ALGORITHMIQUE NUMÉRIQUE

Johann PAVIS
(selon source : J. ROMAN)

Table des matières

Chapitre 1

Introduction à l'algorithmique numérique

Il s'agit en général de traiter des problèmes de **calcul scientifique**.

Problème de taille Les problèmes posés par l'industrie sont souvent de très grande taille, d'où la nécessité d'utiliser des *solver* pour résoudre les systèmes du type $A.X = B$

Exemple : méthode de type Gauss ou de factorisation $\rightarrow$ complexité en $\mathcal{O}(n^3)$, avec n le nombre d'inconnues et si A est une "matrice pleine"

La météo pose des problèmes avec n pouvant facilement atteindre des valeurs de l'ordre de 10^6. On obtient donc 10^{18} opérations à effectuer sur des **réels** et 10^{12} coefficients à stocker.

Dans le cas des matrices creuses, par exemple une matrice tridiagonale, on obtient un nombre d'inconnues de l'ordre de 10^7 à 10^9. On va donc utiliser une algorithme dédiée à ce type de problème.

Nature du calcul Les réels sont stockés *via* un codage **approché**.

$\rightarrow$ pas de connaissance absolue des données
$\rightarrow$ les calculs peuvent être faux (problème d'arrondi)
$\rightarrow$ stabilité **numérique**

1.1 Problème de codage

1.1.1 Binaire et hexadécimal

1.1.1.1 Binaire

On utilise les deux symboles "1" et "0"

L'écriture binaire d'un nombre décimal est la somme des puissances de 2 successives. Il faut ici comprendre que le nombre décimal se calcule comme la somme des puissances successives de 2 (de 0 à l'infini) pondéré par le chiffre binaire correspondant à cette puissance.

Par exemple, le nombre binaire $\overline{1011010}^2$ s'écrit en décimal :

$$0 * 2^0 + 1 * 2^1 + 0 * 2^2 + 1 * 2^3 + 1 * 2^4 + 0 * 2^5 + 1 * 2^6 = 2 + 8 + 16 + 64 = 90_{10}$$

Le bits de *poids fort* est le chiffre le plus à gauche du nombre binaire (c'est le premier chiffre de ce nombre, si on le lit dans le sens conventionnel). Le bit de *poids faible* est le dernier chiffre du nombre binaire.

Propriété 1 *Soit p chiffres binaires. Le plus grand nombre décimal que l'on peut représenter avec ces p bits est l'entier :*

$$1 + 2 + 2^2 + \ldots + 2^{p-1} = \sum_{k=0}^{p-1} 2^k = 2^p - 1$$

Sur p chiffres binaires, on a 2^p configurations représentant des nombre entre 0 et $2^p - 1$.

On a un procédé automatique en ce qui concerne la multiplication (respectivement la division) par 2^n : il suffit de décaler les bits du nombre de n places vers la gauche (respectivement la droite).

1.1.1.2 Hexadécimal

On utilise seize symboles. Les 10 chiffres arabes usuels et les chiffres hexadécimaux A, B, C, D, E et F. Ces derniers ont pour valeurs décimales respectives 10, 11, 12, 13, 14 et 15.

L'écriture hexadécimale d'un nombre décimal s'obtient facilement à partir de son écriture binaire. Il suffit en effet de lire les chiffres binaires par groupe de 4 chiffres et de considérer ces 4 chiffres comme un nombre hexadécimal.

On trouve ainsi que : $90_{10} = \overline{01011010}^2 = 5A$[1]

1.1.2 Codage des entiers

1.1.2.1 Entiers non signés

1 info $\longrightarrow$ une valeur

Le codage est le codage binaire du nombre considéré.

1.1.2.2 Entiers signés

2 infos : *valeur* et *signe*

On utilise la méthode du complément à deux pour coder les entiers négatifs. Pour ce faire, on prend le complémentaire binaire du nombre à coder (pratiquement, il suffit de remplacer les 1 par des 0 et *vice versa* dans l'écriture binaire de la valeur absolue du nombre considéré), puis d'ajouter 1 à ce nombre binaire[2]

On notera la cohérence du calcul, puisque la somme du codage binaire du nombre (considéré comme positif) et de son complément à deux **est nulle**.

1.1.3 Codage des réels

Il faut coder sur une **taille donnée** un nombre fini de chiffres **significatifs**.

On peut coder en *simple* ou en *double* précision. On utilise une écriture du type exponentionelle :

$$\pm 0, m.10^{\pm e}$$

m est un nombre décimal de plusieurs chiffres et est appelé ***mantisse***. e est ***l'exposant***.

[1] $\overline{0101}^2 = 5_{16}$ et $\overline{1010}^2 = A_{16}$

[2] NB : on additionne en **binaire**

1.1.3.1 Simple précision

On utilise dans ce cas une mantisse de 24 bits, un bit de signe et 7 bits pour coder la caractéristique (un entier n tel que $0 \leqslant n + 64 \leqslant 127$)

La capacité de stockage est donc limitée.

La mantisse est un nombre hexadécimal à 6 chiffres[3]. Le nombre x codé est donc :

$$x = \pm \left[\frac{X_1}{16} + \frac{X_2}{16^2} + \frac{X_3}{16^3} + \frac{X_4}{16^4} + \frac{X_5}{16^5} + \frac{X_6}{16^6} \right] \times 16^{\mathbf{e}}$$

On remarquera que l'on a un problème d'unicité. En effet, on a : $1 = \frac{1}{16}.16$ et également $1 = \frac{1}{16^2}.16^2$

Dans le premier cas, on obtient que seul X_1 est non nul et vaut 1. Dans le second, c'est X_2 qui est non nul. Pour normaliser ceci, on choisit, lorsque plusieurs écritures sont possibles, celle pour laquelle $X_1 \neq 0$. Pour le zéro normalisé, l'exposant est choisi nul ($e = 0$).

1.1.3.2 Double précision

Dans ce cas, on augmente le nombre de chiffres significatifs. On les code sur 56 bits (soit un nombre hexadécimal à 14 chiffres).

