Compte rendu des tp 4 et 5

SEWANOU LEODASCE

December 20, 2021

Abstract

Part I

TP 4

Chapter 1

Factorisation de Cholesky

RAPPEL

Définitions :
Soit A une matrice réelle d'ordre n.

- A est dite symétrique si et seulement si $A = A^T$.
- A est positive si elle à toutes valeurs propres positives.
- A est définie si $Ax = 0$ implique que $x = 0$. Attention 0 ici représente le vecteur colonne de taille n dont tous les coefficients sont nuls.
- Si A est symétrique alors toutes ses valeurs propres sont réelles.
- Si A est symétrique définie positive alors ses valeurs propres sont réelles et toutes strictement positives.

Théorème :
Si A est une matrice *réelle Symétrique définie et positive* alors il existe une unique matrice *réelle triangulaire inférieure* L telle que tous ses éléments diagonaux sont strictement positifs et qui vérifie tels $A = LL^T$.

Preuve : L'objectif de ce rapport n'étant pas de faire de l'algèbre linéaire, nous renvoyons ceux qui sont inéréssés par la preuve vers le dépôts suivant:

Algorithme

complexité arithmétique de la décomposition de Cholesky:

Dans le bloc 1 on a 1 opération élémentaire : qui est la racine carrée.
Dans le bloc 2 on a n-1 divisons (/) .
Dans le bloc 3 on a:

- Dans le sous bloc 4

  on a pour chaque valeur de k :

  - 1 racine carré

- 1 soustraction
- k-1 multiplications et k-2 additions dans le produit scalaire.

Ainsi pour chaque k on a 2k - 1 opérations .

- Dans le sous bloc 5

  on a pour chaque i fixé,

  - une division (/)
  - une soustraction (-)
  - k-1 multiplications et k-2 additions.

  Donc pour chaque i fixé on a 2k-1 opérations.

$$\sum_{i=k+1}^{n} (2k-1) = (2k-1) * (n-k)$$

En somme dans le bloc 3 on a :

$$\sum_{k=2}^{n} ((2k-1) + (2k-1) * (n-k)) = \frac{n^3}{3} + \frac{n^2}{2} - \frac{5n}{6}$$

Enfin on a

$$\frac{n^3}{3} + \frac{n^2}{2} - \frac{5n}{6} + n = \frac{n^3}{3} + \frac{n^2}{2} + \frac{n}{6}$$

opérations.

complexité en espace mémoire de la décomposition de Cholesky:

Si nous optons pour un algorithme en place, c'est-à-dire si nous écrasons L dans A pour ainsi prendre par la suite la partie inférieure de A, alors il nous faudra n^2 mots-mémoires pour stocker A.

Remarquons au passage que ce faisant, on peut reconstruire A puisque A est symétrique et nous avions toujours la partie triangulaire strictement supérieure de A. Il suffira donc de calculer que les éléments diagonaux de A.

test et validation de l'algorithme de Cholesky:

Pour le test nous allons généré dans un premier temps une matrice A de façon aléatoire, ensuite nous calculons une matrice définie symétrique et positive $B = AA^T$.

$$A = \begin{pmatrix} 0.7560439 & 0.6283918 & 0.068374 & 0.1985144 \\ 0.0002211 & 0.8497452 & 0.5608486 & 0.5442573 \\ 0.3303271 & 0.685731 & 0.6623569 & 0.2320748 \\ 0.6653811 & 0.8782165 & 0.7263507 & 0.2312237 \end{pmatrix}$$

$$B = \begin{pmatrix} 1.0105615 & 0.6805305 & 0.7720077 & 1.1504861 \\ 0.6805305 & 1.3328342 & 1.0805601 & 1.2796254 \\ 0.7720077 & 1.0805601 & 1.0719184 & 1.3567783 \\ 1.1504861 & 1.2796254 & 1.3567783 & 1.7950459 \end{pmatrix}$$

Nous allons comparer la sortie de notre algorithme à celle de la procédure *chol* de scilab. la matrice L donnée par la procédure *chol* est noté L1 et celle donnée par notre algorithme est noté L2. La procédure *chol* renvoie une matrice triangulaire supérieure.

