Université Grenoble Alpes
Institut Fourier

Travail d'Étude et de Recherche

Implémentation en C/C++ de la résolution exacte de systèmes linéaires à coefficients entiers

Réalisé par
Barnabé Chabaux

Encadré par
Bernard Parisse

2024/2025

Table des matières

1 Introduction et objectifs **2**

2 Pivot de Gauss **3**
2.1 Algorithme . . . 3
2.1.1 Pseudo-code . . . 3
2.1.2 Avec un système . . . 4
2.2 Exemple . . . 4
2.3 Complexité . . . 5

3 Algorithme de Bareiss **6**
3.1 Algorithme . . . 6
3.1.1 Pseudo-code . . . 6
3.1.2 Avec un système . . . 6
3.2 Exemple . . . 7
3.3 Explications . . . 7
3.3.1 Coefficients entiers . . . 7
3.3.2 Borne de Hadamard . . . 8
3.3.3 Complexité de l'algorithme . . . 8

4 Méthode modulaire **9**
4.1 Restes Chinois . . . 9
4.2 Reconstruction rationnelle . . . 9
4.3 Règle de Cramer . . . 10
4.4 Algorithme général . . . 11
4.4.1 Première version . . . 11
4.4.2 Deuxième version . . . 11
4.4.3 Amélioration potentielle . . . 12
4.5 Exemple . . . 12
4.6 Complexité . . . 13
4.7 Variante en parallèle . . . 14

5 Outils utilisés **15**
5.1 La bibliothèque GMP . . . 15
5.2 Représentation des nombres rationnels . . . 15
5.3 Représentation des systèmes linéaires . . . 15
5.4 Génération de systèmes aléatoires . . . 16
5.5 Représentation des éléments de $\mathbb{Z}/p\mathbb{Z}$ . . . 16

6 Mesures du temps de calcul **17**

7 Ajouts potentiels **18**

8 Références **18**

1 Introduction et objectifs

L'objectif de ce TER est d'implémenter en langage C, par diverses méthodes, la résolution exacte de systèmes linéaires à coefficients entiers. Le code écrit pour ce travail peut être consulté sur ma page Github.

On va d'abord naïvement programmer le pivot de Gauss (section 2), puis l'améliorer un peu de sorte à ce qu'il ne fasse intervenir que des entiers (algorithme de Bareiss, section 3), et implémenter une méthode de résolution modulaire, où l'on résout le système (par le pivot de Gauss) dans $\mathbb{Z}/p\mathbb{Z}$ pour différents nombres premiers p (section 4).

Le système linéaire que l'on cherche à résoudre est de cette forme :

$$\begin{cases} a_{1,1}x_1 & + & \cdots & + & a_{1,j}x_j & + & \cdots & + & a_{1,n}x_n & = & b_1 \\ \vdots & & & & \vdots & & & & \vdots & & \vdots \\ a_{i,1}x_1 & + & \cdots & + & a_{i,j}x_j & + & \cdots & + & a_{i,n}x_n & = & b_i \\ \vdots & & & & \vdots & & & & \vdots & & \vdots \\ a_{n,1}x_1 & + & \cdots & + & a_{n,j}x_j & + & \cdots & + & a_{n,n}x_n & = & b_n \end{cases}$$

où $a_{i,j}$ et les b_i sont des entiers donnés, et les inconnues sont les x_j.
On peut le représenter sous forme matricielle :

$$\begin{pmatrix} a_{1,1} & a_{1,2} & \cdots & a_{1,j} & \cdots & a_{1,n} \\ a_{2,1} & a_{2,2} & \cdots & a_{2,j} & \cdots & a_{2,n} \\ \vdots & \vdots & & \vdots & & \vdots \\ a_{i,1} & a_{i,2} & \cdots & a_{i,j} & \cdots & a_{i,n} \\ \vdots & \vdots & & \vdots & & \vdots \\ a_{n,1} & a_{n,2} & \cdots & a_{n,j} & \cdots & a_{n,n} \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ \vdots \\ x_j \\ \vdots \\ x_n \end{pmatrix} = \begin{pmatrix} b_1 \\ b_2 \\ \vdots \\ b_i \\ \vdots \\ b_n \end{pmatrix}$$

Mais pour gagner en place (et, avec un peu de chance, en lisibilité), on le représentera surtout sous la forme suivante, avec le second membre "dans" la matrice :

$$\begin{pmatrix} a_{1,1} & a_{1,2} & \cdots & a_{1,j} & \cdots & a_{1,n} & b_1 \\ a_{2,1} & a_{2,2} & \cdots & a_{2,j} & \cdots & a_{2,n} & b_2 \\ \vdots & \vdots & & \vdots & & \vdots & \vdots \\ a_{i,1} & a_{i,2} & \cdots & a_{i,j} & \cdots & a_{i,n} & b_i \\ \vdots & \vdots & & \vdots & & \vdots & \vdots \\ a_{n,1} & a_{n,2} & \cdots & a_{n,j} & \cdots & a_{n,n} & b_n \end{pmatrix}$$

Cette représentation a l'avantage de correspondre à la façon dont sont stockés les coefficients du système dans mon implémentation, qu'on exposera dans la section 5.3.

2 Pivot de Gauss

La première méthode qui vient à l'esprit pour résoudre des systèmes linéaires est le pivot de Gauss, tel que vu en classe de L1. On peut utiliser cette méthode directement en voyant les coefficients du système comme des rationnels (cela a été implémenté dans `gauss_sys_rat.c`), mais on s'en servira aussi plus tard (dans la section 4) dans des corps finis.

2.1 Algorithme

2.1.1 Pseudo-code

Entrées : Un système de Cramer de taille n, à coefficients entiers
Sorties : Le même système linéaire, échelonné
$\varepsilon \leftarrow 1$ `// Stockera la parité du nombre de permutations de lignes effectuées, utilse si l'on veut connaître le déterminant`
pour *k allant de 1 à n* **faire**
  `// On échange éventuellement des lignes, de sorte que` $a_{k,k}$ `soit non nul et puisse servir de pivot`
  si $a_{k,k} = 0$ **alors**
    si $\exists l > k$ *tel que* $a_{l,k} \neq 0$ **alors**
      Échanger la k-ième ligne avec la l-ième
      $\varepsilon \leftarrow -\varepsilon$
    sinon
      `// Le système n'est pas de Cramer`
      Renvoyer une erreur
    fin
  fin
  pour *i allant de k+1 à n* **faire**
    `// On traite les coefficients de la` i`-ème ligne`
    pour *j allant de k à n* **faire**
      $a_{i,j} \leftarrow a_{i,j} - \frac{a_{i,k}a_{k,j}}{a_{k,k}}$ `// On remarque que` $a_{i,k}$ `vaut ainsi 0`
    fin
    `// On n'oublie pas la` i`-ème coordonnée du second membre`
    $b_i \leftarrow b_i - \frac{a_{i,k}b_k}{a_{k,k}}$
  fin
fin

