NumpyStatistiques_2

August 17, 2021

Probabilités et statistiques

Mémento

Syntaxe	Rôle
	simulation de variables aléatoires
import **numpy.random** as rd	importe le sous-module *random* de Numpy qui contient les fonctions statistiques
rd.random()	génére un nombre aléatoire sur $[0;1]$
rd.uniform(a,b,N)	génére une collection de N nombres aléatoires sur $[a;b]$ avec une loi uniforme
rd.normal(moy,$sigma$,N)	génére une collection de N nombres aléatoires suivant une loi normale (=gaussienne) de moyenne moy et d'écart-type $sigma$
rd.randint(a,b,N)	génère une collection de N **nombres entiers** entre a (inclus) et b (exclu)
	tracé d'histogrammes
import **matplotlib.pyplot** as plt	l'affichage d'histogramme est un outil de *pyplot*
plt.hist(L, bins = 'rice')	affiche l'histogramme de la collection de valeurs L (tableau numpy ou liste Python de valeurs) avec optimisation de la taille des bacs (option bins = 'rice')
	analyse statistique de données
import **numpy** as np	les outils statistiques sont inclus dans le module Numpy
np.mean(L)	calcul la **moyenne** de la collection de valeurs L (tableau numpy ou liste Python de valeurs)
np.std(L,ddof = 1)	calcul un estimateur non biaisé de l'**écart-type** pour la collection de valeurs L (tableau numpy ou liste Python de valeurs) le paramètre *ddof = 1* fait diviser par $N-1$ au lieu de diviser par N.
	régression affine

Syntaxe	Rôle
a,b = np.polyfit(xi,yi,deg = 1)	calcul les paramètres a,b de la regression affine des données yi = f(xi), c'est-à-dire la 'meilleure' droite d'équation $Y = aX + b$ permettant de décrire les couples de points expérimentaux (x_i, y_i). Ne pas oublier le paramètre $deg = 1$.

1 Importation des modules

L'ensemble des outils statistiques utilisés en CPGE sont contenus dans le module *random* de la bibliothèque Numpy.

On peut donc y accéder à l'aide des commandes d'importation suivantes.

[302]:
```
# 1ère solution
import numpy as np # crée un alias sur le module Numpy
np.random.random() # renvoie un flottant aléatoire uniformément distribué sur␣
↪l'intervalle [0;1]
```

[302]: 0.24557271625330468

[303]:
```
# 2ème solution
import numpy.random as rd # crée un alias sur le sous-module random de Numpy
rd.random() # c'est la fonction random() du sous-module random du module Numpy
```

[303]: 0.3076718349328402

1.1 La fonction random() du module numpy.random

L'aide sur cette fonction précise que la *borne supérieure est exclue.* Ceci est un détail car en pratique tout se passe comme si le tirage était uniforme dans l'intervalle $[0; 1]$ fermé.

[304]:
```
help(rd.random) # appel l'aide sur la fonction random() du module numpy.random
```

```
Help on built-in function random:

random(…) method of numpy.random.mtrand.RandomState instance
    random(size=None)

    Return random floats in the half-open interval [0.0, 1.0). Alias for
    `random_sample` to ease forward-porting to the new random API.
```

1.2 Comment créer une listes de valeurs uniformément distribuées dans l'intervalle $[0; 10]$?

Méthode 1 : on remplit une liste en ajoutant à chaque fois un nombre tiré aléatoirement dans l'intervalle $[0; 1]$ que l'on multiplie par 10.

[305]:
```
# Création de N = 12 valeurs aléatoires tirées uniformément dans l'intervalle␣
↪[0;10]
N = 12
L1 = []
for k in range(N): # boucle for
    L1.append(rd.random()*10)
print(L1)
```

```
[8.683949074719111, 9.138724458445656, 5.242008440654563, 3.6744636150753585,
2.121679957079392, 7.566105913077145, 1.3247160596582541, 8.004883714535154,
4.666424629287013, 0.8029572370961502, 0.7403223126154956, 5.364132910317223]
```

1.2.1 Exercice N2 n°1 : tirage uniforme dans un intervalle [a;b] quelconque

Modifier le code ci-dessous pour que les 12 valeurs soient tirées aléatoirement dans l'intervalle $[a; b]$ de manière uniforme. (On prendra $a = 50$ et $b = 80$).

[306]:
```
# Création de N = 12 valeurs aléatoires tirées uniformément dans l'intervalle␣
↪[a;b]
N = 12
a, b = 50, 80 # bornes de l'intervalle
L1 = []
for k in range(N): # boucle for
    L1.append(rd.random()) # LIGNE A MODIFIER
print(L1)
```

```
[0.08176401622012497, 0.6730857369481322, 0.19901315828484767,
0.9422466333240415, 0.5748532952584732, 0.9507814205541812, 0.46754529416076096,
0.9172203996655961, 0.7136960821868948, 0.7782308438347924, 0.8325502988746792,
0.9279284563964776]
```

[307]:
```
# Création de N = 12 valeurs aléatoires tirées uniformément dans l'intervalle␣
↪[a;b]
N = 12
a, b = 50, 80 # bornes de l'intervalle
L1 = []
for k in range(N): # boucle for
    L1.append(rd.random()*(b-a) + a)
print(L1)
```

```
[54.27127639766884, 73.10637705840173, 75.0380606251481, 52.320559430809325,
60.76528493316164, 78.53097198537375, 59.809432545750326, 78.18973704066465,
75.76625752197187, 70.83603659286864, 67.71096431289568, 56.05162673424219]
```

Méthode 2 : on utilise la fonction `rand()` qui admet pour argument le nombre de valeurs à tirer.

[308]:
```
rd.rand(12) # création d'un vecteur Numpy de 12 valeurs tirées uniformément␣
↪dans l'intervalle [0;1]
```

[308]:
```
array([0.75298098, 0.9669626 , 0.63703088, 0.31109888, 0.22924578,
       0.29123486, 0.49953487, 0.37769801, 0.47333005, 0.19691125,
       0.85439424, 0.77518606])
```

[309]:
```
rd.rand(12)*(b-a)+a # 12 valeurs aléatoires tirées entre les bornes a et b
```

[309]:
```
array([75.33873439, 68.10671307, 58.65968824, 79.63359889, 71.90856469,
       74.18405907, 59.15613951, 63.43485455, 70.59696256, 65.69548487,
       56.86739042, 59.48337174])
```

Comparaison des deux méthodes :

- La première méthode renvoie une liste Python (cet objet **n'accepte pas** les additions et/ou les multiplications par des scalaires).
- La seconde méthode renvoie un `ndarray`, c'est un "tableau numpy" qui se manipule comme des vecteurs (ou des matrices) mathématiques: les additions et multiplications par des scalaires sont possibles.

Remarque, on peut toujours convertir une liste Python L de valeurs numériques en objet `ndarray` grâce à la commade

`np.array(L) # convertit la liste python L en objet de type ndarray`

[310]:
```
print(L1)       # affichage de la liste Python de valeurs numériques
L2 = np.array(L1) # conversion de la liste Python L1 en un "tableau Numpy"
print(L2)       # affiche de l'objet de type "tableau Numpy"
```

```
[54.27127639766884, 73.10637705840173, 75.0380606251481, 52.320559430809325,
60.76528493316164, 78.53097198537375, 59.809432545750326, 78.18973704066465,
75.76625752197187, 70.83603659286864, 67.71096431289568, 56.05162673424219]
[54.2712764  73.10637706 75.03806063 52.32055943 60.76528493 78.53097199
 59.80943255 78.18973704 75.76625752 70.83603659 67.71096431 56.05162673]
```

2 Réaliser un tirage aléatoire de nombres entiers

Voici comment obtenir un nombre aléatoire tiré uniformément entre a=5 (inclus) et b = 10 (exclu).

[311]:
```
a, b = 5, 10
x = rd.randint(a,b)
print('nb tiré aléatoirement x = ',x)
```

```
nb tiré aléatoirement x =  6
```

ATTENTION : pour les entiers, il importe de **toujours vérifier dans la spécification de la fonction** si les bornes sont INCLUSES ou EXCLUES.

Selon les modules utilisés, les bornes supérieures sont parfois incluses ou exclues.

[312]:
```
L3 = []
for k in range(50): # 50 valeurs
    L3.append(rd.randint(5,10)) # ajoute un entier aléatoire à la liste L3
print(L3) # on peut vérifier qu'AUCUNE des 50 valeurs tirées n'est égale à 10
```

```
[9, 9, 5, 6, 6, 8, 6, 7, 6, 5, 6, 5, 9, 5, 8, 7, 7, 9, 7, 7, 9, 6, 7, 6, 7, 6,
7, 6, 6, 6, 7, 7, 7, 7, 5, 8, 7, 7, 9, 5, 8, 6, 7, 5, 8, 6, 7, 5, 5, 5]
```

[313]:
```
# Voici une autre syntaxe en utilisant une 'liste en compréhension'
L4 = [rd.randint(5,10) for k in range(50)]
print(L4)
```

```
[9, 7, 6, 8, 6, 5, 7, 7, 7, 5, 5, 7, 8, 8, 6, 6, 9, 5, 8, 7, 6, 7, 7, 8, 9, 9,
5, 9, 9, 5, 5, 5, 6, 9, 8, 5, 5, 9, 5, 9, 8, 7, 7, 9, 8, 7, 7, 5, 6, 6]
```

2.1 La bibliothèque random

Cette bibliothèque possède AUSSI une fonction `randint()` qui ne fonctione pas de la même manière: la borne supérieure est cette fois incluse.

[314]:
```
import random # il s'agit d'un autre module random qui n'est pas dans la␣
 ↪bibliothèque Numpy
L5 = [random.randint(5,10) for k in range(50)]
print(L5) # On peut vérifier que la valeur 10 est atteinte !
```

```
[7, 6, 9, 5, 10, 7, 10, 5, 5, 6, 10, 6, 10, 9, 8, 8, 6, 7, 8, 8, 7, 5, 7, 10, 7,
10, 10, 7, 9, 5, 9, 5, 6, 6, 10, 5, 6, 10, 7, 6, 6, 10, 7, 10, 6, 6, 10, 10, 10,
10]
```

A NOTER : Dans la mesure du possible, on recommande de travailler avec numpy.random mais il faut savoir s'adapter à la bibliothèque random si cela vous est demandé.*

3 Histogrammes

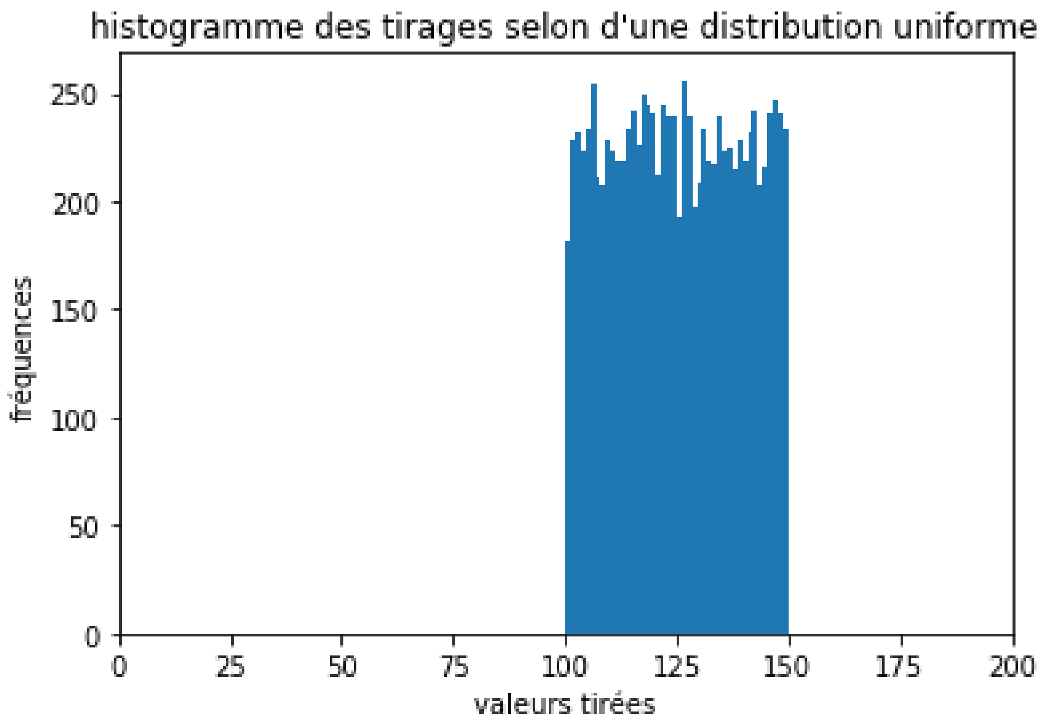
Un **histogramme** est une représentation graphique permettant de visualiser la répartition d'une variable continue en la représentant avec des colonnes.

Pour tracer un histogramme, nous utilisons la fonction `hist()` qui appartient au module matplotlib.pyplot qui contient les outils graphiques.

[315]:
```
import matplotlib.pyplot as plt # import du sous-module pyplot de matplotlib
```

[316]:
```
# Etape 1 : création d'une liste de N valeurs aléatoires uniformément␣
 ↪distribuées dans l'intervalle [100;150]
N = 10**4
L = [rd.random()*(150-100)+100 for k in range(N)] # L est une liste Python de␣
 ↪10000 flottants
```

[317]:
```python
# Etape 2 : tracé de l'histogramme avec la fonction hist
plt.hist(L,bins = 'rice') # l'option 'rice' ajuste automatiquement la taille␣
↪des colonnes
plt.xlim([0,200]) # modification des limites de l'axe horizontal
plt.title("histogramme des tirages selon d'une distribution uniforme")
plt.xlabel('valeurs tirées')
plt.ylabel('fréquences')
plt.show()
```



4 Estimateurs statistiques : moyenne, variance et écart-type

Un estimateur est une fonction permettant d'évaluer un paramètre inconnu relatif à une loi de probabilité.

Les deux estimateurs à notre programme sont :

- la moyenne, noté $\overline{x}$
- l'écart-type, noté σ_x.

4.1 L'estimateur 'moyenne'

Prenons comme exemple les données qui sont contenues dans la liste `L` précédemment générées à partir d'une loi de probabilité uniforme sur l'intervalle $[100; 150]$.

