Méthode du pivot

Structure de données

Codage des matrices

une matrice à *n* ligne et *p* colonnes est codée sous forme d'une liste de *n* termes, chaque terme étant une liste de *p* réels.

Autrement dit, on décrit la matrice ligne par ligne.

Codage des vecteurs

Un vecteur est codé par une liste de *n* réels.

⚠ **Passage des paramètres**

Considérons une fonction dont un des paramètres est une matrice M.

Le passage du paramètre se fait par adresse et non par valeur, c.a.d que c'est l'adresse de l'emplacement mémoire où la matrice est stockée, qui est transmis à la fonction.

conséquence : des instructions du type M[i] = ... , M[i] += ... , M.append(...) , M.sort() , M.reverse() etc ... modifient la matrice M au cours de l'exécution de la fonction. Si on désire conserver intacte la liste M, il faut en faire une vraie copie avant toute instruction.

Par contre des instructions du type M = M + [...] créent une copie locale du paramètre M qui n'est du coup pas affecté.

Accès-modification des coefficients

ma_var = M[i][j] # on stocke l'élément d'indice (i, j) dans la variable ma_var

ma_matrice[i][j] = nouv_valeur # on modifie la valeur du coefficient d'indice (i, j) de la matrice ma_matrice

Créer des matrices

1) Fonction creer_matrice(n , p , val) qui retourne une matrice de taille (n,p) dont toutes les composantes sont égales au flottant val.

```
1  def creer_matrice(n, p, val) :
2  |       "la fonction retourne une matrice de taille (n, p) avec toutes les composantes égales à val"
3  |       M = [] # initialisation de la matrice
4  |       for i in range(n) :
5  |       |       M.append([val]*p)  # i-ième ligne
6  |       return m
```

⚠ La version suivante ne convient pas, pourquoi ?

```
1  def creer_matrice(n, p, val) :
2  |       M, L = [] ,  [val]*p   # initialisation de la matrice
3  |       for i in range(n) :
4  |       |       M.append(L)
5  |       return M
```

Les deux versions suivantes conviennent en revanche

```
1  def creer_matrice(n, p, val) :
2  |       return [[val for j in range(p)] for i in range(n)]
```

```
1  def creer_matrice(n, p, val) :
2  |       return [[val]*p for i in range(n)]
```

2) Fonction identite(n) qui retourne la matrice identité de taille *n*.

```
1  def matrice_identite(n) :
2  |       M = creer_matrice(n, n, 0)
3  |       for i in range(n) :
4  |       |       M[i][i] = 1
5  |       return M
```

3) Fonction transposee(M) qui retourne la matrice transposée de M.

On donne la fonction en pseudo-code. En donner une version python

```
1  FONCTION transposee(M : matrice) résultat : matrice
2  |       n ← longueur(M)  # nombre de lignes de la matrice
3  |       p ← longueur(M[0])  # nombre de colonnes de la matrice
4  |       MT  ← []  # initialisation d'une matrice vide
5  |       POUR i DE 0 A p − 1 FAIRE
6  |       |       L = []  # initialisation de la ligne i de la transposée
7  |       |       POUR j DE 0 A n − 1 FAIRE
8  |       |       |       AJOUTER(L, M[j][i])  # = ajouter M[j][i] dans la liste L
9  |       |       FIN POUR
10 |       |       AJOUTER(MT, L)
11 |       FIN POUR
12 |       RETOURNER MT
13 FIN FONCTION
```

```
1  def  transposee(M) :
2  |       .....
3  |       .....
4  |       .....
5  |       .....
6  |       .....
7  |       .....
8  |       .....
```

Remarque

La matrice M passée en paramètre n'est pas modifiée dans la fonction puisqu'on crée une nouvelle matrice à l'aide de la double boucle.

