Université Lyon 1 Année 2019-2020
Licence de Mathématiques, 3e année
Analyse matricielle

TP 2

Prévu pour être fait en 3 heures

Installation de Python

Le plus simple est d'installer Anaconda. Vous aurez Python avec tous les paquets dont nous avons besoin, les notebook jupyter et Spyder. Le fichier d'installation fait entre 450Mo et 520Mo, il vous faudra une bonne connexion.

Une autre solution consiste à installer Miniconda, bien plus léger, mais qui n'installe que le strict minimum. Il faudra installer le reste « à la main ».

Vous pouvez aussi accéder à un serveur jupyter avec des notebook python en vous connectant avec vos identifiants habituels à cette adresse : `https://jupyter.mecanique.univ-lyon1.fr` Attention, ce serveur n'est toutefois pas capable de supporter plus d'une cinquantaine de personnes en même temps.

Anaconda

Windows : Télécharger Anaconda ici. Il devrait suffire ensuite de double-cliquer sur le fichier téléchargé et de suivre les indications pour installer Anaconda.

MacOS : Comme pour Windows, télécharger Anaconda ici. Il ne reste plus qu'a double-cliquer sur le fichier téléchargé et de suivre les instructions.

Linux : Il faut télécharger le script d'installation ici.

1. Ouvrir un terminal (dépend de votre distribution Linux). Vous pouvez chercher une application qui s'appelle *Terminal* ou *Shell*.
2. Se déplacer à l'endroit où le fichier a été enregistré :
   `cd chemin/vers/le/fichier/téléchargé`
3. Donner les droits d'exécution au fichier :
   `chmod u+x Anaconda3-2020.02-Linux-x86_64.sh`
4. Exécuter le fichier :
   `./Anaconda3-2020.02-Linux-x86_64.sh`
5. Accepter la licence (appuyer sur la touche ESPACE jusqu'à arriver à la fin et entrer *yes*).
6. Laisser le chemin d'installation par défaut en appuyant sur la touche ENTRÉE (vous pouvez le changer si vous voulez. Si vous ne savez pas, laissez celui par défaut).

7. Répondre *yes* lorsque l'on vous demande si vous voulez initialiser Anaconda 3 en lançant *conda init*.

Vous pouvez maintenant ouvrir un nouveau terminal et lancer les notebook ou Spyder :
`jupyter notebook`
`spyder`

Miniconda

L'installation devrait être similaire à celle d'Anaconda. La page avec les téléchargements se trouve ici. Prendre les versions Miniconda 3, 64 bits. Pour MacOS, la version pkg est la plus simple, la version bash se comporte comme pour Linux.

Une fois Miniconda installé, il faudra aussi installer les paquets dont nous avons besoin, à savoir :

— numpy
— scipy
— matplotlib

Si vous voulez Spyder ou les notebook jupyter, vous pouvez les installer avec conda, ce sont des paquets comme les autres.

Pour installer un paquet sous Linux (ou MacOS, si pas d'application graphique) :

1. Ouvrir un terminal
2. Entrer la commande conda suivante :
   `conda install nom_du_paquet`

Par exemple, pour installer les trois paquets dont nous avons besoin :
`conda install numpy scipy matplotlib`

Exercices

Au début de la session, charger les modules numpy, numpy.linalg et matplotlib avec les commandes :

```
import numpy as np
import numpy.linalg as npl
import matplotlib.pyplot as plt
```

Exercice 1. (*Suites récurrentes*)

Soit $n \in \mathbb{N}^*$, $B \in \mathcal{M}_n(\mathbb{R})$ et $b \in \mathbb{R}^n$. On définit une suite récurrente associée à la matrice d'itération B et au vecteur b par

$$\begin{cases} x_0, x_1 \in \mathbb{R}^n \text{ donnés} \\ x_{k+1} = Bx_k + Bx_{k-1} + b \text{ pour tout } k \in \mathbb{N}^*. \end{cases}$$

