MINISTÈRE DE L'ÉDUCATION NATIONALE

éduscol

Ressources pour le lycée

Mathématiques

Ressources pour le lycée général et technologique

Algorithmique et programmation

Ces documents peuvent être utilisés et modifiés librement dans le cadre des activités d'enseignement scolaire, hors exploitation commerciale.

Toute reproduction totale ou partielle à d'autres fins est soumise à une autorisation préalable du Directeur général de l'enseignement scolaire.

La violation de ces dispositions est passible des sanctions édictées à l'article L.335-2 du Code la propriété intellectuelle.

Ministère de l'Education Nationale
http://eduscol.education.fr

Juin 2017

Présentation générale

La circulaire n° 2017-082 du 2 mai 2017 apporte des aménagements au programme de mathématiques de seconde générale et technologique. Elle prévoit une partie dédiée à l'algorithmique et la programmation dont la rédaction a été remaniée et rappelle que le travail correspondant doit être réinvesti dans les trois autres parties (fonctions, géométrie, statistiques et probabilités). À la différence du programme de mathématiques du cycle 4 du collège, il s'agit donc d'adosser explicitement les activités de la partie *algorithmique et programmation* aux mathématiques.

Cet enseignement a un double objectif : faire travailler des notions mathématiques du programme dans un contexte différent, et poursuivre chez les élèves le développement des compétences suivantes, déjà travaillées au cycle 4[1] :

- décomposer un problème ;
- reconnaître des schémas ;
- généraliser et abstraire ;
- concevoir des algorithmes et les traduire dans un langage de programmation.

Les modalités de l'apprentissage correspondant peuvent être variées : travail individuel ou en groupe, en salle informatique ou en salle banale, au tableau ou sur papier, sur tablette ou sur ordinateur.

Il s'agit de consolider les acquis du cycle 4 autour de deux idées essentielles : la notion universelle de fonction d'une part et la programmation comme production d'un texte dans un langage informatique d'autre part.

- Les notions mathématique et informatique de fonction relèvent du même concept universel. En informatique, une fonction prend un ou plusieurs arguments et renvoie une valeur issue d'un calcul.
- Le choix d'un langage textuel, comme Python, au lieu d'un langage par blocs, comme Scratch, permet aux élèves de se confronter à la précision et la rigidité d'une syntaxe proche de celle des expressions mathématiques, avec l'avantage de pouvoir bénéficier du contrôle apporté par l'analyseur syntaxique.

Présentation de la ressource

Ce document présente des activités permettant d'éclairer des résultats et des méthodes mathématiques au travers d'algorithmes simples. **La plupart de ces activités n'ont pas été conçues pour être données telles quelles à des élèves. Leur objectif est de montrer aux enseignants comment on peut utiliser le langage Python pour faire des mathématiques autrement et leur proposer des situations pour se former et dans lesquelles ils pourront puiser des idées pour concevoir des activités de longueurs et de difficultés variées**, allant des questions flash aux mini-projets. **Même si les aménagements de programme ne concernent que la classe de seconde, les activités couvrent un spectre plus large et les professeurs sont libres de les adapter aux classes du cycle terminal.**

Les programmes de ce document sont écrits dans le langage Python, choisi pour la concision et la simplicité de sa syntaxe, la taille de la communauté d'utilisateurs (en particulier dans le cadre éducatif), ainsi que la richesse des ressources disponibles. On trouvera en annexe des indications sur l'installation de Python sur tablette ou sur ordinateur.

Le professeur gardera à l'esprit que l'enseignement de la partie *algorithmique et programmation* n'a pas pour objectif de former des experts dans tel ou tel langage de programmation ou dans la connaissance détaillée de telle ou telle bibliothèque de programme. Il s'agit de prolonger l'enseignement de la pensée algorithmique initié au cycle 4, qui trouve une place naturelle dans tous les champs du programme de mathématiques. L'écriture, la compréhension et la modification

[1] 1 cf. la ressource d'accompagnement du programme de mathématiques du cycle 4 Algorithmique et programmation.

d'algorithmes et de petits programmes permettent aux élèves d'acquérir de bonnes habitudes de rigueur, tout en revisitant les notions de variables et de fonctions.

Quelques concepts importants

Algorithmes

Un algorithme est une procédure de résolution de problème, abstraction faite des caractéristiques spécifiques qu'il peut revêtir. Par exemple, un algorithme de tri ne résout pas le problème du tri d'un jeu particulier de données mais a pour objectif de trier n'importe quel jeu de données : le problème du tri s'applique à différentes instances, c'est-à-dire à différents jeux de données.