Propriété 2 *En simple précision, on peut coder des nombres à 7 chiffres significatifs (pour 6 chiffres hexadécimaux). En double, on peut coder des nombres à 15 chiffres significatifs (pour 14 chiffres hexadécimaux)*

Cependant, on n'a pas de précision absolue sur les calculs et sur les données (du fait de possibles troncatures*)*

De plus, on peut faire des erreurs d'arrondis. Par exemple, si l'on fait la somme $10^9 + 10^{-9}$, le résultat sera erronnée puisqu'il donnera 10^9.

Exemple On veut calculer : $s = \sum_{n=1}^{m} \frac{1}{n^2}$

On peut procéder de deux manières : par une **boucle directe** ou bien par une **boucle retrograde**.

[3]On les note : $X_1, X_2, X_3, X_4, X_5, X_6$

Pour la boucle directe, on part de 1 et on va jusqu'à m. A partir de n assez grand, on va ajouter 0 à la somme. On obtient donc une **somme partielle**.

Pour la boucle rétrograde, la sommation se fait en partant de m et par pas de -1. On obtient un calcul plus équilibré, et **différent**[4] du résultat obtenu par boucle directe.

On voit donc que le "+" informatique ne commute à priori pas. De plus on a, ε représentant la précision de la machine[5] :

$$(a = 0) \;\; \mathsf{OU} \;\; (a = b) \Rightarrow (|a| \leqslant \varepsilon) \;\; \mathsf{OU} \;\; (|b - a| \leqslant \varepsilon)$$

1.2 Structure de données et prise en compte du coût d'accès aux informations

Il s'agit ici *d'adapter des algorithmes à l'organisation des données.*

1.2.1 Algorithme de remontée / descente pour résoudre $A.x = B$

On suppose ici A matrice symétrique définie positive et **pleine**. On sait donc que A est factorisable sous la forme $A = L.D.{}^tL$, L étant une matrice triangulaire inférieure de diagonale identité et D une matrice diagonale.

On obtient donc :

descente : résolution de $L.y = B$

résolution diagonale : résolution de $D.z = y$

remontée : ${}^tL.x = z$

[4] Et d'ailleurs plus juste
[5] 10^{-7} en simple précision et 10^{-15} en double précision

1.2.1.1 Descente : $L.y = B$

On a, pour i fixé : $b_i = y_i + \sum_{j=1}^{i-1} l_{i,j}.y_j$

On obtient donc l'algorithme suivant :

```
1.        y_1 := b_1
2.        Pour i de 1 à n faire
3.             y_i = b_i − Σ_{j=1}^{i−1} l_{i,j}.y_j ;
```

Complexité

Le nombre de multiplication est en $\mathcal{O}(n^2)$ $\left(\text{il y a exactement } \frac{n.(n-1)}{2} \text{ opérations}\right)$

Pour les additions et soustrations, c'est identiques : $\sum_{i=1}^{n} i - 1 = \mathcal{O}(n^2)$

On constate alors qu'il serait utile d'avoir ***un accès physique continu par ligne***.

1.2.1.2 Résolution diagonale : $D.z = y$

L'alogorithme utilisé est le suivant :

```
1.        Pour i de 1 à n faire
2.             z_i = y_i / d_{i,i} ;
```

La complexité de cet algorithme est en $\mathcal{O}(n)$

1.2.1.3 Remontée : ${}^tLx = z$

On obtient : $x_i = z_i - \sum_{j=i+1}^{n} l_{i,j}.y_j$

L'alogrithme utilisé est donc le suivant :

1. $\mathtt{x}_n = \mathtt{z}_n$;
2. <u>Pour</u> i <u>de</u> $n-1$ <u>à</u> 1 <u>par pas de</u> -1 <u>faire</u>
3. $\quad\mathtt{x}_i := \mathtt{z}_i - \sum_{j=i+1}^{n} {}^t\mathtt{l}_{i,j}.\mathtt{x}_j$;

Le nombre de multiplications est en $\mathcal{O}(n^2)$, de même que le nombre d'additions / soustractions.

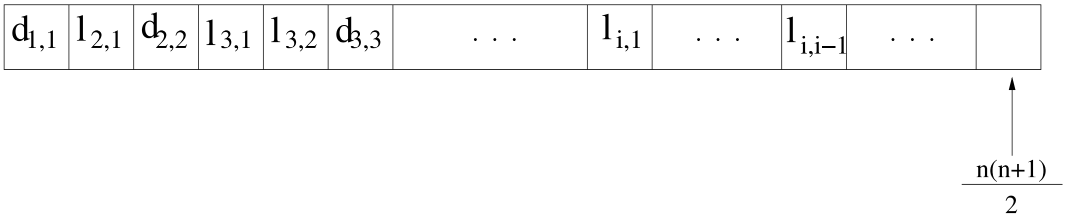
On a cependant ici un **accès par colonne** dans la matrice L, puisque ${}^tl_{i,j} = l_{j,i}$. Ceci est incompatible avec le stockage par ligne de nos données. On change donc l'algorithme de remontée de manière à avoir un accès par ligne.

La ligne 1. est inchangée. On change la suite de l'algorithme en ceci :

2. <u>Pour</u> i <u>de</u> 1 <u>à</u> n-1 <u>faire</u>
3. $\quad\mathtt{x}_i = \mathtt{z}_i$;
4. <u>Pour</u> i <u>de</u> n <u>à</u> 2 <u>par pas de</u> -1 <u>faire</u>
5. % $\mathtt{x}_i$ *est connu : le repercute sur tous les x_j d'indice inférieur*
6. $\quad$<u>Pour</u> j <u>de</u> i-1 <u>à</u> 1 <u>par pas de</u> -1 <u>faire</u>
7. $\quad\quad\mathtt{x}_j = \mathtt{x}_j - {}^t\mathtt{l}_{j,i}.\mathtt{x}_i$;

On alors un <u>un accès continu par ligne dans L</u>.