Après avoir échelonné le système, il reste à "remonter" pour en obtenir la solution :

Entrées : Un système de Cramer échelonné
Sorties : La solution x de ce système
pour *k allant de n à 1* **faire**
  $x_k \leftarrow b_k$
  pour *l allant de k+1 à n* **faire**
    $x_k \leftarrow x_k - a_{k,l}x_l$
  fin
  $x_k \leftarrow \frac{x_k}{a_{k,k}}$
fin

2.1.2 Avec un système

Voyons ce que cet algorithme donne sur un système. On se place dans le cas où les k premières lignes ont été échelonnées.

$$\begin{pmatrix} a_{1,1} & \cdots & a_{1,k-1} & a_{1,k} & a_{1,k+1} & \cdots & a_{1,n} & b_1 \\ \vdots & \ddots & \vdots & \vdots & \vdots & & \vdots & \vdots \\ 0 & & a_{k-1,k-1} & a_{k-1,k} & a_{k-1,k+1} & \cdots & a_{k-1,n} & b_{k-1} \\ 0 & \cdots & 0 & a_{k,k} & a_{k,k+1} & \cdots & a_{k,n} & b_k \\ 0 & \cdots & 0 & a_{k+1,k} & a_{k+1,k+1} & \cdots & a_{k+1,n} & b_{k+1} \\ \vdots & & \vdots & \vdots & \vdots & & \vdots & \vdots \\ 0 & \cdots & 0 & a_{n,k} & a_{n,k+1} & \cdots & a_{n,n} & b_n \end{pmatrix}$$

Après l'itération du k-ème pivot, le système ressemble à ceci :

$$\begin{pmatrix} a_{1,1} & \cdots & a_{1,k-1} & a_{1,k} & a_{1,k+1} & \cdots & a_{1,n} & b_1 \\ \vdots & \ddots & \vdots & \vdots & \vdots & & \vdots & \vdots \\ 0 & & a_{k-1,k-1} & a_{k-1,k} & a_{k-1,k+1} & \cdots & a_{k-1,n} & b_{k-1} \\ 0 & \cdots & 0 & a_{k,k} & a_{k,k+1} & \cdots & a_{k,n} & b_k \\ 0 & \cdots & 0 & 0 & a_{k+1,k+1} - \frac{a_{k+1,k}a_{k,k+1}}{a_{k,k}} & \cdots & a_{k+1,n} - \frac{a_{k+1,k}a_{k,n}}{a_{k,k}} & b_{k+1} - \frac{a_{k+1,k}b_k}{a_{k,k}} \\ \vdots & & \vdots & \vdots & \vdots & & \vdots & \vdots \\ 0 & \cdots & 0 & 0 & a_{n,k+1} - \frac{a_{n,k}a_{k,k+1}}{a_{k,k}} & \cdots & a_{n,n} - \frac{a_{n,k}a_{k,n}}{a_{k,k}} & b_n - \frac{a_{n,k}b_k}{a_{k,k}} \end{pmatrix}$$

La matrice est désormais échelonnée jusqu'à la $(k+1)$-ème ligne.

2.2 Exemple

Considérons un petit système de taille 3, avec des coefficients entiers vus comme des rationnels :

$$\begin{cases} 17x_1 + 2x_2 - 3x_3 = 9 \\ 4x_1 + 7x_2 - 8x_3 = -5 \\ x_1 + 5x_3 = 4 \end{cases}$$

Ou, sous forme "matricielle avec second membre" :

$$\begin{pmatrix} 17 & 2 & -3 & 9 \\ 4 & 7 & -8 & -5 \\ 1 & 0 & 5 & 4 \end{pmatrix}$$

Le premier pivot est $a_{1,1} = 17$

$$\begin{pmatrix} 17 & 2 & -3 & 9 \\ 0 & \frac{111}{17} & \frac{124}{17} & \frac{-121}{17} \\ 0 & \frac{-2}{17} & \frac{88}{17} & \frac{59}{17} \end{pmatrix}$$

Le second pivot est $a_{2,2} = \frac{-124}{17}$

$$\begin{pmatrix} 17 & 2 & -3 & 9 \\ 0 & \frac{111}{17} & \frac{124}{17} & \frac{-121}{17} \\ 0 & 0 & \frac{560}{111} & \frac{371}{111} \end{pmatrix}$$

Le système est échelonné. On en déduit sa solution :

$$\begin{cases} x_1 = \frac{11}{16} \\ x_2 = \frac{-7}{20} \\ x_3 = \frac{53}{80} \end{cases}$$

Notons que pour des système plus grands, la taille des coefficients a tendance à exploser. Par exemple, pour un système de taille 10, avec des coefficients (tirés uniformément) dans $[\![-128, 127]\!]$, on obtient en fin d'excution un coefficient $a_{10,10} = \frac{-3868437346564691901402}{153787212580753328981}$.

2.3 Complexité

Cet algorithme effectue $O(n^3)$ opérations sur le corps dans lequel on l'utilise. C'est intéressant dans un corps fini où ce coût est constant, mais beaucoup moins si on l'utilise directement sur un système à coefficients entiers (donc dans le corps $\mathbb{Q}$), où il dépend de la taille des éléments (qui a tendance à vite croître).

3 Algorithme de Bareiss

L'algorithme de Bareiss est une variante du pivot de Gauss, qui permet d'éviter de se retrouver avec des coefficients rationnels (non entiers). Son implémentation peut être consultée dans le fichier `bareiss.c`.

3.1 Algorithme

3.1.1 Pseudo-code

Entrées : Un système de Cramer de taille n, à coefficients entiers
Sorties : Le même système linéaire, échelonné, et à coefficients entiers
$a_{0,0} \leftarrow 1$ `// On décrète que le "0-ème" pivot est 1, de sorte que diviser par` $a_{0,0}$ `ne change rien`
pour *k allant de 1 à n* **faire**
  si $\exists l \geq k$ *tel que* $a_{l,k} \neq 0$ **alors**
    Choisir $l \geq k$ tel que $|a_{l,k}|$ soit non nul et minimal
    Échanger la k-ième ligne avec la l-ième
    `// Cela réduit un peu la complexité des calculs à effectuer`
  sinon
    `// Le système n'est pas de Cramer`
    Renvoyer une erreur
  fin
  pour *i allant de k+1 à n* **faire**
    `// On traite les coefficients de la` i`-ème ligne`
    pour *j allant de k à n* **faire**
      $a_{i,j} \leftarrow \frac{a_{i,j}a_{k,k} - a_{i,k}a_{k,j}}{a_{k-1,k-1}}$ `// On remarque que` $a_{i,k}$ `vaut ainsi 0`
    fin
    `// On n'oublie pas la` i`-ème coordonnée du second membre`
    $b_i \leftarrow \frac{b_i a_{k,k} - a_{i,k}b_k}{a_{k-1,k-1}}$
  fin
fin

Une fois le système échelonné, il reste à obtenir la solution, en remontant avec le même algorithme qu'après le pivot de Gauss.

