Université Claude Bernard - Lyon 1 **Semestre automne 2023-2024**
L3 - Mathématiques UE : Algèbre linéaire, bilinéaire, analyse matricielle

---

TP 1

Pour commencer, connectez-vous au serveur L3 https://jupyterl3.mecanique.univ-lyon1.fr/ avec vos identifiant et mot de passe de l'université.

Cliquez sur Start pour ouvrir un notebook.

Ensuite, récupérer le notebook du TP 1 en procédant comme suit :

1. Choisir *Assignments*.
2. Dans le menu *Released, downloaded, and submitted assignments for course*, choisir MAT3146L.
3. Puis *Fetch* `TP1_16_novembre` dans la rubrique *Released assignments*.
4. Cliquer sur le lien `TP1_16_novembre` se trouvant à gauche dans la rubrique *Downloaded assignments*, puis juste en-dessous sur le lien vers le notebook `tp_1`.

On charge les modules numpy, numpy.linalg, matplotlib et scipy.stats qui seront utilisés dans le TP.

[ ]:
```
import numpy as np
import numpy.linalg as npl
import matplotlib.pyplot as plt
import scipy.stats
%matplotlib notebook
```

1 Exercice 1 : Décomposition en valeurs singulières

1.1 Première partie : énoncé.

Soit $A \in \mathcal{M}_n(\mathbb{R})$ inversible, avec $n \in \mathbb{N}^*$. En utilisant la décomposition en valeurs singulières de A, $A = V\Sigma U^t$ où U et V sont unitaires, et Σ est diagonale, nous allons montrer que l'image de la boule unité par A est une ellipse de centre 0, dont on sait déterminer les axes.

Nous commençons par montrer qu'une telle décomposition de A existe.

Dans la suite, les vecteurs de $\mathbb{R}^n$ sont identifiés à des vecteurs colonnes et $\|\cdot\|$ désigne la norme euclidienne canonique.

1.1.1 1) Existence de la décomposition

a) Montrer que A^tA est symétrique et définie positive. En déduire qu'il existe des réels strictement positifs $\mu_1, \ldots, \mu_n$ et une base orthonormée $(u_1, \ldots, u_n)$ tels que pour tout $k = 1, \ldots, n$, $A^tAu_k = \mu_k^2 u_k$.

On dit que $\mu_1, \ldots, \mu_n$ sont les valeurs singulières de A.

[ ]:
```
#
#
#
#
```

```
#
#
```

b) Pour tout $k = 1, \ldots, n$, on pose $v_k = \frac{1}{\mu_k} A u_k$. Montrer que $(v_1, \ldots, v_n)$ est une base orthonormée de $\mathbb{R}^n$, constituée de vecteurs propres de AA^t.

[ ]:
```
#
#
#
#
#
#
#
#
#
#
#
#
#
```

c) On note U la matrice dont les vecteurs colonnes sont $(u_1, \ldots, u_n)$ et V la matrice dont les vecteurs colonnes sont $(v_1, \ldots, v_n)$. On note $\Sigma = \text{diag}(\mu_1, \ldots, \mu_n)$. Montrer que $A = V \Sigma U^t$.

[ ]:
```
#
#
#
#
#
#
#
#
#
#
#
#
#
```

1.1.2 2) Image de la sphère unité par A

a) Montrer que l'image de la sphère unité $S = \{x \in \mathbb{R}^n \text{ t.q. } \|x\| = 1\}$ par la matrice unitaire U^t est S.

[ ]:
```
#
#
#
#
#
#
```

b) Montrer que l'image de la sphère unité par la matrice diagonale Σ est l'ellipse

$$E_1 = \left\{ x \in \mathbb{R}^n \text{ t.q. } \sum_{k=1}^{n} \left(\frac{x_k}{\mu_k} \right)^2 = 1 \right\}.$$

E_1 est l'ellipse dont les axes sont ceux du repère canonique et les demi-longueurs sont les $(\mu_k)_{1 \leq k \leq n}$.

