Prépa Math 2 Centrale 2017

Exercice 1. Soit pour tout $n \in \mathbb{N}$, $F_n : x \mapsto e^{-x}\left(\sum_{k=0}^{n} \frac{x^k}{k!}\right)$ et $\mu \in]0,1[$ fixé.

a) Montrer que pour tout $n \in \mathbb{N}$, l'équation $F_n(x) = \mu$ admet une unique solution dans $\mathbb{R}^+$ notée a_n.

b) Montrer que la suite (a_n) est croissante.

c) Avec Python. Définir la fonction F_n comme une fonction python. Définir une fonction `Trace` qui prend en paramètre un entier `n` et retourne le tracé de F_n sur le segment $[0, 2n]$. Que peut-on constater ?

d) Montrer que $a_n \underset{n\to+\infty}{\longrightarrow} +\infty$.

e) Définir une fonction `A(n,mu)` qui retrouve une valeur approchée de a_n avec une précision que l'on explicitera.

f) Tracer les points de coordonnées (n, a_n) pour n entier dans un intervalle raisonnable et le μ de votre choix. Que constate-t-on ?

Exercice 2 (Simulation Python d'une marche aléatoire sur $\mathbb{Z}$)**.** On considère une suite (X_n) de variables aléatoires mutuellement indépendantes telles que :

$$\forall\, n \in \mathbb{N},\ P(X_n = 1) = P(X_n = -1) = \frac{1}{2}.$$

Ces v.a. servent par exemple à modéliser la marche aléatoire sur $\mathbb{Z}$ d'une puce qui part de 0 au temps 0. A chaque étape, la puce ou bien avance de 1 cran vers la droite ou bien avance de 1 cran vers la gauche.

Notons $S_0 = 0$ et $\forall\, n \geq 1,\ S_n = S_0 + \sum_{k=1}^{n} X_k$ la position de la puce à la n-ième étape.

a) Ecrire à l'aide de `numpy` et `matplotlib.pyplot`, un programme python permettant d'afficher différentes marches aléatoires de la puce : on affichera les points de coordonnées (n, S_n), reliés par des segments.

b) Ecrire une fonction `TempsPremierRetour()` qui renvoie la valeur du premier indice $i > 0$ (disons strictement inférieur à `N=101`) tel que $S_i = 0$. S'il n'y a pas de premier retour avant `N` la fonction renverra `N+50` pour avoir une valeur bien à l'écart des autres, ce qui sera utile dans ce qui suit.

c) La fonction `plt.hist()` de `matplotlib.pyplot` permet d'afficher un histogramme à partir d'un tableau de valeurs : elle compte elle-même le nombre d'occurrence de chaque valeurs dans le tableau et affiche en abscisse ces valeurs et en ordonnée le nombre d'occurrences, sous forme de batons. Utilisez-la pour faire un histogramme des valeurs de premier retour pour 2000 essais de marche aléatoire sur $\mathbb{Z}$.

N.B. La fonction `plt.hist` peut s'utiliser simplement avec `plt.hist(L)`. Avec l'argument supplémentaire `bins=50`, elle va découper notre intervalle en 50 bâtons (bins en anglais).

d) Comparer le résultat expérimental obtenu avec le résultat théorique suivant : on peut montrer que la probabilité que la puce revienne à l'origine pour la première fois en $2n$ étapes est $\frac{1}{2^{2n-1}}\frac{1}{(2n-1)}\binom{2n-1}{n}$.

Exercice 3. Soit $A \in M_n(\mathbb{R})$.

a) Montrer qu'il existe une matrice O orthogonale et une matrice T triangulaire supérieure telles que $A = OT$.

On pourra commencer par le cas où la matrice A est inversible.

La fonction `numpy.linalg.qr` de PYTHON donne une telle décomposition.

b) On pose

$$N_1(A) = \sum_{1\leq i,j\leq n} |a_{i,j}|$$

Montrer que N_1 admet un minimum m_n et un maximum M_n sur $O_n(\mathbb{R})$.

c) Utilisation de PYTHON.

