Informatique
Préparation à l'oral.

Algèbre et arithmétique - Corrigés

2017

1. D'après Centrale (PSI) ♠♠

(Planche Officiel de la Taupe Gyroscope 2017 N°164)

1. Puisque A est de degré au plus $q-1$ et que P est de degré p, le produit AP est de degré au plus $p+q-1$. Il en va de même du polynôme BQ et donc également du polynôme $AP+BQ$. L'application linéaire u est donc à valeur dans $\mathbb{C}_{p+q-1}[X]$ qui est de dimension $p+q$. Par ailleurs, l'espace vectoriel $\mathbb{C}_{q-1}[X]\times\mathbb{C}_{p-1}[X]$ est de dimension $q+p$. La matrice $M_{P,Q}$ est donc une matrice carrée d'ordre $p+q$ à coefficients complexes.
Pour l'espace vectoriel $\mathbb{C}_{p+q-1}[X]$, on choisit la base canonique $\left(1, X, X^2, ..., X^{p+q-1}\right)$.
Posons alors $P(X)=\sum_{i=0}^{p} p_i X^i$ et $Q(X)=\sum_{i=0}^{q} q_i X^i$.
Pour tout entier nature k dans $[\![0\,;q-1]\!]$, on a :

$$u\left(\left(X^k,0\right)\right)=X^k\times P(X)+0\times Q(X)=X^k\times\sum_{i=0}^{p}p_iX^i=\sum_{i=0}^{p}p_iX^{k+i}$$

Pour fixer les idées (et appréhender plus facilement la structure de la matrice), on a donc :

$$u\left((1,0)\right)=\sum_{i=0}^{p}p_iX^i=P(X)\text{ et }u\left(\left(X^{q-1},0\right)\right)=X^{q-1}\times P(X)=\sum_{i=0}^{p}p_iX^{q-1+i}=\sum_{i=q-1}^{p+q-1}p_{i-q+1}X^i$$

Et pour tout entier k dans $[\![0\,;p-1]\!]$, on a :

$$u\left(\left(0,X^k\right)\right)=0\times P(X)+X^k\times Q(X)=X^k\times\sum_{i=0}^{q}q_iX^i=\sum_{i=0}^{q}q_iX^{k+i}$$

En particulier :

$$u\big((0,1)\big)=\sum_{i=0}^{q}q_i\mathrm{X}^i=\mathrm{Q}(\mathrm{X})\text{ et et }u\Big(\big(0,\mathrm{X}^{p-1}\big)\Big)=\mathrm{X}^{p-1}\times\mathrm{Q}(\mathrm{X})=\sum_{i=0}^{q}q_i\mathrm{X}^{p-1+i}=\sum_{i=p-1}^{p+q-1}p_{i-p+1}\mathrm{X}^i$$

D'où la matrice cherchée :

$$M_{\mathrm{P,Q}}=\left(\begin{array}{ccccc:ccccc}
p_0 & & & & & q_0 & & & & \\
p_1 & p_0 & & (0) & & q_1 & q_0 & & (0) & \\
\vdots & p_1 & \ddots & & & \vdots & q_1 & \ddots & & \\
p_{p-1} & \vdots & \ddots & \ddots & & q_{q-1} & \vdots & \ddots & \ddots & \\
p_p & p_{p-1} & \ddots & \ddots & p_0 & q_q & q_{q-1} & \ddots & \ddots & q_0 \\
 & p_p & \ddots & \ddots & p_1 & & q_q & \ddots & \ddots & q_1 \\
 & & \ddots & \ddots & \vdots & & & \ddots & \ddots & \vdots \\
 & (0) & & \ddots & p_{p-1} & & (0) & & \ddots & q_{q-1} \\
 & & & & p_p & & & & & q_q
\end{array}\right)$$

Remarque : nous avons <u>apparemment</u> placé p_p et q_q sur les mêmes lignes comme si on avait $p=q$. Ce n'est bien sûr pas le cas ! Nous avons juste retenu une représentation simplifiée de la matrice $M_{\mathrm{P,Q}}$.

2. Dans le code ci-dessous, on tire parti du fait que l'on peut mettre à jour une sous-colonne d'un tableau numpy à l'aide d'une liste Python.

```
def MatGen(P,Q):
    # Rappelons que P et Q sont ici les listes des
    # coefficients des polynômes P et Q.
    p = len(P) - 1
    q = len(Q) - 1
    # Matrice carrée nulle d'ordre p+q
    A = np.zeros((p+q,p+q))
    # Mise à jour des q premières colonnes de A
    # (indicées de 0 à q-1)
    for i in range(q):
        A[i:i+p+1,i] = P
    # Mise à jour des p dernières colonnes de A
    # (indicées de q à p+q-1)
    for i in range(p):
        A[i:i+q+1,i+q] = Q
    return(A)
```

3. L'algorithme d'Euclide nous permet d'écrire les divisions euclidiennes successives suivantes :

$$X^4+X^3+1=(X+1)(X^3-X+1)+X^2$$
$$X^3-X+1=X\times X^2+(-X+1)$$
$$X^2=(-X-1)(-X+1)+1$$

Remarque : classiquement, dans la dernière division, on aurait pu considérer $X-1$ au lieu de $-X+1$ (de façon à diviser systématiquement pas un polynôme unitaire) mais ça ne change pas grand-chose…

Le dernier reste étant une constante (en l'occurrence 1), on en déduit que les polynômes P et Q sont ici premiers entre eux.

D'après le théorème de Bézout, il existe donc deux polynômes U et V tels que :
$PU+QV=1$.