[ ]:
```
#
#
#
#
#
#
```

c) Montrer que l'image par la matrice orthogonale V de l'ellipse E_1 est l'ellipse

$$E_2 = \left\{ x \in \mathbb{R}^n \text{ t.q. } \sum_{k=1}^{n} \left(\frac{(V^t x)_k}{\mu_k} \right)^2 = 1 \right\}.$$

E_2 est l'ellipse dont les axes sont les vecteurs colonnes de V et les demi-longueurs sont les $(\mu_k)_{1 \leq k \leq n}$.

[ ]:
```
#
#
#
#
#
#
#
#
```

d) En déduire que l'image par A de la sphère unité S est l'ellipse, notée E_A, dont les axes sont les vecteurs colonnes v_k de V et les demi-longueurs respectives les μ_k.

[ ]:
```
#
#
#
#
#
#
```

e) En déduire que l'image par la matrice A de la boule unité est l'ellipse « pleine »

$$\left\{ x \in \mathbb{R}^n \text{ t.q. } \sum_{k=1}^{n} \left(\frac{(V^t x)_k}{\mu_k} \right)^2 \leq 1 \right\}.$$

[ ]:
```
#
#
#
```

```
#
#
#
```

1.2 Seconde partie : visualisation.

Le but de la suite cet exercice est de *visualiser* le fait que l'image de la boule unité par une matrice réelle carrée A est un ellipsoïde dont les axes sont les vecteurs propres de AA^t et dont les demi-longueurs sont les valeurs singulières de A. Pour visualiser ceci on choisit la dimension 2.

1.2.1 1)

Définir la matrice

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

[ ]:
```
A = np.array([[1, 1],[0,1]])
print(A)
```

1.2.2 2)

Choisir 1000 points aléatoirement selon la loi uniforme dans $B(0,1)$ et les représenter.

On admet qu'une solution pour définir un tel ensemble de points est de chercher des points aléatoirement selon la loi uniforme dans le carré $[-1,1]^2$ *et de ne sélectionner que ceux qui sont dans la boule.*

[ ]:
```
N = 1000
n = 1
Y = np.empty((N,2))
while n < N:
    X = 2.*np.random.rand(1,2) - 1
    if npl.norm(X) <= 1:
        Y[n,:] = X
        n = n + 1
plt.plot(Y[:,0],Y[:,1],".")
plt.axis('equal')
plt.grid()
plt.show()
```

1.2.3 3)

Calculer l'image de ces 1000 points par l'application dont la matrice est A et les tracer sur le même dessin.

[ ]:
```
Z = np.empty((N,2))
for n in range(N):
    Z[n,:] = # à compléter
plt.plot(Z[:,0],Z[:,1],".")
plt.axis('equal')
plt.grid()
plt.show()
```

1.2.4 4)

À l'aide de la fonction `npl.eig` calculer les (valeurs approchées des) valeurs propres λ_1 et λ_2 et vecteurs propres associés respectivement u_1 et u_2 de A^tA puis calculer Au_1 et Au_2. *On peut aussi faire le calcul exact à la main !*

[ ]:
```
AtA = np.dot(np.transpose(A),A)
valpro, vecpro = npl.eig(AtA)
l1 = valpro[0]
u1 = vecpro[:,0]
l2 = valpro[1]
u2 = vecpro[:,1]
Au1 = # à compléter
Au2 = # à compléter
vs1 = np.sqrt(l1)
vs2 = np.sqrt(l2)

print(Au1)
print(Au2)
print(np.dot(u1,u2))
```

1.2.5 5)

Représenter sur le même dessin les segments ayant une extrémité $0 \in \mathbb{R}^2$, alignés avec Au_1 et Au_2 (ils sont de longueurs $\sqrt{\lambda_1}$ et $\sqrt{\lambda_2}$).

[ ]:
```
plt.plot([0.,Au1[0]],[0.,Au1[1]],'b')
plt.plot([0.,Au2[0]],[0.,Au2[1]],'b')
plt.grid()
plt.show()
```

2 Exercice 2 : conditionnement et stabilité des solutions de systèmes linéaires

On s'intéresse à la solution du problème $Ax = b$ où

$$A = \begin{pmatrix} 1 & 0.1 & 0.01 & 0.001 & 0.0001 \\ 1 & 1 & 1 & 1 & 1 \\ 1 & 1.5 & 2.25 & 3.375 & 5.0625 \\ 1 & 2 & 4 & 8 & 16 \\ 1 & 3 & 9 & 27 & 81 \end{pmatrix} \quad \text{et} \quad b = \begin{pmatrix} 0.01 \\ 1 \\ 2.25 \\ 4 \\ 9 \end{pmatrix}.$$

2.1 1)

Le module `numpy` de python a une fonction de résolution de systèmes linéaires : la commande `npl.solve(A,b)` renvoie le résultat x du système ci-dessus. Quel est le résultat obtenu ? Aurait-on pu le trouver sans calcul ?