3.1.2 Avec un système

Comme pour le pivot de Gauss, voyons à quoi ressemble une itération de la boucle principale de l'algorithme de Bareiss. Ici, les k premières lignes sont échelonnées.

$$\begin{pmatrix} a_{1,1} & \cdots & a_{1,k-1} & a_{1,k} & a_{1,k+1} & \cdots & a_{1,n} & b_1 \\ \vdots & \ddots & \vdots & \vdots & \vdots & & \vdots & \vdots \\ 0 & & a_{k-1,k-1} & a_{k-1,k} & a_{k-1,k+1} & \cdots & a_{k-1,n} & b_{k-1} \\ 0 & \cdots & 0 & a_{k,k} & a_{k,k+1} & \cdots & a_{k,n} & b_k \\ 0 & \cdots & 0 & a_{k+1,k} & a_{k+1,k+1} & \cdots & a_{k+1,n} & b_{k+1} \\ \vdots & & \vdots & \vdots & \vdots & & \vdots & \vdots \\ 0 & \cdots & 0 & a_{n,k} & a_{n,k+1} & \cdots & a_{n,n} & b_n \end{pmatrix}$$

Après l'itération du k-ème pivot, le système ressemble à ceci :

$$\begin{pmatrix}
a_{1,1} & \cdots & a_{1,k-1} & a_{1,k} & a_{1,k+1} & \cdots & a_{1,n} & b_1 \\
\vdots & \ddots & \vdots & \vdots & \vdots & & \vdots & \vdots \\
0 & & a_{k-1,k-1} & a_{k-1,k} & a_{k-1,k+1} & \cdots & a_{k-1,n} & b_{k-1} \\
0 & \cdots & 0 & a_{k,k} & a_{k,k+1} & \cdots & a_{k,n} & b_k \\
0 & \cdots & 0 & 0 & \frac{a_{k+1,k+1}a_{k,k}-a_{k+1,k}a_{k,k+1}}{a_{k-1,k-1}} & \cdots & \frac{a_{k+1,n}a_{k,k}-a_{k+1,k}a_{k,n}}{a_{k-1,k-1}} & \frac{b_{k+1}a_{k,k}-a_{k+1,k}b_k}{a_{k-1,k-1}} \\
\vdots & & \vdots & \vdots & \vdots & & \vdots & \vdots \\
0 & \cdots & 0 & 0 & \frac{a_{n,k+1}a_{k,k}-a_{n,k}a_{k,k+1}}{a_{k-1,k-1}} & \cdots & \frac{a_{n,n}a_{k,k}-a_{n,k}a_{k,n}}{a_{k-1,k-1}} & \frac{b_n a_{k,k}-a_{n,k}b_k}{a_{k-1,k-1}}
\end{pmatrix}$$

Les $k+1$ premières lignes de la matrice sont désormais échelonnées.

3.2 Exemple

Pour éviter de rajouter des étapes (et ainsi rester à peu près lisible), on ne cherchera pas à échanger les lignes pour minimiser la taille des pivots. Considérons le même petit système que précédemment.

$$\begin{pmatrix} 17 & 2 & -3 & 9 \\ 4 & 7 & -8 & -5 \\ 1 & 0 & 5 & 4 \end{pmatrix}$$

Le premier pivot est $a_{1,1} = 17$

$$\begin{pmatrix} 17 & 2 & -3 & 9 \\ 0 & 111 & -124 & -121 \\ 0 & -2 & 88 & 59 \end{pmatrix}$$

Le second pivot est $a_{2,2} = 111$

$$\begin{pmatrix} 17 & 2 & -3 & 9 \\ 0 & 111 & -124 & -121 \\ 0 & 0 & 560 & 371 \end{pmatrix}$$

Le système est échelonné. On en déduit sa solution :

$$\begin{cases} x_1 = \frac{11}{16} \\ x_2 = \frac{-7}{20} \\ x_3 = \frac{53}{80} \end{cases}$$

On remarque que c'est bien la même solution que précédemment.

3.3 Explications

3.3.1 Coefficients entiers

On va voir que pendant l'exécution de l'lgorithme, les coefficients du système sont des déterminants de matrices entières (et sont donc entiers). Notons L_i la i-ième ligne du système. Considérons l'opération suivante, que l'on effectue pour chaque pivot lors de l'algorithme :

$$L_i \leftarrow \frac{a_{k,k}L_i - a_{i,k}L_k}{a_{k-1,k-1}}$$

Contrairement à la simple tranvection que l'on effectue lors du pivot de Gauss, cette opération modifie le déterminant du système, qui se voit multiplié par $\frac{a_{k,k}}{a_{k-1,k-1}}$. Pour le k-ième pivot, on effectue cette opération pour chacune des $n-k$ lignes suivantes, ce qui multiplie le déterminant par

$\frac{a_{k,k}^{n-k}}{a_{k-1,k-1}^{n-k}}$. À la fin de l'algorithme (donc après avoir traité les pivots 1 à $n-1$), on a donc multiplié le déterminant par :

$$a_{1,1}^{n-1} \times \frac{a_{2,2}^{n-2}}{a_{1,1}^{n-2}} \times \frac{a_{3,3}^{n-3}}{a_{2,2}^{n-3}} \times \ldots \times \frac{a_{n-2,n-2}^{2}}{a_{n-3,n-3}^{2}} \times \frac{a_{n-1,n-1}}{a_{n-2,n-2}} = a_{1,1} \times a_{2,2} \times \ldots \times a_{n-1,n-1}$$

On a donc d'une part :

$$\det(A_{fin}) = a_{1,1} \times a_{2,2} \times \ldots \times a_{n-1,n-1} \times \det(A_{début})$$

Et d'autre part, puisque la matrice finale A_{fin} est échelonnée, on a aussi :

$$\det(A_{fin}) = a_{1,1} \times a_{2,2} \times \ldots \times a_{n-1,n-1} \times a_{n,n}$$

Par conséquent $a_{n,n} = \det(A_{début})$.

On obtient ainsi un moyen de connaître le déterminant de la matrice de départ, mais ça implique surtout (puisque $A_{début} \in M_n(\mathbb{Z})$) que $a_{n,n} \in \mathbb{Z}$. On peut appliquer le même raisonnement à des sous-matrices de $A_{début}$ pour arriver à la même conclusion avec les autres coefficients.

