PIVOT DE GAUSS - SYSTÈME DE CRAMER

$$\begin{cases} a_{11}.x_1 + a_{12}.x_2 + \cdots + a_{1n}.x_n &= b_1 \\ a_{21}.x_1 + a_{22}.x_2 + \cdots + a_{2n}.x_n &= b_2 \\ \vdots &= \vdots \\ a_{n1}.x_1 + a_{n2}.x_2 + \cdots + a_{nn}.x_n &= b_n \end{cases} \Rightarrow \begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{11} \\ \vdots & \vdots & \vdots & \vdots \\ a_{n1} & a_{n2} & \cdots & a_{nn} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix} = \begin{bmatrix} b_1 \\ b_2 \\ \vdots \\ b_n \end{bmatrix}$$

Table des matières

1 Introduction

1.1 Système d'équations linéaires

Dans les domaines du calcul de structure, de la simulation multiphysique ou encore en métrologie, il est fréquent d'avoir à résoudre des systèmes d'équations linéaires. L'algorithme de Gauss permet de résoudre ces systèmes. Soit un système de n équations linéaires, on peut l'écrire sous la forme matricielle $A.X = B$:

$$\begin{cases} a_{11}.x_1 + a_{12}.x_2 + \cdots + a_{1n}.x_n &= b_1 \\ a_{21}.x_1 + a_{22}.x_2 + \cdots + a_{2n}.x_n &= b_2 \\ \vdots &= \vdots \\ a_{n1}.x_1 + a_{n2}.x_2 + \cdots + a_{nn}.x_n &= b_n \end{cases} \Rightarrow \begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{11} \\ \vdots & \vdots & \vdots & \vdots \\ a_{n1} & a_{n2} & \cdots & a_{nn} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix} = \begin{bmatrix} b_1 \\ b_2 \\ \vdots \\ b_n \end{bmatrix}$$

1.2 Méthode directe

En utilisant la méthode de Gauss, l'objectif est de résoudre ce système en *une fois*. Ce type de méthode est appelé directe en opposition aux méthodes itératives pour lesquelles on répète des opérations jusqu'à ce que le résultat converge vers une solution.

Les méthodes directes sont encore majoritairement employées dans les codes actuels car elles sont robustes et que l'on est capable de déterminer à l'avance leur coût (nombre d'opération, stockage nécessaire). Cependant, le coût pour la méthode de Gauss augmente fortement pour des systèmes de grande taille.

1.3 Quelques facteurs influents

Les systèmes auxquels on va s'intéresser dans la suite sont supposés linéaires et inversibles.

Si la dimension de la matrice $\dim(A) = n$ est importante le nombre de données à stocker est très élevé (n^2). Afin de réduire les ressources mémoire nécessaires, le stockage des données devra être optimisé.

De plus, lors du stockage numérique de ces données, chaque nombre est codé sur un nombre fini de bits. Selon le principe de la *virgule flottante*, des approximations sont donc commises. Or pour les systèmes de grande taille, ou mal conditionné (), la méthode de Gauss a tendance à amplifier les erreurs d'arrondi (d'où une instabilité numérique).

2 Méthode de Gauss

2.1 Principe du pivot

2.1.1 Transvection

Pour appliquer la méthode de Gauss, on procède par opérations élémentaires sur les lignes L_i d'un système d'équations. Or la solution d'un système linéaire ne change pas si :

- on multiplie tous les termes d'une équation par une constante non nulle :

$$L_i \leftarrow \lambda.L_i \quad \text{avec} \quad \lambda \neq 0$$

- on remplace une équation par la somme, membre à membre de cette équation et d'une autre équation du système :

$$L_i \leftarrow L_i + \lambda.L_j$$

On construit alors la matrice de transvection $T_{i,j}(\lambda)$, de dimension n, définie par :

$$[T_{i,j}(\lambda)]_{k,l} = \begin{cases} 1 & \text{si } k = l \\ \lambda & \text{si } k = i \text{ et } l = j \\ 0 & \text{sinon} \end{cases} \quad \Leftrightarrow \quad T_{i,j}(\lambda) = \begin{matrix} \\ L_i \\ \\ \\ \\ \end{matrix} \begin{matrix} \begin{matrix} & & C_j & \end{matrix} \\ \begin{bmatrix} 1 & & 0 & \\ & 1 & \lambda & \\ & & \ddots & \\ & 0 & 1 & \\ & & & 1 \end{bmatrix} \end{matrix} = \mathbb{I}_n + \lambda.E_{i,j}$$

avec $\mathbb{I}_n$, la matrice identité de dimension n et $E_{i,j}$ la matrice constituée de 0 sauf ligne i et colonne j où elle admet 1 comme cœfficient.

