
Algèbre linéaire

1 Pivot de Gauss et variations

Télécharger les fonctions utiles sur mon site web (notamment le code du pivot ci-dessous).

L'algorithme du pivot de Gauss. On rappelle l'algorithme du pivot de Gauss :

Algorithme 1 : Pivot de Gauss

Entrées : Une matrice A de taille $n \times n$, un vecteur Y

Sortie : La solution du système $AX = Y$, si le pivot se déroule bien

pour chaque i allant de 0 à $n - 1$ **faire**

$j \leftarrow \text{indice_pivot}(A, i);$

si $j \neq i$ **alors**

$\text{echange_ligne}(A, i, j);$

$\text{echange_ligne}(Y, i, j)$

pour chaque j allant de $i + 1$ à $n - 1$ **faire**

$\mu \leftarrow -A[j][i]/A[i][i];$

$\text{transvection}(A, j, i, \mu);$

$\text{transvection}(Y, j, i, \mu)$

$X \leftarrow$ vecteur nul de taille n ;

pour chaque i allant de $n - 1$ à 0 **par pas de** -1 **faire**

$x_i \leftarrow \frac{1}{a_{i,i}}(y_i - \sum_{j=i+1}^{n-1} a_{i,j}x_j)$

Renvoyer X

La fonction `indice_pivot(A,i)` prend en entrée une matrice A et un indice i , et renvoie l'indice de la ligne du plus grand coefficient en valeur absolue parmi les éléments situés en dessous de $a_{i,i}$ (compris). En mathématiques on testerait si $a_{i,i}$ est nul avant de procéder à un échange de lignes éventuel, en informatique il ne faut pas : si $a_{i,i}$ est très petit mais non nul, on introduit d'énormes erreurs en le prenant comme pivot. On préfère prendre le plus grand en valeur absolue. La fonction `echange_lignes` parle d'elle même, et `transvection(A,j,i,mu)` réalise l'opération $L_j \leftarrow L_j + \mu \cdot L_i$ sur la matrice A . L'idée du pivot est de mettre la matrice sous forme triangulaire supérieure, puis de procéder à une phase de remontée pour calculer la solution de $AX = Y$.

On travaille avec une représentation sous forme de tableaux Numpy : si A est une matrice, $A[i]$ est la i -ème ligne, dont les éléments sont les $A[i][j]$ ou $A[i,j]$. Un vecteur est encodé comme un tableau Numpy de dimension 1.

Exercice 1. *Complexité du pivot.* Rappeler/retrouver la complexité du pivot de Gauss

Exercice 2. *Variation.* Que pensez-vous de la complexité du calcul du déterminant via une fonction récursive appliquant le développement par rapport à une ligne/colonne ? Modifier l'algorithme du pivot pour obtenir :

1. une fonction qui calcule le déterminant d'une matrice (rappel : un échange de lignes multiplie le déterminant d'une matrice par -1 , une transvection ne le modifie pas).
2. une fonction qui calcule l'inverse d'une matrice (rappel : effectuer des opérations sur les lignes d'une matrice revient à la multiplier à gauche par des matrices élémentaires. Si on se débrouille pour transformer A en l'identité, cela revient à construire une matrice P telle que $PA = I_n$. Appliquer les mêmes opérations sur une matrice initialement égale à I_n mène à $P = A^{-1}$).

Vérifier vos fonctions à l'aide des fonctions fournies et Numpy (voir page suivante). Vérifier que la complexité est la même que celle du pivot.

```

>>> M=tri_diag(4)
>>> determinant(M)
55.0
>>> alg.det(M)
55.0000000000000014
>>> np.array(inverse(M))
array([[ 0.38181818,  0.14545455,  0.05454545,  0.01818182],
       [ 0.14545455,  0.43636364,  0.16363636,  0.05454545],
       [ 0.05454545,  0.16363636,  0.43636364,  0.14545455],
       [ 0.01818182,  0.05454545,  0.14545455,  0.38181818]])
>>> alg.inv(M)
array([[ 0.38181818,  0.14545455,  0.05454545,  0.01818182],
       [ 0.14545455,  0.43636364,  0.16363636,  0.05454545],
       [ 0.05454545,  0.16363636,  0.43636364,  0.14545455],
       [ 0.01818182,  0.05454545,  0.14545455,  0.38181818]])

```

2 Résolutions particulières.

On donne ici quelques idées sur d'autres méthodes de résolution. L'idée est de proposer un algorithme de meilleure complexité que le pivot de Gauss (qui prend du temps pour une matrice de taille $10^4 \times 10^4$, par exemple!), quitte à sacrifier un peu l'exactitude de la solution. On travaille avec des tableaux Numpy dans la suite.

