Algorithmes numériques

Résolution de systèmes linéaires : Méthodes directes

Dans ce dossier, nous allons vous présenter une implémentation en langage C des méthodes de Gauss et de Cholesky, méthodes directes pour la résolution de systèmes linéaires. Nous étudierons également leurs complexité, leurs stabilité et leur utilisation dans des cas pratiques judicieusement choisis. Nous conclurons par un face à face de ces deux méthodes pour déterminer laquelle privilégier.

Dossier et codes réalisés par :

BAYLE Nathanaël
CORCOS Ludovic

L2 Informatique
Université Clermont-Auvergne

Table des matières

I - Introduction

Nous nous intéressons dans ce dossier à des méthodes de résolution d'un système linéaire $A\,x = b$ où A est une matrice donnée, b un vecteur donné et x l'inconnue du problème. Sauf mention contraire, A est carrée inversible de taille $n \times n$.

Une première méthode d'inversion du système est la considération des formules de Cramer :

$$x_i = \frac{\det\left(a_1 \cdots a_{i-1} \quad b \quad a_{i+1} \cdots a_n\right)}{\det(A)}$$

En procédant par développement selon les lignes ou les colonnes, le nombre d'opérations pour calculer un déterminant d'ordre n est $\geq n!$.On doit ici calculer $(n+1)$ déterminants.
Considérons une matrice 20 × 20 (qui est une très petite matrice par rapport à ce qu'il peut exister). Selon Wikipédia, avec une complexité en $O(n!)$ il faudrait approximativement 770 ans pour résoudre ce système !

Les méthodes que nous allons étudier sont des méthodes de résolution qui ont un coût de calcul en $O(n^3)$. Avec la matrice précédente, de taille 20 × 20, le système se résout alors par ces techniques en quelques dix-millièmes de seconde.

II - Rappel des méthodes

a) Méthode de Gauss

On veut résoudre $A\,x = b$. A est carrée de taille $n \times n$. Le principe est simple, on simplifie le système pour le rendre triangulaire supérieur par utilisation de combinaison linéaires de lignes (c'est-à-dire d'équations).
Elle permet le calcul d'une solution exacte en un nombre fini d'étapes (Méthode de Gauss simple). Des problèmes d'accumulation d'erreurs au cours de la résolution peuvent affecter la solution (Méthode de Gauss avec pivot).

On peut représenter l'algorithme du pivot de Gauss de manière simple et générale :

Etape 1 - Choix du pivot :

On choisit un "pivot", c'est-à-dire l'un des monômes du système. Le premier pivot est le premier monôme de la première ligne, le second monôme de la seconde ligne et ainsi de suite.

Etape 2 - Elimination :

On soustrait aux lignes suivantes la ligne du pivot un nombre suffisant de fois pour que tous les termes en x, en y, etc… puissent s'annule.

Etape 3 - Retour à l'étape 1 :

Rebelote mais avec le pivot suivant.

Fin de l'algorithme :

Lorsqu'on a atteint le n-ième coefficient de la n-ième ligne.

Lorsqu'on a atteint un pivot nul (le système n'admet pas de solution unique).

Représentation de l'algorithme en pseudo-code pour une matrice de taille $n \times p$:

k := 0;
Pour j = 1 jusqu'à p
    S'il existe $i \in [k+1, n]$ tel que $a_{ij} \neq 0$ alors
        k := k+1;
        q := choix ($i \in [k, n]$ tel que $a_{ij} \neq 0$);
        $L_q \leftarrow (\frac{1}{a_{qj}}) L_q$;
        $L_k \leftrightarrow L_q$;
        Pour i = k + 1 jusqu'à n;
            $L_i \leftarrow L_i - a_{ij} L_k$;
        Fin Pour
    Fin Si
Fin Pour

On peut détailler un exemple concret sur les matrices suivantes tel que $Ax = B$

$$A = \begin{pmatrix} 1 & 0 & 1 \\ 0 & 2 & -1 \\ -1 & 1 & -2 \end{pmatrix} \quad x = \begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix} \quad et \quad B = \begin{pmatrix} 2 \\ 1 \\ -2 \end{pmatrix}$$

On écrit la **matrice augmentée**, constituée de la matrice A et du second membre B.