2.1.2 Triangularisation

L'idée principale de la méthode de Gauss est d'obtenir, par opérations élémentaires, un système triangulaire d'équations, où la ligne L_i s'exprime en fonction de i (resp. $n - i + 1$) inconnues dont une seule n'est pas apparue (resp. n'apparaît pas) dans les lignes précédentes (resp. suivantes) :

$$\begin{cases} a'_{11}.x_1 + a'_{12}.x_2 + \cdots + a'_{1n}.x_n &= b'_1 \\ a'_{22}.x_2 + \cdots + a'_{2n}.x_n &= b'_2 \\ \ddots \quad \vdots &= \vdots \\ a'_{nn}.x_n &= b'_n \end{cases} \Rightarrow \begin{bmatrix} a'_{11} & a'_{12} & \cdots & a'_{1n} \\ & a'_{22} & \cdots & a'_{11} \\ & & \ddots & \vdots \\ & & & a'_{nn} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix} = \begin{bmatrix} b'_1 \\ b'_2 \\ \vdots \\ b'_n \end{bmatrix}$$

Pour trianguliser, il faut éliminer des cœfficients sous la diagonale de la matrice. A l'étape i :

$$\forall\, k \in [\![i+1, n]\!],\ L_k \leftarrow L_k - \frac{a_{k,i}}{a_{i,i}}.L_i$$

On utilise donc des pivots, les cœfficients $a_{i,i}$.

Il apparaît donc un problème mathématiquement si $a_{i,i} = 0$ et numériquement si $a_{i,i} \ll 1$.

Pour éviter ce problème deux solutions existent:

- méthode du pivot partiel : permutation des lignes ou des colonnes $\Rightarrow$ méthode simple
- méthode du pivot total : permutation des lignes et des colonnes $\Rightarrow$ méthode plus robuste

Dans le cas du pivot partiel sur les lignes :

Entrées : A,B
pour i *de* 1 *à* $n-1$ **faire**
  déterminer j tel que $a_{j,i} = \sup_{i \le k \le n} |a_{k,i}|$
  inverser ligne i et ligne j sur A et sur B
  $L_i, L_j \leftarrow L_j, L_i$
  pour k *de* $i+1$ *à* n **faire**
    $L_k \leftarrow L_k - \frac{a_{k,i}}{a_{i,i}}.L_i$
Sorties : A,B

2.1.3 Remontée

Une fois la matrice triangularisée, si le problème initial est un système de Cramer, on obtient simplement la solution, ligne par ligne :

$$\forall i \in [\![1,n]\!]\ x_i = \frac{1}{a'_{i,j}}.\left(b'_i - \sum_{k=i+1}^{n} a'_{i,k}.x_k\right)$$

Entrées : A',B'
pour *i de n à* 1 **faire**
  pour *j de* $i+1$ *à n* **faire**
    $b'_i = b'_i - a'_{i,k}.x_k$
  $x_i = \dfrac{b'_i}{a'_{i,i}}$
Sorties : x

2.2 Inversion de matrice

2.2.1 Balayage

Lorsque la matrice est triangularisée, on utilise une méthode identique à celle de la triangularisation pour diagonaliser la matrice A.

