1) *(Centrale)* Le début, c'est des maths... Il vient immédiatement

$$f_1(x) = 2x \text{ et } f_2(x) = 2\int_0^x \sqrt{2t}\,\mathrm{d}t = 2\sqrt{2}\frac{1}{1/2+1}x^{1/2+1}$$

et

$$\text{si } f_n(x) = \alpha_n x^{\beta_n}, \text{ alors } f_{n+1}(x) = 2\sqrt{\alpha_n}\frac{1}{\beta_n/2+1}x^{\beta_n/2+1}.$$

Je définis donc les deux suites (α_n) et (β_n) par

$$\alpha_0 = 1,\ \beta_0 = 0 \text{ et } \forall n \in \mathbb{N} \quad \alpha_{n+1} = \frac{2\sqrt{\alpha_n}}{\beta_n/2+1} \text{ et } \beta_{n+1} = \beta_n/2+1.$$

Une récurrence immédiate (compte tenu du calcul précédent) montre alors que

$$\forall n \in \mathbb{N} \quad \forall x \in [0,1] \quad f_n(x) = \alpha_n x^{\beta_n}.$$

La suite (β_n) est arithmético-géométrique. La recherche d'un point fixe t tel que $t = t/2+1$ donne $t = 2$ et donc (par soustraction membre à membre) la suite $(\beta_n - 2)$ est géométrique, de raison $1/2$ et de premier terme -2. D'où

$$\boxed{\forall n \in \mathbb{N} \quad \beta_n = 2 - 1/2^{n-1}.\ (\beta_n) \text{ converge vers } 2.}$$

Je peux maintenant réécrire la définition de (α_n) :

$$\alpha_0 = 1 \text{ et } \forall n \in \mathbb{N}^* \quad \alpha_n = \frac{\sqrt{\alpha_{n-1}}}{1-1/2^n} \text{ puisque } \beta_{n-1}/2+1 = 2 - 1/2^{n-1}.$$

Une boucle Python permet alors d'obtenir les valeurs demandées : je construis une liste contenant les valeurs successives, en ajoutant à chaque étape la nouvelle valeur à l'aide de la méthode `append`.

On peut conjecturer que (α_n) converge vers 1... La fin, c'est à nouveau des maths !

D'après ce qui précède, les α_n sont tous strictement positifs et

$$\forall n \in \mathbb{N}^* \quad -\ln\alpha_n = -\frac{1}{2}\ln\alpha_{n-1} + \ln\left(1-2^{-n}\right).$$

La relation de l'énoncé s'en déduit par récurrence. J'utilise ensuite la majoration classique

$$\forall t > 0 \quad \ln t \leq t - 1$$

pour écrire

$$\forall k \quad 0 \leq \ln\frac{1}{1-2^{-n-1+k}} \leq \frac{1}{1-2^{-n-1+k}} - 1 = \frac{2^{-n-1+k}}{1-2^{-n-1+k}}$$

d'où

$$0 \leq \ln\alpha_n \leq \sum_{k=1}^{n}\frac{2^{-n}}{1-2^{-n-1+k}} = \sum_{k=1}^{n}\frac{1}{2^n-2^{k-1}} \leq \frac{n}{2^{n-1}}$$

puisque pour $k \leq n$, $2^n - 2^{k-1} \geq 2^n - 2^{n-1} = 2^{n-1}$.

En conclusion, par croissances comparées et théorème d'encadrement, $\ln\alpha_n$ tend vers 0, autrement dit

$$\boxed{(\alpha_n) \text{ converge vers } 1.}$$

Il résulte de ce qui précède que la suite de fonctions (f_n) converge simplement vers $f : x \mapsto x^2$. Pour étudier la convergence uniforme, je fixe $n > 0$ et j'écris, pour tout x dans $[0,1]$,

$$|f_n(x) - f(x)| = \left|\alpha_n x^{\beta_n} - x^2\right| = \left|(\alpha_n - 1)x^{\beta_n} + x^{\beta_n} - x^2\right| \leq |\alpha_n - 1| + \left|x^{\beta_n} - x^2\right|.$$

Comme $\beta_n < 2$, une étude rapide montre que la fonction $\phi_n : x \mapsto x^{\beta_n} - x^2$ est à valeurs positives sur $[0,1]$ et atteint son maximum M_n en x_n défini par

$$\ln x_n = 2^{n-1}\ln\left(1 - \frac{1}{2^n}\right) \sim -\frac{1}{2}.$$

J'en déduis que $(\beta_n \ln x_n)$ converge vers -1 et donc que

$$M_n = \phi_n(x_n) = x_n^{\beta_n} - x_n^2 \underset{n\to\infty}{\longrightarrow} e^{-1} - e^{-1} = 0.$$

Or d'après la majoration précédente $\sup\limits_{[0,1]}|f_n - f| \leq |\alpha_n - 1| + M_n$. Donc par encadrement

$$\boxed{(f_n) \text{ converge uniformément vers } 0 \text{ sur } [0,1].}$$

2) On peut utiliser de simples vecteurs `numpy` pour représenter chaque polynôme par la liste de ses coefficients. La multiplication par X est alors un simple décalage. Mais le calcul des produits de polynômes ou des valeurs prises par une fonction polynomiale sera plus pénible à programmer. Je vais donc plutôt utiliser le module `Polynomial` que les examinateurs de Centrale semblent apprécier... Je construis à l'aide d'une boucle une liste `P` telle que `P[n]` soit le polynôme P_n. On conjecture et l'on montre aisément par récurrence (sans Python !) que

$$\boxed{P_n \text{ est de degré } n, \text{ de la parité de } n \text{ et de coefficient dominant } 2^n.}$$

À nouveau des maths pour justifier (classiquement) que l'application de l'énoncé définit bien un produit scalaire... Comme le module `Polynomial` permet d'utiliser les polynômes ainsi construits comme des fonctions, je peux calculer directement les produits scalaires demandés (en tout cas des valeurs numériques approchées !).

```
import numpy as np
from numpy.polynomial import Polynomial
import scipy.integrate as integr
X=Polynomial([0,1])
P=[]
P.append(Polynomial([1]))
P.append(2*X)
for k in range(2,9):
    P.append(2*X*P[k-1]-P[k-2])

def pscal(A,B):
    return integr.quad(lambda t: sqrt(1-t**2)*A(t)*B(t),-1,1)[0]
G=np.zeros((9,9))
for i in range(9):
    for j in range(9):
        G[i,j]=round(pscal(P[i],P[j]),3)
print(G)
```

La matrice obtenue, arrondie au millième, est diagonale (avec des 0. et des $-0.$ (!) en dehors de la diagonale). Et sur la diagonale, que des 1.571 (ce qui fait penser à $\pi/2\ldots$). On peut apparemment conjecturer que la famille (P_n) est orthogonale. "En déduire" qu'elle l'est à partir de valeurs approchées serait abusif ! Elle l'est en réalité, car il s'agit des (classiques) polynômes de Tchebychev de seconde espèce, qui vérifient

$$\forall n \in \mathbb{N} \quad \forall \theta \in \left]0, \pi\right[\quad P_n(\cos\theta) = \frac{\sin(n+1)\theta}{\sin\theta}.$$

Le changement de variable $t = \cos\theta$ ($\mathcal{C}^1$ bijectif strictement décroissant de $\left]0, \pi\right[$ dans $\left]-1, 1\right[$) permet de démontrer le résultat (et aussi que $(P_n|P_n) = \pi/2$ pour tout n).