$$\tilde{A} = \begin{bmatrix} 1 & 0 & 1 & 2 \\ 0 & 2 & -1 & 1 \\ -1 & 1 & -2 & -2 \end{bmatrix}$$

$$\begin{bmatrix} 1 & 0 & 1 & 2 \\ 0 & 2 & -1 & 1 \\ -1 & 1 & -2 & -2 \end{bmatrix} \xrightarrow{L_3 \leftarrow L_3 + L_1} \begin{bmatrix} 1 & 0 & 1 & 2 \\ 0 & 2 & -1 & 1 \\ 0 & 1 & -1 & 0 \end{bmatrix}$$

$$\begin{bmatrix} 1 & 0 & 1 & 2 \\ 0 & 2 & -1 & 1 \\ 0 & 1 & -1 & 0 \end{bmatrix} \xrightarrow{L_2 \leftarrow L_2 - L_3} \begin{bmatrix} 1 & 0 & 1 & 2 \\ 0 & 1 & 0 & 1 \\ 0 & 1 & -1 & 0 \end{bmatrix} \xrightarrow{L_3 \leftarrow L_3 - L_2}$$

$$\begin{bmatrix} 1 & 0 & 1 & 2 \\ 0 & 1 & 0 & 1 \\ 0 & 0 & -1 & -1 \end{bmatrix} \xrightarrow[L_3 \leftarrow -L_3]{L_1 \leftarrow L_1 + L_3} \begin{bmatrix} 1 & 0 & 0 & 1 \\ 0 & 1 & 0 & 1 \\ 0 & 0 & 1 & 1 \end{bmatrix}$$

On peut donc en conclure que $x = \begin{pmatrix} 1 \\ 1 \\ 1 \end{pmatrix}$

b) Méthode de Cholesky

Cette méthode s'applique quand la matrice du système est symétrique définie positive. On sait qu'une telle matrice vérifie les hypothèses du théorème de décomposition LU, et donc admet une unique factorisation LU. Dans le cas d'une matrice symétrique ($A^T = A$) et définie positive ($x^T Ax > 0,\ x \neq 0$), c'est-à-dire inversible, alors il existe au moins une matrice R triangulaire supérieure telle que $A = R^T R$ avec tous les éléments diagonaux de R strictement positifs alors cette factorisation est unique et A définie positive. Du fait que la factorisation de Cholesky ne nécessite pas de pivot, donc pas de permutation des lignes ou colonnes en cours de calcul, il est donc possible de prévoir à l'avance la structure creuse du facteur de Cholesky R.

Si $A = R^T R$ alors on a : $A = \begin{bmatrix} a & 0 & 0 \\ b & c & 0 \\ d & e & f \end{bmatrix} \begin{bmatrix} a & b & c \\ 0 & c & e \\ 0 & 0 & f \end{bmatrix} = \begin{bmatrix} a^2 & ab & ad \\ ab & b^2 + c^2 & bd + ce \\ ad & bd + ce & d^2 + e^2 + f^2 \end{bmatrix}$

Pour trouver la solution x du système $Ax = R^T Rx = B$, l'idée est de se ramener à un système facilement résolvable : $\begin{cases} R^T y = B & Triangulaire\ supérieure \\ Rx = y & Triangulaire\ inférieure \end{cases}$

Calcul en pratique de la décomposition de Cholesky : Notons $A = (a_{i,j})_{1 \le i,j \le n}$ et $T = \left(t_{i,j} \right)_{1 \le i,j \le n}$ $\left(t_{i,j} = 0 \text{ si } i > j \right)$. Pour tous $i, j \in \{1, ..., n\}$,

$$a_{i,j} = \sum_{k=1}^{min(i,j)} t_{k,i} t_{k,j}$$

Prenons $A = \begin{pmatrix} 2 & -2 & -3 \\ -2 & 5 & 4 \\ -3 & 4 & 5 \end{pmatrix}$ *et* $B = \begin{pmatrix} 7 \\ -12 \\ -12 \end{pmatrix}$

On a donc :

$$A = \begin{pmatrix} 2 & -2 & -3 \\ -2 & 5 & 4 \\ -3 & 4 & 5 \end{pmatrix} = \begin{pmatrix} l_{11} & 0 & 0 \\ l_{21} & l_{22} & 0 \\ l_{31} & l_{32} & l_{33} \end{pmatrix} \begin{pmatrix} l_{11} & l_{21} & l_{31} \\ 0 & l_{22} & l_{32} \\ 0 & 0 & l_{33} \end{pmatrix}$$