Écrire une fonction `randO(n)` qui génère une matrice aléatoire A et qui renvoie la matrice orthogonale O de la question précédente.

Écrire une fonction N_1 de la variable matricielle A qui renvoie $N_1(A)$. On pourra utiliser les fonctions `numpy.sum` et `numpy.abs`.

Écrire une fonction `test(n)` qui, sur 1 000 tests, renvoie le minimum et le maximum des valeurs de N_1 pour des matrices orthogonales aléatoires.

d) Déterminer la valeur de m_n. Pour quelles matrices, ce minimum est-il atteint ?
Montrer qu'il y a un nombre fini de telles matrices.

e) Montrer que $M_n \leq n\sqrt{n}$. Montrer que cette inégalité est une égalité pour $n = 2$ ou $n = 4$, mais que cette égalité est impossible pour n impair.

Exercice 4. Soient $(X_{i,j})_{(i,j)\in[\![1,n]\!]^2}$ des variables aléatoires indépendantes, de même loi définie par $P(X_{i,j} = 1) = \frac{1}{2}$ et $P(X_{i,j} = -1) = \frac{1}{2}$. On considère la variable aléatoire $D = \det(M)$ où M est la matrice aléatoire dont les entrées sont les $X_{i,j}$.

a) **Maths :** déterminer l'espérance et la variance de la v.a. D.

b) Expérimentation statistique avec Python (à l'aide de la documentation de Centrale) :

(i) écrire une fonction `test` qui prend en paramètre un entier `n` et qui renvoie la valeur d'une matrice `M` aléatoire comme définie ci-dessus de taille `n`.

(ii) Pour `N=10000` et `n=5` (par exemple), calculer la moyenne statistique des résultats de `N` test, qui doit s'approcher (on l'espère) de l'espérance de `D`.
Comment obtenir `var(D)` à partir d'une moyenne statistique ?

c) Une calcul avec la formule de l'espérance mathématique :

(i) Ecrire une fonction `Univers` qui reçoit un entier `n` (petit !) en paramètre et retourne la liste de *toutes* les matrices de tailles $n \times n$ ayant comme entrées des `1` et des `-1`.

(ii) En déduire le calcul de l'espérance mathématique et de la variance avec Python . (Le nombre de calculs étant très grand, on se limitera à `n=4`).

Exercice 5. L'algorithme de Gram-Schmidt en python : 1) a) Ecrire une fonction `GS1` qui reçoit en paramètre une matrice (rectangulaire) `X` et qui renvoie une autre matrice `A` dont les colonnes sont obtenues par l'orthogonalisation de Gram-Schmidt des colonnes de `X`.

b) Tester votre programme sur $X = \begin{pmatrix} 1 & 1 & 2 \\ 1 & 0 & -2 \\ -1 & 2 & 3 \end{pmatrix}$.

c) En déduire une fonction `GS2` avec les mêmes types d'entrées et sortie que `GS1` sauf que le colonnes de la matrices renvoyées sont `orthonormalisées`.

2) L'algorithme de Gram-Schmidt donne l'existence d'une décomposition QR comme à l'exercice vu plus haut.

Comparer sur la matrice de Hibert $H_n = (\frac{1}{i+j+1})_{(i,j)\in[\![0,n-1]\!]^2}$ (qui est très sensible aux erreurs de calculs numériques), votre `QR` par Gram-Schmidt, et celui de la fonction `np.linalg.qr`.

Exercice 6 (Algorithme de Fadeev du calcul du polynôme caractéristique)**.** Soit $A \in M_n(\mathbb{K})$ et $\chi_A(X) = \det(IX - A)$ son polynôme caractéristique qu'on note : $\chi_A(X) = X^n + \alpha_1 X^{n-1} + \cdots + \alpha_n$.

a) On note $B = Com(XI - A) = X^{n-1}B_0 + \cdots + XB_{n-1} + B_n$ où : $B_i \in M_n(\mathbb{K})$.