Pour déterminer la matrice de ϕ dans la base $\mathcal{B} = (P_0, \ldots, P_8)$, je pourrais utiliser le fait que $\mathcal{C} = (Q_0, \ldots, Q_8)$ est orthonormale, où $Q_k = \sqrt{2/\pi}P_k$, et utiliser la formule classique $\big(Q_i|\phi(Q_j)\big)$ pour remplir la matrice de ϕ dans $\mathcal{C}$ et en déduire la matrice dans $\mathcal{B}$. Mais il est aussi simple de déterminer directement les coordonnées de $\phi(P_j)$ dans $\mathcal{B}$, en écrivant grâce à l'orthogonalité de $\mathcal{B}$:

$$\phi(P_j) = \sum_{k=0}^{8} a_{k,j} P_k \quad \text{d'où} \quad \big(P_i|\phi(P_j)\big) = a_{i,j}(P_i|P_i) \quad \text{d'où} \quad a_{i,j} = \frac{2}{\pi}\big(P_i|\phi(P_j)\big).$$

```
from math import pi
def phi(A):
    return 3*X*A.deriv(1)-A.deriv(2)
M=np.zeros((9,9))
for i in range(9):
    for j in range(9):
        M[i,j]=round(2/pi*pscal(P[i],phi(P[j])),3)
print(M)
```

On obtient une matrice apparemment triangulaire supérieure, à coefficients entiers. Ceci peut se retrouver grâce aux relations entre les polynômes de Tchebychev de 1re et 2nde espèce et aux équations différentielles qu'ils vérifient (voir par exemple la page Wikipedia intitulée « Polynômes de Tchebychev »).

3) *(Centrale)* Pas de grosse difficulté, attention tout de même aux indices : il est plus commode avec Python de numéroter lignes et colonnes de 0 à $n-1$.

```
import numpy as np

def Matrice(L):
    n=len(L)
    M=np.zeros((n,n))
    for j in range(n-1):
        M[j+1,j]=1
    for i in range(n):
        M[i,n-1]=-L[i]
    return(M)

L=np.random.randint(1,10,8)
print(L)
print(np.poly(Matrice(L)))
```

Noter qu'il y a dans Python (au moins) deux fonctions `randint` : avec celle du module `random`, `randint(a,b)` renvoie un entier aléatoire de $[\![a,b]\!]$ (b compris), tandis qu'avec celle du module `numpy.random` (que j'utilise ici) on obtient un entier de $[\![a,b[\![$ (b exclu !). L'avantage de cette dernière est de fournir directement une vecteur aléatoire de taille n avec la syntaxe `randint(a,b,n)` ou une matrice aléatoire par `randint(a,b,(n,p))`.

On peut conjecturer que les coefficients du polynôme caractéristique sont, après le 1 du terme dominant, ceux de la liste fournie, lus de droite à gauche... C'est un résultat montré dans le D.L.2 *(matrice compagne d'un polynôme)*, la démonstration est dans le corrigé dudit D.L. ! La dernière question apporte un résultat complémentaire.

Notons $\mathcal{B} = (e_1, \dots e_n)$ la base canonique de $E = \mathbb{K}^n$ et $u = \operatorname{Can} M$. Par construction et grâce à une récurrence immédiate,

$$\forall j \in [\![1, n-1]\!] \quad u(e_j) = e_{j+1} \quad \text{d'où} \quad \forall k \in [\![0, n-1]\!] \quad u^k(e_1) = e_{k+1}.$$

Donc, si $P = \sum_{k=0}^{n-1} b_k X^k$, alors $P(u)(e_1) = \sum_{k=0}^{n-1} b_k e_{k+1} = \sum_{j=1}^{n} b_{j-1} e_j$. Comme $\mathcal{B}$ est une famille libre (!), il en résulte que, si P est un polynôme non nul de degré au plus $n-1$, alors $P(u) \neq 0$ (puisqu'il ne s'annule pas en e_1). Par contraposée, j'ai montré que tout polynôme annulateur non nul de u est de degré au moins égal à n.

Notons $A = X^n + \sum_{k=0}^{n-1} a_k X^k$. D'après ce qui précède, $A = \chi_M = \chi_u$. Montrons que M est diagonalisable si et seulement si A admet n racines simples :

- si M est diagonalisable, alors $P = \prod_{\lambda \in \operatorname{Sp} u} (X - \lambda)$ est un polynôme annulateur (non nul !) de u, ce qui implique d'après le résultat ci-dessus que u admet n valeurs propres distinctes et donc que A a admet n racines distinctes, nécessairement simples ;
- réciproquement, c'est un résultat du cours : si A admet n racines simples, alors M admet n sous-espaces propres, qui sont en somme directe et sont donc n droites dont la somme est E ! Noter qu'il n'est pas nécessaire d'invoquer le théorème de Cayley-Hamilton...

4) *(Centrale)* Il s'agit de montrer que, pour n fixé dans $\mathbb{N}$, H_n est un ensemble fini. Or, si un couple (p, q) d'entiers naturels vérifie $2p + 3q = n$, j'ai nécessairement $2p \leq n$ et $3q \leq n$, donc $p \leq \lfloor n/2 \rfloor$ et $q \leq \lfloor n/3 \rfloor$. Plus précisément, pour q fixé entre 0 et $\lfloor n/3 \rfloor$, il y a 0 ou 1 élément de H_n de la forme (p, q), selon la parité de $n - 3q$! Cela prouve que H_n est fini, de cardinal au plus égal à $\lfloor n/3 \rfloor + 1$, lui-même (grossièrement) majoré par n :

$$\boxed{\sigma(n) \text{ existe bien et } \sigma(n) \text{ est un } O(n).}$$

Pour $n \leq 2$, on a vite fait le tour des solutions dans $\mathbb{N}^2$ de l'équation $(E_n) \quad 2p + 3q = n$!

