TD26

Révisions de Python

Programmation

Exercice 1 (★ - Sur la série harmonique)

1. Écrire un programme qui prend comme paramètre un entier n et retourne $u_n = \sum_{k=1}^{n} \frac{1}{k} - \ln(n)$.

   Vérifier alors numériquement que la suite (u_n) est convergente, et donner une valeur approchée de sa limite.

2. Écrire une fonction qui prend comme paramètre un réel $A > 0$ et qui retourne la valeur du plus petit entier n tel que $\sum_{k=1}^{n} \frac{1}{k} > A$.

   Quel est le plus petit entier n tel que $\sum_{k=1}^{n} \frac{1}{k} > 15$?

1. On peut proposer le programme suivant :

```
n = int(input('entier n'))
u = np.sum(np.arange(1,n+1)**(-1))-np.log(n)
print(u)
```

   On pouvait aussi effectuer une boucle `for` :

```
n = int(input('entier n'))
u = 0
for k in range(1,n+1):
      u = u+(1/k)
u = u-log(n)
print(u)
```

   On obtient les valeurs de u_n suivantes :

n	u_n
10	0.6263832
40	0.5896636
100	0.5822073
1000	0.5777156
10000	0.5772657

   La suite (u_n) semble converger vers une limite γ, dont une valeur approchée serait 0.577.

   Remarque. γ est appelé *constante d'Euler*. On ignore à peu près tout de γ, et notamment si elle est rationnel ou non. On connait en revanche, grâce à des algorithmes plus performants que le nôtre, les 119 milliards de premières décimales de γ.

2. On propose la fonction suivante :

```
def plus_petit_entier(A):
    u = 0
```

```
    n = 0
    while u <= A :
        n = n+1
        u = u+1/n
    return(n)
```

Dans cette fonction, la variable `u` contient le terme d'ordre `n` de la somme partielle de la série harmonique. Rappelons que la série harmonique diverge, et que cette somme partielle tend donc vers $+\infty$. La boucle `while` s'arrêtera donc bien.

Testons notre fonction :

```
>>>plus_petit_entier(15)
    1835421
```

Ainsi le plus petit entier n tel que $\sum_{k=1}^{n} \frac{1}{k} > 15$ est $n = 1835421$. Notons que cet entier est « grand » alors que le réel A ne l'était pas tant que ça. On retrouve ici que la série harmonique diverge, mais que sa vitesse de convergence est logarithmique, et donc particulièrement lente.

Exercice 2 (★★ - Série harmonique alternée)

On considère la série harmonique alternée $\sum_{n\geq 1} \frac{(-1)^{n-1}}{n}$. On note $S = (S_n)_{n\in\mathbb{N}^*}$ la suite de ses sommes partielles.

1. Écrire une commande définissant un vecteur ligne `u` tel que pour tout $1 \leq i \leq 50$, `u[i-1]` soit égal à $\frac{(-1)^{i-1}}{i}$.

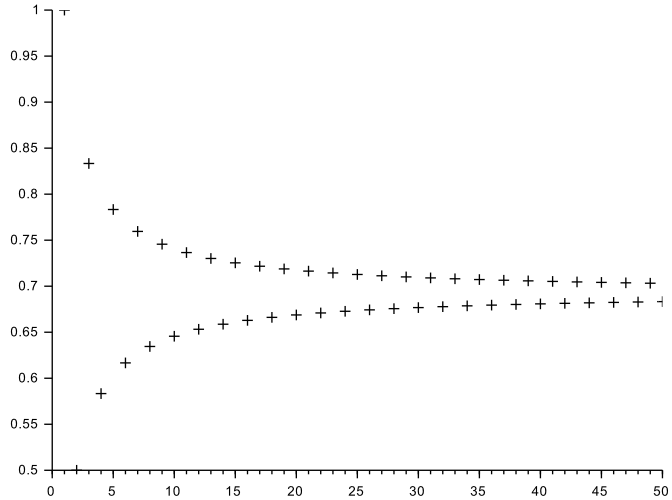
2. Écrire une commande définissant un vecteur ligne `v` tel que pour tout $1 \leq i \leq 50$, `v[i-1]` soit égal à $\sum_{k=1}^{i} \frac{(-1)^{k-1}}{k}$.

3. En exécutant la commande :

   `plt.plot(np.arange(1,51),v,'+')`

   on obtient le graphe ci-contre.

   Que peut-on dire des suites extraites (S_{2n}) et (S_{2n+1}) ? de la série S ?



4. On admet que $\sum_{n=1}^{+\infty} \frac{(-1)^{n-1}}{n} = \ln(2)$. Écrire un script demandant une valeur $\varepsilon > 0$ à l'utilisateur, et qui renvoie le plus petit entier naturel n pour lequel $|S_n - \ln(2)| \leq \varepsilon$.

1. On pouvait faire une boucle `for` éventuellement, ou directement :

```
u = (np.arange(1,51)**(-1))*((-1)**np.arange(50))
```

2. La commande suivante convient :

```
v = np.cumsum(u)
```

3. On voit que l'une des deux suites extraites est croissante, l'autre décroissante, et que la différence de ces deux suites tend vers 0. Donc les suites extraites paires et impaires sont adjacentes, et convergent vers la même limite. On peut aussi conjecturer que S converge vers cette limite commune.

4. On propose le script suivant :

```
eps = float(input('entrer un reel strict positif'))
s = 1
n = 1
while np.abs(s-np.log(2))> eps :
      n = n+1
      s = s + ((-1)**(n-1))/n
print(n)
```

Exercice 3 (★★)

Soit (u_n) la suite définie par récurrence par $u_0 = \dfrac{1}{2}$ et pour tout $n \in \mathbb{N}$, $u_{n+1} = u_n^2 - u_n \ln(u_n)$.

1. Écrire une fonction `suite` qui prend comme paramètre un entier n et renvoie la valeur de u_n correspondante.

2. On se propose de vérifier graphiquement que la suite (u_n) est croissante et tend vers 1.

   Écrire à cet effet un programme qui trace un graphique sur lequel se trouvent les points (n, u_n), pour $n \in [\![0, 100]\!]$.

3. Écrire une fonction nommée `plus_petit_n` qui prend comme paramètre un entier p et retourne la plus petite valeur de n pour laquelle $|1 - u_n| < 10^{-p}$.

1. On propose le code suivant :

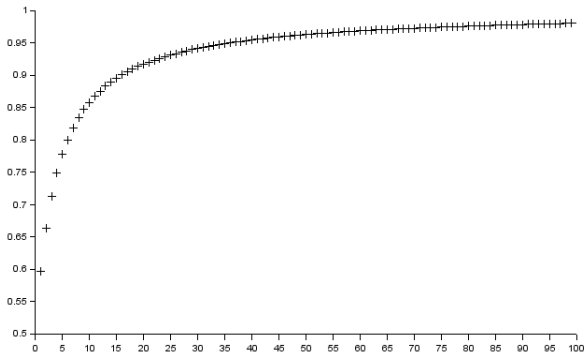
```
def suite(n):
    u = 1/2
    for k in range(n)
        u = u^2-u*log(u)
    return(u)
```

2. On utilise les lignes de commandes suivantes :

```
x = np.arange(101)
y = np.zeros(101)
for k in range(101):
```

```
        y[k] = suite(k)
plt.plot(x,y,'+')
```

En exécutant ce code, on obtient :



3. On utilise le code suivant :

```
def plus_petit_n(p)
    u = 1/2
    n = 0
    while np.abs(1-u)>=10**(-p):
        n = n+1
        u = u**2-u*np.log(u)
    return(p)
```

Exercice 4 (★★)

Soit (u_n) la suite définie par $u_0 = u_1 = u_2 = 1$ et pour tout $n \in \mathbb{N}$, $u_{n+3} = u_{n+2} - 2u_{n+1} + u_n$.
Écrire une fonction d'entête `def suite_rec(n)` qui prend comme paramètre $n \in \mathbb{N}$ et retourne la valeur de u_n.

On utilise la fonction suivante :

```
def suite_rec(n):
    u = 1
    v = 1
    w = 1
    for k in range(3,n+1):
        t = w-2*v+u
        u = v
        v = w
        w = t
    return(w)
```

Notons que ce programme fonctionne aussi pour $n = 0, 1, 2$ car alors il renvoie la valeur 1 pour ces trois cas.