On peut donc en déduire :

$$R = \begin{pmatrix} \sqrt{2} & -\sqrt{2} & -\frac{3}{\sqrt{2}} \\ 0 & \sqrt{3} & \frac{1}{\sqrt{3}} \\ 0 & 0 & \frac{1}{\sqrt{6}} \end{pmatrix}$$

Dans un premier temps, résolvons $R^T y = B$:

$$\begin{pmatrix} \sqrt{2} & 0 & 0 \\ -\sqrt{2} & \sqrt{3} & 0 \\ -\frac{3}{\sqrt{2}} & \frac{1}{\sqrt{3}} & \frac{1}{\sqrt{6}} \end{pmatrix} \begin{pmatrix} y_1 \\ y_2 \\ y_3 \end{pmatrix} = \begin{pmatrix} 7 \\ -12 \\ -12 \end{pmatrix} \quad \Leftrightarrow \begin{cases} y_1 = \frac{7}{\sqrt{2}} \\ y_2 = -\frac{5}{\sqrt{3}} \\ y_3 = \frac{1}{\sqrt{6}} \end{cases}$$

Résolvons maintenant $Rx = y$:

$$\begin{pmatrix} \sqrt{2} & -\sqrt{2} & -\frac{3}{\sqrt{2}} \\ 0 & \sqrt{3} & \frac{1}{\sqrt{3}} \\ 0 & 0 & \frac{1}{\sqrt{6}} \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix} = \begin{pmatrix} \frac{7}{\sqrt{2}} \\ -\frac{5}{\sqrt{3}} \\ \frac{1}{\sqrt{6}} \end{pmatrix} \quad \Rightarrow \begin{cases} x_3 = 1 \\ x_2 = -2 \\ x_3 = 3 \end{cases}$$

Ainsi, la solution du système est $\begin{pmatrix} 3 \\ -2 \\ 1 \end{pmatrix}$. Par ailleurs, si la matrice A n'est pas définie positive, certains éléments de R risquent d'être complexes.

Représentation de l'algorithme en pseudo-code pour une matrice de taille $n \times n$:

$t_{1,1} := \sqrt{a_{1,1}}$

Pour $i = 2$ à n faire

$$t_{i,1} := \frac{a_{1,i}}{t_{1,1}}$$

Fin Pour

Pour $k = 2$ à $n-1$ faire

$$t_{k,k} := \left(a_{k,k} - \sum_{i=1}^{k-1} t_{k,i}^2 \right)^{\frac{1}{2}}$$

Pour $i = k+1$ à n faire

$$t_{i,k} := \frac{\left(a_{i,k} - \sum_{j=1}^{k-1} t_{i,j} t_{j,k} \right)}{t_{k,k}}$$

Fin Pour

Fin Pour

$$t_{n,n} := \left(a_{n,n} - \sum_{i=1}^{n-1} t_{n,j}^2 \right)^{\frac{1}{2}}$$

III - Présentation de nos programmes et algorithmes

- **Fichier main.c :**

Dans ce fichier, nous allons retrouver l'inclusion des bibliothèques et des autres fichiers qui sont nécessaires au bon fonctionnement du programme. Ligne 40 et 41, nous avons **l'allocation dynamique** des matrices colonnes x et B, nous verrons l'allocation de la matrice A juste après. Nous initialisons la variable **"time"** à la ligne 57, c'est grâce à elle que nous allons pouvoir savoir combien de temps notre programme met pour tourner ! La fonction **clock()** ligne 64 et 76 est issue de la bibliothèque **time.h** et sert de "fin de repère de temps" pour la variable time.

- **Fichier matrices.c :**

Ce fichier permet de générer les matrices de test du TP, mais également une matrice creuse avec au moins 70 % de valeurs nulles, et bien sur, la possibilité de pouvoir rentrer une matrice de taille $n \times n$ manuellement.
Pour les génération de la matrice creuse, on se sert de la fonction **"rand()"** qui fait partie de la bibliothèque **stdlib.h**. Voyons comment cela fonctionne :
On génère un nombre aléatoire entre 0 et 100, et si ce nombre est inférieur à 70 alors la fonction **alea()** renverra un 0. Si le nombre aléatoire généré est supérieur à 70, alors la fonction **alea()** renverra un nombre aléatoire entre 0 et 100.

```
5  int alea() {
6      int result;
7      const int MAX = 100, MIN = 0;
8      int n = (rand() % (MAX - MIN + 1)) + MIN;
9      if (n < 70){
10         result = 0;
11     }
12     else{
13         result = (rand() % (MAX - MIN + 1)) + MIN;
14     }
15     return result;
16 }
```

Nous avons également dans ce fichier, entre la ligne 29 et 35, la déclaration de **"matrice"** qui nous permet d'avoir une allocation dynamique. Nous rappelons que l'allocation dynamique permet de **demander "manuellement de la mémoire"** grâce à la bibliothèque **stdlib.h**.
La fonction malloc demande au système la permission d'utiliser de la mémoire.
La fonction free permet de lui indiquer que l'on n'a plus besoin de la mémoire qu'on avait demandée.