Exprimer les B_i comme des polynômes en la matrice A dont les coefficients sont données par les coefficients α_i du polynôme caractéristique et en déduire le théorème de Cayley-Hamilton.

b) En exprimant $\chi'_A(X)$ comme la trace d'une matrice à préciser, en déduire les formules de Fadeev suivantes : $\forall\, i \in [\![1, n]\!]$, $\mathrm{Tr}(A^i) + \alpha_1 \mathrm{Tr}(A^{i-1}) + \cdots + \alpha_{i-1}\mathrm{Tr}(A) + i\alpha_i = 0$.

c) En déduire un algorithme pratique de calcul des coefficients α_i du polynôme caractéristique de A : précisément si on pose $M_0 = A$ et pour tout $k \geq 1$, $M_k = A(M_{k-1} - \frac{1}{k}\mathrm{Tr}(M_{k-1})I_n)$, alors les $\mathrm{Tr}(M_k)$ permettent de calculer les coefficients de ce polynôme caractéristique.

d) Implementer la méthode de Fadeev et la tester sur une matrice compagnon.

Exercice 7 (Méthode de Newton pour approcher les racines d'un polynôme)**.** a) Justifier que si P est un polynôme unitaire $P = X^n + a_1X^{n-1} + \cdots + a_n$ et si $z \in \mathbb{C}$ est une racine de P alors :

$$|z| \leq \max(1, \sum_{i=1}^{n} |a_i|).$$

b) Ecrire un fonction `Newton` qui prend en paramètre un polynôme unitaire P donné par sa liste de coefficients et cherche sa plus grande racine réelle à l'aide de la méthode de Newton .

c) Tester la rapidité de convergence de cette méthode sur $P = (X-1)(X-2)^3$ et sur $Q = (X-1)(X-2)$.

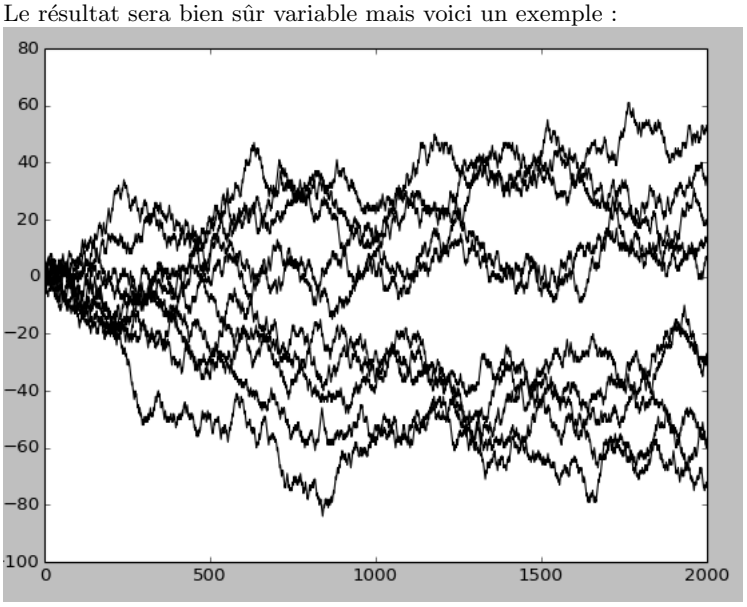
d) Ecrire une fonction Python qui transforme un polynôme Python P (donné par sa liste de coefficients) et un polynôme P_1 (donné par sa list de coeff.) ayant les mêmes racines mais sans multiplicités. On pourra utiliser les fonctions de la bibliothèque polynôme.

Solution 2 a)

```
import numpy as np
def marchealea(N):
    #renvoie un tableau des valeurs S[i]
    S=np.zeros(N)
    for i in range(1,N):
        S[i]=S[i-1]+2*np.random.randint(0,2)-1
    return S

import matplotlib.pyplot as plt
plt.clf()
for i in range(10):
    L=marchealea(2000)
    plt.plot(L,'k')# par défaut la valeur en absicce sera n donc ici les points seront (n,S[n]).
plt.show()
```