Entrées : A',B'
pour *i de n à* 2 **faire**
  pour *k de* $i-1$ *à* 1 **faire**
    $L_k \leftarrow L_k - \dfrac{a'_{k,i}}{a'_{i,i}}.L_i$
Sorties : A',B'

2.2.2 Résolution

Le système d'équation ayant été diagonalisé, le problème de Cramer initial peut alors s'écrire :

$$\begin{cases} a''_{11}.x_1 &= b''_1 \\ a''_{22}.x_2 &= b''_2 \\ \vdots &= \vdots \\ a''_{nn}.x_n &= b''_n \end{cases} \Rightarrow \begin{bmatrix} a''_{11} & 0 & \cdots & 0 \\ 0 & a''_{22} & \ddots & \vdots \\ \vdots & \ddots & \ddots & 0 \\ 0 & \dots & 0 & a''_{nn} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix} = \begin{bmatrix} b''_1 \\ b''_2 \\ \vdots \\ b''_n \end{bmatrix} \quad \Rightarrow \quad \forall x \in [\![1,n]\!]\ x_i = \frac{b''_i}{a''_{i,i}}$$

Entrées : A",B"
pour *i de* 1 *à n* **faire**
  $x_i = \dfrac{b''_i}{a''_{i,i}}$
Sorties : x

2.2.3 Matrice inverse

On suppose la matrice A inversible, donnant une solution unique au problème de Cramer $A.X = B$. On appelle alors :

- $(T_i)_1^N$ les N matrices de transvection permettant de trianguliariser la matrice A (passer de A à A') :

$$\underbrace{T_N.T_{N-1}.\dots.T_1}_{T}.A = A'$$

- $(S_i)_1^M$ les M matrices de transvection permettant de diagonaliser A' (passer de A' à A'')

$$\underbrace{S_M.S_{M-1}.\ldots.S_1}_{S}.A' = A$$

- D, la matrice diagonale ayant pour cœfficient diagonaux $d_{i,i} = \frac{1}{a''_{i,j}}$

On obtient donc $D.S.T.A = I_n$. La matrice A étant inversible, $D.S.T$ apparaît donc comme l'inverse de A.

Ainsi pour déterminer l'inverse d'une matrice inversible A, il suffit d'appliquer à la matrice identité les mêmes opérations élémentaires sur les lignes et/ou les colonnes qui permettent de passe de A à I_n.

3 Approximations

3.1 Sources d'approximation dans le calcul numérique

Les deux sources d'erreur qui interviennent systématiquement dans le calcul numérique 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.

RAPPEL
Dans un ordinateur, une variable réelle $x \neq 0$ est représentée en virgule flottante par

$$x = (-1)^s.m.b^e$$

où m est la mantisse (ou partie fractionnelle), b la base (2 classiquement dans un ordinateur) et e est l'exposant.
La répartition suivant la norme IEEE pour coder un nombre en virgule flottante est:

Simple précision (32 bits)	Exposant codé sur 8 bits, mantisse 23 bits plus 1 pour le signe.
Double précision (64 bits)	Exposant sur 11 bits, mantisse 52 bits plus 1 pour le signe.

La précision en virgule flottante est définie comme le plus petit nombre positif ε tel que $float(1+\varepsilon) > 1$.

On peut montrer que pour une mantisse comportant nb bits, si on arrondit avec la fonction float() selon la technique dite "perfect rounding", on obtient $\varepsilon = \frac{1}{2^{nb}}$.

Si on utilise des mots en double précision, la précision machine est alors $\varepsilon = \frac{1}{2^{52}} \approx 2,22.10^{-16}$.

Pour déterminer directement la précision machine en utilisant python on peut utiliser **np.finfo** :

```
%\begin{python}[H]
>>> np.finfo(np.float64).eps
2.2204460492503131e-16
>>> np.finfo(np.float32).eps
1.1920929e-07
%\end{python}
```

Au final l'épsilon machine est ici de:

Simple précision	Double précision
$\varepsilon \approx 1,2.10^{-7}$	$\varepsilon \approx 2,2.10^{-16}$

3.2 Conditionnement

3.2.1 Définition

Pour A symétrique et inversible, on appelle conditionnement de A (que l'on note $cond(A)$), la valeur définit par :

$$cond(A) = \|A\|.\|A^{-1}\|$$

où $\|.\|$ est une norme sur l'espace des matrices.

EXEMPLE :

$$\text{norme 1} \quad \|A\|_1 = \max_j \sum_i |a_{i,j}|$$

$$\text{norme infinie} \quad \|A\|_\infty = \max_i \sum_j |a_{i,j}|, \ldots$$

En général, il est très coûteux de calculer le conditionnement de A. En revanche, un certain nombre de techniques peuvent permettre de l'estimer.