Un algorithme s'applique donc à une famille d'instances d'un problème et produit, en un nombre fini d'étapes constructives, effectives, non-ambigües et organisées, la réponse au problème pour toute instance de cette famille.

De la même façon qu'un script Scratch se construit en accolant des briques élémentaires, un algorithme s'appuie sur un ensemble très réduit de constructions : l'affectation d'une variable, la séquence d'instructions, l'instruction conditionnelle, les boucles (bornées ou non bornées), les fonctions.

Ci-après sont décrites ces différentes constructions. Toutefois, il n'est pas recommandé de les présenter de façon magistrale aux élèves, qui les ont déjà rencontrées au cycle 4, mais de les mobiliser de façon naturelle dans les activités travaillées, en introduisant progressivement leur imbrication.

Affectation d'une variable

La ressource d'accompagnement du thème Algorithmique et programmation du programme de mathématiques au cycle 4 explique la différence entre les notions mathématique et informatique de la variable.

En mathématiques, la variable apparaît dans des formules comme celle du périmètre d'un cercle ou l'expression symbolique des fonctions. On distingue :

- les variables, comme dans les formules du type $P = 2\pi R$ ou l'expression symbolique $x \longmapsto f(x)$ d'une fonction ;
- les indéterminées, comme dans une identité remarquable $(a+b)^2 = a^2 + 2ab + b^2$, qui indique que l'égalité est vraie pour toutes les valeurs données à a et b ;
- les inconnues, comme dans l'équation $2x + 3 = 4x - 7$, où il s'agit cette fois de déterminer pour quelles valeurs de la variable x l'égalité est vraie ;
- les paramètres, qui conservent une valeur fixe, mais de portée générale, comme a qui désigne le coefficient de la fonction linéaire $x \longmapsto ax$.

En informatique, on est amené à écrire des instructions comme `x = x+1` qui sont d'une nature totalement différente : il ne s'agit pas là d'une égalité, ni d'une équation, mais d'une instruction d'affectation, qui va modifier le contenu de la variable `x`. Il est d'ailleurs recommandé de lire cette instruction « *x reçoit la valeur $x+1$* » et surtout pas « *x égale $x+1$* ».

C'est la sémantique particulière de l'affectation, qui n'a pas d'analogue en mathématiques, qui conduit les informaticiens à une notion différente de la variable.

Un modèle rigoureux de la variable informatique consiste à dire qu'une variable est une **étiquette** collée sur une **boîte** qui peut **contenir** différentes valeurs. Le contenu de chaque boîte varie au cours de l'exécution d'un programme (ce qui n'est pas le cas d'une variable mathématique). Quand l'ordinateur évalue une **expression** dans laquelle figure une variable, comme `x+4`, le résultat de l'évaluation est la somme du nombre contenu dans la boîte d'étiquette `x` et de 4.

Une instruction d'affectation comme `y = x+4` est donc exécutée de la façon suivante :

- évaluer l'expression `x+4` en ajoutant 4 à la valeur contenue dans la boîte étiquetée par `x` ;
- jeter le contenu de la boîte étiquetée par `y`, et le remplacer par la valeur calculée à l'étape précédente.

On vérifie que l'instruction `x = x+1` s'explique alors exactement de la même façon.

On observe également que le symbole `=` a été choisi pour l'opération d'affectation en Python. Le symbole `=` n'est donc pas disponible pour le test d'égalité qui s'écrit `==`. On sera attentif à cette différence : l'affectation est absente en mathématiques et omniprésente en informatique, le test d'égalité d'un usage moins fréquent en informatique.

Séquence d'instructions

En Python, une séquence d'instructions s'obtient en écrivant simplement à la suite, dans un ordre déterminé, différentes instructions, chacune sur une ligne, avec la même indentation (on reviendra sur l'utilisation et la place essentielle de l'indentation).

Par exemple :

```
a = 4
b = 5
c = maFonction(a+b,a-b)
```

affecte 4 à la variable `a`, puis 5 à la variable `b`. Enfin, on affecte à la variable `c` la valeur renvoyée par la fonction `maFonction` en prenant comme arguments la somme et la différence de `a` et `b`.

Il est à noter qu'on aurait pu condenser les deux premières affectations en une seule, en utilisant une affectation multiple : `a, b = 4, 5`.