Le résultat sera bien sûr variable mais voici un exemple :



b)

```
def TempsPremierRetour(N=101):
    # par soucis d'optimisation
    # on n'utilise pas marchealea
    # pour s'arrêter dès le premier retour à l'origine
    S=0
    for i in range(N):
        S=S+2*np.random.randint(0,2)-1
        if S==0 :
            return i
    return N+50
```

c)

```
L=[]
NbEssai=5000
for i in range(NbEssai):
    L.append(TempsPremierRetour())
plt.clf()
plt.hist(L,bins=50)
```

Avec le résultat :

Remarque : a priori 50 bâtons ce serait très bien pour diviser l'intervalle de 1 à 100 puisque les valeurs impaires ne comptent pas. En réalité, à cause de la valeur 150 prise pour le cas où il n'y a pas de retour avant 100, les 50 bâtons fabriquent des intervalles strictement plus grand que 2. C'est ce qui explique que le premier bâtons est bien au-dessus de la valeur moitié. Avec `bins=100` par exemple, on rectifie cela.

d)

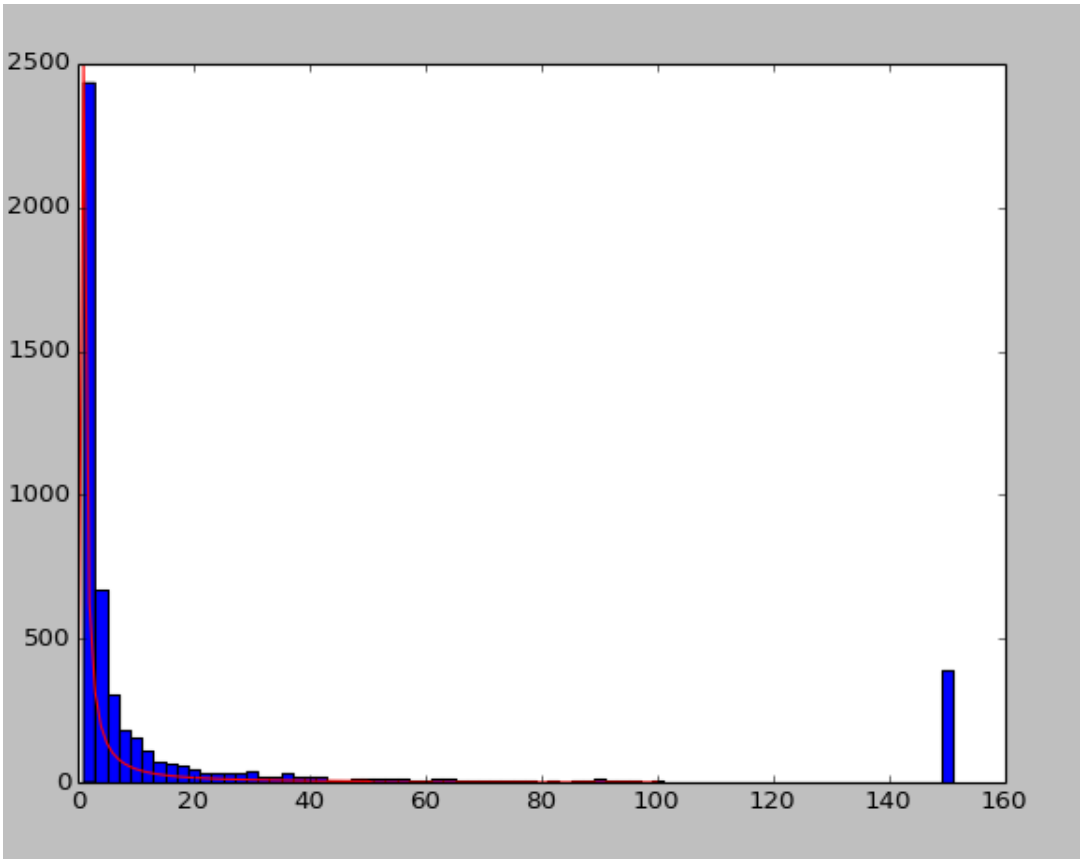
```
def factorial(n):
    p=1
    for i in range(1,n+1):
        p=p*i
    return p

def binomial(n,p):
    return factorial(n)//(factorial(p)*factorial(n-p))

#from scipy.misc import comb # si on voulait une fonction faisant les binomiaux  mieux que cela !

Tabtheorique=np.zeros(101)
for i in range(1,51):
    Tabtheorique[i]=NbEssai*(1/(2**(2*i-1)))*(1/(2*i-1))*binomial(2*i-1,i)
plt.plot(Tabtheorique,color="red") # on trace la courbe théorique en rouge.
plt.show()
```