Si la suite (u_n) était définie par $u_0 = 1$, $u_1 = 2$ et $u_2 = 3$ et par la même relation de récurrence, le programme aurait été un peu plus difficile, puisqu'il aurait fallu traiter à part les cas $n = 0, 1, 2$ à l'aide de boucles conditionnelles `if`. Cela donnerait alors :

```
def suite_rec(n):
    u = 1
    v = 2
    w = 3
    if n == 0 :
          y = 1
    else :
          if n == 1 :
                y = 2
          else :
                for k in range(3,n+1):
                      t = w-2*v+u
                      u = v
                      v = w
                      w = t
                y = w
    return(y)
```

Exercice 5 (★★)

Soient (a_n) et (b_n) les suites définies par $a_0 = 2$, $b_0 = 3$ et :

$$\forall n \in \mathbb{N}, \quad a_{n+1} = \sqrt{a_n b_n} \quad \text{et} \quad b_{n+1} = \frac{a_n + b_n}{2}.$$

Écrire une fonction d'entête `def suites(n)` prenant comme paramètre un entier n et retournant les valeurs de a_n et b_n correspondantes.
Constater que ces suites convergent vers une même limite.

On peut utiliser la fonction suivante.

```
def suites(n):
    a = 2
    b = 3
    for k in range(n)
        c = np.sqrt(a*b)
        b = (a+b)/2
        a = c
    return(np.array([a,b]))
```

On calcule a_n et b_n pour différentes valeurs de n à l'aide par exemple de l'instruction :

```
>>>suites(1)
    np.array([2.4494897,2.5])
```

On obtient :

n	a_n	b_n
1	2.4494897	2.5
2	2.474616	2.4747449
5	2.4746804	2.4746804
10	2.4746804	2.4746804

> Les suites (a_n) et (b_n) semblent en effet converger vers une limite commune, la convergence semblant très rapide.

Exercice 6 (★★★ - Extrait d'Edhec 2019)

Dans cet exercice, n désigne un entier naturel non nul.

1. On admet que, si a et b sont des entiers tels que $a < b$, la commande `rd.randint(a,b)` permet à `Python` de simuler une variable aléatoire suivant la loi uniforme discrète sur $[\![a, b-1]\!]$.
   Compléter le script suivant pour que les lignes (5), (6), (7) et (8) permettent d'échanger les contenus des variables $A(j)$ et $A(p)$.

```
1  n=int(input('entrez une valeur pour n :'))
2  A=np.arange(1,n+1)
3  p=n-1
4  for k=1:n
5          j=rd.randint(0,p+1)
6          aux=----
7          A[j]=----
8          A[p]=----
9          p=p-1
10 print(A)
```

2. On suppose dorénavant qu'après exécution du script précédent correctement complété, le vecteur A est rempli de façon aléatoire par les entiers de $[\![1, n]\!]$ de telle sorte que les $n!$ permutations soient équiprobables.
   On considère alors les commandes `Python` suivantes (exécutées à la suite du script précédent) :

```
1 m=A[0]
2 c=1
3 for k in range(1,n):
4     if A[k]>m :
5           m=A[k]
6           c=k
7 print(c)
```

   (a) Expliquer pourquoi, à la fin de la boucle `for`, la variable `m` contient la valeur n.
   (b) Quel est le contenu de la variable c affiché à la fin de ces commandes ?

On admet que les contenus des variables `A[0]`, `A[1]`, ..., `A[n-1]` sont des variables aléatoires notées A_1, A_2, ..., A_n et que le nombre d'affectations concernant la variable informatique c effectuées au cours du script présenté au début de la question 2., y compris la première, est aussi une variable aléatoire, notée X_n.
On suppose que ces variables aléatoires sont toutes définies sur le même espace probabilisé $(\Omega, \mathscr{A}, P)$.

3. Donner la loi de X_1.

4. (a) Montrer que $X_n(\Omega) = [\![1, n]\!]$.
   (b) Déterminer $P(X_n = 1)$ et $P(X_n = n)$. En déduire les lois de X_2 et X_3.
   (c) En considérant le système complet d'événements $((A_n = n), (A_n < n))$, montrer que :

$$\forall n \geq 2,\ \forall j \in [\![2, n]\!],\ P(X_n = j) = \frac{1}{n} P(X_{n-1} = j-1) + \frac{n-1}{n} P(X_{n-1} = j)$$

   (d) Donner la loi de X_4.

1. On veut permuter les contenus des variables $A(j)$ et $A(p)$, ce qui nécessite l'utilisation d'une variable auxiliaire `aux` pour stocker temporairement l'une de ces variables afin de faire la permutation. On peut procéder comme suit :

```
5 j = rd.randint(1,p+1)
6 aux = A[j]
7 A[j] = A[p]
8 A[p] = aux
```

2. (a) On commence par affecter à `m` le contenu de `A[0])`. Ensuite la boucle `for` permet de parcourir tout le vecteur `A`. Pour chaque coefficient `A[k]`, on le compare au contenu de la variable `m`, et donc à tous les coefficients `A[0]` jusqu'à `A[k-1]`. S'il est strictement supérieur à `m`, et donc à tous les coefficients `A[0]` jusqu'à `A[k-1]`, on affecte ce coefficient à `m`. Ainsi, à la fin de la boucle `for`, `m` contient le plus grand des coefficients du vecteur `A`.

   Le vecteur `A` étant une permutation des entiers de $[\![1, n]\!]$, son plus grand coefficient est n. Ainsi à la fin de la boucle `for`, la variable `m` contient la valeur n.

   (b) `c` contient l'indice `k` pour lequel `m = A[k]` : il s'agit de la position de l'entier n dans le vecteur `A`.

3. Lorsque $n = 1$, la boucle `for` n'est pas exécutée. Il n'y a donc qu'une seule affectation concernant la variable `c`, à savoir `c = 1` en ligne 2, et X_1 est la variable aléatoire certaine égale à 1.

4. (a) Le minimum du nombre d'affectations de `c` a lieu si, et seulement si, le maximum `n` figure en première place, cas où il y a l'affectation `c=1` et c'est tout (les tests de la boucle `for` retournent tous `false`). La valeur minimale prise par X_n est donc 1.

   Le maximum du nombre d'affectations de `c` a lieu si, et seulement si, tous les tests de la boucle `for` retournent `true`. Ceci est possible si, et seulement si, le vecteur `A` satisfait :

   `A[1]<A[2]<...<A[n]`.

   Or ceci est réalisé pour une unique permutation `A = np.array([1,2,...,n])`. Et pour cette permutation, il y a l'affectation `c=1` et une affectation par passage dans la boucle, ce qui en fait $1 + (n-1) = n$ affectations. Ainsi la valeur maximale prise par X_n est n.

   Les situations intermédiaires sont toutes possibles. En effet, pour $i \in [\![1, n]\!]$, cherchons une permutation `A` telle que les $(i-1)$ premiers tests de la boucle `for` donnent `true`, et les $(n-i)$ derniers donnent `false`, c'est-à-dire qui satisfait :

   `A[1]<...<A[i]` et `A[i+1],..., A[n]<A[i]`.

   La permutation suivante convient par exemple :

   `A = [n-i+1,n-i+1,...,n,1,2, ... , n-i]`

   Ainsi, $X_n = [\![1, n]\!]$.

   (b) On l'a dit précédemment, $[X_n = 1]$ si, et seulement si, le maximum `n` figure en première place dans la permutation `A`, soit avec les notations de l'énoncé $A_1 = n$. Or il y a $(n-1)!$ permutations de $[\![1, n]\!]$ débutant par n (on a $n-1$ choix possibles pour 2, tous

les entiers entre 1 et $n-1$, puis $n-2$ choix possibles pour 3, ... , puis 1 seul choix possible pour n, ce qui donne bien $(n-1)!$ permutations commençant par n), et $n!$ permutations de $[\![1,n]\!]$ en tout. Comme on est dans le cas équiprobable, on obtient :

$$P(X_n=1)=\frac{(n-1)!}{n!}=\frac{1}{n}.$$

On a également montré à la question précédente que $[X_n=n]$ si, et seulement si, `A = np.array([1,2,...,n])`. Ce qui donne, puisqu'on est dans le cas équiprobable :

$$P(X_n=n)=\frac{1}{n!}.$$

Pour la loi de X_2, on a $X_2(\Omega)=\{1,2\}$, et avec les calculs précédents :

$$P(X_2=1)=\frac{1}{2}\quad\text{et}\quad P(X_2=2)=\frac{1}{2!}=\frac{1}{2}.$$

Ainsi on a $X_2\hookrightarrow\mathscr{U}([\![1,2]\!])$.