4) Faire une copie d'une matrice (à compléter et commenter)

```
def copie_matrice(M) :
    n, p = .....  # dimensions de la matrice
    M_copie = .....  # initialisation de la copie
    for i in range(n) :
        .....
        .....
    .....
```

5) Fonction arrondir_matrice(M, prec) qui retourne une copie de la matrice M où tous les coefficients flottants sont arrondis à la précision 10^{-prec} (commande round(x, p), p nombre de décimales)

```
def arrondir_matrice(M, prec) :
    n, p = .....  # dimensions de la matrice
    M1 = [[round(M[i][j], prec) for j in range(p)] for i in range(n)]
    return M1
```

Programmer les opérations matricielles

6) Fonction addition(M1, M2) qui retourne la somme des deux matrices de même taille M1 et M2.

```
def addition(M1, M2) :
    n1, p1 = .....  # dimensions de la matrice m1
    n2, p2 = .....  # dimensions de la matrice m2
    if .....  # vérification que M1 et M2 ont mêmes dimensions
        M = []
        for i in range(n1):
            M.append([M1[i][j] + M2[i][j] for j in range(p1)])
        return M
    else:
        return None
```

⚠ M1[i] + M2[i] ne fait pas l'addition des *p*-uplets composante par composante, mais crée une nouvelle liste en rajoutant les termes de M2 à la suite des termes de M1.

7) Fonction mult_scalaire_vecteur(v, k) qui effectue la multiplication du vecteur v par le réel k.

Fonction mult_scalaire_matrice(M, k) qui effectue la multiplication de la matrice M par le réel k.

```
1  def mult_scalaire_vecteur(v, k) :
2  |       n = len(v)  #  taille du vecteur
3  |       .....
4  |       .....
5  |       .....
```

```
1  def mult_scalaire_matrice(M, k) :
2  |       n, p = .....
3  |       M1 = []
4  |       for i in range(n) :
5  |       |       L = mult_scalaire_vecteur(M[i], k) .....
6  |       |       M1.append(L) .....
7  |       return M1 .....
```

8) Fonction produit(M1, M2) qui retourne le produit des deux matrices M1 × M2 dont les tailles sont compatibles.

On applique la formule $(M_1M_2)_{i,j} = \sum_{k=1}^{p} (M_1)_{i,k} (M2)_{k,j}$

Compléter le pseudo-code puis donner une version python

```
1  FONCTION produit(M1, M2 : matrices) résultat : matrice
2  |       n1 ← longueur(M1)  # nombre de lignes de la matrice M1
3  |       SI nombre de colonnes de M1 != nombre de lignes de M2 ALORS RETOURNER NIL
4  |       M ← [] # initialisation de la matrice produit
5  |       POUR i DE 0 A n1 − 1 FAIRE # n1 − 1 inclus
6  |       |       L = []  # initialisation de la i-ième ligne
7  |       |       POUR j DE 0 A ..... FAIRE
8  |       |       |       ..... # somme initialisée à 0
9  |       |       |       .....
10 |       |       |       |       .....
11 |       |       |       .....
12 |       |       |       AJOUTER( .....
13 |       |       FIN POUR
14 |       |       AJOUTER( .....
15 |       FIN POUR
16 |       RETOURNER M
```

```
1  def produit(M1, M2) :
2  |       .....
3  |       .....
4  |       .....
5  |       .....
6  |       .....
7  |       .....
8  |       .....
9  |       .....
10 |       .....
11 |       .....
12 |       .....
13 |       .....
14 |       .....
```

Programmer les opérations élémentaires sur les lignes ou les colonnes

⚠ On choisit de faire les transformations sur place, c'est à dire que la matrice qui est passée en paramètre est elle-même modifiée.