- **Fichier affichage.c :**

Nous avons choisi d'adopter une pseudo interface graphique, même si elle ne reste que dans le shell, et c'est justement ce fichier qui permet de gérer ces différentes options.

La fonction **afficher_systeme(double **A, float *b)** nous permet d'afficher le système sous forme matricielle après avoir choisi un type de matrice prédéfini. Il en est de même pour la fonction **afficher_vecteur(float *A).**
De la ligne 58 à 122, correspond l'affichage du menu permettant de choisir parmis les différentes matrices de test proposés. et à partir de la ligne 128, nous avons la présentation du deuxième menu nous permettant de choisir entre une résolution du système, par Gauss ou Cholesky.

- **Fichier gauss.c**

Ce fichier correspond à l'implémentation de l'algorithme de Gauss. Détaillons-le un peu :
Cet algorithme **réduit le système à une matrice triangulaire supérieure** à partir de laquelle les inconnues sont dérivées par l'utilisation de la méthode de substitution en arrière.
Dans un premier temps, on cherche l'élément minimum (non nul) en valeur absolue dans la colonne k et d'indice i supérieur ou égal à k **(ligne 16).**
Si l'élément minimum est nul, on peut en déduire que la matrice est singulière (non inversible). Le programme est alors interrompu **(ligne 31).**
Si la matrice n'est pas singulière, on inverse les éléments de la ligne imax avec les éléments de la ligne k. On fait de même avec le vecteur b **(ligne 33).**
Une fois ceci fait, on procède à la réduction de la matrice par la méthode d'élimination de Gauss **(ligne 41).**
Ensuite, on vérifie que la matrice n'est toujours pas singulière, si c'est le cas, on interrompt le programme **(ligne 51).**
Une fois le système réduit, on obtient une matrice triangulaire supérieure et la résolution du système se fait très simplement **(ligne 53).**

- **Fichier cholesky.c :**

Ce fichier est une première version de l'algorithme de Cholesky, c'est également celle qui est relié au programme principale.
Tout d'abord, on commence par vérifier que la matrice est symétrique.
Afin de prouver la symétrie de la Matrice A, on doit **comparer les valeurs de A[i][j] et A[j][i] entre elles.** Or on se heurte à un problème dans la comparaison de deux float. En effet les float sont codés sur 32 bits dans la représentation **IEEE 754,** ce qui rend la comparaison impossible. Pour éviter ce problème, on utilise la fonction **"fabs()"** qui arrondit la valeur absolue.

```
12  for( i = 0 ; i < taille ; i++ )
13  {
14    for ( j = 0 ; j < taille ; j++ )
15    {
16      if ( fabs(A[i][j] - A[j][i]) > 0.0001 )
17      {
18        printf("La matrice n'est pas symétrique\n");
19        exit(EXIT_FAILURE);
20      }
21    }
```

Ensuite, le programme cherche à définir que la matrice est bien définie positive, pour pouvoir appliquer les formules mathématiques vu au début de ce dossier dans le rappel des méthodes.

- **Fichier cholesky_2.c :**

Une deuxième version du fichier, qui n'est pas accessible par le programme principale. La différence, c'est qu'ici, au lieu d'avoir des floats, on a des doubles. Le float a une simple précision et le double a une double précision ou vous pouvez dire deux fois plus de précision que float. Selon la norme **IEEE 754,** un float à une précision codée sur 32 bits alors qu'un Double sur 64 bits, ce qui implique qu'un float soit sur sur 4 octet alors qu'un double est sur 8 octets. L'algorithme nous permet de calculer la matrice R.

IV - Mise en route sur des matrices pertinentes

Nous allons à présent mettre en route nos algorithms sur différentes matrices. Testons notre programme sur les matrices de test suivant : Bord Carrée et Lehmer, vu qu'elles sont définies positives et symétriques, elles nous permettront de comparer les deux méthodes implémentées.

