Algorithmique et informatique avec Python : le cours de première année... pour les élèves de deuxième année

I Les commandes de base en Python

1) Différents types d'objets

Si `x` et `y` sont deux variables Python, alors la commande `x,y=y,x` échange leurs contenus respectifs (sauf rares exceptions comme, par exemple, lorsqu'il s'agit de lignes ou de colonnes d'un tableau du type `ndarray`).

En Python (comme en Mathématiques) il y a différents types d'objets que l'on peut affecter à des variables avec la commande **nom=expression**. Le contenu de `expression` (une expression mathématique) est alors stocké dans une variable appelée `nom`.
Parmi les types d'objets, il y a notamment :

- Les entiers (type `int`).
- Les réels ou flottants (type `float`).

  En mathématiques les entiers sont des réels. En Python ce sont deux types différents : par exemple `1` est entier et `1.` est un réel et Python ne considèrera pas le deuxième comme un entier. Il faut donc y penser quand on utilise une commande demandant un entier (comme `range`) et que l'on dispose d'un réel. La commande `int(a)` transforme un réel `a` en un entier (en enlevant les chiffres après la virgule).

Il y a plein de subtilités autour des entiers et des flottants pour Python qu'il n'est pas utile d'explorer en ECG.

- Les chaînes de caractère (type `str`). Pour définir une chaîne de caractères, on écrit les caractères entre guillements (`"`) ou bien entre apostrophes (`'`). Si dans la chaîne, il y a déjà des guillements ou des apostrophes, on les fait précéder de `\`.

  Par exemple `"Je m\'appelle Bond, James Bond."`

  Une chaine de caractère contenant un mot ne doit pas être confondue avec le mot lui-même (sans `"` ou `'`, Python considèrera qu'il s'agit d'une variable).

  Par exemple `"eps"` *affiche le mot* eps *alors que* eps *affiche le contenu de la variable* eps.

  La commande `str(x)` transforme le contenu de la variable `x` en la chaîne de caractère correspondant au contenu de `x`.

Si `x=5`, alors la commande `'x'` renvoie `'x'`, alors que la commande `str(x)` renvoie `'5'`.

- Les booléens (type `bool`) qui ne prennent que deux valeurs : `True` (vrai) ou `False` (faux).
- Les listes (type `list`, cf. paragraphe 3).
- Les tableaux (type `ndarray`, cf. paragraphe 4 et partie V).
- Les fonctions (type `function`, paragraphe 5 et partie II).

La commande `type(A)` renvoie le type de l'objet `A`.

2) Opérations de base sur les variables

a) Opérations sur les réels

- Soient x et y deux réels implémentés en Python dans les variables `x` et `y`. Les commandes **`x+y,x-y,x*y`** renvoient respectivement la somme de x et y, la soustraction de x par y et la multiplication de x par y. Si $y \neq 0$, **`x/y`** renvoie la division de x par y. Enfin, sous réserve d'existence, **`x**y`** renvoie x^y.

  La commande `x+=y` ajoute définitivement `y` à la variable `x` (c'est-à-dire elle remplace le contenu de `x` par le résultat de `x+y`). Elle fait la même chose que `x=x+y`.

Les commandes `x-=y`, `x*=y`, `x/=y` et `x**=y` fonctionnent sur le même principe mais avec la soustraction, la multiplication, la division et l'élévation à la puissance.

- Si x et y deux entiers implémentés en Python dans les variables `x` et `y`, alors la commande `x%y` renvoie le reste de la division euclidienne de x par y et la commande `x//y` le quotient.
- On peut également évaluer des réels par des fonctions usuelles (cf. paragraphe 5) ou des fonctions que l'on définit soi-même (cf. partie II).
- Python nous permet d'accéder à des approximations de π et de $e = \exp(1)$ via les commandes **`np.pi`** et **`np.e`** respectivement.

La commande `x%y` est utile (pour savoir si x est un multiple de y notamment) mais elle n'apparaît pas dans le programme.

Mais avant cela il faut importer la bibliothèque `numpy` en utilisant la commande `import numpy as np` (on en reparlera).

b) Opérations sur les booléens

Les booléens sont très utiles puisqu'ils apparaissent dans les conditions des structures conditionnelles et des boucles while (cf. partie II).

- En général les booléens sont créés en faisant des comparaisons entre réels (ou entre vecteurs, entre matrices, cf. paragraphe 4). Si x et y sont des réels implémentés en Python par `x` et `y`, alors **`x==y,x<y,x<=y,x>y,x>=y,x!=y`** renvoient `True` si x est respectivement égal à y, strictement inférieur à y, inférieur à y, strictement supérieur à y, supérieur à y et différent de y. Elles renvoient `False` sinon.

Ne pas confondre `==` qui teste l'égalité et `=` qui sert à l'affectation dans une variable.

- On peut faire des opérations sur les booléens avec les commandes **`and,or,not`** qui implémentent respectivement les $\text{et}, \text{ou}, \text{non}$ logiques.

  Par exemple, `x>3 and x<=5` *renvoie* `True` *si et seulement si* $x \in]3\,;5]$. *La commande* `x>=8 or x<-2` *renvoie* `True` *si et seulement si* $x \in]-\infty\,;-2[\cup [8\,;+\infty[$.

- Si x est un réel implémentés en Python par `x`, alors la commande `int(x)==x` renvoie `True` si x est un entier, `False` sinon.

Ces trois commandes ne sont pas explicitement au programme mais sont très utiles.

- Si n et p sont des entiers (avec $p \neq 0$) implémentés en Python par `n` et `p`, alors :
  - `(n%2)==0` renvoie `True` si n est pair, `False` sinon.
  - `(n%2)==1` renvoie `True` si n est impair, `False` sinon.
  - `(n%p)==0` renvoie `True` si n est divisible par p, `False` sinon.

3) Listes

- On peut construire une liste d'objets (pas forcément du même type) entre crochets et séparés par des virgules.

  Par exemple `L=[1,2,0,8,-9,7,4,0]`.

- La commande `[]` crée une liste vide.
- Si `x` est une variable, alors la commande `x in L` renvoie `True` si la variable `x` est un élément de la liste `L` et `False` sinon.
- Si `L` est une liste, la commande **`len(L)`** renvoie le nombre d'éléments de la liste.
- Python numérote les indices d'une liste à partir de 0 et non de 1. Ainsi le premier élément de la liste est, pour Python, celui d'indice 0. Le deuxième élément de la liste est, pour Python, celui d'indice 1, etc.

Dans ce cours, on différentiera indice (numérotation à partir de 0) et position (numérotation à partir de 1) dans la liste.

- `L[i]` renvoie la $(i+1)^{\text{ième}}$ coordonnée de `L`. Ainsi, pour accéder au $i^{\text{ième}}$ élément de `L`, la commande est `L[i-1]`.
- `L[-1]` renvoie le dernier coefficient de `L`.
- On peut modifier des éléments en utilisant la syntaxe usuelle pour l'affectation dans une variable.

  Par exemple `L[2]=5` *remplace le* $3^{ième}$ *(celui d'indice* 2*) élément de* L *par* 5.

Les commandes append, pop et len ne sont pas explicitement au programme mais sont très utiles.

Concaténer deux listes signifie les regrouper pour en créer une nouvelle.

- On peut ajouter un élément a à une liste L avec la commande `L.append(a)`.
  La commande `L.pop(i)` renvoie et enlève l'élément d'indice `i` (i.e. celui en position `i+1`) de la liste L.
- Si L et M sont deux listes, alors la commande `L+M` concatène les deux listes.
  Ainsi la commande `L=L+[a]` ajoute aussi l'élément a à la liste L.
- La commande `L[:i]` renvoie la sous-liste de L constituée des `i` premiers éléments de L (i.e les éléments de l'indice 0 à l'indice `i-1`).
  La commande `L[i:]` renvoie la sous-liste de L constituée des éléments de L à partir de l'indice `i` (i.e de la position `i+1`).

  Par exemple `L[2:]` *renvoie* `[0,8,-9,7,4,0]`.

  Plus généralement la commande `i:j` renvoie la sous-liste des éléments de L de l'indice `i` à l'indice `j-1` (i.e. de la position `i+1` à la position `j`).
  Ainsi la commande `L=L[:i]+L[i+1,:]` supprime l'élément d'indice 5 (i.e. en position 6). La commande `L.pop(i)` fait la même chose mais renvoie en plus l'élément supprimé (que l'on peut stocker dans une variable).
  Enfin, la commande `L[:]` renvoie (une copie de) L.

On touche ici à une subtilité de Python : si jamais on écrit `M=L`, alors toute modification de `M` entraînera la même modification de L (c'est comme ça : il s'agit du même objet car il est stocké au même endroit dans la mémoire) et vice versa. Mais, si on écrit `M=L[:]`, alors les listes `M` et L sont identiques mais elles sont désormais indépendantes.

- Si n et m sont deux entiers tels que $0 \leqslant m < n$ implémentés en Python par `n` et `m`, alors **`range(n)`** renvoie la liste des entiers compris au sens large entre 0 et $n-1$. La commande **`range(m,n)`** renvoie la liste des entiers compris au sens large entre m et $n-1$. Et donc `range(m,n+1)` contient la liste des entiers compris au sens large entre m et n.

  En fait, il faudrait plutôt écrire `list(range(m,n))` pour qu'il s'agisse vraiment d'une liste. A part si on veut lui ajouter ou enlever des éléments, il ne sera pas nécessaire en pratique d'ajouter `list`.

4) La bibliothèque `numpy`

L'utilisation de Python requiert l'importation préalable de bibliothèques (à chaque démarrage de l'environnement de travail). Nous allons en utiliser plusieurs mais, pour le moment, concentrons-nous sur la plus utile : la bibliothèque `numpy`. Celle-ci contient des commandes permettant de créer et manipuler des tableaux à plusieurs dimensions (des vecteurs en dimension 1, des matrices en dimension 2).

Pour le moment on limite notre étude aux vecteurs et on étudiera les matrices dans la partie VI.

Un vecteur est plus ou moins similaire à une liste mais tous les éléments sont considérés comme étant du même type.

- Pour importer la bibliothèque `numpy`, on utilise la commande

```
import numpy as np
```

`L` et `np.array(L)` sont de type différent.

- Si L est une liste de nombres, alors **`np.array(L)`** crée le vecteur ligne (tableau unidimensionnel) contenant, dans l'ordre, les nombres de la liste.
- On peut créer ainsi un vecteur « à la main » à partir d'une liste mais on peut aller plus vite pour des vecteurs remarquables (surtout si il y a beaucoup de coordonnées) : soient $n \in \mathbb{N}$ et $(a,b,h) \in \mathbb{R}^3$ implémentés en Python par `n`,`a`,`b` et `h` respectivement :
  - **`np.zeros(n)`** renvoie un vecteur avec n coordonnées toutes nulles.
  - **`np.ones(n)`** renvoie un vecteur avec n coordonnées valant toutes 1.
  - **`np.linspace(a,b,n)`** renvoie un vecteur contenant n valeurs comprises entre a et b (a et b inclus) régulièrement espacées.

    Par exemple `np.linspace(3,4,6)` *renvoie*

    `array([3.,3.2,3.4,3.6,3.8,4.])`

— Si $a < b$ et $h > 0$, **`np.arange(a,b,h)`** renvoie un vecteur contenant $a, a+h, a+2h, a+3h, a+4h$ etc. jusqu'à b (b non inclus). Si $a > b$ et $h < 0$, alors cela fait la même chose mais dans l'ordre décroissant.

Par exemple `np.arange(3,4,0.2)` *renvoie*
`array([3.,3.2,3.4,3.6,3.8])`
et `np.arange(6,0,-1)` *renvoie* `array([6,5,4,3,2,1])`

Encore une fois Python numérote à partir de 0 et non de 1 donc on fera la distinction entre indice et position dans la liste.

- On accède aux coordonnées du vecteur (et on les modifie) de la même façon qu'on accède aux éléments d'une liste.
- Si on ne connaît pas le nombre de coordonnées d'un vecteur `V`, il suffit d'utiliser la commande `len(V)`.
- On peut effectuer des opération coefficient par coefficient sur des vecteurs via les commandes **`+,-,*,/,**`**.

  Par exemple :

  — `V**3` *renvoie le vecteur* V *dont tous les coordonnées ont été élevées à la puissance* 3.

  — *Si* `V=np.array([1,2,3])` *et* `W=np.array([0,7,-1])`, *alors* `V*W` *renvoie* `array([0,14,-3])`.

Si on écrit `M=L`, alors `M` contient le vecteur `L` bien sûr. Mais toute modification du vecteur `L` provoquera la même modification sur `M` implémente (car c'est le même objet en fait). Si on veut éviter cela, il faut remplacer par `M=L` par `L=np.copy(L)` ou `M=L.copy()` (mais ces commandes ne sont pas exigibles) : les modifications du vecteur `L` n'auront alors pas d'effet sur le vecteur `M`.

On ne peut sommer, soustraire, multiplier, diviser deux vecteurs entre eux que s'ils ont le même nombre de coordonnées.