On va montrer dans la suite qu'avec un conditionnement grand la résolution du système conduit à des problèmes de précision (cumul d'arrondis numériques).

3.2.2 Initialisation

Un premier calcul utilisant l'algorithme de Gauss défini précédemment est conduit en utilisant des nombres simple précision.

On crée une matrice A, dont on fixe la précision avec float32, puis on construit B de manière à être sûr que la solution $x = [2, -2]$ soit la solution du problème:

```
A=np.array([[0.2161, 0.1441],[1.2969001, 0.8648]],dtype=np.float32)
B=np.array([[ 2*A[0,0]-2*A[0,1]],[ 2*A[1,0]-2*A[1,1]]],dtype=np.float32)
```

On obtient alors $A' = \begin{bmatrix} 0,21610001 & 0,1441 \\ 1,29690015 & 0,86479998 \end{bmatrix}$ et $B' = \begin{bmatrix} 0,14400002 \\ 0,86420035 \end{bmatrix}$.

3.2.3 Triangularisation

```
for k in range(N):
    # Mise a zeros
    for i in range(k+1,M):
        #On modifie le second membre avant A
        B[i] -= B[k]*A[i,k]/A[k,k]
        A[i,:] -= A[k,:]*A[i,k]/A[k,k]
```

On obtient alors:

$$A = \begin{bmatrix} 2,16100007e-01 & 1,44099995e-01 \\ -1,19209290e-07 & -1,19209290e-07 \end{bmatrix}$$

$$B = \begin{bmatrix} 1,44000024e-01 \\ 2,38418579e-07 \end{bmatrix}$$

3.2.4 Remontée

Pour une précision de 32 bits on obtient alors $sol = \begin{bmatrix} 5,88888787 \\ -1,94309494 \end{bmatrix}$. Cette solution est très loin de la solution exacte.

Lorsqu'on réalise le calcul avec une précision de 64bits la solution en revanche est bien $sol = \begin{bmatrix} 2. \\ -2. \end{bmatrix}$.

```
for k in range(N-1,0,-1):   #-1 pour pas negatif
        for i in range(k):
            B[i] -= B[k]*A[i,k]/A[k,k]
            A[i,:] -= A[k,:]*A[i,k]/A[k,k]
sol= np.zeros((1,N))
for k in range(N):
    sol[0,k]=B[k]/A[k,k]
```

3.2.5 Calcul du résidu

On appelle résidu ou erreur résiduel r tel que:

$$r = B - A.X$$

```
res=b-np.dot(A,X)
```

Ce résidu devrait normalement donner une indication sur la qualité de la solution.

Cependant, on obtient :

- Résidu pour une précision de 32: $\begin{bmatrix} 0,00000000e+00 \\ 7,02010139e-07 \end{bmatrix}$
- Résidu pour une précision de 64: $\begin{bmatrix} 0. \\ 0. \end{bmatrix}$

Alors que la solution est très différente, on obtient bien dans les deux cas un résultat proche de 0. Si on tient compte de la précision machine correspondant au calcul, pour le calcul en 32 bits on a en effet $\varepsilon \approx 1,2.10^{-7}$.

3.2.6 Calcul du conditionnement de A

On utilise la norme infinie pour déterminer la norme de A:

$$||A||_\infty = \max_{1\leq i\leq n} \sum_{j=1}^{n} |a_{ij}| = \max(2.1617, 0.3602) = 2.1617$$

Exceptionnellement pour cet exemple on inverse la matrice A:

$$A^{-1} = \begin{bmatrix} 14410000.00 & -86480000.00 \\ -21610000.00 & 129690000.0 \end{bmatrix} \Rightarrow ||A^{-1}||_\infty = 151300000$$

D'où le conditionnement de A:

$$cond(A) = ||A||.||A^{-1}|| = 3,2.10^{8}$$

Le conditionnement de A est grand $cond(A) \gg 1$.

Le résidu (dans le cas du calcul en 32bits) est petit puisque $B - A.X = [0.00000000e+00; 7.02010139e-07]^T$ alors que la solution exacte $[2-2]^T$ est très différente de $X = [5.88888787; -1.94309494]^T$. Il apparaît donc que le résidu ne soit pas un indicateur totalement fiable.