- **Méthode de Gauss**

Pour la matrice Bord carrée de dimension **n = 3** on avec remplissage aléatoire du vecteur B on a :

```
( 1.000  0.500  0.250 ) (X1) = 46.000

( 0.500  1.000  0.000 ) (X2) = 35.000

( 0.250  0.000  1.000 ) (X3) = 36.000
```

Avec un temps d'exécution de 0.01 ms pour avoir trouvé les 3 solutions :

```
Temps d'execution = 0.0000 s
Soit 0.01 ms

Les valeurs de x sont egales à :
(x1) = 28.3636398

(x2) = 20.8181801

(x3) = 28.9090900
```

Pour la matrice Bord carrée de dimension **n = 100** on avec remplissage aléatoire du vecteur B on a (on se heurte à un problème, les valeures sont si petites qu nous somms obligé de mettre une précision à 6 décimales près. En effet, une division par 0 entraîne l'erreur "-nan". Mais on constate que cela n'y fait rien, l'algorithme tourne toujours et ne trouve pas de solutions au système.


```
Temps d'execution = 0.0120 s
Soit 11.96 ms

Les valeurs de x sont egales à :
(x1) = -nan

(x2) = -nan

(x3) = -nan

(x4) = -nan

(x5) = -nan
```

Pour la matrice Bord carrée de dimension **n = 500** on avec remplissage aléatoire du vecteur B on a toujours pas de solutions :

```
Temps d'execution = 0.2537 s
Soit 253.66 ms

Les valeurs de x sont egales à :
(x1) = -nan

(x2) = -nan

(x3) = -nan

(x4) = -nan

(x5) = -nan
```

```
Temps d'execution = 1.9212 s
Soit 1921.20 ms

Les valeurs de x sont egales à :
(x1) = -nan

(x2) = -nan

(x3) = -nan
```

Pour la matrice Bord carrée de dimension **n = 1000** on avec remplissage aléatoire du vecteur B on a toujours pas de solutions non plus, comme dans toutes les matrices qui vont suivre :

Pour la matrice Bord carrée de dimension **n = 2000** on avec remplissage aléatoire du vecteur B

```
Temps d'execution = 15.6402 s
Soit 15640.23 ms

Les valeurs de x sont egales à :
(x1) = -nan

(x2) = -nan

(x3) = -nan
```

Passons maintenant à la matrice de Lehmer de dimension **n = 3** on avec remplissage aléatoire du vecteur B on a :

```
( 1.000000  0.500000  0.333333 ) (X1) = 41.000000

( 0.500000  1.000000  0.666667 ) (X2) = 55.000000

( 0.333333  0.666667  1.000000 ) (X3) = 21.000000
```

```
Temps d'execution = 0.0000 s
Soit 0.01 ms

Les valeurs de x sont egales à :
(x1) = -44.3999939

(x2) = 96.0000000

(x3) = -28.2000008
```

Passons maintenant à la matrice de Lehmer de dimension **n = 100** on avec remplissage aléatoire du vecteur B on a :

```
Temps d'execution = 0.0043 s
Soit 4.26 ms

Les valeurs de x sont egales à :
(x1) = 162256011264.0000000

(x2) = -81128005632.0000000

(x3) = -5083.8247070

(x4) = 4013.5937500
```

Passons maintenant à la matrice de Lehmer de dimension **n = 500** on avec remplissage aléatoire du vecteur B on a :

```
Temps d'execution = 0.2365 s
Soit 236.52 ms

Les valeurs de x sont egales à :
(x1) = -161247805440.0000000

(x2) = 80623902720.0000000

(x3) = 5082.0610352

(x4) = -3971.2683105
```

Passons maintenant à la matrice de Lehmer de dimension **n = 1000** on avec remplissage aléatoire du vecteur B on a :

```
Temps d'execution = 1.9335 s
Soit 1933.51 ms

Les valeurs de x sont egales à :
(x1) = -702912004096.0000000

(x2) = 351456002048.0000000

(x3) = 21463.2851562
```

Passons maintenant à la matrice de Lehmer de dimension **n = 2000** on avec remplissage aléatoire du vecteur B on a :

```
Temps d'execution = 15.7044 s
Soit 15704.42 ms

Les valeurs de x sont egales à :
(x1) = -982595141632.0000000

(x2) = 491297570816.0000000

(x3) = 30175.4921875
```