Pour la loi de X_3, on a $X_3(\Omega)=\{1,2,3\}$, et :

$$P(X_3=1)=\frac{1}{3},\quad P(X_3=3)=\frac{1}{3!}=\frac{1}{6},$$

$$P(X_3=2)=1-P(X_3=1)-P(X_3=3)=1-\frac{1}{3}-\frac{1}{6}=\frac{1}{2}.$$

(c) Soit $n\geq 2$ et $j\in[\![2,n]\!]$. Les événements $[A_n=n]$ et $[A_n<n]$ forment un système complet d'événements. Par la formule des probabilités totales :

$$\begin{aligned}P(X_n=j)&=P([X_n=j]\cap[A_n=n])+P([X_n=j]\cap[A_n<n])\\&=P(A_n=n)P_{[A_n=n]}(X_n=j)+P(A_n<n)P_{[A_n<n]}(X_n=j)\end{aligned}$$

Réfléchissons à $P_{[A_n=n]}([X_n=j])$. Lorsque $[A_n=n]$ est réalisé, comme `m` vaudra n et sera obtenu pour le dernier passage dans la boucle `for`, il y aura une affectation pour `c` en dernier passage de la boucle `for`. X_n prend dans ce cas la valeur j si, et seulement si, il y a eu $j-1$ affectations à l'issue des $n-1$ premiers passages dans la boucle `for`, qui concernent en fait les permutations de $[\![1,n-1]\!]$. On a donc :

$$P_{[A_n=n]}(X_n=j)=P(X_{n-1}=j-1).$$

Enfin, il y a $(n-1)!$ permutations qui finissent par n sur les $n!$ permutations possibles, ce qui donne donc $P(A_n=n)=\frac{(n-1)!}{n!}=\frac{1}{n}$.

Pour $P_{[A_n<n]}([X_n=j])$, lorsque $[A_n<n]$ est réalisé, le maximum `m` (qui vaut n) aura été trouvé avant la position n du tableau, et sera obtenu avant le dernier passage dans la boucle `for`, il n'y aura donc aucune affectation pour `c` en dernier passage de la boucle `for`. X_n prend dans ce cas la valeur j si, et seulement si, il y a eu j affectations à l'issue des $n-1$ premiers passages dans la boucle `for`, qui concernent là aussi des permutations de $(n-1)$ valeurs. On a donc :

$$P_{[A_n<n]}(X_n=j)=P(X_{n-1}=j).$$

Enfin, $P(A_n<n)=1-P(A_n=n)=1-\frac{1}{n}$, ce qui donne finalement :

$$\forall n\geqslant 2,\ \forall j\in[\![2,n]\!],\quad P(X_n=j)=\frac{1}{n}P(X_{n-1}=j-1)+\frac{n-1}{n}P(X_{n-1}=j).$$

(d) On a vu que $X_4(\Omega) = \{1,2,3,4\}$ et $P(X_4 = 1) = \frac{1}{4}$ et $P(X_4 = 4) = \frac{1}{24}$. Par 4.(c), on obtient :

$$\begin{aligned} P(X_4 = 2) &= \frac{1}{4}P(X_3 = 1) + \frac{3}{4}P(X_3 = 2) = \frac{1}{4} \times \frac{1}{3} + \frac{3}{4} \times \frac{1}{2} = \frac{11}{24}, \\ P(X_4 = 3) &= \frac{1}{4}P(X_3 = 2) + \frac{3}{4}P(X_3 = 3) = \frac{1}{4} \times \frac{1}{2} + \frac{3}{4} \times \frac{1}{6} = \frac{1}{4}. \end{aligned}$$

Simulation de variables aléatoires

Exercice 7 (★)

En utilisant `rd.random()`, mais sans `rd.binomial`, écrire une fonction d'entête `def binomiale(n,p)` qui simule une réalisation d'une loi $\mathscr{B}(n,p)$.
Indication : on se rappellera qu'une loi binomiale correspond au nombre de succès lors d'une répétition d'épreuves de Bernoulli indépendantes.

Rappelons tout d'abord comment simuler une variable de Bernoulli $\mathscr{B}(p)$ à l'aide de la fonction `rd.random()`. Il s'agit essentiellement de proposer un test ayant une probabilité p de succès, et de renvoyer 1 en cas de succès et 0 en cas d'échec. On peut procéder ainsi :

```
X = 0
if rd.random()<=p :
      X=1
print(X)
```

Le programme retourne la valeur 1 si le test dans la boucle `if` est satisfait, ce qui a lieu avec la probabilité :

$$P(U \leq p) = F_U(p) \underset{p\in[0,1]}{=} p$$

où $U \hookrightarrow \mathscr{U}([0,1])$. Ce qui correspond donc bien effectivement à simuler la loi $\mathscr{B}(p)$.

Pour obtenir une simulation d'une loi $\mathscr{B}(n,p)$, il reste à répéter n fois ce test, à l'aide d'une boucle `for` par exemple, et à sommer le nombre de succès. On obtient la fonction suivante :

```
def binomiale(n,p) :
      y = 0
      for k in range(n)
            if rand()<=p :
                  y = y + 1
      return y
```

On pouvait également tout faire en une ligne :

```
def binomiale(n,p) :
      return np.sum(rd.random(n)<=p)
```

En effet, `rd.random(n)<=p` est un vecteur de taille n contenant uniquement des booléens, chaque composante étant égale à `True` lorsque `rd.random()<=p` est réalisé, et à `False` sinon. Quand on appliquer `np.sum` à un tel vecteur, les booléens `True` sont vus comme des 1, `False` comme des 0. On obtient donc le nombre de réalisations lors d'une répétition (de manière indépendante) d'une épreuve ayant une probabilité p de réussite.

Exercice 8 (★★)

1. Compléter la fonction suivante pour qu'elle simule une réalisation d'une loi géométrique de paramètre p.

```
def geom(p):
      y = 1
      while rd.random() ----- :
            y = -----
      return( ----- )
```

2. On dit qu'une variable suit la loi binomiale négative de paramètres n et p si elle a la même loi que $\sum_{i=1}^{n} X_i$ où les X_i sont des variables i.i.d. suivant la loi $\mathscr{G}(p)$.

   Écrire un programme d'entête `def bin_neg(n,p)` qui simule une réalisation d'une loi binomiale négative de paramètres n et p.

1. Rappelons qu'une loi géométrique de paramètre p correspond au rang du premier succès dans une succession d'épreuves de Bernoulli indépendantes et identiques (lancer d'une pièce avec probabilité p de faire « pile »).

   Pour simuler une loi géométrique de paramètre p, on va donc répéter une même épreuve qui a une probabilité $1-p$ d'être réalisée (ce qui correspondra à obtenir « face ») et on s'arrêtera dès que cette épreuve ne se réalisera pas (ce qui correspondra à obtenir « pile » et qui se produit avec probabilité p).

   Reste donc à déterminer quelle épreuve utiliser, qui doit avoir une probabilité $1-p$ de se réaliser. Pour cela, on nous suggère d'utiliser la fonction `rd.random()` qui renvoie une réalisation d'une variable aléatoire U suivant la loi $\mathscr{U}([0,1])$. On remarque que :

$$P(U \leq 1-p) = F_U(1-p) \underset{1-p\in[0,1]}{=} 1-p.$$

   Ainsi la condition `rd.random()<=1-p` se réalise avec une probabilité $1-p$.

   On peut donc compléter notre programme, en augmentant y de 1 à chaque fois que la condition `rd.random()<=1-p` est réalisée. On obtient :

```
def geom(p):
      y = 1
      while rd.random()<=1-p :
            y = y+1
      return y
```

   Remarque. On aurait pu utiliser `rd.random()>p` comme condition d'arrêt de la boucle `while` puisque :

$$P(U > p) = 1 - P(U \leq p) = 1 - F_U(p) \underset{p\in[0,1]}{=} 1-p.$$

2. On doit simuler n réalisations d'une loi $\mathscr{G}(p)$, et les sommer. On peut utiliser le code suivant :

```
def bin_neg(n,p):
      y = 0
      for k in range(n) :
            y = y + geom(p)
      return y
```

Exercice 9 (★★)

Une urne contient initialement des boules numérotées de 2 à n. On effectue un tirage dans cette urne, et on enlève de l'urne toutes les boules portant un numéro supérieur ou égal à celui de la boule tirée. On ajoute alors la boule numéro 1 dans l'urne, et on effectue un nouveau tirage, et on note X le numéro de la boule obtenue.
Écrire un programme qui simule la variable aléatoire X.