9) Fonction echanger_lignes(M, i1, i2) qui retourne la matrice M où les lignes d'indices i1 et i2 ont été échangées.

```
1  def echanger_lignes(M, i1, i2) :
2  |       p = len(M[0])
3  |       for i in range(p) :
4  |       |       M[i1] , M[i2]  =  M[i2] , M[i1]  # affectations simultanées
5  |       # pas de return car la matrice M est passée en paramètre via son adresse en mémoire
6  |       # et ce sont les valeurs stockées dans cette zone mémoire qui ont été modifiées.
```

10) Fonction dilatation(M, i, k) qui retourne la matrice M où la ligne d'indice i a été multipliée par le réel k.

```
1  def dilatation(M, i, k) :
2  |       .....
3  |       .....
4  |       .....
5  |       .....
```

⚠ M[i] = k*M[i] ne convient pas car cela répète k fois la liste M[i] si k est entier strictement positif, retourne la liste vide si k entier négatif et retourne une erreur si k n'est pas entier. Cela marcherait en revanche avec des tableaux numpy avec lesquels on peut faire simplement des combinaisons linéaires.

11) Fonction transvection(M, i1, i2, k) qui retourne la matrice M où on a rajouté à la ligne d'indice i1, la ligne d'indice i2 multipliée par le réel k : Li1 ← Li1 + k*Li2

```
1  def transvection(M, i1, i2, k) :
2  |       .....
3  |       .....
4  |       .....
5  |       .....
```

⚠ M[i1] + k*M[i2] ne marche pas car cela concatène les listes au lieu de faire une addition composante par composante.

12) Fonction rech_pivot(M, j, i) qui retourne l'indice k ≥ i tel que | M[k][j] | soit maximum.

Quand on implémente la méthode du pivot, à la i-ième étape, on recherche un pivot sur la ligne L_i et dans une certaine colonne j qui dépend de l'historique de ce qui a été réalisé auparavant. Le pivot doit être non nul et on peut-être amené à permuter la ligne L_i avec une ligne L_k (k > i) qui contient un meilleurs pivot (toujours dans la colonne j). Le meilleurs pivot sera le plus grand en valeur absolue. C'est l'objet de cette fonction de rechercher ce meilleurs pivot à partir d'une position (i, j) donnée.

Analyser, commenter la fonction ci-dessous :

```
1  def rech_pivot(M, i, j) :
2  |       n, i0 = len(M), i
3  |       for k in range(i + 1 , n):
4  |       |       if abs(M[k][j] > abs(M[i0][j]):
5  |       |       |       i0 = i
6  |   return i0
```

Programmer la méthode du pivot

Cela recouvre :

- La résolution d'un système linéaire de matrice carrée inversible
- La recherche de la matrice échelonnée réduite associée à une matrice de taille quelconque
- Le calcul de l'inverse d'une matrice carrée.

13) Fonction resol_triang_sup(A, Y) qui prend en argument une matrice triangulaire inversible A et une matrice unicolonne Y et qui résout le système AX = Y.

Compléter, commenter le programme ci-dessous (la solution X devra être présentée sous forme de matrice unicolonne).

```
1  def resol_triang_sup(A, Y) :
2  |      n = len(A) # .....
3  |      X = [0.]*n  #  .....
4  |      X[n − 1] = Y[n − 1][0]/A[n − 1][n − 1]  # .....
5  |      # "remontée" : on calcule les inconnues à partir de X[n − 1]
6  |      for i in range(n − 2, − 1, − 1):  # indices de.....
7  |      |      s = 0  # initialisation du calcul de ∑ A_{i,j} X_j , j > i
8  |      |      for j in range( .....
9  |      |      |      s = s +  .....
10 |      |      X[i] =    .....
11 |      return ([[x] for x in X])  # on convertit X en matrice unicolonne
```