Pour obtenir deux tels polynômes, on procède classiquement (en remontant) :

$$X^4+X^3+1=(X+1)(X^3-X+1)+X^2 \qquad \times(-X(X+1)+1)$$
$$X^3-X+1=X\times X^2+(-X+1) \qquad \times(X+1)$$
$$X^2=(-X-1)(-X+1)+1$$

On additionne alors les trois égalités membre à membre et on obtient finalement, après simplification :

$$(-X^2-X+1)(X^4+X^3+1)+(X^3+2X^2+X)(X^3-X+1)=1$$

On a bien trouvé deux polynômes A et B de $\mathbb{C}_2[X]\times\mathbb{C}_3[X]$ qui vérifient $AP+BQ=1$.

Pour ce qui est de l'unicité, on peut supposer l'existence d'un autre couple A' et B' vérifiant également $A'P+B'Q=1$.
On aura alors : $(A-A')P+(B-B')Q=0$, soit : $(A-A')P=(B'-B)Q$.
Ainsi, P divise $(B'-B)Q$. Mais comme P et Q sont premiers entre eux, P divise $B'-B$ (théorème de GAUSS). Or, P est de degré 4 tandis que B et B' sont de degrés au plus égaux à 3, leur différence étant donc elle-même de degré au plus 3. On en déduit ainsi que cette différence est nulle : $B=B'$. On en tire alors immédiatement $A=A'$.
On a bien l'unicité.

Rappelons, dans le cas général, la démarche permettant d'établir que si P et Q sont deux polynômes de degrés *p* et *q* respectivement, premiers entre eux alors il existe deux polynômes U et V de degrés au plus égaux à $q-1$ et $p-1$ respectivement tels que $UP+VQ=1$.

Si P et Q sont premiers entre eux dans $\mathbb{C}[X]$, le théorème de Bézout nous garantit l'existence de deux polynômes A et B tels que $AP+BQ=1$. En divisant respectivement A et B par Q et P, on a : $A=A'Q+R_A$, avec $d°R_A<q$, et $B=B'P+R_B$, avec $d°R_B<p$. D'où :

$$AP+BQ=1\Leftrightarrow(A'Q+R_A)P+(B'P+R_B)Q=1\Leftrightarrow(A'+B')PQ+R_AP+R_BQ=1$$

On a : $A'+B'=0$ ou $d°\big((A'+B')PQ\big)\geq p+q$. Or $d°(R_AP)<p+q$ et $d°(R_BQ)<p+q$.

On en tire immédiatement $A'+B'=0$ et $R_AP+R_BQ=1$.

Comme $(R_A,R_B)\in\mathbb{C}_{q-1}[X]\times\mathbb{C}_{p-1}[X]$, on a bien le résultat cherché (existence).

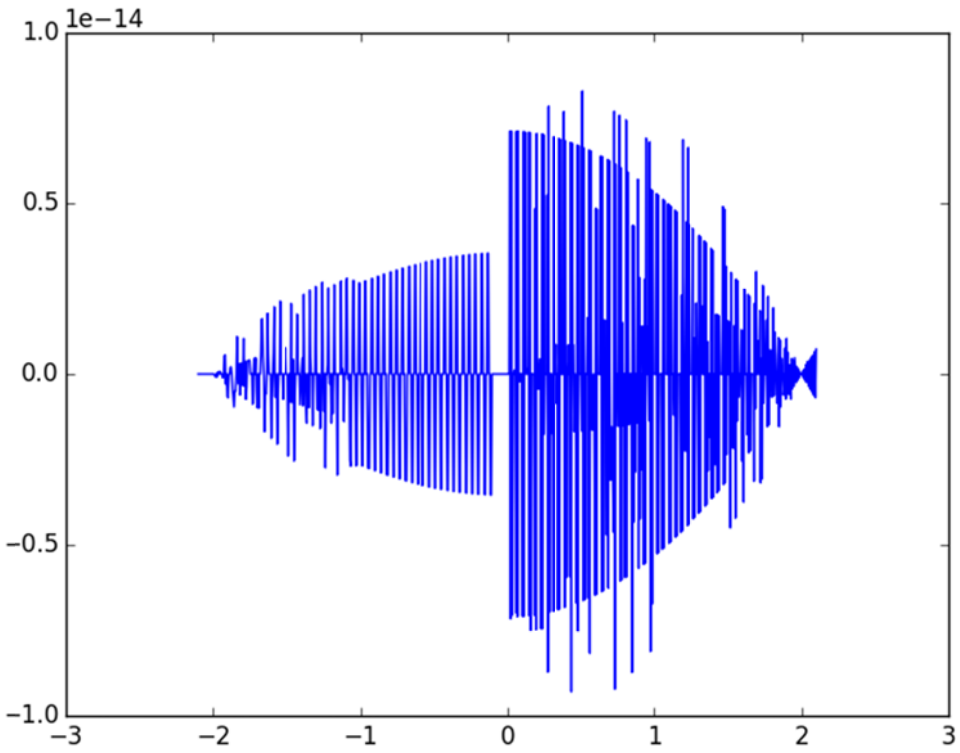
4. On a $d°P=d°Q_a=3$. La matrice M_{P,Q_a} sera donc une matricé carrée d'ordre 6.
On a $P(X)=X^3-X^2-4X+4$ et, pour tout réel a : $Q_a(X)=X^3-(a+1)X^2+aX$.
D'où :

$$M_{P,Q_a}=\begin{pmatrix}4&0&0&0&0&0\\-4&4&0&a&0&0\\-1&-4&4&-(a+1)&a&0\\1&-1&-4&1&-(a+1)&a\\0&1&-1&0&1&-(a+1)\\0&0&1&0&0&1\end{pmatrix}$$

a) Pour le calcul du déterminant, nous utilisons bien sûr la fonction `det` du module `numpy.linalg` (penser à l'importer !). On peut alors considérer le script suivant :

```
pas = 0.005
a, b = -2.1, 2.1
T = np.linspace(a,b,int((b-a)/pas)+1)
P = [4,-4,-1,1]
# La liste D sert à stocker les déterminants
# des matrices M(P,Qt).
D = []
for t in T:
# La liste Q des coefficients du polynôme Qt change
# à chaque itération.
    Q = [0,t,-(t+1),1]
    D.append(det(MatGen(P,Q)))
# Le graphique
plt.clf()
plt.plot(T,D)
plt.show()
```