On peut **estimer** la 'valeur centrale' de cette loi à partir des N **réalisations** $\{x_k, \quad k = 1 \ldots N\}$.

Pour cela, on effectue le calcul de la moyenne $\overline{x}$ dont l'expression est la somme des valeurs divisée par le nombre de valeurs :

$$\overline{x} = \frac{1}{N}\sum_{k=1}^{N} x_k$$

On donne ci-dessous deux méthodes pour calculer la moyenne des N valeurs de la liste `L`.

[318]:
```
# 1ère méthode pour le cacul de la moyenne des valeurs d'une liste
moy = 0 # initialisation de la moyenne
for k in range(len(L)): # boucle sur les indices des éléments de la liste
 ↪(len(L) renvoie le nb d'éléments)
    moy = moy + L[k] # à chaque itération de la boucle, on additionne la k-ième
 ↪valeur de la liste
moy = moy / len(L) # on divise le résultat par le nombre d'éléments de L
print('moyenne = ',moy) # affichage du résultat
```

```
moyenne =  125.13124171125895
```

[319]:
```
# 2ème méthode pour le cacul de la moyenne des valeurs d'une liste
print('moyennne = ',np.mean(L)) # la méthode mean() du module Numpy donne
 ↪directement le résultat
```

```
moyennne =  125.13124171125877
```

Conclusion

On constate que la moyenne des N valeurs tirées est "proche" de la valeur centrale de l'intervalle $[100; 150]$.

Ainsi, la moyenne est une fonction des N réalisations de la loi $\overline{x} = f(\{x_k\})$ qui permet d'estimer la valeur centrale de la loi uniforme.

4.2 L'estimateur 'écart-type'

La dispersion des valeurs peut être quantifier par le calcul de l'écart-type.

Par définition, l'**écart-type** σ_x est la *racine carrée de la moyenne de l'écart quadratique à la moyenne*. En anglais, on dit aussi valeur RMS = **Root Mean Square**.

Un estimation de l'écart-type peut donc être calculé de la manière suivante:

(1) On soustrait chaque valeur x_k à la moyenne $\overline{x}$ des valeurs de

(2) On prend le carré de cet écart à la moyenne, $(x_k - \overline{x})^2$ représente un écart *quadratique*

(3) On prend la moyenne de ces écarts quadratiques (aussi appelée **variance**, notée $V(x)$):

$$V(x) = \frac{1}{N}\sum_{k=1}^{N} (x_k - \overline{x})^2$$

(4) Enfin, on prend la racine carrée de ce résultat:

$$\sigma_x = \sqrt{\frac{1}{N}\sum_{k=1}^{N}(x_k - \overline{x})^2}$$

On donne ci-dessous deux méthodes pour calculer l'écart-type des N valeurs de la liste `L`.

[320]:

```
# 1ère méthode pour le cacul de l'écart-type des valeurs d'une liste
moy = np.mean(L) # calcul de la moyenne des valeurs de L (en dehors de la␣
↪boucle !)
etype = 0 # initialisation de l'écart-type
for k in range(len(L)): # boucle sur les indices des éléments de la liste
    etype += (L[k]-moy)**2 # à chaque itération de la boucle, on additionne le␣
↪carré de l'écart à la moyenne
etype = etype / len(L) # on prend la moyenne de ces écarts au carré
etype = etype**(1/2) # on prend la racine carrée de cette moyenne
print('écart-type = ',etype) # affichage du résultat
```

```
écart-type =  14.394941249247399
```

[321]:

```
# 2ème méthode pour le cacul de l'écart-type des valeurs d'une liste
print('écart-type = ',np.std(L)) # appel à la méthode std (standard deviation)␣
↪du module Numpy
```

```
écart-type =  14.394941249247395
```

Remarque : écart-type sans biais

En théorie des probabilités (hors programme), on peut montrer que l'estimateur précédent n'est pas optimum : il possède un possède **un biais** d'autant plus important que le nombre d'échantillons N est faible.

C'est pourquoi nous utilisons (sauf indication contraire) l'estimateur suivant appelé **estimateur sans biais de l'écart-type**:

$$\sigma_x = \sqrt{\frac{1}{N-1}\sum_{k=1}^{N}(x_k - \overline{x})^2}$$

La calcul de cet estimateur se fait avec en appelant la méhode `std()` de Numpy avec le paramètre `ddof = 1`. Le paramètre ddof signifie "Delta Degree Of Freedom" (= nombres de degrés de libertés).

[322]:

```
print('écart-type sans biais = ',np.std(L,ddof = 1)) # estimateur non biaisé de␣
↪l'écart-type
```

```
écart-type sans biais =  14.395661050295386
```

A savoir

Pour une loi de probabilité uniforme sur un intervalle $[a;b]$, l'écart-type vaut la *demi-largeur de l'intervalle divisée par racine carrée de 3.*

On peut vérifier l'estimation obtenue pour l'écart-type est "proche" de la valeur théorique:

$$\frac{(b-a)/2}{\sqrt{3}} = \frac{25}{\sqrt{3}} \approx 14,44$$

4.3 Exercice N2 n°1 (corrigé)

a) Ecrire les instructions en python permettant de générer une liste X de N valeurs aléatoires tirées uniformément sur l'intervalle $[-2;12]$.

b) Calculer la moyenne et l'écart-type des valeurs de la liste pour $N = 10$, $N = 100$, $N = 10^4$ et $N = 10^6$.

c) Comparer les résultats obtenus avec les valeurs théoriques. Conclure

Correction

[323]:
```
# Question a, ici le nb d'échantillons N vaut 10
N = 10**1
X = [np.random.random()*(12-(-2))-2 for k in range(N)] # création de la liste␣
↪Python
print('moyenne = ',np.mean(X),              # calcul de la moyenne
      ' écart-type = ',np.std(X,ddof = 1)) # calcul de l'écart-type sans biais
```

```
moyenne =  3.9923299178578118  écart-type =  4.166980767349701
```

[324]:
```
## Question b, ici on utilise une boucle sur les valeurs de N pour envisager␣
↪les différentes valeurs
Nlist = [10, 100, 10**4, 10**6]
for N in Nlist : # boucle sur les valeurs de N dans la liste
    X = [np.random.random()*(12-(-2))-2 for k in range(N)] # création de la␣
↪liste Python de N valeurs
    print('N = ', N,                            # affichage de valeur de N
        '\t moyenne = ',np.mean(X),              # calcul de la moyenne
        ' \t écart-type = ',np.std(X,ddof = 1)) # calcul de l'écart-type sans␣
↪biais
```

```
N =  10          moyenne =  6.040098312764025    écart-type =  4.036028973974946
N =  100         moyenne =  5.404071725991465    écart-type =  4.384141963540723
N =  10000       moyenne =  4.948363832003256    écart-type =  4.037261715226672
N =  1000000     moyenne =  5.0040985805200355          écart-type =
4.03945972389119
```

[325]:
```
# Question c : comparaison avec les valeurs théoriques
moyTheorique = (12+(-2))/2 # moyenne des bornes de l'intervalle
etypeTheorique = (12-(-2))/2/np.sqrt(3) # demi-largeur de l'intervalle divisé␣
↪par racine carré de 3
```

```
print('valeurs théoriques : moyenne = ',moyTheorique, ' écart-type =␣
↪',etypeTheorique)
```

```
valeurs théoriques : moyenne =  5.0  écart-type =  4.041451884327381
```

Concusion : on constate que plus le nombre N d'échantillons est grand, plus les estimateurs semblent "proches" des valeurs théoriques.

4.4 Simulation d'une loi uniforme (type rectangulaire)

Dans le paragraphe précédent, nous avions généré N valeurs aléatoires tirées selon une loi uniforme à l'aide du tirage d'une unique valeur (fonction `random()`).

Le module `random` de Numpy contient la méthode `uniform` qui permet d'effectuer directement le tirage de N valeurs sur un intervalle $[a; b]$.

Attention, dans ce cas les valeurs générées sont de type `nd.array` (tableau Numpy) et ne sont plus une simple liste Python de valeurs comme c'était le cas dans le paragraphe précédent.

[326]:
```
import numpy as np
import matplotlib.pyplot as plt
```

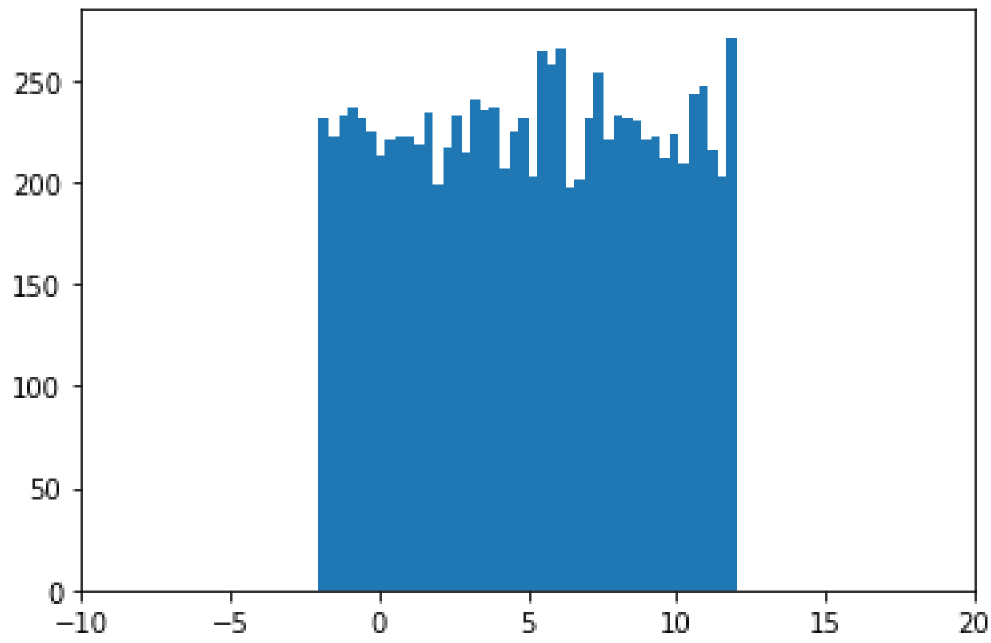
[327]:
```
help(np.random.uniform)
```

```
Help on built-in function uniform:

uniform(…) method of numpy.random.mtrand.RandomState instance
    uniform(low=0.0, high=1.0, size=None)

    Draw samples from a uniform distribution.

    Samples are uniformly distributed over the half-open interval
    ``[low, high)`` (includes low, but excludes high).  In other words,
    any value within the given interval is equally likely to be drawn
    by `uniform`.

    .. note::
        New code should use the ``uniform`` method of a ``default_rng()``
        instance instead; see `random-quick-start`.

    Parameters
    ----------
    low : float or array_like of floats, optional
        Lower boundary of the output interval.  All values generated will be
        greater than or equal to low.  The default value is 0.
    high : float or array_like of floats
        Upper boundary of the output interval.  All values generated will be
        less than high.  The default value is 1.0.
    size : int or tuple of ints, optional
```

```
    Output shape.  If the given shape is, e.g., ``(m, n, k)``, then
    ``m * n * k`` samples are drawn.  If size is ``None`` (default),
    a single value is returned if ``low`` and ``high`` are both scalars.
    Otherwise, ``np.broadcast(low, high).size`` samples are drawn.

Returns
-------
out : ndarray or scalar
    Drawn samples from the parameterized uniform distribution.

See Also
--------
randint : Discrete uniform distribution, yielding integers.
random_integers : Discrete uniform distribution over the closed
                  interval ``[low, high]``.
random_sample : Floats uniformly distributed over ``[0, 1)``.
random : Alias for `random_sample`.
rand : Convenience function that accepts dimensions as input, e.g.,
       ``rand(2,2)`` would generate a 2-by-2 array of floats,
       uniformly distributed over ``[0, 1)``.
Generator.uniform: which should be used for new code.

Notes
-----
The probability density function of the uniform distribution is

.. math:: p(x) = \frac{1}{b - a}

anywhere within the interval ``[a, b)``, and zero elsewhere.

When ``high`` == ``low``, values of ``low`` will be returned.
If ``high`` < ``low``, the results are officially undefined
and may eventually raise an error, i.e. do not rely on this
function to behave when passed arguments satisfying that
inequality condition.

Examples
--------
Draw samples from the distribution:

>>> s = np.random.uniform(-1,0,1000)

All values are within the given interval:

>>> np.all(s >= -1)
True
>>> np.all(s < 0)
True
```

```
Display the histogram of the samples, along with the
probability density function:

>>> import matplotlib.pyplot as plt
>>> count, bins, ignored = plt.hist(s, 15, density=True)
>>> plt.plot(bins, np.ones_like(bins), linewidth=2, color='r')
>>> plt.show()
```

[328]:
```
X1 = np.random.uniform(-2,12,10**4) # 10^4 valeurs dans l'intervalle [-2;12]
plt.hist(X1,bins = 'rice') # affichage de l'histogramme, rice = ajustement␣
↪automatique
plt.xlim([-10,20]) # modification des limites de l'axe X
plt.show()
```