14) Fonction ech_red(A) qui prend en argument une matrice A de taille quelconque et qui retourne la matrice échelonnée réduite par ligne, équivalente par lignes à la matrice A ainsi que le rang de A. Commenter et compléter.

```
1  def ech_red(A) :
2  |      n, p = len(A), len(A[0])  # taille de la matrice m
3  |      A1 = copie_matrice(A)  # copie sur laquelle on effectue les calculs en place, A est préservée
4  |      i0, j0 = 0, 0  #on recherche un pivot à partir de la position (i0, j0)
5  |      while .....
6  |      |      k = rech_pivot(A1, i0, j0)  .....
7  |      |      if  A1[k][j0] != 0:  # pivot non nul
8  |      |      |      echanger_lignes(.....
9  |      |      |      for i in range(n):
10 |      |      |      |      if  i != i0:  .....
11 |      |      |      |      |      transvection(A1,
12 |      |      |      coeff = 1/A1[i0][j0]
13 |      |      |      dilatation(A1, i0, coeff )  .....
14 |      |      |      i0 = .....
15 |      |      j0 = .....
16 |   return(A1, i0)  # .....
```

Remarques

- Lors de l'appel des fonctions echanger_lignes et transvections (lignes 11 et 13), la matrice A1 est modifiée sur place. L'appel se fait donc sans affectation.
- On ne tient pas compte des problèmes numériques éventuels dans le cas où des pivots seraient très proches de 0. Cependant, un flottant n'est jamais rigoureusement égal à 0. Plutôt que de tester pivot = = 0 (ligne 7), il faut mieux utiliser un test du genre abs(pivot) < $10^{-\text{précision}}$, la précision pouvant alors être passée en paramètre par l'utilisateur.
- On peut appeler la fonction arrondir_matrice(A1, prec) avant de retourner A1 (ligne 16) pour éviter les affichages peu lisibles en raison d'un grand nombre de décimales.

15) Fonction resolution(A, Y, precision) où A est une matrice de taille (n, p), Y une matrice avec n lignes et q colonnes, précision un entier positif permettant de tester la nullité des pivots. La fonction retourne la matrice augmentée réduite par ligne associée au système AX = Y où X est une matrice de *p* lignes et q colonnes.

```
1  def resolution(A, Y, precision) :
2  |     n, p = len(A), len(A[0])
3  |     A1 = copie_matrice(A) # les opérations élémentaires vont se faire sur A1 et Y1
4  |     Y1 = copie_matrice(Y)
5  |     i0, j0 = 0, 0  #on recherche un pivot à partir de la position (i0, j0)
6  |     while .....................................................
7  |     |     k = rech_pivot(A1, i0, j0)
8  |     |     if  abs(A1[k][j0]) > 10**(– precision) :
9  |     |     # on considère le pivot comme étant non nul sinon
10 |     |     |     echanger_lignes(.....................................
11 |     |     |     echanger_lignes(.....................................
12 |     |     |     indices = list(range(0, i0)) + list(range(i0+1, n))  .....................................
13 |     |     |     for i in indices:  # on modifie les autres lignes
14 |     |     |     |     coeff  = ..........................................................
15 |     |     |     |     transvection(A1, i, i0, coeff)
16 |     |     |     |     transvection(.....................................
17 |     |     |     coeff  = 1/A1[i0][j0]
18 |     |     |     dilatation(.....................................
19 |     |     |     dilatation(.....................................
20 |     |     |     i0 = i0 + 1
21 |     |     j0 = j0 + 1
22 |     return(arrondi_matrice(A1, precision), arrondi_matrice(Y1, precision))
```

Sur Y1, on lit la compatibilité du système : les lignes de Y1 d'indices i0 + 1 à n doivent être nulles aux erreurs d'arrondis près.

16) Fonction inverse_matrice(A) qui prend en argument une matrice carrée A et qui retourne None si la matrice n'est pas inversible et qui retourne la matrice inverse sinon.

Utiliser la fonction resolution(A, Y, precision) avec Y qui est la matrice identité.

```
1   def inverse_matrice(A) :
2   |       n, p = len(A), len(A[0])
3   |       if n != p: return None
4   |       Y = matrice_identite(n)
5   |       echelonnee, inverse = .....
6   |       inversible = True
7   |       for i in range(n):
8   |       |       if abs(echelonnee[i][i]) < 1E-5:  # si un pivot est < 10^-5, on considère A non inversible
9   |       |       |       inversible = False
10  |       if inversible:
11  |       |       return inverse
12  |       else:
13  |       |       return None
```

FIABILITE DES RESULTATS

Les calculs sont ici tous numériques. On veut résoudre $AX = B$. On a déjà parlé du problème de la comparaison à 0. Cela peut engendrer des résultats grossièrement faux.