- **Méthode de cholesky**

Nous faisons de même pour la méthode de Cholesky, nous allons tester l'algorithme sur les même matrices avec les mêmes valeurs de n que précédemment, comme ça nous allons pouvoir les comparer.

```
( 1.000000  0.500000  0.250000 ) (X1) = 9.000000

( 0.000000  0.866025  -0.288675 ) (X2) = 98.000000

( 0.000000  0.000000  0.777282 ) (X3) = 0.000000

Transposée :

( 1.000000  0.000000  0.000000 ) (X1) = 9.000000

( 0.500000  0.866025  0.000000 ) (X2) = 98.000000

( 0.250000  -0.288675  0.777282 ) (X3) = 0.000000


Temps d'execution = 0.0001 s
Soit 0.09 ms
```

Pour la matrice Bord carrée de dimension **n = 3** on avec remplissage aléatoire du vecteur B on a :

Pour la matrice Bord carrée de dimension **n = 100** on avec remplissage aléatoire du vecteur B on a :
La présence de valeurs "-nan" signifie que l'on effectue des opérations impossible, mais l'algorithme tourne quand même.

```
 ) (X99) = 39.000000

( 0.000000  -0.288675  -0.187620  -
-nan  -nan  -nan  -nan  -nan  -nan
nan  -nan  -nan  -nan  -nan  -nan
X100) = 54.000000


Temps d'execution = 0.0131 s
Soit 13.07 ms
```

```
 -nan  -nan  -nan  -nan  -nan
-nan  -nan  -nan  -nan  -nan  -
nan  -nan  -nan  -nan  -nan  -n


Temps d'execution = 0.2673 s
Soit 267.30 ms
```

Pour la matrice Bord carrée de dimension **n = 500** on avec remplissage aléatoire du vecteur B on a :

Pour la matrice Bord carrée de dimension **n = 1000** on avec remplissage aléatoire du vecteur B on a :

```
an  -nan  -nan  -nan  -nan  -na
n  -nan  -nan  -nan  -nan  -nan


Temps d'execution = 1.6190 s
Soit 1619.00 ms
```

```
nan  -nan  -nan  -nan  -nan  -na
an ) (X2000) = 54.000000


Temps d'execution = 16.1180 s
Soit 16118.02 ms
```

Pour la matrice Bord carrée de dimension **n = 2000** on avec remplissage aléatoire du vecteur B on a :

Passons maintenant à la matrice de Lehmer de dimension **n = 3** on avec remplissage aléatoire du vecteur B on a :

```
( 1.000000  0.500000  0.333333 ) (X1) = 59.000000

( 0.000000  0.866025  0.481125 ) (X2) = 24.000000

( 0.000000  0.000000  0.638285 ) (X3) = 33.000000

Transposée :

( 1.000000  0.000000  0.000000 ) (X1) = 59.000000

( 0.500000  0.866025  0.000000 ) (X2) = 24.000000

( 0.333333  0.481125  0.638285 ) (X3) = 33.000000


Temps d'execution = 0.0001 s
Soit 0.10 ms
```

```
  -nan   -nan   -nan   -nan   -n
00


Temps d'execution = 0.0097 s
Soit 9.73 ms
```

Pour la matrice de Lehmer de dimension **n = 100** on avec remplissage aléatoire du vecteur B on a :

Pour la matrice de Lehmer de dimension n **= 500** on avec remplissage aléatoire du vecteur B on a :

```
  -nan   -nan   -nan   -nan   -nan


Temps d'execution = 0.2533 s
Soit 253.30 ms
```

```
-nan   -nan   -nan   -nan   -nan


Temps d'execution = 1.6148 s
Soit 1614.79 ms
```

Pour la matrice de Lehmer de dimension **n = 1000** on avec remplissage aléatoire du vecteur B on a :