On obtient alors le graphique suivant :



b) Le graphique précédent nous indique que toutes les valeurs calculées du déterminant sont inférieures à 10^{-14} en valeur absolue. On peut raisonnablement conjecturer que le déterminant de $M_{\mathrm{P},\mathrm{Q}_a}$ est en fait nul (les petites fluctuations résultant d'erreurs de calcul liées à des problèmes d'arrondi).

c) On a :

$$\det\left(M_{\mathrm{P},\mathrm{Q}_a}\right)=\begin{vmatrix} 4 & 0 & 0 & 0 & 0 & 0 \\ -4 & 4 & 0 & a & 0 & 0 \\ -1 & -4 & 4 & -(a+1) & a & 0 \\ 1 & -1 & -4 & 1 & -(a+1) & a \\ 0 & 1 & -1 & 0 & 1 & -(a+1) \\ 0 & 0 & 1 & 0 & 0 & 1 \end{vmatrix}$$

En développant suivant la première ligne, il vient immédiatement :

$$\det\left(M_{\mathrm{P},\mathrm{Q}_a}\right)=4\times\begin{vmatrix}4&0&a&0&0\\-4&4&-(a+1)&a&0\\-1&-4&1&-(a+1)&a\\1&-1&0&1&-(a+1)\\0&1&0&0&1\end{vmatrix}$$

On ajoute la première ligne à la seconde et la 4ème à la 3ème :

$$\det\left(M_{\mathrm{P},\mathrm{Q}_a}\right)=4\times\begin{vmatrix}4&0&a&0&0\\0&4&-1&a&0\\0&-5&1&-a&-1\\1&-1&0&1&-(a+1)\\0&1&0&0&1\end{vmatrix}$$

On ajouté à la première ligne -4 fois la 4ème :

$$\det\left(M_{\mathrm{P},\mathrm{Q}_a}\right)=4\times\begin{vmatrix}0&4&a&-4&4(a+1)\\0&4&-1&a&0\\0&-5&1&-a&-1\\1&-1&0&1&-(a+1)\\0&1&0&0&1\end{vmatrix}$$

On développe suivant la première colonne :

$$\det\left(M_{\mathrm{P},\mathrm{Q}_a}\right)=-4\times\begin{vmatrix}4&a&-4&4(a+1)\\4&-1&a&0\\-5&1&-a&-1\\1&0&0&1\end{vmatrix}$$

On retranche la première colonne à la dernière :

$$\det\left(M_{\mathrm{P},\mathrm{Q}_a}\right)=-4\times\begin{vmatrix}4&a&-4&4a\\4&-1&a&-4\\-5&1&-a&4\\1&0&0&0\end{vmatrix}=-16\times\begin{vmatrix}4&a&-4&a\\4&-1&a&-1\\-5&1&-a&1\\1&0&0&0\end{vmatrix}$$

On développe suivant la dernière ligne :

$$\det\left(M_{\mathrm{P},\mathrm{Q}_a}\right)=16\times\begin{vmatrix}a&-4&a\\-1&a&-1\\1&-a&1\end{vmatrix}$$

La deuxième et la troisième lignes sont opposées.

On en conclut immédiatement que le déterminant est nul pour toute valeur du réel a.

$$\forall a \in \mathbb{R}, \det\left(M_{P,Q_a}\right) = 0$$

2. ENSAM 2017 (PSI) ♠

(Planche Officiel de la Taupe Gyroscope 2017 N°242 – Exercice I)

1. Précisons d'abord que nous ne traitons que les entiers supérieurs ou égaux à 2.
Ensuite, rappelons que dès que nous avons un diviseur d de n, nous en avons un deuxième : $q = \frac{n}{d}$ sauf, bien sûr, si $q = d$.
La deuxième remarque nous conduit classiquement à tester tous les entiers compris entre 2 et $\mathrm{E}\left(\sqrt{n}\right)$.
Le principe général de notre fonction est alors le suivant : si nous trouvons un diviseur d de n dans $[\![2\,;\mathrm{E}\left(\sqrt{n}\right)]\!]$, nous l'ajoutons à une liste `Linf` et si $q = \frac{n}{d}$ est différent de d, nous l'ajoutons à une liste `Lsup`.
A la fin de la boucle, la liste Linf est une liste d'entiers inférieurs ou égaux à $\mathrm{E}\left(\sqrt{n}\right)$ et croissante tandis que Lsup est une liste d'entiers strictement supérieurs à $\mathrm{E}\left(\sqrt{n}\right)$ et décroissante. On inverse alors Lsup et, enfin, on renvoie la liste obtenue en concaténant Linf et Lsup :

```
def DS(n):
    if n <= 1:
        raise ValueError("L'entier doit être supérieur ou égal à 2 !")
    else:
        # On considère tous les entiers inférieurs ou égaux à la partie
        # entière de la racine carrée de n.
        # Chaque fois que l'on obtient un diviseur (on le place dans
        # Linf), on en obtient un deuxième (que l'on place dans Lsup).
        Linf, Lsup = [1], []
        dmax = floor(sqrt(n))
        for d in range(2,dmax+1):
            (q,r) = divmod(n,d)
            if r == 0:
                Linf.append(d)
                if d != q:
                    Lsup.append(q)
        Lsup.reverse()
        return(Linf+Lsup)
```

L'appel `DS(100)` dans la console donne `[1,2,4,5,10,20,25,50]`.