$$L1 = \begin{pmatrix} 1.0052669 & 0.676965 & 0.7679629 & 1.1444583 \\ 0. & 0.9351752 & 0.5995412 & 0.5398637 \\ 0. & 0. & 0.350288 & 0.4402268 \\ 0. & 0. & 0. & 0.0029314 \end{pmatrix}$$

$$L2 = \begin{pmatrix} 1.0052669 & 0. & 0. & 0. \\ 0.676965 & 0.9351752 & 0. & 0. \\ 0.7679629 & 0.5995412 & 0.350288 & 0. \\ 1.1444583 & 0.5398637 & 0.4402268 & 0.0029314 \end{pmatrix}$$

$$L1 - L2^T = \begin{pmatrix} 0. & 0. & 0. & 0. \\ 0. & 0. & 0. & 0. \\ 0. & 0. & 0. & 0. \\ 0. & 0. & 0. & 1.420D-14 \end{pmatrix}$$

Notre algorithme pourra donc être utilisé pour la Factorisation LL^T. Mais allons un peu plus loins en vérifinant la cohérences entre les complexités théorique et empirique.

n	temps d'exécution en seconde	espace mémoires
10	0.000295	1,0 kB
50	0.00833	20,2 kB
100	0.0551	80,2 kB
200	0.562	320,2 kB
500	29	2,0 MB

Tableau 1: Distribution de la complexité temporelle et de la complexité en spaces suivants la taille des matrices.

Notre complexité arithmétique théorique nous dit que si nous multiplions la taille de la matrice par n alors le nombre d'opérations sera multiplié par n^3 environ. Donc partant, le temps d'exécution devra suivre la même loi. Dans le tableau 1 nous voyons que pour n = 10 et n = 200 cette loi ce confirme à facteurs près .En effet n'oublions pas que les optimisations relatives à l'architecture de la machine notament du processeur que nous ne maitrisons pas ont une incidence sur le temps d'exécution. Ce qui couplé aux terme quadratiques et linéaires dans le calcul de la complexité théoriquee font que nous n'avons pas forcément un résultat juste juste.

Compte tenu des résultats ci-dessus nous validons notre algorithme.

Chapter 2

Factorisation LDL^T

RAPPEL

Théorème : Si A est une matrice *réelle inverisble et à tous ses sous-matrice principales de déterminant non nul*, alors il existe une unique matrice triangulaire inférieure L avec des 1 sur la diagonale et une unique matrice diagonale D, telle que $A = LDL^T$.

Remarque

Comme on peut le voir la Factorisation LL^T est très exigeante à cause des hypothèses fortes (*A est symétrique, définie et positive*) . Par ailleurs, dans l'implémentation de l'algorithme, le calcul des racines carrés impose des radicandes positives. Or à cause des erreurs d'arrondi on peut avoir des nombres négatifs sous la radicale au quel cas l'algorithme se verra arrêté. Ce qui est très probable surtout si la matrice est très mal conditionée. La Factorisation LDL^T permet d'échapper à cet éceuil.

Pour chaque j fixé on a :

- Dans le bloc 1 on a (j-1) multiplications.
- Dans le bloc 2 on a :
  - (j-1) multiplications
  - (j-2) additions
  - 1 soustraction.

  Donc dans ce bloc on a (2j-2) opérations.
- Dans le bloc 3 on a :
  - 1 division
  - (n-j)*(j-1) multiplications et (n-j)*(j-2) additions pour le produit matrice-vecteur.
  - (n-j) soustractions

  Donc dans ce bloc on a $(n-j)*(2j-2)+1$ opérations.

En somme on a :

$$\sum_{j=1}^{n}((3j-3)+(n-j)*(2j-2)+1)=\frac{n^3}{3}+\frac{n^2}{2}+\frac{n}{6}$$

opérations.

complexité en espace mémoire la décomposition LDL^T :

Si nous optons pour un algorithme en place, il nous faut seulement stocker A. Donc il faut n^2 mots-mémoires.

test et validation de l'algorithme de LDL^T :

Pour le test nous allons généré dans un premier temps une matrice A de façon aléatoire, ensuite nous calculons une matrice définie symétrique et positive $B = AA^T$.

$$A=\begin{pmatrix}0.6241452 & 0.6036367 & 0.690612 & 0.8774892\\ 0.8572099 & 0.0704719 & 0.3679751 & 0.2234013\\ 0.8582152 & 0.9067191 & 0.0513523 & 0.4883557\\ 0.5949214 & 0.2276711 & 0.4857946 & 0.3172589\end{pmatrix}$$

$$B=\begin{pmatrix}2.0008668 & 1.0277231 & 1.5469712 & 1.1226348\\ 1.0277231 & 0.9250889 & 0.9275644 & 0.7756533\\ 1.5469712 & 0.9275644 & 1.7998012 & 0.8968862\\ 1.1226348 & 0.7756533 & 0.8968862 & 0.7424152\end{pmatrix}$$

