if (q!=k) echangeLigne(A,B,n,q,k);
154             endroit
155             for (i = k+1; i < n; i++) // limination des valeurs
156             sous le pivot
157             {
158                 soustractionLignes(A, B, n, i, j, k);
159             }
160         }
161     }
162
163 float * resolution(float ** A, float * B, int n)
164 {
165     float * X = (float *)malloc(n*sizeof(float));
166     float somme;
167
168     for (int i = n-1; i>=0; i--) //indice de la ligne
169     {
170         iteration++;
171         somme = 0;
172         for (int j = n-1; j>i; j--) //indice de la colonne
173         {
174             somme += A[i][j]*X[j];
175         }
176         X[i] = (B[i]-somme);
177     }
178     return X;
179 }

```

./prog/gauss.c

Cholesky

```

1  #include "matrix.h"
2  #include <math.h>
3  #include <stdio.h>
4  #include <stdlib.h>
5  #include <time.h>
6  #define LEN 4
7  #define EXAMPLE 1
8
9  int nbtours;
10
11 void make_R(float **matrix, float **r, int n) {
12     int i, j, k;    // variables de boucles
13     float sum = 0; // Utilise pour calculer la somme des r_ki*r_kj
14     float s;
15     for (i = 0; i < n; i++) {
16         s = matrix[i][i];
17         for (j = 0; j < i; j++) {
18             s -= r[j][i] * r[j][i];
19             nbtours++;
20         }
21         if (s <= 0) {
22             printf("La matrice n'est pas d finie positive\n");
23             exit(EXIT_FAILURE);
24         } else {
25             r[i][i] = sqrtf(s);
26             for (j = (i + 1); j < n; j++) {
27                 sum = 0;
28                 for (k = 0; k < i; k++) {
29                     sum += r[k][i] * r[k][j];
30                     nbtours++;
31                 }
32                 r[i][j] = (matrix[i][j] - sum) / r[i][i];
33             }
34         }
35     }
36 }
37
38 // transpose la matrice en entree
39 void transpose(float **matrix, int n) {
40     int i, j;
41     float tmp;
42     for (i = 0; i < n; i++) {
43         for (j = (i + 1); j < n; j++) {
44             tmp = matrix[i][j];
45             matrix[i][j] = matrix[j][i];
46             matrix[j][i] = tmp;
47             nbtours++;
48         }
49     }
50 }
51
52 void solver_cholesky(float **matrix, float *solus_x, float *b, int n) {
53     int i, j;
54     float tmp;
55     float *solus_y = malloc(n * sizeof(float));
56     /*Creation de la matrice R:*/
57     float **r = create_mat(n);
58     mat_0(r, n);
59     make_R(matrix, r, n);
60     /*Resolution de R^T * y = b*/
61     transpose(r, n);
62     for (i = 0; i < n; i++) {
63         tmp = b[i];
64         for (j = 0; j < i; j++) {
65             tmp -= solus_y[j] * r[i][j];
66             nbtours++;
67         }
68         solus_y[i] = tmp / r[i][i];
69     }
70     /*Resolution de R * x = y*/

```

```

71     transpose(r, n);
72     for (i = (n - 1); i >= 0; i--) {
73         for (j = (n - 1); j > i; j--) {
74             solus_y[i] -= solus_x[j] * r[i][j];
75             nbtours++;
76         }
77         solus_x[i] = solus_y[i] / r[i][i];
78     }
79     /*FREE*/
80     free(solus_y);
81     for (i = 0; i < n; i++) {
82         free(r[i]);
83     }
84     free(r);
85 }
86
87 void show(float **matrix, int n){
88     int i, j;
89     for (i = 0; i < n; i++) {
90         printf(" | ");
91         for (j = 0; j < n; j++) {
92             printf(" %f | ", matrix[i][j]);
93         }
94         printf("\n");
95     }
96 }
97
98 int main() {
99     nbtours = 0;
100    // Time
101    float temps;
102    clock_t t1, t2;
103    int i;
104    float *solus = malloc(LEN * sizeof(float));
105    if (solus == NULL)
106        exit(EXIT_FAILURE);
107    float *b = malloc(LEN * sizeof(float));
108    if (b == NULL)
109        exit(EXIT_FAILURE);
110    for (i = 0; i < LEN; i++) {
111        b[i] = 1;
112    }
113    float **matrix = create_mat(LEN);
114    mat(matrix, LEN, EXAMPLE);
115    show(matrix, LEN);
116    t1 = clock();
117    solver_cholesky(matrix, solus, b, LEN);
118    t2 = clock();
119    // Affiche les solutions
120    for (i = 0; i < LEN; i++) {
121        printf("%f\n", solus[i]);
122    }
123    printf("nb it ration: %d\n", nbtours);
124    temps = (float)(t2 - t1) / CLOCKS_PER_SEC;
125    printf("temps = %f\n", temps);
126    /*FREE*/
127    free(solus);
128    free(b);
129    for (i = 0; i < LEN; i++) {
130        free(matrix[i]);
131    }
132    free(matrix);
133    return 0;
134 }

```

./prog/cholesky.c

Jacobi