Exemple de rendu ave un `bins=75` pour le diagramme en bâton



Solution 4 a) On va utiliser l'expression combinatoire du déterminant.

$$\det(M) = \sum_{\sigma \epsilon S_n} \varepsilon(\sigma) m_{\sigma(1),1} \dots m_{\sigma(n),n},$$

autrement dit :

$$D = \sum_{\sigma \in S_n} \varepsilon(\sigma) X_{\sigma(1),1} \dots X_{\sigma(n),n}.$$

Par linéarité de l'espérance et indépendance des v.a. $X_{i,j}$ (qui permet de dire que l'espérance du produit est le produit des espérances)

$$E(D) = \sum_{\sigma \in S_n} \varepsilon(\sigma) E(X_{\sigma(1),1}) \dots E(X_{\sigma(n),n}).$$

Mais chacune des v.a. $X_{i,j}$ est d'espérance nulle donc $\boxed{E(D) = 0}$.

Pour la variance : $V(D) = E(D^2) - E(D)^2 = E(D^2)$. Or

$$\begin{aligned} D^2 &= \left(\sum_{\sigma \in S_n} \varepsilon(\sigma) X_{\sigma(1),1} \dots X_{\sigma(n),n} \right) . \left(\sum_{\tau \in S_n} \varepsilon(\tau) X_{\tau(1),1} \dots X_{\tau(n),n} \right), \\ &= \sum_{(\sigma,\tau) \in S_n^2} \varepsilon(\sigma)\varepsilon(\tau) X_{\sigma(1),1} \dots X_{\sigma(n),n} X_{\tau(1),1} \dots X_{\tau(n),n} \end{aligned}$$

Par linéarité de l'espérance

$$\begin{aligned} E(D^2) &= \sum_{(\sigma,\tau) \in S_n^2} \varepsilon(\sigma)\varepsilon(\tau) E(X_{\sigma(1),1} \dots X_{\sigma(n),n} X_{\tau(1),1} \dots X_{\tau(n),n}), \\ &= \underbrace{\sum_{\sigma \neq \tau} \varepsilon(\sigma)\varepsilon(\tau) E(X_{\sigma(1),1} \dots X_{\sigma(n),n} X_{\tau(1),1} \dots X_{\tau(n),n})}_{A} + \underbrace{\sum_{\sigma \in S_n} E(X^2_{\sigma(1),1} \dots X^2_{\sigma(n),n})}_{B} \end{aligned}$$

Or pour tout (i,j), $X_{i,j}^2 = 1$, donc B est une somme de 1 et donc $B = n!$.

Etudions le terme A.

Dans le produit $X_{\sigma(1),1} \dots X_{\sigma(n),n} X_{\tau(1),1} \dots X_{\tau(n),n})$, si pour un indice $i \in [\![1,n]\!]$ donné, $\sigma(i) = \tau(i)$ alors les deux facteurs $X_{\sigma(i),i} X_{\tau(i),i}$ donnent $X^2_{\sigma(i),i}$ qui est constant égal à 1 et donc se simplifie.

Donc ce produit se simplifie un peu en : $X_{\sigma(1),1} \dots X_{\sigma(n),n} X_{\tau(1),1} \dots X_{\tau(n),n}) = \prod_{i \in [\![1,n]\!], \sigma(i) \neq \tau(i)} X_{\sigma(i),i} X_{\tau(i),i}$.

Mais alors (tous les termes étant d'indice distincts), les v.a. qui apparaissent dans ce produit sont mutuellement indépendantes et donc l'espérance de ce produit est le produit des espérances.