2. Cette fonction est simple puisque, lorsque l'entier n est supérieur ou égal à 2, il suffit de renvoyer la somme des éléments de la liste obtenue grâce à la question précédente.
On peut donc considérer le code suivant :

```
def SDS(n):
    if n <= 1:
        raise ValueError("L'entier doit être supérieur ou égal à 2 !")
    else:
        return(sum(DS(n)))
```

L'appel `SDS(100)` dans la console donne `117`.

3. Ici encore, pas de difficultés particulières. On a une boucle principale balayant les entiers de 2 à n et on teste si un tel entier est égale à la somme de ses diviseurs stricts :

```
def Parfaits(n):
    if n <= 1:
        raise ValueError("L'entier doit être supérieur ou égal à 2 !")
    else:
        LP = []
        for k in range(2,n+1):
            if k == SDS(k):
                LP.append(k)
                print(k,"est parfait !")
        return(LP)
```

L'appel `Parfaits(500)` donne les quatre affichages :

```
6 est parfait !
28 est parfait !
496 est parfait !
[6,28,496]
```

4. Notons que *x* et *y* jouent des rôles symétriques. Nous allons donc nous contenter de déterminer les couples amicaux (x, y) avec $x < y$.

```
def Amicaux(n):
    if n <= 1:
        raise ValueError("L'entier doit être supérieur ou égal à 2 !")
    else:
        Lfriends = []
        for x in range(2,n+1):
            for y in range(x+1,n+1):
                if SDS(x) == y and SDS(y) == x:
                    print("Les nombres",x,"et",y,"sont amicaux !")
                    Lfriends.append((x,y))
        return(Lfriends)
```

L'appel `Amicaux(1500)` donne alors : `[(220,284),(1184,1210)]`.

3. ENSAM 2017 (PSI) ♠

(Planche Officiel de la Taupe Gyroscope 2017 N°241 – Exercice I)

1. Dans cette question, on va bien sûr comparer la matrice passée en argument (sous la forme d'un tableau `numpy`) à sa transposée obtenue grâce à la fonction `transpose`.

   Rappelons cependant que lorsque l'on teste une égalité entre deux tableaux `numpy` via une syntaxe comme `M == N`, le résultat est encore un tableau `numpy` `L` de même dimension que les tableaux `M` et `N` mais contenant cette fois des booléens tels que : `L[i,j]` prend la valeur logique du test `M[i,j] == N[i,j]`. Il y aura donc égalité des deux tableaux si TOUS les éléments de `L` valent `True`. Pour tester si tous les éléments d'un tableau numpy (ou d'une liste) ne contenant que des booléens valent `True`, on utilise la méthode `all()`.

   En définitive, on peut considérer la fonction suivante :

```
def sym(M):
    return (M == np.transpose(M)).all()
```

2. On a la décomposition classique : $A=\frac{1}{2}\left(M-M^t\right)$ et $S=\frac{1}{2}\left(M+M^t\right)$.

   D'où la fonction demandée :

```
def decomp(M):
    MT = np.transpose(M)
    return(0.5 * (M - MT),0.5 * (M + MT))
```

3. Une matrice $M\in\mathcal{M}_n(\mathbb{R})$ est orthogonale si, et seulement si, elle est inversible d'inverse sa transposée. On peut donc tester la nullité de MM^t-I_n. Plutôt qu'effectuer n^2 comparaison, nous préférons tester la nullité de $\left\|MM^t-I_n\right\|$ en choisissant la norme matricielle euclidienne : $\|A\|=\sqrt{\mathrm{Tr}\left(AA^t\right)}$. Concrètement, on teste donc la nullité de $\left\|MM^t-I_n\right\|^2=\mathrm{Tr}\left(\left(MM^t-I_n\right)\left(MM^t-I_n\right)^t\right)=\mathrm{Tr}\left(\left(MM^t-I_n\right)^2\right)$.

   D'où le code :

```
def ortho(M):
D = np.dot(M,np.transpose(M)) - np.identity(M.shape[0])
return((np.dot(D,D)).trace() < 10**-8)
```

2016

1. D'après Centrale (PSI) ♠♠

(RMS 2016 – Planche N°1002)

a) Pour justifier l'existence de $S(P)$, il suffit de montrer que $S(P)$ existe pour tout polynôme *P* de la base canonique de $\mathbb{K}[X]$, c'est-à-dire tout monôme de la forme X^n où *n* est un entier naturel.

Pour tout entier naturel *n*, on s'intéresse donc à la série : $S(X^n)=\sum_{k=0}^{+\infty}\frac{k^n}{k!}$.

Nous avons affaire à une série dont le terme général $u_k=\frac{k^n}{k!}$ est positif dont la forme nous conduit à utiliser la règle de D'Alembert :

$$\frac{u_{k+1}}{u_k}=\frac{\frac{(k+1)^n}{(k+1)!}}{\frac{k^n}{k!}}=\frac{(k+1)^n}{(k+1)!}\times\frac{k!}{k^n}=\frac{1}{k+1}\times\left(\frac{k+1}{k}\right)^n=\frac{1}{k+1}\times\left(1+\frac{1}{k}\right)^n$$

On a immédiatement : $\lim_{k\to+\infty}\frac{1}{k+1}=0$ et $\lim_{k\to+\infty}\left(1+\frac{1}{k}\right)^n=1$.

