FIGURE 1: ISIMA

FIGURE 2: Université de Clermont-Ferrand

Compte Rendu Algorithme Numérique

Gauss, Choleski, Jacobi et Gauss-Seidel

Réalisé par : Théo Cliquot et Valérian Capon

En : Prep'ISIMA 1 (voir la figure 1 et et la figure 2)

Travail à rendre pour le : 24/10/2020

Table des matières

Table des figures

1 Choleski

J'ai seulement mit 2 chiffre après la virgule pour avoir une meilleur lisibilité dans le compte rendu et tester plus facilement les résultats ce qui peut des fois arrondir une erreur dans le calcul. Cependant cela n'empêche pas de voir quelques erreurs (comme dans Gauss Seidel)

1.1 Rappel rapide de la méthode

La méthode de choleski permet de résoudre le système $A*x=B$ (A étant une matrice carrée de taille n , x une matrice colonne inconnue que l'on essaie de trouver et B une autre matrice colonne.

Pour ce faire on simplifie la matrice A sous la forme $R^{-1}*R$ (R étant une matrice triangulaire supérieure de taille n). Ce qui permet de transformer le système en $R^{-1}*R*x=B$ ou encore $R^{-1}*y=B$ avec $R*x=y$, C'est 2 équations deviennent beaucoup plus faciles à résoudre du faites que R et R^{-1} sont triangulaires.

Cependant la méthode de Choleski n'est possible que pour les matrices défini strictement positives et symétrique, en effet si ce n'est pas le cas pour la dernière on aura une racine carrée avec une valeur négatif (ce qui est impossible sur (R)).

1.1.1 Exemple

Prenons la matrice A : $\begin{pmatrix}1&1&1\\1&2&2\\1&2&3\end{pmatrix}$

R est la matrice A après Choleski. On définit i la ligne qu'on regarde actuellement et j la colonne. Dans un premier temps avec la formule 1

$$r_{ii}=a_{i,i}-\sum_{j=1}^{i-1}r_{ji}^2 \tag{1}$$

Soit = $r_{11}=\sqrt{1}=1$ seulement si $a_{1,1}-\sum_{j=1}^{i-1}r_{ji}^2>0$ (sinon la méthode de choleski est impossible).

On a donc R = $\begin{pmatrix}1&x&x\\0&x&x\\0&0&x\end{pmatrix}$

Pour tout les autres elements sur la même ligne qui suivent on applique la formule 2 :

$$r_{i,j}=(a_{ij}-\sum_{k=1}^{i-1}r_{ki}*r_{kj})/r_{ii} \tag{2}$$

Ainsi : $r_{12}=1$ et $r_{13}=1$ Ce qui nous donne pour R : $\begin{pmatrix}1&1&1\\0&x&x\\0&0&x\end{pmatrix}$

Si on continue à appliquer c'est 2 méthodes ligne par ligne on obtindra à la fin R = $\begin{pmatrix}1&1&1\\0&1&1\\0&0&1\end{pmatrix}$

1.2 Présentation du programme (fonction dans A)

La fonction `Choleski()` va suivre le même principe que vu au dessus. Pour cela elle va prendre en argument une tableau 2D de float *matA* de taille *taille* (taille étant aussi un argument) (dans notre exemple la matrice A) et une autre matrice 2D carrée *C1* de taille *taille* (notre matrice R dans l'exemple). cette fonction va renvoyer un entier : soit 0 ou 1. 0 représentant une erreur durant l'exécution de la méthode et 1 un bon fonctionnement du programme.

Cette fonction va dans un premier temps parcourir chaque ligne. Dans chacune de ces lignes il va appliquer les 2 formules vu juste avant. La première formule ()1) va être appliquer pour les éléments de la diagonale et à chaque fois on va vérifier si $a_{1,1} - \sum_{j=1}^{i-1} r_{ji}^2 > 0$. Dans le cas contraire on arrête l'algorithme en renvoyant 0.Si ce n'est pas le cas on continue en appliquant cette fois ci la seconde formule (2) et ceux pour tout les éléments dans les colonnes suivantes (et en restant toujours dans la même ligne), c'est ce que fait la seconde boucle `for`. Ensuite il nous suffit d'appliquer la formule tout en faisant attention qu'on ne divise pas par 0 (dans ce cas on arrête le programme et on renvoie 0).