- On peut comparer deux vecteurs coefficient par coefficient ou comparer un vecteur coefficient par coefficient avec un nombre en utilisant les commandes **`==,>,<,>=,<=,!=`**. Le résultat est une matrice de booléens.

  Par exemple, si `V=np.array([-1,2,3,4])` *et* `W=np.array([7,1,3,5])`, *alors* `V>=W` *renvoie* `array([False,True,True,False])`.

5) Fonctions usuelles

Les fonctions $\exp$, $\ln$, $\sin$, $\cos$, racine carrée, valeur absolue et partie entière sont implémentées en Python respectivement par **`np.exp`**, **`np.log`**, **`np.sin`**, **`np.cos`**, **`np.sqrt`**, **`np.abs`** et **`np.floor`**.

La commande `int(a)` ne renvoie pas la partie entière de `a` si `a` est réel négatif non entier, contrairement à `np.floor(a)`.

`log` désigne le logarithme népérien pour Python (c'est la convention anglo-saxonne).

Ces fonctions peuvent s'appliquer à des variables numériques ou vectoriellement (à des vecteurs réels) coordonnée par coordonnée.

Par exemple `np.log(np.e)` *renvoie* `1.0`.
La commande `np.sin([0,np.pi,np.pi/2])` *renvoie* `array([0.0,0.0,1.0])`.

Remarques :

On peut aussi utiliser **`np.tan(x)`** mais cette commande n'est pas exigible.

- Si $x \in D_{\tan}$ est implémenté en Python par `x`, alors **`np.sin(x)/np.cos(x)`** renvoie (une approximation de) $\tan(x)$.
- Si $x \in \mathbb{R}_+$ et $n \in \mathbb{N}\backslash\{0;1\}$ sont implémentés en Python par `x` et `n`, alors **`x**(1/n)`** renvoie (une approximation de) $\sqrt[n]{x}$.
- Si $x \in \mathbb{R}$ est implémenté en Python par `x`, alors **`np.arctan(x)`** renvoie (une approximation de) $\mathrm{Arctan}(x)$ mais cette commande n'est pas exigible. On peut aussi programmer la fonction Arctan avec la méthode des rectangles (cf. partie VII) ou un algorithme de dichotomie (cf. partie VI).

II Structures de bases de la programmation

Avant de commencer une commande très utile : `#`. Elle permet d'écrire un commentaire : tout ce qui suit cette commande (jusqu'au retour à la ligne) ne sera pas exécuté par Python et qui sert par exemple à expliquer ce que l'on fait, à s'y retrouver. Citons le programme : « les étudiants doivent savoir faire un usage judicieux des commentaires ».

1) Définition et utilisation d'une fonction

Pour créer une fonction en Python, appelée `f`, qui prend en entrée des variables `x1,x2,...xn` et qui renvoie une variable `y`, voici la syntaxe :

```
def f(x1,x2,...xn):
    <bloc d'instructions>
    return y
```

Remarquons les alinéas après la commande `def` (il y en a aussi après les commandes `if,elif,else`, `for,while`, cf. paragraphes suivants). Ce n'est pas facultatif : on appelle cela l'indentation. En Python, il n'y a pas de `begin` ou de `end`, ni de marqueurs du début ou de la fin d'un code. Les seuls délimiteurs sont les `:` et l'indentation (l'indentation démarre la structure et la « désindentation » la termine).

On remplace `<bloc d'instructions>` par des commandes permettant de transformer les variables d'entrée en la variable de sortie. En exécutant la fonction dans la console Python puis la commande `f(x1,x2,...xn)`, on obtient l'image de `x1,x2,...xn` par `f`.

Exemple : *Voici une fonction qui prend en argument un réel x et un entier n et qui renvoie* $\dfrac{\lfloor 10^n x \rfloor}{10^n}$.

```
def tronc(x,n):
    y=10**n
    return np.floor(y*x)/y
```

La commande `tronc(np.pi,5)` *renvoie* `3.14159`. *Elle tronque l'écriture décimale de π à 5 chiffres.*

Remarques :

- Les variables `x1,x2,...,xn,y` peuvent tout à fait être de types différents. Notamment `y` peut être une liste, un vecteur ou une matrice s'il y a plusieurs arguments de sortie.
- ⚠ Ne pas confondre `return` et `print` (qui ne fait même pas partie des commandes exigibles) qui afficherait la variable de sortie mais sans la retourner (celle ci serait inutilisable dans la suite car on ne pourrait pas récupérer ce qui est affiché dans une variable).
- S'il y a plusieurs `return` dans une fonction (cela arrive souvent lorsqu'il y a des structures conditionnelles), la fonction renvoie l'argument du premier d'entre eux.
- Si `f` est une fonction qui prend en entrée un réel `x` et qui renvoie un réel `y`, il est classique de vouloir appliquer `f` à une liste ou un tableau `X` pour récupérer une liste ou un tableau `Y` contenant les images des éléments de `X`. Pour cela on utilise la syntaxe **`Y=[f(x) for x in X]`**.

  Si `X=range(-2,5)`, *alors la commande* `[x**2+1 for x in X]` *renvoie* `[5,2,1,2,5,10,17]`.

On peut aussi « vectoriser » la fonction avec la commande **`f=np.vectorize(f)`** puis utiliser **`Y=f(X)`**

2) Structures conditionnelles

Si on veut que le bloc d'instructions `<bloc d'instructions>` soit exécuté si la condition `<condition>` est remplie, et que le bloc d'instructions `<bloc d'instructions sinon>` soit exécuté sinon, voici la syntaxe :

```
if <condition>:
   <bloc d'instructions>
else:
   <bloc d'instructions sinon>
```

S'il n'y a pas d'instruction alternative, alors on ne met pas les deux dernières lignes.

Si on veut que le bloc d'instructions <bloc d'instructions 1> soient exécuté si la condition <condition 1> est remplie, le bloc d'instructions <bloc d'instructions 2> si la condition <condition 2> est remplie, le bloc d'instructions <bloc d'instructions 3> si la condition <condition 3> est remplie... et enfin le bloc d'instructions <bloc d'instructions sinon> si aucune de ces dernières conditions n'est remplie, voici la syntaxe :

Vous aurez compris que elif est une contraction de else if.

```
if <condition 1>:
   <bloc d'instructions 1>
elif <condition 2>:
   <bloc d'instructions 2>
elif <condition 3>:
   <bloc d'instructions 3>
...
else:
   <bloc d'instructions sinon>
```

Exemple : *Voici une fonction qui prend en entrée trois réels a, b, c avec $a \neq 0$ et qui renvoie une liste contenant les solutions de l'équation $ax^2 + bx + c = 0$.*

```
def trinome(a,b,c):
    D=b**2-4*a*c
    if D>0:
        return [(-b-np.sqrt(D))/(2*a),(-b+np.sqrt(D))/(2*a)]
    elif D==0:
        return [-b/(2*a)]
    else:
        return []
```

3) Structures répétitives

Les structures répétitives (ou itératives) permettent d'effectuer plusieurs opérations à la suite. En général, la construction d'une telle structure au sein d'un programme se base sur quatre étapes :

- On identifie les variables d'entrées (qui proviennent du début du programme ou qui sont des arguments d'entrée d'une fonction par exemple).
- On définit des variables qui vont être modifiées lors de la boucle. C'est l'étape d'initialisation.
- On écrit la boucle en tant que telle. Elle utilise les variables d'entrée et, à chaque étape de la boucle, les variables définies à l'étape d'initialisation sont mises à jour.
- On renvoie les variables de sorties (en général des fonctions des quantités calculées pendant la boucle).

a) Boucles for

On emploie les boucles for lorsqu'on veut exécuter un bloc d'instructions un certain nombre de fois, nombre que l'on connaît. Soit T une liste ou un tableau.

Souvent T=range(a,b) (la liste des entiers compris au sens large entre l'entier a et l'entier b-1).

```
for x in T:
    <bloc d'instructions>
```

La variable x va prendre tour à tour et dans l'ordre les valeurs présentes dans T et, pour chacune d'entre elle, elle va exécuter le bloc d'instructions <bloc d'instructions>. Ce dernier peut dépendre ou non de x.

⚠ x est une variable muette dans la boucle (il faut la voir comme un indice de sommation par exemple) et ne doit pas être introduite.

Pour cela parcourt la liste avec une boucle `for`. Pour chaque élément, elle teste si il est strictement positif ou non et, si c'est le cas, elle incrémente un compteur de 1. Ici `n` est le compteur et sa valeur est mise à jour (1 de plus) si on rencontre un élément strictement positif.

Exemple : *La fonction suivante prend en argument une liste (ou un vecteur) et renvoie le nombre d'éléments qui sont strictement positifs.*

```
def nb_st_positif(L):
    n=0#Au départ, aucun élément n'est strictement positif.
    for x in L:#Pour chaque élément de la liste
        if x>0:#Si cet élément est strictement positif
            n=n+1#Ca en fait un de plus
    return n
```

La commande `nb_st_positif([1,-8,5,9,-7,-4,7,3,2])` *renvoie* 6.

L'une des fonctionnalités les plus intéressantes de Python est la possibilité de définir des listes « par compréhension » : si `f` est une fonction et `T` une liste ou un tableau, alors la commande `[f(x) for x in T]` construit la liste des images des éléments de `T` par la fonction `f`. On a vu un exemple dans le paragraphe précédent.

b) Boucles while

Les boucles `for` servent principalement à calculer des sommes, produits ou les termes successifs d'une suite. C'est l'objet de la prochaine partie.

On emploie les boucles while lorsqu'on veut exécuter un bloc d'instructions tant qu'une certaine condition est remplie (souvent lorsqu'on ne l'on connaît pas précisément le nombre d'itérations à exécuter).

```
while <condition>:
    <bloc d'instructions>
```

Les variables intervenant dans la condition `<condition>` doivent être introduites préalablement contrairement à l'indice dans une boucle `for`.

Tant que la condition `<condition>` est remplie, le bloc d'instructions `<bloc d'instructions>` sera exécuté. Il est donc essentiel que l'exécution de `<bloc d'instructions>` vienne modifier la condition `<condition>` afin que cette dernière finisse par devenir fausse (sinon la boucle va durer éternellement).

Exemple : *On sait que* $n^3 - 7n + 2021 \xrightarrow[n\to+\infty]{} +\infty$ *et donc, pour tout* $A > 0$*, il existe* $n_A \in \mathbb{N}$ *tel que, pour tout* $n \geqslant n_A$*,* $n^3 - 7n + 2021 \geqslant A$*. La fonction suivante prend* A *en argument et détermine* n_A*.*

Pour trouver n_A, l'idée est de démarrer avec $n = 0$ et, tant que $n^3-7n+2021 < A$, on augment de 1 la valeur de n.

```
def rang_min(A):
    n=0
    while n**3-7*n+2021<A:
        n=n+1
    return n
```

La commande `rang_min(10000)` *renvoie* 21*, la commande* `rang_min(100000)` *renvoie* 47 *et la commande* `rang_min(1000000)` *renvoie* 100.

On verra d'autres exemples dans la partie III consacré aux sommes.

III Sommes, produits et suites avec Python

1) Calcul de sommes et produits « simples »

L'outil informatique trouve son intérêt lorsque l'on n'arrive pas à trouver une formule pour la somme ou le produit.

Le but de cette partie est de calculer des valeurs approchées avec Python d'une somme $\sum_{i\in I} u_i$ ou d'un produit $\prod_{i\in I} u_i$ d'une famille de réels $(u_i)_{i\in I}$ indexée par un ensemble **fini** I. Comment faire ?

- On code la famille I en une liste Python `I`. En général :
  - si $n \in \mathbb{N}^*$ et $I = [\![0\,;n]\!]$, on prend `I=range(n+1)`.
  - si $n \in \mathbb{N}^*$ et $I = [\![1\,;n]\!]$, on prend `I=range(1,n+1)`.
  - si m et n sont des entiers relatifs tels que $m < n$ et si $I = [\![m\,;n]\!]$, on prend `I=range(m,n+1)`.

  Bien sûr I peut être quelconque (fini) mais, en général, soit on arrive à se ramener à l'un des cas précédents soit on construit `I` à la main (lorsqu'il y a peu d'éléments).

On peut aussi sommer/multiplier des termes d'une suite. Nous verrons cela dans le prochain paragraphe.

- Si les expressions des réels de la famille $(u_i)_{i\in I}$ sont compliquées, on peut construire une fonction en Python (appelons-la `u`), qui prend en argument un élément i de I et qui renvoie une expression de u_i.
- Pour les sommer, on dispose essentiellement de deux options :
  — Faire une boucle sur les éléments de I telle que, à chaque étape, on ajoute le terme suivant à la somme.

Ne pas oublier d'initialiser la somme à 0 (une somme vide est nulle).

```
S=0
for i in I:
    S=S+u[i]
```

  — Créer une liste contenant les réels u_i, $i \in I$ et les sommer.

```
import numpy as np
S=np.sum([u(i) for i in I])
```

  Dans les deux cas, la variable `S` contient la somme que l'on cherchait à calculer.
- Pour les multiplier, on dispose essentiellement de deux options :
  — Faire une boucle sur les éléments de I telle que, à chaque étape, on multiplie le produit par le terme suivant.