Finalement $\lim_{k\to+\infty}\frac{u_{k+1}}{u_k}=0<1$ et on en déduit immédiatement que la série converge.

Ainsi, pour un polynôme $P(X)=\sum_{k=0}^{n}a_kX^k$, $S(P)$ existe et on a :

$$S(P)=\sum_{k=0}^{+\infty}\frac{P(k)}{k!}=\sum_{k=0}^{+\infty}\frac{\sum_{k=0}^{n}a_kk^k}{k!}=\sum_{k=0}^{n}\left(a_k\sum_{k=0}^{+\infty}\frac{k^n}{k!}\right)$$

Pour montrer que *S* est une forme linéaire, on a immédiatement :

$$\forall(P,Q)\in\mathbb{K}[X]^2,\forall(\alpha,\beta)\in\mathbb{K}^2$$

$$\begin{aligned}S(\alpha P+\beta Q)&=\sum_{k=0}^{+\infty}\frac{(\alpha P+\beta Q)(k)}{k!}=\sum_{k=0}^{+\infty}\frac{\alpha P(k)+\beta Q(k)}{k!}\\&=\alpha\sum_{k=0}^{+\infty}\frac{P(k)}{k!}+\beta\sum_{k=0}^{+\infty}\frac{Q(k)}{k!}\\&=\alpha S(P)+\beta S(Q)\end{aligned}$$

L'application S est bien une forme linéaire.

b) On utilise la fonction `factorial` du module `math`.

Pour calculer $S\left(X^d\right)$ on peut utiliser la fonction `Smonome` suivante :

```
def Smonome(d):
    S = 0
    for k in range(51):
        S += (k**d)/factorial(k)
    return(S)
```

Ensuite, pour calculer $\sum_{k=0}^{50} \frac{k^d}{k!}$ avec $d \in [\![0,10]\!]$, on utilise le script suivant :

```
for d in range(11):
    print("S(X^"+str(d)+") =",Smonome(d))
```

On obtient alors l'affichage :

```
S(X^0) = 2.7182818284590455
S(X^1) = 2.7182818284590455
S(X^2) = 5.43656365691809
S(X^3) = 13.591409142295227
S(X^4) = 40.77422742688568
S(X^5) = 141.35065507987028
S(X^6) = 551.8112111771861
S(X^7) = 2383.9331635585822
S(X^8) = 11253.686769820446
S(X^9) = 57483.50582642343
S(X^10) = 315252.73505553784
```

Nous choisissons le polynôme de degré 9 suivant :

$$X^9 + 10X^8 + \frac{1}{2}X^7 + 2X^6 + 5X^5 + 3X^2 - 2X + 1$$

Il est représenté dans le script par le tuple : `P = (1,-2,3,0,0,5,2,0.5,10,7)`.

On calcule alors $S(P)$:

```
S = 0
for k in range(10):
    S += P[k]*Smonome(k)
print("\n"+str(S))
```

Et on obtient : `517937.3421718438`.

c) Le polynôme H_0 est de degré 0 et la relation de récurrence fournie nous donne immédiatement $\forall k \in [\![0\,;n]\!],\ d^\circ H_k = k$. Ainsi, la famille $(H_k)_{0\le k\le n}$ est étagée en degré. C'est donc une base de $\mathbb{R}_n[X]$.

Notons également que l'on a facilement :

$$\begin{aligned}
&H_0(X) = 1\\
&H_1(X) = X\\
&H_2(X) = X(X-1)\\
&H_3(X) = X(X-1)(X-2)\\
&\ldots\\
&H_n(X) = X(X-1)(X-2)\ldots(X-(n-1))
\end{aligned}$$

d) Pour tout entier naturel n, on a : $H_n(X) = X(X-1)(X-2)\ldots(X-(n-1))$.
D'où :

$$\begin{aligned}
S(H_n) &= \sum_{k=0}^{+\infty} \frac{k(k-1)(k-2)\ldots(k-(n-1))}{k!}\\
&= \sum_{k=n}^{+\infty} \frac{k(k-1)(k-2)\ldots(k-(n-1))}{k!}\\
&= \sum_{k=n}^{+\infty} \frac{1}{(k-n)!}\\
&= \sum_{k=0}^{+\infty} \frac{1}{k!}\\
&= e
\end{aligned}$$

$$\forall n \in \mathbb{N},\ S(H_n) = e$$

Soit *P* un polynôme de degré *n*.

On a vu que $(H_k)_{0\le k\le n}$ était une base de $\mathbb{R}_n[X]$. On peut donc écrire : $P(X) = \sum_{k=0}^{n} \alpha_k H_k$.