3.3.2 Borne de Hadamard

Soit $A \in M_n(\mathbb{R})$ (ou même $M_n(\mathbb{C})$). On note $C_1, \ldots, C_n$ ses vecteurs colonnes. On a :

$$|\det(A)| \leq \|C_1\|_2 \times \ldots \times \|C_n\|_2$$

Notons n la taille du système (son nombre de lignes), et c la taille de ses coefficients initiaux (le nombre de bits nécessaires pour les écrire). Lors de l'exécution de l'algorithme, les coefficients du système étant des déterminants de sous-matrices, leur taille est en $O(cn\ \ln(n))$.

En effet, $\|C_i\|_2 = \sqrt{a_{1,i}^2 + \ldots + a_{n,i}^2}$ a une taille en $O(c\ \ln(n))$, et donc $\|C_1\|_2 \ldots \|C_n\|_2$ a bien une taille en $O(cn\ \ln(n))$.

À noter que lorsqu'on va utiliser cette inégalité pour borner les numérateurs et les dénominateurs des coordonnées de la solution du système (sous-sections 4.2 et 4.3), il sera nécessaire d'aussi prendre en compte le second membre b :

$$|\det(A)| \leq \|C_1\|_2 \times \ldots \times \|C_n\|_2 \times \|b\|_2$$

3.3.3 Complexité de l'algorithme

Lors de l'exécution de l'algorithme, les coefficients sont des déterminants de sous-matrices, donc leur taille est en $O(cn\ \ln(an))$. Les multiplications $a_{i,j}a_{k,k}$ et $a_{i,k}a_{k,j}$ ont chacune un coût en $O(c^2n^2\ \ln(n)^2)$ avec la multiplication classique "naïve". Le calcul de chaque coefficient à chaque étape vaut donc lui aussi $O(c^2n^2\ \ln(n)^2)$. Ce calcul étant fait de l'ordre de n^3 fois, la complexité totale de l'algorithme de Bareiss est en $O(c^2n^5\ \ln(n)^2)$. Avec une meilleure multiplication (FFT), le calcul de chaque coefficient coûte seulement $O(cn\ \ln(n)^2)$ (à des $\ln(\ln(n))$ près), ce qui porte la complexité totale de l'algorithme à $O(c^2n^4\ \ln(n)^2)$.

4 Méthode modulaire

L'idée de la méthode modulaire est de résoudre le système modulo divers nombres premiers, et d'utiliser ces solutions modulaires pour construire la solution du système de départ. Une telle méthode confronte d'emblée à différentes problématiques :

— Il est nécessaire de "combiner" des éléments de $\mathbb{Z}/n_1\mathbb{Z}$ et de $\mathbb{Z}/n_2\mathbb{Z}$ pour obtenir des éléments de $\mathbb{Z}/n_1n_2\mathbb{Z}$. Cela correspond précisément au théorème des restes chinois (4.1).

— Il faut obtenir une solution (du système initial) à la fin, et donc un moyen de construire celle-ci. Deux approches ont été essayées : la construction de rationnels à partir d'éléments de $\mathbb{Z}/a\mathbb{Z}$ (4.2), et une méthode utilisant la règle de Cramer (4.3).

— Enfin, il est nécessaire de savoir quand s'arrêter de faire des résolutions modulaires et renvoyer la solution. Là aussi, deux approches ont été essayées : construire un candidat solution à chaque itération et comparer avec celui de l'itération précédente, ou comparer le produit des nombres premiers utilisés avec la borne de Hadamard (voir 3.3.2) du système.

4.1 Restes Chinois

On cherche à obtenir $x \in \mathbb{Z}/n_1n_2\mathbb{Z}$ tel que :

$$\begin{cases} x \equiv x_1 \pmod{n_1} \\ x \equiv x_2 \pmod{n_2} \end{cases}$$

Ceci va nécessiter une relation de Bézout minimale entre n_1 et n_2, c'est-à-dire des entiers u et v tels que :

$$\begin{cases} n_1u + n_2v = 1 \\ |u| \leq \frac{|n_2|}{2} \\ |v| \leq \frac{|n_1|}{2} \end{cases}$$

Dans un premier temps, d'après le cours, $x = x_1n_2v + x_2n_1u$ convient. Cependant, dans le cas où n_1 est grand par rapport à n_2 (ça sera par exemple le cas lorsque n_1 sera le produit de tout les nombres premiers utilisés précédemment dans l'algorithme), x_1 et v sont grands également, et sont multipliés entre eux dans le premier terme. Cette multiplication de deux "grands" facteurs est coûteuse, et peut être évitée.

En calculant $x = x_1 + ((x_2 - x_1)u \pmod{n_2})n_1$, on évite de faire un produit entre n_1 et un autre "grand" facteur, puisque l'autre facteur a été réduit modulo n_2. Dans le cas où la taille de n_2 est négligeable, cette simplification fait passer la complexité du calcul des restes chinois de $O(\ln(n_1)^2)$ à $O(\ln(n_1))$.

4.2 Reconstruction rationnelle

À partir d'un $b \in \mathbb{Z}/a\mathbb{Z}$, on cherche à obtenir un rationnel $\frac{num}{den}$, avec $num \in \mathbb{Z}$ et $den \in \mathbb{N}^*$ premiers entre eux, tel que :

$$\begin{cases} num\ den^{-1} \equiv b \pmod{a} \\ |num| < \frac{\sqrt{a}}{2} \\ 0 < den < \frac{\sqrt{a}}{2} \\ den \wedge a = 1 \end{cases}$$

Si une telle solution existe, on la trouve en utilisant une variante de l'algorithme d'Euclide étendu, où l'on s'arrête dès qu'on obtient un reste strictement inférieur à $\sqrt{a}$.

Entrées : $a \in \mathbb{N}^*,\ b \in \mathbb{Z}/a\mathbb{Z}$
Sorties : $\frac{num}{den}$, vérifiant les conditions souhaitées
$r_0 \leftarrow a$
$r_1 \leftarrow b$
$v_0 \leftarrow 0$
$v_1 \leftarrow 1$
// u_0 et u_1 ne sont pas nécessaires ici
$k \leftarrow 1$
tant que $r_k \geq \sqrt{a}$ **faire**
  $k \leftarrow k + 1$
  $q \leftarrow$ quotient de la division euclidienne de r_{k-2} par r_{k-1}
  $r_k \leftarrow r_{k-2} - qr_{k-1}$ // Reste de la division euclidienne ci-dessus
  $v_k \leftarrow v_{k-2} - qv_{k-1}$
fin
// À ce stade, $den = |v_k|$ et $num = \pm v_k$ (selon le signe de v_k)
// On veut s'assurer que le signe soit "porté" par num :
si $v_k < 0$ **alors**
  retourner $\frac{-r_k}{-v_k}$
sinon
  retourner $\frac{r_k}{v_k}$
fin

Typiquement, dans ce qui va suivre, on utilisera la reconstruction rationnelle pour calculer chacune des coordonnées de la solution, donc n fois. On verra dans la section 4.6 que la reconstruction rationnelle est trop coûteuse pour être utilisée à chaque itération.