Ne pas oublier d'initialiser le produit à 1 (un produit vide vaut 1).

```
P=0
for i in I:
    P=P*u[i]
```

  — Créer une liste contenant les réels u_i, $i \in I$ et les multiplier.

```
import numpy as np
P=np.prod([u(i) for i in I])
```

  Dans les deux cas, la variable `P` contient le produit que l'on cherchait à calculer.

Exemples :

- *Pour calculer* $\displaystyle\sum_{i=0}^{20} i^2$ *:*

```
np.sum([i**2 for i in range(21)])
```

- *Pour calculer* $\displaystyle\sum_{i=17}^{2021} \frac{i}{3^{1+i^2}}$ *:*

```
np.sum([i/(3**(1+i**2)) for i in range(17,2022)])
```

Cette commande fonctionne même si on prend $n = 0$. En effet, dans ce cas, on fait le produit d'une liste vide et celui-ci vaut 1.

- *Pour tout* $n \in \mathbb{N}^*$, $n! = \displaystyle\prod_{k=1}^{n} k$ *donc pour calculer* $n!$ *:*

```
np.prod([k for k in range(1,n+1)])
```

On n'utilise surtout pas la formule avec les trois factorielles. On verra pourquoi en exercice.

- *Pour tous* $n \in \mathbb{N}^*$ *et* $p \in [\![0\,;n]\!]$,

$$\binom{n}{p} = \frac{n(n-1)(n-2)\cdots(n-p+1)}{p(p-1)(p-2)\cdots 1} = \prod_{k=0}^{p-1} \frac{n-k}{p-k}.$$

  Ainsi pour calculer $\binom{n}{p}$ *:*

```
np.prod([(n-k)/(p-k) for k in range(p)])
```

On peut même calculer des sommes doubles, qui ne sont rien d'autres que des sommes de sommes.

- *Pour calculer* $T_n = \sum_{1 \leqslant i < j \leqslant n} ij$:

```
S=0
for i in range(1,n):
    for j in range(i+1,n+1):
        S=S+i*j
```

ou encore (si on ne veut pas prendre la peine de réfléchir à comment couper en deux sommes) :

```
S=0
for i in range(1,n+1):
    for j in range(1,n+1):
        if i<j:
            S=S+i*j
```

2) Suites réelles avec Python

L'idée générale est d'écrire un programme avec une variable `u` initialisée en le premier terme de la suite. On fait ensuite une boucle `for` (si on veut calculer le terme d'un certain rang donné) ou une boucle `while` (si on veut calculer un terme vérifiant une certaine condition) telle que, à chaque étape, on transforme `u` en le terme suivant.

L'outil informatique est précieux pour calculer des approximations des termes successifs d'une suite définie par récurrence lorsqu'on ne peut pas trouver une formule générale explicite. Il permet aussi de représenter graphiquement l'évolution d'une suite et de conjecturer éventuellement sa nature et la valeur de sa limite (si elle existe). Supposons que $(u_n)_{n\in\mathbb{N}}$ est une suite telle que, pour tout $n \in \mathbb{N}$, $u_{n+1} = F(u_n, n)$ avec F une fonction de $\mathbb{R}^2$ dans $\mathbb{R}$.

Par exemple :

— *Si* $F : (x, y) \longmapsto 3x - 1$, *alors* $u_{n+1} = 3u_n - 1$ *pour tout* $n \in \mathbb{N}$.

— *Si* $F : (x, y) \longmapsto \dfrac{x^2}{y+1}$, *alors* $u_{n+1} = \dfrac{u_n^2}{n+1}$ *pour tout* $n \in \mathbb{N}$.

On suppose que l'on a implémenté F au préalable en Python.

a) Calculer et représenter les premiers termes de la suite

Si le premier rang de la suite est n_0, on remplace `range(n)` par `range(n0,n)`.

Soient $n \in \mathbb{N}^*$ et $x \in \mathbb{R}$. Le script suivant calcule le $n^{\text{ième}}$ terme de la suite de terme initial x et le stocke dans la variable `u` :

```
u=x
for k in range(n):
    u=F(u,k)
```

Généralement on s'est donné $\varepsilon > 0$ petit et on a déterminé au préalable un entier n afin que $|u_n - \ell| \leqslant \varepsilon$, par exemple à l'aide de l'inégalité des accroissements finis.

Si la suite converge vers $\ell \in \mathbb{R}$ alors, lorsque n est grand, on obtient une approximation de ℓ. Le script suivant calcule les premiers (du $0^{\text{ième}}$ au $n^{\text{ième}}$) termes de la suite de terme initial x et les stocke dans un tableau `L` :

```
import numpy as np
u=x; L=np.zeros(n+1); L[0]=u
for k in range(n):
    u=F(u,k)#On calcule le terme suivant
    L[k+1]=u#On le met dans la coordonnée suivante
```

Dans le script précédent, on a commencé par créer une liste avec que des 0 et on a remplacé les 0 tour à tout par les valeurs successives de la suites. On aurait aussi pu créer une liste vide et lui ajouter à chaque étape le terme suivant :

```
u=x; L=[u]
for k in range(n):
    u=F(u,k)#On calcule le terme suivant
    L.append(u)#On l'ajoute à la liste
```

b) Le cas des suites récurrentes d'ordre 2

C'est un classique à connaître. On a besoin de deux variables que l'on met à jour à chaque étape puisque, pour calculer un terme, on a besoin des deux précédents.

Soient a, b, x, y des réels tels que $b \neq 0$. Considérons la suite $(u_n)_{n\in\mathbb{N}}$ telle que $u_0 = x$, $u_1 = y$ et

$$\forall n \in \mathbb{N}, \qquad u_{n+2} = au_{n+1} + bu_n.$$

Si $n \in \mathbb{N}^*$, le script suivant calcule le $n^{\text{ième}}$ terme de la suite (en renvoyant `u`) :

```
u=x; v=y
for k in range(n):
    w=v
    v=a*v+b*u#v contient le terme que l'on vient de calculer à l'aide des deux précédents
    u=w#u devient le terme précédent
```

Il fallait introduire une variable `w` intermédiaire car, en écrivant `v=a*v+b*u`, on perd l'ancienne valeur de `v`.

Autre possibilité, pour condenser :

```
u=x; v=y
for k in range(n):
    u,v=v,a*v+b*u
```

On peut aussi utiliser une liste contenant les valeurs précédentes :

```
L=np.zeros(n+1); L[0]=x; L[1]=y
for k in range(2,n+1):
    L[k]=a*L[k-1]+b*L[k-2]
```

c) Sommer ou multiplier les premiers termes d'une suite

On a déjà vu dans le paragraphe précédent comment calculer une valeur approchée d'une somme à l'aide de Python. Lorsqu'il s'agit d'une suite récurrente, on fait exactement comme dans les paragraphes précédents mais, à chaque étape de la boucle, on ajoute le nouveau terme aux précédents.

Le script suivant calcule la somme des premiers (du $0^{\text{ième}}$ au $n^{\text{ième}}$) termes de la suite de terme initial x et la stocke dans `s` :

```
u=x; s=u
for k in range(n):
    u=F(u,k)
    s=s+u
```

Remarques :

- On peut remplacer `s=s+u` par `s+=u`.
- Si on désire multiplier les termes d'une suite, on remplace juste `s=s+u` par `s=s*u` ou `s*=u`

Exemple : *Soit $x \in \mathbb{R}$. Pour tout $n \in \mathbb{N}$, posons*

$$S_n(x) = \sum_{k=0}^{n} \frac{x^k}{k!}.$$

On pourrait calculer cette somme comme dans le paragraphe précédent :

```
S=1//Le 0-ième terme
for k in range(1,n+1):
    S=S+x**k/np.prod([i for i in range(1,k+1)])
```

Si on connaît 72 ! et que l'on veut calculer 73 !, il est dommage de recommencer tout le calcul, alors qu'il suffit de multiplier 72 ! par 73...

C'est une mauvaise idée puisque, à chaque terme ajouté, on calcule la factorielle du dénominateur sans utiliser le fait qu'on l'avait presque calculée au rang d'avant. Même remarque pour la puissance. On va faire mieux : pour tout $k \in \mathbb{N}$, posons $u_k(x) = \frac{x^k}{k!}$. On remarque que $u_0(x) = 1$ et

$$\forall k \in \mathbb{N}, \qquad u_{k+1}(x) = \frac{x}{k+1} u_k(x).$$

Le script ci-dessous crée une fonction qui prend en entrée x et calcule $S_n(x)$:

```
import numpy as np
def S(n,x):
    u=1; s=1
    for k in range(n):
        u=x*u/(k+1)
        s=s+u
    return s
```

d) Calculer le plus petit rang pour lequel la suite vérifie une condition

Il est classique de chercher à calculer le nombre d'itérations nécessaires pour obtenir une approximation de la limite (la connaissant) d'une suite convergente avec une précision donnée. Si on connait la limite de la suite et qu'on la stocke dans une variable `lim`, alors le script suivant calcule le plus petit rang `n` tel que $|u_{\texttt{n}} - \texttt{lim}| \leqslant \texttt{eps}$:

```
u=x; n=0
while abs(u-lim)>eps:
    u=F(u,n)
    n=n+1
```

Si on sait que la suite tend vers $+\infty$, alors le script suivant calcule le plus petit rang `n` tel que $u_{\texttt{n}} \geqslant \texttt{A}$, pour `A` un réel :

```
u=x; n=0
while u<A:
    u=F(u,n)
    n=n+1
```

Exemple : *Pour tout $n \in \mathbb{N}^*$, posons $H_n = \sum_{k=1}^{n} \frac{1}{k}$. La suite $(H_n)_{n\in\mathbb{N}^*}$ tend vers $+\infty$. Écrivons une fonction en Python qui prend $A > 0$ en entrée et qui détermine le plus petit rang $n \in \mathbb{N}^*$ pour lequel $H_n \geqslant A$ (ce rang existe puisque la suite tend vers $+\infty$).*

La suite semble converger très lentement vers $+\infty$. C'est normal, on montrera dans le chapitre 13 que $\frac{H_n}{\ln(n)} \xrightarrow[n\to+\infty]{} 1$. Elle converge aussi lentement vers $+\infty$ que $(\ln(n))_{n\geqslant 1}$.

```
def Rang(A):
    H=1; n=1
    while H<A:
        H=H+1/(n+1)
        n=n+1
    return n
```

L'exécution de `Rang(10)` *renvoie* 12367, *l'exécution de* `Rang(15)` *renvoie* 1835421 et *l'exécution de* `Rang(20)` *renvoie* 272400600.

IV Représentations graphiques

Pour faire des représentations graphiques, on commence par importer la bibliothèque `matplotlib.pyplot` avec la commande

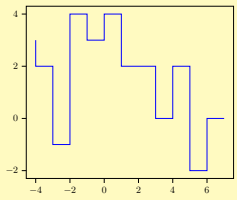
```
import matplotlib.pyplot as plt
```

1) Relier des points du plan

Considérons des points du plan dont on connaît les abscisses et les ordonnées, disons $A_1(x_1, y_1)$, $A_2(x_2, y_2)$, …, $A_n(x_n, y_n)$. On implémente ces points en Python en mettant $x_1, \ldots, x_n$ dans une liste ou un vecteur `X` et $y_1, \ldots, y_n$ dans une liste ou un vecteur `Y`. Alors

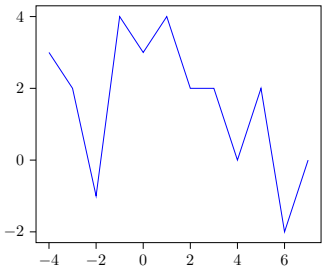
- **`np.plot(X,Y)`** relie les points $A_1, A_2, \ldots, A_n$ (dans cet ordre).
- **`plt.show()`** affiche la représentation graphique. Cette commande doit être mise à la toute fin (après avoir appelé différents `plt.plot` et ajouté d'éventuelles options).

On peut aussi tracer des fonctions en escalier avec `plt.step()` mais cette commande n'est pas officiellement au programme. Toutefois on l'utilisera dans la partie VII pour tracer des fonctions de répartitions. Avec les points de l'exemple ci-contre cela donnerait



Exemple :

```
1 import matplotlib.pyplot as plt
2 X=range(-4,8)
3 Y=[3,2,-1,4,3,4,2,2,0,2,-2,0]
4 plt.plot(X,Y)
5 plt.show()
```



2) Options de tracé

Les commandes de ce paragraphe ne sont pas au programme. Il est inutile le jour des concours de mettre de la couleur à des courbes, des légendes, etc. Mais en TP, on les utilise ne serait-ce que pour différencier les tracés lorsqu'on en superpose.