Pour exploiter la symétrie, on stocke la partie triangulaire inférieure de L, sans les 1 de la diagonale, dans un vecteur v et où on stocke continuement les lignes. v est un vecteur de taille $\frac{n.(n+1)}{2}$. On peut alors stocker les valeurs de la matrice D, soit les $d_{i,i}$, dans ce vecteur v (en les intercalant à la fin de chaque ligne de L), ou alors utiliser un autre vecteur.



1.2.2 Algorithme global

On utilisera les notations suivantes :

- `x` est un vecteur qui va contenir successivement dans le temps `b, y, z` et le `x` final
- `ad` est un pointeur donnant, à tout moment, l'adresse du prochain élément utilisé

L'algorithme dans son intégralité est le suivant :

```
x := b ;
% descente %
ad := 2 ;
Pour i de 2 à n faire
       début
            Pour j de 1 à i-1 faire
                 début
                      s := s + v[ad] * x[j] ;
                      ad := ad + 1 ;
                 fin ;
            x[i] := x[i] - s ;
            ad := ad + 1 ;
       fin ;
% résolution diagonale %
ad := 1 ;
Pour i de 1 à n faire
       début
            x[i] := x[i] / v[ad] ;
            ad := ad + 1 ;
       fin ;
```

```
% remontée : on a z dans x%
ad := n * n(n-1) / 2 - 1 ;
Pour i de n à 2 par −1 faire
       début
            Pour j de i-1 à 1 par −1 faire
                 début
                      x[j] := x[j] - v[ad] * x[i]
                     ad := ad - 1 ;
                 fin ;
            ad := ad - 1 ;
       fin ;
```

Chapitre 2

Problème du calcul de l'inverse d'une matrice

2.1 Méthode de **Sherman** et **Morison**

On considère une matrice $A \in \mathcal{GL}_n(\mathbb{R})$[1] On suppose connue la matrice A^{-1}. Soit B une permutation de A. On cherche à calculer $(A+B)^{-1}$.

Remarque D'un point de vue pratique, la matrice B contient un nombre important de 0. On effectue donc des modifications locales.

Propriété 3

$$(A+B)^{-1} = (I_n + A^{-1}B)^{-1}A^{-1}$$

Démonstration :

$C = (I + A^{-1}B)^{-1}A^{-1} = (A^{-1}A + A^{-1}B)^{-1}A^{-1}$

$C = (A^{-1}(A+B))^{-1}A^{-1}$

$C = ((A+B)^{-1}(A^{-1})^{-1})A^{-1}$

$C = (A+B)^{-1}AA^{-1}$

$C = (A+B)^{-1}$

[1] C'est à dire que A est une matrice **inversible**

Propriété 4 *Soit une matrice B une matrice "produit simple",* i.e. *de la forme : $B = x.^ty$.*

On a alors le résultat suivant :

$$(I + A^{-1}B)^{-1} = I + \frac{A^{-1}B}{1 + {}^tyA^{-1}x}$$

Bien sûr, cette égalité n'a de sens que si la fraction existe.

Application Soit A une matrice inversible, de matrice inverse A^{-1} connue. Soient i et j fixés. On ajoute ε à $a_{i,j}$.

On prend $B = \varepsilon.E_{i,j}$. On a alors :

$$(A + B)^{-1} = (A + \varepsilon.E_{i,j})^{-1} = A^{-1} + c\varepsilon A^{-1}e_i{}^te_jA^{-1}$$

On note $A^{-1} = (\alpha_{k,l})_{(k,l)\in[1,n]^2}$. On a alors :

$$\forall (k,l) \in [1,n]^2 \ , \quad (A + \varepsilon E_{i,j})^{-1} = \alpha_{k,l} + \varepsilon.\frac{\alpha_{k,l}.\alpha_{j,l}}{1 + \alpha_{i,j}}$$

2.2 Caclul général de l'inverse : méthode de Gauss avec pivotage partiel maximum

On cherche à résoudre : $Ax = b$, avec $A \in \mathcal{GL}_n(\mathbb{R})$.

On triangularise la matrice A : $MAx = Mb$. On doit donc résoudre un système : $Ux = b'$, avec U triangulaire supérieure.

En notant e_i le ième vecteur colonne de la base canonique de $\mathbb{R}^n$, on obtient, en résolvant $Ax = e_l$, que x est la colonne l de la matrice A^{-1}.

On a donc à effectuer :
- 1 triangularisation
- n remontées pour les systèmes $Ax = e_l$, $l \in [1,n]$

Ceci donne une complexité globale en $\mathcal{O}(n^3)$, c'est à dire une complexité équivalente à celle du produit de deux matrices.

Le principe de l'algorithme consiste à diviser la ligne k par le pivot, de manière à obtenir des 1 sur la diagonale. Ensuite, une triangularise la colonne

k. Notons que les opérations se font également sur le vecteur b (qui est en fait la $n+1$ème colonne de la matrice).

Ceci peut se généraliser au cas d'un nombre quelconque de seconds membres.

Ensuite, il reste à effectuer les n remontées. La matrice A^{-1} est obtenue dans la partie droite de la matrice.

La **stabilité numérique des calculs** est assurée en choisissant pour pivot à l'étape k le coefficient de plus grande valeur absolue de la colonne k, soit le coefficient de la ligne j, et de permuter les lignes j et k. Dans le cas où l'on trouve un coefficient de la matrice A tel que $|a_{i,j}| \leqslant \varepsilon$, la matrice est dite *non inversible à ε près*.

Par exemple, si l'on prend la matrice $A = \left(\dfrac{1}{i+j-1}\right)_{(i,j)\in[1,n]^2}$, les coefficients deviennent quasiment nul pour $n \simeq 15$.