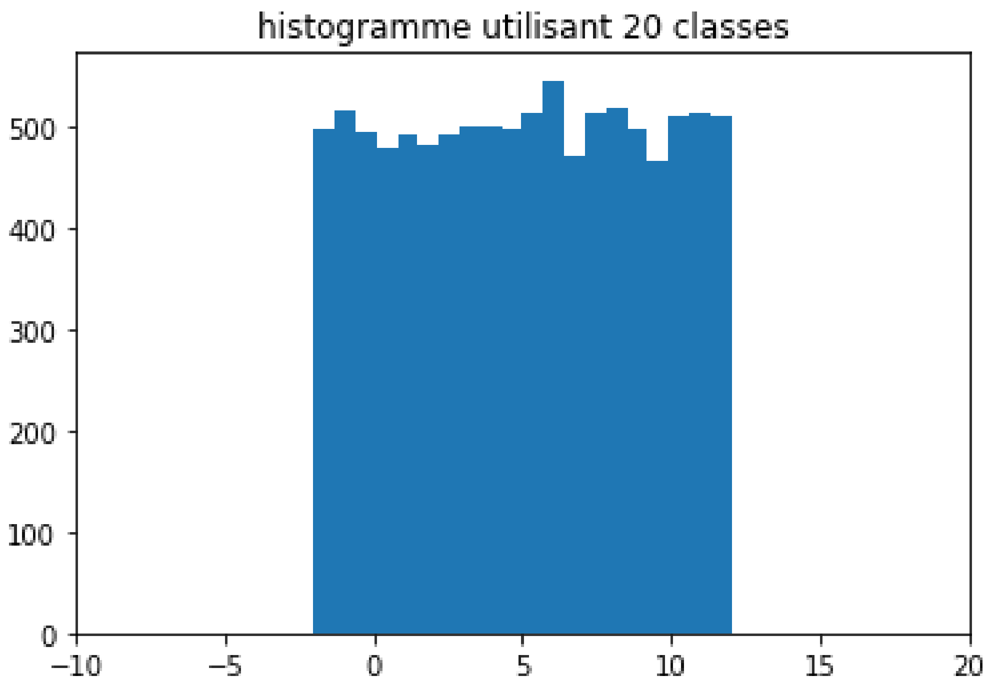


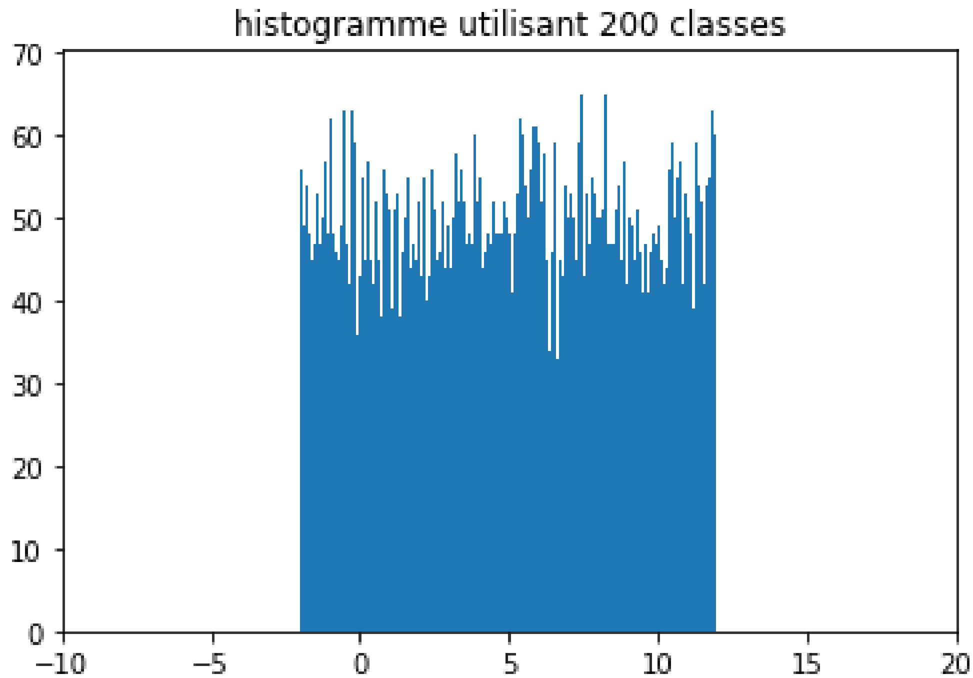
4.4.1 Influence du nombre de "bins" d'un histogramme

On peut spécifir le nombre de *bins* (= nb de classes, nb de "bacs") lors de la construction d'un histogramme.

[329]:
```
plt.figure(1)
plt.hist(X1,bins = 20) # affichage de l'histogramme, rice = ajustement␣
↪automatique
plt.xlim([-10,20]) # modification des limites de l'axe X
plt.title('histogramme utilisant 20 classes')
```

```
plt.show()

plt.figure(2)
plt.title('histogramme utilisant 200 classes')
plt.hist(X1,bins = 200) # affichage de l'histogramme, rice = ajustement␣
 ↪automatique
plt.xlim([-10,20]) # modification des limites de l'axe X
plt.show()
```





En conclusion, on voit que le nombre de "bins" doit être doit choisi de manière judicieuse pour représenter convenablement un échantillon de valeurs.

L'option `bins ='rice'` fournit automatiquement une valeur généralement acceptable.

4.5 Simulation d'une loi normale (type gaussienne)

Une **variable aléatoire gaussienne** (ou normale) décrit un processus aléatoire dont la probabilité d'obtenir une valeur numérique entre x et $x+\mathrm{d}x$ est $f(x)\mathrm{d}x$ où $f(x)$ est appelée *densité de probabilité normale* est donnée par:

$$f(x) = \frac{1}{\sqrt{2\pi}\sigma} \exp\left(-\frac{(x-\mu)^2}{2\sigma^2}\right)$$

Les paramètres μ et σ sont les deux paramètres de la loi normale qui sont appelées, respectivement, moyenne et écart-type.

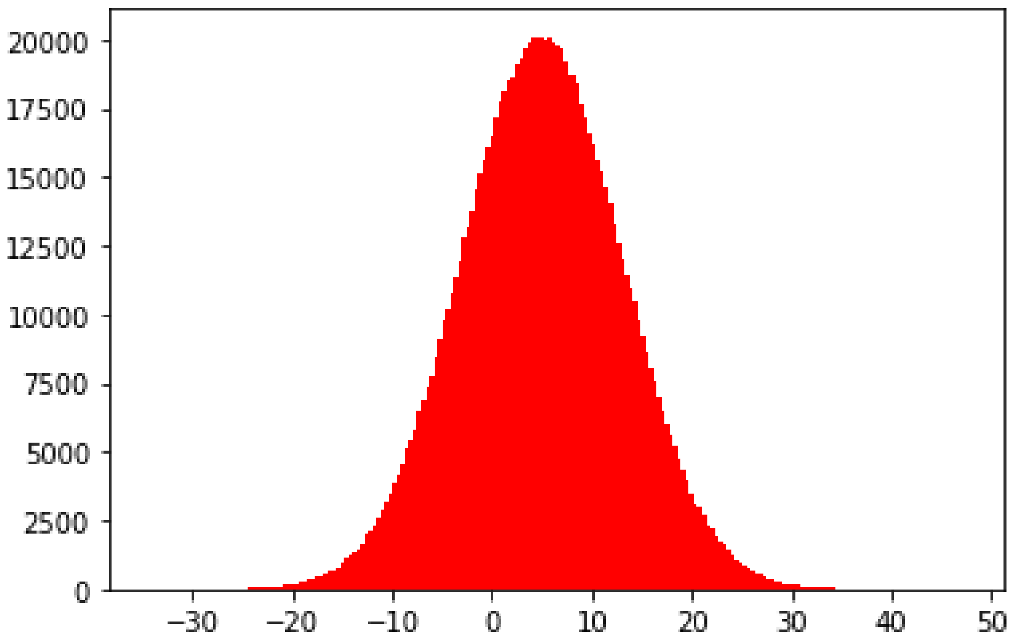
La courbe de cette densité de probabilité est appelée courbe de Gauss (ou *courbe en cloche*).

Pour simuler une loi gaussienne, on peut utiliser la fonction `normal()`.

[330]:
```
N = 10**6 # nombre de valeurs
X2 = np.random.normal(5,14/np.sqrt(3),N)  # Loi normale (distribution␣
 ↪gaussienne)
                                          # moyenne, écart-type, nombre de␣
 ↪tirages)
```

```
plt.hist(X2,bins='rice',color ='red')
plt.plot()
```

[330]: []



4.5.1 Autre méthode pour simuler un tirage aléatoire

A partir d'échantillons tirées selon une loi gaussienne de moyenne nulle et d'écart-type égal à un, il est possible **par multiplication et addition** d'obtenir des échantillons tirées selon une moyenne μ et un écart-type σ quelconques.

[187]:
```
N = 10**6 # nb d'échantillons
## Tirage de moyenne nulle et de variance unitaire
X1 = np.random.normal(0,1,N)

## Obtenu d'un tirage de moyenne moy = 5. et d'écart type sigma =2.
mu, sigma = 5., 2.
X2 = X1*sigma + mu # multiplication par sigma et 'translation' de moy
```

De même, à partir d'un tirage uniforme sur l'intervalle $[0; 1]$ il est possible d'obtenir tirage uniforme sur l'intervalle $[a; b]$.

4.6 O2 n°2 Exercice (corrigé)

Compléter le script ci-dessous de manière à créer, à partir de la la liste d'échantillons *X1*, une liste d'échantillons *X2* tirées uniformément sur l'intervalle $I = [x_0 - \Delta x; x_0 + \Delta x]$ avec $x_0 = 12$ et $\Delta x = 0,25$.

[331]:
```python
N = 50000 # nb d'échantillons
X1 = np.random.uniform(0,1,N) # tirage uniforme sur [0;1]

x0, deltax = 12., 0.25
X2 = # à compléter

plt.hist(X2,bins='rice')
plt.xlim([11,13]) # mise à l'échelle des x
plt.show()
```

```
  File "<ipython-input-331-1189e216caec>", line 5
    X2 = # à compléter
                      ^
SyntaxError: invalid syntax
```

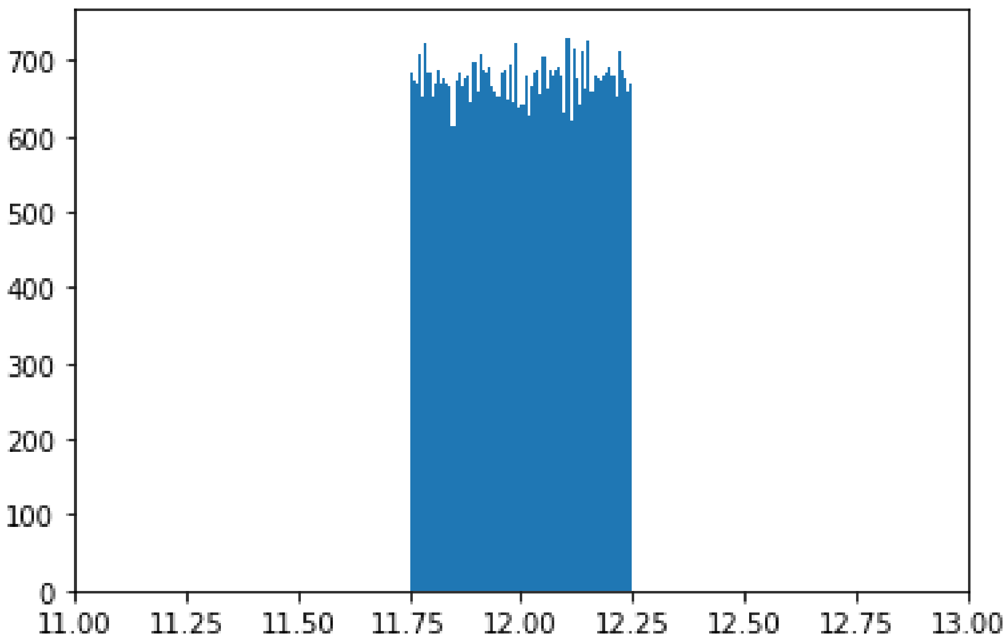
Correction La ligne correctement complétée est la suivante:

`X2 = (X1-0.5)*2*deltax + x0`

Voici le raisonnement utilisé :

- (X1-0.5) est une collection de valeurs centrées sur zéro, son étendue est $\pm 0,5$ autour de zéro,
- on multiplie alors les valeurs par $2\Delta x$ pour avoir la bonne largeur d'intervalle,
- enfin on applique une *translation* sur les valeurs en ajoutant la valeur centrale x_0 de l'intervalle.

[332]:
```python
N = 50000 # nb d'échantillons
X1 = np.random.uniform(0,1,N) # tirage uniforme sur [0;1]

x0, deltax = 12., 0.25
X2 = (X1-0.5)*2*deltax + x0

plt.hist(X2,bins='rice')
plt.xlim([11,13]) # mise à l'échelle des x
plt.show()
```



5 Régression linéaire (ou régression affine)

Supposons que l'on dispose de N couples valeurs (x_k, y_k) avec $k = 1, \ldots, N$.

Ces valeurs peuvent être représentées dans un plan en tant que nuage de point.

On dit que l'on effectue une régression linéaire sur les données $\{(x_k, y_k)$ avec $k = 1, \ldots, N\}$ lorsque l'on cherche à faire passer une droite "au mieux par tous les points" comme cela est illustré sur la figure ci-dessous.

Définition : Un modèle de régression linéaire est un modèle de régression qui cherche à établir une relation linéaire entre une variable, dite expliquée noté Y, et une ou plusieurs variables, dites explicatives (notée X).

5.1 La fonction polyfit de numpy

Pour effectuer la régression linéaire, nous utiliserons la fonction `polyfit()` du module Numpy.