- Il est possible de préciser la couleur, le style de ligne et de symbole en ajoutant une chaîne de caractères en option dans la commande `plt.plot` :
  - pour la couleur : `b` (bleu), `g` (vert), `r` (rouge), `c` (cyan), `m` (magenta), `y` (jaune), `k` (noir), `w` (blanc).
  - pour le style : `-` (ligne continue), `--` (tirets), `:` (ligne en pointillé), `-.` (tirets points).
  - pour les symboles : `.` (point), `,` (pixel), `o` (cercle), `v` (triangle bas), `^` (triangle haut), `<` (triangle gauche), `>` (triangle droit), `s` (carré), `p` (pentagone), `h` (hexagone), `d` (diamant), `D` (gros diamant), `*` (étoile), `+` (plus), `x` (croix), etc.

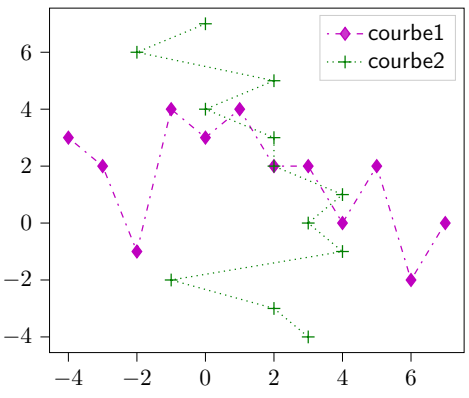
    Par exemple `plt.plot(X,Y,'r-*')` *affiche la courbe avec des étoiles rouges à chaque point et les relie en pointillés rouges.*

- Pour afficher une légende à une courbe, on ajoute l'option **`label('Légende_courbe')`** à l'intérieur de la commande `plt.plot` qui trace la courbe, où `'Légende_courbe'` est une chaîne de caractères contenant la légende. Ensuite on place la commande **`plt.legend()`**.
- Pour afficher un titre à une fenêtre graphique, on ajoute la commande **`plt.title('Titre_graphique')`** où `'Titre_graphique'` est une chaîne de caractères contenant le titre.
- Si on fait appelle plusieurs fois à la commande `plt.plot`, les courbes successives seront tracées dans la même fenêtre. Si on veut changer de fenêtre, on utilise la commande **`plt.figure()`**. Dorénavant les tracés se feront dans une nouvelle fenêtre (la même tant qu'on ne fait pas appel de nouveau à `plt.figure()`).

Exemple :

```
1 X=range(-4,8)
2 Y=[3,2,-1,4,3,4,2,2,0,2,-2,0]
3 plt.plot(X,Y,'m-.D',
      label='courbe1')
4 plt.plot(Y,X,'g:+',
      label='courbe2')
5 plt.legend()
6 plt.title('Un exemple de tracé
      personnalisé')
7 plt.show()
```

Un exemple de tracé personnalisé



3) Tracer le graphe d'une fonction de $\mathbb{R}$ dans $\mathbb{R}$

Tracer la courbe représentative d'une fonction sur un intervalle, revient à tracer une infinité de points. Or il n'est évidemment pas possible pour un ordinateur de réaliser une infinité d'instructions. L'approche de Python pour tracer une courbe est de représenter un nombre fini de points et de les relier par une ligne continue. La puissance de calcul des ordinateurs permet de considérer un très grand nombre de points et d'obtenir ainsi une très bonne approximation de la courbe par une ligne brisée dont les abscisses sont très proches afin de donner l'illusion de courbe.

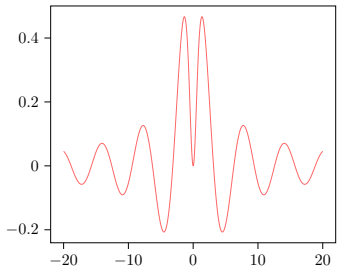
Soit f une fonction à valeurs réelles définie sur $[a\,;b]$ avec a et b deux réels tels que $a<b$. Supposons que l'on ait implémenté a, b, f en Python par `a,b,f` respectivement. Représenter graphiquement le graphe de f sur $[a\,;b]$ consiste en essentiellement trois étapes :

`X` contient donc un tableau contenant `n` nombres régulièrement espacés entre les réels `a` et `b`.

- On « discrétise » l'intervalle $[a\,;b]$ en un grands nombre de points régulièrement espacés (disons `n` avec, en général, `n`=1000 ou 10000) via la commande `X=np.linspace(a,b,n)`. Il s'agit du vecteur des abscisses.
- On crée le vecteur des ordonnées qui contient les images de tous les points du vecteur `X` par la fonction `f` via la commande `Y=[f(x) for x in X]`.
- Enfin, on trace la courbe via la commande `plt.plot(X,Y)` puis on l'affiche avec `plt.show()`

Exemple : *Représentons la fonction* $x\longmapsto \dfrac{x\sin(x)}{1+x^2}$ *sur* $[-20\,;20]$ *en rouge :*

```
def f(x):
    return x*np.sin(x)/(1+x**2)
X=np.linspace(-20,20,1000)
Y=[f(x) for x in X]
plt.plot(X,Y,'r')
plt.show()
```



Il y un cas particulier à connaître : celui consistant à tracer une droite. Un premier réflexe serait d'implémenter une fonction affine en Python puis d'effectuer la démarche précédente. Cela serait ultra fastidieux : pour tracer une droite, il suffit de relier deux points. Ainsi, si c et d sont deux réels implémentés en Python par `c` et `d` respectivement,

i.e. la droite d'équation $y=\dfrac{d-c}{b-a}(x-a)+c$.

- la commande `plt.plot([a,b],[c,d])` trace la droite passant par les points de coordonnées (a,c) et (b,d).
- la commande `plt.plot([a,b],[c,c])` trace la droite d'équation $y=c$. Cela est particulièrement pratique pour visualiser la convergence d'une suite (cf. paragraphe suivant) ou tracer une asymptote horizontale.

4) Représentation des termes d'une suite

Soit $n\in\mathbb{N}$. Représenter les $n+1$ premières valeurs d'une suite $(u_n)_{n\in\mathbb{N}}$ consiste à relier les points $(0,u_0)$, $(1,u_1)$, $(2,u_2)$, …, (n,u_n).

$n+1$ car il y a u_0.

Supposons que l'on ait stocké les $n+1$ premières valeurs d'une suite dans une liste `L`. Si on veut ensuite représenter graphiquement les termes en question, on utilise les commandes :

```
import matplotlib.pyplot as plt
plt.plot(range(n+1),L)
plt.show()
```

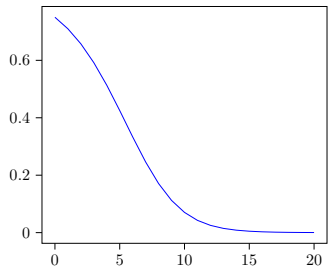
C'est le cas particulier de la suite du paragraphe V.3.b. Sur le graphique obtenu, on conjecture que la suite converge vers 0 (ce que l'on a montré).

Exemples :

- *Représentons les 21 premières valeurs de la suite $(u_n)_{n\in\mathbb{N}}$ telle que $u_0 = 3/4$ et*

$$\forall n \in \mathbb{N}, \qquad u_{n+1} = \frac{u_n^2}{2} + \frac{4}{7}u_n.$$

```
n=20; u=3/4;
L=np.zeros(n+1); L[0]=u
for k in range(n):
    u=u**2/2+4*u/7
    L[k+1]=u
plt.plot(range(n+1),L,'b')
plt.show()
```

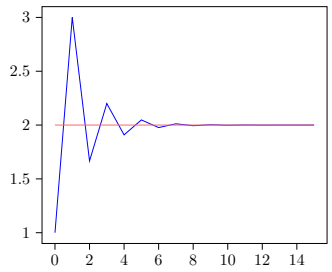


On conjecture que la suite converge vers 2. Pour aider à la lecture, on a superposé la droite d'ordonnée 2.

- *Représentons les 16 premières valeurs de la suite $(u_n)_{n\in\mathbb{N}}$ telle que $u_0 = 1$ et*

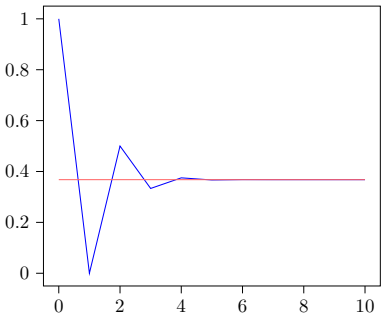
$$\forall n \in \mathbb{N}, \qquad u_{n+1} = 1 + \frac{2}{u_n}.$$

```
n=16; u=1;
L=np.zeros(n+1); L[0]=u
for k in range(n):
    u=1+2/u
    L[k+1]=u
plt.plot(range(n+1),L,'b')
plt.plot([0,15],[2,2],'r')
plt.show()
```



- *Soit $x \in \mathbb{R}$. Reprenons l'exemple de $S_n(x) = \sum_{k=0}^{n} \frac{x^k}{k!}$, $n \in \mathbb{N}$. Le script ci-dessous crée une fonction qui prend en entrée x et qui représente graphiquement les $n+1$ premières valeurs de la suite $(S_n(x))_{n\in\mathbb{N}}$ et lui superpose la droite d'ordonnée e^x pour obtenir ce graphique. On conjecture bien que la somme tend effectivement vers e^x.*

```
import matplotlib.pyplot as plt
import numpy as np
def S(n,x):
    u=1; s=1; L=[s]
    for k in range(n):
        u=x*u/(k+1)
        s=s+u
        L.append(s)
    plt.plot(range(n+1),L,'b')
    Y=[np.exp(x),np.exp(x)]
    plt.plot([0,n],Y,'r')
    plt.show()
```



Ci-contre, on a exécuté `S(10,-1)`.

5) Diagrammes à barres et histogrammes

Cette commande est très utile mais n'apparaît pas officiellement dans le programme.

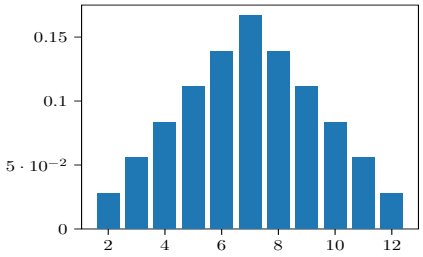
On peut aussi tracer des diagrammes à barres (ou en bâton) : à chaque valeur d'abscisses, on trace un bâton dont la longueur est l'ordonnée correspondante à l'abscisse. Pour cela on remplace `plt.plot()` par `plt.bar()`.

Exemple : *On a vu en cours que la loi d'une variable aléatoire finie représentant la somme de deux dés lancés simultanément est donnée par le tableau :*

k	2	3	4	5	6	7	8	9	10	11	12
p_k	$\frac{1}{36}$	$\frac{1}{18}$	$\frac{1}{12}$	$\frac{1}{9}$	$\frac{5}{36}$	$\frac{1}{6}$	$\frac{5}{36}$	$\frac{1}{9}$	$\frac{1}{12}$	$\frac{1}{18}$	$\frac{1}{36}$

Traçons en Python le diagramme à barres correspondant :

```
X=range(2,13)
Y=np.array
    ([1,2,3,4,5,6,5,4,3,2,1])/36
plt.bar(X,Y)
plt.show()
```



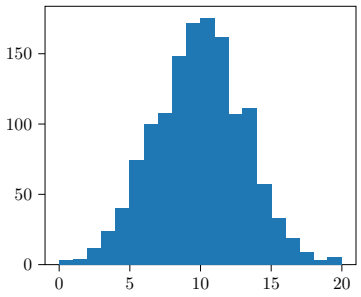
Nous y reviendrons dans la partie VII.

6) Histogramme

Si on dispose d'une liste (ou d'un vecteur) `L` contenant des données numériques, on peut tracer son histogramme avec la commande **`plt.hist(L)`**. Cette commande trie les données en différentes classes et, pour chaque classe, elle représente un bâton dont la hauteur est le nombre de données dans cette classe. On peut préciser le nombre de classes voulues en ajoutant l'option `bins` (mais les options des histogrammes ne sont pas exigibles).

On peut également renormaliser un histogramme en lui ajoutant l'option `density=True` de telle sorte que l'aire qu'il occupe soit égale à 1. C'est particulièrement pratique pour lui superposer un diagramme à barres de la loi d'une variable aléatoire ou une courbe de densité (cf. TP de deuxième année).

Exemple : *Considérons une épreuve passée par 1367 élèves. Les notes vont de 0 à 20 (et ne sont pas nécessairement des entiers) et on les a rangées dans un tableau Python appelé `NOTES`. Pour bien visualiser la répartition des notes, on peut construire un histogramme avec 20 classes (il y aura donc 20 classes : les notes dans* $[0\,;1[$*, celles dans* $[1\,;2[$*, etc. celles dans* $[18\,;19[$ *et enfin celles dans* $[19\,;20]$*).*




```
plt.hist(NOTES, bins=20)
plt.show()
```

V Matrices et systèmes avec Python

Pour créer et manipuler des matrices, on commence par importer la bibliothèque `numpy` avec la commande

`import numpy as np`

et la bibliothèque `numpy.linalg` avec la commande

`import numpy.linalg as al`

Cette dernière contient des commandes supplémentaires pour manipuler des matrices, notamment l'inversion de matrices ou la résolution de systèmes linéaires.