Cet algorithme aurait aussi pu servir dans le cadre d'une méthode p-adique, qui n'a pas été implémentée.

4.3 Règle de Cramer

Considérons un système linéaire de taille n :

$$Ax = b$$

(On a donc $A \in M_n(\mathbb{Z})$ et $b \in \mathbb{Z}^n$ donnés, et l'inconnue est $x \in \mathbb{Q}^n$)
Notons A_j la matrice A dont la j-ième colonne a été remplacée par le second membre b :

$$A_j = \begin{pmatrix} a_{1,1} & a_{1,2} & \cdots & a_{1,j-1} & b_1 & a_{1,j+1} & \cdots & a_{1,n} \\ a_{2,1} & a_{2,2} & \cdots & a_{2,j-1} & b_2 & a_{2,j+1} & \cdots & a_{2,n} \\ \vdots & \vdots & & \vdots & \vdots & \vdots & & \vdots \\ a_{i,1} & a_{i,2} & \cdots & a_{i,j-1} & b_i & a_{i,j+1} & \cdots & a_{i,n} \\ \vdots & \vdots & & \vdots & \vdots & \vdots & & \vdots \\ a_{n,1} & a_{n,2} & \cdots & a_{n,j-1} & b_n & a_{n,j+1} & \cdots & a_{n,n} \end{pmatrix} \in M_n(\mathbb{Z})$$

On a alors :

$$\forall j \in [\![1, n]\!], x_j = \frac{det(A_j)}{det(A)}$$

En général, calculer une solution de cette façon est extrêmement coûteux (le calcul des déterminants est en $O(n!)$). Dans notre cas, on va pouvoir facilement calculer les déterminants dans $\mathbb{Z}/p\mathbb{Z}$. À partir de la forme échelonnée dans $\mathbb{Z}/p\mathbb{Z}$ du système, de sa solution (modulo p) y, et du nombre l de

permutation effectuées pendant le pivot de Gauss, on a :

$$\begin{cases} \det(A) = (-1)^l \prod_{i=1}^{n} a_{i,i} \\ \forall j \in [\![1, n]\!], \det(A_j) = \det(A) x_j \end{cases}$$

On peut alors utiliser les restes chinois pour obtenir les déterminants dans $\mathbb{Z}/prod\mathbb{Z}$ avec $prod$ assez grand, et en déduire la solution du système.

4.4 Algorithme général

Différentes variantes de l'algorithme modulaire ont été programmés, avec des résultats variables.

4.4.1 Première version

Entrées : Un système de Cramer de taille n, à coefficients entiers
Sorties : La solution x du système
$prod \leftarrow 1$ `// Représente le produit des nombres premiers utilisés`
tant que *le nouveau candidat solution est différent de l'ancien* **faire**
    Choisir p premier
    Effectuer le pivot de Gauss dans $\mathbb{Z}/p\mathbb{Z}$
    En déduire une solution du système modulo p
    Utiliser les restes chinois pour obtenir une solution modulo $prod \times p$
    $prod \leftarrow prod \times p$
    $x \leftarrow$ candidat solution obtenu par reconstruction rationnelle
fin

Cette version s'est vite montrée beaucoup trop lente, à cause de l'exécution de la reconstruction rationnelle à chaque étape. Elle a en outre le défaut de nécessiter une vérification de la solution à la fin, car il n'est pas impossible que la reconstruction rationnelle renvoie deux fois d'affilée le même candidat solution sans que celui-ci ne soit effectivement la solution.

4.4.2 Deuxième version

L'utilisation de la borne de Hadamard permet de savoir quand s'arrêter sans avoir à comparer les résultats des itérations. Cela enlève le besoin de construire une solution rationnelle à chaque étape (il suffit de le faire une unique fois, à la fin), et évite aussi d'avoir à vérifier la solution obtenue.

Entrées : Un système de Cramer de taille n, à coefficients entiers
Sorties : La solution x du système
$prod \leftarrow 1$ `// Représente le produit des nombres premiers utilisés`
$hada \leftarrow$ Borne de Hadamard du système
tant que $\frac{\sqrt{prod}}{2} < hada$ **faire**
    Choisir p premier
    Effectuer le pivot de Gauss dans $\mathbb{Z}/p\mathbb{Z}$
    En déduire une solution du système modulo p `// Obtenue en remontant le système échelonné, voir section 2`
    Utiliser les restes chinois pour obtenir une solution modulo $prod \times p$
    $prod \leftarrow prod \times p$
fin
$x \leftarrow$ solution obtenue par reconstruction rationnelle

Comme la borne de Hadamard est une majoration de la valeur absolue du déterminant, les numérateurs et dénominateurs des coordonnées de la solution sont eux-mêmes bornés par cette borne. Or la reconstruction rationnelle donne, à partir d'un élément de $\mathbb{Z}/prod\mathbb{Z}$, un numérateur et

un dénominateur plus petits que $\frac{\sqrt{prod}}{2}$ en valeur absolue. On a donc besoin de l'utiliser avec $prod$ tel que $\frac{\sqrt{prod}}{2} \geq hada$.

4.4.3 Amélioration potentielle

Pour ne pas avoir du tout à utiliser de reconstruction rationnelle, on peut utiliser la règle de Cramer. Cela nécessite d'utiliser la représentation symétrique pour les éléments de $\mathbb{Z}/prodZ$. Comme celle-ci permet d'obtenir des numérateurs en dénominateurs dans $[\![-\frac{prod}{2}, \frac{prod}{2}]\!]$, il suffit maintenant que $prod$ soit tel que $\frac{prod}{2} < hada$ pour pouvoir obtenir la solution.

On rappelle que A_k désigne la matrice des coefficients du système, dans laquelle on remplace la k-ième colonne par b (voir 4.3).