D'autre part il semble que le conditionnement du système soit très fortement lié à la qualité de la solution (en terme d'influence des arrondis numériques) que l'on puisse espérer avoir. On va démontrer ceci dans la suite.

On voit qu'avec une précision de 64 bits on n'a pas ici de problème de résolution. En augmentant la précision sur les nombres (choix d'une précision double ou simple), on évite les problèmes liés aux erreurs d'arrondis. Cependant on augmente la taille mémoire demandée pour stocker les données.

3.2.7 Système perturbé

Pour illustrer l'influence du conditionnement, on est amené à considérer un système perturbé (par des perturbations généralement supposées petites) $\tilde{A}.\tilde{X} = \tilde{B}$.

Ici on choisit de modifier A très légèrement et de rester dans la suite du calcul avec une précision de 64 bits sur les nombres.

On introduit sur un seul terme une perturbation de 1.10^{-7} correspondant donc de l'ordre de grandeur de l'epsilon machine quand on fait le calcul avec une précision de 32 bits et donc au type d'erreur introduite lors d'un calcul en simple précision.

```
A=np.array([[0.2161001, 0.1441],[1.2969001, 0.8648]],dtype=np.float64)
B=np.array([[ 0.144] ,[0.8642002]],dtype=np.float64)
```

La solution est alors $[-0,78653134; 2,17883068]$ et le résidu de $[0,0; 0,0]$.

On constate que pour une valeur de conditionnement élevé (loin de 1) le résultat (loin de la solution $[-2, 2]$) est très sensible aux perturbations et par conséquent, aux approximations numériques.

De plus, le calcul du résidu n'est donc pas fiable pour un problème mal conditionné.

3.2.8 Influence des perturbations

Considérons le système $\tilde{A}.\tilde{X} = \tilde{B}$. Si $\tilde{B} = B + \delta B$ avec $||\delta B|| \ll ||B||$. A-t-on $||\delta X|| \ll ||X||$?

Calculons simplement, en utilisant l'inégalité de Milkowski :

$$A.X = B \Rightarrow ||A.X|| = B|| \quad \Rightarrow \quad ||A||.||X|| \geq ||B|| \quad \Rightarrow \quad \frac{1}{||X||} \leq \frac{||A||}{||B||}$$

Dans le cas du système perturbé au niveau de B :

$$A.(X + \delta X) = B + \delta B \Rightarrow A.\delta X = \delta B \Rightarrow \delta X = A^{-1}\delta B \quad \Rightarrow \quad ||\delta X|| \leq ||A^{-1}||.||\delta X||$$

par conséquent:

$$\frac{||\delta X||}{||X||} \leq cond(A)\frac{||\delta B||}{||B||}$$

Dans le cas du système perturbé au niveau de A :
Si $\tilde{A} = A + \delta A$ et $\tilde{X} = X + \delta X$ alors

$$(A + \delta A).(X + \delta X) = B \Rightarrow A.\delta X + \delta A.(X + \delta X) = 0 \Rightarrow \delta X = -A^{-1}.\delta A.(X + \delta X)$$

$$\Rightarrow \quad ||\delta X|| \leq ||A^{-1}||.||\delta A||.||X + \delta X||$$

soit au final :

$$\frac{||\delta x||}{||x + \delta x||} \leq cond(A).\frac{||\delta A||}{||A||}$$

Même si la perturbation est faible, que ce soit sur A ou B, si le conditionnement est grand, alors l'influence sur le résultat peut être important.

3.3 Choix du pivot

3.3.1 Problème de stabilité

Au-delà du problème de conditionnement l'algorithme de Gauss peut présenter des problèmes de stabilité.

Le pivot de Gauss est le terme $a_{i,i}$ de la matrice A au début de l'étape i. Étant donné que les calculs se font avec une arithmétique en précision finie, la grandeur relative du pivot influence la précision des résultats.