1) Création d'une matrice

Il y a une différence entre `np.array(L1)` et `np.array([L1])`. La première crée un vecteur (tableau unidimensionnel) et la deuxième créer une matrice (tableau bidimensionnel) ayant une seule ligne.

Soient `L1,L2,L3,...,Ln` des listes ou des vecteurs lignes ayant le même nombre de coordonnées. La commande **`np.array([L1,L2,L3,...,Ln])`** renvoie une matrice (tableau bidimensionnel) dont les lignes sont, dans l'ordre, `L1,L2,L3,...,Ln`.

Par exemple `np.array([[1,-1,0,2],[0,7,8,-5],[4,6,0,0]])` *implémente* $\begin{pmatrix} 1 & -1 & 0 & 2 \\ 0 & 7 & 8 & -5 \\ 4 & 6 & 0 & 0 \end{pmatrix}$.

On peut créer ainsi des matrices « à la main » mais on peut aller plus vite pour des matrices remarquables (surtout si il y a beaucoup de coordonnées). Soient n et p des entiers strictement positifs implémentés en Python par `n` et `p` respectivement. La commande

- `np.zeros([n,p])` renvoie la matrice nulle à n lignes et p colonnes.
- `np.ones([n,p])` renvoie la matrice à n lignes et p colonnes dont tous les coefficients sont des 1.
- `np.eye(n)` renvoie la matrice identité d'ordre n.

2) Extraction et modification

Soient A une matrice et i, j deux entiers naturels implémentés en Python par `A`, `i` et `j`.

Encore une fois Python numérote à partir de 0 et non de 1.

Avant toute chose, si on ne connaît pas la taille de la matrice A, on utilise la commande `a,b=np.shape(A)` qui stocke dans `a` le nombre de lignes de A et dans `b` le nombre de colonnes de A.

- `A[i,j]` renvoie le coefficient d'indice $(i+1, j+1)$ de A. Ainsi, pour accéder au coefficient d'indice (i,j) de A, la commande est `A[i-1,j-1]`.
- `A[i,:]` renvoie la $(i+1)^{\text{ième}}$ ligne de A. Ainsi, pour accéder à la $i^{\text{ième}}$ ligne de A, la commande est `A[i-1,:]`.
- `A[:,j]` renvoie la $(j+1)^{\text{ième}}$ colonne de A. Ainsi, pour accéder à la $j^{\text{ième}}$ colonne de A, la commande est `A[:,j-1]`.

Si on écrit `B=A`, alors `B` contient la matrice A bien sûr. Mais toute modification de la matrice `A` provoquera la même modification sur la matrice que `B` implémente (car c'est le même objet en fait). Si on veut éviter cela, il faut remplacer par `B=A` par `B=np.copy(A)` ou `B=A.copy()` (mais ces commandes ne sont pas exigibles) : les modifications de la matrice `A` n'auront alors pas d'effet sur la matrice `B`.

Ensuite on peut modifier des coefficients, des lignes et des colonnes en utilisant la syntaxe usuelle pour l'affectation dans une variable.

> *Par exemple* `L[2]=5` *remplace la* $3^{\text{ième}}$ *coordonnée de* L *par* 5, `A[0,3]=-7` *remplace le coefficient d'indice* $(1,4)$ *de* A *par* -7. *Si* A *a* 4 *colonnes, alors* `A[1,:]=np.ones(4)` *met des* 1 *à tous les coefficients de la* $2^{\text{ième}}$ *ligne de* A.

3) Opérations matricielles

Soient A et B deux matrices, c un réel et n un entier naturel implémentés en Python par `A`, `B`, `c` et `n`.

- Si A et B ont la même taille, alors `A+B` renvoie l'addition de A et B, `A-B` renvoie la soustraction de A par B et `c*B` renvoie la multiplication de A par c.
- `np.dot(A,B)` renvoie le produit matriciel de A par B, si celui-ci a un sens.
- `np.transpose(A)` renvoie la transposée de A.
- `al.inv(A)` renvoie l'inverse de A, si elle est inversible.
- `al.matrix_power(A,n)` renvoie A^n, si A est carrée (si n est un entier négatif et que A est inversible, elle renvoie $(A^{-1})^{-n}$).
- `al.rank(A)` renvoie le rang de A.
- Si A est carrée d'ordre n et B une matrice colonne à n lignes, la commande `al.solve(A,B)` renvoie la solution du système $AX = B$, s'il y en a une, et un message d'erreur sinon.

La commande `A*B` ne renvoie PAS le produit matriciel mais, si A et B ont la même taille, elle renvoie le produit coefficient par coefficient.

Cela fonctione aussi si `B` est un vecteur ou une liste à n coordonnées qui implémente le second membre

4) Opérations coefficient par coefficient

Soient A et B deux matrices de même taille et n un entier naturel implémentés en Python par `A`, `B`, `n`.

- On peut effectuer des opération coefficient par coefficient via les commandes `+,-,*,/,**`. Par conséquent `*` n'est PAS le produit matriciel et `**` n'est PAS le passage à la puissance.

  > *Par exemple,* `A**3` *renvoie la matrice* A *dont tous les coefficients ont été élevés à la puissance* 3, *ce qui n'est PAS a priori la matrice* A^3.

- On peut comparer A et B coefficient par coefficient ou comparer A coefficient par coefficient avec un nombre en utilisant les commandes `==,>,<,>=,<=,!=`. Le résultat est une matrice de booléens.

Par exemple, si `A=np.array([[1,-2,3],[0,-4,7]])` *et* `B=np.array([[-2,7,3],[-1,0,4]])`, alors

`A<=B` *renvoie* `array([[False, True, True], [False, True, False]])`

et `A>0` *renvoie* `array([[ True, False, True], [False, False, True]])`

VI Analyse numérique

1) Résolution de manière approchée de l'équation $f(x) = 0$

a) Méthode de dichotomie

Si on cherche c tel que $f(x) = m$, alors il suffit de considérer ce qui suit avec $\widetilde{f} = f - m$ au lieu de f : $\widetilde{f}(c) = 0$ si et seulement si $f(c) = m$.

Soient a et b deux réels tels que $a < b$ et f une fonction continue sur $[a\,;b]$ telle que 0 est compris entre $f(a)$ et $f(b)$. Le TVI nous garantit l'existence de $c \in [a\,;b]$ tel que $f(c) = 0$.

L'algorithme de dichotomie vu dans la démonstration du TVI permet de construire deux suites $(a_n)_{n\in\mathbb{N}}$ et $(b_n)_{n\in\mathbb{N}}$ adjacentes qui convergent vers un réel c de $[a\,;b]$ vérifiant $f(c) = 0$. Elle vérifient notamment $a_n \leqslant c \leqslant b_n$ pour tout $n \in \mathbb{N}$. A $\varepsilon > 0$ fixé, il suffit donc d'itérer l'algorithme jusqu'à ce que le rang n soit tel que $b_n - a_n \leqslant \varepsilon$. On a aura alors

$$0 \leqslant c - a_n \leqslant b_n - a_n \leqslant \varepsilon \qquad \text{et} \qquad 0 \leqslant b_n - c \leqslant b_n - a_n \leqslant \varepsilon,$$

i.e. a_n et b_n sont des approximations à ε près du point c recherché.

On dit que a_n est une approximation de c par défaut et b_n une approximation de c par excès.

Elle renvoie l'approximation donnée par b_n mais on aurait pu prendre celle donnée par a_n à la place.

La fonction suivante prend en entrée f, a, b, ε (f étant une fonction continue sur $[a\,;b]$ telle 0 est compris entre $f(a)$ et $f(b)$) et renvoie une approximation d'un point en lequel f s'annule à ε près.

```
def dichotomie(f,a,b,eps):
    while b-a>eps:
        c=(a+b)/2
        if f(a)*f(c)<0:#Si f change de signe entre a et c
            b=c#Alors f s'annule entre a et c
        else:
            a=c#Sinon f s'annule entre c et b
    return b
```

Si on veut construire un algorithme permettant de trouver une valeur approchée d'un point fixe de f (dont on a montré l'existence, voire l'unicité, au préalable), on remplace f par $g : x \longmapsto f(x) - x$ dans l'algorithme précédent.

Exemple : *La fonction $f : x \longmapsto x^2 - 2$ est continue sur $[1\,;2]$ et $f(1) < 0 < f(2)$. Puisqu'elle est strictement croissante sur $[1\,;2]$, le corollaire du TVI nous garantit même qu'elle s'annule un unique réel de $[1\,;2]$ (il s'agit de $\sqrt{2}$). Le script suivant*

```
def f(x):
    return x**2-2

dichotomie(f,1,2,0.0001)
```

renvoie `1.41424560546875`. *Ainsi $1,4142$ est une approximation de $\sqrt{2}$ à 10^{-4} près.*

Remarque : Dans la pratique, inutile de faire un algorithme général. Pour l'exemple précédent, on peut écrire immédiatement :

```
def dichotomie(eps):
    a=1;  b=2
    while b-a>eps:
        c=(a+b)/2
        if c**2>2:
            b=c
        else:
            a=c
    return c
```

b) Méthode de point fixe

Supposons que résoudre $f(x) = 0$ soit équivalent à résoudre l'équation $g(x) = x$, pour une certaine fonction g (il en existe forcément : par exemple $g : x \mapsto f(x) + x$), et donc équivalent à déterminer un point fixe de g. Soit $(x_n)_{n\in\mathbb{N}}$ la suite définie par $x_0 \in D_g$ (bien choisi) et, pour tout $n \in \mathbb{N}$, $x_{n+1} = g(x_n)$. On sait que, si $(x_n)_{n\in\mathbb{N}}$ converge vers un réel ℓ et si g est continue au voisinage en ℓ, alors ℓ est un point fixe de g.

Pour que la suite $(x_n)_{n\in\mathbb{N}}$ converge, il faut bien choisir la fonction g. C'est le cas si g est définie sur un intervalle fermé I vérifiant :

- $g(I) \subset I$,
- g est dérivable sur I,
- g' est bornée par un réel $M \in]0\,;1[$ sur I,
- g admet un point fixe x sur I.

On a même mieux sous les hypothèses précédentes :

$$\forall n \in \mathbb{N}, \qquad |x_n - x| \leqslant M^n\,|x_0 - x|.$$

On fait cette majoration préalable de $|x_0 - x|$ car il ne faut surtout pas que n_0 dépende de x. En effet, on ne connaît pas x et le but est de l'approcher. On ne va donc pas utiliser x pour trouver x...

On prend x_0 suffisamment proche de x afin que $|u_0 - x| \leqslant 1$ par exemple. On se donne $\varepsilon > 0$. On a

$$M^n \leqslant \varepsilon \qquad \Longleftrightarrow \qquad n\ln(M) \leqslant \ln(\varepsilon) \qquad \Longleftrightarrow \qquad n \geqslant \frac{\ln(\varepsilon)}{\ln(M)}.$$

Par conséquent, si $n_0 = \left\lfloor \dfrac{\ln(\varepsilon)}{\ln(M)} \right\rfloor + 1$, alors

$$|x_{n_0} - x| \leqslant M^{n_0}\,|x_0 - x| \leqslant \varepsilon \times 1 = \varepsilon.$$

La fonction suivante prend en entrée g, M, x_0, ε et renvoie une approximation de x à ε près.

```
import numpy as np
def MethodePointFixe(g,M,x0,eps):
    x=x0
    n=int(np.log(eps)/np.log(M))+1
    for k in range(n):
        x=g(x)
    return x
```

En général, cet algorithme permet de trouver une approximation de la solution à une précision donnée, plus rapidement que l'algorithme de dichotomie. Cependant il est difficile de savoir quelle fonction g choisir de telle sorte qu'elle remplisse les hypothèses ci-dessus. Alors que la méthode de dichotomie fonctionne à tous les coups !

Si f est de classe C^2 (deux fois dérivable et la dérivée seconde est continue), on peut néanmoins toujours trouver une fonction g qui convient grâce à la méthode de Newton (cf. poly d'exercices d'informatique).

Exemple : *On vérifie aisément que* $g : x \longmapsto \dfrac{x}{2} + \dfrac{1}{x}$ *est dérivable sur* $[1\,;2]$ *et que* $g([1\,;2]) \subset [1\,;2]$. *De plus elle* $\sqrt{2}$ *pour unique point fixe sur* $[1\,;2]$. *Pour tout* $x \in [1\,;2]$, *on a* $g'(x) = \dfrac{1}{2} - \dfrac{1}{x^2}$. *Comme* $\dfrac{1}{4} \leqslant \dfrac{1}{x^2} \leqslant 1$, *on obtient* $-\dfrac{1}{2} \leqslant g'(x) \leqslant \dfrac{1}{4}$. *Ainsi* g' *est bornée par* $M = \dfrac{1}{2}$ *sur* $[1\,;2]$. *Ainsi la fonction suivante prend en argument* ε *et renvoie une approximation de* $\sqrt{2}$ *à* ε *près :*

```
def PointFixeSQRT2(eps):
    x=1
    n=int(np.log(eps)/np.log(1/2))+1
    for k in range(n):
        x=x/2+1/x
    return x
```

La commande `PointFixeSQRT2(0.0001)` *renvoie* `1.414213562373095`.