[ ]:
```
A = np.array([[1,0.1,0.01,0.001,0.0001],[1,1,1,1,1],[1,1.5,2.25,3.375,5.
↪0625],[1,2,4,8,16],[1,3,9,27,81]])
b = np.array([0.01,1,2.25,4,9])
print(A)
print(b)
print(npl.solve(A,b))
```

2.2 2)

Rappeler la définition de $cond_2(A)$ et en donner une valeur approchée en utilisant la commande `npl.cond`.

[ ]:
```
#
#
```

[ ]:
```
print(npl.cond(A,2))
```

2.3 3)

À l'aide de la commande `10**-3*np.random.rand(np.shape(A)[0],np.shape(A)[1])`, définir une matrice $B = A + \delta A$ où δA est définie de manière « aléatoire ». Résoudre $By = b$ et évaluer la quantité

$$\frac{\|y - x\|_2}{\|y\|_2} \frac{|||A|||_2}{|||\delta A|||_2}$$

en utilisant la commande `npl.norm`. Faire ceci pour différentes perturbations δA.

[ ]:
```
dA = 10**-3*np.random.rand(np.shape(A)[0],np.shape(A)[1])
B = A + dA
x = # à compléter
y = # à compléter
rapportdA = npl.norm(y - x)/npl.norm(y)*npl.norm(A,2)/npl.norm(dA,2)
# Attention : par défaut pour un vecteur npl.norm renvoie la norme 2,
# mais par défaut pour une matrice c'est la norme de Frobenius (considérée
# donc comme un vecteur) : pour avoir la norme subordonnée 2 il faut préciser le 2␣
↪en second argument.
condA = npl.cond(A,2)
print(rapportdA)
print(condA)
```

[ ]:
```
# Noter ici les différentes valeurs trouvées.
#
#
#
#
```

Avec quoi peut-on majorer cette quantité (résultat de cours et travaux dirigés) ?

[ ]:
```
#
#
#
```

2.4 4)

En s'inspirant de la question précédente, calculer la solution z du problème $Az = b + \delta b$ où δb est défini aléatoirement, puis évaluer la quantité

$$\frac{\|z - x\|_2}{\|x\|_2} \frac{\|b\|_2}{\|\delta b\|_2}.$$

Faire ceci pour différentes perturbations δb.

[ ]:
```
db = 10**-3*np.random.rand(np.size(b))
c = b + db
z = # à compléter
rapportdb = npl.norm(z - x)/npl.norm(x)*npl.norm(b)/npl.norm(db)
print(rapportdb)
print(condA)
```

[ ]:
```
# Noter ici les différentes valeurs trouvées.
#
#
#
#
```

Avec quoi peut-on majorer cette quantité (résultat de cours et travaux dirigés) ?

[ ]:
```
#
#
#
```

2.5 5)

Soit $D = \mathrm{diag}(A_{1,1}, \ldots, A_{n,n}) \in \mathcal{M}_5(\mathbb{R})$. On pose $A_2 = D^{-1}A$ et $b_2 = D^{-1}b$ et on considère le système linéaire $A_2 x = b_2$. Construire la matrice A_2 et le vecteur b_2 en utilisant deux fois `np.diag`, sans inverser D.

Reprendre les questions 3) et 4) avec $\delta A_2 = D^{-1}\delta A$ et $\delta b_2 = D^{-1}\delta b$. Que peut-on en conclure ?