Pour tout ce qui concerne les fonctions autour de choleski, la plupart servent à allouer dynamiquement des matrices 2D (avec malloc) ou à les libérer, à afficher des matrices 2D ou colonnes, à remplir les matrices soit aléatoirement (en respectant 70% de 0 pour la matrice creuse) soit définies (comme Hilbert,Lehmer ...).

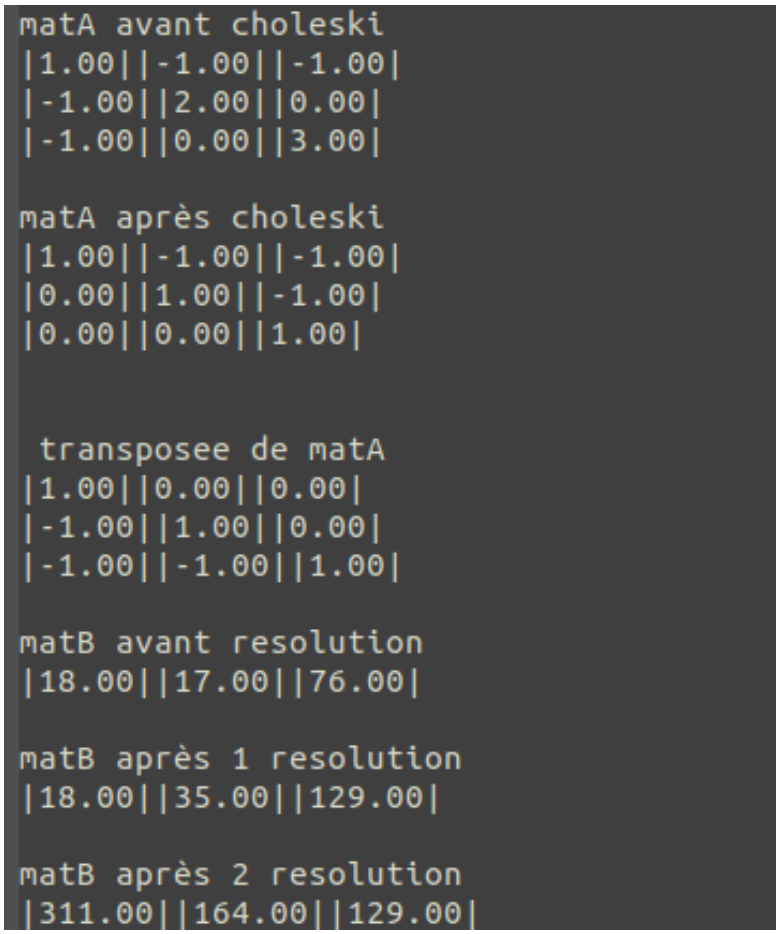
La seule fonction un peu plus complexe permet de résoudre Ax = b que l'on appelle 2 fois : une fois pour résoudre $R^{-1} * y = matB$ et la seconde $R * x = y$. Pour cela la fonction va résoudre la première ou dernière ligne du système (en fonction de si on à rentré R^{-1} ou R). Puis elle va descendre (ou remonter) en utilisant les résultats précédents.

1.3 Présentations de jeux d'essais pertinents

On observe dans les matrices test que seuls 4 d'entre elles peuvent être transformer via l'algo de choleski (les 2 Hilbert, Lehmer et Moler),

La matrice de Franc est assez intéressantes car elle pourrait être résolu par choleski si il n'existait pas la règle $a_i j = 0 sii \geq j + 2$ et elle donnerait une matrice triangulaire uniquement composé de 1 (comme la matrice de l'exemple).

Moler donne aussi une matrice après Choleski assez particulière puisque qu'elle est à ça diagonale formée de 1 et le reste de -1. De plus comme elle est bien symétrique elle résout l'équation sans aucun problème (exemple avec une matrice de taille 3 (image 3)). Dans le cas des matrices creuses l'algorithme de Choleski ne marche que dans de rare cas en effet entre le fait que la matrice généré aléatoirement n'est pas forcément symétrique, définie strictement positive ou le faite qu'on peut rencontrer pendant l'exécution de choleski une division par 0 rends l'obtention d'une matrice faisable assez rare.