$$\boxed{\sigma(0) = 1 \; ; \; \sigma(1) = 0 \; ; \; \sigma(2) = 1.}$$

Pour $n \geq 3$, je remarque (habilement) que n et $n-3$ sont deux entiers naturels de parités différentes, donc l'un des deux est pair ! Ce qui me donne une solution de (E_n), puisque je peux écrire soit $n = 2p$ (auquel cas $(p, 0) \in H_n$), soit $n = 2p + 3$ (auquel cas $(p, 1) \in H_n$). Ainsi

$$\boxed{\text{Si } n \geq 3, \text{ alors } \sigma(n) \geq 1.}$$

Il n'est pas clair de conjecturer une formule générale pour $\sigma(n)$... Python fournit les valeurs (pour n "petit"), grâce à une double boucle (en testant tous les couples (p, q) évoqués au début), **ou plus efficacement** à l'aide d'une simple boucle, puisque pour q fixé entre 0 et $\lfloor n/3 \rfloor$, il suffit de tester la parité de $n - 3q$ (cf. la remarque du début). D'où le programme, avec une complexité linéaire :

```
def sigma(n):
    nb=0
    for q in range(1+n//3):
        if (n-3*q)%2==0: nb+=1
    return nb

print([sigma(k) for k in range(26)])
```

Ce qui donne : `[1,0,1,1,1,1,2,1,2,2,2,2,3,2,3,3,3,3,4,3,4,4,4,4,5,4]`.

L'encadrement déjà établi, pour $n \geq 3$, $1 \leq \sigma(n) \leq n$ donne (sachant que les deux séries entières $\sum x^n$ et $\sum nx^n$ ont pour rayon de convergence 1) :

$$\boxed{\text{Le rayon de convergence de } \textstyle\sum \sigma(n)\, x^n \text{ vaut 1.}}$$

Plus précisément, puisque $\sigma(n)$ ne tend pas vers 0, l'ensemble de définition de $S : x \mapsto \sum_{n=0}^{\infty} \sigma(n)\, x^n$ est $]-1, 1[$ (divergence grossière en ± 1). Pour établir la relation de l'énoncé, j'utilise les sommes des séries géométriques classiques, pour x fixé dans $]-1, 1[$:

$$f : x \mapsto \frac{1}{1 - x^2} = \sum_{p=0}^{\infty} x^{2p} = \sum_{i=0}^{\infty} a_i x^i \quad \text{et} \quad g : x \mapsto \frac{1}{1 - x^3} = \sum_{q=0}^{\infty} x^{3q} = \sum_{j=0}^{\infty} b_j x^j$$

où a_i (*resp.* b_j) vaut 1 si i (*resp.* j) est multiple de 2 (*resp.* 3) et vaut 0 sinon. Le produit de Cauchy de ces deux séries absolument convergentes s'écrit :

$$f(x)\, g(x) = \sum_{n=0}^{\infty} c_n x^n \quad \text{où} \quad \forall n \in \mathbb{N} \quad c_n = \sum_{i+j=n} a_i b_j.$$

Comme $a_i b_j$ vaut 0 ou 1, c_n est le nombre de couples (i, j) de $\mathbb{N}^2$ tels que ($i + j = n$ et $a_i b_j = 1$), c'est-à-dire ($i + j = n$ et i pair et j impair). Autrement dit $c_n = \sigma(n)$! En conclusion

$$\boxed{\forall x \in \,]-1, 1[\quad \sum_{n=0}^{\infty} \sigma(n)\, x^n = \frac{1}{1 - x^2} \cdot \frac{1}{1 - x^3}.}$$

Pour retrouver les valeurs de $\sigma(n)$ pour $n \leq 25$, il suffit d'obtenir les termes de degré au plus 25 dans le développement de $f(x)\, g(x)$. Ils sont fournis par le développements des sommes partielles, qui sont des polynômes. Le module `Polynomial` permet de les obtenir facilement :

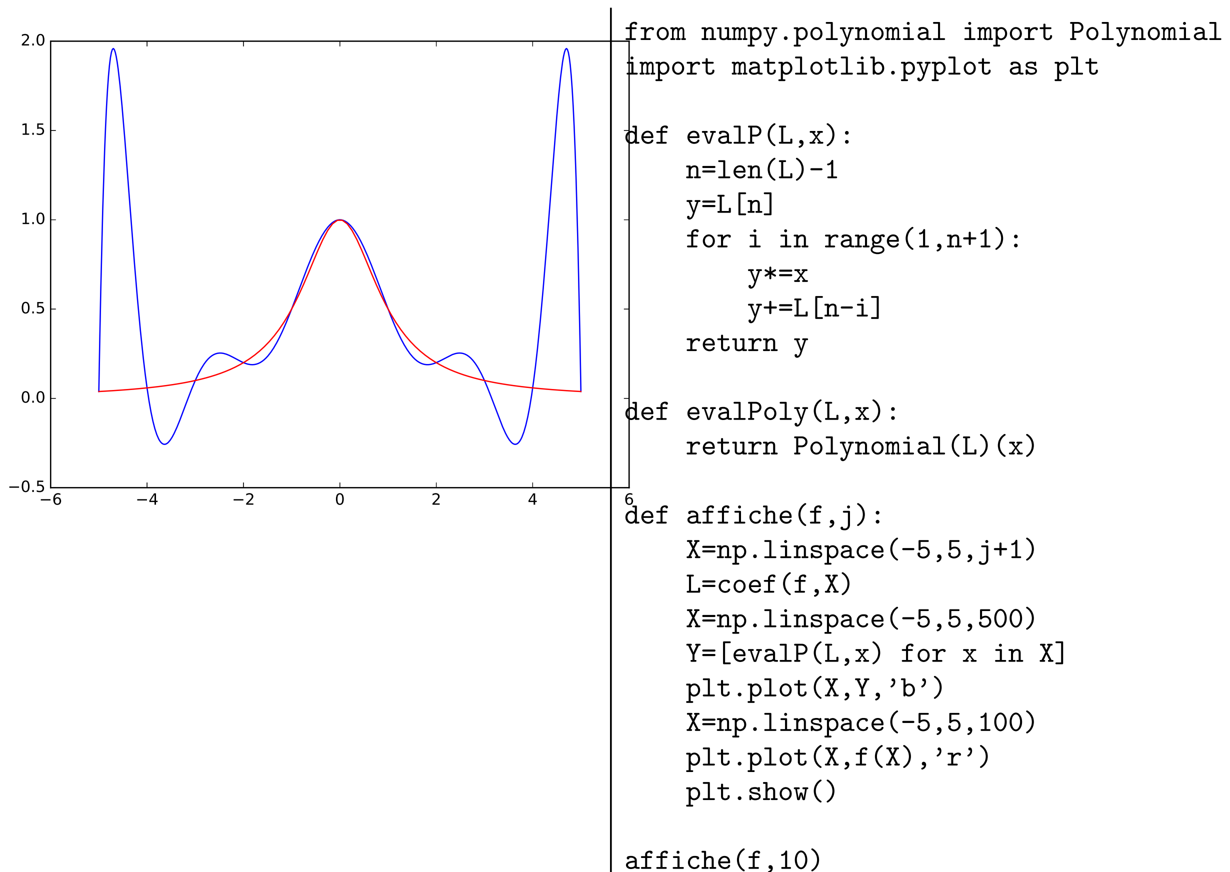
```
from numpy.polynomial import Polynomial

def a(k,d):
    if k%d==0:
        return 1
    else:
        return 0

A=Polynomial([a(k,2) for k in range(26)])
B=Polynomial([a(k,3) for k in range(26)])

print((A*B).coef[:26])
```