2) Calcul approché d'intégrales

La méthode des rectangles est la méthode algorithmique consistant à approcher la valeur d'une intégrale (aire sous la courbe) par des sommes de Riemann (somme d'aires de rectangles).

Soit f une fonction continue sur $[a\,;b]$. Supposons que a, b, n, f ont été implémentées en Python dans les variables `a,b,n,f`. D'après le théorème de convergence des sommes de Riemann, le script ci-dessous calcule une valeur approchée de $\displaystyle\int_a^b f(t)\,dt$ (en utilisant la méthode des rectangles à droite) et la stocke dans la variable `S`.

```
S=0
for k in range(1,n+1):
    S=S+f(a+k*(b-a)/n)

S=S*(b-a)/n
```

ou plus simplement

```
S=np.sum([f(a+k*(b-a)/n) for k in range(1,n+1)])*(b-a)/n
```

Mais quelle valeur de n doit-on prendre pour avoir une valeur approchée à une précision donnée ?

a) Le cas où f est monotone

Supposons que f est croissante sur $[a\,;b]$. Notons $(S_n(f))_{n\geqslant 1}$ et $(T_n(f))_{n\geqslant 1}$ les suites des sommes de Riemann de f à droite et à gauche respectivement.

Soit $n\in\mathbb{N}^*$. Pour tous $k\in[\![1\,;n]\!]$ et $t\in\left[\dfrac{k-1}{n}\,;\dfrac{k}{n}\right]$, on a

$$f\left(a+\frac{(k-1)(b-a)}{n}\right)\leqslant f\left(a+t(b-a)\right)\leqslant f\left(a+\frac{k(b-a)}{n}\right)$$

donc, par croissance de l'intégrale,

$$\frac{1}{n}f\left(a+\frac{(k-1)(b-a)}{n}\right)\leqslant\int_{\frac{k}{n}}^{\frac{k-1}{n}} f\left(a+t(b-a)\right)\,dt\leqslant\frac{1}{n}f\left(a+\frac{k(b-a)}{n}\right).$$

On somme et on utilise la relation de Chasles :

$$\forall k\in[\![0\,;n-1]\!],\qquad T_n(f)\leqslant(b-a)\int_0^1 f\left(a+t(b-a)\right)\,dt\leqslant S_n(f).$$

Le changement de variable affine (donc de classe C^1) $t=\dfrac{x-a}{b-a}$ donne enfin :

$$T_n(f)\leqslant\int_a^b f(x)\,dx\leqslant S_n(f).$$

Pour obtenir une approximation de l'intégrale à ε près (avec $\varepsilon>0$), il suffit donc que n soit tel que $S_n(f)-T_n(f)\leqslant\varepsilon$. Or on remarque que

$$S_n(f)-T_n(f)=\frac{b-a}{n}\sum_{k=1}^{n}\left(f\left(a+k\frac{b-a}{n}\right)-f\left(a+(k-1)\frac{b-a}{n}\right)\right).$$

On reconnait une somme télescopique : $S_n(f)-T_n(f)=\dfrac{b-a}{n}\left(f(b)-f(a)\right)$. Ainsi

$$S_n(f)-T_n(f)\leqslant\varepsilon\qquad\Longleftrightarrow\qquad n\geqslant\frac{(b-a)(f(b)-f(a))}{\varepsilon}.$$

On choisit $n=\left\lfloor\dfrac{(b-a)(f(b)-f(a))}{\varepsilon}\right\rfloor+1$ et alors $S_n(f)$ et $T_n(f)$ sont des approximations de $\displaystyle\int_a^b f(t)\,dt$ à ε-près respectivement par excès et par défaut.

On remplace f par $-f$ dans la preuve ci-dessus.

Dans le cas où f est décroissante sur $[a\,;b]$, on choisit $n = \left\lfloor \dfrac{(b-a)(f(a)-f(b))}{\varepsilon} \right\rfloor + 1$ et alors $S_n(f)$ et $T_n(f)$ sont des approximations de $\displaystyle\int_a^b f(t)\,dt$ à ε-près respectivement par défaut et par excès.

Exemple : *La fonction $f : t \longmapsto \dfrac{1}{1+t}$ est continue et décroissante sur $[0\,;1]$. On a $\displaystyle\int_0^1 f(t)\,dt = [\ln(1+t)]_0^1 = \ln(2)$. Ainsi, pour tout $\varepsilon > 0$, si on prend*

$$n = \left\lfloor \frac{(1-0)(f(0)-f(1))}{\varepsilon} \right\rfloor + 1 = \left\lfloor \frac{1}{2\varepsilon} \right\rfloor + 1,$$

alors $S_n(f)$ et $T_n(f)$ sont des approximations de π à ε-près respectivement par excès et par défaut. La fonction suivante prend ε en entrée et renvoie une liste contenant une approximation de $\ln(2)$ par défaut et par excès à ε près.

```
def rectangle(eps):
    def f(t):
        return 1/(1+t)
    n=int(1/(2*eps))+1
    S=0; T=0
    for k in range(1,n+1):
        S=S+f(k/n)
        T=T+f((k-1)/n)
    return [S/n,T/n]
```

b) Le cas où f est de classe C^1

Si f est de classe C^1 sur $[a\,;b]$, alors le théorème d'approximation par les sommes de Riemann dans le cas C^1 nous assure que

$$\left| S_n(f) - \int_a^b f(x)\,dx \right| \leqslant \frac{(b-a)^2}{2n} \max_{[a\,;b]} |f'|.$$

Ainsi, pour tout $\varepsilon > 0$, il suffit de prendre $n = \left\lfloor \dfrac{(b-a)^2}{2\varepsilon} \max\limits_{[a\,;b]} |f'| \right\rfloor + 1$ pour que $S_n(f)$ soit une approximation de $\displaystyle\int_a^b f(x)\,dx$ à ε près.

C'est exactement la même valeur de n qu'avec la méthode précédente (ce qui n'est pas le cas en général a priori).

Exemple : *Reprenons l'exemple du paragraphe précédent. La fonction $f : t \longmapsto \dfrac{1}{1+t}$ est de classe C^1 sur $\mathbb{R}$. Sa dérivée $f' : t \longmapsto \dfrac{-1}{(1+t)^2}$ est borné par 1. Ainsi, on prend*

$$n = \left\lfloor \frac{(1-0)^2}{2\varepsilon} \times 1 \right\rfloor + 1 = \left\lfloor \frac{1}{2\varepsilon} \right\rfloor + 1.$$

On peut faire encore mieux avec plus d'hypothèses que f, comme on le verra en exercice.

VII Probabilités avec Python

Pour faire des simulations de variables aléatoires avec Python, on commence par importer la bibliothèque `numpy.random` avec la commande

```
import numpy.random as rd
```

1) Du hasard avec un ordinateur ?

Anyone who considers arithmetical methods of producing random digits is, of course, in a state of sin.

John von Neumann

John von Neumann (1903-1957) est un mathématicien et physicien américano-hongrois.

Simuler une variable aléatoire réelle d'une loi donnée à l'aide d'un ordinateur consiste à construire un nombre réel que l'on peut assimiler à une réalisation $X(\omega)$ d'une variable aléatoire X ayant cette loi.

Ce procédé repose en général sur deux étapes :

- Simuler des variables aléatoires indépendantes de loi uniforme sur $]0\,;1[$.
- Réaliser des opérations sur les simulations afin d'obtenir la réalisation d'une variable aléatoire de loi voulue.

Il existe plusieurs générateurs basés sur des phénomènes physiques (et certains sont brevetés et disponibles dans le commerce). Ils reposent par exemple sur des capteurs de bruit thermique dans les résistances de circuits électroniques, ou sur d'autres mécanismes basés sur la physique quantique, etc. Ces générateurs contiennent du vrai hasard mais ils sont encombrants, pas toujours fiables, pas facilement reproductibles et ne sont pas accessibles à une analyse mathématique rigoureuse.

La première étape pose des problèmes à la fois conceptuels et pratiques... notamment comment demander à un ordinateur de faire un choix aléatoire, alors que celui-ci est programmé pour n'effectuer qu'une suite d'instructions déterministes ? Aucune solution totalement satisfaisante n'a été proposée à ce jour. La plupart des générateurs sont fondés sur des calculs de congruences sur des grands nombres de façon déterministe (mais que l'on peut initialiser avec l'horloge de l'ordinateur pour ajouter une dose de hasard). On parle alors de générateurs de nombres pseudo-aléatoires : les nombres générés possèdent les propriétés apparentes d'une suite de variables aléatoires indépendantes et de loi uniforme sur $]0\,;1[$:

- La suite de nombres se comporte de façon chaotique, de sorte que les éléments successifs de cette suite de nombres semblent imprévisibles. Plus précisément, pour tout $k \in \mathbb{N}^*$, la connaissance des $k-1$ premiers termes de la suite ne semble pas nous donner d'information sur le $k^{\text{ième}}$ terme.
- La moyenne des premiers nombres de la suite semble se concentrer asymptotiquement (quand le nombre de termes moyennés tend vers $+\infty$) vers une quantité fixe.

Notamment il est totalement inadapté à des fins cryptographiques

Presque toutes les fonctions du module dépendent de la fonction de base `random()`, qui génère un réel dans $[0\,;1[$. Python utilise l'algorithme *Mersenne Twister* comme générateur de base. C'est l'un des générateurs de nombres aléatoires les plus largement testés qui existent. Cependant, étant complètement déterministe, il n'est pas adapté à tous les usages.

2) Simulation d'une variable aléatoire de loi discrète usuelle

a) Loi de Bernoulli

Supposons que $p \in]0\,;1[$ a été implémenté en Python dans la variable `p`. Pour obtenir une réalisation d'une variable aléatoire de loi de $\mathcal{B}(p)$, on peut utiliser la fonction :

```
X=rd.binomial(1,p)
```

Une autre possibilité est d'utiliser la commande `U=rd.random()` qui renvoie un réel choisi uniformément entre 0 et 1. Si on pose `X=1` lorsque `U<p` et `X=0` sinon, alors `X=1` avec fréquence p et `X=0` avec fréquence $1-p$, c'est-à-dire `X` contient une réalisation d'une variable aléatoire de loi $\mathcal{B}(p)$.

Cette deuxième possibilité, bien que plus compliquée au premier abord, est préférable car on peut facilement l'adapter si on veut autre chose que des 0 et 1 ou si on veut plusieurs alternatives à l'expérience.

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

b) Loi binomiale

Supposons que $n \in \mathbb{N}^*$ et $p \in]0\,;1[$ ont été implémentés en Python dans les variables `n` et `p`. Pour obtenir une réalisation d'une variable aléatoires de loi de $\mathcal{B}(n,p)$,

- on peut utiliser

```
X=rd.binomial(n,p)
```

- on peut utiliser simuler n variables de loi $\mathcal{B}(p)$ avec la `rd.random()` (cf. paragraphe précédent) et les sommer. En effet on peut considérer que les différents appels de cette fonction sont des réalisations de variables aléatoires mutuellement indépendants. Ainsi les sommer revient à compter le nombre de 1, donc compter le nombre de succès. On utilise alors :

Cette deuxième possibilité, bien que plus compliquée au premier abord, est préférable car on peut facilement l'adapter si on somme des succès d'expériences non indépendantes ou non identiques.

```
X=0
for k in range(n):
    if rd.random()<p:
        X=X+1
```

- on peut utiliser la commande `U=rd.random(n)` qui renvoie un vecteur contenant n réels choisis uniformément entre 0 et 1. La commande `U<p` renvoie un vecteur contenant des `True` aux coordonnées de `U` qui sont inférieurs à p et des `False` aux autres. Il suffit de sommer les coordonnées de `U<p` pour compter le nombre de `True` (le nombre de succès). Ainsi :

```
np.sum(rd.random(n)<p)
```

contient une réalisation d'une variable aléatoire de loi $\mathcal{B}(n,p)$.

c) Loi uniforme

Supposons que les réels a et b (tels que $a<b$) ont été implémentés en Python dans les variables `a` et `b`. Pour obtenir une réalisation d'une variable aléatoire de loi de $\mathcal{U}([\![a\,;b]\!])$, on peut utiliser la fonction :

```
X=rd.randint(a,b+1)
```

d) Loi géométrique

Pour une implémentation en Python, on commence par :

```
import numpy.random as rd
```

La deuxième possibilité, bien que plus compliquée au premier abord, est préférable car on peut facilement l'adapter si on s'intéresse au premier succès lors de la répétition d'expériences non indépendantes ou non identiques.