[39]:
```
help(np.polyfit) # polyfit(xData,yData,degreDuPolynôme)
```

```
Help on function polyfit in module numpy:

polyfit(x, y, deg, rcond=None, full=False, w=None, cov=False)
    Least squares polynomial fit.

    Fit a polynomial ``p(x) = p[0] * x**deg + … + p[deg]`` of degree `deg`
    to points `(x, y)`. Returns a vector of coefficients `p` that minimises
```

```
the squared error in the order `deg`, `deg-1`, … `0`.

The `Polynomial.fit <numpy.polynomial.polynomial.Polynomial.fit>` class
method is recommended for new code as it is more stable numerically. See
the documentation of the method for more information.

Parameters
----------
x : array_like, shape (M,)
    x-coordinates of the M sample points ``(x[i], y[i])``.
y : array_like, shape (M,) or (M, K)
    y-coordinates of the sample points. Several data sets of sample
    points sharing the same x-coordinates can be fitted at once by
    passing in a 2D-array that contains one dataset per column.
deg : int
    Degree of the fitting polynomial
rcond : float, optional
    Relative condition number of the fit. Singular values smaller than
    this relative to the largest singular value will be ignored. The
    default value is len(x)*eps, where eps is the relative precision of
    the float type, about 2e-16 in most cases.
full : bool, optional
    Switch determining nature of return value. When it is False (the
    default) just the coefficients are returned, when True diagnostic
    information from the singular value decomposition is also returned.
w : array_like, shape (M,), optional
    Weights to apply to the y-coordinates of the sample points. For
    gaussian uncertainties, use 1/sigma (not 1/sigma**2).
cov : bool or str, optional
    If given and not `False`, return not just the estimate but also its
    covariance matrix. By default, the covariance are scaled by
    chi2/sqrt(N-dof), i.e., the weights are presumed to be unreliable
    except in a relative sense and everything is scaled such that the
    reduced chi2 is unity. This scaling is omitted if ``cov='unscaled'``,
    as is relevant for the case that the weights are 1/sigma**2, with
    sigma known to be a reliable estimate of the uncertainty.

Returns
-------
p : ndarray, shape (deg + 1,) or (deg + 1, K)
    Polynomial coefficients, highest power first.  If `y` was 2-D, the
    coefficients for `k`-th data set are in ``p[:,k]``.

residuals, rank, singular_values, rcond
    Present only if `full` = True.  Residuals is sum of squared residuals
    of the least-squares fit, the effective rank of the scaled Vandermonde
    coefficient matrix, its singular values, and the specified value of
    `rcond`. For more details, see `linalg.lstsq`.
```

```
V : ndarray, shape (M,M) or (M,M,K)
    Present only if `full` = False and `cov`=True.  The covariance
    matrix of the polynomial coefficient estimates.  The diagonal of
    this matrix are the variance estimates for each coefficient.  If y
    is a 2-D array, then the covariance matrix for the `k`-th data set
    are in ``V[:,:,k]``


Warns
-----
RankWarning
    The rank of the coefficient matrix in the least-squares fit is
    deficient. The warning is only raised if `full` = False.

    The warnings can be turned off by

    >>> import warnings
    >>> warnings.simplefilter('ignore', np.RankWarning)

See Also
--------
polyval : Compute polynomial values.
linalg.lstsq : Computes a least-squares fit.
scipy.interpolate.UnivariateSpline : Computes spline fits.

Notes
-----
The solution minimizes the squared error

.. math ::
    E = \sum_{j=0}^k |p(x_j) - y_j|^2

in the equations::

    x[0]**n * p[0] + … + x[0] * p[n-1] + p[n] = y[0]
    x[1]**n * p[0] + … + x[1] * p[n-1] + p[n] = y[1]
    …
    x[k]**n * p[0] + … + x[k] * p[n-1] + p[n] = y[k]

The coefficient matrix of the coefficients `p` is a Vandermonde matrix.

`polyfit` issues a `RankWarning` when the least-squares fit is badly
conditioned. This implies that the best fit is not well-defined due
to numerical error. The results may be improved by lowering the polynomial
degree or by replacing `x` by `x` - `x`.mean(). The `rcond` parameter
can also be set to a value smaller than its default, but the resulting
fit may be spurious: including contributions from the small singular
```

```
values can add numerical noise to the result.

Note that fitting polynomial coefficients is inherently badly conditioned
when the degree of the polynomial is large or the interval of sample points
is badly centered. The quality of the fit should always be checked in these
cases. When polynomial fits are not satisfactory, splines may be a good
alternative.

References
----------
.. [1] Wikipedia, "Curve fitting",
       https://en.wikipedia.org/wiki/Curve_fitting
.. [2] Wikipedia, "Polynomial interpolation",
       https://en.wikipedia.org/wiki/Polynomial_interpolation

Examples
--------
>>> import warnings
>>> x = np.array([0.0, 1.0, 2.0, 3.0,  4.0,  5.0])
>>> y = np.array([0.0, 0.8, 0.9, 0.1, -0.8, -1.0])
>>> z = np.polyfit(x, y, 3)
>>> z
array([ 0.08703704, -0.81349206,  1.69312169, -0.03968254]) # may vary

It is convenient to use `poly1d` objects for dealing with polynomials:

>>> p = np.poly1d(z)
>>> p(0.5)
0.6143849206349179 # may vary
>>> p(3.5)
-0.34732142857143039 # may vary
>>> p(10)
22.579365079365115 # may vary

High-order polynomials may oscillate wildly:

>>> with warnings.catch_warnings():
…     warnings.simplefilter('ignore', np.RankWarning)
…     p30 = np.poly1d(np.polyfit(x, y, 30))
…
>>> p30(4)
-0.80000000000000204 # may vary
>>> p30(5)
-0.99999999999999445 # may vary
>>> p30(4.5)
-0.10547061179440398 # may vary

Illustration:
```

```
>>> import matplotlib.pyplot as plt
>>> xp = np.linspace(-2, 6, 100)
>>> _ = plt.plot(x, y, '.', xp, p(xp), '-', xp, p30(xp), '--')
>>> plt.ylim(-2,2)
(-2, 2)
>>> plt.show()
```

La syntaxe est la suivante:

```
p = polyfit(xData,yData,deg = 1) # on précise le degré du polynôme.
```

Le résultat `p` est une liste Python de n valeurs qui représente les coefficients d'un polynôme de degré $n-1$ écrit sous la forme suivante:

$$P(X) = p[0]X^{n-1} + p[1]X^{n-2} + p[2]X^{n-3} + \ldots + p[n-2]X + p[n-1]$$

Exemple `p = [3,2,1]` représente le polynôme de degré deux suiant:

$$P(X) = 3X^2 + 2X + 1$$

Pour une régression affine, le degré du polynôme est `deg = 1`.

Les coefficients de la droite de régression $Y = AX + B$ sont donc données par:

- `A = p[0]`, est la pente de la droite, c'est le coefficient du terme de degré 1,
- `B = p[1]`, est le terme constant (l'ordonnée à l'origine), c'est le coefficient du terme de degré zéro.

5.2 N2 n°3 Exercice-type pour la régression linéaire : exemple de situation expérimentale (corrigé)

- On mesure l'absorbance de cinq solutions de complexe $[Fe(SCN)]^{2+}$ de concentrations connues. L'absorbance de chacune des solutions est mesurée à 580 nm.
- On dispose d'une solution (s) de la même espèce chimique dont on souhaite connaître la concentration C_s munie de son incertitude-type.

Données du problème - Les résultats sont consignés dans le tableau ci-dessous ($\lambda = 580$ nm)

C / mol L^{-1}	$2.5 \cdot 10^{-4}$	$5.0 \cdot 10^{-4}$	$1.0 \cdot 10^{-3}$	$1.5 \cdot 10^{-3}$	$2.0 \cdot 10^{-3}$
A	0.143	0.264	0.489	0.741	0.998

- L'absorbance de la solution (S) lue est $A_s = 0.571$
- Dans la notice du spectrophotomètre, le constructeur indique que la précision sur la mesure de A est ±2%. On l'interprète comme une variable aléatoire à distribution uniforme sur un intervalle de demi-étendue $\Delta A = \frac{2}{100}A$;
- pour les solutions, le technicien fournit une « précision » de la concentration C à 2 %. On l'interprète comme une variable aléatoire à distribution uniforme sur un intervalle de demi-étendue $\Delta C = \frac{2}{100}C$.

Questions

a) Déterminer l'équation de la droite d'étalonnage par une régression affine $C = f(A)$.

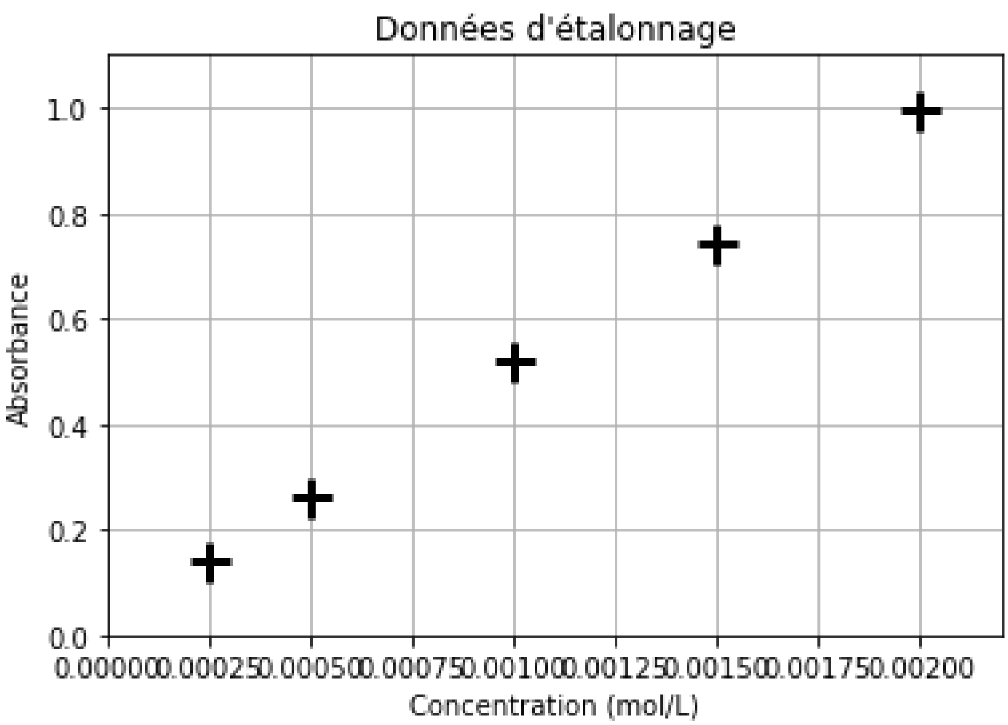
b) Représenter la droite de régression ainsi que le jeu des cinq donnnées.

c) En déduire la concentration A_s de la solution inconnue.

5.2.1 a) Détermination des coefficients de la droite de régression

Il suffit d'appeler la fonction poylfit sur le jeu de données: - $x = C$, concentration en abscisses, - $y = A$, absorbance en ordonnées.

[75]:
```
# Etape zéro : Entrée des données du problème : C et A sont des tableaux Numpy␣
↪de même taille
C = np.array([2.5e-4, 5.0e-4, 1.0e-3, 1.5e-3, 2.0e-3])
A = np.array([0.143, 0.264, 0.520, 0.741, 0.998])

# Etape 1: visualistion des données (à faire systématiquement même si non␣
↪demandé)
plt.plot(C,A,'+k',ms = 15,mew = 3) #on trace l'absorbance A en fonction de la␣
↪concentration C
plt.xlabel('Concentration (mol/L)') , plt.ylabel('Absorbance'), plt.grid()
plt.xlim([0,2.2e-3]), plt.ylim([0,1.1]) # ajustement des limites d'axes pour␣
↪voir le "zéro"
plt.title("Données d'étalonnage")
plt.show()
```



[57]:
```python
# Détermination des coefficients de la régression: utilisation de la fonction␣
↪polyfit
p = np.polyfit(C, A, deg = 1) # On effectue le régression Y = f(X) = p1.x + p0␣
↪soit A = p1.C + p0
print(p)
```

```
[4.85829268e+02 2.30792683e-02]
```

[58]:
```python
# Affichage du résultat (ATTENTION aux unités des coefficients!)
print('La pente de la droite de régression est p1 = ',format(p[0],"#.4g"), 'L/
↪mol')# la pente est en L/mol
print("L'ordonnée à l'origine est p0 = ",format(p[1],"#.4g"))
```

```
La pente de la droite de régression est p1 =  485.8 L/mol
L'ordonnée à l'origine est p0 =  0.02308
```

Remarque : la commande `format(x,"#.4g")` permet de mettre en forme le nombre flottant en supprimant les chiffres non significatifs.

5.2.2 b) Représentation de la droite de régression

Pour tracer une droite, il suffit de deux points.

Nous construisons donc un vecteur Numpy contenant les deux valeurs extrêmes des abscisses :

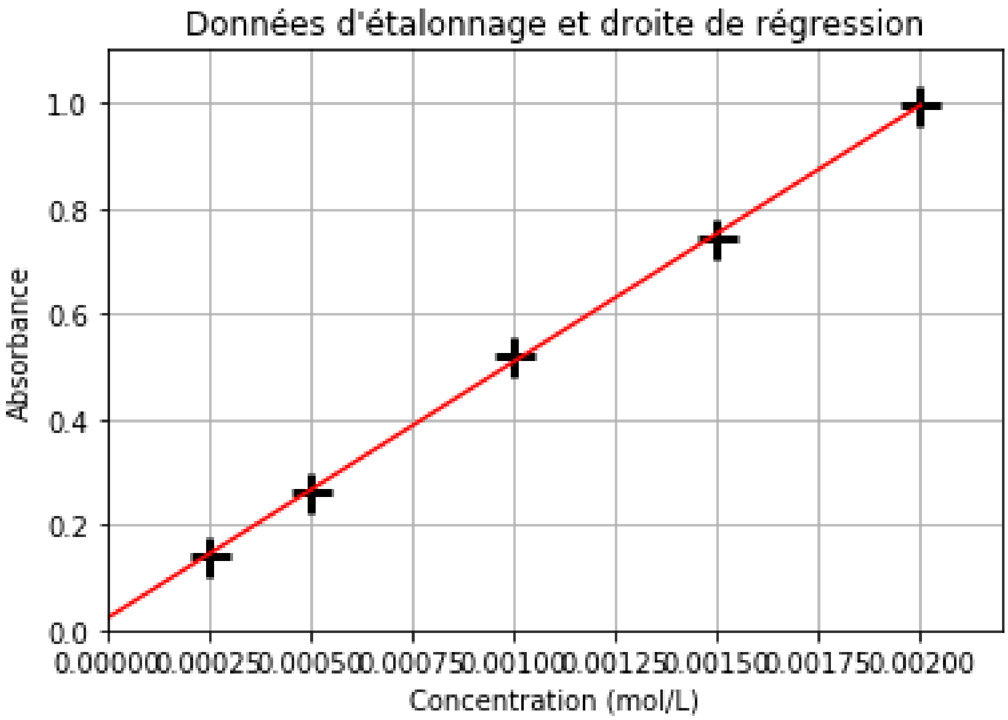
`xi = np.array([np.min(C),np.max(C)])`

Puis nous calculons les valeurs y_i par l'équation de la droite

`yi = p[0]*xi + p[1]`

[82]:
```python
# Superposition des données et de la droite de régression
# Affichage des données

plt.plot(C,A,'+k',ms = 15,mew = 3) #on trace l'absorbance A en fonction de la␣
↪concentration C
plt.xlabel('Concentration (mol/L)') , plt.ylabel('Absorbance'), plt.grid()
plt.xlim([0,2.2e-3]), plt.ylim([0,1.1]) # ajustement des limites d'axes pour␣
↪voir le "zéro"

# Détermination des valeurs minimales et maximales d'abscisses
xi = np.array([np.min(C), np.max(C)]) # les deux valeurs d'abscisses permettant␣
↪le tracé de la droite
# remarque: ici, on souhaite visualiser l'équation de droite à proximité du␣
↪point origine
xi = np.array([0, np.max(C)]) # les deux valeurs d'abscisses permettant le␣
↪tracé de la droite
```

```python
# Calcul des deux valeurs d'ordonnées pour les points extrêmes
yi = p[0]*xi + p[1]
plt.plot(xi,yi,'-r') # droite de régression en rouge

plt.title("Données d'étalonnage et droite de régression")
plt.show()
```



5.2.3 c) Détermination de la concentration inconnue.