Donc $A = \sum_{\sigma \neq \tau} \varepsilon(\sigma)\varepsilon(\tau) \prod_{i \in [\![1,n]\!], \sigma(i) \neq \tau(i)} E(X_{\sigma(i),i}) E(X_{\tau(i),i})$.

Or comme chaque v.a. $X_{i,j}$ est d'espérance nulle, on conclut que $A = 0$ et $\boxed{V(D) = n!}$.

b)

```
import numpy as np
def test(n):
    L=np.random.randint(0,2,(n,n)) # matrice aléatoire
    # de taille n*n avec entier entre 0 (large)  et 2 (strict) donc 0 ou 1.
    M=2*L-1 # pour que les entrées vaillent 1 et -1
    return np.linalg.det(M)

n=5# la taille
N=10000 # le nombre de tests
#  calcul de moyenne la valeur du déterminant
# qui doit tendre vers l'espérance.
moyenne=0
for i in range(N):
    moyenne+=test(n)
print("espérance : ",moyenne/N)

# pour la variance E(X^2)-E(X)^2.
# en sachant E(X)=0 on calcule la moyenne de X^2
# donc :
moyenne=0
for i in range(N):
    moyenne+=test(n)**2
print("variance : ",moyenne/N)
```

c)

```
# on crée la liste de 2^{n^2} matrices possibles  toute équiprobables. Puis on calcule
#la moyenne du  det la dessus : joli mais pas réaliste !
import copy
def creeUnivers(n):
    """crée la liste des 2**(n**2) matrices de tailles n**2 possibles avec entrées 1 ou -1"""
     # on commence par créer les matrices en lignes
     # on va gérer une liste de listes (liste de matrices lignes)
    L=[[]] # ini
     # sous fonction :
    def rajoute(L,a):
         #""" rajoute a à chaque liste figurant dans  la liste de listes L"""
        for sousL in L:
            sousL.append(a)
        return L
    for i in range(n**2): #(le nombre d'entrées)
        L2=copy.deepcopy(L)
        L=rajoute(L,1)+rajoute(L2,-1)
    # reste à les remettre en matrice
    M=[]
    for a in L:
        M.append(np.array(a).reshape((n,n)))
    return M


def esperanceVarianceMath(n):
    L=creeUnivers(n)
    TabDet=[np.linalg.det(a) for a in L]
    TabDetCarre=[d**2 for d in TabDet]
    esperance=sum(TabDet)/len(L) # proba unif.
    Variance=sum(TabDetCarre)/len(L) # Esperance du carré puisque E(D)=0.
    return esperance,Variance
```

Solution 5

```
import numpy as np
def normecarre(X):
    "renvoie la norme euclidienne au carre d'un vecteur X"
    return (np.vdot(X,X))
# vdot calcule le p.s.
def GS1(X):
    """ le programme fonctionne avec X une np.matrix"""
    m,n=np.shape(X) # n est le nombre de colonnes
    A=X[:,0] # ceci renvoie la premiere colonne de X
    # sous forme d'une matrice colonne pourvu que X soit
    # une np.matrix et pas un np.array
    for i in range(1,n):
        # à l'etape i on calcule la colonne i orthogonalisée
        C=X[:,i]# on va orthogonaliser le i ieme vecteur.
        for k in range(i):
            mu=-np.vdot(np.array(X[:,i]),np.array(A[:,k]))/normecarre(np.array(A[:,k]))
            C=C+mu*A[:,k]
        A=np.concatenate((A,C),axis=1)
    return A
```

b) Valeur retournée :

```
[[ 1.          1.33333333  1.        ]
 [ 1.          0.33333333 -1.5       ]
 [-1.          1.66666667 -0.5       ]]
```

c)

```
def GS2(X):
    Y=GS1(X)
    print(Y)
    m,n=np.shape(X)
    A=Y[:,0]/np.sqrt(normecarre(np.array(Y[:,0])))
    for i in range(1,n):
        C=Y[:,i]/np.sqrt(normecarre(np.array(Y[:,i])))
        A=np.concatenate((A,C),axis=1)
    return A
```