Ci-dessous, nous avons les résultats de notre algorithme sur B.

$$L=\begin{pmatrix}1. & 0. & 0. & 0.\\ 0.5136389 & 1. & 0. & 0.\\ 0.7731505 & 0.3347843 & 1. & 0.\\ 0.5610742 & 0.5010554 & -0.0674302 & 1.\end{pmatrix}$$

$$D=\begin{pmatrix}2.0008668 & 0. & 0. & 0.\\ 0. & 0.3972103 & 0. & 0.\\ 0. & 0. & 0.5592401 & 0.\\ 0. & 0. & 0. & 0.0102687\end{pmatrix}$$

$$B-L*D*L^T=\begin{pmatrix}0. & 0. & 0. & 0.\\ 0. & 0. & 0. & 0.\\ 0. & 0. & 0. & 0.\\ 0. & 0. & 0. & 0.\end{pmatrix}$$

n	temps d'exécution en seconde	espace mémoires
10	0.000265	1,0 kB
50	0.00112	20,2 kB
100	0.00568	80,2 kB
200	0.0438	320,2 kB
500	0.629	2,0 MB
1000	4.85	8,0 MB

Tableau 2: Distribution de la complexité temporelle et de la complexité en spaces suivants la taille des matrices.

Notre complexité arithmétique théorique nous dit que si nous multiplions la taille de la matrice par n alors le nombre d'opérations sera multiplié par n^3 environ. Donc partant, le temps d'exécution devra suivre la même loi. Dans le tableau 2 nous voyons que pour n = 10 et n = 500 cette loi ce confirme à facteurs près .En effet n'oublions pas que les optimisations relatives à l'architecture de la machine notament du processeur que nous ne maitrisons pas ont une incidence sur le temps d'exécution. Ce qui couplé aux terme quadratiques et linéaires dans le calcul de la complexité théoriquee font que nous n'avons pas forcément un résultat juste juste. Par ailleurs, pour n = 1000 on voit que nous somme trop loins de la complexité théorique en temps espérés. Nous expliquons ceci par des optimisations faites par le processeurs (que nous ne maitrisons pas).

Compte tenu des résultats ci-dessus nous validons notre algorithme.

Réolutions de systèmes linéaires $Ax = b$:

- LU

  $$\begin{cases} Ly = b \\ Ux = y \end{cases}$$

  une descente(appel de lsolve) suivi d'une remonté(appel de usolve).

- LL^T

  $$\begin{cases} Ly = b \\ L^T x = y \end{cases}$$

  une descente(appel de lsolve) suivi d'une remonté(appel de usolve).

- LDL^T

  $$\begin{cases} Lz = b \\ Dy = z \\ L^T x = y \end{cases}$$

  une descente(appel de lsolve), ensuite n égalités(appel de inv) suivi d'une remonté(appel de usolve.

Comparaison LU, LL^T et LDL^T :

Pour la comparaison nous allons pour diferentes tailles de matrice résoudre le système $Ax = b$, avec les trois algorithmes. Pour ce faire :

- nous allons commencée par fixer une taille n;
- généré une matrice aléatoire A avec la fonction rand() puis une matrice symétrique définie positive $B = A * A^T$;
- généré un vecteur xex de taille n avec rand();
- généré $b = B * xex$;
- résoudre $Bx = b$ avec les différéntes algorithmes;

- calculer l'erreur arrière (l'erreur par rapport aux données ayant conduit à la solution), pour évaluer la précision numérique;
- calculer l'erreur avant (l'écart à la solution exacte).
- ensuite calculer le conditionnement et le produit du conditionnement par l'erreur arrière afin dans un premier temps avoir une idée de la capacité de la matrice à généré du bruit et dans un second temps obtenir une majoration de l'erreur avant.

La procédure ci-dessus sera répété pour differéntes valeur de n.