Exercice 3. Méthode de Jacobi. On se donne une matrice A à diagonale dominante, c'est à dire que $A = (a_{i,j})_{0 \leq i,j \leq n-1}$ vérifie $|a_{i,i}| > \sum_{j \neq i} |a_{i,j}|$ pour tout $i \in \llbracket 0, n-1 \rrbracket$ (une telle matrice est classiquement inversible). On décompose A sous la forme $D + H$, avec D la matrice diagonale constituée des coefficients diagonaux de A et H la matrice à coefficients diagonaux nuls égale à $A - D$. D étant inversible (c'est évident), le système $Ax = y$ s'écrit $Dx = y - Hx$ soit encore $x = D^{-1}(y - Hx)$. La méthode de Jacobi consiste à partir d'un certain $x_0 \in \mathbb{R}^n$ (absolument quelconque, par exemple 0) et itérer à partir de x_0 la fonction $f : z \mapsto D^{-1}y - D^{-1}Hz$. On pose $(x_n)_{n \in \mathbb{N}}$ la suite ainsi obtenue, dont on peut montrer qu'elle converge vers x , solution de $Ax = y$. Remarquez que :

- l'inverse de D n'est pas coûteux à calculer car D est diagonale ;
- le produit d'une matrice $n \times n$ et d'un vecteur a une complexité $O(n^2)$. Si le nombre d'étapes effectuées par la méthode de Jacobi est faible par rapport à n , on obtient une meilleure complexité que l'algorithme du pivot de Gauss (mais la solution n'est qu'approchée).

1. On pose $M = \max_{0 \leq i \leq n-1} \frac{1}{|a_{i,i}|} \sum_{j \neq i} |a_{i,j}|$. Comme A est à diagonale dominante, $M < 1$. On admet¹ que la fonction f est M -lipschitzienne, pour la norme infinie définie par $\|(y_i)_{0 \leq i < n-1}\|_\infty = \max |y_i|$, ce qui implique² que $\|x - x_k\|_\infty \leq \frac{M^k}{1-M} \|x_0 - x_1\|_\infty$. En déduire une estimation du nombre d'étapes à effectuer dans la méthode de Jacobi, si l'on vise une précision de ε sur chaque coefficient de x .
2. Écrire une fonction `couple(A)` renvoyant le couple (D^{-1}, H) de matrices associées à A sous forme de tableaux Numpy. On partira de deux tableaux `np.zeros((n,n))` qu'on remplira comme il se doit (la matrice D^{-1} se calcule facilement : ses coefficients sont les inverses des éléments digonaux de A).
3. Écrire une fonction donnant la norme infinie du vecteur passé en paramètre.
4. Implémenter effectivement une fonction `jacobi(A,Y,eps)` en Python (on commencera avec x_0 égal au vecteur nul, `np.zeros(n)`). La fonction `np.dot` donne le produit matriciel.
5. Quelle est sa complexité, en fonction de ε et de n ?

```

>>> A=tri_diag(4)
>>> Y=np.zeros(4) ; Y[0]=1
>>> jacobi(A,Y,10**-3) #17 est le nombre d'étapes effectuées
17
array([ 0.38181624,  0.14545285,  0.05454232,  0.01818077])
>>> alg.solve(A,Y)
array([ 0.38181818,  0.14545455,  0.05454545,  0.01818182])

```

Exercice 4. Calcul de la valeur propre de plus grand module. Dans cet exercice, on utilise la norme 2 (euclidienne), définie comme suit : $\|(x_i)_{0 \leq i < n}\|_2 = \sqrt{\sum_{i=0}^{n-1} x_i^2}$. On suppose que la matrice $A = (a_{i,j})_{0 \leq i,j \leq n-1}$ possède une unique

1. Ceci se vérifie facilement.
2. Cela aussi n'est pas trop dur à montrer.

valeur propre de module maximal, de multiplicité 1. On définit les suites suivantes, en partant d'un vecteur x_0 arbitraire (on pourra le choisir au hasard) et $y_0 = \frac{x_0}{\|x_0\|_2}$:

$$\begin{cases} x_k = Ay_{k-1} \\ y_k = \frac{x_k}{\|x_k\|_2} \\ \lambda_k = y_k^T A y_k \end{cases}$$

On peut montrer que pour « presque tout » x_0 initial, $(\lambda)_k$ converge vers la valeur propre de module maximal, et (y_k) vers un vecteur propre associé, de norme 1.