Comme *S* est linéaire, il vient alors :

$$S(P) = S\left(\sum_{k=0}^{n} \alpha_k H_k\right) = \sum_{k=0}^{n} \alpha_k S(H_k) = \sum_{k=0}^{n} (\alpha_k e) = \left(\sum_{k=0}^{n} \alpha_k\right) e$$

Ainsi :

Pour calculer $S(P)$, il suffit de multiplier *e*
par la somme des coordonnées de *P* dans la base $(H_k)_{0\le k\le n}$.

e) Au regard de la définition des H_n, on va écrire une fonction récursive.
Supposons que les coefficients de H_n soient stockés dans la liste $L^{(n)} = \left[l_0^{(n)}, l_1^{(n)}, l_2^{(n)}, ..., l_{n-1}^{(n)}, l_n^{(n)} \right] = \left[l_0^{(n)}, l_1^{(n)}, l_2^{(n)}, ..., l_{n-1}^{(n)}, 1 \right]$ (les polynômes H_n sont tous unitaires). La relation de récurrence $H_{n+1} = (X - n) H_n$ nous donne alors immédiatement :

$$\forall 1 \leq k \leq n, l_k^{(n+1)} = -n l_k^{(n)} + l_{k-1}^{(n)} \text{ et } l_0^{(n+1)} = 0$$

On peut alors considérer le code suivant :

```
def CoeffsHn(n,L=[0,1]):
    if n == 0:
        return [1]
    elif n == 1:
        return L
    else:
        m = len(L)
        M = [0]
        for k in range(1,m):
            M.append(-(m-1) * L[k] + L[k-1])
        M.append(1)
    return CoeffsHn(n-1,M)
```

Remarque : initialiser le second argument (à `[0,1]`) dans la définition de la fonction `CoeffHn` permet d'effectuer des appels tels `CoeffHn(12)` ou `CoeffHn(78)`…

f) Grâce à la fonction précédente, nous allons pouvoir construire une grande liste dont nous ferons un tableau (`array`) `numpy` grâce à la méthode `reshape` qu'il conviendra bien sûr de transposer. La seule difficulté provient du fait que la fonction CoeffHn renvoie des listes dont la longueur dépend du degré souhaité. Il convient donc, la matrice cherchée ici étant carrée d'ordre $n+1$, de compléter éventuellement par des 0.

On peut donc considérer le code :

```
def MatPassageHn(n):
    Passage = []
    for k in range(0,n+1):
        Passage += CoeffsHn(k)+(n-k)*[0]
    return(((np.array(Passage)).reshape(n+1,n+1)).transpose())
```

g) Soit *X* la matrice colonne des coefficients d'un polynôme de degré *n* dans la base canonique de $\mathbb{R}_n[X]$ et X' la matrice colonne de ses coefficients dans la base $(H_k)_{0 \leq k \leq n}$.
En notant *P* la matrice de passage de la base canonique de $\mathbb{R}_n[X]$ à $(H_k)_{0 \leq k \leq n}$, on a classiquement : $X = PX'$. Soit : $X' = P^{-1}X$.
La fonction `MatPassageHn` nous permet d'obtenir *P*. que l'on inversera grâce à la méthode `inv`. D'où le code :

```
def SPolyEff(X):
   return(sum(np.dot(np.linalg.inv(MatPassageHn(len(X)-1)),np.array(X).transpose()))*e)
```

2. Centrale (PSI) ♠♠

A venir…

3. Centrale (PSI) ♠♠

(Planche RMS 2016-2017 N°1014)

a) On peut utiliser la bibliothèque `numpy` et construire la matrice M_n via une fonction `MatM_Create` que l'on appelle ensuite dans une boucle `for` : $(2 \leq n \leq 10)$:

```
import numpy as np

def MatM_Create(n):
     M = np.ones((n,n))
     for i in range(1,n):
          for j in range(1,n):
               M[i,j] = min(i,j) + 1
     return(M)

for n in range(2,11):
     print("\nn = ",n)
     print(MatM_Create(n))
```

b) On peut utiliser la fonction `det` du module `linalg` de la bibliothèque `numpy` pour obtenir le déterminant cherché. On peut alors remplacer la boucle `for` précédente par :

```
for n in range(2,11):
     print("\nn = ",n)
     M = MatM_Create(n)
     print(M)
     print("det(M) = ",np.linalg.det(M))
```

L'exécution du script nous conduit alors à conjecturer :

$$\forall n \in \mathbb{N} \setminus \{0;1\}, \det(M_n) = 1$$

c) On peut cette fois proposer la fonction `MatT_Create` suivante :

```
def MatT_Create(n):
     M = np.ones((n,n))
     for i in range(n):
          for j in range(i):
               T[i,j] = 0
     return(T)
```

d) On peut cette fois calculer le produit ${}^tT_nT_n$ à l'aide des fonctions dot et transpose de `numpy` :

```
for n in range(2,11):
    print("\nn = ",n)
    T = MatT_Create(n)
    print(np.dot(np.transpose(T),T))
```

L'exécution du script nous conduit alors à conjecturer :

$$\forall n \in \mathbb{N} \setminus \{0;1\},\ {}^tT_nT_n = M_n$$

e) Pour tout entier naturel n non nul, on peut considérer le déterminant $\det(M_{n+1})$. En retranchant la première colonne à chacune des autres colonnes, il vient :

$$\det(M_{n+1}) = \begin{vmatrix} 1 & 1 & \cdots & \cdots & 1 \\ 1 & 2 & \cdots & \cdots & 2 \\ \vdots & \vdots & \ddots & & \vdots \\ \vdots & \vdots & & n & n \\ 1 & 2 & \cdots & n & n+1 \end{vmatrix} = \begin{vmatrix} 1 & 0 & 0 & \cdots & \cdots & 0 \\ 1 & 1 & 1 & \cdots & \cdots & 1 \\ 1 & 1 & 2 & \cdots & \cdots & 2 \\ \vdots & \vdots & \vdots & \ddots & & \vdots \\ \vdots & \vdots & \vdots & & n-1 & n-1 \\ 1 & 1 & 2 & \cdots & n-1 & n \end{vmatrix}$$