Il suffit d'utiliser la relation affine pour laquelle les constantes A et B sont connnues:

$$Y = A.X + B$$

La valeur de Y étant connue, on en déduit la valeur de X.

Dans notre cas,

$$A = \mathrm{p}[0]\ C + \mathrm{p}[1]$$

On en tire donc

$$C = \frac{A - \mathrm{p}[1]}{\mathrm{p}[0]}$$

[88]:
```python
As = 0.571 # donnée mesurée
Cs = (As-p[1])/p[0]
print('La concentration de la solution inconnue est estimée à Cs = ',format(Cs,"#.4g"), ' mol/L' )
```

```
La concentration de la solution inconnue est estimée à Cs =  0.001128  mol/L
```

Conclusion : la concentration de la solution est estimée à $C_s \approx 1,128 \times 10^{-3}$ mol.L^{-1}.

En revanche, nous n'avons aucune information quant à la précision de ce résultat.

L'estimation des incertitudes est au programme en MPSI et sera vue ultérieurement.

6 Exercices d'entraînement

6.1 O2 n°4 Simulation de la somme de deux variables aléatoires gaussiennes

a) Ecrire le code python qui génère deux variables aléatoires gaussiennes X_1 et X_2 de paramètres respectifs $\mu_1 = 5$, $\sigma_1 = 0.5$ et $\mu_2 = 10$, $\sigma_2 = 1$. On utilisera la fonction `normal()` du module numy.random. et on choisira $N = 10^6$ échantillons de chacune de ces variables.

b) Représenter graphiquement les histogrammes des échantillons X_1, X_2 et $X_s = X_1 + X_2$ sur un même graphe.

c) Ecrire le code Python qui estimer la moyenne et l'écart-type de la variable X_s.

d) Comparer l'estimation de l'écart-type σ_s à la valeur théorique donnée par l'expression suivante:

$$\sigma_s = \sqrt{\sigma_1^2 + \sigma_2^2}$$

On s'aidera de la trâme proposée (à compléter) pour résoudre l'exercice.

[ ]:
```python
## a) génération des deux variables aléatoire
# Etape 1: import du module numpy et du module random de numpy
# à compléter

# Etape 2: utilisation de la fonction normal(moy, ecarttype, nbSamples)
N = 10**6 # nb d'échantillons
X1 = # à compléter# génère 10^6 nb aléatoires selon la loi normale de moyenne 5 et d'écart-type 1
X2 = # à compléter# génère 10^6 nb aléatoires selon la loi normale de moyenne 10 et d'écart-type 2
```

[ ]:
```python
## b) Représentation des histogrammes des échantillons X1 et X2
# Etape 1: import du module pyplot
# à compléter

# Etape 2: construction des valeurs de la variable somme Xs

Xs = # à compléter
```

```python
# Etape 2: tracé des histogrammes avec la fonction hist
# à compléter # histogramme de X1
# à compléter# histogramme de X2
# à compléter # histogramme de Xs

# Etape 3: ajout des titres des axes, titre de figure
plt.xlabel('Valeurs tirées')
plt.ylabel('Fréquences')
plt.title('histogrammes des variables X1, X2 et X1+X2')
plt.legend()
plt.show()
```

[ ]:
```python
# c) Estimation de la moyenne et de l'écart-type de la variable Xs
moyS  # à compléter #calcule la moyenne
sigmaS # à compléter # calcule l'écart type
print("Estimateurs : moyenne = ",moyS," écart-type = ",sigmaS)
```

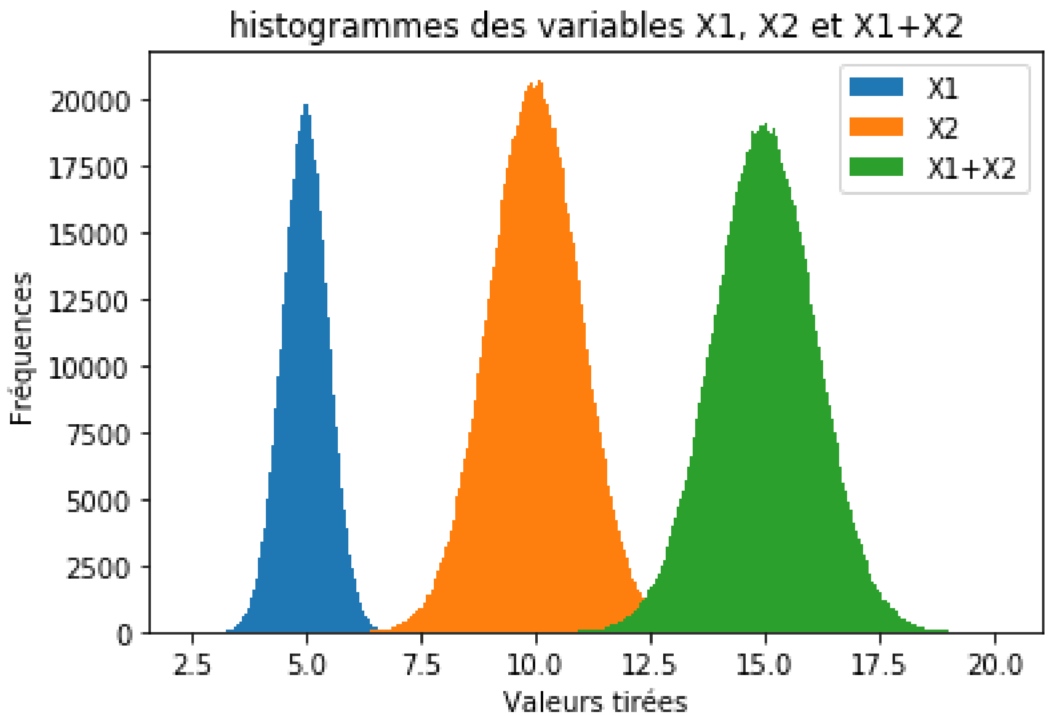
[ ]:
```python
# d) Comparaison avec la valeur théorique
s1, s2 = 0.5, 1. # variables auxilaires
sigmaStheo = # à compléter
print("valeur théorique de l'écart-type ",sigmaStheo)
```

Correction de l'exercice O2 n4

[96]:
```python
# a) génération des deux variables aléatoire
# Etape 1: import du module numpy et du module random de numpy
import numpy as np
import numpy.random as rd

# Etape 2: utilisation de la fonction normal(moy, ecarttype, nbSamples)
N = 10**6 # nb d'échantillons
X1 = rd.normal(5., .5 , N) # génère 10^6 nb aléatoires selon la loi normale de␣
↪moyenne 5 et d'écart-type 1
X2 = rd.normal(10., 1., N) # génère 10^6 nb aléatoires selon la loi normale de␣
↪moyenne 10 et d'écart-type 2
```

[103]:
```python
# b) Représentation des histogrammes des échantillons X1 et X2
# Etape 1: import du module pyplot
import matplotlib.pyplot as plt

# Etape 2: construction des valeurs de la variable somme Xs

Xs = X1+X2

# Etape 2: tracé des histogrammes avec la fonction hist
plt.hist(X1, bins = 'rice',label='X1') # histogramme de X1
```

```
plt.hist(X2, bins = 'rice',label='X2') # histogramme de X2
plt.hist(Xs, bins = 'rice',label='X1+X2') # histogramme de Xs

# Etape 3: ajout des titres des axes, titre de figure
plt.xlabel('Valeurs tirées')
plt.ylabel('Fréquences')
plt.title('histogrammes des variables X1, X2 et X1+X2')
plt.legend()
plt.show()
```



[109]:
```
# c) Estimation de la moyenne et de l'écart-type de la variable Xs
moyS   = np.mean(Xs)           # la fonction mean() calcule la moyenne
sigmaS = np.std(Xs,ddof  = 1) # la fonction std() calcule l'écart type
print("Estimateurs : moyenne = ",moyS," écart-type = ",sigmaS)
```

```
Estimateurs : moyenne =  14.99930926202917  écart-type =  1.116657911553038
```

[111]:
```
# d) Comparaison avec la valeur théorique
s1, s2 = 0.5, 1.
sigmaStheo = (s1**2+s2**2)**(0.5) # racine carré de la somme des carrés␣
 ↪(expression de type "Pythagore")
print("valeur théorique de l'écart-type ",sigmaStheo)
```

```
valeur théorique de l'écart-type  1.118033988749895
```

Conclusion : on constate que la valeur estimée est “proche” de la valeur théorique.

Remarque: on pourrait davantage préciser la notion de “proche” en utilisant la notion de “Z-score” (cf chapitre sur les incertitudes), mais ce n’est pas l’objet de cet exercice.

6.2 O2 n°5 Simulation de la différence deux variables aléatoires uniforme

On considère deux variables aléatoires de loi uniforme X_1 et X_2 telles que :

- X_1 est uniformément répartie sur l’intervalle de valeurs $I_1 = [x_1 - \Delta_1; x_1 + \Delta_1]$;
- X_2 est uniformément répartie sur l’intervalle de valeurs $I_2 = [x_2 - \Delta_2; x_2 + \Delta_2]$.

Dans toute la suite, on prendra : $x_1 = 5$, $\Delta_1 = 0,5$, $x_2 = 20$ et $\Delta_2 = 1$.

a) Ecrire le code python qui génère $N = 10^6$ réalisation de ces variables aléatoires. On pourra utiliser la fonction `uniform(a,b,nbSamples)` du module numy.random.

b) Représenter graphiquement les histogrammes des échantillons X_1, X_2 et $X_d = X_2 - X_1$ sur un même graphe.

c) Ecrire le code Python qui estimer la moyenne et l’écart-type σ_d de la variable X_d.

d) Comparer l’estimation de l’écart-type σ_d à la valeur donnée par l’expression suivante :

$$\sigma_d = \sqrt{\sigma_1^2 + \sigma_2^2}$$

dans laquelle les écarts-types σ_1 et σ_2 sont donnés par la **demi-largeur de l’intervalle de valeurs divisé par racine carrée de trois**:

$$\sigma_i = \frac{\Delta_i}{\sqrt{3}}$$

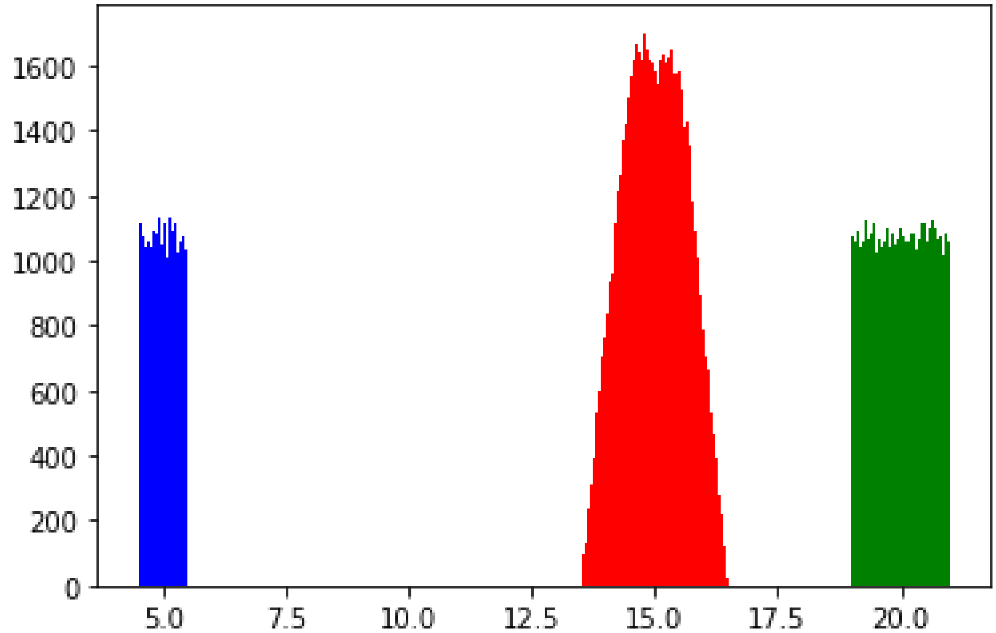
Ci-dessous une trame à compléter.

[ ]:
```
# Données numériques
N = 10**5
x1, delta1, x2, delta2 = 5, 0.5, 20, 1.

# Génération des variables X1, X2 et Xd
X1 = # à compléter
X2 = # à compléter
Xd = # à compléter

# Tracé des histogrammes
plt.hist(X1,bins='rice',color = 'b', label='X1')
plt.hist(X2,bins='rice',color = 'g', label='X2')
plt.hist(Xd,bins='rice',color = 'r', label='Xd')
plt.show()

# Estimation de la moyenne et de l'écart-type de Xd
moyd =# à compléter
sigmad = # à compléter
```

```python
print("Estimateurs : moyenne = ",moyd," écart-type = ",sigmad)

# Comparaison avec la théorie
sigma1, sigma2 = delta1/3**0.5, delta2/3**0.5
sigmad_bis = # à compléter
print("Expression de l'écart-type = ",sigmad_bis)
```

Correction 02 n°5

[124]:
```python
# Données numériques
N = 10**5
x1, delta1, x2, delta2 = 5, 0.5, 20, 1.

# Génération des variables X1, X2 et Xd
X1 = rd.uniform(x1-delta1, x1+delta1,N)
X2 = rd.uniform(x2-delta2, x2+delta2,N)
Xd = X2-X1

# Tracé des histogrammes
plt.hist(X1,bins='rice',color = 'b', label='X1')
plt.hist(X2,bins='rice',color = 'g', label='X2')
plt.hist(Xd,bins='rice',color = 'r', label='Xd')
plt.show()

# Estimation de la moyenne et de l'écart-type de Xd
moyd =np.mean(Xd)
sigmad = np.std(Xd,ddof = 1)
print("Estimateurs : moyenne = ",moyd," écart-type = ",sigmad)

# Comparaison avec la théorie
sigma1, sigma2 = delta1/3**0.5, delta2/3**0.5
sigmad_bis = np.sqrt(sigma1**2 + sigma2**2)
print("Expression de l'écart-type = ",sigmad_bis)
```




```
Estimateurs : moyenne =  15.000213289442291  écart-type =  0.6436471135060081
Expression de l'écart-type =  0.6454972243679029
```

Conclusion: la simulation donne une valeur en accord avec l'expression $\sigma_d = \sqrt{\sigma_1^2 + \sigma_2^2}$.