311*1 - 164 – 129 = 18 ;
-311 + 2*164 = 17 ;
-311+129*3 = 76

FIGURE 3: Un exemple en utilisant la matrice de Moler

Une erreur d'arrondie peut apparaître, en effet pour une meilleure lisibilité les variables dans les matrices ont était limitait à 2 chiffres après la virgule, ainsi si le résultat s'étend sur 3 chiffre après la virgule des erreurs vont apparaître.

1.4 Commentaires des jeux d'essais

Il est assez facile de remarquer avec le temps d'exécution de choleski qu'elle n'est pas de complexité linéaire (en fonction de la taille du tableau) . Cependant on a un temps d'éxécution correctes (un peu plus d'une demi seconde pour un tableau de taille 1000 comme le montre la Figure 4c).Il a était dit dans le cours que l'algorithme de Choleski est de complexité $n^3/3$. Pour le vérifier on calcule le nombre d'opérations effectué par l'algorithme avec un tableau de taille 5 et de taille 100. On obtient pour le premier 55 opérations 338350 pour le second. ce qui est plus que attendu (41 et 333334). Cependant on reste dans une complexité temporelle avec un ordre de grandeur de n^3 (si on essaie de trouver avec n^4 on à $5^4/11.5 \simeq 55$ mais $100^4/11.5 \simeq 8695652 \neq 333334$)

```
matA avant choleski
|1.00||-1.00||-1.00||-1.00||-1.00|
|-1.00||2.00||0.00||0.00||0.00|
|-1.00||0.00||3.00||1.00||1.00|
|-1.00||0.00||1.00||4.00||2.00|
|-1.00||0.00||1.00||2.00||5.00|

Choleski c'est exécuté en 0.000069 sec
```

(a) taille 5

```
Choleski c'est exécuté en 0.002284 sec
```

(b) taille 100

```
matA avant choleski
Choleski c'est exécuté en 0.572994 sec
```

(c) taille 1000

FIGURE 4: Temps d'exécution avec des matrices de Moler de différentes tailles

On peut aussi facilement observer que la méthode de Choleski est stable nottament avec un écart de 10% (Image 5a et 5b)

```
matA avant choleski
|1.00||-1.00||-1.00|
|-1.00||2.00||0.00|
|-1.00||0.00||3.00|

matA après choleski
|1.00||-1.00||-1.00|
|0.00||1.00||-1.00|
|0.00||0.00||1.00|


 transposee de matA
|1.00||0.00||0.00|
|-1.00||1.00||0.00|
|-1.00||-1.00||1.00|

matB avant résolution
|10.00||20.00||30.00|

matB après 1 résolution
|10.00||30.00||70.00|

matB après 2 résolution
|180.00||100.00||70.00|
```

(a) matB = {10 20 30}

```
matA avant choleski
|1.00||-1.00||-1.00|
|-1.00||2.00||0.00|
|-1.00||0.00||3.00|

matA après choleski
|1.00||-1.00||-1.00|
|0.00||1.00||-1.00|
|0.00||0.00||1.00|


 transposee de matA
|1.00||0.00||0.00|
|-1.00||1.00||0.00|
|-1.00||-1.00||1.00|

matB avant résolution
|11.00||22.00||33.00|

matB après 1 résolution
|11.00||33.00||77.00|

matB après 2 résolution
|198.00||110.00||77.00|
```

(b) matB = {11 22 33}

FIGURE 5: Preuve de la stabilité pour Choleski

On a bien : $198 = 180 * 1.1; 110 = 100 * 1.1; 77 = 70 * 1.1$

1.5 Conclusion Algorithme

Choleski permet de trouver la bonne réponse comme gauss (par rapport au méthode indirectes comme Jacobi et Gauss Seidel) Cependant elle possède trois différences majeures avec Gauss. De 1 elle à une complexité moins élevé que Gauss : $n^3/3 \leq 2n^3/3$. Cependant et c'est la seconde et troisième différence elle a besoin de règles très précise pour fonctionner, en effet la matrice *matA* doit être symétrique et définie positive, Ce qui limite beaucoup les matrices possibles. De plus on est obliger de créer un autre tableau 2D pour sauvegarder R, ce qui peut être négligeable si A est de petite taille mais peut se réveler bien plus problématique si à l'inverse A est de grande taille. Il est donc assez intéressant d'utiliser choleski lorsque c'est possible et si une matrice supplémentaire de même taille que A ne va pas surcharger la mémoire.