[ ]:
```
A2 = # à compléter
b2 = # à compléter
x2 = npl.solve(A2,b2)
dA2 = # à compléter
B2 = A2 + dA2
y2 = npl.solve(B2,b2)
db2 = # à compléter
c2 = b2 + db2
z2 = npl.solve(A2,c2)
```

[ ]:
```
print(x)
print(x2)
print(y)
print(y2)
print(z)
print(z2)
```

```
[ ]: #rapportdA = npl.norm(y - x)/npl.norm(y)*npl.norm(A,2)/npl.norm(dA,2)
     #condA = npl.cond(A,2)
     print(rapportdA)
     print(condA)
     rapportdA2 = npl.norm(y2 - x2)/npl.norm(y2)*npl.norm(A2,2)/npl.norm(dA2,2)
     condA2 = npl.cond(A2,2)
     print(rapportdA2)
     print(condA2)
```

```
[ ]: print(rapportdb)
     print(condA)
     rapportdb2 = npl.norm(z2 - x2)/npl.norm(x2)*npl.norm(b2)/npl.norm(db2)
     print(rapportdb2)
     print(condA2)
```

```
[ ]: # Conclusion.
     #
     #
     #
     #
```

3 Exercice 3 : deux méthodes itératives pour la résolution de systèmes linéaires, les méthodes Gauss-Seidel et de Jacobi.

Une méthode itérative pour résoudre le système linéaire $Ax = b$ consiste en une décomposition de la matrice A sous la forme $A = M - N$ puis la définition de la suite

$$\begin{cases} x_0 \in \mathbb{R}^n \\ \text{pour tout } k \in \mathbb{N},\ x_{k+1} = M^{-1}Nx_k + M^{-1}b \end{cases}$$

On vérifie sans peine que si la suite est stationnaire ($x_{k+1} = x_k$ pour tout k), x_k est la solution cherchée, car $(M - N)x_k = b$ pour tout k. On identifiera en cours diverses conditions sur M et N sous lesquelles la suite *converge* (pour tout x_0) vers la solution cherchée (quelques indications apparaissent déjà dans la suite de cet exercice).

La méthode de *Gauss-Seidel* (cf. cours) est obtenue en posant $M = D - E$ et $N = F$ où D est la matrice diagonale dont la diagonale est celle de A, $-E$ correspond à la partie sous-diagonale de A et $-F$ à sa partie sur-diagonale. On définit donc la suite approximante par

$$\begin{cases} x_0 \quad \text{donné} & \text{(Gauss-Seidel)} \\ \forall k \in \mathbb{N},\ (D - E)x_{k+1} = Fx_k + b \end{cases}$$

Pour la méthode de *Jacobi*, on pose $M = D$ et $N = E + F$ avec les mêmes définitions ; l'itération s'écrit

$$\begin{cases} x_0 \quad \text{donné} & \text{(Jacobi)} \\ \forall k \in \mathbb{N},\ Dx_{k+1} = (E + F)x_k + b \end{cases}$$

3.0.1 1)

Utiliser les commandes Python `numpy.tril` et `numpy.triu` pour définir D, E et F et vérifier que $D - E - F = A$. **Attention aux signes**.

[ ]:

3.0.2 2)

Soit $n \in \mathbb{N}^*$, $B \in \mathcal{M}_n(\mathbb{R})$ et $c \in \mathbb{R}^n$. Soit la suite récurrente définie par :

$$\begin{cases} x_0 \in \mathbb{R}^n \\ \text{pour tout } k \in \mathbb{N},\ x_{k+1} = Bx_k + c \end{cases}$$

Montrer que la suite $(x_k)_{k\in\mathbb{N}}$ converge, quel que soit x_0, si et seulement si $\rho(B) < 1$. On pourra introduire la quantité $x_k - \overline{x}$ où $\overline{x}$ est *la* solution de $\overline{x} = B\overline{x} + c$.

[ ]:
```
#
#
#
#
#
#
#
#
#
#
#
#
#
#
#
```

En déduire une condition nécessaire et suffisante pour que les méthodes de Jacobi et Gauss-Seidel convergent.

[ ]:
```
#
#
```

Prédire, à l'aide de Python, le comportement (convergence ou non) de ces deux méthodes pour la matrice A de l'exercice précédent.