Par exemple, pour le système

$$\begin{pmatrix} 0.1 & 0.15 \\ 2. & 3. \end{pmatrix}\begin{pmatrix} x \\ y \end{pmatrix} = \begin{pmatrix} 3 \\ 5 \end{pmatrix}$$

la fonction « résolution » tout comme la commande de numpy np.linalg.solve, donnent comme couple solution

$$x = 1.48618788e+17 \quad , \quad y = -9.90791918e+16$$

alors que la matrice n'est pas inversible et que le système n'est pas compatible.

cela vient de

$$3. - (0.15/0.1)*2. = 4.440892098500626e-16$$

alors que cela devrait être nul.

La fonction « resolution » possède un paramètre qui permet d'éviter les pivots trop proches de 0.

Lorsqu'il est fixé à 20, on obtient la solution mentionnée plus haut. Mais lorsqu'il est fixé à 5 (on refuse les pivots < 10^{-5}), on obtient comme matrice augmentée réduite :

$$\left(\begin{array}{cc|c} 1. & 1.5 & 2.5 \\ 0. & -0.0 & 2.75 \end{array}\right)$$

et là, on voit immédiatement que le système est incompatible.

La fonction « np.linalg.solve » ne possède pas un tel paramètre. On peut à la fin effectuer le produit AX où X est la solution fournie par linalg.solve et comparer le résultat avec le résultat attendu B. Dans le cas présent, le produit donne [[2.], [0.]] , ce qui est loin du second membre [[3.] , [5.]] attendu.

Ce genre de phénomène ne peut pas survenir dans un système de calcul formel où la comparaison à 0 est exacte.

Un autre problème concerne l'amplification des erreurs :

Si on note δX l'écart sur la solution lorsqu'on perturbe le second membre de δB, alors il existe une constante C telle que

$$\|\delta X\| \leq C \frac{\|X\|}{\|B\|} \|\delta B\|$$

et il existe des vecteurs B et des perturbations δB pour lesquels, il y a égalité. On dispose d'une inégalité similaire lorsque ce sont les coefficients de la matrice A qui sont perturbés.

Le facteur C $\frac{\|X\|}{\|B\|}$ peut-être important.

Exemple : $A = \begin{pmatrix} 10 & 7 & 8 & 7 \\ 7 & 5 & 6 & 5 \\ 8 & 6 & 10 & 9 \\ 7 & 5 & 9 & 10 \end{pmatrix}$, $B = \begin{pmatrix} 32 \\ 23 \\ 33 \\ 31 \end{pmatrix}$; **AX = B a pour solution** $\begin{pmatrix} 1 \\ 1 \\ 1 \\ 1 \end{pmatrix}$

Pour $A' = \begin{pmatrix} 10 & 7 & 8.1 & 7.2 \\ 7.08 & 5.04 & 6 & 5 \\ 8 & 5.98 & 9.89 & 9 \\ 6.99 & 4.99 & 9 & 9.98 \end{pmatrix}$, **AX = B a pour solution** $\begin{pmatrix} -6.95\ldots \\ 14.00\ldots \\ -2.19\ldots \\ 2.95\ldots \end{pmatrix}$

Pour $B' = \begin{pmatrix} 32,1 \\ 22,9 \\ 33,1 \\ 30,9 \end{pmatrix}$, **AX = B' a pour solution** $\begin{pmatrix} 9.2 \\ -12.6 \\ 4.5 \\ -1.1 \end{pmatrix}$

COMPLEXITE

- **Cas du produit matriciel**

  n^2 coefficients à calculer, chacun demandant une somme de n produits flottants.