5) *(ENSAM)* Les polynômes de Lagrange ne sont plus au programme en PSI... Rappelons tout de même que, pour toute famille $(a_k)_{0\le k\le n}$ de $n+1$ scalaires distincts et toute famille $(b_k)_{0\le k\le n}$ de scalaires (non nécessairement distincts), il existe un unique polynôme P de $\mathbb{K}_n[X]$ vérifiant : $\forall k \in [\![0, n]\!] \quad P(a_k) = b_k$ (cf. chapitre 1, p. 11).

Pour le calcul des coefficients, on peut comme le suggère l'énoncé résoudre brutalement le système linéaire dont les coefficients de P sont les solutions. Je remplis donc la matrice (de Vandermonde !) dudit système et le second membre, avant d'appeler `numpy.linalg.solve`.

```
import numpy as np
from numpy.linalg import solve
def coef(h,a):
    n=len(a)
    A=np.zeros((n,n))
    for i in range(n):
        for j in range(n):
            A[i,j]=a[i]**j
    X=alg.solve(A,[h(a[i]) for i in range(n)])
    return list(X)
```

Le résultat de `solve` étant un vecteur `numpy`, je le convertis en liste. Pour tracer le graphe de P, je programme la fonction polynomiale associée en utilisant, soit le module `Polynomial`, soit l'algorithme de Horner (cf. le TD 1 d'info, exo 3). Voici le graphique pour $j = 10$ et les commandes l'ayant produit.




```
from numpy.polynomial import Polynomial
import matplotlib.pyplot as plt

def evalP(L,x):
    n=len(L)-1
    y=L[n]
    for i in range(1,n+1):
        y*=x
        y+=L[n-i]
    return y

def evalPoly(L,x):
    return Polynomial(L)(x)

def affiche(f,j):
    X=np.linspace(-5,5,j+1)
    L=coef(f,X)
    X=np.linspace(-5,5,500)
    Y=[evalP(L,x) for x in X]
    plt.plot(X,Y,'b')
    X=np.linspace(-5,5,100)
    plt.plot(X,f(X),'r')
    plt.show()

affiche(f,10)
```

Où l'on voit que l'approximation est loin d'être "uniforme" : le graphe de P passe bien par les 11 points requis, mais P prend par endroit des valeurs très éloignées de celles prises par f... Et cela ne s'arrange pas lorsque j augmente ! Pour constater le phénomène, **ne pas oublier** d'utiliser **deux** `linspace` : le premier pour définir les points d'interpolation, le second pour tracer les graphes, avec beaucoup plus de points pour une précision convenable...

6) *(ENSAM)* Il s'agit ici de "marche aléatoire" sur une droite, mais sans retour en arrière ! La position finale de la balle, après n étapes, est déterminée par un entier de $[\![0, n]\!]$. Pour la simulation numérique, j'écris directement les fonctions avec un paramètre p correspondant à la probabilité p d'avancer d'une case. Pour la fonction `tirer(n,p)`, il suffit de répéter n fois le tirage d'un flottant aléatoire de $[0, 1[$ (ce que fait la fonction `random` du module `random`, aussi bien que la fonction `random` du module

`numpy.random` !) et de compter le nombre de valeurs inférieures à p obtenues. J'en déduis une fonction d'en-tête `simul(n,N,p)`, j'initialise un tableau de $n+1$ zéros qui contiendra le nombre de fois que chaque position aura été atteinte durant les N appels de `tirer`. Il n'y a plus qu'à le diviser par N pour obtenir les fréquences (utiliser un **vecteur** numpy et non une liste, qui ne supporterait pas la division par N...) :

```
import numpy as np

def tirer(n,p):
    r=0
    for k in range(n):
        if np.random.random()<p:
            r+=1
    return r

def simul(n,N,p):
    L=np.zeros(n+1)
    for k in range(N):
        L[tirer(n,p)]+=1
    return L/N
```

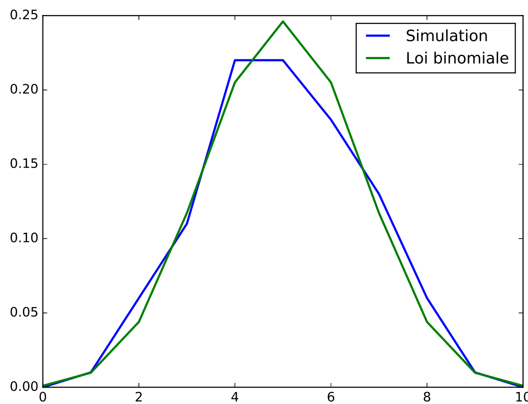
La "courbe théorique" évoquée par l'énoncé est bien sûr l'histogramme de la *loi binomiale*. D'où l'intérêt de calculer les coefficients du binôme ! Le programme le plus court est donné par la formule $\dfrac{n!}{k!\,(n-k)!}$, qui ne pose pas de problème pour de petites valeurs, mais qu'il vaut mieux éviter... Utiliser plutôt la formule simplifiée

$$\binom{n}{k} = \frac{n\,(n-1)\cdots(n-k+1)}{k!} = \prod_{j=0}^{k-1} \frac{n-j}{j+1}$$

