Université de Rouen
M1 IBIOM
Année 2012-2013

Mathématiques Appliquées
Python et les math (n°2.)

1. Systèmes linéaires

D'après la documentation pour résoudre un système linéaire on peut
- `x=np.linalg.solve(A,b)` qui résout le système $Ax = b$ où A est une matrice carrée et b un vecteur ou une matrice
- `x=np.dot(np.linalg.inv(A),b)` qui calcule $A^{-1}b$. Cette commande n'est cependant pas recommandée. Comme on le verra elle calcule d'abord A^{-1} puis le vecteur $A^{-1}b$, ce qui fait plus de calculs que la «simple» résolution de $Ax = b$.

Vous êtes libre d'utiliser le type `mat` ou `array` pour gérer les matrices.

Exercice 1. Choisir des exemples de systèmes linéaires à résoudre (ceux de T.D. par exemple).

Exercice 2. [Comparaison `solve` et `inv`]

Les deux méthodes donnent un résultat équivalent, cependant le temps d'exécution n'est pas le même. Pour comparer il faut tout d'abord trouver la commande qui retourne le temps écoulée et prendre des exemples de grande dimension. (`import time` et la commande `time.time()`). Construire la matrice A de taille 300 et le vecteur b telle que

$$A = \begin{pmatrix} 2 & -1 & \cdots & 0 \\ -1 & \ddots & \ddots & \\ & \ddots & \ddots & -1 \\ 0 & & -1 & 2 \end{pmatrix} \quad \text{et} \quad b = \begin{pmatrix} 1 \\ \vdots \\ \vdots \\ 1 \end{pmatrix}.$$

Exécuter une boucle qui résout n fois le système linéaire par `solve` et une autre par `np.dot(np.linalg.inv(A),b)` et comparer les temps d'exécution. Si l'ordinateur est très rapide on peut modifier le n en 300, 600 ou plus encore ! Explications ?

2. Décomposition *LU* et Cholesky

Les choses sérieuses commencent ! La décomposition *LU* ne semble pas être incluse dans Numpy mais dans Scipy. Nous chargeront Numpy et Scipy via

```
>>> from numpy import *
>>> from scipy import linalg
```

On dispose alors des commandes `linalg.lu` et `linalg.cholesky`. Par rapport au cours la commande `linalg.lu` effectue la décomposition PLU où P est une matrice de permutation. Pour des questions de stabilité numérique et d'existence c'est préférable à la décomposition LU sans stratégie de pivot. Pour la résolution de système linéaire via la décomposition PLU nous disposons de deux routines dédiées : `linalg.lu_factor(A)` qui retourne PLU sous forme compacte (une matrice qui contient L et U ainsi qu'un vecteur qui représente les indices de ligne du choix des pivots) et `linalg.lu_solve((lu,p),b)` où `(lu,p)` est la décomposition de `lu_factor` et `b` le vecteur.

Pour la décomposition de Cholesky, nous avons de la même façon les deux routines dédiées à la résolution de système linéaire `linalg.cholesky_factor` et `linalg.cholesky_solve`

Exercice 3. Tester sur les matrices de la feuille de T.D. les routines décrites plus haut et résoudre un système linéaire via l'enchaînement `linalg.lu_factor, linalg.lu_solve((lu,p),b)`.

```
>>> A=mat('[1, 2, 3; 4, 5, 6; 6, 7, 9]')
>>> mad=linalg.lu_factor(A)
>>> b=mat('[1; 2; 3]')
>>> linalg.lu_solve(mad,b)
```

Exercice 4. [Faculatif] Générer la matrice A de taille 100 avec une diagonale remplie de 200 et 1 ailleurs et un vecteur b de taille 100 de votre choix. Tester la différence de rapidité entre `linalg.lu_solve((lu,p),b)` et la méthode `w=linalg.solve(u,b)` puis `linalg.solve(l,w)`. Pour cela il faudra effectuer plusieurs fois

(une boucle par exemple) la même opération. Bien sûr la décomposition LU (`linalg.lu_factor(A)` et `linalg.lu(A)` est effectuée en dehors de la boucle.

Exercice 5. [décomposition LU sans stratégie] Programmer une routine `madecomplu(A)` qui effectue la décomposition LU de la matrice `A` sans stratégie et retourne L et U. On adoptera l'algorithme suivant

```
1 : récupérer la taille de la matrice A : n cette taille
2 : générer la matrice identité de même taille que A : L cette matrice
3 : faire une copie de A, U=cette copie sur laquelle on effectuera l'élimination
4 : for i=1 to n-1 do
5 :    p=pivot {U(i,i) est pivot}
6 :    for k=i+1 to n do
7 :       L(k,i)= U(k,i)/p {on remplit L}
8 :       {On élimine}
9 :       ligne k de U remplacée par ligne k de U - U(k,i)/p ligne i de U (encore une petite boucle de i+1 à n)
10 :   end for
11 : end for
```

(a) Tester la procédure avec des exemples du cours ou de TD.

(b) Pour des matrices mal conditionnées et de grande taille (comme la matrice de l'exercice 2) étudier $A - LU$ où L et U sont données par votre procédure et $A - PLU$ où P, L et U sont données par `decomp.lu`.

3. Moindres carrés

Scipy nous fournit `linalg.lstsq`. Dans la suite utiliser `linalg.lstsq` et la méthode de la résolution de l'équation normale.

Exercice 6. Soit le tableau où σ désigne la force appliquée et ε la déformation

force	0	0.06	0.14	0.25	0.31	0.47	0.60	.70
déformation	0	0.08	0.14	0.20	0.23	0.25	0.28	0.29

Pour chaque recherche de moindres carrés on écrira tout d'abord le problème, la matrice, le vecteur sur une feuille.

(a) Trouver (avec Python) la droite de régression linéaire.

(b) Trouver le polynôme de degré 2 qui approche le nuage de points au sens des moindres carrés.

(c) Donner deux approximations de la déformation pour $\sigma = 0.9$.

(d) Tracer tout ce petit monde (`import matplotlib.pyplot as plt`).

3.1. Comparaisons des méthodes «équation normale», «QR» et `linalg.lstsqr`.

On utilise pour un problème de moindres carrés trois méthodes de résolution qui sont la résolution de l'équation normale $A^tAx = A^ty$ (`linalg.solve` tout simplement), l'utilisation de la décomposition QR de A (`linalg.qr` et `linalg.solve`) et le plus simple la routine Scipy `linalg.lstsqr`.

Soit la fonction

$$y(s) = x_1\frac{1}{s} + x_2\frac{1}{s^2} + x_3\frac{1}{s^3}$$

avec $x_1 = x_2 = x_3 = 1$! On considère plusieurs 10-uplets $(s_i, y(s_i))_{i=1\ldots10}$ et on utilise chaque méthode pour résoudre le problème des moindres carrés (dont la solution triviale est $(1,1,1)^t$). Les échantillons sont du type $s_i = s_0 + i$, $1 \le i \le 10$ avec différentes valeurs de s_0. Chaque méthode nous retourne une solution $\bar{x}$ qui est comparée à la solution exacte $x = (1,1,1)^t$ par $\|\Delta x\|_2 = \|\bar{x} - x\|_2$ avec la convention $\|\Delta x\|_{\text{ortho}}$ pour la méthode QR, $\|\Delta x\|_{\text{normale}}$ pour la (simple) résolution de l'équation normale et $\|\Delta x\|_{\text{scipy}}$.

Faire une routine Python pour chaque méthode et tester pour $s_0 \in \{10, 20, 50, 100, 250, 500\}$.

Vos conclusions ?