  Donc $\boxed{\text{complexité en } O(n^3)}$ **.**

  Il existe des méthodes permettant de faire légèrement mieux, basées sur la technique de "diviser pour régner".

- **Résolution d'un système triangulaire**

  $\boxed{\text{Complexité en } O(n^2) \text{ opérations}}$

- **Cas de la méthode du pivot pour une matrice carrée de taille n avec réduction triangulaire**

  Recherche du pivot n°i : n – i comparaisons ; permutation éventuelle de deux lignes : O(n) affectations

  Transvections : on annule les coefficients sous et au-dessus du pivot, chaque ligne à traiter comporte n – 1 coefficients à modifier, chacun au prix d'une division flottante. En tout $(n-1)^2$ multiplications ou divisions.

  En tout $\sum_{i=1}^{n-1} (n-1)^2 = O(n^3)$ **opérations flottantes**

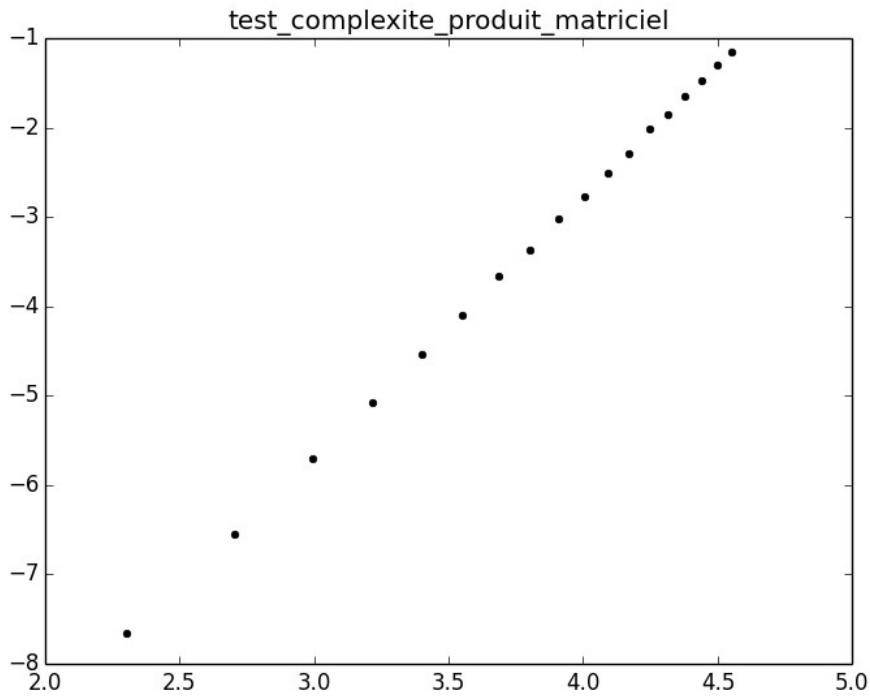
  $\boxed{\text{La méthode du pivot demande donc } O(n^3) \text{ multiplications}}$

  La résolution d'un système linéaire est donc aussi en $O(n^3)$ multiplications.

  Si on a des hypothèses supplémentaires sur la matrice du système, on peut faire mieux : c'est le cas avec des matrices symétriques, des matrices tridiagonales, des matrices creuses (avec beaucoup de zéros), ...

  Il existe aussi des méthodes itératives convergeant vers la solution du système.