```

1  #include "matrix_tp2.h"
2  #include <math.h>
3  #include <stdio.h>
4  #include <stdlib.h>
5  #include <time.h>
6  #define LEN 3
7  #define SIGMA 0.0000001
8  #define MAX 100
9  #define EXAMPLE 4
10
11 int test(float *solus_k, float *solus_k1, int n) {
12     int i;
13     float *tmp = malloc(n * sizeof(float));
14     float s_quadra = 0;
15     for (i = 0; i < n; i++) {
16         tmp[i] = solus_k1[i] - solus_k[i];
17         s_quadra += tmp[i] * tmp[i];
18     }
19     // FREE
20     free(tmp);
21     return (sqrtf(s_quadra) > SIGMA);
22 }
23
24 void jacobi(float **matrix, float *b, float *solus_k, int n) {
25     int i, j;
26     int compt = 0;
27     int bool = 1;
28     float *solus_k1 = malloc(n * sizeof(float));
29     while (bool && compt < MAX) {
30         for (i = 0; i < n; i++) {
31             solus_k1[i] = b[i];
32             for (j = 0; j < n; j++) {
33                 if (j != i)
34                     solus_k1[i] -= matrix[i][j] * solus_k[j];
35             }
36             solus_k1[i] /= matrix[i][i];
37         }
38         bool = test(solus_k1, solus_k, n);
39         for (i = 0; i < n; i++) {
40             solus_k[i] = solus_k1[i];
41         }
42         compt++;
43     }
44     printf("Nb tours: %d\n", compt);
45     // FREE
46     free(solus_k1);
47 }
48
49 void show(float **mat, int n){
50     int i, j;
51     for (i = 0; i < n; i++) {
52         printf("|");
53         for (j = 0; j < n; j++) {
54             printf(" %f |", mat[i][j]);
55         }
56         printf("\n");
57     }
58 }
59
60 int main() {
61     float temps;
62     clock_t t1, t2;
63     int i;
64     // init b
65     float *b = malloc(LEN * sizeof(float));
66     if (b == NULL)
67         exit(EXIT_FAILURE);
68     // init solus
69     float *solus = malloc(LEN * sizeof(float));
70     if (solus == NULL)

```

```
71     exit(EXIT_FAILURE);
72     for (i = 0; i < LEN; i++)
73         solus[i] = 0;
74     // init matrix
75     float **matrix = create_mat(LEN);
76     // Initialise matrix et b en fonction de l'exemple choisit
77     init_mat(matrix, b, LEN, EXAMPLE);
78     printf("\n");
79     show(matrix, LEN);
80     t1 = clock();
81     jacobi(matrix, b, solus, LEN);
82     t2 = clock();
83     for (i = 0; i < LEN; i++) {
84         printf("%f\n", solus[i]);
85     }
86     temps = (float)(t2 - t1) / CLOCKS_PER_SEC;
87     printf("temps = %f\n", temps);
88     // FREE
89     free(solus);
90     free(b);
91     for (i = 0; i < LEN; i++) {
92         free(matrix[i]);
93     }
94     free(matrix);
95     return 0;
96 }
```

./prog/jacobi.c

Gauss Seidel

```

1  #include "matrix_tp2.h"
2  #include <math.h>
3  #include <stdio.h>
4  #include <stdlib.h>
5  #include <time.h>
6  #define LEN 3
7  #define SIGMA 0.0000001
8  #define MAX 100
9  #define EXAMPLE 4
10
11 int test(float *solus_k, float *solus_k1, int n) {
12     int i;
13     float *tmp = malloc(n * sizeof(float));
14     float s_quadra = 0;
15     for (i = 0; i < n; i++) {
16         tmp[i] = solus_k1[i] - solus_k[i];
17         s_quadra += tmp[i] * tmp[i];
18     }
19     // FREE
20     free(tmp);
21     return (sqrtf(s_quadra) > SIGMA);
22 }
23
24 void gauss_seidel(float **matrix, float *b, float *solus_k, int n) {
25     int i, j;
26     int compt = 0;
27     int bool = 1;
28     float *solus_k1 = malloc(n * sizeof(float));
29     while (bool && compt < MAX) {
30         for (i = 0; i < n; i++) {
31             solus_k1[i] = b[i];
32             for (j = 0; j < i; j++) {
33                 solus_k1[i] -= matrix[i][j] * solus_k1[j];
34             }
35             for (j = i + 1; j < n; j++) {
36                 solus_k1[i] -= matrix[i][j] * solus_k[j];
37             }
38             solus_k1[i] /= matrix[i][i];
39         }
40         bool = test(solus_k1, solus_k, n);
41         for (i = 0; i < n; i++) {
42             solus_k[i] = solus_k1[i];
43         }
44         compt++;
45     }
46     printf("Nb tours: %d\n", compt);
47     // FREE
48     free(solus_k1);
49 }
50
51 void show(float **mat, int n){
52     int i, j;
53     for (i = 0; i < n; i++) {
54         printf("|");
55         for (j = 0; j < n; j++) {
56             printf(" %f |", mat[i][j]);
57         }
58         printf("\n");
59     }
60 }
61
62 int main() {
63     float temps;
64     clock_t t1, t2;
65     int i;
66     // init b
67     float *b = malloc(LEN * sizeof(float));
68     if (b == NULL)
69         exit(EXIT_FAILURE);
70     // init solus

```

```
71 float *solus = malloc(LEN * sizeof(float));
72 if (solus == NULL)
73     exit(EXIT_FAILURE);
74 for (i = 0; i < LEN; i++)
75     solus[i] = 0;
76 // alloc matrix
77 float **matrix = create_mat(LEN);
78 // Initialise matrix en fonction de l'exemple choisit
79 init_mat(matrix, b, LEN, EXAMPLE);
80 printf("\n");
81 show(matrix, LEN);
82 t1 = clock();
83 gauss_seidel(matrix, b, solus, LEN);
84 t2 = clock();
85 for (i = 0; i < LEN; i++) {
86     printf("%f\n", solus[i]);
87 }
88 temps = (float)(t2 - t1) / CLOCKS_PER_SEC;
89 printf("temps = %f\n", temps);
90 // FREE
91 free(solus);
92 free(b);
93 for (i = 0; i < LEN; i++) {
94     free(matrix[i]);
95 }
96 free(matrix);
97 return 0;
98 }
```

./prog/gauss_seidel.c