Le programme doit prendre en entrée un entier n supérieur à 2, et retourner un entier X. On utilisera pour cela les fonctions d'entrée `input` et de sortie `disp`.

Les tirages étant équiprobables, il nous faut simuler des lois uniformes. Rappelons qu'on simule une loi uniforme discrète $\mathscr{U}(\llbracket a, b-1 \rrbracket)$ à l'aide de la commande `rd.randin(a,b)`.

Reste maintenant à traduire informatiquement les différentes étapes de l'expérience. On propose le programme suivant.

```
n = int(input('entier plus grand que 2'))
p = rd.randint(2,n+1) # premier tirage d'une boule de 2 à n
X = rd.randint(1,p) # deuxième tirage dans une urne contenant les boules de 1 à p-1
print(X)
```

Il est possible de se passer de `rd.randint` en utilisant l'astuce suivante, donnée dans le TP4 : `a + np.floor((b-a+1)*rd.random())` simule une loi uniforme sur $\llbracket a, b \rrbracket$.

```
n = int(input('entier plus grand que 2'))
p = 2+np.floor((n-1)*rd.random())
X = 1+np.floor((p-1)*rd.random())
print(X)
```

Exercice 10 (★★ - Ecricome 2016)

Soient a, b deux entiers strictement positifs. Une urne contient initialement a boules rouges et b boules blanches. On effectue une succession d'épreuves, chaque épreuve étant constituée des trois étapes suivantes :

- on pioche une boule au hasard dans l'urne,
- on replace la boule tirée dans l'urne,
- on rajoute dans l'urne une boule de la même couleur que celle qui vient d'être piochée.

Après n épreuves, l'urne contient donc $a+b+n$ boules. Pour tout $n \in \mathbb{N}^*$, on note X_n le nombre de boules rouges qui ont été **ajoutées** dans l'urne (par rapport à la composition initiale) à l'issue des n premières épreuves.

On souhaite simuler l'expérience grâce à `Python`.

1. Compléter la fonction suivante, qui simule le tirage d'une boule dans une urne contenant x boules rouges et y boules blanches et qui retourne la valeur 0 si la boule est rouge et 1 si elle est blanche.

```
def tirage(x,y):
    r = rd.random()
    if ..... :
        res = 0
    else :
        res = 1
    return res
```

2. Compléter la fonction suivante, qui simule n tirages successifs dans une urne contenant initialement a boules rouges et b boules blanches (selon le protocole décrit ci-dessus) et qui retourne la valeur de X_n :

```
def experience(a,b,n):
    x = a
    y = b
    for k in range(n):
        r = tirage(x,y)
        if r == 0 :
            x = .........
        else :
            ............
    Xn = .........
    return Xn
```

3. Écrire une fonction `simulation(a,b,n,m)` qui fait appel `m` fois à la fonction précédente pour estimer la loi de X_n. Le paramètre de sortie sera un vecteur à $n+1$ composantes contenant les approximations de $P(X_n = 0)$, $P(X_n = 1), \ldots, P(X_n = n)$.

4. On s'intéresse au cas où $a = b = 1$. On rappelle les commandes suivantes :

   - Si `x` et `y` sont des vecteurs de même taille, `plt.bar(x,y)` trace le diagramme en bâtons d'abscisses contenues dans `x` et d'ordonnées dans `y`.
   - `plt.figure(n)` ouvre une fenêtre graphique et trace la figure `n` dans celle-ci.

   On exécute le code suivant :

```
for n in range(1,6):
    x = np.arange(n+1)
    y = simulation(1,1,n,100000)
    plt.figure(n)
    plt.bar(x,y)
plt.show()
```

   On obtient les figures suivantes :

Figure 1

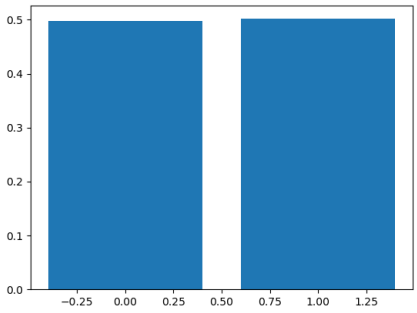


Figure 2

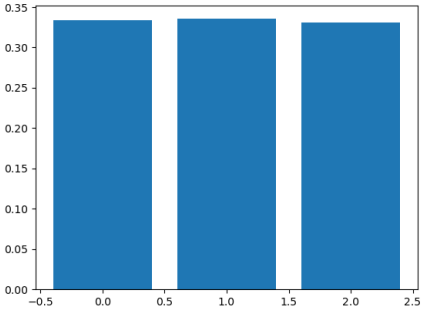


Figure 3

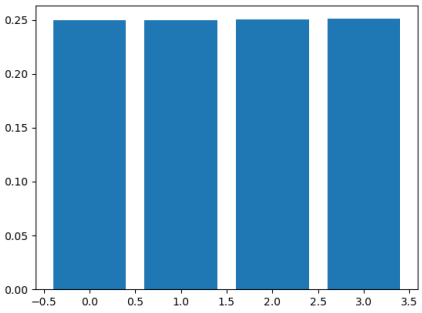


Figure 4

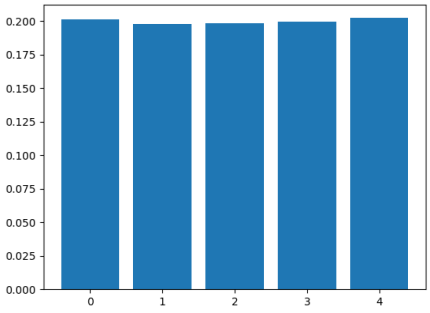
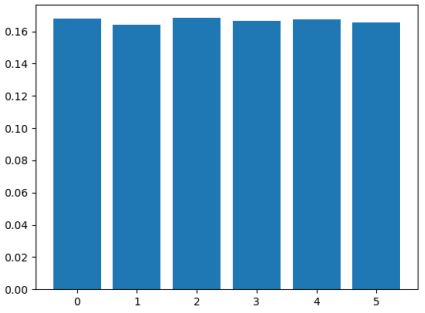


Figure 5



À l'aide de ces résultats, conjecturer la loi de X_n.

1. On a une probabilité de $\frac{x}{x+y}$ de tirer une boule rouge. Donc le programme doit retourner 0 avec une probabilité $\frac{x}{x+y}$. Il faut donc que la condition soit satisfaite avec cette probabilité. On va pour cela prendre comme condition $r \leq \dfrac{x}{x+y}$ qui est satisfaite avec une probabilité de $P(R \leq \frac{x}{x+y}) \underset{\frac{x}{x+y} \in [0,1]}{=} \frac{x}{x+y}$ où $R \hookrightarrow \mathscr{U}([0,1])$.

```
def tirage(x,y):
    r = rd.random()
    if r <= x/(x+y) :
        res = 0
    else :
        res = 1
    return res
```

2. Si $r = 0$ alors on ajoute une rouge sinon on ajoute une blanche, d'où `x = x+1` ou `y = y+1` ; le nombre de rouges ajoutées est donc la différence entre le nombre de boules rouges x au final et le nombre de boules rouges a au départ, soit `Xn = x-a`. Ainsi on a :

```
def experience(a,b,n):
    x = a
    y = b
    for k in range(n):
        r = tirage(x,y)
        if r == 0 :
            x = x+1 #on a tiré une boule rouge
        else :
```

```
            y = y+1 #on a tiré une boule blanche
    Xn = x-a
    return Xn
```

3. On va stocker (une approximation de) la loi de X_n dans la variable `loi`. Commençons par noter que $X_n(\Omega) = [\![0, n]\!]$, et qu'il nous faut une approximation des probabilités $P(X_n = k)$ pour tout $k = 0, \dots, n$. L'idée est habituelle : on répète l'expérience un grand nombre de fois (ici `m` fois), et on observe la fréquence de chaque issue, qui donnera une approximation de la probabilité théorique. On la stocke alors dans la k-ème composante `loi[k]` du vecteur `loi`. On renvoie alors ce vecteur.

   Voici comment on va procéder plus concrètement :

   - On initialise la variable `loi` en lui affectant un vecteur avec que des 0 à $n+1$ composantes (numérotées de 0 à n), à l'aide de la commande `loi = np.zeros(n+1)`.
   - On répète l'expérience `m` fois à l'aide d'une boucle `for`. Pour chaque résultat `r = experience(a,b,n)` de l'expérience, on ajoute 1 à la r-ème composante du vecteur `loi`.
     Ainsi, à l'issue de la boucle `for`, le vecteur `loi` contient en k-ème composante, le nombre d'expériences (parmi les `m` effectuées) renvoyant l'issue `k`, c'est-à-dire le nombre d'expériences pour lesquelles k boules rouges ont été ajoutées à l'urne.
   - Il reste alors à diviser par `m` le vecteur `loi` pour obtenir la fréquence de chaque issue, et donc une approximation de $P(X_n = k)$.