On peut rencontrer des pivots très petits sans que le problème soit mal conditionné:

$$A = \begin{bmatrix} \varepsilon & 1 \\ 1 & 0 \end{bmatrix} \quad \text{et} \quad cond(A) = 1$$

3.3.2 Exemple d'instabilité de l'algorithme de Gauss

Soit le système $\begin{cases} 0,0001.x_1 + 1,0.x_2 = 1 & (1) \\ 1,0000.x_1 + 1,0.x_2 = 2 & (2) \end{cases}$ pour lequel la solution exacte est $x_1 = 1,0001$ et $x_2 = 0,9998$.

Appliquons l'élimination de Gauss avec une arithmétique en base 10 et $nb = 3$.

On a $a_{1,1} = 0,0001$ le pivot et $m_{G1,2} = \frac{a_{2,1}}{a_{1,1}} = \frac{1}{0,0001} = 10000$ le multiplicateur de Gauss pour la deuxième ligne.
L'équation (2) devient alors:

$$\begin{aligned} (2) : \quad & 1.x_1 + 1.x_2 = 2 \\ (1).m_{G1,2} : \quad & -1.x_1 + -10000.x_2 = -10000 \\ (2) + (1).m_{G1,2} : \quad & -999.x_2 = -9998 \end{aligned}$$

Avec 3 digits significatifs on obtient alors $x_2 = 1$. On remplace x_2 dans (1) et on obtient $x_1 = 0$. C'est très loin de la solution attendue.

Or ceci ne se produit pas si on permute les lignes.

Ainsi, l'élimination de Gauss est un algorithme numériquement instable. Il est donc nécessaire d'introduire dans l'algorithme un mécanisme de permutation des lignes et/ou des colonnes afin d'éviter des petits pivots.

En conclusion, pour obtenir un résultat valable à l'issu d'une résolution de système linéaire il faut avoir un problème bien conditionné afin d'avoir un résultat précis, mais il faut aussi veiller à ce que l'algorithme soit numériquement stable afin de produire la bonne solution.

4 Technique de stockage et coût

4.1 Notion de coût

Le coût de résolution d'un problème fait naturellement intervenir le temps mis pour effectuer l'ensemble des opérations, mais aussi la taille mémoire requise.

Le temps peut être mesuré de deux façons :

- temps horloge : mesure globale du temps extérieur écoulé pendant la résolution, dépendante de la machine utilisée, du problème traité et de la compilation du programme. Le résultat est alors une estimation du temps pour effectuer un programme dans sa globalité, il n'est pas uniquement lié à l'algorithme employé.
- temps CPU (Central Processing Unit) : cette fois lié aux opérations effectuées.

Pour avoir un indicateur sur l'algorithme uniquement on utilise le nombre d'opérations en virgule flottante nécessaire à la résolution.

4.2 Coût du stockage et optimisation

4.2.1 Détermination de l'espace mémoire nécessaire pour une matrice pleine

La structure de données naïve utilisée pour stocker une matrice $M_{m,n}$ est un tableau à deux dimensions. Pour une matrice $M_{m,n}$ il faut au moins $m.n$ espaces mémoire (un pour chaque terme) de taille fixe pour représenter la matrice. Chaque terme

Nombre de nœuds	5	13	49	149	449	999
Taille matrice	7.7	23.23	95.95	295.295	895.895	1995.1995
Taille mémoire	392 o	4ko	72 ko	696 ko	6.4 Mo	31.8 Mo

TABLE 1 – Evolution de la taille mémoire pour l'exemple du treillis FIG 1.

est représenté classiquement par un flottant (double précision) codé sur 64 bits. Lors de simulations numériques il est courant d'avoir des applications avec des tailles de système de $n = 10^3$.

4.2.2 Réorganisation des équations du système

En simulations numériques, il est fréquent que chacune des équations à résoudre ne concerne qu'une faible partie de l'ensemble des inconnues du système. Par exemple, la ligne i peut être fonction de x_i et des ses deux plus proches voisins (x_{i-1} et x_{i+1}). Chaque ligne présente alors un grand nombre de zéros. On parle alors de matrices creuses.

Avec quelques opérations sur les lignes et les colonnes, il est souvent possible de réarranger le problème initial pour obtenir une matrice sous forme plus classique (zeros, ones, diagonale, bande, ligne de ciel,...).