- $\boxed{\text{Inversion d'une matrice carrée : complexité en } O(n^3) \text{ opérations}}$ **.**

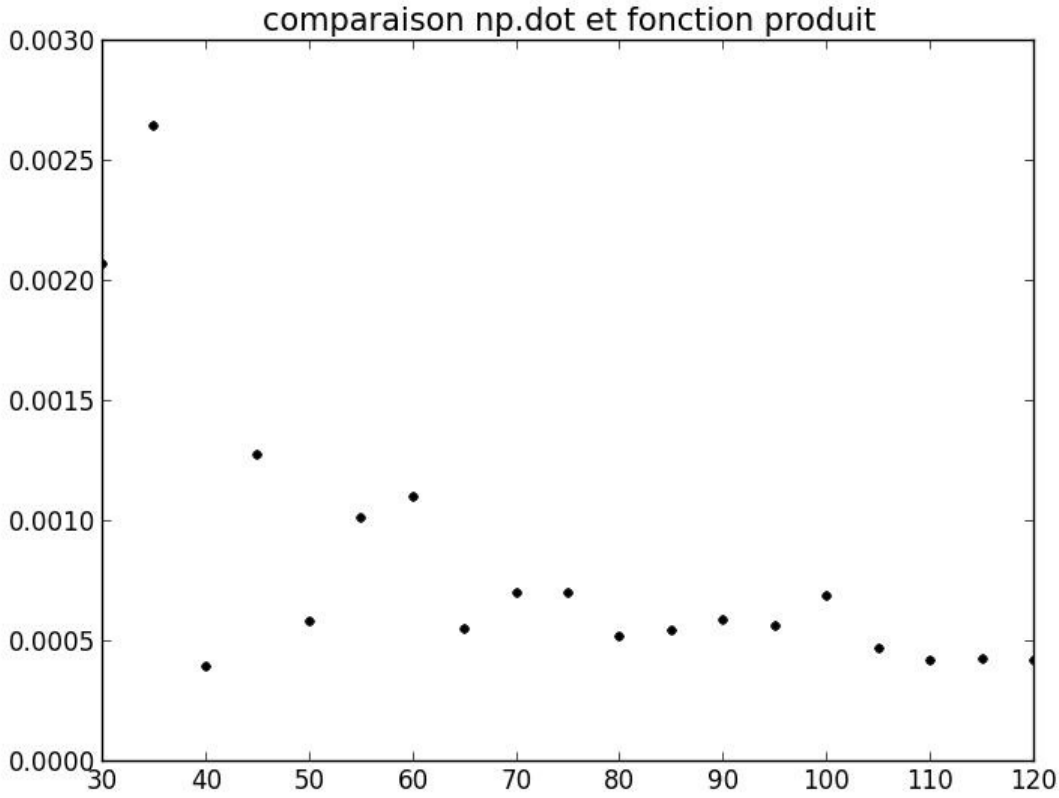


On multiplie deux matrices carrées aléatoires (composantes dans [0,1]) de taille $n \leq 1000$. On porte en abscisses log(n) et en ordonnées log(temps d'exécution) (On considère que la complexité en nombre d'opérations est proportionnelle au temps d'exécution). Si la complexité est en $O(n^p)$, alors on doit obtenir une droite de pente p.

COMPARAISON DES ALGORITHMES

On réalise trois fonctions test, l'un pour le produit, le second pour la résolution d'un système, le troisième pour le calcul de l'inverse.

Pour le produit, le test consiste à choisir deux matrices aléatoires de taille n au format array de numpy. On en fait une copie comme liste de listes.
Ensuite on mesure le temps nécessaire pour en faire le produit, d'abord avec la commande np.dot puis avec l'algorithme produit. Bien noter que c'est la même matrice qui est employée dans les deux calculs.
On retourne le quotient des temps d'exécution qui est plus signifiant et a priori indépendant de l'ordinateur sur lequel on exécute le test.
On appelle ensuite ce test pour un échantillon de valeurs de n.



(n en abscisses , quotient des temps d'exécution en ordonnées)

La version numpy est environ 1000 fois plus rapide !

On procède de même pour la résolution d'un système $mX = b$, m et b étant respectivement une matrice aléatoire et un vecteur colonne aléatoire. On compare cette fois « np.linalg.solve » et la fonction « resol_mat_inv ». Là encore, le score est sans appel.

Enfin, on compare sur le même schéma, le calcul de l'inverse d'une matrice carrée.