[ ]:
```
# À compléter.
```

3.0.3 3)

Voici une fonction qui calcule `niter` itérations de la méthode de Gauss-Seidel, et qui renvoie la solution approchée `u` et la matrice `res` contenant la suite des itérés calculés : x_0, x_1, $x_2 \ldots$

[ ]:
```
def gauss_seidel(A,b,x0,niter):
    D = np.diag(np.diag(A))
    E = -np.tril(A) + D
    F = -np.triu(A) + D
    sol = npl.solve(A,b)
    u = x0
    res = x0
    resn = npl.norm(u-sol)
```

```
    for k in range(niter):
        u = npl.solve((D-E),(np.dot(F,u)+b))
        res = np.vstack((res, u))
        resn = np.vstack((resn,npl.norm(u-sol)))
    return(u,res,resn)
```

Tester cette fonction (choisir `x0` et `niter`). Que contiennent `u`, `res` et `resn` ? Comment afficher l'erreur entre la solution approchée et la solution exacte ?

[ ]:
```
#
#
#
#
```

[ ]:
```
# À compléter.
```

3.0.4 4) Vitesse de convergence.

On note pour tout $k \in \mathbb{N}$, $e_k = \|x_k - x\|$, où x est la solution de $Ax = b$, une norme de l'erreur à l'itération k. Lors de l'utilisation d'une méthode itérative, s'il existe deux constantes positives α et C telles que

$$\lim_{k \to +\infty} \frac{e_{k+1}}{(e_k)^\alpha} = C,$$

on dit que l'ordre de la méthode est α et que sa vitesse de convergence est C.

On s'intéresse à l'ordre et la vitesse de convergence de la méthode de Gauss-Seidel. Pour cela, on va tracer des fonctions de l'erreur. Exécuter la suite d'instructions suivante et interpréter la figure tracée.

[ ]:
```
x0 = np.array([1,5,1,5,1])
n = 50
u,res,resn = gauss_seidel(A,b,x0,n)
plt.figure("Étude de la vitesse de convergence de la méthode de␣
 ↪Gauss-Seidel",figsize=(14,8))
plt.plot(np.log(resn[:n]),np.log(resn[1:]),'*')
plt.grid()
plt.show()
resn = resn.flatten()
pente,oao,_,_,_ = scipy.stats.linregress(np.log(resn[:n]),np.log(resn[1:]))
print("pente : ",pente)
print("ordonnée à l'origine : ",oao)
```

Que calcule la commande `linregress` ? Où, sur la figure, peut-on retrouver les nombres calculés ?

[ ]:
```
#
#
#
#
#
```

3.0.5 5)

Refaire les questions 3) et 4) pour la méthode de Jacobi.

[ ]:

[ ]:

[ ]:

4 Exercice 4 : factorisation QR par Gram-Schmidt (cas de matrices à coefficients réels).

4.0.1 1)

Programmer une fonction `QR_GS` qui à une matrice réelle A associe sa factorisation QR obtenue par l'algorithme de Gram-Schmidt vu en cours.

La tester sur une matrice de $\mathcal{M}_3(\mathbb{R})$.

[ ]:
```
def QR_GS(A):
    m,n = A.shape # m doit être supérieur ou égal à n, m = n si A est carrée.
    R = np.zeros((n,n))
    Q = np.zeros((m,n))
    for j in range(n):
        Q[:,j] = A[:,j].copy()
        for k in range(j):
            # à compléter
        Q[:,j] = Q[:,j]/npl.norm(Q[:,j])
    R = np.dot(np.transpose(Q),A)
    return (Q,R)
```

4.0.2 2)

Tester l'algorithme avec la matrice

$$B = \begin{pmatrix} 1 & 1 & 0 \\ 2 & 1 & 0 \\ 1 & 1 & 1 \end{pmatrix}$$

et estimer la précision du résultat en calculant la norme de $Q^TQ - I_3$ ainsi que la norme de $B - QR$.

[ ]:
```
B = np.array([[1,1,0],[2,1,0],[1,1,1]])
Q,R = QR_GS(B)
print("Q = ",Q)
print("R = ",R)
print("Q^T Q -I = ", ) # à compléter
print("QR - B = ", ) # à compléter
```

4.0.3 3)

Même question avec

$$B = \begin{pmatrix} 1 & 0.1 & 0.01 & 0.001 & 0.0001 \\ 1 & 1 & 1 & 1 & 1 \\ 1 & 1.5 & 2.25 & 3.375 & 5.0625 \\ 1 & 2 & 4 & 8 & 16 \\ 1 & 3 & 9 & 27 & 81 \end{pmatrix}.$$