Algorithme complet

```
Procédure pivot (k,l : entiers , ok : booléen)
       début
            max := |A[k,k]| ;
            l := k ;
            Pour i de k+1 à n faire
                 si max < |A[i,k]|
                      alors début
                           max := |A[i,k]| ;
                           l := i ;
                      fin ;
            ok := (max > ε) ;
       fin ;
```

```
Procédure permuta (k,l : entiers)
       début
            si k≠l
                 alors
                      pour j de k à 2*n faire
                           début
                                c := A[k,j] ;
                                A[k,j] := A[l,j] ;
                                A[l,j] := c ;
                           fin ;
       fin ;
Procédure calcul (A : matrice)
       début
            Pour k de 1 à n faire
                 début
                      pivot(k,l,ok) ;
                      si (¬ok)
                           alors début
                                écrire ‘‘Matrice non inversible’’ ;
                                aller à échec ;
                           fin ;
                      permeta(k,l) ;
                      Pour j de k+1 à 2*n faire
                           A[j,k] := A[k,j] / A[k,l] ;
                      Pour i de k+1 à n faire
                           Pour j de k+1 à 2*n faire
                                A[i,j] := A[i,j] - A[i,k]*A[k,j] ;
                 fin ;
            Pour k de 1 à n faire
                 Pour i de n-1 à 1 par −1 faire
                      début
                           s :=0 ;
                           Pour j de i+1 à n faire
                                s := s + A[i,j]*A[j,n+k] ;
                           A[i,n+k] := A[i,n+k] - s ;
                      fin ;
                 échec : fin ;
       fin ;
```

La complexité globale de cet algorithme est en $\mathcal{O}(n^3)$.

Chapitre 3

Méthodes de factorisation

3.1 Factorisation $L.U$

- L'objectif est de résoudre le problème $Ax = b$, avec A factorisée sous la forme d'un produit d'une **matrice triangulaire inférieure de diagonale identité** et d'une **matrice triangulaire supérieure stricte**
- Son existence est assurée par la méthode de Gauss
- L'unicité d'une telle décomposition tiens au fait que la matrice L est de diagonale identité

Il s'agit d'un problème de *remontées / descentes*. En effet, on résout les problèmes $Ly = b$, puis $Ux = y$.

Remarque Le second membre n'est pas modifié.

3.1.1 Calcul par identification

On va tenir compte compte du fait que U et L sont toutes deux **triangulaires**.

On sait déjà que la diagonale de L ne comporte qu'une seule valeur : 1. **On ne la calcule donc pas et on ne la stocke pas**.

On procède en deux temps :

1. On calcule les $l_{i,j}$ pour $j \in [1, i-1]$.

   On a : $a_{i,j} = \displaystyle\sum_{k=1}^{j} l_{i,k} u_{k,j}$.

On en déduit : $l_{i,j} = \dfrac{1}{u_{j,j}} \left[a_{i,j} - \sum_{k=1}^{j-1} l_{i,k} u_{k,j} \right]$

Remarquer que l'on peut faire le calcule "sur place", car on utilise plus les $a_{i,j}$ pour le calcul des coefficients suivants.

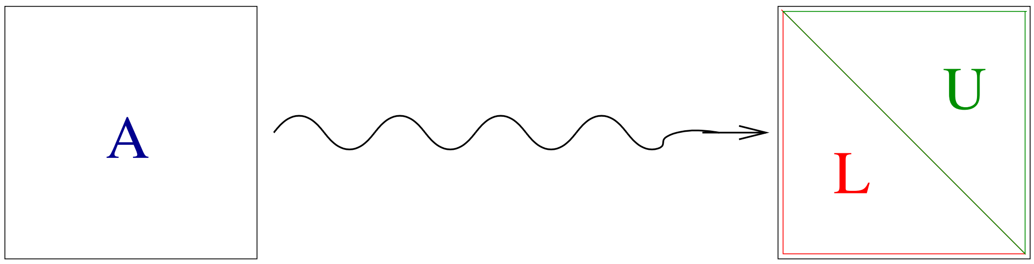
2. On calcule les $u_{i,j}$ pour $j \in [i, n]$.

   On a la même formule que précédemment. Ce qui permet de déduire aisément : $u_{i,j} = a_{i,j} - \sum_{k=1}^{i-1} l_{i,k} u_{k,j}$

3.1.2 Algorithme global

```
Pour i de 1 à n faire
       début
            Pour j de i-1 à n faire
                 calcul 1 ;
                 % l[i,i] := 1 ; %
            Pour j de i à n faire
                 calcul 2 ;
       fin ;
```

Le calcul se fait sur place :

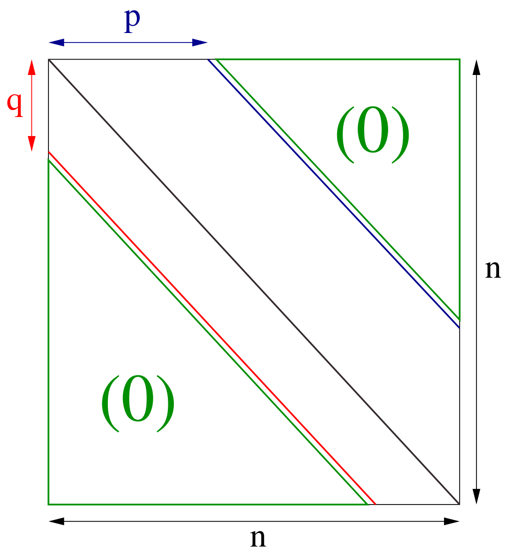


On peut avoir un autre point de vue :
- Accès *par ligne* dans L
- Accès *par colonne* dans U

La complexité globale en temps est en : $\mathcal{O}(n^3)$

3.1.3 Cas des matrices par bandes

Il s'agit des matrice de la forme suivante, p et q étant deux entiers non nécessairement égaux :



On appelle *largeur de bande* la somme $w = p + q$. On a un nombre de termes à stocker **de l'ordre de** $w.n$.

En pratique, on a :
- $w = 3$ $(p = q = 1)$ $\longrightarrow$ on parle de matrice *tridiagionale*
- $w = 5$ $(p = q = 2)$ $\longrightarrow$ on parle de matrice *pentadiagionale*

L'onjectif est clairement de ne devoir stocker que la bande. Le problème est de "*savoir ce qui se passe lors d'une factorisation LU sur une matrice par bande*".