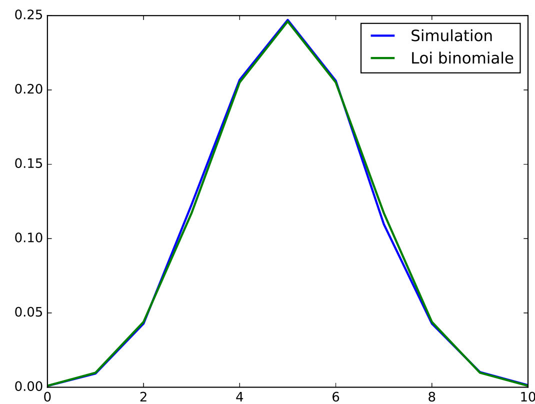
après avoir pris soin de remplacer k par $n-k$ lorsque $n-k<k$! Noter qu'avec Python, cette formule sera efficace mais renverra un flottant. Si l'on a le temps et que l'on veut la valeur exacte de l'entier, on calcule les deux entiers $\prod_{j=0}^{k-1}(n-j)$ et $k!$, puis le quotient (forcément exact !) **avec un double slash**...
Pour tracer le graphe les flottants suffiront.

```
import matplotlib.pyplot as plt

def binom(n,k):
    if n-k<k: k=n-k
    if k<0:
        return 0
    else:
        f=1
        for j in range(k): f*=(n-j)/(j+1)
        return f

def graphe(n,N,p):
    L1=simul(n,N,p)
    L2=[binom(n,k)*p**k*(1-p)**(n-k) for k in range(n+1)]
    X=[k for k in range(n+1)]
    plt.plot(X,L1,linewidth=2,label='Simulation')
    plt.plot(X,L2,linewidth=2,label='Loi binomiale')
    plt.legend()
    plt.show()
```

Même s'il s'agit *a priori* d'histogrammes, le tracé des lignes brisées permet de mieux comparer les valeurs obtenues. Page suivante les graphiques obtenus avec `graphe(10,100,0.5)` demandé par l'énoncé, puis avec `graphe(10,10000,0.5)` où l'on apprécie la "*convergence en loi*"...



$N = 100$



$N = 10\ 000$

7) *(ENSAM)* Il s'agit de l'algorithme historique du *crible d'Ératosthène* (savant grec et éclectique du III$^{\text{e}}$ siècle avant J.-C., algorithme apprécié à l'époque car il pouvait s'exécuter aisément sans le moindre ordinateur ! L'idée est de barrer dans la liste des entiers de 0 à n, supposés rangés dans l'ordre et dans la liste L, successivement 0, 1, puis les multiples de 2 (à partir de 4), les multiples de 3 (à partir de 6), etc. Informatiquement, le fait de "barrer" l'entier k se traduira par le passage de `True` à `False` de la valeur $L[k]$.

Pour la fonction d'en-tête `modifier(L,p)`, après avoir traité les cas où $p < 2$, je mets à `False` tous les multiples de p à partir de $2p$ et jusqu'à la fin de L (plus efficace que de balayer **tous** les indices et de tester s'ils sont multiples de p !).

Pour la fonction d'en-tête `premiers(n)`, le principe du crible est d'initialiser une liste L indexée de 0 à n et remplie de `True`, puis d'exécuter `modifier(L,p)` pour des valeurs successives de p, de sorte que les derniers `True` dans L correspondent aux nombres premiers. La première idée (naïve) et de balayer tous les p de 2 à n. Classiquement, on peut s'arrêter à $\sqrt{n}$, ce qui est une optimisation importante ! En effet, si $m \leq n$ n'est pas premier, il admet nécessairement un diviseur au plus égal à $\sqrt{m}$ (donc inférieur ou égal à $\sqrt{n}$!), puisque si $q > \sqrt{m}$ divise m, alors $m/q < \sqrt{m}$!! Autre optimisation non négligeable : il suffit d'exécuter `modifier(L,p)` pour les p tels que $L[p]$ vaut `True` puisque sinon les multiples de p sont déjà "barrés" (un multiple d'un multiple est un multiple...).

```
from math import sqrt

def modifier(L,p):
    if p<2: L[p]=False
    else:
        for k in range(2*p,len(L),p): L[k]=False
    return L

def premiers(n):
    L=[True for k in range(n+1)]
    p=0
    for p in range(int(sqrt(n))+1):
        if L[p]: L=modifier(L,p)
    return [k for k in range(n+1) if L[k]]
```

Pour répondre à la dernière question, la liste générée étant très longue, j'exécute `premiers(823417)[-1]`, qui me donne le plus grand nombre premier inférieur ou égal à 823 417 : c'est 823 399, donc

$$\boxed{\text{823 417 n'et pas premier.}}$$

8) *(ENSAM)* Petit rappel sur la lecture d'un fichier texte : ouvrir avec `open`, lire la ligne (unique dans ce cas) avec `readline` et fermer le fichier avec `close`, c'est un exercice d'anglais...

Noter que Pyzo est capable de trouver le fichier `Pi.txt` dans le répertoire courant, **à condition d'utiliser la commande "Exécuter/Démarrer le script"**, qui redémarre le shell et exécute le script courant. Raccourci clavier Ctrl+F5 ou Maj+Ctrl+E. Sinon, il peut s'avérer nécessaire d'indiquer le chemin complet vers le fichier ; dans ce cas, penser — pour le paramètre de `open` — à faire précéder la chaîne contenant ledit chemin d'un `r` (pour *rawstring*, chaîne brute), afin d'empêcher Python d'interpréter d'éventuels caractères spéciaux ; par exemple `open(r'U:\Python\Oral.py')`.

Une fois la chaîne chargée, penser au *slicing* pour obtenir les 10 premiers caractères (`[0:10]` ou `[:10]` pour les indices de 0 à 9) et les 10 derniers (`[-11:]` ou `[n-11:n]` à condition que `n` contienne la longueur de la chaîne...).

Pour la recherche d'un motif, une version naïve suffira. Je profite là encore du *slicing* pour comparer d'un coup le motif avec une sous-chaîne de la source (sinon il suffit d'une boucle pour comparer caractère par caractère...). On pourrait aussi utiliser les fonctions de Python, mais pas sûr que ce soit au goût du jury ! L'appel `source.find(motif)` renvoie la même chose que `pos(motif,source)`, à savoir l'indice de sa première occurrence si `motif` est une sous-chaîne de `source` et -1 sinon.

```
f=open('Pi.txt')
chaine=f.readline()
f.close()

print('10 premières : ',chaine[:10])
print('10 dernières : ',chaine[-11:])

def pos(motif,source):
    p,n=len(motif),len(source)
    k=0
    while k<=n-p:
        if source[k:k+p]==motif: return k
        k+=1
    return -1
```

Il apparaît que `'000'` se trouve à partir de la 601e décimale mais que `'0000'` est absente.

De même `'123'` est à partir de la 1 924e décimale, `'1234'` est introuvable.

Enfin `'314'` se trouve à partir de la 2 120e décimale (et pas au début puisqu'on a exclu la partie entière !).