Remarque : en fait il est plus efficace de faire l'orthonormalisation directement : comme dans le code ci-dessous :

```
def GS(X):
    Y=np.transpose(X)
    A=np.zeros((len(Y),len(Y[0])))
    A[0]=Y[0]
    A[0]=A[0]/np.sqrt(np.vdot(A[0],A[0]))
    for i in range(1,len(Y)):
        S=np.zeros(len(Y))
        for k in range(i):
            S=S+np.vdot(Y[i],A[k])*A[k]
        A[i]=(Y[i]-S)
        A[i]=A[i]/np.sqrt(np.vdot(A[i],A[i]))
    return np.transpose(A)
```

Solution 6 Soit $A \in M_n(\mathbb{K})$, on note $\chi_A(X) = \det(XI_n - A)$ son polynôme caractéristique. Soit B la transposée de la comatrice de $XI_n - A$, on sait que :

$$B(XI_n - A) = (XI_n - A)B = \chi_A(X)I_n \quad (\dagger)$$

égalité dans $M_n(\mathbb{K}[X])$. (Le théorème vu en cours est vrai dans $M_n(L)$ où $L = \mathbb{K}(X)$ car il est vrai pour tout corps L).

Or chaque entrée de B est dans $\mathbb{K}_{n-1}[X]$, on peut donc écrire : $B = X^{n-1}B_0 + X^{n-2}B_1 + \cdots + B_{n-1}$ où $(B_0, \ldots, B_{n-1}) \in M_n(\mathbb{K})^n$. On note enfin $\chi_A(X) = X^n + \alpha_1 X^{n-1} + \cdots + \alpha_n$.

Alors $(\dagger)$ s'écrit :

$$(X^{n-1}B_0 + X^{n-2}B_1 + \cdots + B_{n-1})(XI - A) = (X^n + \alpha_1 X^{n-1} + \cdots + \alpha_n)I_n$$

donc en identifiant on a :

$$\begin{cases} B_0 = I_n, \\ B_1 - A = \alpha_1 I_n, \\ B_2 - AB_1 = \alpha_2 I_n, \\ \vdots \\ B_{n-1} - AB_{n-2} = \alpha_{n-1} I_n, \\ -AB_{n-1} = -\alpha_n I_n. \end{cases}$$

Comme il s'agit d'un système triangulaire, on obtient immédiatement :

$$\begin{cases} B_0 = I_n, \\ B_1 = A + \alpha_1 I_n, \\ B_2 = AB_1 + \alpha_2 I_n = A^2 + \alpha_1 A + \alpha_2 I_n \\ \vdots \\ B_{n-1} = AB_{n-2} + \alpha_{n-1} I_n = A^{n-1} + \alpha_1 A^{n-2} + \cdots + \alpha_{n-1} I_{n-1}, \\ -A(A^{n-1} + \alpha_1 A^{n-2} + \cdots + \alpha_{n-1} I_{n-1}) = -\alpha_n I_n \quad (*) \end{cases}$$

On remarque que $(*)$ est exactement Cayley-Hamilton. □

Deuxième pas : trace et dérivée du déterminant

On note $B = (B_{i,j})_{(i,j)\in[[1,n]]^2}$. Bien sûr chaque $B_{i,j}$ est un élément de $\mathbb{K}_{n-1}[X]$.

On sait que $\chi'_A(x) \overset{def}{=} \dfrac{d}{dX}(\det(XI_n - A)) \overset{prop}{=} \displaystyle\sum_{i=1}^{n} \det(M_i(X)) \quad (\dagger\dagger)$

où $M_i(X)$ est la matrice obtenue à partir de $M = XI_n - A$ en remplaçant la colonne i par la colonne $E_i = {}^t(0 \ldots 0\, 1\, 0 \ldots 0)$ où le 1 est à la i-ème ligne.

En effet en écrivant $M = (C_1(X), \dots, C_n(X))$, *on sait que* $\dfrac{d}{dx}(\det(M)) = \sum_{i=1}^{n} \det(C_1, \dots, C_i', \dots, C_n)$. *Et ici dans chaque colonne* C_i *seule l'entrée diagonale fait intervenir* X *et dans cette entrée on a* $X - a_{i,i}$ *d'où* $C_i' = E_i$ *et* $(\dagger\dagger)$

Mais alors par déf. de M_i, $\det(M_i)$, développé par rapport à la i-ème colonne, est le cofacteur d'indice (i,i) de $M = XI_n - A$, autrement dit $\det(M_i) = B_{i,i}$.