   Voici une possibilité de programme :

```
def simulation(a,b,n,m):
    loi = np.zeros(n+1) # initialisation
    for k in range(m):
        r = experience(a,b,n) # on effectue l'expérience
        loi[r] = loi[r]+1 # +1 à l'effectif de l'issue r
    loi = loi/m # pour obtenir les fréquences
    return loi
```

4. La distribution des fréquences semble équiprobable, donc on peut conjecturer que $X_n \hookrightarrow \mathscr{U}([\![0, n]\!])$.

   Remarque. Ce résultat était ensuite démontré dans la suite du sujet d'Ecricome.

Exercice 11 (★★)

On lance n fois une pièce équilibrée, et on note X le nombre de fois où l'on obtient deux résultats identiques consécutifs.
Écrire un programme qui simule la variable aléatoire X. Utiliser ensuite ce programme pour estimer la valeur de $E(X)$.

Le plus simple est probablement de créer un grand vecteur contenant directement les résultats des n lancers, puis de chercher combien de fois ce vecteur contient deux coefficients successifs identiques. On va exceptionnellement écrire une fonction afin de pouvoir l'exploiter ensuite pour estimer la valeur de $E(X)$.

```
def simulation(n):
```

```
    lancers = rd.binomial(1,0.5,n)
    x = 0
    for k in range(n-1) :
        if lancers(k)==lancers(k+1) :
            x = x+1
    return x
```

Pour estimer la valeur de $E(X)$, on va utiliser la loi faible des grands nombre : on génère un échantillon observé de la loi de X de grande taille ($m = 10000$ par exemple), et on renvoie la moyenne de cet échantillon à l'aide de la commande `np.mean`.

```
n = int(input('entrer un entier'))
m = 10000
X = np.zeros(m)
for k in range(m) :
     X[k] = simulation(n)
print np.mean(X)
```

Pour $n = 10$, on obtient par exemple `0.0036`.

Exercice 12 (★★)

Si $(X_1, \dots, X_n)$ sont des variables aléatoires indépendantes suivant la loi $\mathcal{N}(0,1)$, la loi suivie par la variable aléatoire $\sum_{i=1}^{n} X_i^2$ est appelée loi du χ^2 (prononcer « chi-deux ») de paramètre n.

1. Écrire une fonction `def chi2(n)` qui prend en paramètre un entier $n \geq 1$, et qui simule une variable suivant la loi $\chi^2(n)$.
2. Écrire une fonction qui simule la variable T_p , où $T_p = \max(Y_1, \dots, Y_p)$, où $Y_1, \dots, Y_p$ sont p variables aléatoires indépendantes suivant la loi $\chi^2(n)$.
3. Proposer une méthode pour obtenir une valeur approchée de $E(T_p)$.

1. Il suffit de simuler n lois normales centrées réduites, et d'ajouter leurs carrés.

```
def chi_deux(n):
      y = 0
      for i in range(n) :
            y = y + rd.normal(0,1)**2
      return y
```

Il y avait aussi une solution plus rapide :

```
def chi_deux(n):
      return np.sum(rd.normal(0,1,n)**2)
```

2. Il nous suffit de faire appel p fois à la fonction `chi_deux`, et de ne garder que le plus grand résultat.

```
def T(p,n):
      y = chi_deux(n)
      for i in range(1,p):
            y = max(y,chi_deux(n))
```

```
5       return y
```

On compare ici chaque nouvelle réalisation de la loi du $\chi^2(n)$ aux précédentes. Une autre méthode consisterait à créer un vecteur contenant p réalisations de la loi du $\chi^2(n)$ et à en prendre le maximum, ce qui donne :

```
def T(p,n):
      t = np.zeros(p)
      for i in range(p):
            t[i] = chi_deux(n)
      return np.max(t)
```

3. On fait comme d'habitude : on réalise un grand nombre de simulations de T_p, et on calcule la moyenne de ces simulations !

```
n = int(input('choisir la valeur de n'))
p = int(input('choisir la valeur de p'))
m = 10000
c = 0
for i in range(m)
      c = c + T(p,n)
return c/m
```

On pouvait aussi utiliser la commande `np.mean` comme suit :

```
n = int(input('choisir la valeur de n'))
p = int(input('choisir la valeur de p'))
m = 10000
c = np.zeros(m)
for i in range(m):
      c[i] = T(p,n)
print(np.mean(c))
```

Exercice 13 (★★ - Loi de Rayleigh)

1. Soit $\sigma > 0$. Montrer que la fonction $f : x \mapsto \begin{cases} \frac{1}{\sigma} x e^{-\frac{x^2}{2\sigma}} & \text{si } x \geq 0 \\ 0 & \text{sinon.} \end{cases}$ est une densité de probabilité.

La loi d'une variable aléatoire admettant une telle densité est appelée loi de Rayleigh de paramètre σ. Dans toute la suite, on note X une variable aléatoire suivant une telle loi.

2. Donner la fonction de répartition F_X de X.

3. (a) Montrer que F_X réalise une bijection de $\mathbb{R}_+^*$ dans $]0,1[$. Déterminer une expression explicite de F_X^{-1}.

   (b) Soit U une variable aléatoire suivant une loi $\mathscr{U}(]0,1[)$. Montrer que $F_X^{-1}(U)$ suit la même loi que X.

   (c) Écrire une fonction `Python` d'entête `def rayleigh(sigma,n)` qui, étant donné un réel $\sigma > 0$ et un entier $n \in \mathbb{N}^*$, simule n réalisation d'une loi de Rayleigh de paramètre σ.

   (d) Vérifier la pertinence de cette simulation en comparant l'histogramme des fréquences et la densité, puis les fonctions de répartitions empirique et théorique.

4. Estimer numériquement l'espérance et l'écart-type d'une variable aléatoire suivant une loi de Rayleigh de paramètre σ. Retrouver ces résultats par le calcul.

5. (a) Si X suit une loi de Rayleigh de paramètre σ, quelle est la loi de $Y = X^2$.

   (b) À l'aide de la question précédente, proposer une nouvelle fonction `Python` d'entête `def rayleigh2(sigma)` qui, étant donné un réel $\sigma > 0$, simule une réalisation d'une loi de Rayleigh de paramètre σ.

1. Montrons que $f : x \mapsto \begin{cases} \dfrac{1}{\sigma} x e^{-\frac{x^2}{2\sigma}} & \text{si } x \geq 0 \\ 0 & \text{sinon.} \end{cases}$ est une densité de probabilité.

   - f est positive car pour tout $\dfrac{1}{\sigma} x e^{-\frac{x^2}{2\sigma}} \geq 0$ pour tout $x \geq 0$.
   - f est continue sur $\mathbb{R}_+$ comme composée de fonctions qui le sont. Donc f est continue sur $\mathbb{R}$ sauf éventuellement en 0.
   - On étudie la convergence de l'intégrale $\displaystyle\int_{-\infty}^{+\infty} f(t)\,\mathrm{d}t = \int_0^{+\infty} \frac{1}{\sigma} t e^{-\frac{t^2}{2\sigma}}\,\mathrm{d}t$. La fonction $t \mapsto \dfrac{1}{\sigma} t e^{-\frac{t^2}{2\sigma}}$ étant continue sur $\mathbb{R}_+$, l'intégrale est généralisée en $+\infty$. Pour tout $x \geq 0$, on a :
   $$\int_0^x \frac{1}{\sigma} t e^{-\frac{t^2}{2\sigma}}\,\mathrm{d}t = \left[-e^{-\frac{t^2}{2\sigma}}\right]_0^x = 1 - e^{-\frac{x^2}{2\sigma}}$$
   Et $\displaystyle\lim_{x\to+\infty} 1 - e^{-\frac{x^2}{2\sigma}}$ existe et vaut 1. Donc $\displaystyle\int_0^{+\infty} \frac{1}{\sigma} t e^{-\frac{t^2}{2\sigma}}\,\mathrm{d}t$ converge et vaut 1.

   Donc f est bien une densité de probabilité.