9) *(ENSAM)* Le transtypage permet de faire faire le travail à Python avec un minimum de programmation ! Attention tout de même au type des objets manipulés : `list(str(n))` renvoie la liste des chiffres composant n, mais chaque chiffre y figure en tant que **caractère** ! D'où cette solution (j'utilise la fonction `sum` sans trop de scrupules, si besoin une boucle peut la remplacer...).

```
def chiffres(n):
    return [int(c) for c in list(str(n))]

def narcissique(n):
    L=chiffres(n)
    p=len(L)
    return sum(L)**p==n

print([n for n in range(10001) if narcissique(n)])
```

Le résultat est `[0,1,2,3,4,5,6,7,8,9,81,512,2 401]`.

10) *(Centrale)* Lorsqu'on a besoin de précision dans ce type de calcul de somme, il vaut mieux commencer par ajouter à 0 les termes les plus petits, afin que les retenues puissent se reporter sur les plus gros termes (sinon les plus petits termes sont purement et simplement considérés comme nuls).

```
from math import factorial,exp

def u(n):
    s=0
    for k in range(n-1,-1,-1): s+=1/factorial(k)
    return (exp(1)-s)**(1/n)
```

Alors `[n*u(n) for n in range(10,31)]` renvoie une liste dont les 13 derniers termes sont nuls.

Mais en augmentant la précision à 50 décimales (`mp` comme multiprécision), à condition de remplacer `factorial` par `mp.factorial` et `exp` par `mp.exp`…

```
from mpmath import mp
mp.dps=50

def u(n):
    s=0
    for k in range(n-1,-1,-1): s+=1/mp.factorial(k)
    return (mp.exp(1)-s)**(1/n)
```

on obtient des valeurs qui croissent doucement, $30u_{30}$ valant environ 2.49. Où l'on voit les limites du calcul numérique…

La formule de Taylor avec reste intégral à l'ordre $n+2$, appliquée entre 0 et 1 à la fonction exp, donne

$$e = \sum_{k=0}^{n+2} \frac{1}{k!} + \int_0^1 \frac{(1-t)^{n+2}}{(n+2)!} e^t \mathrm{d}t$$

d'où

$$e - \sum_{k=0}^{n+2} \frac{1}{k!} = \frac{1}{n!} \left(1 + \frac{1}{n+1} + \frac{1}{(n+1)(n+2)} + \int_0^1 \frac{(1-t)^{n+2}}{(n+1)(n+2)} e^t \mathrm{d}t \right)$$

donc

$$c_n = \ln \left(1 + \frac{1}{n+1} + \frac{1}{(n+1)(n+2)} + \int_0^1 \frac{(1-t)^{n+2}}{(n+1)(n+2)} e^t \mathrm{d}t \right) \text{ convient}$$

et en majorant classiquement la dernière intégrale

$$c_n = \ln \left(1 + \frac{1}{n+1} + \frac{1}{(n+1)(n+2)} + O\left(\frac{1}{n^3}\right) \right) \; ;$$

or

$$\frac{1}{n+1} + \frac{1}{(n+1)(n+2)} = \frac{2}{n+1} - \frac{1}{n+2} = \frac{2}{n}\left(1 - \frac{1}{n}\right) - \frac{1}{n}\left(1 - \frac{2}{n}\right) + O\left(\frac{1}{n^3}\right) = \frac{1}{n} + O\left(\frac{1}{n^3}\right)$$

d'où

$$\boxed{c_n = \frac{1}{n} - \frac{1}{2n^2} + O\left(\frac{1}{n^3}\right).}$$

En particulier (c_n) converge vers 0, d'où grâce à la formule de Stirling

$$\ln u_n = \frac{1}{n}(c_n - \ln n!) = \frac{1}{n}\left(-n \ln n + n - \frac{\ln n}{2} - \frac{\ln(2\pi)}{2} + o(1) \right)$$

et

$$u_n = \exp \ln u_n = \frac{e}{n} \exp\left(-\frac{\ln n}{2n} - \frac{\ln(2\pi)}{2n} + o\left(\frac{1}{n}\right) \right)$$

soit finalement

$$\boxed{u_n = \frac{e}{n}\left(1 - \frac{\ln n}{2n} - \frac{\ln(2\pi)}{2n} + o\left(\frac{1}{n}\right) \right).}$$

Où l'on voit que (nu_n) converge vers e, ce qui n'était pas évident au vu des valeurs approchées.

11) *(Centrale)* Même genre de surprise que dans l'exercice précédent...

Pour le calcul des sommes partielles successives, **il faut** les stocker (ou les afficher) au fur et à mesure et non pas les recalculer l'une après l'autre à partir de rien.

Avec la précision par défaut, les premières valeurs semblent converger vers une limite proche de -1.052 puis se mettent à s'en éloigner... Avec 50 décimales, la convergence apparaît de façon plus convaincante. Et en effet la formule du binôme montre (après simplifications) que $\left(2+\sqrt{3}\right)^n + \left(2-\sqrt{3}\right)^n$ est un entier pair, d'où $u_n = -v_n$ or $\sum v_n$ est absolument convergente par comparaison à une série géométrique, d'où la convergence absolue de $\sum u_n$ malgré les premières apparences. Le calcul numérique de u_n devient vite imprécis car l'argument du sinus tend vers l'infini, alors que pour v_n il tend très vite vers 0.

```
from mpmath import mp
mp.dps=50

def u(n):
    return mp.sin(mp.pi*(2+mp.sqrt(3))**n)

def S(p):
    s=0
    for n in range(p+1):
        s+=u(n)
        print(n,' : ',s)
```

Noter que pour bénéficier de la multiprécision il faut utiliser `mp.pi`, `mp.sin` et `mp.sqrt` à la place de la constante et des fonctions habituelles !

12) *(ENSAM)* Pour obtenir un vecteur tangent, je fais subir un quart de tour au vecteur normal fourni par le gradient. Comme les expressions ici sont assez simples, je programme les formules exactes des dérivées partielles (on pourrait les approcher par un taux de variation...).

Voir les figures ci-dessous et les programmes sur la page suivante.

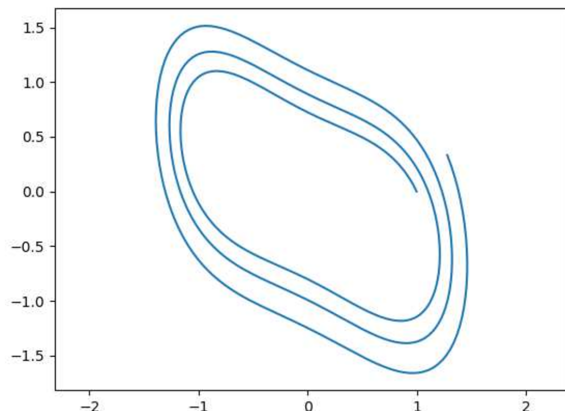
Je prends comme point de départ $A_0 = (1, 0)$ et j'obtiens une espèce de spirale déformée... (à gauche).