Propriété fondamentale 5 *Lors d'une factorisation LU d'une matrice par bande,* ***la bande se conserve****.*

On on a donc une structure de données *statique et connue à l'avance*. On peut donc faire **un calcul sur place**

Remarque Le pivotage est une opération incompatible avec une gestion économique de la bande par un algorithme de type Gauss.

3.2 Cas des matrices symétriques définies positives

On notera désormais : matrice SDP.

<u>Théorème</u> 1 *Théorème de CHOLESKI*

Soit $A \in \mathcal{M}_n(\mathbb{R})$. On a alors la relation suivante :

$$A \text{ matrice SDP } \Leftrightarrow \exists!\ L \in \mathcal{T}_n^-(\mathbb{R})\ ,\ A = L.{}^tL$$

ou

$$A \text{ matrice SDP } \Leftrightarrow \exists!\ (L, D) \in \mathcal{T}_n^- \times \mathcal{D}_n\ ,\ L \text{ de diagonale identité, } A = L.D.{}^tL$$

Le premier cas correspond à une factorisation de CHOLESKI. Le second à une factorisation de CHOLESKI - CROUT

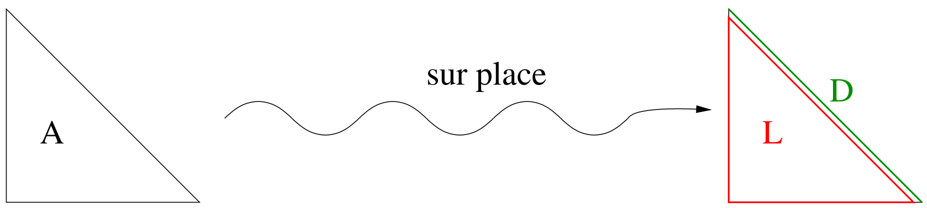
Comparaison

- même comportement numérique
- Choleski nécessite n calculs de racine carrée supplémentaires
- pour la remontée / descente $Ax = b$, on résout :
  - soit $Ly = b$, puis ${}^tLx = y$
  - soit $Ly = b$, $Dz = y$, puis ${}^tLx = z$

  Choleski nécessite donc n divisions supplémentaires.
- la factorisation de Choleski - Crout est valable même si $A \in \mathcal{M}_n(\mathbb{C})$, *i.e.* sur les matrices à coefficients complexes.

3.2.1 Calcul de L et D à partir de A par identification

On exploite la symétrie de A :



On ne stocke ni ne calcule les 1 de la diagonale de L. On n'utilise pas ${}^tL$

3.2.1.1 Démarche de la factorisation LU

On identifie la matrice U à la matrice $D.^tL$.

On a donc : $\forall(k,j) \in [1,n]^2 \ , \quad u_{k,j} = d_{k,k}.l_{j,k}$

On utilise l'algorithme par ligne de la factorisation LU avec, à chaque ligne i, le calcul des $l_{i,j}$ avec $j \in [1, i-1]$ et le calcule des $u_{i,i}$ (pour le calcul des $d_{i,i}$). Ce qui donne donc :

```
Pour i de 1 à n faire
    début
        Pour j de 1 à i-1 faire
            l[i,j] := [a[i,j] − Σ_{k=1}^{j−1} l[i,k]*d[k,k]*l[j,k]] 1/(l[j,j]*d[j,j]) ;
        % l[i,i] := 1 ; %
        d[i,i] := a[i,i] - Σ_{k=1}^{i−1} l[i,k]*l[i,k]*d[k,k] ;
    fin ;
```

Les accès dans A et dans L se font par ***ligne***. On doit donc utiliser **une structure de données par ligne**.

3.2.1.2 Optimisation du nombre d'opérations

On pose : $\forall(i,j) \ , \ l_{i,j}.d_{j,j} = c_j$. On a donc :

$$\forall(i,j) \in [1,n]^2 \ , \quad c_j = a_{i,j} - \sum_{k=1}^{j-1} c_k.l_{j,k}$$

On effectue plus qu'une seule multiplication au lieu de deux comme précédemment. Il faut cependant créer un vecteur de taille $n-1$ supplémentaire pour stocker les c_j.

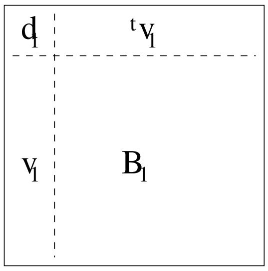
On utilise alors l'alogorithme suivant :

```
Pour i de 1 à n faire
    début
        Pour j de 1 à i-1 faire
            début
                c[j] := a[i,j] - $\sum_{k=1}^{j-1}$ c[k]*l[j,k] ;
                l[i,j] := c[j] / d[j,j] ;
            fin ;
        d[i,i] := a[i,i] - $\sum_{k=1}^{i-1}$ c[k]*l[i,k] ;
    fin ;
```

On a alors une perte en mémoire en $\mathcal{O}(n)$ mais un gain en temps en $\mathcal{O}(n^3)$!

3.2.2 Algorithme par colonnes

On procède par "*découpage*" de la matrice en blocs. On procède comme suit :



Sur le schéma précédent :
- d_1 est un **scalaire**[1]
- v_1 est un vecteur colonne de taille $n-1$
- B_1 est une matrice carrée de taille $n-1$

[1] C'est à dire $d_1 \in \mathbb{R}$

On note A_i la matrice obtenue après le découpage numéro i. On écrit cette matrice A_i comme le produit de trois matrices. Pour A_1, on a donc :

- La matrice "centrale" du produit possède une seul coefficient non nul sur sa première colonne et sa première ligne (d_1). La sous-matrice intérieure est la matrice B'_1 définie par : $B'_1 = B_1 - \frac{v_1 \cdot {}^t v_1}{d_1}$
- La première matrice est une matrice triangulaire inférieure, de diagonale identité. Sa première colonne est le vecteur $\frac{1}{d_1}.v_1$ et sa sous-matrice intérieure est la matrice identité de taille $n-1$.
- La troisième matrice est la matrice transposée de la première.