Donc $(\dagger\dagger)$ devient $\chi_A'(x) = \sum_{i=1}^{n} B_{i,i} = \mathrm{Tr}(B) \quad (**)$.

Remarque – Cette formule peut aussi s'obtenir directement avec la formule sur la différentielle du déterminant avec justement la trace et la transposée de la comatrice.

On déduit de $(**)$ que :

$$nX^{n-1} + (n-1)\alpha_1 X^{n-2} + \cdots + \alpha_{n-1} = X^{n-1}\,\mathrm{Tr}(B_0) + X^{n-2}\,\mathrm{Tr}(B_1) + \cdots + \mathrm{Tr}(B_{n-1}).$$

Ceci fournit par identification (on rappelle que les α_i sont les coeff. du poly. caract.) :

$$\mathrm{Tr}(B_0) = n,\ \mathrm{Tr}(B_1) = (n-1)\alpha_1,\ \mathrm{Tr}(B_2) = (n-2)\alpha_2,\ \mathrm{Tr}(B_{n-1}) = \alpha_{n-1}$$

En utilisant le dernier système obtenu au a) on sait que :

$\mathrm{Tr}(B_1) = \mathrm{Tr}(A) + \alpha_1\,\mathrm{Tr}(I_n)$, donc avec ce qui précède, on obtient : $\mathrm{Tr}(A) = -\alpha_1$ (ce qu'on sait !), mais ensuite :

$\mathrm{Tr}(B_2) = \mathrm{Tr}(A^2) + \alpha_1\,\mathrm{Tr}(A) + \alpha_2 n$ et $\mathrm{Tr}(B_2) = (n-2)\alpha_2$ donne :

$2\alpha_2 + \alpha_1\,\mathrm{Tr}(A) + \mathrm{Tr}(A^2) = 0$ et à un ordre quelconque on obtient :

Formules de Fadeev : Pour tout $i \in [\![1, n]\!]$, $\mathrm{Tr}(A^i) + \alpha_1\,\mathrm{Tr}(A^{i-1}) + \cdots + \alpha_{i-1}\,\mathrm{Tr}(A) + i\alpha_i = 0$.

Scholie — Ces formules permettent de calculer successivement les coeff. α_i du polynôme caract. à partir des traces des puissances de A. (Elles sont en fait équivalentes aux formules de Newtons via le lien coefficients et racines en se plongeant dans un corps algébriquement clos, mais la démonstration est plus naturelle ici via l'algèbre linéaire !).

Mise en oeuvre pratique – Si on pose $M_0 = A$ et pour tout $k \geq 1$, $M_k = A(M_{k-1} - \dfrac{1}{k}\,\mathrm{Tr}(M_{k-1})I_n)$, on a $\mathrm{Tr}(M_k) = \alpha_{k+1}$ ce qui donne :

$$\chi_A(X) = X^n - \sum_{k=0}^{n-1} \frac{1}{k+1}\,\mathrm{Tr}(M_k)X^{n-k-1}.$$

c) A vous de jouer !

Solution 7 a) On écrit $z^n = -a_1 z^{n-1} - \cdots - a_n$. Donc $|z|^n \leq \sum_{i=1}^{n} |a_i||z|^{n-i}$. Ou bien $|z| \leq 1$ ou bien $|z| > 1$ et dans ce cas tous les $|z|^{n-i}$ du membre de droite sont inférieur à $|z|^{n-1}$.

Donc $|z|^n \leq |z|^{n-1}(|a_1| + \cdots + |a_n|)$, d'où laconclusion en divisant par $|z|^{n-1}$.

b) c) La présence d'une racine multiple ralentit la méthode de Newton ! Voir le CB d'analyse des sup. de cette année !

Pour enlever les racines multiples, on peut considérer $P/(P \wedge P')$.