6.3 O2 n°6 Simulation de la somme deux variables aléatoires uniformes

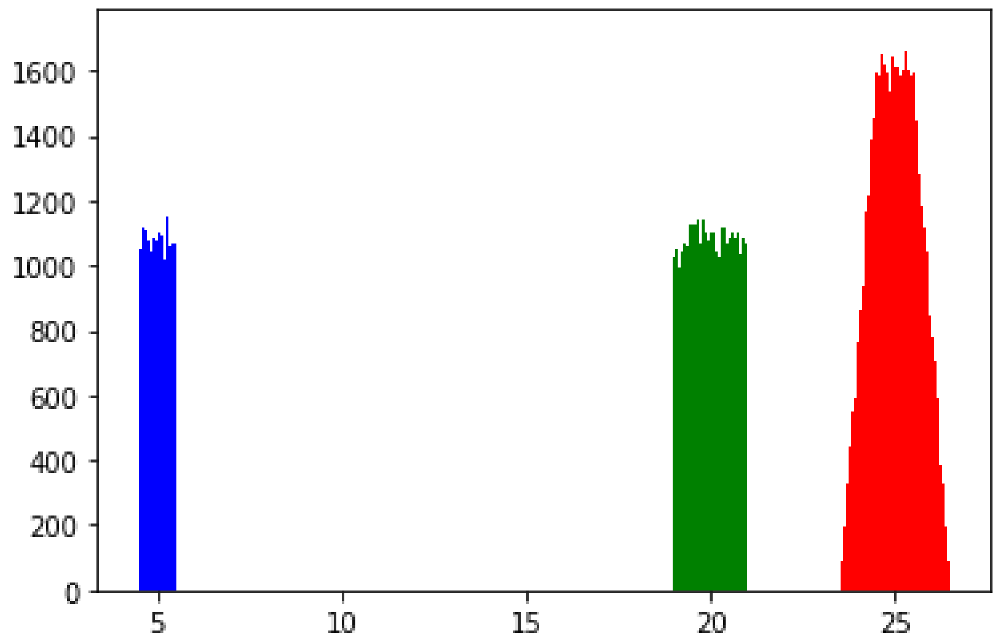
Reprendre l'exercice O2 n°5 mais en calculant cette fois la grandeur somme $X_1 + X_2$. Les mêmes constatations s'appliquent-elles?

Correction il suffit de recopier les mêmes instructions en remplaçant le signe "moins" par un signe plus.

On constate que la relation donnant l'écart-type demeure toujours valide.

[133]:
```
# Données numériques
N = 10**5
x1, delta1, x2, delta2 = 5, 0.5, 20, 1.

# Génération des variables X1, X2 et Xd
X1 = rd.uniform(x1-delta1, x1+delta1,N)
X2 = rd.uniform(x2-delta2, x2+delta2,N)
Xd = X2+X1 # Xd est CETTE fois une SOMME : on garde la notation Xd pour éviter␣
 ↪de tout taper à nouveau!

# Tracé des histogrammes
plt.hist(X1,bins='rice',color = 'b', label='X1')
plt.hist(X2,bins='rice',color = 'g', label='X2')
```

```python
plt.hist(Xd,bins='rice',color = 'r', label='Xd')
plt.show()

# Estimation de la moyenne et de l'écart-type de Xd
moyd =np.mean(Xd)
sigmad = np.std(Xd,ddof = 1)
print("Estimateurs : moyenne = ",moyd," écart-type = ",sigmad)

# Comparaison avec la théorie
sigma1, sigma2 = delta1/3**0.5, delta2/3**0.5
sigmad_bis = np.sqrt(sigma1**2 + sigma2**2)
print("Expression de l'écart-type = ",sigmad_bis)
```




```
Estimateurs : moyenne =  25.001932503761235  écart-type =  0.6431162008513457
Expression de l'écart-type =  0.6454972243679029
```

6.4 O2 n°7 Simulation du quotient de deux variables aléatoires gaussiennes (à chercher)

On considère deux variables aléatoires de loi normale U et I telles que :

- U possède une moyenne $\overline{U} = 1,2457$ V et d'écart-type $\sigma_U = 1,2\,\mathrm{mV}$;
- I possède une moyenne $\overline{I} = 12,383$ mA et d'écart-type $\sigma_I = 5,2\,\mu\mathrm{A}$.

a) Simuler à l'aide du module Numpy un grand nombre de réalisations de ces variables aléatoires.

On considère la grandeur

$$R = U/I$$

.

b) Tracer l'histogramme de la grandeur R et déterminer les paramètres statistiques de cette distribution (moyenne $\overline{R}$ et écart-type σ_R).

c) Comparer le résultat obtenu aux expressions théoriques:

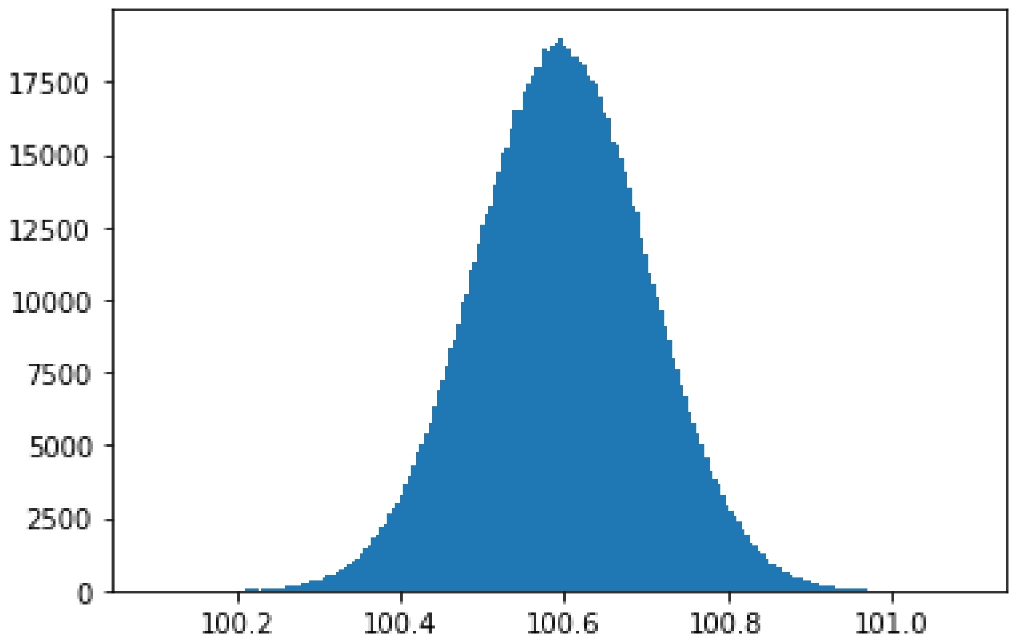
$$\overline{R} = \frac{\overline{U}}{\overline{I}}$$

et

$$\frac{\sigma_R}{\overline{R}} = \sqrt{\left(\frac{\sigma_U}{\overline{U}}\right)^2 + \left(\frac{\sigma_I}{\overline{I}}\right)^2}$$

Correction de O2 n°7

[162]:
```python
# import des modules
import numpy as np
import matplotlib.pyplot as plt

# tirages aléatoires
N = 10**6 # "grand nombre"
U = np.random.normal(1.2457,    1.2e-3, N) # attention à la conversion mV en V
I = np.random.normal(12.383e-3, 5.2e-6, N) # l'intensité électrique est␣
 ↪convertie en ampère (A)

# Grandeur R
R = U/I

# Histogramme
plt.hist(R,bins='rice')
plt.show()

# Paramètres statistiques
moyR, sigmaR = np.mean(R), np.std(R, ddof=1) # fonction mean (moyenne) et std␣
 ↪(écart-type)
print("Estimateurs : \t moyenne = ",moyR, " écart-type = ",sigmaR)

# Valeurs théoriques
# Pour les applications numériques on utilise des variables auxiliaires pour␣
 ↪"alléger" les notations
u, su, i, si = 1.2457, 1.2e-3, 12.383e-3, 5.2e-6
moyTheo = u/i
sigmaTheo = (u/i)*np.sqrt( (su/u)**2+(si/i)**2)
print("Valeurs théoriques :\t moyenne = ", moyTheo, " écart-type = ", sigmaTheo)
```




```
Estimateurs :    moyenne =  100.59759394126196  écart-type =
0.10565441655114423
Valeurs théoriques :     moyenne =  100.5975934749253  écart-type =
0.10571438930348316
```

Conclusion: on constate un bon accord entre les données simulées et les valeurs théoriques.

6.5 O2 n°8 Régression linéaire (1) : période d'un pendule en fonction de la longueur de fil

Un étudiant souhaite modéliser la relation entre la longueur d'un pendule pesant et la période des oscillations de ce pendule (cf figure ci-desssous).

Pour différentes valeurs de la longueur ℓ de ce pendule, il mesure la période T des oscillations.

longueur ℓ (cm)	$10,0$	$15,0$	$31,25$	$47,5$	$63,8$	$80,0$
période d'oscillation T (s)	$0,632$	$0,782$	$1,119$	$1,377$	$1,603$	$1,791$

Dans tout l'exercice, on prendra $g = 9,81$ m.s^{-2} pour l'accélération de la pesanteur.

Questions

a) Tracer l'évolution de la période T en fonction de la longueur ℓ du pendule. Les points expérimentaux semblent-ils alignés?

La relation théorique entre la période et l'allongement se déduit des relations suivantes:

$$\omega = \sqrt{\frac{g}{\ell}}$$

et

$$T = \frac{2\pi}{\omega}$$

b) Justifier qu'il est pertinent de proposer le modèle de régression affine $Y = aX + b$ avec $Y = T^\alpha$ et $X = \ell$ sous la forme

$$T^\alpha = a\ell + b$$

où l'exposant α est un coefficient numérique que l'on déterminera.

c) Déterminer par régression affine les valeurs numériques des paramètres a et b de ce modèle (on ne demande pas d'estimer les incertitudes ni de valider ce modèle).

d) Représenter la droite de régression en superposition sur le jeu de données.

e) Comparer la valeur obtenue pour la pente a de la régression à la valeur théorique $a_{\text{théo}}$ que l'on précisera.

Autre méthode pour estimer la pente de la régression

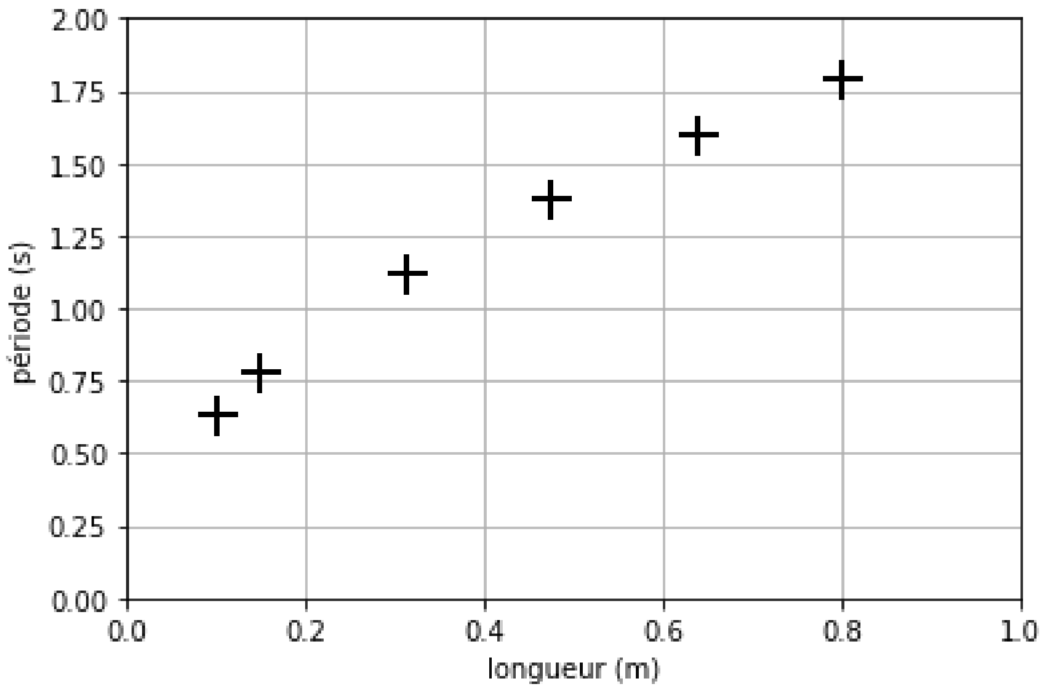
Une méthode alternative pour estimer la pente de la régression est de proposer un modèle de la forme $Y = aX$, c'est-à-dire purement "linéaire" (sans composante affine):

- pour chaque couple de valeurs (x_i, y_i), déterminer la valeur a_i de la pente par $a_i = y_i/x_i$,
- effectuer la moyenne de l'ensemble des pentes $\{a_i\}$ obtenues.

f) Mettre en oeuvre cette méthode et comparer l'estimation de a dans ce cas à celle obtenue avec la régression affine.