On pose : $A_1 = L_1.A_2.{}^tL_1$. On découpe alors la sous-matrice B'_1 de A_2 et ainsi de suite.

On remarquera que les matrices A_i sont *symétriques définies positives*.

En itérant $n-1$ le procédé, on obtient : $A = (L_1.L_2 \ldots L_n)A_n{}^t(L_1.L_2 \ldots L_n)$, où A_n est une matrice diagonale.

On pose alors : $A_n = D$ et $L = \prod_{i=1}^{n} L_i$. On remarquera également que l'on a l'égalité suivante : $L = \sum_{i=1}^{n-1} L_i - (n-2) \cdot I$

Algorithme utilisé

```
Pour k de 1 à n-1 faire
       début
            Pour j de k+1 à n faire
                 début
                      d := a[j,k] / a[k,k] ;
                      Pour i de j à n faire
                           a[i,j] := a[i,j] - a[i,k]*d ;
                      a[j,k] := d ;
                 fin ;
       fin ;
```

On doit avoir une structure de données permettant **de stocker continuement les colonnes** de la matrice.

Une condition de modification d'un coefficient $a_{i,j}$ dans la matrice B'_k est : $a_{j,k} \cdot a_{j,k} \neq 0$

Chapitre 4

Étude du remplissage lié à la factorisation de CHOLESKI - CROUT

Dans tout ce qui suit, la matrice A est supposée carrée (de dimension n) symétrique définie positive. De plus, on suppose la matrice A **creuse**[1].

On dit qu'une matrice M est creuse si le stockage de ses coefficients et des informations supplémentaires est inférieur à n^2, n désignat la dimension de la matrice.

On veut étudier le problème du remplissage dans le cadre de la résolution de $Ax = b$ par la factorisation de Choleski - Crout.

4.1 Problème du remplissage

Un terme (i, j) est dit de remplissage si $a_{i,j} = 0$ et $l_{i,j} \neq 0$.

On peut ainsi se retrouver à devoir stocker n^2 termes alors qu'au départ on n'en avait que $2n - 1$.

[1] C'est à dire qu'un grand nombre de ses coefficients sont nuls.

4.2 Notion de renumérotation

Il s'agit de permuter les inconnues.

On définit une permutation σ de $[1, n]$ dans lui-même. Par exemple, pour $n = 5$, on pose : $\sigma(1) = 5$, $\sigma(5) = 1$ et $\sigma(i) = i$ pour i différent de 1 et 5. On note P la matrice de permutation associée à σ. On trnasforme A en $P.A.{}^tP$.

On rappelle que[2] : $\forall(i,j) \in [1,n]^2$, $P_{i,j} = \delta_{i,j}$. De plus, P est une matrice orthogonale. Donc : $P.{}^tP = I_n$.

Donc, le problème $Ax = b$ est équivalent au problème $P.Ax = P.b$

Le remplissage dépend de l'ordre selon lequel on fait la "boucle externe en k", c'est à dire de la numérotation des inconnues.

Le problème de savoir s'il existe une permutation σ minimisant le remplissage est un NP complet. On va donc se limiter à la recherche de solutions "sous-optimales", mais raisonnable en temps.

On introduit à cet effet la notion de *graphe de liaison*.

Définition 1 *Soit $A \in \mathcal{M}_n(\mathbb{R})$ une matrice SDP. Soit σ une numérotation des inconnues.*

On lui associe un graphe $G = (V, E, \sigma)$ non orienté tel que :

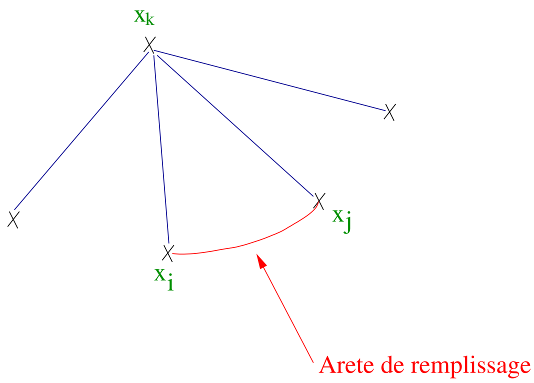
- $|V| = n$
- $\sigma : \left| \begin{array}{l} [1,n] \to V \\ \sigma(i) = x_i \end{array} \right.$
- $(x_i, x_j) \in E \ , \ i > j \Leftrightarrow a_{i,j} \neq 0$

Définition 2 *Le* graphe d'élimination *est le graphe $G^* = (V, E^*, \sigma)$ associé à la matrice L obtenue par factorisation. On a $E^* = E \cup \mathcal{R}$, où $\mathcal{R}$ est l'ensemble des arrêtes de remplissages correspondant aux termes non nuls crés.*

Le problème est maintenant de caractériser et de calculer cet ensemble E^*. Par le calcul, on trouve : $a_{i,j} = a_{i,j} - \dfrac{a_{j,k}.a_{i,k}}{a_{k,k}}$. Le terme $a_{i,j}$ ne devient non nul que si les deux termes du dénominateur de la fraction sont non nuls simultanément.