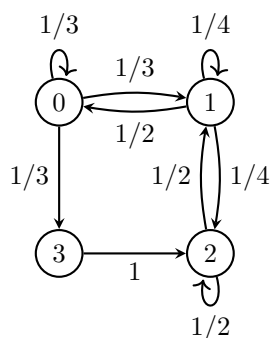
Programmer le calcul de cette suite sous la forme d'une fonction `vp(A,eps)`. On prendra comme vecteur initial x_0 un vecteur choisi aléatoirement, et comme condition d'arrêt $|\lambda_k - \lambda_{k-1}| < \varepsilon$, et on renverra la valeur λ_k et le vecteur y_k obtenus.

Comparer avec `alg.eig` qui renvoie valeurs et vecteurs propres (ces derniers sous la forme de la matrice des vecteurs propres).

```
>>> A=tri_diag(4)
>>> vp(A,10**-3)
(4.6170266970284484, array([-0.34993929,  0.59213796, -0.6130905 ,  0.38863243]))
>>> alg.eigvals(A) #la vp de module maximal est bien 4.61...
array([ 1.38196601,  2.38196601,  4.61803399,  3.61803399])
```

3 Ballades sur un graphe et théorème de Perron-Frobenius

Un graphe orienté est un couple (V, E) où les éléments de V sont appelés sommets et les éléments de $E \subseteq V^2$ sont appelés des arcs. On considère une marche aléatoire sur ce graphe : on part d'un sommet initial v_0 , et à chaque étape, si l'on se situe au sommet v_i on se dirige vers l'un des voisins v_j de v_i (l'un des v tels que $(v_i, v) \in E$) avec une certaine probabilité $p_{i,j}$ ne dépendant que de v_i et v_j (i.e indépendante du parcours effectué précédemment). Formellement, on numérote les sommets de 0 à $n-1$ et on représente le graphe par une matrice de taille $n \times n$ dont les éléments sont les $p_{i,j}$. Cette matrice est la matrice d'adjacence du graphe, et naturellement $\sum_{j=0}^{n-1} p_{i,j} = 1$ pour tout i : les lignes de la matrice sont toutes de somme 1. Une telle matrice est dite *stochastique*. L'exemple suivant montre un graphe et la matrice d'adjacence associée.



$$G = \begin{pmatrix} 1/3 & 1/3 & 0 & 1/3 \\ 1/2 & 1/4 & 1/4 & 0 \\ 0 & 1/2 & 1/2 & 0 \\ 0 & 0 & 1 & 0 \end{pmatrix}$$

Question 1. *Se diriger vers un nouveau sommet.* La fonction `random` du module du même nom fournit un réel aléatoire de l'intervalle $]0,1[$ (suivant la loi continue uniforme sur cet intervalle). Écrire une fonction `suivant(L)` prenant en entrée une liste $[p_0, p_1, \dots, p_{n-1}]$ de probabilités de somme totale 1, et renvoyant l'élément $i \in \llbracket 0, n-1 \rrbracket$ avec la probabilité p_i . *Indication :* il est très commode de calculer d'abord les sommes cumulées $[s_0, s_1, \dots, s_{n-1}]$, avec $s_i = p_0 + \dots + p_i$, de tirer x au hasard dans $[0, 1[$ via `random()` et de renvoyer le premier indice i tel que $s_i \geq x$.

Remarque : on pourrait calculer ces sommes cumulées une seule fois, et procéder par recherche dichotomique à chaque étape, pour gagner en complexité.

```
>>> [suivant([0.3, 0.2, 0.5]) for i in range(20)]
[2, 0, 0, 0, 1, 2, 2, 2, 2, 2, 0, 2, 1, 2, 1, 2, 2, 1, 2, 2]
```

Question 2. *Ballade aléatoire.* Dédurre de la fonction précédente une fonction `ballade(G,s,N)`, prenant en entrée — le graphe, donné par sa matrice d'adjacence ;

- un sommet de départ $s \in \llbracket 0, n-1 \rrbracket$, si le graphe a n sommets ;
- un entier $N \geq 0$.

et réalisant une marche aléatoire à N étapes sur le graphe, la fonction renvoyant le sommet atteint à la fin. À l'étape i , on utilisera la fonction précédente sur $G[i]$ pour savoir quel sommet visiter ensuite.

```
>>> [ballade(G,0,100) for i in range(20)]
[2, 0, 0, 1, 3, 0, 1, 1, 3, 1, 0, 0, 2, 1, 2, 1, 1, 2, 1, 2]
```