Correction O2 n°8

[264]:
```python
## a) Tracé de l'évolution de la période en fonction de la longueur
#import des modules
import numpy as np
import matplotlib.pyplot as plt

# Données expérimentales
l = np.array([0.10, 0.15, 0.3125,0.475, 0.638, 0.80])    # longueurs en mètres
T = np.array([0.632,0.782, 1.119, 1.377, 1.603, 1.791]) # périodes en secondes

# Evolution de la période en fonction de la longueur du pendule
plt.plot(l,T,'+k',ms = 15,mew = 2)
plt.xlim([0,1.0]),plt.ylim([0,2.]) # visualisation du zéro
plt.xlabel('longueur (m)'), plt.ylabel('période (s)')
plt.grid()
```



b) D'après les relations précédentes:

$$T^2 = \frac{4\pi^2}{g}\ell$$

qui est bien de la forme $Y = T^2 = aX$ avec $X = \ell$ et la pente théorique qui vaut

$$a_{\text{théo}} = \frac{4\pi^2}{g}$$

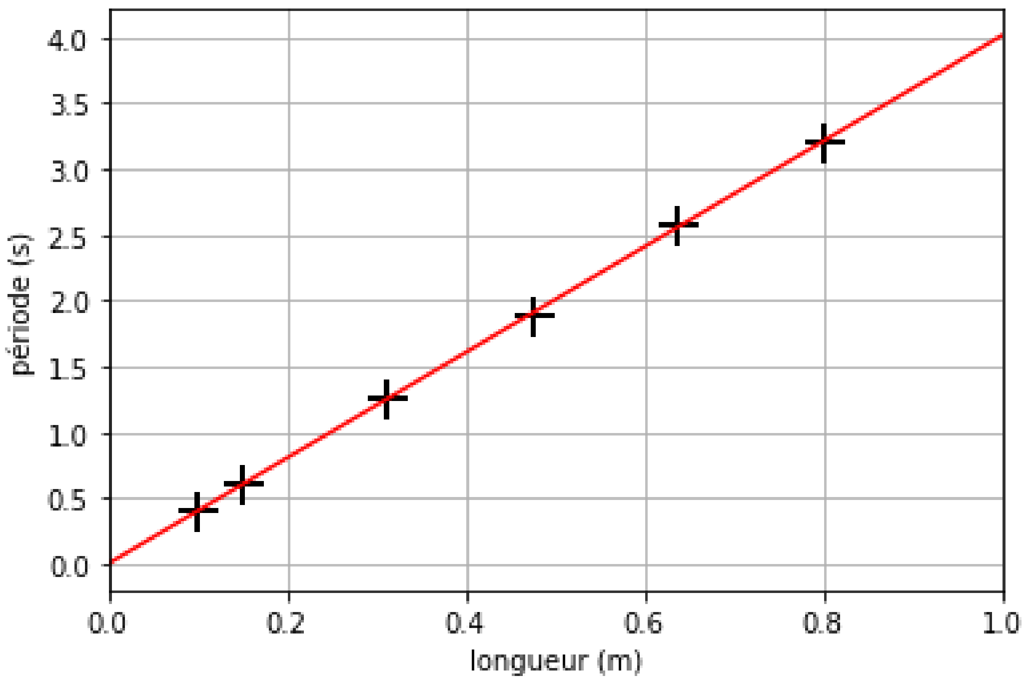
Le modèle affine est donc compatible avec la relation théorique à condition de choisir $\alpha = 2$.

Le résultat de la régression affine doit nous fournir la valeur $b \approx 0$

[265]:
```
## c) Détermination des valeurs de a et b par régression affine
X = l
Y = T**2
a, b = np.polyfit(X,Y,deg = 1) # on effectue la régression de Y = T^2 par la␣
↪variable X = longueur
print('Regression affine : pente a = ',a,' s²/m ',' b = ',b,' s²')

## d) Visualisation de la droite de régression sur le jeu de données
plt.plot(l,T**2.,'+k',ms = 15,mew = 2)
plt.xlim([0,1.0]) # visualisation du zéro
plt.xlabel('longueur (m)'), plt.ylabel('période (s)') # Attention aux unités␣
↪des paramètres a et b
xi = np.array([0,1.])
plt.plot(xi,a*xi+b,'-r') # affichage de la droite de régression
```

```
plt.grid()

## e) Comparaison avec la valeur théorique
g = 9.81 # m/s²
atheo = 4*np.pi**2/g
print('valeur théorique de la pente athéo = ', atheo)
```

```
Regression affine : pente a =  4.009545186914175  s²/m   b =
0.0018164816323252826  s²
valeur théorique de la pente athéo =  4.024303527457434
```



[267]:
```
## f) Autre estimation de la pente
ai = T**2/l
print("estimation de la pente a = ",np.mean(ai)) # moyenne des pentes estimées
```

```
estimation de la pente a =  4.0178389927436795
```

6.6 O2 n°9 Régression linéaire (2) : traction d'un élastique

Un étudiant souhaite **caractériser le comportement en traction** d'un élastique en caoutchouc.

Pour cela, il met en oeuvre le dispositif expérimental suivant (cf figure ci-dessous) :

- l'élastique à tester est accroché à une potence,
- on fixe à son extrémité inférieure un masse m de valeur variable,

- pour plusieurs valeurs de la masse m, il mesure la longueur ℓ de l'élastique à l'aide d'un mètre ruban.

Illustration du dispositif expérimental

Instruments de mesure utilisé : - Balance de cuisine (au gramme) ; - Mètre ruban (gradué en millimètre)

Tableau des mesures

masse m (g)	0	200	300	400	500	600	700
longueur ℓ (cm)	$12,4$	$12,9$	$13,1$	$13,3$	$13,6$	$14,2$	$14,6$

Questions

On note $\Delta\ell$ l'**allongement** de l'élastique la grandeur définie par

$$\Delta\ell = \ell - \ell_0$$

où ℓ_0 est la longueur de l'élastique à vide, c'est-à-dire lorsque qu'il est soumis à un allongement nul.

a) Représentation des mesures : tracer l'évolution de l'allongement $\Delta\ell$ (en mètres) en fonction de la masse (en kilogrammes).

b) Etablir l'équation de la droite de régression $\Delta\ell = f(m)$ et représenter la droite de régression sur le jeu de données.

On suppose que la loi de Hooke d'applique, c'est-à-dire que la relation *force vs allongement* est linéaire, soit:

$$F = k\Delta\ell$$

c) En prenant $g = 9,81\text{m.s}^{-2}$ déterminer la valeur de la raideur k correspond à la loi de Hooke.

Remarques:

- l'étude des incertitudes n'est pas demandé dans cet exercice,
- rien de permet d'affirmer que la loi de Hooke est valide dans ce cas de cet élastique (et d'ailleurs, il n'en est rien ! nous montrerons que les résultats de ces mesures permettent **d'invalider le modèle linéaire proposé**).

Indications : pour répondre aux questions, on s'aidera de la trame pré-remplie ci-dessous.

[ ]:
```
## Import des modules
  # import de numpy (à compléter)
  # import de pyplot (à compléter)

## Saisie des données expérimentales
m = np.array([0, 200, 300, 400, 500, 600, 700])          # masses appliquées
 ↪(en g)
u_m = 0.5/np.sqrt(3)                                     # incertitude-type
 ↪sur les valeurs de m (en g)
```

```
l0 = 12.4                                            # longueur du ressort␣
 ↪à vide (en cm)
l = np.array([12.4,12.9, 13.1, 13.3, 13.6, 14.2, 14.6])   # longueur du ressort␣
 ↪en charge (en cm)
u_l = 0.05/np.sqrt(3)

## a) Tracé des courbes
dl = # allongements en mètres (à compléter)
mkg = # masse en kg (à compléter)
plt.plot(...) # à compléter
plt.xlabel('masse appliquée (kg)')
plt.ylabel('allongement (m)')
plt.grid()

## b) Régression linéaire
a,b = # extraction des paramètres a et b de la régression linéaire (à compléter)
print('la pente est a = ',a, ' ') # affichage de la pente avec l'unité correcte␣
 ↪(à compléter)
print("l'ordonnée à l'origine est n = ",b,' ') # affichage de l'ordonnée à␣
 ↪l'origine (compléter l'unité)

# Ajout de la droite de régression en rouge sur le jeu de données (à compléter)

plt.show()
```

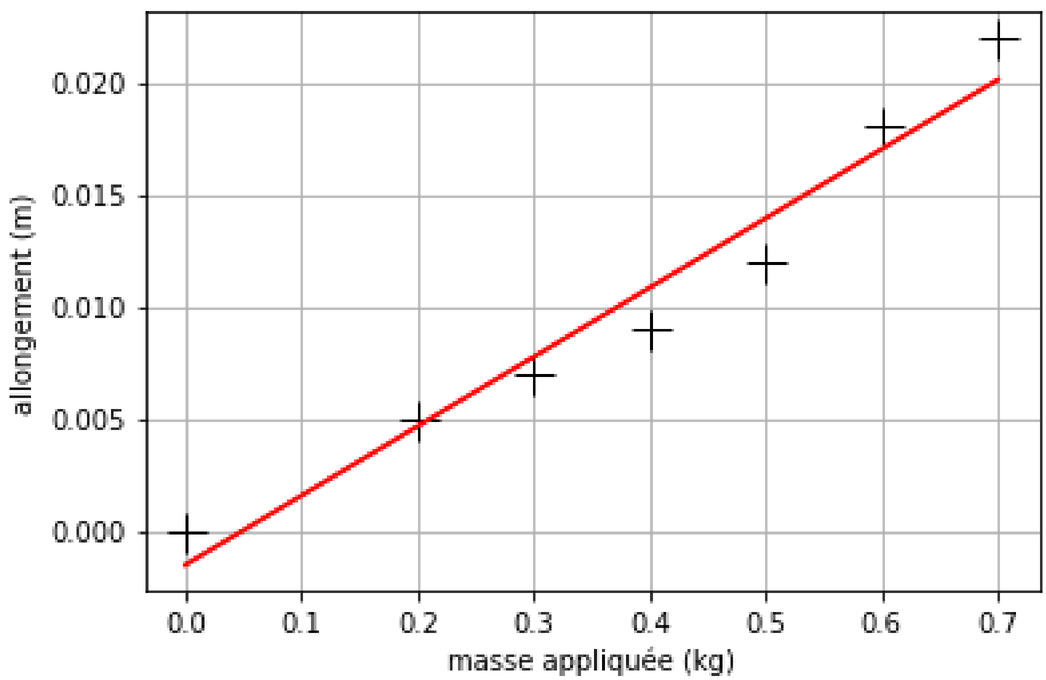
c) Détermination de la raideur k de la loi de Hooke $F = k\Delta\ell$ (avec l'unité correcte)

Correction de O2 n°9

[179]:
```
## Import des modules
import numpy as np               # import de numpy
import matplotlib.pyplot as plt # import de pyplot

## Saisie des données expérimentales
m = np.array([0, 200, 300, 400, 500, 600, 700])          # masses appliquées␣
 ↪(en g)
u_m = 0.5/np.sqrt(3)                                     # incertitude-type␣
 ↪sur les valeurs de m (en g)

l0 = 12.4                                            # longueur du ressort␣
 ↪à vide (en cm)
l = np.array([12.4,12.9, 13.1, 13.3, 13.6, 14.2, 14.6])   # longueur du ressort␣
 ↪en charge (en cm)
u_l = 0.05/np.sqrt(3)

## a) Tracé des courbes
dl = (l-l0)*1e-2 # allongements en mètres (à compléter)
```

```python
mkg = m*1e-3 # masse en kg (à compléter)
plt.plot(mkg,dl,'+k',ms = 15) # à compléter
plt.xlabel('masse appliquée (kg)')
plt.ylabel('allongement (m)')
plt.grid()


## b) Régression linéaire
a,b = np.polyfit(mkg, dl, deg = 1) # extraction des paramètres a et b de la␣
 ↪régression linéaire
print('la pente est a = ',a, ' m/kg') # affichage de la pente avec l'unité␣
 ↪correcte
print("l'ordonnée à l'origine est b = ",b,' m') # affichage de l'ordonnée à␣
 ↪l'origine avec l'unité correcte

# Ajout de la droite de régression en rouge sur le jeu de données
plt.plot(mkg, a*mkg + b, '-r') # on utilise les valeurs de masses X = mkg en␣
 ↪abscisses pour tracer Y = aX+b
# autre méthode pour tracer la droite de régression
xi = np.array([0,0.7]) # valeurs extrêmes
plt.plot(xi,a*xi+b,'-r') # on utilise deux points pour tracer la droite de␣
 ↪régression
plt.show()
```

```
la pente est a =  0.030819672131147537  m/kg
l'ordonnée à l'origine est  -0.0014590163934426186  m
```



c) Détermination de la raideur k (en newton par mètre : N.m^{-1}) de la loi de Hooke

$$F = k\Delta\ell$$

La force F est reliée à la masse par (poids d'un corps de masse m dans le champ de pesanteur g) :

$$F = mg$$

Or, en utilisant la valeur de la pente a obtenue par régression affine, on peut écrire,à condition de négliger l'ordonnée à l'origine b, que l'allongement et la masse sont des grandeurs proportionnelles :

$$Y = aX \quad \text{soit} \quad \Delta\ell = a \times m$$

On a donc $m = \frac{1}{a}\Delta\ell$, en remplaçant dans l'expression de la force $F = mg$, il vient:

$$F = \frac{g}{a}\Delta\ell$$

Donc, en identifiant avec la loi de Hooke:

$$k = \frac{g}{a}$$

Application numérique

[181]:
```
g = 9.81 # m/s2
k = g/a
print("La raideur k de l'élastique est estimée à k = ",k, ' N/m.')
```

```
La raideur k de l'élastique est estimée à k =  318.30319148936184  N/m.
```

Remarque : au vu des données expérimentales, le modèle affine peut sembler discutable mais, sans **prendre en compte rigoureusement les incertitudes de mesure**, aucune conclusion ne peut être tirée.

6.7 O2 n°10 Régression linéaire (3) : oscillations d'un système masse-ressort (à chercher)

Un étudiant s'intéresse aux oscillations verticales d'un système masse ressort (cf figure).

Pour cela, il effectue deux séries de mesures.