2. Pour tout $x \in \mathbb{R}$, on a :

   - si $x < 0$, $F_X(x) = \displaystyle\int_{-\infty}^x f(t)\,\mathrm{d}t = \int_{-\infty}^x 0\,\mathrm{d}t = 0$.
   - si $x \geq 0$, $F_X(x) = \displaystyle\int_{-\infty}^x f(t)\,\mathrm{d}t = \int_0^x \frac{1}{\sigma} t e^{-\frac{t^2}{2\sigma}}\,\mathrm{d}t = 1 - e^{-\frac{x^2}{2\sigma}}$.

   La fonction de répartition de X est donc $F_X : x \mapsto \begin{cases} 1 - e^{-\frac{x^2}{2\sigma}} & \text{si } x \geq 0 \\ 0 & \text{si } x < 0 \end{cases}$.

3. (a) F_X est continue sur $\mathbb{R}_+^*$ (elle est même de classe $\mathscr{C}^1$ en tant que primitive de f qui est continue sur $\mathbb{R}_+^*$) et strictement croissante car $F_X'(x) = f(x) > 0$ sur $\mathbb{R}_+^*$). De plus on a $\displaystyle\lim_{x\to0} F_X(x) = 0$ et $\displaystyle\lim_{x\to+\infty} F_X(x) = 1$. F_X réalise donc une bijection de $]0, +\infty[$ sur $]0, 1[$.

   Pour tout $x \in \mathbb{R}_+^*$ et pour tout $y \in]0, 1[$, on résout :
   $$\begin{aligned} y = F_X(x) &\Leftrightarrow y = 1 - e^{-\frac{x^2}{2\sigma}} \Leftrightarrow e^{-\frac{x^2}{2\sigma}} = 1 - y \\ &\underset{1-y\in]0,1[}{\Leftrightarrow} -\frac{x^2}{2\sigma} = \ln(1-y) \\ &\Leftrightarrow x^2 = -2\sigma\ln(1-y) \\ &\underset{x>0}{\Leftrightarrow} x = \sqrt{-2\sigma\ln(1-y)} \end{aligned}$$
   Ainsi on a pour tout $y \in]0, 1[$, $F_X^{-1}(y) = \sqrt{-2\sigma\ln(1-y)}$

   (b) Soit $U \hookrightarrow \mathscr{U}(]0, 1[)$, et posons $Y = F_X^{-1}(U)$. Comme $U \in]0, 1[$ et que $F_X^{-1} :]0, 1[\to \mathbb{R}_+^*$, on a $Y(\Omega) = \mathbb{R}_+^*$. En particulier, on a pour tout $x \leq 0$, $F_Y(x) = 0$.

Supposons $x > 0$. On a :

$$F_Y(x) = P(F_X^{-1}(U) \leq x) \underset{F_X \text{ croiss.}}{=} P(U \leq F_X(x)) = F_U(F_X(x)) = F_X(x)$$

car $F_X(x) \in [0,1]$. Ainsi Y et X ont même fonction de répartition, et suivent donc la même loi.

(c) Il s'agit ici d'une application de la méthode d'inversion qu'on avait abordée dans le **TP5 - Simulation de variables à densité**.

```
def rayleigh(sigma,n):
      u = rd.random(n)
      v = np.sqrt(-2*sigma*np.log(1-u))
      return v
```

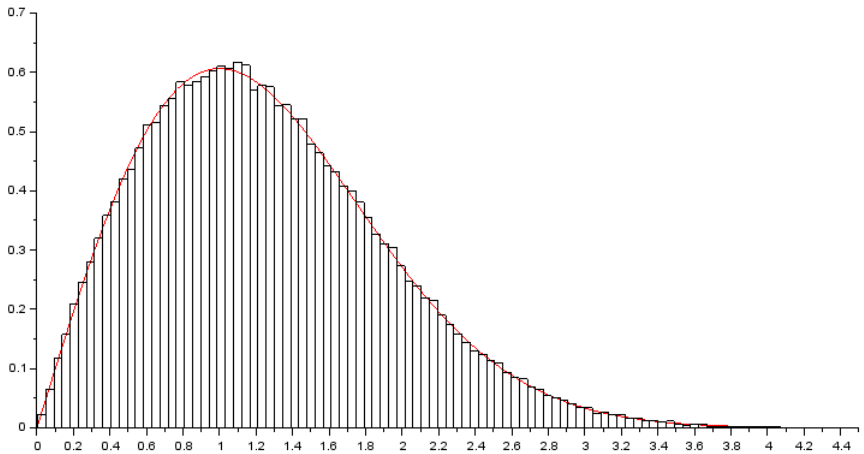
(d) Commençons par comparer histogramme des fréquences et densité lorsque $\sigma = 1$. On utilise pour cela le code suivant (je vous renvoie une nouvelle fois au TP5).

```
s = 1
# tracé de la densité
def f(x,s):
      y = (x/s)*np.exp(-(x**2)/(2*s))
      return y

x = np.linspace(0,4,100)
y = f(x,s)
plt.plot(x,y)
plt.show()

# tracé de l'histogramme des fréquences
n = 100000
e = rayleigh(s,n)
plt.hist(e,100)
```

Voici ce qu'on obtient :



L'histogramme des fréquences est « proche » de la densité, donc notre simulation semble pertinente pour $\sigma = 1$.

Comparons maintenant fonction de répartition théorique et empirique.

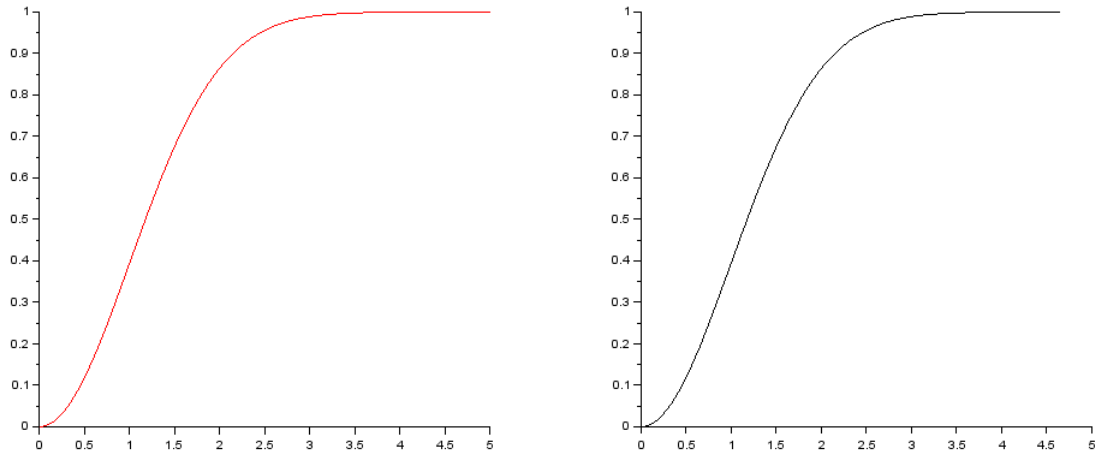
```
s=1
# Fonction de répartition théorique
def F(x):
    y = 1-np.exp(-x**2/(2*s))
    return y

x = np.linspace(0,5,100)
plt.subplot(1,2,1)
y = F(x)
plt.plot(x,y)

# Fonction de répartition empirique
n = 100000
e = rayleigh(s,n)
plt.subplot(1,2,2)
plt.step(np.sort(e),np.arange(0,1,1/n))

plt.show()
```

Voici ce qu'on obtient :



Là aussi, les fonctions de répartitions empiriques et théoriques sont « proches », donc notre simulation semble pertinente pour $\sigma = 1$.

4. Pour estimer numériquement espérance et écart-type, on utilise les estimateurs de moyenne empirique et d'écart-type empirique étudiés en TD :

$$\overline{X_n} = \frac{1}{n}\sum_{i=1}^{n} X_i \quad \text{et} \quad S'_n = \sqrt{\frac{1}{n-1}\sum_{i=1}^{n}(X_i - \overline{X_n})^2}.$$

On utilise les commandes suivantes :

```
>>> np.mean(rayleigh(1,100000)) ; np.std(rayleigh(1,100000))
```

Les valeurs renvoyées sont des estimations de $E(X)$ et de $\sigma(X)$ pour $\sigma = 1$. On obtient `1.2505934` et `0.6556909`.