Entrées : Un système de Cramer de taille n, à coefficients entiers
Sorties : La solution x du système
$prod \leftarrow 1$ `// Représente le produit des nombres premiers utilisés`
$hada \leftarrow$ Borne de Hadamard du système
tant que $\frac{prod}{2} < hada$ **faire**
  Choisir p premier
  Effectuer le pivot de Gauss dans $\mathbb{Z}/p\mathbb{Z}$ `// En représentation symétrique`
  $\det(A^{(p)}) \leftarrow a_{1,1} \times \ldots \times a_{n,n}$ `// C'est le déterminant de` A `modulo p`
  pour k *allant de 1 à* n **faire**
    $y_k \leftarrow$ solution du système dans $\mathbb{Z}/p\mathbb{Z}$ `// Obtenue en remontant le système échelonné, voir section 2`
    $\det(A_k^{(p)}) \leftarrow \det(A^{(p)}) \times y_k$ `// C'est le déterminant de` A_k `modulo p`
  fin
  Utiliser les restes chinois pour obtenir tous ces déterminants modulo $prod \times p$
  $prod \leftarrow prod \times p$
fin
`// Utilisation de la règle de Cramer pour obtenir la solution :`
pour k *allant de 1 à* n **faire**
  $x_k \leftarrow \frac{\det(A_k) \bmod prod}{\det(A) \bmod prod}$ `// En représentation symétrique`
fin

Malgré la correction de plusieurs bugs et erreurs, je n'ai pas encore réussi à faire fonctionner correctement cette variante de l'algorithme, elle ne sera donc pas présente dans les mesures de temps de calcul (section 6)

4.5 Exemple

Considérons le même exemple que d'habitude, et utilisons l'algorithme 4.4.3).

$$\begin{pmatrix} 17 & 2 & -3 & 9 \\ 4 & 7 & -8 & -5 \\ 1 & 0 & 5 & 4 \end{pmatrix}$$

La borne de Hadamard de ce système est :

$$hada = \sqrt{306} \times \sqrt{53} \times \sqrt{98} \times \sqrt{122} \simeq 13925$$

On a donc besoin de nombres premiers dont le produit soit supérieur à 6963, par exemple $p = 11$ et $q = 701$ (en pratique, on utiliserait des nombres premiers bien plus grands). Il faut maintenant

échelonner le système dans $\mathbb{Z}/p\mathbb{Z}$ et $\mathbb{Z}/q\mathbb{Z}$ avec le pivot de Gauss (ici en représentation symétrique) :

$$\begin{pmatrix} -5 & 2 & -3 & -2 \\ 0 & 2 & 5 & 0 \\ 0 & 0 & -1 & -3 \end{pmatrix} \text{ dans } \mathbb{Z}/p\mathbb{Z} \qquad \begin{pmatrix} 17 & 2 & -3 & 9 \\ 0 & 89 & -131 & -337 \\ 0 & 0 & -96 & -344 \end{pmatrix} \text{ dans } \mathbb{Z}/q\mathbb{Z}$$

On en déduit les solutions modulaires du système :

$$\begin{cases} y_1 \equiv 0 \\ y_2 \equiv -2 \\ y_{\equiv} 3 \end{cases} \text{ dans } \mathbb{Z}/p\mathbb{Z} \qquad \begin{cases} y_1 \equiv -306 \\ y_2 \equiv 245 \\ y_3 \equiv 62 \end{cases} \text{ dans } \mathbb{Z}/q\mathbb{Z}$$

Il faut alors calculer les déterminants apparaissant dans la règle de Cramer :

$$\begin{cases} \det(A) = -1 \\ \det(A_1) \equiv -1 \times 0 \equiv 0 \\ \det(A_2) \equiv -1 \times -2 \equiv 2 \\ \det(A_3) \equiv -1 \times 3 \equiv -3 \end{cases} \text{ dans } \mathbb{Z}/p\mathbb{Z} \qquad \begin{cases} \det(A) = -141 \\ \det(A_1 \equiv -141 \times -306 \equiv -316 \\ \det(A_2) \equiv -141 \times 245 \equiv -196 \\ \det(A_3) \equiv -114 \times 62 \equiv -330 \end{cases} \text{ dans } \mathbb{Z}/q\mathbb{Z}$$

Avec les restes chinois, on obtient :

$$\begin{cases} \det(A) = 560 \\ \det(A_1) = 385 \\ \det(A_2) = -196 \\ \det(A_3) = 371 \end{cases} \text{ dans } \mathbb{Z}/pq\mathbb{Z}$$

Il reste à calculer $x_i = \frac{\det(A_i)}{\det(A)}$ pour chaque i. On obtient bien la même solution qu'avec les algorithmes de Gauss et de Bareiss :

$$\begin{cases} x_1 = \frac{11}{16} \\ x_2 = \frac{-7}{20} \\ x_3 = \frac{53}{80} \end{cases}$$

4.6 Complexité

Comme précédemment, on note n la taille du système et c la taille des coefficients initiaux.

Pour atteindre la majoration du déterminant du système donnée par la borne de Hadamard, il est nécessaire d'effectuer $O(cn \ln(n))$ exécutions du pivot de Gauss dans des $\mathbb{Z}/p\mathbb{Z}$ (avec des valeurs de p différentes). Dans la variante 4.4.2 (à ce stade la meilleure variante qui fonctionne), il s'agit même d'en effectuer $O(c^2n^2 \ln(n)^2)$, pour atteindre le carré de la borne de Hadamard.

Le pivot de Gauss étant ici utilisé dans des corps finis, son coût est donc en $O(n^3)$ à chaque itération. Il faut aussi obtenir une solution (en "remontant" le système échelonné), mais cette étape est en $O(n^2)$ et est donc négligeable.

On utilise les restes chinois avec n_1 qui est le produit de tous les nombres premiers précédents (donc majoré par la borne de Hadamard ou son carré, selon la variante de l'algorithme utilisée), et n_2 qui est un nombre premier de "petite" taille. Comme vu précédemment (4.1), la méthode naïve de calcul des restes chinois a une complexité en $O(\ln(n_1)^2)$, et la meilleure méthode a une complexité en $O(\ln(n_1))$ seulement. Les restes chinois sont appelés n fois par itération, juste après un calcul des coefficients de Bézout (négligeable). En négligeant les $\ln(\ln(n))$ et les constantes sortant des logarithmes (cette constante est assez lourde dans le cas de l'algorithme 4.4.2, à cause de la borne de Hadamard qui est au carré), la complexité du calcul des restes chinois est en $O(n \ln(cn)^2)$ pour la version naïve, et $O(n \ln(cn))$ pour la version améliorée, à chaque itération.

Quant à la reconstruction rationnelle (effectuée à partir de $b \in \mathbb{Z}/a\mathbb{Z}$), on a que $\frac{\sqrt{a}}{2}$ est majoré par la borne de Hadamard (3.3.2), donc est de taille $O(c^2n^2\ \ln(n)^2)$, et donc b aussi. La complexité de l'algorithme d'Euclide étendu est donc en $O(c^4n^4\ \ln(n)^4)$. Comme il faut faire une reconstruction rationnelle pour chacun des n coefficients de la solution, la complexité totale de la reconstruction rationnelle atteint $O(c^4n^5\ \ln(n)^4)$, ce qui est supérieur à la complexité en $O(n^3)$ du pivot de Gauss, et rend déraisonnable l'utilisation de la reconstruction rationnelle à chaque étape.