Dans la première série de mesures, il varie la masse m accrochée au ressort tout en mesurant l'allongement $\Delta\ell$ du ressort à l'équilibre.

masse m (g)	50	100	150	250
allongement à l'équilibre $\Delta\ell$ (cm)	16	31	49	82

Dans la seconde série de mesures, il détermine aussi précisément que possible la période T des oscillations pour différentes valeurs de la masse m.

masse m (g)	50	100	150	250
période d'oscillation T (s)	$0,812$	$1,15$	$1,41$	$1,82$

Questions

a) Etablir, à l'aide de la première série de mesures, la valeur de la raideur k du ressort que l'on supposera obéir à la loi de Hooke

$$F = k\Delta\ell$$

, la force appliquée sur le ressort étant donnée par le poids de la masse m (on prendra $g = 9,81$ m.s^{-2}).

b) A partir de la seconde série de mesures, proposer un modèle affine compatible avec la relation théorique entre la masse m et la fréquence des f des oscillations

$$f = \frac{1}{T}$$

avec

$$\omega = 2\pi f = \sqrt{\frac{k}{m}}$$

On déterminera les paramètres (a, b) de la régression affine et on comparera ces valeurs aux valeurs théoriques.

Aucune étude d'incertitude ni de validation de modèle n'est demandé dans cette question.

[333]:
```
## Données mesurées
m = np.array([50,100,150,250])*1e-3    # masses en kg
deltaL = np.array([16,31,49,82])*1e-2  # allongement en m
T = np.array([0.812, 1.15, 1.41, 1.82])# périodes en s
```

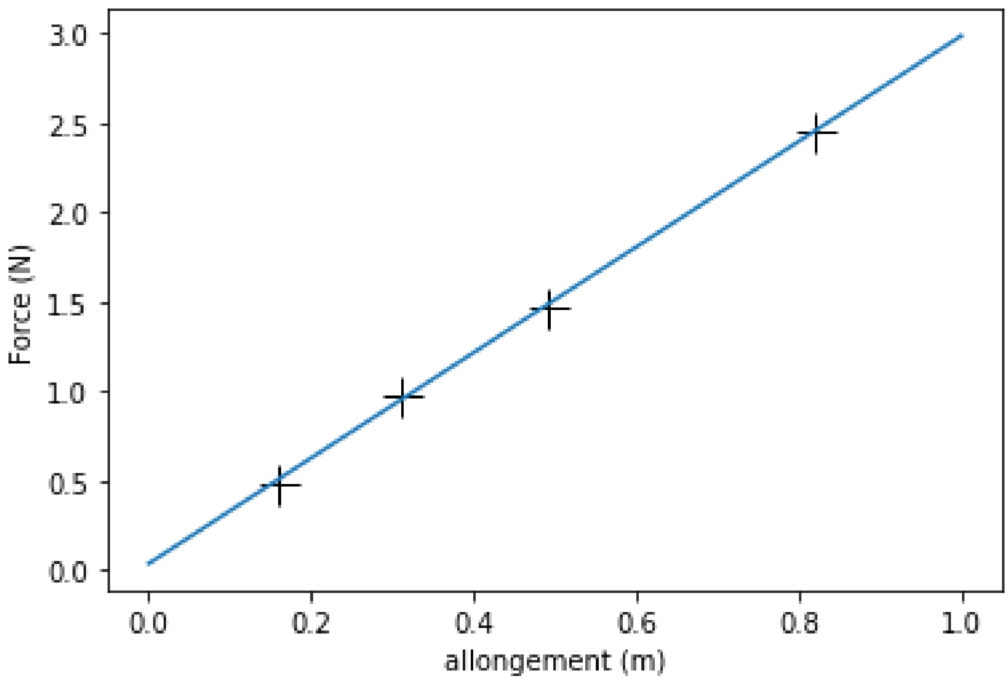
Correction O2 n°10

[284]:
```
## Données
m = np.array([50,100,150,250])*1e-3    # masses en kg
deltaL = np.array([16,31,49,82])*1e-2  # allongement en m
T = np.array([0.812, 1.15, 1.41, 1.82])# périodes en s

## a) détermination de la raideur du ressort F = k DeltaL
g = 9.81
F = m*g # Force (en N)
plt.figure(1)
plt.plot(deltaL,F,'+k',ms = 15) # force en fonction de l'allongement
p = np.polyfit(deltaL,F,deg=1) # régression linéaire
print("Regression : pente = ", p[0]," N/m", " b = ",p[1]," N") # paramètres de␣
 ↪la régression avec unités
plt.xlabel('allongement (m)')
plt.ylabel('Force (N)')
xi = np.array([0.,1.]) # valeurs extrêmes
```

```
plt.plot(xi,np.polyval(p,xi))     # polyval : évalue le polynôme p aux points de␣
↪coordonnées
```

```
Regression : pente =  2.947862453531598  N/m  b =  0.037076208178438604  N
```

[284]:
```
[<matplotlib.lines.Line2D at 0x22589cf2e88>]
```



D'après les données du sujet, l'expression de la période T en fonction de la masse m est donnée par:

$$T = \frac{2\pi}{\omega} = 2\pi\sqrt{m}k = \left(\frac{2\pi}{\sqrt{k}}\right) \times m^{1/2}$$

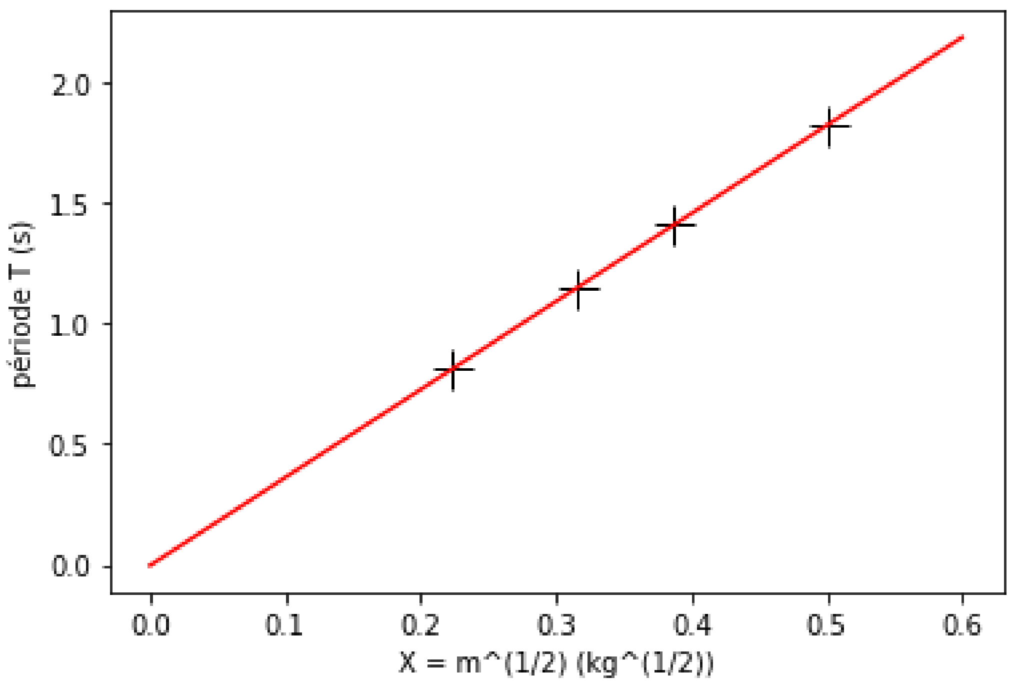
La régression linéaire se fait donc avec $Y = T$ et $X = m^{1/2}$, la pente théorique est donc

$$a_{\text{théo}} = \frac{2\pi}{\sqrt{k}}$$

[294]:
```
# b) Détermination des paramètres a,b de la régression affine

## Représentation des données Y = période, X = racine carrée de la masse
X = m**(1/2) # variable d'abscisse de la régression
plt.plot(X,T,'+k',ms = 15) # données expérimentales
plt.xlabel('X = m^(1/2) (kg^(1/2))') # variable X
plt.ylabel('période T (s)') # variable Y
```

```
a, b = np.polyfit(X,T,deg = 1) # régression linéaire
xi = np.array([0,0.6]) # valeurs extrêmes
plt.plot(xi,a*xi+b,'-r') # droite de régression
print('Regression affine : pente a = ',a," s.kg^(-1/2) ; b = ",b," s") #␣
↪attention aux unités

# Comparaison avec la valeur théorique
atheo = 2*np.pi/p[0]**(0.5) # valeur théorique, la raideur étant donnée par le␣
↪coefficient p[0]
print("valeur théorique de la pente = ",atheo, " s.kg^(-1/2)")
```

```
Regression affine : pente a =  3.647529156947755  s.kg^(-1/2) ; b =
-0.0033772144269951223  s
valeur théorique de la pente =  3.659537994984427  s.kg^(-1/2)
```



Autre possibilité, on peut tracer l'évolution de T^2 en fonction de la masse:

$$T^2 = \frac{(2\pi)^2}{k} \times m$$

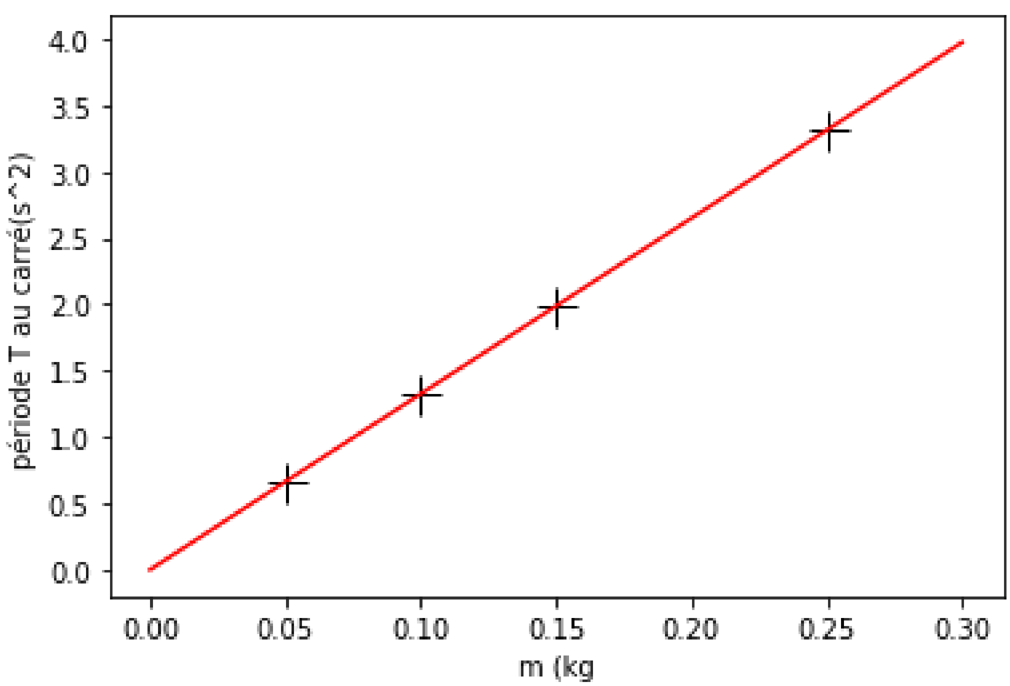
Dans ce cas, la pente théorique est

$$a'_{\text{théo}} = \frac{(2\pi)^2}{k}$$

[300]:

```
# b2) Détermination des paramètres a,b de la régression affine [METHODE␣
↪ALTERNATIVE]

## Représentation des données Y = période, X = racine carrée de la masse
Y = T**2                                    # variable d'ordonnée de la régression
plt.plot(m,Y,'+k',ms = 15)                  # données expérimentales
plt.xlabel('m (kg')                         # variable X
plt.ylabel('période T au carré(s^2)')  # variable Y


a, b = np.polyfit(m,Y,deg = 1) # régression linéaire
xi = np.array([0,0.3]) # valeurs extrêmes
plt.plot(xi,a*xi+b,'-r') # droite de régression
print('Regression affine : pente a = ',a," s^2.kg^(-1) ; b = ",b," s^2")#␣
↪attention aux unités

# Comparaison avec la valeur théorique
atheo = (2*np.pi)**2/p[0] # valeur théorique, la raideur étant donnée par le␣
↪coefficient p[0]
print("valeur théorique de la pente = ",atheo, " s^2.kg^(-1)")
```

```
Regression affine : pente a =  13.266738285714284  s^2.kg^(-1) ; b =
-0.0035905142857144441  s^2
valeur théorique de la pente =  13.39221833673464  s^2.kg^(-1)
```



Encore une autre méthode alternative : au lieu d'une régression affine, on peut faire la moyenne des pentes obtenues pour chacun des couples (y_i, x_i).

[301]:
```
ai = T**2/m # liste des rapports Y_i/X_i
print("estimation de la pente a = ", np.mean(ai)) # on prend la moyenne des␣
 ↪pentes
```

```
estimation de la pente a =  13.228869999999999
```

7 Complément : description mathématique de la régression linéaire ordinaire

Faire une régression linéaire revient à chercher le minimum d'une fonction.

Soit une série de N couples (x_i, y_i) de valeurs numériques, la régression linéaire ordinaire consiste à trouver les deux valeurs des paramètres (a, b) de la droite d'équation $y = ax + b$ qui **minimise les écarts verticaux** entre la droite modèle et les points de données (cf figure).

L'**écart vertical** ε_i entre le point numéro i et la droite est la quantité (représentée en tirés verts sur le figure) telle que:

$$\varepsilon_i = y_i - (ax_i + b)$$

On décide alors de **minimiser la somme des carrés de ces écarts**, c'est-à-dire la quantité notée $\varepsilon_N(a, b)$ qui dépend des N points de mesure et des deux variables a et b :

$$\varepsilon_N(a, b) = \sum_{i=1}^{N} \varepsilon_i^2 = \sum_{i=1}^{N} (y_i - (ax_i + b))^2$$

Comme la fonction $(a, b) \mapsto \varepsilon_N(a, b)$ est une fonction quadratique de deux variables, sa minimisation est aisée et se ramène à au système linéaire de deux équations suivant:

$$\frac{\partial \varepsilon_N}{\partial a} = 0 \quad \Leftrightarrow a \sum_i xi^2 + b \sum_i x_i = \sum_i x_i y_i$$

$$\frac{\partial \varepsilon_N}{\partial b} = 0 \quad \Leftrightarrow a \sum_i xi + b \sum_i 1 = \sum_i y_i$$

Qui se résoud en

$$a = \frac{\sum_i (x_i - \overline{x})(y_i - \overline{y})}{\sum_i (x_i - \overline{x})^2}$$

et

$$b = \overline{b} - a\overline{x}$$

[ ]:
```
```