Enfin nous allons tracer les courbes d'évolutions de la complexité en temporelle des trois méthodes de résolutions.

n	tempsChol	temps lu	temps ldlt	rlschol	rlslu	rlsldlt	errchol	errlu	err
10	0.00955	0.0182	0.0195	1.53e-16	1.44e-16	1.71e-16	2.38e-13	2.14e-13	2.33
50	0.0167	0.0188	0.0333	2.63e-16	3.99e-16	2.74e-16	1.73e-08	3.74e-08	3.56
100	0.0649	0.478	0.0503	4.31e-16	1.25e-05	3.87e-16	1.42e-10	0.858	2.03
200	0.588	3.61	0.0913	5.42e-16	1.35e-07	6.34e-16	2.46e-09	1	2.35
500	30.5	56.2	0.624	7.37e-16	3.91e-08	1.08e-15	1.83e-08	0.993	1.26
1000	728	454	4.92	9.62e-16	1.04e-15	9.39e-16	5.27e-08	3.64e-08	1.22

Tableau 3: Comparaison LU, LL^T et LDL^T

L'analyse de ce tableau révèles plusieurs choses. La première en analysant les colonnes relatives aux temps d'éxéxution , il en ressort que LDL^T est globalement 100 fois plus rapide que Cholesky et LU. Aussi , observe-t-on que LU prend en moyenne 2 fois plus temps que Cholesky. Par ailleurs, si numériquement , ldlt et Cholesky restent globalement *stable numériquement* (cf. les colonnes relatives aux erreurs arrières), tel n'est pas le cas pour LU.En effet à partir de n=100, LU perds en précision numérique. Par contre globalement l'écart à la solution exacte augmente avec la taille des matrices. surtout avec la méthode LU. Nous concluons donc que ces algorithme bien que pouvant être utilisé sur de petites matrices ($n < 100$), ils sont par assez fiables pour résoudres des problèmes de grandes tailles. Ce sont donc des algorithmes naives à but pédagogique mais dont une bonne compréhension de leurs fonctionnement et de leurs limites permettent une utilisations efficaces des DSL comme Scilab, Matlab, pour ne citer que ces deux.

Chapter 3

Stockage matrices creuses

3.1 Généralités

Définition : matrice creuse

Une matrice creuse est une matrice ayant une matrice majorité d'éléments nuls.

- La fraction des éléments nuls dans une matrice creuse est appelée **vacuité (sparsity en anglais).**
- La fraction des éléments non nuls dans une matrice est appelée **densité.**

Ainsi, une matrice réelle A d'ordre N(N assez grand), ayant NZ éléments nuls est qualifiée de creuse si NZ est très grand.

Quelsques exemples de matrices creuses :

- matrices de toplitz tridiagonales;
- matrices diagonales;
- matrices de Hessenberg.

Formats de compression :

Pour des raisons de performances de calcul, l'exploitation de la structure creusedes matrices au moyen d'un format de Stockage des éléments non nuls peut s'avérer très utiles(ref).

plusieurs format de Stockage existent, parmi lesquels: CSR(Compressed Sparse Row format), COO (Coordinate format), CSC(Compressed Sparse Column format).

Soit une matrice A ayant nnz éléments non nuls.

stockage COO

Il permet de représenter une matrice creuse sous la forme de trois tableaux de tailles nnz:

- un tableau notons AA contenant tous les éléments non nuls de A dans n'importe quel ordre;

- un tableau notons JR d'entiers naturels contenant l'indices des lignes correspondantes;
- un tableau notons JC d'entiers naturels contenant l'indices des colonnes correspondantes.

exemple:

$$A = \begin{pmatrix} a_{11} & 0 & 0 & a_{13} \\ 0 & 0 & a_{23} & 0 \\ 0 & a_{32} & a_{33} & 0 \\ a_{41} & 0 & a_{43} & 0 \end{pmatrix}$$

$$AA = \begin{bmatrix} a_{11} & a_{23} & a_{13} & a_{41} & a_{32} & a_{43} & a_{33} \end{bmatrix}$$

$$JR = \begin{bmatrix} 1 & 2 & 1 & 4 & 3 & 4 & 3 \end{bmatrix}$$

$$JR = \begin{bmatrix} 1 & 3 & 3 & 1 & 2 & 3 & 3 \end{bmatrix}$$

stockage CSR

Ce format utilise trois tableaux :

- AA : de taille nnz contient les valeurs non nuls de la matrice A, lues de la gauches vers la droite , ligne par ligne du haut en bas;
- JA : tableau d'entiers naturels de même taille que AA contient les indices des colonnes des éléments dans AA. Par exemple $JA[i]$ correspond à la colonne de $AA[i]$ dans A;
- IA : tableau d'entiers naturels contient les positions des éléments de AA qui correspondent à une nouvelle ligne.