[ ]:
```
B = np.array([[1,0.1,0.01,0.001,0.0001],[1,1,1,1,1],[1,1.5,2.25,3.375,5.
↪0625],[1,2,4,8,16],[1,3,9,27,81]])
Q,R = QR_GS(B)
print("Q = ",Q)
print("R = ",R)
print("||Q^T Q -I|| = ", ) # à compléter
print("||QR - B|| = ", ) # à compléter
```

5 Exercice 5 : factorisation QR par les symétries de Householder.

Le but de cet exercice est d'obtenir la décomposition $A = QR$ par l'algorithme de Householder, pour une matrice réelle. Il consiste à multiplier la matrice A de départ par une suite de matrices orthogonales très simples pour obtenir une matrice triangulaire supérieure.

Notations. Soit $n \in \mathbb{N}^*$. Dans la suite, les vecteurs de $\mathbb{R}^n$ sont identifiés à des vecteurs colonnes et $\|\cdot\|$ désigne la norme euclidienne canonique. Ainsi, pour tout $u \in \mathbb{R}^n$, $\|u\|^2 = u^T u$.

5.1 Première partie : définition de l'algorithme.

5.1.1 1)

À tout vecteur $u \in \mathbb{R}^n$ on associe la matrice de Householder définie par

$$H(u) = \begin{cases} I_n - 2\frac{uu^T}{\|u\|^2} & \text{si } u \neq 0, \\ I_n & \text{sinon.} \end{cases}$$

a. Montrer que, pour tout $u \in \mathbb{R}^n$, $H(u)$ est symétrique et othogonale.
$H(u)$ est la matrice de la symétrie orthogonale par rapport à l'hyperplan orthogonal à u. Faire un dessin.

[ ]:
```
#
#
#
#
#
#
#
#
#
#
#
#
```

b. Soit e un vecteur unitaire de $\mathbb{R}^n$.
Montrer que, pour tout $v \in \mathbb{R}^n$, si v et e ne sont pas colinéaires, alors

$$H(v + \|v\|e)\, v = -\|v\|\, e \qquad \text{et} \qquad H(v - \|v\|e)\, v = \|v\|\, e.$$

[ ]:
```
#
#
#
#
#
#
#
#
#
#
```

5.1.2 2)

Soit $A \in \mathcal{M}_n(\mathbb{R})$.

a. Déterminer une matrice de Householder H telle que la matrice HA n'ait que des zéros sous la diagonale dans sa première colonne.

[ ]:
```
#
#
#
#
#
#
#
#
#
#
```

b. Construire une suite de matrices de Householder $(H^{(k)})_{1 \leq k \leq n-1}$ et une suite de matrices $(A^{(k)})_{0 \leq k \leq n-1}$ telles que
i) $A^{(0)} = A$;
ii) pour tout $0 \leq k \leq n-2$, $A^{(k+1)} = H^{(k+1)} A^{(k)}$;
iii) $A^{(n-1)}$ est triangulaire.

[ ]:
```
#
#
#
#
#
#
#
#
#
#
```

5.2 Seconde partie : mise en œuvre.

5.2.1 1)

Programmer une fonction `householder` qui à un vecteur v associe la matrice de Householder $H(v)$.
Programmer une fonction `symetrie` qui à un vecteur e et un vecteur x associe le vecteur $x + \|x\|e$ (ou le vecteur $x - \|x\|e$).
Les tester.

[ ]:
```
# Pour une matrice réelle :
def householder(v):
    n = np.size(v)
    Hv = # à compléter
    return(Hv)
def symetrie(e,x): # e doit être normé, et x non nul, de même dimension.
    n = np.size(e)
    lamb = np.dot(e.T,x)/abs(np.dot(e.T,x))*npl.norm(x)
    if npl.norm(x + lamb*e) > npl.norm(x - lamb*e):
        # On choisit le plus grand des deux vecteurs possibles, pour des raisons de␣
 ↪stabilité de l'algorithme.
        v = # à compléter
    else:
        v = # à compléter
    return(v)

v = np.array([[1,1,1]])
v = np.transpose(v)
e = np.array([[1,0,0]])
e = np.transpose(e)
print(np.dot(householder(symetrie(e,v)),v))
```

5.2.2 2)

Programmer une fonction `QR_HH` qui à une matrice réelle A associe une factorisation QR de A obtenue par l'algorithme de Householder (cette fonction utilisera explicitement les fonctions `householder` et `symetrie`, au bénéfice de la lisibilité du programme mais aux dépens de sa vitesse d'exécution). La tester sur les matrices de l'exercice 4.