2 Gauss-Seidel

2.1 Rappel rapide de la méthode

Gauss Seidel comme Jacobi est une méthode itérative,c'est à dire qu'elle va s'approcher de la bonne solution mais pas forcément trouver la réponse exacte du système $Ax = B$. Cependant pour cela il faut que Gauss Seidel converge, c'est à dire soit A est diagonale dominante soit a est définis strictement positive. La méthode de Gauss Seidel consiste a appliquer Jacobi mais dans chaque itérations on utilise les résultats précédent.

2.1.1 Exemple

Prenons la matrice A : $\begin{pmatrix} 5 & 2 & -1 \\ 1 & 6 & -3 \\ 2 & 1 & 4 \end{pmatrix}$ La matrice B : $\begin{pmatrix} 6 \\ 4 \\ 7 \end{pmatrix}$ Prenons ensuite comme point de départ P = $\begin{pmatrix} 0 \\ 0 \\ 0 \end{pmatrix}$ et une marge d'erreur de 0.15 (ce qui est une grosse marge surtout pour un ordinateur mais elle permet de ne pas trop faire d'itérations.)

Comme dit précédemment la méthode de Gauss Siedel suit la même logique que Jacobi. Le seul point qu'il n'ont pas en commun et le faite que Gauss Seidel va utiliser directement les nouvelles solutions trouvés dans l'itération. Ainsi prenons un systéme avec 3 inconnu $x_1; x_2$ et x_3. Pour une certaine itération k, la première formule de Jacobi et de Gauss Seidel pour calculer x_1^k vont être la même, ils vont tous les deux utiliser x_2^{k-1} et x_3^{k-1}. Cependant à partir de la seconde formule Gauss Seidel ne va non pas utiliser x_1^{k-1} comme Jacobi mais il va plutôt utiliser x_1^k, et ainsi de suite.

On peut retrouver cette différence dans l'équation de Gauss Siedel (Formule (3)) :

$$x_i^k = \frac{1}{a_{ii}} * (b_i - \sum_{j<i} a_{ij} * x_j^k - \sum_{j>i} a_{i,j} * x_j^{k-1}). \tag{3}$$

Lorsqu'on regarde la deuxième $\sum$ dans cette formule on se rends compte que c'est exactement pareil que Jacobi, la seule différence vient donc des inconnus avant i (la première $\sum$) correspondant aux variables calculé avant dans l'itération.

Ainsi si on reprends notre exemple. On obtient lors de la première itérations $P_1 = \begin{pmatrix} 1.200 \\ 0.467 \\ 1.033 \end{pmatrix}$

Seconde itérations $P_2 = \begin{pmatrix} 1.220 \\ 0.980 \\ 0.895 \end{pmatrix}$

Troisième itérations $P_3 = \begin{pmatrix} 0.987 \\ 0.950 \\ 1.019 \end{pmatrix}$

Quatrième itérations $P_4 = \begin{pmatrix} 1.024 \\ 1.006 \\ 0.9866 \end{pmatrix}$

On va regarder si notre résultats pour l'itération 4 est bien compris dans la marge d'erreur, pour ça on va regarder si $A * p_4 - B \simeq 0$

On trouve : $\begin{pmatrix} 0.145 \\ 0.100 \\ 0.000 \end{pmatrix}$

Toutes les valeurs de cette matrice sont $\leq$ à 0.15 donc notre résultats est compris de la marge d'erreur.

2.2 Présentation du programme (fonction dans B)

Comme pour Choleski, Bon nombre de fonctions sont réutilisés (allocation d'une matrice 2D, affichaged'une matrice2D ...) La seule qui est rajouté est une fonction permettant de calculer rapidement le produit matriciel entre une matrice 2D et une matrice colonne (pour calculer la marge d'erreur).