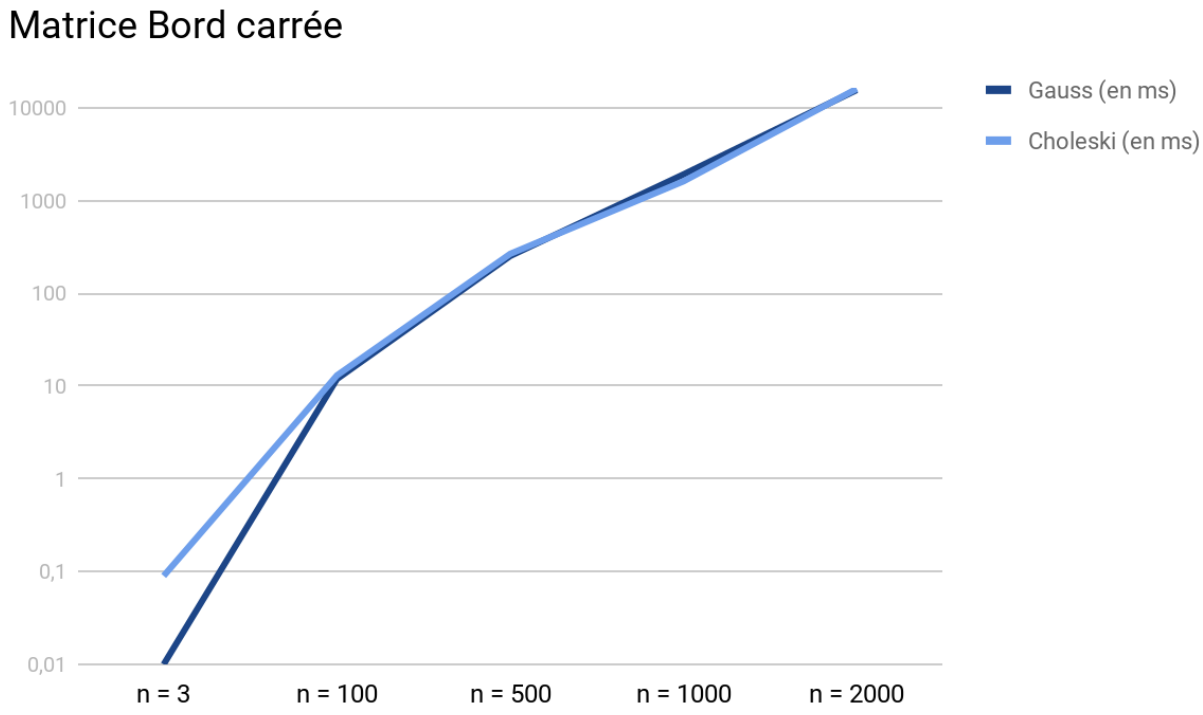
Pour la matrice de Lehmer de dimension **n = 2000** on avec remplissage aléatoire du vecteur B on a :

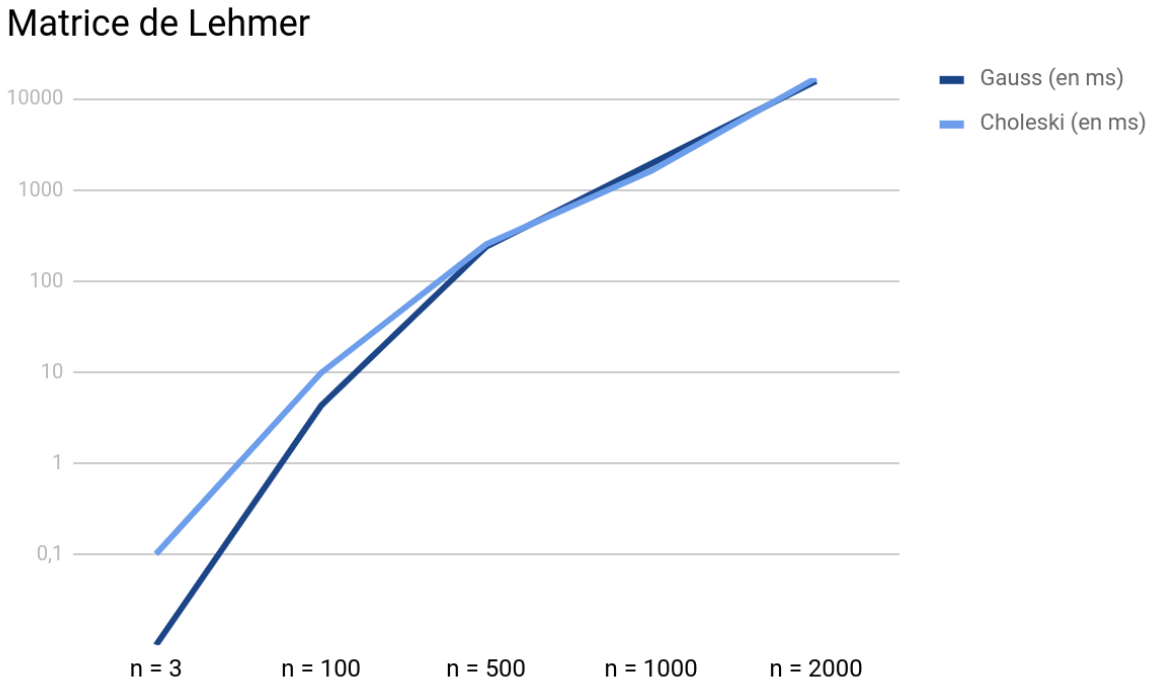
```
  -nan   -nan   -nan   -nan   -nan
38.000000


Temps d'execution = 16.9514 s
Soit 16951.37 ms
```