4.2.3 Stockage des matrices creuses

L'idée est ici de ne stocker que les termes non nuls de la matrice. Il existe de nombreux formats pour cela. On va utiliser le format *Yale Sparse Matrix* qui stocke une matrice M de taille $m.n$ sous la forme de trois tableaux unidimensionnels. Soit k le nombre de termes non nuls de M :

- le premier tableau noté A est de longueur k. Il contient toutes les valeurs des entrées non nulles de M de gauche à droite et de haut en bas.
- le deuxième tableau est noté IA de longueur $m + 1$ (le nombre de lignes plus un). L'élément i de IA contient l'index dans le tableau A de la première entrée non nulle de la ligne i de la matrice M. La ligne i de la matrice originale est composée des éléments de A depuis l'index $IA(i)$ jusqu'à l'index $IA(i + 1) - 1$.
- le troisième tableau, noté JA, de longueur k, contient le numéro de la colonne de chaque élément de A

Les tableaux IA et JA ne stockent que des nombres entiers.

4.2.4 Stockage des matrices bande

On parle de matrice Bande lorsque les cœfficients non nuls d'une matrice creuse se regroupent autour de la diagonale. Tous les cœfficients $a_{i,j}$ non nuls sont situés dans une bande délimitée par des parallèles à la diagonale principale.

Une telle bande est déterminée par deux entiers k_1 et k_2 positifs ou nuls, tels que : $a_{i,j} = 0$ si $j < i - k_1$ ou $j > i + k_2$

Toutes les diagonales situées entre les deux diagonales contenant les éléments non nuls les plus éloignés de la diagonale principale sont stockées comme colonnes d'une matrice rectangulaire. Le nombre de lignes de cette matrice est égal à l'ordre de la matrice creuse. Son nombre de colonnes est égale à la largeur de la bande.

4.3 Coût en opération et optimisation

Le fait de stocker un nombre important de termes ne pose pas uniquement un problème pour le stockage.

En effet, le nombre d'opérations pour résoudre le système $A.X = B$ est lié aussi au nombre de termes stockés de la matrice.

4.3.1 Complexité en triangularisation

On s'intéresse ici à la complexité de la triangularisation pour une matrice $n.n$ sans tenir compte de la recherche du pivot.

Considérons l'étape $i \in [\![1, n]\!]$. Pour chaque ligne k allant de $i+1$ à n, on recense les opérations suivantes:

- une division (div) afin de calculer, pour toute la ligne le cœfficient permettant d'obtenir des zéros sous le pivot ($m_{Gi,k}$)
- une addition (add) et une multiplication (mult) permettant de mettre à jour chaque cœfficient de la ligne k, ce qui correspond à toutes les colonnes de numéros $j \in [\![i, n]\!]$
- une addition (add) et une multiplication (mult) permettant de mettre à jour le second membre de la ligne k, i.e. b_k.

Le nombre d'opérations (nop) avec prise en compte du second membre s'écrit donc

$$\begin{aligned}
nop &= \sum_{i=1}^{n-1}\sum_{k=i+1}^{n}\left[1.div + \left(\sum_{i=j}^{n} 1.add + 1.mult\right) + 1.add + 1.mult\right] \\
&= \sum_{i=1}^{n-1}\sum_{k=i+1}^{n}[1.div + (n-i+1).add + (n-i+1).mult] \\
&= \sum_{i=1}^{n-1}(n-i).div + (n-i).(n-i+1).add + (n-i).(n-i+1).mult \\
&= \sum_{i=1}^{n-1}(n-i).div + \sum_{i=1}^{n-1}(n-i).add + \sum_{i=1}^{n-1}(n-i)^2.add + \sum_{i=1}^{n-1}(n-i).mult + \sum_{i=1}^{n-1}(n-i)^2.mult \\
&= \frac{(n-1).n}{2}.div + \frac{(n-1).n}{2}.add + \frac{n.(n-1).(2n-1)}{6}.add + \frac{(n-1).n}{2}.mult + \frac{n.(n-1).(2n-1)}{6}.mult
\end{aligned}$$

4.3.2 Complexité en remontée

Considérons l'étape $i \in [\![1, n]\!]$. Le calcul de x_i par l'algorithme de remontée demande 1 division, $n-i$ additions et $n-i$ multiplications. De plus le calcul de x_n requiert 1 division. Ainsi :

$$nop = 1.div + \left(\sum_{i=1}^{n-1}(n-i).add + (n-i).mult + 1.div\right) = \frac{(n-1).n}{2}.add + \frac{(n-1).n}{2}mult + n.div$$