La fonction Gauss Seidel va prendre plusieurs arguments en entrés. nottament 3 matrices : une matrice carrée 2D `matA`, une matrice colonne `matB`, une autre matrice colonne `valeurActuelle` toute de taille `taille` (qui est aussi un argument). Et enfin un float `erreurAccepté` qui correspond à la marge d'erreur. Comme pour la fonction `Choleski`, Gauss Seidel retourne 1 si tout c'est bien passée et qu'il à trouver une solution respectant la marge d'erreur. Sinon il retourne 0 (nottament si il a fait trop d'itérations pour éviter une boucle infini ou qu'il y a une division par 0 à un moment). La fonction de Gauss Seidel se compose d'une boucle while qui va continuer tant que l'une de ces deux conditions n'est pas vérifié :

— La fonction à dépassé un nombre de boucle autorisé (pour éviter la boucle infinie).
— Le résultat obtenu est assez proche de la solution (dépend de la marge d'erreur).

A l'intérieur de cette boucle (qui correspond a une itération) On va appliquer la formule au dessus en écrasant à chaque fois `valeurActuelle`. Une fois qu'on à parcourue et modifié entièrement `valeurActuelle`. On réalise le même test vu dans l'exemple pour vérifier si notre résultat et compris dans la marge d'erreur et ceux en calculant $A * x$ grâce à une fonction extérieur qu'on stock dans matA_X, une matrice colonne alloué. Si oui on arrête la boucle là (2eme condition vérifié), sinon on continue. A la sortie de la boucle, un `if` sépare les 2 cas possibles pour la sortie de la boucle. Si la sortie de la boucle est dû à la première condition on arrête la function qui renvoie 1, sinon la fonction renvoie 0 car elle à finit sans problème. Dans les 2 cas on libère matA_X pour éviter de gaspiller de la mémoire.

2.3 Présentations de jeux d'essais pertinents

Il y a 3 matrices test dans les 9 présentes qui respectent les conditions de Gauss Seidel (Moler, Lehmer et Franc). Exemple avec la matrice de Lehmer de taille 3 :

```
Pour la case n°0 :1

Pour la case n°1 :1

Pour la case n°2 :1

MatA :
|1.00||0.50||0.33|
|0.50||1.00||0.67|
|0.33||0.67||1.00|

MatB :
|58.00||46.00||70.00|

|46.66||-24.53||70.80|
```

FIGURE 6: Un exemple en utilisant la matrice de Lehmer

46,66 - 24,53*1/2 + 0,33*70,80 = 57.759
46,66*0,5 – 24,53 + 0,67*70,80 = 46,236
46,66*0,33 – 24,53*0,67 + 70,80 = 69,7627

On observe que les résultats obtenu sont assez proches mais pas forcément égales. Ce qui est logiques puisque on accepte une marge d'erreur. Si on veut avoir des calculs plus précis il faudrait encore baisser notre marge d'erreur (et aussi afficher plus de chiffres après la virgule), ce qui implique plus d'itérations et donc un temps d'exécution plus long.

2.4 Commentaires des jeux d'essais

On remarque assez vite que le temps d' exécution n'est pas influencé par la taille des tableaux seulement, mais plus par le nombre d'itérations. En effet une matrice de taille 100 peut très bien prendre 1 itération (si notre point de départ est la solution) alors qu'une matrice de taille 5 va en prendre 250. Cependant la taille joue comme même un rôle car plus la matrice est grande, plus les itérations vont prendre de temps. Ainsi la complexité ne peut pas être calculé avec la taille de la matrice seulement, et de même avec seulement l'itération. (voir image (7a) et (7b))

```
Fin de Gauss-Seidel au bout de 256821 itérations
Gauss Seidel c'est exécuté en 19.597019 sec
```