Question 3. Des statistiques. Écrire une fonction `stats(G,N,M)` prenant en entrée deux entiers N et M et un graphe G , et effectuant M marches aléatoires de longueur N partant du sommet 0. La fonction renvoie une liste de somme 1, l'élément en case i étant la proportion de marches aléatoires de longueur N effectuées depuis le sommet 0 et ayant terminé au sommet numéro i . On testera avec $N = 200$ et $M = 1000$, par exemple.

```
>>> stats(G,200,1000)
[0.258, 0.316, 0.358, 0.068]
```

Question 4. Théorème de Perron Frobenius. Comparer avec le vecteur propre dont la somme des coefficients vaut 1, associé à la valeur propre 1 de la matrice tG . On pourra utiliser à nouveau `alg.eig`, et `np.transpose` pour la transposée.

4 Orthonormalisation de Gram-Schmidt et décomposition QR

Soit $A \in \mathcal{GL}_n(\mathbb{R})$. La famille de vecteurs donnée par les colonnes de A étant une base de $\mathbb{R}^n$, on peut lui appliquer le procédé d'orthonormalisation de Gram-Schmidt pour obtenir une base orthonormée, dont les vecteurs forment une matrice $Q \in \mathcal{O}_n(\mathbb{R})$.

Il est facile de voir que les opérations d'orthonormalisation se résument en une matrice triangulaire supérieure avec des coefficients strictement positifs sur la diagonale S , telle que $AS = Q$. On a donc $A = QR$, avec $R = S^{-1}$, également triangulaire supérieure avec des coefficients diagonaux strictement positifs : c'est la décomposition QR . On peut de plus montrer que la décomposition $A = QR$ avec Q orthogonale, et R triangulaire supérieure à coefficients strictement positifs est unique.

Question 1. Écrire une fonction `ortho(A)` prenant en entrée une matrice inversible A et renvoyant la matrice Q orthogonale, obtenue par orthonormalisation de Schmidt des colonnes de A . On pourra utiliser `np.copy` pour faire une copie de A , on pourra écrire deux fonctions auxiliaires (courtes avec les opérations Numpy !) calculant le produit scalaire de deux vecteurs et la norme 2 d'un vecteur.

Question 2. Si Q est telle que $A = QR$, alors $R = {}^tQA$. En déduire une fonction `decompo_QR(A)` renvoyant la décomposition QR de A sous la forme d'un couple de matrices. Faire des tests !

Question 3. Quelle est la complexité du calcul de cette décomposition ?

5 Annexe : rappels Numpy

- `np.zeros(n)` : crée un vecteur (ligne) à n composantes nulles.
- `np.zeros((n,m))` : crée une matrice à n lignes et m colonnes remplie de zéros ; marche aussi avec des dimensions supplémentaires.
- `np.array(L)` : convertir une liste en tableau Numpy. Si L est une liste de listes, on obtient une matrice, etc...
- Les opérations usuelles se font terme à terme sur les tableaux Numpy. Les opérations scalaires également, par exemple si v est un tableau Numpy, $2*v$ est le tableau où toutes les entrées ont été multipliées par 2.
- `M.dot(N)`, `np.dot(M,N)` : produit matriciel de M par N (si dimensions compatibles). Remarque : si M est une matrice et N un vecteur ligne, le produit est le produit matrice-vecteur standard, le résultat est donné comme un vecteur ligne.
- `np.sum(M)` : somme des éléments de M .
- `M.transpose()`, `np.transpose(M)` : copie transposée de M .
- `alg.det(M)` : déterminant de M .

- `alg.inv(M)` : inverse de M .
- `alg.matrix_power(M,n)` : calcule M^n pour $n \in \mathbb{N}$.
- `alg.solve(M,Y)` : résolution de $MX = Y$.
- `alg.eigvals(M)` : valeurs propres de M (sous forme de tableau Numpy).
- `alg.eig(M)` : valeurs et vecteurs propres de M . Les vecteurs propres sont donnés sous la forme d'une matrice de passage. Avec $T, P = \text{alg.eig}(M)$, on a $M = PDP^{-1}$, avec D la matrice diagonale dont les coefficients diagonaux sont donnés par T , si M est diagonalisable. Attention : avec des flottants, les égalités ne sont vraies qu'à ε près...
- `np.random.random()` renvoie un réel aléatoire de $[0, 1[$. Avec un paramètre supplémentaire, on peut avoir un vecteur Numpy constitué d'entrées aléatoires :

```
>>> np.random.random(4)
array([ 0.3158855 ,  0.01930876,  0.57718712,  0.1082994 ])
```