$E(X)$ existe si et seulement si $\displaystyle\int_{-\infty}^{+\infty} tf(t)\,\mathrm{d}t = \int_0^{+\infty} \frac{t^2}{\sigma} e^{-\frac{t^2}{2\sigma}}\,\mathrm{d}t$ converge absolument, donc converge puisque la fonction intégrée est positive. Cette fonction est continue sur $\mathbb{R}_+$. On procède à une intégration par parties sur le segment $[0, A]$ où $A > 0$:

$$\begin{array}{c|ccc} + & t & & \frac{t}{\sigma}e^{-\frac{t^2}{2\sigma}} \\ & & \searrow & \\ - & 1 & \xleftarrow{\int} & -e^{-\frac{t^2}{2\sigma}} \end{array}$$

Les fonctions $t \mapsto t$ et $t \mapsto \frac{t}{\sigma}e^{-\frac{t^2}{2\sigma}}$ sont de classe $\mathscr{C}^1$. On obtient que pour tout $A > 0$:

$$\begin{aligned} \int_0^A t^2 f(t)\,\mathrm{d}t &= \left[-te^{-\frac{t^2}{2\sigma}}\right]_0^A + \int_0^A e^{-\frac{t^2}{2\sigma}}\,\mathrm{d}t \\ &= -Ae^{-\frac{A^2}{2\sigma}} + \int_0^A e^{-\frac{t^2}{2\sigma}}\,\mathrm{d}t \end{aligned}$$

On a $\lim\limits_{A\to+\infty} Ae^{-\frac{A^2}{2\sigma}} = 0$ par croissances comparées. D'autre part, l'intégrale $\sqrt{2\pi\sigma}\displaystyle\int_0^{+\infty} \frac{1}{\sqrt{2\pi\sigma}} e^{-\frac{t^2}{2\sigma}}\,\mathrm{d}t$ converge et vaut $\dfrac{\sqrt{2\pi\sigma}}{2}$ (en reconnaissant la densité de la loi normale $\mathscr{N}(0, \sigma)$). On en déduit que $E(X)$ existe bien et vaut donc :

$$E(X) = \frac{\sqrt{2\pi\sigma}}{2} = \sqrt{\frac{\pi\sigma}{2}}.$$

Pour $\sigma = 1$, on trouve $E(X) \approx 1.2533141$, ce qui correspond à la valeur numérique attendue.

Calculons enfin la variance de X. $E(X^2)$ existe si et seulement si $\displaystyle\int_{-\infty}^{+\infty} t^2 f(t)\,\mathrm{d}t = \int_0^{+\infty} \frac{t^3}{\sigma} e^{-\frac{t^2}{2\sigma}}\,\mathrm{d}t$ converge absolument, donc converge puisque la fonction intégrée est positive. Cette fonction est également continue sur $\mathbb{R}_+$. On procède à une intégration par parties sur le segment $[0, A]$ avec $A > 0$:

$$\begin{array}{c|ccc} + & t^2 & & \frac{t}{\sigma}e^{-\frac{t^2}{2\sigma}} \\ & & \searrow & \\ - & 2t & \xleftarrow{\int} & -e^{-\frac{t^2}{2\sigma}} \end{array}$$

Les fonctions $t \mapsto t^2$ et $t \mapsto \frac{t}{\sigma}e^{-\frac{t^2}{2\sigma}}$ sont de classe $\mathscr{C}^1$. On obtient que pour tout $A > 0$

$$\begin{aligned} \int_0^A t^2 f(t)\,\mathrm{d}t &= \left[-t^2 e^{-\frac{t^2}{2\sigma}}\right]_0^A + 2\int_0^A te^{-\frac{t^2}{2\sigma}}\,\mathrm{d}t \\ &= -A^2 e^{-\frac{A^2}{2\sigma}} + 2\int_0^A te^{-\frac{t^2}{2\sigma}}\,\mathrm{d}t \end{aligned}$$

On a $\lim\limits_{A\to+\infty} A^2 e^{-\frac{A^2}{2\sigma}} = 0$ par croissances comparées. D'autre part, l'intégrale $\sigma \int_0^{+\infty} \frac{t}{\sigma} e^{-\frac{t^2}{2\sigma}}\,\mathrm{d}t$ converge et vaut σ (en reconnaissant la densité de la loi de Rayleigh). On en déduit que $E(X^2)$ existe bien et vaut :

$$E(X^2) = 2\sigma.$$

Enfin par la formule de Huygens, on obtient que $V(X)$ existe et vaut :

$$V(X) = E(X^2) - E(X)^2 = 2\sigma - \frac{\pi\sigma}{2} = \frac{(4-\pi)\sigma}{2}.$$

En prenant $\sigma = 1$, on trouve que $\sigma(X) = 0.6551364$, ce qui correspond à la valeur numérique estimée.

5. (a) Soit X une variable suivant une loi de Rayleigh de paramètre σ. On cherche la loi de $Y = X^2$. Puisque $f \neq 0$ sur $\mathbb{R}_+^*$, on a $X(\Omega) = \mathbb{R}_+^*$, et donc $Y(\Omega) = \mathbb{R}_+^*$. En particulier, on a $F_Y(x) = 0$ pour tout $x \leq 0$.

   Supposons que $x > 0$, on obtient :

   $$F_Y(x) = P(X^2 \leq x) = P(X \leq \sqrt{x}) = 1 - e^{-\frac{\sqrt{x}^2}{2\sigma}} = 1 - e^{-\frac{x}{2\sigma}}.$$

   On reconnait la fonction de répartition de la loi $\mathscr{E}(\frac{1}{2\sigma})$. Donc Y suit une loi exponentielle de paramètre $\dfrac{1}{2\sigma}$.

   (b) On peut simuler une loi exponentielle $\mathscr{E}(\lambda)$ grâce à la fonction `rd.exponential(1/lbd)`. Il s'agit ensuite d'en prendre la racine carrée. On propose le code suivant.

```
def rayleigh2(sigma):
      y = rd.exponential(2*sigma)
      return sqrt(y)
```

Exercice 14 (★★★ - QSP HEC 2016)

Donner la finalité du programme suivant :

```
N = 100000 ; S = 0
for i in range(N):
      u = rd.random()
      S = S + (4/N)*1/(1+u**2)
print(S)
```

On pourra penser à la loi faible des grands nombres.

Au k-ème passage dans la boucle for, le programme simule une réalisation u_k de la loi $\mathscr{U}([0,1])$ et ajoute à S la quantité $\dfrac{4}{N}\dfrac{1}{1+u_k^2}$. À la fin de la boucle for, la variable S contient donc :

$$S = \frac{1}{N}\sum_{k=1}^{N} \frac{4}{1+u_k}$$

où $(u_1, \ldots, u_N)$ est un N-échantillon indépendant de la loi $\mathscr{U}([0,1])$. Puisque $N = 100000$ est

grand, on sait par la loi faible des grands nombre que la valeur de S est proche de :

$$E\left(\frac{4}{1+U^2}\right) \quad \text{où} \quad U \hookrightarrow \mathscr{U}([0,1]).$$

Notons $f : x \mapsto \begin{cases} 1 & \text{si } x \in [0,1] \\ 0 & \text{sinon} \end{cases}$ une densité de U. Par le théorème de transfert, cette espérance est égale (sous réserve de convergence absolue) à :

$$\int_{-\infty}^{+\infty} \frac{4}{1+t^2} f(t)\,\mathrm{d}t = \underbrace{\int_0^1 \underbrace{\frac{4}{1+t^2}}_{\text{fct. cont. sur } [0,1]} \mathrm{d}t}_{\text{l'intég. converge}} = [4\arctan(t)]_0^1 = 4\arctan(1) = \pi.$$

Ainsi le programme précédent renvoie une valeur approchée de π.

Remarque. On reconnait ici un calcul approché d'une intégrale par la méthode de Monte Carlo. Je vous renvoie à ce sujet au **TP8 - Méthode de Monte Carlo**.

Exercice 15 (★★★ - QSP HEC 2021)

On considère le code `Python` suivant :

```
def g(x):
    X = (1/2)*rd.random(10000)
    Y = (1/3)*rd.random(10000)
    S = 0
    for k in range(10000):
        if X[k]+Y[k] <= x :
            S = S+1
    return S/10000

def graph2():
    x = np.linspace(-0.1,1,101)
    y = g(x)
    plt.plot(x,y)
    plt.show()
```

1. Commenter les fonctions `Python`.
2. Dessiner le graphe de sortie de la fonction `graphe2`.

1. Dans la fonction g, X est un tableau contenant 10000 réalisations d'une variable U suivant une loi uniforme sur $[0,1/2]$, de même pour Y pour une variable V suivant la loi uniforme sur $[0,1/3]$. À la fin de la boucle `for`, S contient le nombre d'occurrences dans le tableau $X+Y$ qui sont inférieures à x. Par la loi faible des grands nombres, la fréquence $S/10000$ de réalisation de cette condition est proche de la probabilité théorique $P(U+V \leq x) = F_{U+V}(x)$. Ainsi, g représente la fonction de répartition (empirique) de la variable $U+V$.