Si on fait le bilan (et qu'on s'autorise de négliger les logarithmes), on obtient une complexité de :

— $O(c^6n^7)$ pour la variante 4.4.1 (avec reconstruction rationnelle à chaque étape).

— $O(c^4n^5)$ pour la variante 4.4.2 (avec borne de Hadamard et une seule reconstruction rationnelle à la fin). Ici, le coût de la reconstruction rationnelle est légèrement supérieur au coût du reste des calculs, qui est en $O(c^2n^5)$.

— $O(c^2n^4)$ pour la variante 4.4.3 (utilisant la règle de Cramer). C'est en fait $O(c^2n^4\ \ln(n)^2)$, soit autant que l'algorithme de Bareiss.

4.7 Variante en parallèle

Pour les algorithmes des sous-sections 4.4.2 et 4.4.3, il est possible de profiter de la multiplicité des cœurs sur les processeurs modernes, et d'effectuer les calculs modulaires en parallèle dans plusieurs fils d'exécution. À chaque itération, on peut lancer T fils d'exécution différents, qui vont travailler avec des valeurs de p différentes, et donner T résultats qu'il faudra recombiner à l'aide des restes chinois. Ceci ne change pas la complexité à proprement parler des algorithmes, mais permet théoriquement d'essentiellement diviser le temps de calcul par le nombre de cœurs du processeur de la machine où l'on exécute l'algorithme (donc en général par 4 ou par 8, sur les ordinateurs portables de la dernière décennie).

5 Outils utilisés

En plus du langage de programmation C et de ses librairies standard, plusieurs outils ont été nécessaires pour mener à bien ce travail.

5.1 La bibliothèque GMP

GMP (GNU Multiple Precision) est une bibliothèque libre et open source qui permet de manipuler des nombres entiers, rationnels et flottants, avec une précision arbitraire (contrairement aux entiers et flottants disponibles nativement). GMP met à disposition des fonctions permettant d'effectuer diverses opérations sur les entiers, allant des calculs basiques aux recherches de PGCD et de coefficients de Bézout, en passant par des tests de primalité, de la génération de nombres pseudo-aléatoires, et des choses plus terre-à-terre comme l'affichage des nombres dans un fichier ou dans le terminal. La résolution de systèmes à coefficients entiers ayant tendance à faire apparaître des nombres d'assez grande taille, c'est avec cette bibliothèque que j'ai représenté la plupart des entiers dans ce travail.

5.2 Représentation des nombres rationnels

Les systèmes à coefficients entiers ayant des solutions rationnelles, il est nécessaire de trouver une façon de représenter les nombres rationnels. Plutôt que d'utiliser les rationnels fournis par GMP, j'ai choisi d'implémenter moi-même des nombres rationnels à partir des entiers de GMP, principalement car c'est une tâche assez simple qui permettait un premier contact avec la bibliothèque et sa documentation. Un nombre rationnel est représenté par deux entiers : son numérateur et son dénominateur, supposés premiers entre eux.

$$r = \frac{p}{q}, \text{ avec } p \in \mathbb{Z} \text{ et } q \in \mathbb{N}^*$$

De cette façon, les nombres rationnels sont représentés de façon exacte (ça ne serait pas le cas en utilisant des flottants), et leurs numérateurs et dénominateurs peuvent être de taille arbitraire. À la fin de chaque calcul, pour s'assurer que le numérateur et le dénominateur d'un rationnel restent premiers entre eux (ce qui évite de ralentir les calculs en traînant des entiers énormes inutilement), ils sont divisés par leur PGCD. Il restait alors à écrire quelques fonctions pour manipuler ces rationnels : affectation, opérations de base, test d'égalité, initialisation/suppression... D'autres petites fonctions auraient raisonnablement pu être écrites, comme une comparaison entre deux rationnels, mais elles n'avaient pas vraiment d'utilité dans le cadre de ce travail. Tout le code en lien avec cette implémentation se trouve dans les fichiers `rationnels.c` et `rationnels.h`.

5.3 Représentation des systèmes linéaires

Lors de l'exécution du programme, les systèmes sont représentés par une matrice d'entiers de GMP, qui est elle-même représentée par un type structuré contenant le nombre de lignes `n`, le nombre de colonnes `m`, un tableau (1D) contenant tous les coefficients (y compris le second membre). Le nombre de colonnes a initialement été rajouté dans l'optique de permettre de résoudre un système pour plusieurs seconds membres à la fois, mais cette idée n'a finalement pas vu le jour, et donc en pratique `m = n+1`. Pour accéder plus simplement aux coefficients du système, des fonctions `lit_coeff` et `ecrit_coeff` ont été écrites pour accéder aux coefficients du système à partir de leur numéro de ligne et de colonne. D'autres fonctions ont été écrites pour afficher un système dans le terminal, vérifier si un vecteur est solution d'un système, ou encore pour initialiser, copier ou détruire la structure de système... Toutes ces fonctions sont dans les fichiers `systemes.c` et `systemes.h`.

Les systèmes sont lus depuis un fichier texte, ce qui permet à l'utilisateur de modifier et de consulter, relativement facilement, les systèmes que le programme va manipuler. Ces fichiers texte

contiennent (sur une seule ligne, séparés par des espaces) `n`, `m`, puis tous les coefficients du système (ordonnés ligne par ligne, de haut en bas et de gauche à droite). Le code gérant la lecture des systèmes peut être consulté dans `io.c`.

5.4 Génération de systèmes aléatoires

Pour pouvoir tester mes algorithmes sur des systèmes raisonnablement grands, il a été nécessaire d'implémenter un moyen de générer des systèmes aléatoires. Le tirage des coefficients a été simplement effectué à l'aide de la librairie GMP, qui permet de tirer uniformément des entiers dans $[\![0, 2^b - 1]\!]$ (où b peut être arbitrairement grand). Les nombres ainsi obtenus sont ensuite changés de signe avec une probabilité $\frac{1}{2}$, pour obtenir des coefficients uniformément distribués dans $[\![-2^b, 2^b - 1]\!]$, s'écrivant donc avec $b + 1$ bits. Les systèmes ainsi obtenus sont alors stockés dans des fichiers texte, et lus par le programme. Ceci a été implémenté dans `io.c`.