(a) taille de 125

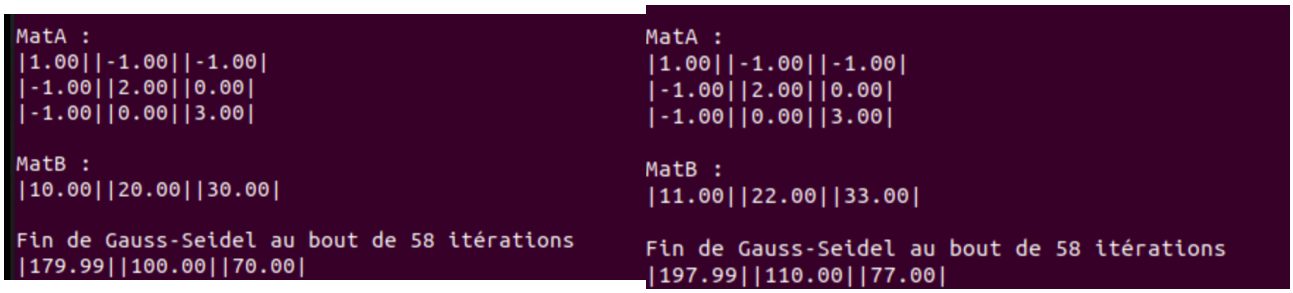
```
Fin de Gauss-Seidel au bout de 44516 itérations
Gauss Seidel c'est exécuté en 3.406529 sec
```

(b) même taille ... et pourtant un écart important

FIGURE 7: Temps exécution ainsi que le nombre d'itérations selon la taille

Si la matrice que l'on passe à l'entrée n'est pas diagonales dominantes où défini strictement positive, la méthode de Gauss Seidel ne va jamais converger ce qui va entraîner dans notre algorithme une fin de boucle non pas car on à trouver une solution mais pour éviter une boucle infini. On va aussi arrêter notre fonction si on divise par 0. Ces 2 critères rendent beaucoup de matrices creuses invalides.

En ce qui concerne la stabilité on remarque aussi assez rapidement que Gauss Siedel est stable (voir image (8a) et (8b))



(a) matB = {10 20 30}

(b) matB = {11 22 33}

FIGURE 8: Preuve de la stabilité pour Gauss Siedel

Si on ne prends pas en compte l'erreur au centième pour la première case de matB ($179.99 \simeq 180.00$ et $197.99 \simeq 198.00$) on remarque tout de suite que c'est le même résultat que choleski. Donc Gauss Seidel est très très proche d'être stable (pour ne pas dire totalement).

2.5 Conclusion Algorithme

Gauss Seidel converge plus rapidement que Jacobi et il est plus large dans ses possibilités que Jacobi. Cependant on n'est pas sûr de trouver la réponse exacte (elle peut être très proche mais à 0,0.....01 près) ce qui est un désavantage par rapport au méthode non itérative tel Gauss ou Choleski. Cependant en ce qui concerne qui est le plus rapide entre les méthodes itératives et les directes est assez flou et on ne peut pas être sûr à 100%.

En ce qui concerne les matrices creuses. La plupart ne sont pas valides car la fonction Gauss Seidel va diviser par 0 au bout d'un moment, si jamais il n'y a aucune division il faut encore que la matrice `matA` soit à diagonale dominante ou défini strictement positive, ce qui limite encore plus les choix.

A Fonction Choleski

```
// Cette fonction applique la méthode de choleski sur la marice
 matA en enregistre le réultat dans la matrice C1
// Renvoie 1 (true) si l'algorithme c'est passé sans aucun
 problème, 0 (false) sinon.
int Choleski(float ** matA,float ** C1, int taille)
{

  /* matA correspond à notre matrice A dans le système Ax = B
  C1 est la matrice obtenue après que Choleski est travaillé
 avec matA
  taille représente la longueur de la matrice carrée A et C1 */

  for (int i = 0; i < taille; i++) //On parcours chaque ligne de
 la matrice matA
  {
    float t = matA[i][i];
    for(int l = 0; l<i ; l++)
    {
      t -= pow(C1[l][i],2);
    }
    // Si t inférieure à 0, alors on renvoie 0 (erreur);
    if(t <= 0)
    {
      printf("Algorithme de Choleski impossible\n");
      return 0;
    }
    // Sinon on peut continuer d'appliquer Choleski sans problème
    else
    {
      C1[i][i] = sqrtf(t);
      for(int j = i+1;j < taille;j++)
      {
        float s=0;
        for (int l = 0; l < i; l++)
        {
          s += C1[l][i] * C1[l][j] ;
        }
        // De même si on divise par 0 Choleski renvoie une erreur
        if(C1[i][i] == 0)
        {
          printf("Division par 0\n");
          return 0;
        }
        else
        {
          C1[i][j] = (matA[i][j] - s)/C1[i][i];
        }
```

```
        }
      }
    }
    return 1;
  }
```