   La deuxième fonction `graph2()` trace cette fonction de répartition sur l'intervalle $[-0.1, 1]$.
2. Puisque U et V sont indépendantes et de densités respectives f_U et f_V bornées, $T = U+V$

est à densité, de densité h (continue sur $\mathbb{R}$) donnée par :

$$\forall x \in \mathbb{R}, \quad h(x) = \int_{-\infty}^{+\infty} f_U(t) f_V(x-t)\,\mathrm{d}t.$$

On obtient après calculs :

$$h(x) = \begin{cases} 0 & \text{si } x < 0 \\ 6x & \text{si } 0 \leq x \leq 1/3 \\ 2 & \text{si } 1/3 < x < 1/2 \\ 5 - 6x & \text{si } 1/2 \leq x \leq 5/6 \\ 0 & \text{si } x > 5/6 \end{cases}.$$

Reste à calculer la fonction de répartition H de T. On obtient :

$$H(x) = \begin{cases} 0 & \text{si } x < 0 \\ 3x^2 & \text{si } 0 \leq x \leq 1/3 \\ 2x - \frac{1}{3} & \text{si } 1/3 < x < 1/2 \\ 5x - 3x^2 - \frac{13}{12} & \text{si } 1/2 \leq x \leq 5/6 \\ 1 & \text{si } x > 5/6 \end{cases}.$$

Exercice 16 (★★★★ - QSP HEC 2021)

On considère le code `Python` suivant :

```
def smul1(n,p):
    X = rd.geometric(p,n)
    Z = np.max(X)
    T = np.zeros(Z)
    for k in range(n)
        if T[X[k]]==0 :
            T[X[k]] = 1
    return np.sum(T)

def smul2(n,p):
    X = rd.geometric(p,10000,n)
    Y = np.zeros((10000,1))
    for j in range(10000):
        Z = np.max(X[j,:])
        T = np.zeros(Z)
        for k in range(n):
            if T[X[j,k]]==0 :
                T[X[j,k]] = 1
        Y[j] = np.sum(T)
    m = np.mean(Y)
```

1. Commenter la fonction `smul1`. Quelles sont les valeurs possibles de sortie ? Donner la loi de la variable aléatoire `smul1(2,0.75)`.

2. Le résultat de `smul2(2,0.75)` est `1.3971`. Commenter ce résultat.

1. La variable X contient un vecteur de taille n contenant n réalisations indépendantes d'une loi géométrique de paramètre p. T contient un vecteur de taille la valeur maximal du vecteur X. On parcourt ensuite le vecteur X à l'aide d'une boucle `for`, et on teste grâce au vecteur T si la k-ème valeur `X[k]` a déjà ou non été obtenue. Si ce n'est pas le cas, on affecte à la `(X[k]-1)`-ème composante de `T` un 1. À la fin de la boucle `for`, `T` est un vecteur contenant des 1 et des 0, un 1 en `(k-1)`-ème position indiquant que la valeur `k` apparait dans le vecteur `X`. On renvoie enfin la somme des coefficients de `T`, qui est donc le nombre de valeurs distinctes apparaissant dans le vecteur `X`.

   Pour conclure, `smul1` simule la variable aléatoire Y_n qui est égale au nombre de valeurs distinctes prises par n variables $X_1, \dots, X_n$ indépendantes suivant des lois géométriques de paramètre p. En particulier, on notera que $Y_n(\Omega) = [\![1, n]\!]$.

   La loi simulée par `smul1(2,0.75)` est celle de la variable Y_2, donnée par $Y_2(\Omega) = [\![1, 2]\!]$ et :

$$P(Y_2 = 1) = P(X_1 = X_2) = \sum_{k=1}^{+\infty} P(X_1 = k, X_2 = k)$$
$$= \sum_{k=1}^{+\infty} P(X_1 = k)^2 \text{ car les v.a. sont i.i.d.} \quad = \sum_{k=1}^{+\infty} p^2((1-p)^2)^{k-1} = \frac{p^2}{1-(1-p)^2} = \frac{p}{2-p} = \frac{3}{5}$$

   et

$$P(Y_2 = 2) = 1 - P(Y_2 = 1) = \frac{2(1-p)}{2-p} = \frac{2}{5}.$$

2. `smul2` donne la moyenne empirique d'un échantillon de 10000 variables de même loi que Y_n. Elle approche donc l'espérance de Y_n. On calcule pour $n = 2$:

$$E(Y_2) = \frac{4-3p}{2-p}$$

   pour $p = 3/4$. On obtient $E(Y_2) = 7/5 = 14/10$. Le résultat obtenu par `smul2` est bien cohérent.

Exercice 17 (★★★★ - QSP HEC 2015)

Dans une classe de 30 élèves, on considère une expérience consistant d'abord à demander à chaque élève sa date d'anniversaire. La suite de l'expérience (simulée 1000 fois sur ordinateur) est décrite par le programme `Python` suivant :

```
Nexp = 1000 ; Neleve = 30 ; test = 0 ;
for n in range(Nexp):
     anniv = np.zeros(Neleve)
     for i in range(Neleve):
          anniv[i] = np.floor(365*rd.random())+1
     anniv = np.sort(anniv) # np.sort = tri par ordre croissant
     ok = 0
     for j in range(Neleve-1):
          if anniv[j]=anniv[j+1] :
               ok = 1
     test = test + ok
print(test/Nexp)
```

1. Le code retourne une valeur (à chaque fois différente) autour de $0,71$. Que représente cette valeur ?

2. Calculer la valeur exacte de la probabilité simulée par ce programme.
3. Écrire un programme `Python` permettant de déterminer le nombre d'élèves à partir duquel cette valeur dépasse 0.5.

1. Les lignes 3 à 5 ont pour effet de choisir au hasard la date de naissance de chacun des élèves (qui sont au nombre de `Neleve`, fixé ici à 30).

   La liste `anniv` ainsi obtenue est ensuite triée à la ligne 6. Elle est ensuite parcourue grâce à la boucle de la ligne 8, et le code des lignes 9 à 10 a pour effet, de donner à `ok` la valeur 1 si la même date apparait deux fois dans anniv. Le fait que `anniv` soit triée par ordre croissant implique que si la même date apparaît deux fois dans `anniv`, ce sera nécessairement à deux positions consécutives.

   Enfin, la variable `test` compte le nombre de fois où, lors des `Nexp` répétitions de l'expérience, la variable `ok` valait 1.

   Puisqu'enfin on divise `test` par `Nexp`, la valeur affichée par le programme correspond environ à la probabilité qu'il y ait deux élèves avec la même date d'anniversaire dans la classe. Cette probabilité est donc visiblement proche de 0,71.

2. Il y a 365^{30} possibilités pour les choix des dates d'anniversaires des trente élèves. Pour que tous ces 30 élèves aient des dates d'anniversaire différentes, il y a 365 choix possibles pour la date d'anniversaire du premier, puis 364 choix pour la date d'anniversaire du second, etc, 336 choix pour la date d'anniversaire du dernier.

   Donc la probabilité que tous les élèves aient une date d'anniversaire différente vaut

   $$\frac{365 \times 364 \times \cdots \times 336}{365^{30}} = \frac{365!}{365^{30} \times 335!}$$

   Et donc la probabilité cherchée est, en passant à l'événement contraire :

   $$1 - \frac{365!}{365^{30} \times 335!} = 1 - \prod_{i=1}^{30} \frac{365 - i + 1}{365}.$$

3. Utilisons le résultat de la question précédente : pour une classe de N élèves, la probabilité que deux élèves aient la même date d'anniversaire vaut :

   $$1 - \frac{365!}{365^N \times (365 - N)!} = 1 - \prod_{i=1}^{N} \frac{365 - i + 1}{365}.$$

   On utilisera la deuxième expression en `Python`, l'expression avec les factoriels n'étant pas utilisable puisque 365! et $365^N \times (365 - N)!$ sont trop grand pour que `Python` parvienne à les calculer.

   Voici une suggestion de code :

```
1 N = 1
2 p = 1
3 while p >0.5 :
4       p = p*(365-N)/365
5       N = N+1
6 print(N)
```

   Ce programme retourne alors $N = 23$: dans une classe de 23 élèves, il y a plus d'une chance sur deux que deux étudiants soient nés le même jour.