[2] $\delta_{i,j}$ est le symbole de Kronecker usuel

Au nivau du graphe, les arrêtes de remplissage sont les arrêtes rouges :



4.3 Notations

On va définir les ensembles d'adjacences : $Madj(k)$. On pose donc :

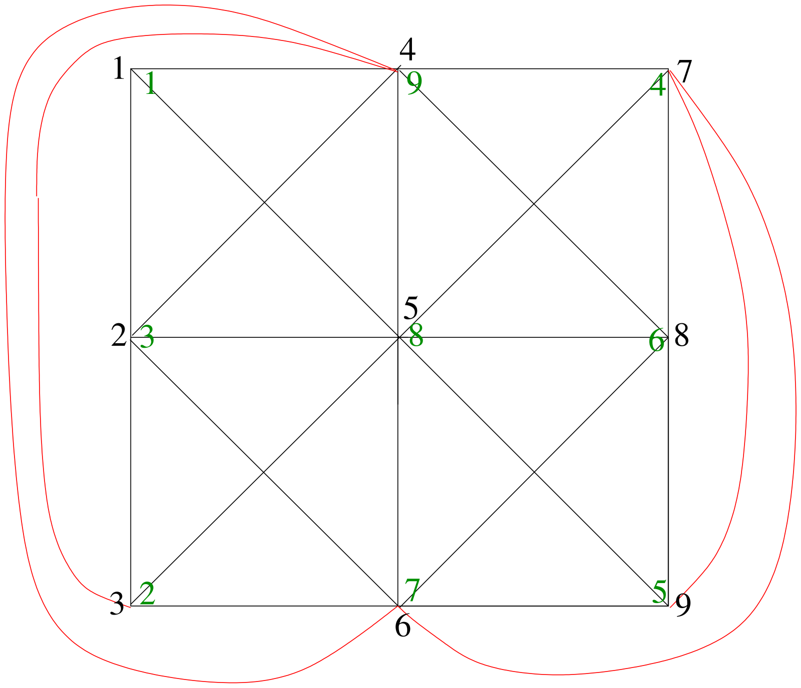
$$\forall k \in [1, n-1] \ , \ Madj(k) = \{x_l / x_l \text{ et } x_k \text{ adjacents et } l > k\}$$

Voici l'algorithle qui calcule ces ensembles :

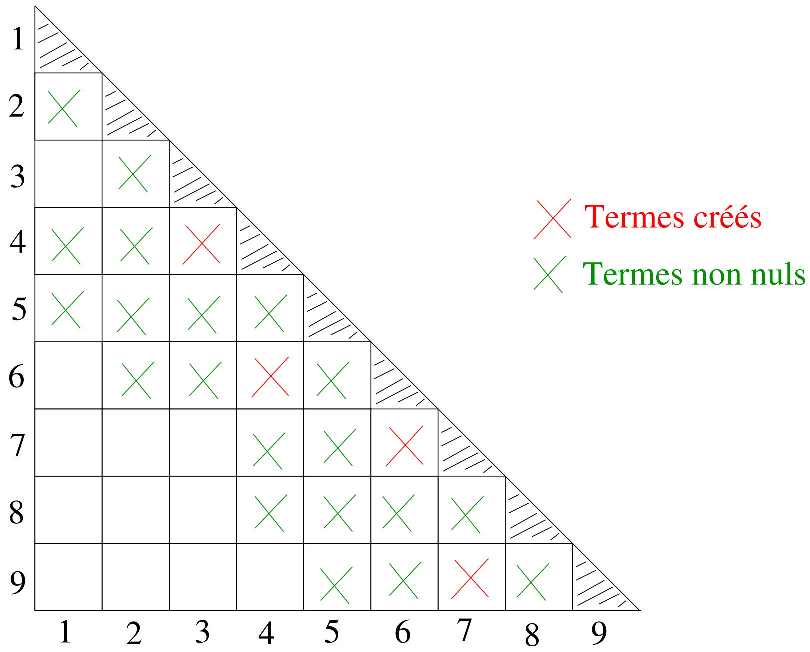
```
Pour k de 1 à n-1 faire
       début
            calculer Madj(k) ;
            rajouter des arêtes entre sommet de Madj(k) de
            manière que tout couple de sommets soit connecté ;
       fin ;
```

A la fin on a la structure du graphe G^*_σ : c'est la *fatorisation symbolique.*

Exemple On prend $n = 9$



On obtient la matrice suivante (numérotation noire) :



Cependant, en prenant la numérotation verte, on ne créera qu'un seul terme !

Remarque sur la complexité Pour ajouter des arêtes durant la construction de Madj(k), on a une complexité en $\mathcal{O}\left[\text{Madj}(k)\right]$.

La complexité globale est une complexité cubique, c'est à dire celle de la factorisation numérique. Cet algo **est trop cher**.

4.4 Factorisation symbolique

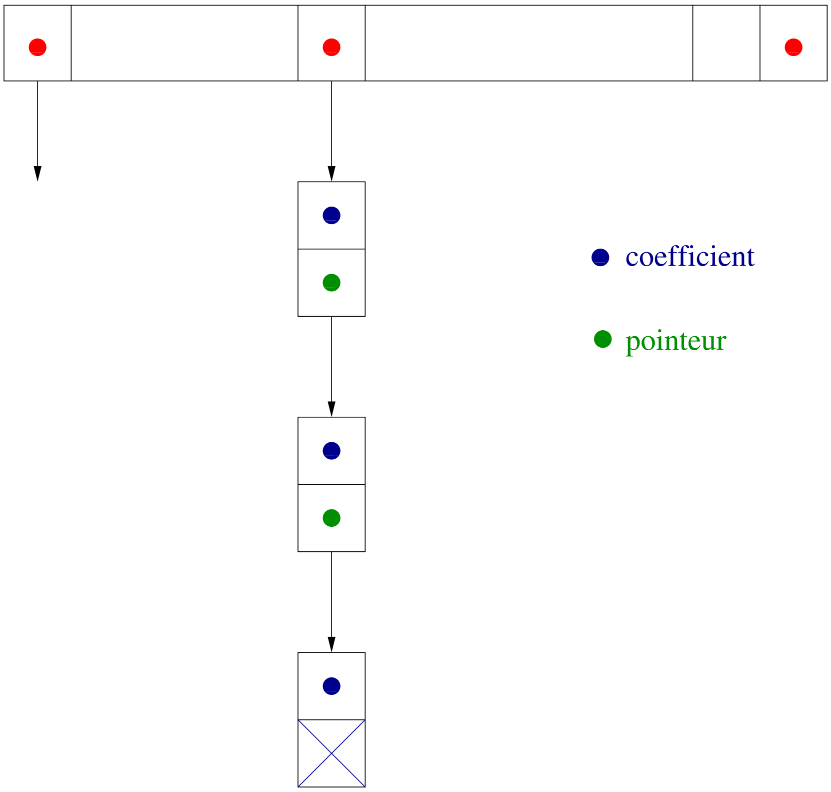
On va créer des listes monotones d'adjacence, triées par ordre de numéro croissant. Voici l'algorithme utilisé :

```
début
    Pour k de 1 à n-1 faire
         construire la liste triée Madj(k) relativement à σ
    Pour k de 1 à n-1 faire
         début
              m[k] := tête(Madj(k)) ;
              Madj(k) :=fusion(Madj(x[m[k]]),Madj(x[k])-{m[k]}) ;
         fin ;
```