exemple:

$$A = \begin{pmatrix} 15 & 0 & 0 & 22 & 0 & -15 & 0 & 0 \\ 0 & 11 & 3 & 0 & 0 & 0 & 2 & 0 \\ 0 & 0 & 0 & -6 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 91 & 0 & 0 & 0 & 0 & 0 & 25 & 7 \\ 0 & 0 & 28 & 0 & 0 & 0 & 0 & -2 \end{pmatrix}$$

$$AA = \begin{bmatrix} 15 & 22 & -15 & 11 & 3 & 2 & -6 & 91 & 25 & 7 & 28 & -2 \end{bmatrix}$$

$$JA = \begin{bmatrix} 1 & 4 & 6 & 2 & 3 & 7 & 4 & 1 & 7 & 8 & 3 & 8 \end{bmatrix}$$

$$IA = \begin{bmatrix} 1 & 4 & 7 & 8 & 8 & 11 & 13 \end{bmatrix}$$

Quelques Remarques sur CSR

- $IA[i+1]$ - $IA[i]$: correspond au nombre d'éléments non nuls pour la ligne i. Par exemple $IA[2] = 4$, $IA[1] = 1$ et $IA[2]$ - $IA[1] = 3$. Et nous avons bien 3 éléments non nuls sur la ligne i =1;

- posant $k_0 = IA[i]$ et $k_1 = IA[i+1] - 1$ alors pour $k = k_0$ à $k = k_1$, $JA[k]$ correspond à l'indice de colonne des éléments non nuls de la lignes i.

  exemple: $k_0 = IA[2] = 4$, $k_1 = IA[3] = 7 - 1 = 6$.

  - $JA[4] = 2$ indice de colonne de 11
  - $JA[5] = 3$ indice de colonne de 3
  - $JA[6] = 7$ indice de colonne de 2

- posant $k_0 = IA[i]$ et $k_1 = IA[i+1] - 1$ alors pour $k = k_0$ à $k = k_1$, $AA[k]$ correspond aux valeurs non nuls de la ligne i.

  exemple: $k_0 = IA[2] = 4$, $k_1 = IA[3] = 7 - 1 = 6$.

  - $AA[4] = 1$
  - $AA[5] = 3$
  - $AA[6] = 2$

Si on a une matrice de taille n ayant nnz éléments non nuls, nous proposons l'algorithme suivant pour reconstituer la matrice initiale à partir de son stockage au format CSR.

stockage CSC

Il s'agit d'une variante du CSR. Ici, AA contient les valeurs non-nuls de A lues du haut vers le bas, colonne par colonne, de la gauche vers la droite. JA contient les indices des lignes des éléments dans AA. Par exemple $JA[i]$ correspond à la ligne de $AA[i]$. IA contient la position des éléments de AA correspondants à une nouvelle colonne.

3.2 Produit Matrice Vecteur Creux

Nous allons nous focaliser sur le stockage CSR.

soit

$$A = \begin{pmatrix} a_{11} & 0 & 0 & a_{14} \\ 0 & a_{22} & a_{23} & 0 \\ 0 & 0 & 0 & a_{34} \\ a_{41} & 0 & a_{42} & 0 \end{pmatrix}$$

et

$$x = \begin{pmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \end{pmatrix}$$

On a

$$v = Ax = \begin{pmatrix} a_{11} * x_1 + a_{14} * x_4 \\ a_{22} * x_2 + a_{23} * x_3 \\ a_{34} * x_4 \\ a_{41} * x_1 + a_{42} * x_4 \end{pmatrix}$$

Format CSR de la matrice A

$$AA = \begin{bmatrix} a_{11} & a_{14} & a_{22} & a_{23} & a_{34} & a_{41} & a_{42} \end{bmatrix}$$

$$JA = \begin{bmatrix} 1 & 4 & 2 & 3 & 4 & 1 & 2 \end{bmatrix}$$

$$IA = \begin{bmatrix} 1 & 3 & 5 & 6 & 8 \end{bmatrix}$$

Etudes préliminaires et démarche de mise en place de l'algorithme

- Pour $i = 1$ à $taille(IA) - 1 = (5 - 1) = 4$
  $k_0 = IA[1] = 1$ et $k_1 = IA[1 + 1] - 1 = 3 - 1$
  - $k = k_0 = 1 < k_1$
    $AA[k] = a_{11}$
    $j = JA[k] = 1$
    $x[j] = x[1] = x_1$
    $v[1] = v[1] + a_{11} * x_1$
  - $k = k_0 + 1 = 2 = k_1$
    $AA[k] = a_{14}$
    $j = JA[k] = 4$
    $x[j] = x[4] = x_4$
    $v[1] = v[1] + a_{14} * x_4$