Supposons que $p \in]0\,;1[$ a été implémenté en Python dans la variable `p`. Pour obtenir une réalisation d'une variable aléatoires de loi de $\mathcal{G}(p)$, on peut utiliser la fonction :

```
X=rd.geometric(p)
```

Une autre possibilité est d'utiliser la commande `U=rd.random()` qui renvoie un réel choisi uniformément entre 0 et 1. La commande `U<=p` renvoie `True` avec fréquence p et `False` avec fréquence $1-p$. Tant qu'on obtient `False` (c'est-à-dire `U>p`), on fait appel à cette commande et on compte le nombre d'occurrences avant d'avoir `True`. Ainsi :

```
X=0
while rd.random()>p:
    X=X+1
X=X+1
```

On ajoute 1 pour compter l'ultime expérience conduisant au succès.

contient une réalisation d'une variable aléatoire de loi $\mathcal{G}(p)$.

e) Loi de Poisson

Pour une implémentation en Python, on commence par :

```
import numpy.random as rd
```

Supposons que $\lambda > 0$ a été implémenté en Python dans la variable `lam`. Pour obtenir une réalisation d'une variable aléatoire de loi de $\mathcal{P}(\lambda)$, on peut utiliser la fonction :

```
X=rd.poisson(lam)
```

Dans le prochain paragraphe, nous superposerons un histogramme construit à partir d'un grand nombre de réalisation d'une variable aléatoire de loi $\mathcal{B}(n, \lambda/n)$ (avec n grand) avec le diagramme à barre d'une loi $\mathcal{P}(\lambda)$ pour illustrer cette approximation.

On a aussi vu en cours que l'on peut approcher une loi Binomiale par une loi de Poisson. Comme lois binomiales sont faciles à simuler avec un ordinateur. Cela nous permet donc de simuler (de façon approchée) une loi de Poisson avec Python. Le code ci-dessous simule une réalisation d'une variable aléatoire qui suit approximativement une loi $\mathcal{P}(\texttt{lambda})$.

```
n=10000, p=lambda/n
X=0
for k in range(n):
    if rd.random()<p:
        X=X+1
```

f) Et les autres lois discrètes

On peut simuler n'importe quelle loi mais il n'y a rien à connaître de plus en ECG. Laissez-vous guider par les énoncés !

3) Vecteurs et matrices de variables de lois discrètes usuelles

On peut bien sûr utiliser des boucles `for` pour construire des listes contenant des variables aléatoires de lois usuelles en utilisant les commandes du paragraphe précédent. Pour les commandes `rd.binomial`, `rd.randint`, `rd.geometric` et `rd.poisson`, on peut plus simplement :

- rajouter pour paramètre `m` à l'une de ces commandes et on obtient un vecteur contenant m réalisations de variables aléatoires de la loi (que l'on peut considérer indépendantes).
- rajouter pour paramètre `[n,m]` à l'une de ces commandes et on obtient une matrice de taille $n \times m$ contenant des réalisations de variables aléatoires de la loi (que l'on peut considérer indépendantes).

Nous verrons les matrices dans la partie VI. Pour faire simple, ce sont des tableaux bidimensionnels (du type `ndarray`).

Exemples :

- `rd.binomial(10,0.3,7)` renvoie `array([2, 3, 3, 4, 5, 0, 3])`.
- `rd.binomial(10,0.3,[3,5])` renvoie
```
array([[4, 4, 5, 2, 4], .
       [2, 2, 2, 3, 5],
       [3, 3, 5, 3, 2]])
```
- `rd.geometric(0.2,8)` renvoie `array([10, 8, 1, 3, 3, 6, 11, 7])`.
- `rd.poisson(7,[4,6])` renvoie
```
array([[4,10, 7, 6, 4, 5], .
       [5, 5, 9, 6, 8, 4],
       [6, 8,10, 6, 9, 8],
       [5,10, 9, 2, 7, 4]])
```

4) Diagrammes à barres d'une variable aléatoire discrète

a) Le cas des variables discrètes finies

Dans la partie IV, nous avons vu comment tracer le diagramme à barres d'une variable aléatoire finie. Voyons l'exemple d'une variable aléatoire de loi binomiale.

Exemple : *Soient $n \in \mathbb{N}^*$ et $p \in]-1\,;1[$. Si $X \hookrightarrow \mathcal{B}(n,p)$, on a $\mathbb{P}(X=0) = (1-p)^n$ et on vérifie que, pour tout $k \in [\![1\,;n]\!]$,*

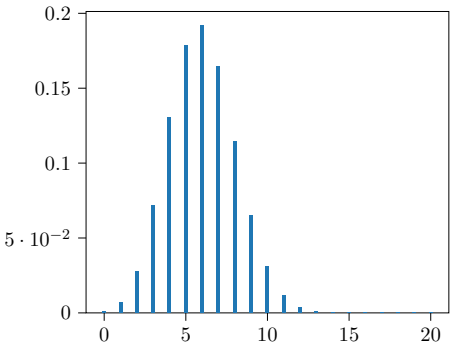
$$\mathbb{P}(X=k) = \frac{n-k+1}{k}\frac{p}{1-p}\mathbb{P}(X=k-1).$$

Il suffit de simplifier
$$\frac{\mathbb{P}(X=k)}{\mathbb{P}(X=k-1)}.$$

La fonction suivante prend n et p en entrée et trace le diagramme à barres d'une loi binomiale de paramètres n et p (avec un exemple quand $n=20$ et $p=0,3$) :

L'option `0.2` qui se trouve dans le `plt.bar` permet d'avoir des barres avec une plus petite largeur.

```
import numpy as np
import matplotlib.pyplot as plt
def DiagBarreBin(n,p):
    X=np.zeros(n+1)
    X[0]=(1-p)**n
    for k in range(1,n+1):
        c=(n-k+1)*p/(k*(1-p))
        X[k]=c*X[k-1]
    plt.bar(range(n+1),X,0.2)
    plt.show()
```



b) Le cas des variables discrètes infinies

Si X est une variable aléatoire discrète infinie, il est bien sûr plus possible de représenter toutes les barres mais, puisque la série $\sum \mathbb{P}(X=n)$ converge, on a $\mathbb{P}(X=n) \xrightarrow[n\to+\infty]{} 0$. Il suffit de ne tracer que les barres correspondant à des probabilités qui ne sont pas « trop petites » (celles qui sont supérieures à $\varepsilon = 10^{-3}$ par exemple).

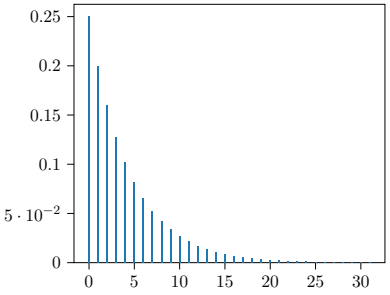
Exemples :

- *Soient $p \in]0\,;1[$ et $X \hookrightarrow \mathcal{G}(p)$, pour tout $n \in \mathbb{N}^*$, $\mathbb{P}(X>n) = (1-p)^n$. Donc, dans le diagramme, on peut se limiter aux $n_0^{\text{ième}}$ première barres avec $n_0 = \left\lfloor \dfrac{\ln(\varepsilon)}{\ln(1-p)} \right\rfloor + 1$.*

  De telle sorte que $(1-p)^{n_0} \leqslant \varepsilon$

  La fonction suivante prend p en entrée et trace le diagramme à barres d'une loi géométrique de paramètre p (avec un exemple quand $p=0,2$) :

```
def DiagBarreGeo(p):
    eps=10**(-3)
    n=int(np.log(eps)
          /np.log(1-p))+1
    X=[(1-p)**(k-1)*p
       for k in range(n+1)]
    plt.bar(range(n+1),X,0.2)
    plt.show()
```



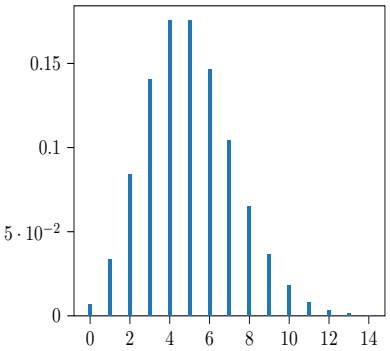
- *Soient $a>0$ et $X \hookrightarrow \mathcal{P}(a)$. Il est difficile de trouver mathématiquement le plus petit entier m_a tel que $\mathbb{P}(X=m_a) \leqslant \varepsilon$. Mais ce n'est pas un problème : dans l'algorithme, au lieu de construire la liste des probabilités des valeurs entre 0 et m_a, on construit la liste des probabilités jusqu'à ce qu'on rencontre une valeur inférieur à ε.*

On a $\mathbb{P}(X=0)=e^{-a}$ et, pour tout $k\in\mathbb{N}^*$,

$$\frac{\mathbb{P}(X=k)}{\mathbb{P}(X=k-1)}=\frac{a}{k}.$$

La fonction suivante prend a en entrée et trace le diagramme à barres d'une loi de Poisson de paramètre a (avec un exemple quand $a=5$) :

```
def DiagBarrePoi(a):
    esp=10**(-3)
    prob=np.exp(-a)
    X=[prob]
    k=0
    while prob>esp:
        k=k+1
        prob=prob*a/k
        X.append(prob)
    plt.bar(range(k+1),X,0.2)
    plt.show()
```



5) Tracé de fonctions de répartition

a) Le cas des variables finies

La fonction de répartition d'une variable aléatoire réelle finie X est une fonction en escalier qui ne prend qu'un nombre fini de valeurs. Il suffit de connaître les probabilités $\mathbb{P}(X=x)$ pour chaque $x\in X(\Omega)$ pour la construire.

Soyons plus précis : supposons que $X(\Omega)=\{x_1,\dots,x_n\}$ avec $x_1<x_2<\cdots<x_n$.

- F_X est nulle sur $]-\infty\,;x_1[$.
- Pour tout $i\in[\![1\,;n-1]\!]$, F_x est constante égale à $\displaystyle\sum_{k=1}^{i}\mathbb{P}(X=x_k)$ sur $[x_i\,;x_{i+1}[$.
- F_X est constante égale à 1 sur $[x_n\,;+\infty[$.

Supposons que l'on ait construit une liste ou un vecteur `X` contenant $x_1,\dots,x_n$ et une liste ou un vecteur `L` en Python qui contient $\mathbb{P}(X=x_1),\mathbb{P}(X=x_2),\dots,\mathbb{P}(X=x_n)$.

- On en déduit un vecteur content $F_X(x_1),F_X(x_2),\dots,F_X(x_n)$ via la commande **`np.cumsum(L)`**. Mais on peut aussi le construire « à la main » :

```
F=np.zeros(n)
F[0]=L[0]
for k in range(1,n):
    F[k]=F[k-1]+L[k]
```

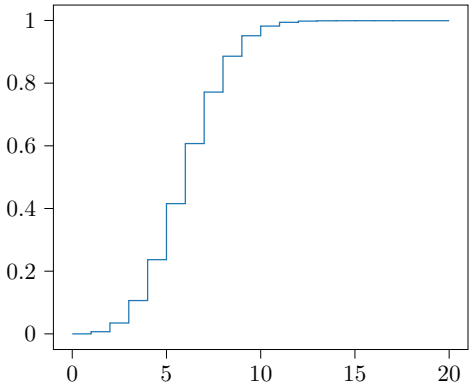
`plt.step` ne figure pas dans le programme...

- On utilise ensuite la commande `plt.step(X,F,where='post')` qui trace une courbe en escalier à partir des valeurs de `X` et `F`. La fonction de répartition est tracée !

Exemple : *On reprend l'exemple du paragraphe précédent avec la loi binomiale que l'on modifie légèrement pour tracer la fonction de répartition (avec un exemple quand $n=20$ et $p=0,3$) :*

Les lignes verticales au niveau des sauts sont dues au fonctionnement de la fonction `plt.plot`.

```
def FoncRepBin(n,p):
    X=np.zeros(n+1)
    F=np.zeros(n+1)
    X[0]=(1-p)**n
    F[0]=(1-p)**n
    for k in range(1,n+1):
        c=(n-k+1)*p/(k*(1-p))
        X[k]=c*X[k-1]
        F[k]=F[k-1]+X[k]
    plt.step(range(0,n+1),F,
    where='post')
    plt.show()
```



Remarque : On peut aussi construire la fonction de répartition « à la main » directement à partir des listes `X` et `L` (définies ci-dessus) puis la tracer avec `plt.plot` :

```
def FoncRep(X,L,t):
    if t<X[0]:
        return 0
    elif t>=X[-1]:
        return 1
    else:
        i=1
        while t>=X[i]:#Quand cette boucle s'arrête, i est tel
    que X[i-1]<=t<X[i]
            i=i+1
        F=0
        for k in range(i):#On somme les probas
            F=F+L[k]
        return F

T=np.linspace(X[0]-1,X[-1]+1,1000)
TT=[FoncRep(X,L,t) for t in T]
plt.plot(T,TT)
plt.show()
```

Si on est gêné par le fait que les sauts soient reliés par des lignes verticales, on peut implémenter directement le tracé de la fonction de répartition qui n'est rien d'autre qu'une succession de tracés de droites horizontales :

```
def TraceFrep(X,L):
    n=len(X)
    y=0
    plt.plot([X[0]-1,X[0]],[0,0],'b')#Elle est nulle avant le
    premier saut
    for k in range(n-1):
        y=y+L[k]#On somme les probas
        plt.plot([X[k],X[k+1]],[y,y],'b')
    plt.plot([X[-1],X[-1]+1],[1,1],'b')#Pour qu'elle "continue"
     un peu après le dernier saut
    plt.show()

TraceFrep(X,L)
```

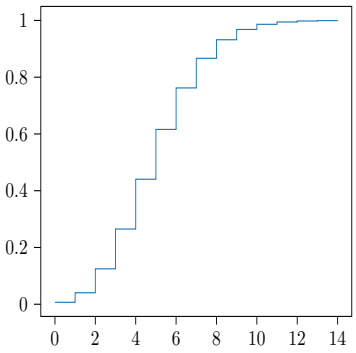
b) Le cas des variables discrètes infinies

On se limite dans ce paragraphe au cas d'une variable aléatoire discrète infinie X telle que $X(\Omega) \subset \mathbb{N}$. Comme pour les diagrammes à barres (cf. paragraphe 1), on se limite au tracé de la fonction de répartition sur l'intervalle $]0\,;n_0[$ avec $n_0 \in \mathbb{N}$ tel que $\mathbb{P}(X > n_0) \leqslant \varepsilon$ (avec par exemple $\varepsilon = 10^{-3}$).