4.3.3 Complexité de l'ensemble

OPÉRATIONS	DIVISION	ADDITIONS	MULTIPLICATIONS
coût triangularisation	$\frac{n.(n-1)}{2}$	$\frac{n.(n-1).(n+1)}{3}$	$\frac{n.(n-1).(n+1)}{3}$
coût remontée	n	$\frac{n.(n-1)}{2}$	$\frac{n.(n-1)}{2}$
coût total	$\frac{n.(n+1)}{2}$	$\frac{n.(n-1).(2.n+5)}{6}$	$\frac{n.(n-1).(2.n+5)}{6}$

Au final la partie triangularisation est de l'ordre n^3 ($nop = O(n^3)$), la remontée est de l'ordre n^2 donc au total l'algorithme est de l'ordre 3 $O(n^3)$.

4.3.4 Complexité du pivotage

L'opération de pivotage nécessite lui aussi quelques opérations pour rechercher le meilleur pivot sur la ligne. A chaque ligne i, il faudra donc $n-i$ opérations élémentaires :

$$nop = \sum_{k=1}^{n-1} n-k = \sum_{k=1}^{n-1} k = \frac{n.(n-1)}{2} \text{ comparaisons}$$

Ceci ne modifie donc pas l'ordre de complexité de l'algorithme qui reste de 3.

4.4 Utilisation de librairies

Il est plus efficace d'utiliser des libraires que de reprogrammer soi-même des routines existantes.

D'un point de vu pratique, l'usage de librairie permet d'augmenter la lisibilité du code. De plus les librairies qui sont programmées de façon plus efficace sont souvent plus performantes.

Le sous-module **linalg** de **numpy** permet la résolution de systèmes linéaires.

En remplaçant toute la partie résolution par x=np.linalg.solve(A,b) on obtient les résultats suivants:

```
1 x=np.linalg.solve(A,b)
```

nombre de nœuds	5	13	49	149	449	999
Total	0.00399994850159	0.00399994850159	0.00699996948242	0.0160000324249	0.161999940872	0.923000097275
Mise en données	0.00399994850159	0.00399994850159	0.00600004196167	0.0120000839233	0.055999994278	0.180999994278
Résolution	0.0	0.0	0.000999927520752	0.00399994850159	0.105999946594	0.742000102997

TABLE 2 – Evolution du temps de calcul (en s) pour l'exemple du treillis FIG 1.

5 Application à un problème de treillis

La FIG 1 ci-contre représente un treillis de 5 nœuds et 7 barres. Chaque barre a une longueur de 6 m. La hauteur de l'ouvrage est de 3m.

Le nœud central est soumis à une charge de 15 kN (par exemple: effort exercé par un véhicule au centre du pont).

Par une application du PFS à l'ensemble de la structure on peut déterminer les efforts dans chaque appui.

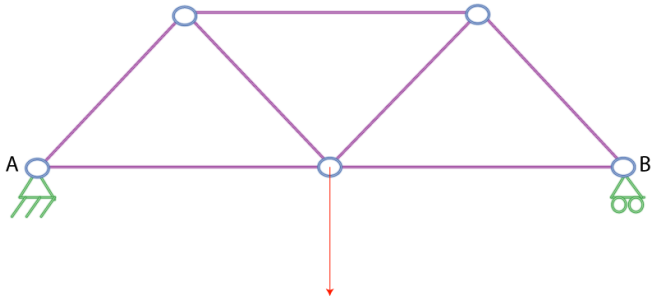

FIGURE 1 – Structure métallique treillis

Soit $-F.\vec{y}$ l'effort central. Alors $\overrightarrow{F_{ext_A}} = \frac{F}{2}.\vec{y}$ et $\overrightarrow{F_{ext_B}} = \frac{F}{2}.\vec{y}$.

Les inconnues que l'on cherche à déterminer sont les forces de traction qui s'exercent sur chaque barre.

L'application de l'équilibre à chacun des nœuds permet d'obtenir alors le système d'équations à résoudre.