5.5 Représentation des éléments de $\mathbb{Z}/p\mathbb{Z}$

Les nombres premiers p que l'on choisit pour les calculs dans $\mathbb{Z}/p\mathbb{Z}$ s'écrivent sur 30 bits (pas 31, pour éviter un dépassement d'entiers en faisant des sommes), donc les éléments de $\mathbb{Z}/p\mathbb{Z}$ peuvent être représentés par des entiers signés de 32 bits (c'est-à-dire des `int`). Les additions et soustractions sont effectuées en utilisant les opérations natives, puis en ajoutant ou enlevant p de sorte à ce que le résultat soit dans $[\![0, p - 1]\!]$. Les multiplications sont effectuées de façon semblable, en multipliant d'abord les opérandes (il faut alors utiliser des `long int` pour le résultat intermédiaire, afin d'éviter un dépassement d'entiers) et en effectuant une division euclidienne par p. L'inverse modulaire est calculé à l'aide de l'algorithme d'Euclide étendu.

En revanche, en général, pour $\mathbb{Z}/n\mathbb{Z}$ avec n grand, on représente les éléments par des entiers de la librairie GMP.

Pour l'algorithme 4.4.3, on a représenté $\mathbb{Z}/n\mathbb{Z}$ de façon symétrique, avec les éléments représentés par des entiers de $[\![-\frac{n}{2}, \frac{n}{2}]\!]$.

6 Mesures du temps de calcul

Les mesures qui suivent ont été effectuées en comparant la valeur de retour de deux appels de la fonction `clock` (avant et après l'appel de la fonction appliquant un algorithme), et en divisant le tout par la constante `CLOCKS_PER_SEC` de la librairie `time.h`. Les durées indiquées, bien qu'elles soient écrites en secondes, ne sont donc pas des durées "réelles", mais sont théoriquement proportionnelles au temps d'exécution des algorithmes sur ma machine (et semblent malgré tout varier d'une machine à l'autre). Seulement trois chiffres significatifs ont été gardés. Les valeurs entre parenthèses (à côté des durées) sont les nombres d'itérations, pour les algorithmes où cette donnée a un sens. Les cases avec une croix dénotent de calcul vraiment trop long, qui a fini par être interrompu.

Pour des valeurs de n et c données, un système de taille n avec des coefficients de taille c a été généré aléatoirement, et tous les algorithmes ont été exécutés sur ce même système, qui peut être trouvé sur le Github sous le nom `systeme-nNcC`, en remplaçant `N` par la valeur de n et `C` par la valeur de c. Dans le tableau, "Modulaire 1" désigne l'algorithme de la sous-section 4.4.1, "Modulaire 2" désigne celui de la sous-section 4.4.2, et "Modulaire 2P" désigne aussi l'algorithme de la sous-section 4.4.2, mais en parallèle avec 8 fils d'exécution (les itérations sont donc faites 8 à la fois, et le nombre indiqué est leur total).

↓ Taille ; Algorithme →	Gauss	Bareiss	Modulaire 1	Modulaire 2	Modulaire 2P
$n = 5$, $c = 96$	0,000233	0,000093	0,00233 (35)	0,000362 (42)	0,00291 (48)
$n = 5$, $c = 512$	0,00144	0,000382	0,0957 (180)	0,00293 (216)	0,00960 (232)
$n = 5$, $c = 2048$	0,00736	0,00265	3,48 (719)	0,0261 (860)	0,0500 (864)
$n = 50$, $c = 12$	0,0844	0,00934	0,0439 (45)	0,00896 (47)	0,0131 (48)
$n = 50$, $c = 96$	1,03	0,124	4,50 (337)	0,103 (347)	0,129 (352)
$n = 50$, $c = 512$	12,4	1,72	493 (1796)	1,35 (1829)	1,41 (1840)
$n = 50$, $c = 2048$	102	13,4	×	18,3 (7321)	17,3 (7320)
$n = 200$, $c = 12$	26,1	1,36	5,28 (188)	1,47 (199)	1,64 (200)
$n = 200$, $c = 96$	444	30,9	900 (1361)	12,2 (1379)	13,1 (1384)
$n = 200$, $c = 512$	×	388	×	117 (7729)	117 (7240)
$n = 700$, $c = 12$	×	298	644 (698)	207 (734)	236 (736)

On remarque d'emblée que la version parallélisée de la méthode modulaire ne tient pas ses promesses, et que la première version de la méthode modulaire est effectivement très lente, de même que le pivot de Gauss naïf.

Le seul potentiel "avantage" de la méthode 4.4.1 aurait pu être que les reconstructions rationnelles à chaque étape peuvent permettre de s'arrêter plus tôt que la borne de Hadamard si la solution est déjà trouvée, mais on constate qu'on n'effectue pas beaucoup moins d'itérations qu'avec la méthode 4.4.2, qui est de toute façon bien plus rapide.

On constate également que l'algorithme de Bareiss est souvent plus rapide que l'algorithme 4.4.2, sauf pour des valeurs de n particulièrement grandes. Cette observation va à l'encontre des complexités théoriques établies dans la sous-section 4.6, ce qui n'est certainement pas une bonne nouvelle.

Quelques tests ont montré que la variante 4.4.3 de la méthode modulaire avait tendance à s'exécuter significativement plus vite que l'algorithme de Bareiss ou la variante 4.4.2, mais cela n'a aucun intérêt pour l'instant, puisque le résultat obtenu avec cet algorithme est très souvent faux. En outre, la majorité du temps d'exécution de la variante 4.4.2 est passé à effectuer la reconstruction modulaire, donc il serait souhaitable d'arriver à se débarrasser de cette étape.

7 Ajouts potentiels

Quelques améliorations peuvent être envisagées, et certaines seront potentiellement effectuées entre la remise de ce mémoire et la soutenance :

- Trouver et corriger le problème avec mon implémentation actuelle de l'algorithme 4.4.3
- Améliorer la version en parallèle de la méthode modulaire, par exemple faire en sorte que le calcul sur un seul fil d'exécution utilise le même code que la méthode en parallèle, et/ou utiliser plusieurs fils d'exécution pour calculer les restes chinois entre les résultats des différents fils à la fin de chaque itération
- Utiliser des `long int` au lieu des `int` pour représenter les éléments de $\mathbb{Z}/p\mathbb{Z}$, pour pouvoir prendre des nombres premiers de 62 bits
- Réécrire la fonction qui calcule les restes chinois, en prenant en argument des entiers de GMP pour n_1 (qui est grand) et des entiers natifs pour n_2 (qui est petit)

8 Références

- Mon code source pour ce travail : https://github.com/Barni072/TER
- Documentation de la bibliothèque GMP, par le projet GNU : https://gmplib.org/manual
- Le cours d'algèbre effective du second semestre de M1MG, par Samuel Le Fourn et Bernard Parisse
- Page des algorithmes de Xcas, par Bernard Parisse : https://www-fourier.univ-grenoble-alpes.fr/ parisse/giac/doc/fr/algo.html