La complexité d'un tel algorithme est quadratique. Le *prétraitement est rapide*. A la fin, on obtient les listes monotones d'adjacence relatives à $G_\sigma^*(L)$.

Aspect structure de données

Les matrices sont stockées sous la forme d'un ensemble de listes chaînées ne contenant par colonne que les termes non nuls :



Si on a m termes non nuls dans la matrice L, on a :
- $m + n$ pointeurs
- m coefficients réels

Ensuite, il faut adapter l'algorithme par colonne pour ne faire les calculs que sur les termes non nuls présents dans la structure.

4.5 Approche du profil ligne

<u>Définition</u> 3 *Soit une matrice carré A de taille n.*

Pour tout i dans $[1, n]$, on définit l'indice c_i de colonne par :

$$\forall k < i \ , \ a_{i,k} = 0$$

c_i est l'indice du permier coefficient non nul de la colonne.

On définit la **contribution au profil ligne** *de A de la ligne i est l'ensemble des coefficients entre les colonnes c_i et i, soit donc $i - c_i + 1$ termes.*

Le profil est ***l'ensemble de ces vecteurs de contributions mis bout à bout****.*

Remarque Il suffit de connaître c_i pour identifier tous les coefficients présents au titre de la ligne i.

La taille des profils est : $m = \sum_{i=1}^{n} i - c_i + 1$

Propriété fondamentale 6 *Le profil ligne se conserve au cours de la factorisation. On a ainsi :* $\text{Profil}(A) = \text{Profil}(L)$*. On en déduit que l'ensemble des termes de remplissage est inclus dans le profil de A :*

$$\mathcal{R} \subset \text{Profil}(A)$$

En minimisant le profil, on diminue le remplissage (il y a en effet une renumérotation). Le profil est facile à implémenter et ne nécessite pas un grand nombre de pointeurs[3].

Adaptation de l'algorithme par ligne de la factorisation au profil

On effectue le calcul suivant pour les $l_{i,j}$:

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

Minimisation du profil ligne

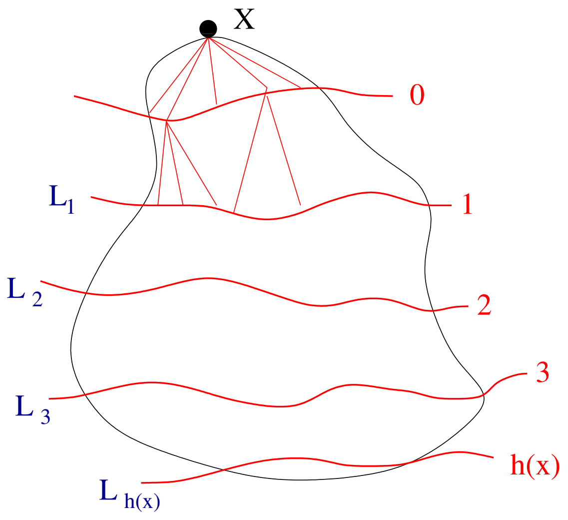
On veut diminuer, autant que faire se peut, la valeur de $m = \sum_{i=1}^{n} i - c_i + 1$. Sur le graphe, cela correspond à minimiser **l'écart d'indice entre deux sommets adjacents**.

Le problème qui vise à trouver une numérotation minimisant l'écart entre sommets adjacents est un problème *NP-complet*. On va donc chercher une solution **quasi**-optimale.

[3]Nombre en $\mathcal{O}(n)$

Numérotation par couche

On part d'un sommet quelconque, soit x. On construit alors une structure en niveau à partir de x :



L_k est l'ensemble des sommets situés sur la couche d'indice k, soit l'ensemble **des sommets situés à une distance k du sommet x**.

Le calcul d'un arbre recouvrant en largeur (avec une file) enraciné en x se fait en $\mathcal{O}$(nbr d'aretes).

On numérote chaque couche de gauche à droite, couche après couche. L'écart moyen de numéro entre deux couches est lié à la largeur moyenne de ces couches. On a donc intérêt à ce que toutes les couches soient aussi petites que possible.

Comme le nombre de sommets est fixe, il faut avoir le plus grand nombre possible de couches.

On cherche un sommet x tel que $h(x)$ soit quasi-maximal. Il faut donc trouver le "*bon sommet*".

On appelle ***sommet périphérique*** un sommet x qui réalise $h(x)$ maximum. Le seul algorithme qui construit un sommet périphérique est celui qui teste tous les sommets successivement. Il a une complexité en $\mathcal{O}(|V| \cdot |E|)$. Cet algorithme est évidemment trop coûteux.

A nouveau, on va chercher une solution algorithmique pseudo-optimale. On calcule, étant donné un sommet x, $h(x)$. On choisit ensuite un sommet y de la dernière couche : $L_{h(x)}$. On réitère le procédé (calcul des couches) à partir de ce sommet y. Si $h(x) > h(y)$, c'est terminé. Sinon, on recommence avec

y', un sommet de la couche $L_{h(y)}$. L'algorithme ce termine en cas d'égalité où si le nombre de couches trouvées est plus petit que lors de l'itération précédente.

Le nombre d'itération est faible[4].

[4] Il est en $\mathcal{O}(1)$