En développant alors suivant la première ligne, il vient immédiatement :

$$\det(M_{n+1}) = \begin{vmatrix} 1 & 0 & 0 & \cdots & \cdots & 0 \\ 1 & 1 & 1 & \cdots & \cdots & 1 \\ 1 & 1 & 2 & \cdots & \cdots & 2 \\ \vdots & \vdots & \vdots & \ddots & & \vdots \\ \vdots & \vdots & \vdots & & n-1 & n-1 \\ 1 & 1 & 2 & \cdots & n-1 & n \end{vmatrix} = \begin{vmatrix} 1 & 1 & \cdots & \cdots & 1 \\ 1 & 2 & \cdots & \cdots & 2 \\ \vdots & \vdots & \ddots & & \vdots \\ \vdots & \vdots & & n-1 & n-1 \\ 1 & 2 & \cdots & n-1 & n \end{vmatrix} = \det(M_n)$$

On en déduit : $\forall n \in \mathbb{N}^*, \det(M_n) = cte$. Comme $M_1 = (1)$ et $\det(M_1)$, il vient finalement :

$$\forall n \in \mathbb{N}^*, \det(M_n) = 1$$

La première conjecture est ainsi démontrée.

Pour tout *n* entier naturel non nul, posons $T_n = (t_{ij})$ avec $\begin{cases} t_{ij} = 1 & \text{si } j \geq i \\ t_{ij} = 0 & \text{si } i < j \end{cases}$ et ${}^tT_n = (t'_{ij}) = (t_{ji})$.

Il vient alors : ${}^tT_nT_n = A = (a_{ij})$ avec $a_{ij} = \sum_{k=1}^{n} t'_{ik}\, t_{kj} = \sum_{k=1}^{n} t_{ki}t_{kj}$.

On a alors : $t_{ki}t_{kj}=1\Leftrightarrow t_{ki}=t_{kj}=1\Leftrightarrow\begin{cases}k\le i\\k\le j\end{cases}\Leftrightarrow k\le\min(i,j)$.

D'où : $a_{ij}=\sum_{k=1}^{n}t_{ki}t_{kj}=\sum_{k=1}^{\min(i,j)}1=\min(i,j)=m_{ij}$.

On a bien :

$$\forall n\in\mathbb{N}^*,\ {}^tT_nT_n=M_n$$

La deuxième conjecture est ainsi démontrée.

f) D'après le deuxième résultat démontré à la question précédente, la matrice M_n est le produit d'une matrice réelle et de sa transposée. Elle est donc réelle symétrique. On en déduit immédiatement qu'elle est diagonalisable dans $\mathbb{R}$.

Soit alors λ une valeur propre de M_n. Il existe une matrice non nulle *X* de $\mathcal{M}_{n,1}(\mathbb{R})$ telle que $M_nX=\lambda X$, c'est-à-dire : ${}^tT_nT_nX=\lambda X$. D'où : ${}^tX\left({}^tT_nT_nX\right)={}^tX\left(\lambda X\right)$, soit ${}^t\left(T_nX\right)\left(T_nX\right)=\lambda\left({}^tXX\right)$.

Comme les éléments diagonaux de la matrice T_n sont égaux à 1, donc non nuls, elle est inversible. Et comme *X* n'est pas nulle, on en déduit immédiatement que la matrice colonne T_nX n'est pas nulle. On a alors : ${}^t\left(T_nX\right)\left(T_nX\right)>0$ (en confondant classiquement $\mathcal{M}_1(\mathbb{R})$ et $\mathbb{R}$). Enfin, comme ${}^tXX>0$, il vient enfin $\lambda>0$.

Finalement :

$$\mathrm{Sp}(M_n)\subset\mathbb{R}_+^*$$

g) Notons S_n la somme des valeurs propres de la matrice M_n. On a :

$S_n=\mathrm{tr}(M_n)=1+2+3+...+n=\dfrac{n(n+1)}{2}$. Mais on a également, la matrice M_n comportant *n* valeurs propres (en tenant compte de leur multiplicité) : $S_n\le n\times\lambda_n$.

On en tire : $\dfrac{n(n+1)}{2}\le n\times\lambda_n$, c'est-à-dire $\lambda_n=\max\left(\mathrm{Sp}(M_n)\right)\ge\dfrac{n+1}{2}$.

Le résultat est établi.

h) On a : $M_n^{-1}=\left({}^tT_nT_n\right)^{-1}=T_n^{-1}\left({}^tT_n\right)^{-1}=T_n^{-1}\,{}^t\left(T_n^{-1}\right)$.

On peut donc chercher à déterminer T_n^{-1}.

Pour ce faire, on peut s'intéresser au système :

$$T_nX=Y\Leftrightarrow\begin{pmatrix}1&&&\\&\ddots&(1)&\\&(0)&\ddots&\\&&&1\end{pmatrix}\begin{pmatrix}x_1\\x_2\\\vdots\\x_n\end{pmatrix}=\begin{pmatrix}y_1\\y_2\\\vdots\\y_n\end{pmatrix}$$

On a :

$$\begin{pmatrix} 1 & & \\ & \ddots & (1) \\ & (0) & \ddots \\ & & 1 \end{pmatrix}\begin{pmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{pmatrix} = \begin{pmatrix} y_1 \\ y_2 \\ \vdots \\ y_n \end{pmatrix} \Leftrightarrow \begin{cases} x_1 + x_2 + ... + x_n = y_1 \\ x_2 + ... + x_n = y_2 \\ ... \\ x_{n-1} + x_n = y_{n-1} \\ x_n = y_n \end{cases}$$

En effectuant une « remontée », on obtient facilement :

$$\begin{cases} x_1 + x_2 + ... + x_n = y_1 \\ x_2 + ... + x_n = y_2 \\ ... \\ x_{n-1} + x_n = y_{n-1} \\ x_n = y_n \end{cases} \Leftrightarrow \begin{cases} x_1 = y_1 - y_2 \\ ... \\ x_{n-2} = y_{n-2} - y_{n-1} \\ x_{n-1} = y_{n-1} - y_n \\ x_n = y_n \end{cases}$$