B Fonction Gauss-Siedel

```
// Cette fonction calcul la solution de matA*X = matB en fontion
   d'une erreurAccepté et d'un tableau contenant les valeurs de
   recherches, puis stock le résultat dans ce dernier tableau.
// Renvoie 1 (true) si la fonction n'a rencontré aucun problème 0
   (false) sinon avec un message d'erreur
int Gauss_Seidel(float  ** matA, float* matB, int taille, float
   erreurAccepte, float* valeurActuelle)
{

  /* matA et matB sont les matrices qui écrivent l'équation Ax = B
   (matA = A, matB = B) matA étant une matrice Carrée et matB une
   matrice colonne de longueur taille.
     erreurAccepté est la valeur influant sur la marge d'erreur
     valeuActuelle et le point de départ, à chaque itérations
   valeurActuelle se rapprochera de x (la solution) */


  // matA-X et le produit entre matA et notre résultat actuelle,
   cette matrice va permettre de confronter les résultats obtenu
   avec les vrais résultats
  float * matA_X = malloc(sizeof(float)*taille);

  // les 2 int qui suivent servent comme test d'arrêt, stop permet
   d'arrêter la boucle au bout de 100 itérations et valeurProche
   test si nos résultats sont cohérents (avec une marge d'erreur
   défini par erreurAccepte)
  int stop = 0;
  int valeurProche = 0;

  // Tant que l'un de nos test d'arrêt n'est pas vérifié.
  while (stop < 1000000000 && valeurProche == 0 )
    {

      valeurProche = 1;
      //Pour chaque ligne du système
      for(int i = 0; i < taille;i++)
    {

      // Permet d'éviter les divisions par 0.
```

```
      if(matA[i][i] == 0)
        {
          printf("Division par 0\n");
          // On libère matA_X puisque on en a plus besoin
          free(matA_X);
          return 0;
        }

      float s = 0;

      for (int j = 0; j < taille;j++)
        {
          if( i != j)
        {
          s -= matA[i][j] * valeurActuelle[j];
        }
        }
      valeurActuelle[i] = (matB[i] + s)/matA[i][i] ;
    }

      produitDeAParX(matA_X,matA,valeurActuelle,taille);

      // Cette boucle vérifie si notre résultat et proche de celui
   attendu (1 si oui 0 sinon)
      for(int i = 0; i < taille;i++)
    {
      if (fabs(matA_X[i] - matB[i]) > erreurAccepte)
        {
          valeurProche = 0;
        }
    }
      stop++;
    }

  // On libère matA_X puisque on en a plus besoin
  free(matA_X);

  // On retourne faux si on arrête la boucle non pas car le
   résultat est proche mais à cause du nombre d'itérations
   important
  if(stop == 1000000000)
    {
      printf("Arrêt de la boucle par sécurité\n");
      return 0;
    }
  printf("Fin de Gauss-Seidel au bout de %d itérations\n",stop);
  return 1;
}
```

```
void sauvegarde(taquin t)
{
  int num = 0;
  char nomSauvegarde[12];
  FILE * f;
  printf("Sauvegarde en cours \n");
  sprintf(nomSauvegarde,"sauv_%d.txt",num);
  while((f = fopen(nomSauvegarde, "r")))
  {
    fclose(f);
    num++;
    sprintf(nomSauvegarde,"sauv_%d.txt",num);
  }
  fclose(f);
  f = fopen(nomSauvegarde,"w+");
  if(f == NULL)
  {
    perror("ERREUR CREATION FICHIER");
  }
  printf("Sauvegarde du taquin dans le fichier :
 %s\n",nomSauvegarde);
  for (int i = 0;i<t.taille*t.taille;i++)
  {

    fprintf(f,"%d",t.tab[i/t.taille][i%t.taille]);
    if(i%t.taille != t.taille-1)
    {
      fputc(' ',f);
    }
    else
    {
      fputc('\n',f);
    }
  }
  fclose(f);
}
```