[ ]:
```
def QR_HH(A):
    m,n = A.shape
    R = np.zeros((n,n))
    Q = np.zeros((m,n))
    H = np.eye(m,m)
    for j in range(n):
        x = np.zeros((n-j,1))
        x[:,0] = A[j:,j]
        e = np.zeros((n-j,1))
        e[0,0] = 1.
        if npl.norm(x) > 0.:
            Hj = np.eye(m,m)
            Hj[j:,j:] = # à compléter
```

```
            A = np.dot(Hj,A)
            H = np.dot(Hj,H)
    R = A
    Q = np.conj(H).T
    return(Q,R)
```

[ ]:
```
B = np.array([[1,1,0],[2,1,0],[1,1,1]])
Q,R = QR_HH(B)
print("Q = ",Q)
print("R = ",R)
print("QR - B = ",np.dot(Q,R) - B)
```

[ ]:
```
B = np.array([[1,0.1,0.01,0.001,0.0001],[1,1,1,1,1],[1,1.5,2.25,3.375,5.
↪0625],[1,2,4,8,16],[1,3,9,27,81]])
Q,R = QR_HH(B)
print("Q = ",Q)
print("R = ",R)
print("||Q^T Q -I|| = ",npl.norm(np.dot(np.transpose(Q),Q) - np.eye(5,5)))
print("||QR - B|| = ",npl.norm(np.dot(Q,R) - B))
```

5.3 Comparer la précision des méthodes de factorisation QR de Gram-Schmidt et de Householder sur la matrice de Hilbert en dimension 30.

La matrice de Hilbert $H \in \mathcal{M}_n(\mathbb{R})$ est définie par

$$H_{i,j} = \frac{1}{i+j-1}, \quad i,j = 1,\dots n.$$

5.3.1 1)

Construire la matrice de Hilbert en dimenson $n \in \mathbb{N}$.

[ ]:
```
def hilbert(n):
    H = np.empty((n,n))
    for i in range(n):
        for j in range(n):
            H[i,j] = # à compléter
    return(H)
```

5.3.2 2)

Calculer avec python le conditionnement de la matrice pour $n = 30$.

[ ]:
```
H = hilbert(30)
print(H)
print(npl.cond(H))
```

5.3.3 3)

Calculer une factorisation QR de cette matrice avec l'algorithme de Gram-Schmidt et estimer la précision du résultat en calculant la norme de $Q^TQ - I_{30}$ ainsi que la norme de $H - QR$.

[ ]:
```
Q,R = QR_GS(H)
I = np.eye(len(H))
print("||Q^T Q -I|| = ",npl.norm(np.dot(np.transpose(Q),Q) - I))
print("||QR - H|| = ",npl.norm(np.dot(Q,R) - H))
```

5.3.4 4)

Calculer une factorisation QR de cette matrice avec l'algorithme de Householder et estimer la précision du résultat en calculant la norme de $Q^TQ - I_{30}$ ainsi que la norme de $H - QR$.

[ ]:
```
Q,R = QR_HH(H)
I = np.eye(len(H))
print("||Q^T Q -I|| = ",npl.norm(np.dot(np.transpose(Q),Q) - I))
print("||QR - H|| = ",npl.norm(np.dot(Q,R) - H))
```

Commenter.

[ ]:
```
#
#
#
#
#
#
```

5.3.5 5) Une application de la factorisation QR

Calculer le déterminant d'une matrice (par exemple une matrice de Hilbert) en utilisant sa factorisation QR. On pourra comparer le résultat avec celui de la fonction `npl.det`.

[ ]:
```
H = hilbert(10)
Q,R = QR_GS(H)
determinant = 1.
for j in range(R.shape[0]):
    determinant = determinant* # à compléter
print(determinant)
Q,R = QR_HH(H)
determinant = 1.
for j in range(R.shape[0]):
    determinant = determinant* # à compléter
print(determinant)
print(npl.det(H))
```