V - Analyse des résultats

Comme nous avons pu le voir précédemment, nous avons les temps d'exécution des deux algorithmes implémentés, en fonction des deux matrices de tests généré.
Nous réalisons donc **deux graphiques**, qui montre une comparaison de la vitesse des deux algorithmes en fonction de la taille de la matrice. Nous utilisons une **échelle logarithmique** sur l'axe des ordonnées pour pouvoir représenter de manière simple les données ayant un grand écart.





On constate que, sur les deux matrices de tests proposés, **Gauss est plus performant sur les matrice de petites tailles, mais, Cholesky tend à ratrapper son retard plus la taille de la matrice est grande.**
Et en effet, si on compare la complexité des deux algorithmes on peut dresser le tableau suivant :

	Gauss	Cholesky
Additions	$\frac{(n^3-n)}{3}$	$\frac{n^3}{6}$
Multiplication	$\frac{(n^3-n)}{3}$	$\frac{n^3}{6}$
Division	$\frac{n(n-1)}{2}$	$n^2,\ 2$
Racine		n
TOTAL	$\frac{2n^3}{3}$	$\frac{n^3}{3}$

On constate donc que l'algorithme de Cholesky est plus stable.

Par ailleurs, vu que nous faisons du calcul numérique, il est impossible d'échapper aux erreurs de représentation des décimaux en langage machine.
Les deux sources d'erreur qui interviennent systématiquement dans le calcul num´erique sont :

- **Les erreurs de troncature ou de discrétisation** qui proviennent de simplifications du modèle mathématique comme par exemple le remplacement d'une dérivée par une différence finie, le développement en série de Taylor limité, etc.
- **Les erreurs d'arrondi** qui proviennent du fait qu'il n'est pas possible de représenter (tous) les réels exactement dans un ordinateur

Afin de pouvoir effectuer des analyses d'erreur d'arrondi, on fait des hypothèses sur l'exactitude des opérations arithmétiques de base. On suivra le modèle standard :

Type	Taille	Signe	Mantisse	Exposant	Unité d'arrondi u	Bornes
Simple	32 bits	1 bit	23 bits	8 bits	$2^{-24} \simeq 5.96 \times 10^{-8}$	$10^{\pm 38}$
Double	64 bits	1 bit	52 bits	11 bits	$2^{-53} \simeq 1.11 \times 10^{-16}$	$10^{\pm 308}$

VI - Conclusion

Les inconvénient de la méthode de Gauss sont essentiellement de deux types :

- **Un nombre d'opération élevé**

Pour un grand système plein, l'algorithme de GAUSS nécessite de l'ordre de $O\left(\frac{2n^3}{3}\right)$ opérations. soit le double de la méthode de Cholesky.

- **À cause des erreurs d'arrondies**, il vaut mieux choisir le « meilleur pivot », celui dont la taille est la plus grande

Le problème provient du fait qu'en cours d'algorithme les pivots décroissent et qu'ils sont utilisés comme dénominateur pour les étapes suivantes.
Mais la méthode permet de **résoudres n'importe quel type de système**, de calculer un déterminant et d'inverser une matrice.

Vu que la méthode de Cholesky dérive de la décomposition LU , le principal intérêt de cette méthode par rapport à la décomposition LU est le **gain en mémoire** lorsqu'on implémente les méthodes et **sa stabilité**. En effet, avec la méthode de Cholesky, on ne stocke que B triangulaire, tandis qu'avec la décomposition LU on doit stocker deux matrices triangulaires.
Mais il faut noter que la **matrice doit être non seulement symétrique, mais aussi définie positive, pour appliquer Cholesky.**