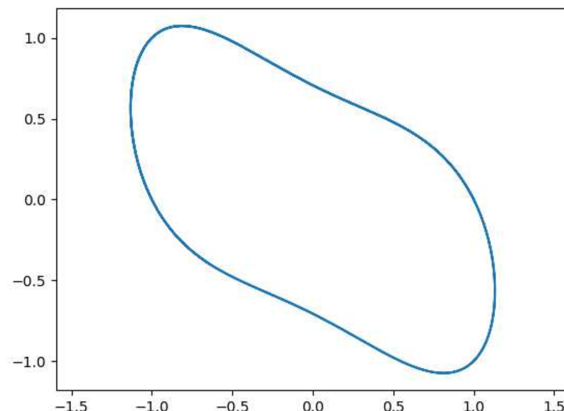
Pour améliorer la précision, je considère le développement limité vérifié par f, qui est de classe $\mathcal{C}^1$:

$$f(a+h) \underset{h\to 0}{=} f(a) + (\nabla f(a) \,|\, h) + o(h).$$

En prenant pour point a le point déplacé le long de la tangente (1re ligne de la boucle `for`), je choisis pour h un terme correctif le long de la normale en a à la courbe (dirigée par $\nabla f(a)$), en faisant en sorte de compenser le terme $f(a)$, qui n'est probablement par nul ! Il est immédiat que $h = -\dfrac{f(a)}{\|\nabla f(a)\|^2} \cdot \nabla f(a)$ convient.

Avec cette amélioration, j'obtiens une courbe fermée... (à droite).





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

def f(u):
    x, y = u
    return x**4 + 2*y**2 + 2*x*y - 1

def nablaf(u):
    x, y = u
    return np.array([4*x**3+2*y, 4*y+2*x])

def rot(v):
    x, y = v
    return np.array([-y, x])

def points(A0, pas, n):
    L = [A0]
    A = A0
    for k in range(n):
        A = A + pas*rot(nablaf(A))
        L.append(A)
    return L

A0 = np.array([1, 0])
A = points(A0, 0.01, 500)
X = [P[0] for P in A]
Y = [P[1] for P in A]
plt.plot(X, Y)
plt.axis('equal')
plt.show()
```

```
def n2(v):
    x, y = v
    return x**2 + y**2

def points(A0, pas, n):
    L = [A0]
    A = A0
    for k in range(n):
        A = A + pas*rot(nablaf(A))
        G = nablaf(A)
        A = A - (f(A)/n2(G))*G
        L.append(A)
    return L
```

13) *(Centrale)* Un piège à déjouer : il faut observer deux tirages consécutifs, mais après avoir examiné les deux premiers, il ne faut **surtout pas refaire** le 2e tirage pour le comparer au 3e !! Une solution pourrait être de remplir au départ un vecteur avec n résultats de tirages, mais c'est évidemment de la place mémoire et du temps perdus…

La relation de récurrence s'obtient par un raisonnement classique. Je note $A_{i,j}$ l'événement “ne jamais obtenir deux Pile d'affilée du tirage i inclus au tirage j exclu” (dans le style Python…). Ainsi $p_n = P(A_{0,n})$ (en numérotant les tirages à partir de 0). L'indépendance mutuelle des tirages fait que $P(A_{i,j})$ ne dépend que du nombre $j-i$ de tirages.

Soit F_i l'événement “le tirage i donne Face”. J'utilise de système complet d'événements $\left(F_0, \overline{F_0}\cap F_1, \overline{F_0}\cap\overline{F_1}\right)$ pour calculer p_{n+2} à l'aide de la formule des probabilités totales, pour $n\in\mathbb{N}$:

$$\begin{aligned} P(A_{0,n+2}) &= P(F_0)\,P_{F_0}(A_{0,n+2}) + P\left(\overline{F_0}\cap F_1\right)P_{\overline{F_0}\cap F_1}(A_{0,n+2}) + P\left(\overline{F_0}\cap\overline{F_1}\right)P_{\overline{F_0}\cap\overline{F_1}}(A_{0,n+2}) \\ &= (1-p)\,P(A_{1,n+2}) + p(1-p)\,P(A_{2,n+2}) + 0 \end{aligned}$$

soit, grâce à la remarque précédente :

$$\boxed{p_{n+2} = (1-p)\,p_{n+1} + p(1-p)\,p_n}$$

(ce qui permet d'expliciter (p_n) sachant que $p_0 = 1$ et $p_1 = 1-p^2$).

Nous avons donc une relation de récurrence linéaire double ; les solutions de l'équation caractéristique sont les racines du polynôme $Q = X^2 - (1-p)X - p(1-p)$. Q est de degré 2 et

$$Q(-1) = 2 - 2p + p^2 = 1 + (1-p)^2 > 0, \quad Q(0) = -p(1-p) < 0 \quad \text{et} \quad Q(1) = p^2 > 0$$

donc Q admet une racine dans $]-1,0[$ et une dans $]0,1[$.

En particulier, la suite (p_n) est une combinaison linéaire de deux suites géométriques qui convergent vers 0, donc (p_n) converge vers 0. Or l'événement A : “ne jamais obtenir deux Pile d'affilée” n'est autre que $\bigcap_{n\in\mathbb{N}^*} A_{0,n}$ et la suite $(A_{0,n})$ est décroissante par construction.

Ainsi, grâce à la continuité décroissante de la probabilité, $P(A) = 0$; autrement dit :

$$\boxed{\text{L'événement “obtenir deux Pile d'affilée” est presque certain.}}$$

Pour approcher p_n, j'effectue N simulations et je calcule la fréquence de survenue de l'événement $E_n = A_{0,n}$.

Par ailleurs le calcul de p_n grâce à la relation de récurrence ci-dessus s'effectue à l'aide d'une boucle banale (attention aux programmes récursifs de complexité exponentielle...).