Exemples : *On reprend les exemples usuels du paragraphe précédent, que l'on modifie légèrement pour tracer la fonction de répartition.*

- *Le cas d'une loi de Poisson (avec un exemple quand $a = 5$) :*

```
def FoncRepPoi(a):
    prob=np.exp(-a)
    frep=prob
    F=[frep]
    k=0
    while prob>10**(-3):
        k=k+1
        prob=prob*a/k
        frep=frep+prob
        F.append(frep)
    S=range(k+1)
    plt.step(S,F,where='post')
    plt.show()
```

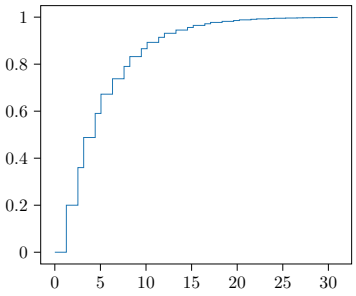


- *Pour une loi géométrique, on peut faire la même modification mais ici, on connaît une formule explicite pour la fonction de répartition d'une loi $\mathcal{G}(p)$. Il s'agit de :*

$$t \longmapsto \begin{cases} 1-(1-p)^{\lfloor t \rfloor} & \text{si} \quad t \geqslant 1 \\ 0 & \text{sinon.} \end{cases}$$

D'où le script Python (avec un exemple quand $p = 0,2$) :

```
def FoncRepGeo(p):
    def Frep(t):
        if t>=1:
            x=1-(1-p)**int(t)
        else:
            x=0
        return x
    n=int(-3*np.log(10)
              /np.log(1-p))+1
    Abs=np.linspace(0,n)
    F=[Frep(t) for t in Abs]
    plt.step(Abs,F,where='post')
    plt.show()
```



6) Statistiques d'échantillon

a) Statistiques descriptives

Soit `A` une liste, un vecteur ou une matrice de réels en Python.

- **`np.sum(A)`** renvoie la somme des coordonnées de `A`.
- **`np.prod(A)`** renvoie le produit des coordonnées de `A`.
- **`np.cumsum(A)`** renvoie le vecteur des sommes cumulées des coordonnées de `A`.

  Par exemple `np.cumsum([1,4,2,7])` *renvoie* `array([1,5,7,14])`.
- **`np.mean(A),np.var(A),np.std(A),np.median(A),`** renvoient respectivement la moyenne, la variance, l'écart type et la médiane des coordonnées de `A`.
- **`np.min(A),np.max(A)`** renvoient respectivement la plus petite et la plus grande des coordonnées de `A`.

Ces opérations peuvent s'appliquer sur une matrice entière ou bien pour chaque colonne (en ajoutant l'option `0`) ou chaque ligne (en ajoutant l'option `1`).

b) Statistiques d'échantillon

Considérons des données réelles associées à une expérience aléatoire (par exemple des réalisations d'une variable aléatoire). On peut ranger ces données dans un tableau dont la première ligne dresse la liste des différentes valeurs prises par les données dans l'ordre croissant et la deuxième ligne contient leurs effectifs respectifs (i.e. le nombre de fois qu'on retrouve chaque valeur parmi les données). En divisant chaque effectif par le nombre de données, on obtient leurs fréquences. Enfin on ajoute souvent une ligne correspondant aux fréquences cumulées.

La loi faible des grands nombres garantit que si les données sont des réalisations indépendantes d'une variable aléatoire discrète X admettant une variance, alors la moyenne empirique des données est une approximation de $\mathbb{E}(X)$. On en déduit que, si X admet une variance, alors la variance empirique des données est une approximation de $\mathbb{V}(X)$.

En effet la moyenne empirique des carrés des données approche $\mathbb{E}(X^2)$ et on conclut avec la formule de Koenig-Huygens.

On a vu dans la paragraphe IV.2 que, si A est un événement associé à une expérience et que, l'on réalise un grand nombre de fois cette expérience de façon indépendante, alors la proportion (ou fréquence) d'expériences ayant conduit à la réalisation de A (on voit la réalisation de A comme le succès de l'expérience de Bernoulli consistant à observer ou ne pas observer la réalisation de A) est une approximation de $\mathbb{P}(A)$. C'est une conséquence de la loi faible des grands nombres. Ainsi :

Ici $A = [X = x]$.

Ici $A = [X \leqslant x]$.

- Pour tout $x \in X(\Omega)$, la fréquence des données valant x approche $\mathbb{P}(X = x)$. Autrement dit le tableau des fréquences approche la loi de X.
- Pour tout $x \in X(\Omega)$, la fréquence des données inférieures à x approche $\mathbb{P}(X \leqslant x)$. Autrement dit le tableau des fréquences cumulées approche la fonction de répartition de X.

La convergence de la loi faible des grands nombres est très lente en pratique (d'où le $N = 10000$) tandis que l'approximation Binomiale/Poisson est très rapide (on l'utilise dès que $n \geqslant 30$ dans la pratique).

Voyons cela à travers un exemple : illustrons le fait que, si $\lambda > 0$ alors $\mathcal{B}(n, \lambda/n)$ est une approximation de la loi $\mathcal{P}(\lambda)$ lorsque n est grand (cf. paragraphe précédent).

- Prenons $n = 1000$ et $\lambda = 5$ puis construisons un vecteur contenant $N = 10000$ réalisations de $X \hookrightarrow \mathcal{B}(n, \lambda/n)$:

```
n=1000; lam=5; N=10000; X=rd.binomial(n,lam/n,N)
```

- Créons le vecteur des fréquences d'apparition des entiers de 0 à n dans les données (mais on peut se limiter au maximum m des données au lieu de n). Pour cela on crée un vecteur nul à $m+1$ coordonnées, on parcourt les données une à une et on incrémente de 1 la coordonnée qui lui correspond. Enfin on divise par N :

```
m=np.max(X); Eff=np.zeros(m+1)
for x in X:
    Eff[int(x)]=Eff[int(x)]+1
Freq=Eff/N
```

On utilise le fait que, pour tout $k \in \mathbb{N}$,

$$\frac{\mathbb{P}(Z = k+1)}{\mathbb{P}(Z = k)} = \frac{\lambda}{k+1}.$$

- Maintenant créons un vecteur contenant les valeurs de $\mathbb{P}(Z = k)$ pour $k \in [\![0; m]\!]$ et $Z \hookrightarrow \mathcal{P}(\lambda)$:

```
Loi=np.zeros(m+1); Loi[0]=np.exp(-lam)
for k in range(m):
    Loi[k+1]=Loi[k]*lam/(k+1)
```

Le `0.4` fait en sorte que les barres du deuxième diagramme soient plus fines.

- Enfin on superpose le diagramme en bâtons des fréquences avec celui de la loi :

```
plt.bar(range(m+1),Freq,label='frequences empiriques')
plt.bar(range(m+1),Loi,0.4,label='loi de Poisson')
plt.legend()
plt.show()
```

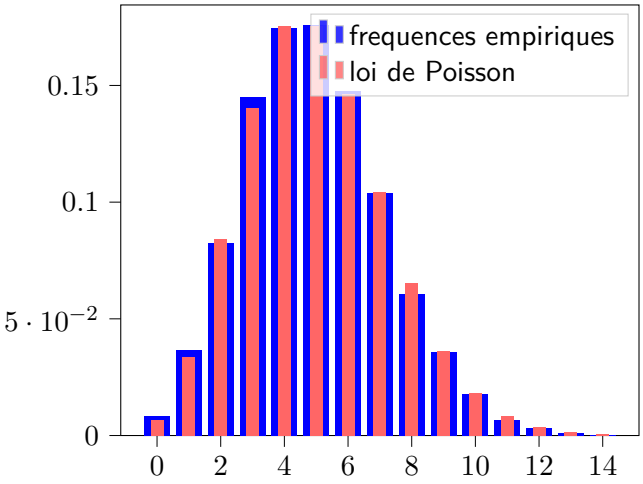
- A l'aide de la commande `np.cumsum`, qui permet de former le vecteur des sommes cumulées d'un vecteur, on peut superposer de même la courbe (en escalier) des fréquences cumulées et fonction de répartition de Z :

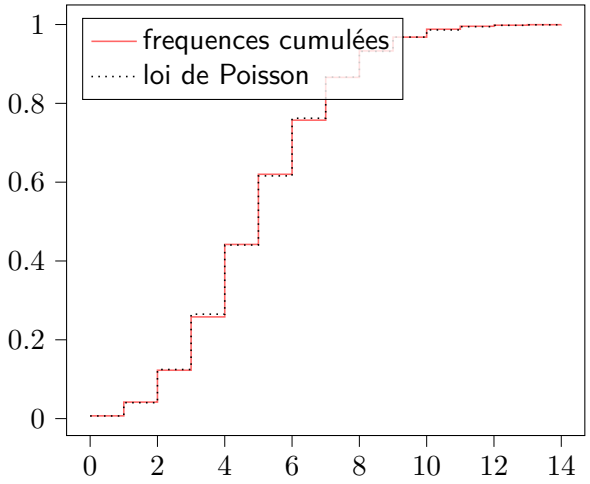
La commande `plt.step` n'est pas au programme (mais elle serait rappelée) : elle permet de tracer des fonctions en escalier.

```
FreqCum=np.cumsum(Freq); FonctRep=np.cumsum(Loi)
plt.step(range(m+1),FreqCum,'r',label='frequences cumulées',
    where='post',)
plt.step(range(m+1),FonctRep,'k:',label='loi de Poisson',
    where='post')
plt.legend()
plt.show()
```

Nous verrons en TP une autre méthode pour construire la courbe des fréquences cumulées, que nous appellerons plutôt fonction de répartition empirique.

Voici les deux graphiques que l'on obtient (à gauche celui des fréquences et à droite des fréquences cumulées). On constate bien qu'ils se superposent.





Pour tracer l'histogramme des fréquences empiriques, on aurait aussi pu utiliser directement `plt.hist` avec des options :

```
plt.hist(X,range(m+1),density=True,align='left',rwidth=0.7)
```

L'option `density=True` sert à ce que l'histogramme soit celui des effectifs. L'option `align='left'` sert à ce les barres soient centrées sur la valeur (oui le `left` n'est pas intuitif. Enfin l'option `rwidth = 0.7` sert à réduire la largeur des barres.

c) Approche expérimentale de la loi de Gauss

Soit $p \in]0\,;1[$. Pour tout $n \in \mathbb{N}^*$, considérons une variable aléatoire X_n de loi $\mathcal{B}(n,p)$. Un des théorèmes les plus importants de probabilités (qui sera vu en deuxième année) est le Théorème Central Limite. Un cas particulier est que, pour tous réels a et b tels que $a < b$,

$$\mathbb{P}\left(a \leqslant \frac{X_n - np}{\sqrt{np(1-p)}} \leqslant b\right) \xrightarrow[n\to+\infty]{} \int_a^b \frac{1}{\sqrt{2\pi}} e^{-x^2/2}\,dx.$$

Le théorème Central Limite est même l'un des plus importants des mathématiques tout court !

Interprétation graphique : quand n est grand, la courbe représentative de la fonction $\varphi : x \longmapsto \frac{1}{\sqrt{2\pi}} e^{-x^2/2}$ se superpose à l'histogramme d'observations de $\frac{X_n - np}{\sqrt{np(1-p)}}$.

L'option `bins=20` permet d'avoir 20 classes dans l'histogramme. L'option `density=True` permet de normaliser l'histogramme pour que son aire soit égale à 1.

```
n=10000; p=0.3
N=10000 #taille des observations
X=rd.binomial(n,p,N)
Y=(X-n*p)/np.sqrt(n*p*(1-p))
plt.hist(Y,bins=20,density=True,label="histogramme")
a=min(Y); b=max(Y); T=np.linspace(a,b,1000);
U=[np.exp(-t**2/2)/np.sqrt(2*np.pi) for t in T]
plt.plot(T,U,label="y=phi(x)")
plt.legend()
plt.show()
```