- Pour $i = 2 < 5 - 1$
  $k_0 = IA[2] = 3$ et $k_1 = IA[2 + 1] - 1 = 5 - 1$
  - $k = k_0 = 3 < k_1$
    $AA[k] = a_{22}$
    $j = JA[k] = 4$
    $x[j] = x[4] = x_4$
    $v[2] = v[2] + a_{22} * x_4$
  - $k = k_0 + 1 = k_1$
    $AA[k] = a_{23}$
    $j = JA[k] = 3$
    $x[j] = x[3] = x_3$
    $v[2] = v[2] + a_{23} * x_3$

- Pour $i = 3 < 5 - 1$
  $k_0 = IA[3] = 5$ et $k_1 = IA[3 + 1] - 1 = 6 - 1$

- $k = k_0 = 5 = k_1$
  $AA[k] = a_{34}$
  $j = JA[k] = 4$
  $x[j] = x[4] = x_4$
  $v[3] = v[3] + a_{34} * x_4$

- Pour $i = 4 = 5 - 1$
  $k_0 = IA[4] = 6$ et $k_1 = IA[4 + 1] - 1 = 8 - 1$
  - $k = k_0 = 6 < k_1$
    $AA[k] = a_{41}$
    $j = JA[k] = 1$
    $x[j] = x[1] = x_1$
    $v[4] = v[4] + a_{41} * x_1$

  - $k = k_0 + 1 = 7 = k_1$
    $AA[k] = a_{42}$
    $j = JA[k] = 2$
    $x[j] = x[2] = x_2$
    $v[4] = v[4] + a_{42} * x_2$

Algorithme

Entré : AA, JA, IA, x
nnz nombre d'éléments non nuls dans A.
AA vecteur de taille nnz contenant les valeurs non nuls de A, lues de gauche vers la droite, ligne par ligne , de haut en bas.
JA vecteur de taille nnz contenant les indices des colonnes des éléments correspondants dans AA.
IA vecteur contenant les positions des éléments de AA correspondants à une nouvelle ligne dans A.
$v = Ax$

Corps de l'algorithme : $n =$ taille de AI - 1;
$v =$ zeros(n,1) ;
for i = 1 : n

- $k_0 = IA[i]$, $k_1 = IA[i + 1] - 1$
- for $k = k_0 : k_1$
  $v[i] = v[i] + AA[k] * x[JA[k]]$;
  end

end

Sortie : $v = Ax$

Complexité arithmétique

Pour i fixé on a:

$n_i = 2(k_1 - k_0 + 1)$ opérations élémentaires. $(k_1 - k_0 + 1)$ étant le nombre d'éléments non nuls de la ligne i dans A.

Ainsi, en tout on a :

$$\sum_{i=1}^{n} 2n_i$$

opérations élémentaires.

Mais alors en remarquant que

$$densite = \frac{\sum_{i=1}^{n} n_i}{n^2}$$

. Donc

$$\sum_{i=1}^{n} 2n_i = 2 * densite * n^2$$

.

Plus la matrice est dense , plus le nombre d'opérations est élévé. inversement moins la matrice est dense mmoins on a d'opérations.

Par ailleurs, comme le nombre de valeurs non-nuls par ligne varie suivant l'application (i.e n'est pas le même d'une matrice creuse à une autre), alors considérons le cas extrême d'une matrice triangulaire de taille N. Dans ce cas n_i= 1,2,3,..., N.

Et alors,

$$\sum_{j=i}^{N} 2n_i = \frac{N^2 + N}{2}$$

En comparaison au produit matrice-vecteur classique qui nécessite $2N^2 - N$ opérations le nombre d'opéartions est pratiquement divisé par 4.

Test et validation de l'algorithme

Pour les besoins du test nous allons considéré des matrices denses de tailles différentes , de densité différentes. Le vecteur considéré sera le vecteur de taille adéquat pour que le produit matrice-vecteur soit défini, ayant tous ses coefficients égaux à 1.

Ensuite on applique l'algorithme sur ces matrices et comparer le résultat à celui obtenu si nous prenons le structure matrice dense. On validera notre algorithme si nous obtenons la même chose.