On rappelle (cf. examen partiel du 15 mars 2019) que la suite $(x_k)_{k\in\mathbb{N}}$ converge, quels que soient x_0 et x_1, si et seulement si $\rho(C) < 1$ où

$$C = \begin{pmatrix} B & B \\ I_3 & 0 \end{pmatrix}$$

1. On considère la matrice et les vecteurs

$$B = \frac{1}{20}\begin{pmatrix} 1 & 2 & 3 \\ 2 & 0 & 2 \\ 3 & 4 & 5 \end{pmatrix}, \quad b = \begin{pmatrix} 1 \\ 1 \\ 1 \end{pmatrix}, \quad u^{(0)} = \begin{pmatrix} 1 \\ 1 \\ 1 \end{pmatrix}, \quad u^{(1)} = \begin{pmatrix} 2 \\ 2 \\ 2 \end{pmatrix}$$

et la suite de vecteurs définie par

$$\forall n \in \mathbb{N}^*, \quad u^{(n+1)} = Bu^{(n)} + Bu^{(n-1)} + b\,.$$

(a) Calculer les premiers termes de la suite $(u^{(n)})_n$. On pourra utiliser `print("%.20f",...)` pour que les nombres soient affichés sous python avec 20 chiffres après la virgule. Qu'observe-t-on ?

RÉPONSE :

(b) Quel rayon spectral doit-on calculer pour justifier la convergence ? Déterminer sa valeur.

RÉPONSE :

(c) Si elle converge, quelle est la limite de la suite $(u^{(n)})_n$? Justifier la réponse.

RÉPONSE :

2. Répondre aux mêmes questions avec

$$B = \frac{1}{10}\begin{pmatrix} 1 & 2 & 3 \\ 2 & 0 & 2 \\ 3 & 4 & 5 \end{pmatrix}, \quad b = \begin{pmatrix} 1 \\ 1 \\ 1 \end{pmatrix}, \quad u^{(0)} = \begin{pmatrix} 1 \\ 1 \\ 1 \end{pmatrix}, \quad u^{(1)} = \begin{pmatrix} 2 \\ 2 \\ 2 \end{pmatrix}$$

RÉPONSES :

(a)

(b)

(c)

Exercice 2. (*Calcul de valeurs propres*)

Méthode de la puissance itérée. On rappelle qu'il s'agit d'une méthode itérative très simple permettant de calculer une approximation de la valeur propre de plus grand module (lorsqu'il n'y en a qu'une qui ait ce module) d'une matrice A et un vecteur propre associé.

Une implémentation sous python de cette fonction est donnée ci-dessous.

```
def puissance(A,z0,tol,nitermax):
    q = z0/npl.norm(z0)
    q2 = q
    err = []
    nu1 = []
    res = tol + 1
    niter = 0
    z = np.dot(A,q)
    while ((res > tol) and (niter <= nitermax)):
        q = z/npl.norm(z)
        z = np.dot(A,q)
        lam = np.dot(q,z)
        x1 = q
        z2 = np.dot(q2,A)
        q2 = z2/npl.norm(z2)
        y1 = q2
        c = np.dot(y1,x1)
        if c > 5E-2:
            res = npl.norm(z - lam*q)/c
            niter = niter + 1
            err.append(res)
```

```
        nu1.append(lam)
    else:
        print("Problème de convergence !")
        break
return(nu1,x1,niter,err)
```

Les valeurs en entrée `z0`, `tol` et `nitermax` sont respectivement le vecteur initial, la tolérance pour le test d'arrêt et le nombre maximum d'itérations admissible. En sortie, `x1` et `niter` sont respectivement les approximations du vecteur propre unitaire x_1 et le nombre d'itérations nécessaire à la convergence de l'algorithme, le vecteur `nu1` contient les approximations successives de la valeur propre cherchée, tandis que `err` contient la taille des résidus successifs.

1. Programmer la fonction `puissance`.

2. On considère la matrice
$$A = \begin{pmatrix} 15 & -1 & 1 \\ 1 & 9 & -2 \\ -2 & 2 & 0 \end{pmatrix}$$
   (a) Utiliser la fonction `puissance` pour rechercher la valeur propre dominante de A, ainsi qu'un vecteur propre associé en prenant le vecteur $z_0 = \begin{pmatrix} 1 & 1 & 1 \end{pmatrix}^T / \sqrt{3}$ comme vecteur initial, et une tolérance égale à 10^{-8} pour le critère d'arrêt.

   (b) Valider le résultat en utilisant la commande `eigvals` dans `numpy.linalg` de python.

RÉPONSES :

(a)

(b)

3. On veut évaluer la vitesse de convergence de la méthode. Pour cela, on considère, pour $a \in \mathbb{R}$, la matrice
$$A = \begin{pmatrix} 1 & a & a \\ a & 1 & a \\ a & a & 1 \end{pmatrix}$$
   dont on rappelle qu'elle est diagonalisable et a pour valeurs propres $1-a$ (double) et $1+2a$.