D'où :

$$T_n^{-1} = \begin{pmatrix} 1 & -1 & & & \\ & 1 & -1 & (0) & \\ & & 1 & \ddots & \\ & (0) & & \ddots & -1 \\ & & & & 1 \end{pmatrix}$$

C'est-à-dire :

$$T_n^{-1} = (u_{ij}) \text{ avec } u_{ij} = \begin{cases} 1 & \text{si } j = i \\ -1 & \text{si } j = i+1 \\ 0 & \text{sinon} \end{cases}$$

Posons alors : ${}^t\left(T_n^{-1}\right) = \left(u'_{ij}\right)$ avec $u'_{ij} = \begin{cases} 1 & \text{si } j = i \\ -1 & \text{si } j = i-1. \\ 0 & \text{sinon} \end{cases}$

On a : $M_n^{-1} = T_n^{-1}\,{}^t\left(T_n^{-1}\right) = \begin{pmatrix} 1 & -1 & & & \\ & 1 & -1 & (0) & \\ & & 1 & \ddots & \\ & (0) & & \ddots & -1 \\ & & & & 1 \end{pmatrix}\begin{pmatrix} 1 & & & & \\ -1 & 1 & & (0) & \\ & -1 & 1 & & \\ & (0) & \ddots & \ddots & \\ & & & -1 & 1 \end{pmatrix} = (b_{ij})$.

Or : $b_{ij} = \sum_{k=1}^{n} u_{ik} u'_{kj}$ avec $u_{ik} \neq 0 \Leftrightarrow k = i$ ou $k = i+1$ et $u'_{kj} \neq 0 \Leftrightarrow k = j$ ou $k = j+1$.

Ainsi, ces contraintes sur *k* entraînent, pour obtenir des produits $u_{ik}u'_{kj}$ non nuls, des contraintes sur i et j : si $k = i$ alors $j = i$ ou $j = i-1$ et si $k = i+1$ alors $j = i+1$ ou $j = i$.

En d'autres termes, seules la diagonale principale ($i = j$) et les deux diagonales la bordant ($j = i+1$ et $j = i-1$) comporteront des coefficients non nuls.
Etudions ces trois situations :

- Pour $j = i$, on a :

  Pour $i \in [\![1; n-1]\!]$, $b_{ii} = \sum_{k=1}^{n} u_{ik} u'_{ki} = u_{ii} u'_{ii} + u_{i,i+1} u'_{i+1,i} = 1\times 1 + (-1)\times(-1) = 2$.

  Pour $i = n$, $b_{ii} = b_{nn} = \sum_{k=1}^{n} u_{nk} u'_{kn} = u_{nn} u'_{nn} = 1\times 1 = 1$

- Pour $j = i+1$ ($i \in [\![1; n-1]\!]$), on a :

  $b_{ij} = b_{i,i+1} = \sum_{k=1}^{n} u_{ik} u'_{k,i+1} = u_{i,i+1} u'_{i+1,i+1} = -1\times 1 = -1$

- Pour $j = i-1$ ($i \in [\![2; n]\!]$), on obtient encore -1.

En définitive, on a :

$$M_n^{-1} = \begin{pmatrix} 2 & -1 & & & & \\ -1 & 2 & -1 & & (0) & \\ & -1 & 2 & -1 & & \\ & & -1 & \ddots & \ddots & \\ & (0) & & \ddots & 2 & -1 \\ & & & & -1 & 1 \end{pmatrix}$$

i) Pour obtenir M_n^{-1}, on utilise la fonction `inv` du module `linalg`.
On peut alors considérer le script suivant :

```
for n in range(2,11):
    print("\nn = ",n)
    print(np.linalg.inv(MatM_Create(n)))
```

On vérifie la correction du résultat obtenu à la question précédente.

2015

1. D'après ENSAM 2015 (PSI) ♠

(Planche Officiel de la Taupe Gyroscope 2015 N°267)

1. On peut utiliser la syntaxe efficace suivante :

```
L30 = [True for i in range(31)]
```

2. On a, par exemple :

```
def modifier(L,p):
    if p == 0:
        L[0] = False
    elif p == 1:
        L[1] = False
    else:
        k = 2*p
        while k < len(L):
            L[k] = False
            k += p
```

3. On réutilise bien sûr la fonction `modifier` afin d'éliminer les multiples de p (à partir de $2p$) :

```
def premiers(n):
    # Génération d'une liste contenant n+1 True
    L = [True for i in range(n+1)]
    # Mise à jour de la liste (crible d'Erathostène)
    for p in range(n+1):
        modifier(L,p)
    # Génération de la liste des entiers premiers <= n
    LP = []
    for i in range(len(L)):
        if L[i] == True:
            LP.append(i)
    return(LP)
```

4. Il suffit ici d'appeler la fonction `premiers` avec 823 417 comme argument et d'afficher le dernier terme de la liste renvoyée. Ce dernier terme étant 823 399, on en déduit que 823 417 n'est pas premier (d'ailleurs : $823\,417 = 7 \times 117\,631$.

5. Il s'agit du crible d'Erathostène.

2. ENSAM 2015 (PSI) ♠♠

A venir…

3. ENSAM 2015 (PSI) ♠♠

(Planche Officiel de la Taupe Gyroscope 2015 N°269)

A venir…

4. Centrale Supélec 2015 Exercice type N°1 (PSI) ♠♠

A venir…

5. ENSAM 2015 (PT) ♠

A venir…