```
import numpy as np

def E(n, p):
    if n < 2:
        return True
    else:
        P1 = np.random.rand() < p
        k = 1
        while k<n:
            P0, P1 = P1, np.random.rand() < p
            if P0 and P1: return False
            k += 1
        return True

def freqE(n, p, N):
    nb = 0
    for K in range(N):
        if E(n, p): nb += 1
    return nb/N

def pE(n, p):
    p0, p1 = 1, 1
    for k in range(1, n):
        p0, p1 = p1, p*(1-p)*p0 + (1-p)*p1
    return p1
```

La comparaison est probante pour N assez grand...

Avec les conventions précédentes, T prend ses valeurs dans $\mathbb{N}^*$ et, par construction, pour $n \geq 1$

$$P(T = n) = P(T > n-1) - P(T > n) = p_{n-1} - p_n$$

d'où la valeur en x de la fonction génératrice (au moins pour $x \in [-1, 1]$...)

$$G_T(x) = \sum_{n=1}^{\infty} (p_{n-1} - p_n) x^n$$

D'après le résultat précédent, $\sum p_n$ est absolument convergente, puisque c'est une combinaison linéaire de deux séries géométriques convergentes. Je peux donc couper la somme en deux, pour x fixé dans $[-1, 1]$:

$$G_T(x) = \sum_{n=1}^{\infty} p_{n-1} x^n - \sum_{n=1}^{\infty} p_n x^n.$$

Par ailleurs, en reprenant la relation de récurrence du début, en multipliant par x^{n+2} et en sommant (toutes les séries sont absolument convergentes) j'obtiens :

$$\sum_{n=0}^{\infty} p_{n+2} x^{n+2} = (1-p) \sum_{n=0}^{\infty} p_{n+1} x^{n+2} + p(1-p) \sum_{n=0}^{\infty} p_n x^{n+2}$$

soit, en notant $g(x) = \sum_{n=0}^{\infty} p_n x^n$, sachant que $p_0 = 1$ et $p_1 = 1 - p^2$, grâce à quelques réindexations,

$$g(x) - 1 - \left(1 - p^2\right) x = (1-p) x \left[g(x) - 1\right] + p(1-p) x^2 g(x)$$

c'est-à-dire

$$g(x) \left[1 - (1-p) x - p(1-p) x^2\right] = 1 + p(1-p) x$$

où $\varphi : x \mapsto 1 - (1-p) x (1 + px)$ est un polynôme de degré 2 admettant $-\infty$ pour limite en $\pm\infty$ et vérifiant

$$\varphi(-1) = 1 + (1-p)^2 > 0 \quad \text{et} \quad \varphi(1) = p^2 > 0.$$

Donc φ admet une racine α dans $]-\infty,-1[$, l'autre β dans $]1,+\infty[$ et φ est strictement positive sur $[-1,1]$. Ainsi

$$\forall x\in\,]\alpha,\beta[\quad g(x)=\frac{1+p(1-p)x}{\varphi(x)}$$

or après réindexation, compte tenu de $p_0=1$

$$G_T(x)=xg(x)-[g(x)-1]=1-(1-x)g(x)$$

soit finalement

$$\boxed{\forall x\in\,]\alpha,\beta[\quad G_T(x)=1-\frac{(1-x)(1+p(1-p)x)}{1-(1-p)x(1+px)}.}$$

Comme $1\in\,]\alpha,\beta[$, G_T est dérivable en 1 donc T est d'espérance finie et (inutile d'expliciter g' !)

$$E(T)=G_T'(1)=g(1)=\frac{1+p(1-p)}{p^2}$$

soit

$$\boxed{E(T)=\frac{1+p(1-p)}{p^2}.}$$

Pour les simulations, je reprends l'idée du début, mais en comptant le nombre de lancers jusqu'à l'obtention de deux Pile consécutifs.

```
def T(p):
    n = 1
    P0 = np.random.rand() < p
    P1 = np.random.rand() < p
    while not (P0 and P1):
        P0, P1 = P1, np.random.rand() < p
        n += 1
    return n

def moyT(p, N):
    s = 0
    for K in range(N):
        s += T(p)
    return s/N
```

Là encore, résultats probants !

14) a) Calculons la fonction génératrice de S, par indépendance des deux lancers :

$$\begin{aligned}G_S(x)&=\frac{1}{6}\left(x+2x^2+2x^3+x^4\right)\cdot\frac{1}{6}\left(x+x^3+x^4+x^5+x^6+x^8\right)\\&=\frac{1}{36}\left(x^2+2x^3+3x^4+4x^5+5x^6+6x^7+5x^8+4x^9+3x^{10}+2x^{11}+x^{12}\right)\end{aligned}$$

cela tous calculs faits, par exemple à l'aide du module `Polynomial` !).

On reconnaît la loi triangulaire, la même que pour le lancer de deux dés "classiques", dont les faces sont numérotées de 1 à 6 !! Sa fonction génératrice est en effet

$$\begin{aligned}G_T(x)&=\left[\frac{1}{6}\left(x+x^2+x^3+x^4+x^5+x^6\right)\right]^2\\&=\frac{1}{36}\left(x^2+2x^3+3x^4+4x^5+5x^6+6x^7+5x^8+4x^9+3x^{10}+2x^{11}+x^{12}\right)\end{aligned}$$

b) Le résultat précédent vient de deux répartitions possibles des facteurs irréductibles dans

$$X^2(X+1)^2\left(X^2+X+1\right)^2\left(X^2-X+1\right)^2.$$

En effet

$$X(X+1)\left(X^2+X+1\right)=X+2X^2+2X^3+X^4$$

et

$$X(X+1)\left(X^2+X+1\right)\left(X^2-X+1\right)^2=X+X^3+X^4+X^5+X^6+X^8$$

Pour savoir si d'autres répartitions sont possibles, il suffit de tester les 3^4 répartitions possibles…

En laissant les $\frac{1}{6}$ en facteur, je remarque qu'un produit correspond à la fonction génératrice associée à un dé équilibré à 6 faces portant chacune un entier naturel non nul si et seulement si son coefficient constant est nul et ses autres coefficients sont des entiers naturels de somme 6.

Voici une version naïve avec une quadruple boucle (qui permet d'accepter des 0 sur les faces des dés, en ôtant la condition que le coefficient soit nul dans le test d'un produit. . .).

```
from numpy.polynomial import Polynomial

L = [[0, 1], [1, 1], [1, 1, 1], [1, -1, 1]]
P = [Polynomial(lst) for lst in L]
nf = 6

def coeffs(P):
    return [int(c) for c in P.coef]

def test(P, nf):
    L = coeffs(P)
    if L[0] == 0 and sum(L) == nf:
        for c in L:
            if c < 0: return False
        return True
    else:
        return False

def prod(P, L):
    produit = Polynomial([1])
    for k in range(len(L)):
        produit = produit * P[k]**L[k]
    return produit

Lok = []
n = len(L)

for i0 in range(3):
    for i1 in range(3):
        for i2 in range(3):
            for i3 in range(3):
                La = [i0, i1, i2, i3]
                Lb = [2-i for i in La]
                if Lb not in Lok:
                    A = prod(P, La)
                    B = prod(P, Lb)
                    if test(A, nf) and test(B, nf):
                        Lok.append(La)
                        print(La, A)
                        print(Lb, B)
                        print('-'*20)
```

Dans les sources Python, le lecteur acharné trouvera une généralisation récursive pour d'autres types de dés, avec les factorisations similaires à celles ci-dessus, aimablement fournies par Maple. . .

Il en existe pour les dés "classiques" à 4, 6, 8, 10, 12 et 20 faces !