   (a) Tester la méthode avec par exemple un vecteur z_0 choisi aléatoirement et tracer, sur une même figure, le logarithme de la norme du résidu en fonction du numéro d'itération, ainsi que la droite passant par 0 et de pente $(\log|\lambda_2| - \log|\lambda_1|)$ où λ_1 est la valeur propre de plus grand module et λ_2 l'autre valeur propre, pour différentes valeurs de $a \in]0,1[$. Pour la valeur $a = 1/2$, donner un titre (commande `plt.title()`), une légende à chaque

courbe (commande `plt.legend()`) et nommer les axes (commandes `plt.xlabel()`, `plt.ylabel()`), et
↪ *Sauvegarder la figure dans un fichier* `figure1.jpg` (au format jpg).
Qu'observe-t-on ? Pourquoi ?

RÉPONSE :

(b) Faire la même chose pour $a = -1$.
↪ *Sauvegarder la figure dans un fichier* `figure2.jpg` (au format jpg).
Le théorème vu en cours permet-il de conclure à la convergence de la méthode dans ce cas ? Qu'observe-t-on ?

RÉPONSE :

4. On considère maintenant la matrice

$$A = \begin{pmatrix} 1 & 1 \\ 0 & 1 \end{pmatrix}.$$

Le théorème vu en cours permet-il de conclure à la convergence de la méthode dans ce cas ? Pourquoi ?

RÉPONSE :

En initialisant avec $z_0 = (1, 1)$, tester la méthode de la puissance et tracer, sur une même figure, le logarithme de la norme du résidu en fonction du *logarithme* du numéro d'itération, ainsi que la droite passant par 0 et de pente -1. Donner un titre et nommer les axes, et
↪ *Sauvegarder la figure dans un fichier* `figure3.jpg` (au format jpg).

Quel semble être le comportement du résidu en fonction du numéro d'itération ?

RÉPONSE :

Méthode QR et calcul de valeurs propres. On rappelle que lorsqu'une matrice A a des valeurs propres de modules tous distincts, les matrices de la suite (T^k) définie par la récurrence

$$\begin{array}{l} T^0 = A \\ Q^k R^k = T^{(k-1)} \quad \text{(décomposition QR de } T^{(k-1)}\text{)} \\ T^k = R^k Q^k \end{array}$$

voient leurs termes sous-diagonaux converger vers 0, leurs termes sur-diagonaux rester bornés et leurs termes diagonaux converger vers les valeurs propres de A. Si A est symétrique alors chacune de ces matrices est symétrique, et elles convergent vers une matrice diagonale.

La commande `qr` dans `numpy.linalg` de python contient une implémentation de la décomposition QR. On peut alors écrire une implémentation directe de la méthode QR utilisant cette fonction de la manière suivante

```
def methode_qr(A,niter):
    T = A
    for i in range(niter):
        Q,R = npl.qr(T);
        T = np.dot(R,Q);
    return(T,Q,R)
```

où `niter` est le nombre d'itérations souhaité.

1. Programmer la fonction `methode_qr`.

2. Construire une matrice symétrique A de taille 4×4 telle que $a_{ij} = 4+i-j$ pour $1 \leq i \leq j \leq 4$.

   RÉPONSE :

   A =

3. Vérifier que la matrice T^{20} obtenue après 20 itérations de la méthode QR (avec la matrice A de la question 2) est « presque » diagonale (les éléments non diagonaux sont presque nuls).

   RÉPONSE :

4. Utiliser la fonction `npl.eigvals` pour vérifier les valeurs propres obtenues.

   RÉPONSE :

5. Programmer la version suivante (vue en cours) de l'algorithme global de recherche de valeurs propres.

```
def methode_qr_2(A,niter):
    Z = A
    for i in range(niter):
        Q,R = npl.qr(Z)
        Z = np.dot(A,Q)
    T = np.dot(np.dot(np.transpose(Q),A),Q)
    return(T,Q,R)
```

où `niter` est le nombre d'itérations souhaité. Utiliser cette fonction pour la même matrice A de la question 2, avec 20 itérations. Vérifier que les colonnes de Q sont des approximations des vecteurs propres de A (ceci est dû à la symétrie de A). Quel est l'avantage de cette version de l'algorithme ?

RÉPONSE :