Dans le cas d'une affectation multiple, on évalue d'abord les expressions à droite du symbole `=` puis on affecte les résultats aux variables qui figurent à gauche. On peut écrire simplement `a, b = b, a` pour échanger les valeurs des deux variables `a` et `b`.

Instruction conditionnelle

En Scratch, on a le choix entre une instruction conditionnelle avec ou sans clause « sinon ». Il en est de même avec Python. Voici un exemple sans clause « sinon » :

```
delta = b*b - 4*a*c
if delta > 0:
    x1 = (-b+sqrt(delta))/(2*a)
    x2 = (-b-sqrt(delta))/(2*a)
s = -b/a
p = c/a
```

Les deux points (symbole `:`) annoncent l'ouverture d'un bloc, contenant éventuellement plusieurs instructions. Le bloc est signalé par l'indentation : on ajoute en tête de chaque ligne du bloc le même nombre d'espaces (ou de tabulations). Le retour à l'indentation précédente signale ainsi tout naturellement la fin du bloc.

Cette utilisation de l'indentation est obligatoire.

Il y a ici deux affectations (à `x1` et à `x2`) dans le bloc, c'est-à-dire que ces deux affectations ne sont exécutées que si la condition `delta>0` est vérifiée. En revanche les affectations à `s` et `p` sont toujours effectuées.

Les environnements de programmation usuels aident à maintenir une indentation cohérente.

On dispose également d'une instruction conditionnelle avec une clause « sinon » (`else`), et même d'une abréviation pour « sinon si » (`elif`).

Voici un exemple, permettant de définir une fonction définie par morceaux.

```
def f(x):
    if x < 0:
        y = 2*x+3
    elif     x < 2: # ici 0 <= x < 2
        y = 3-x
    else:          # ici 2 <= x
        y = x*x-3
    return y
```

On a ajouté des commentaires pour expliciter les trois cas différents : ces commentaires sont annoncés par le symbole `#`. Les environnements usuels de programmation colorent automatiquement les commentaires pour les distinguer du reste du code.

Boucles bornées

Python propose une instruction `for` *variable* `in` *liste* : qui permet d'exécuter un bloc d'instructions (dont l'ouverture est signalée par le symbole :) en donnant successivement à la *variable* les différentes valeurs de la *liste*.

Pour retrouver le comportement du bloc `répéter 6 fois` de Scratch, il suffira d'utiliser l'instruction `for i in range(6):` puisque `range(6)` permet d'itérer sur la liste [0,1,2,3,4,5]. Plus généralement `range(a,b)` permet d'itérer sur la liste des entiers compris entre `a` (**inclus**) et `b` (**exclu**).

Par exemple le programme suivant permet de calculer la moyenne des 100 premiers nombres entiers impairs.

```
somme,n = 0,100
for x in range(n):
    somme = somme + 2*x+1
moyenne = somme/n
```

Boucle non bornée

Scratch propose un bloc `répéter jusqu'à …` , Python propose plutôt des boucles `while` (c'est-à-dire « tant que »).

Le programme suivant permet par exemple de chercher l'indice du premier terme d'une suite géométrique supérieur ou égal à 10000.

```
def indicePremierTerme(q,M):
    u = 1
    n = 0
    while u < M:
        u = u * q
        n = n + 1
    return n
```

L'appel `indicePremierTerme(1.25,10000)` renvoie par exemple `42`.

Fonctions

Quelques exemples de fonctions ont déjà été donnés, dont la définition commence par le mot-clé `def`. Précisons qu'une fonction peut avoir un nombre quelconque d'arguments (on dit aussi *paramètres*). Elle peut renvoyer zéro, une ou plusieurs valeurs à l'aide d'une instruction `return` qui stoppe l'exécution de la fonction. Par exemple, voici comment renvoyer la moyenne et la variance d'une liste de nombres.

Les listes **ne font pas** l'objet d'un enseignement spécifique en seconde, mais peuvent être utilisées comme objets simples et pratiques. Une liste `L` s'écrit entre crochets, `L = [2,3,5,7,11,13,17]` par exemple ; sa longueur est simplement `len(L)`.

```
def moyenneVariance(L):
    s,s2 = 0,0
    for x in L:
        s = s + x
        s2 = s2 + x ** 2 # on note **2 le carré
    # ici s est la somme et s2 la somme des carrés
    n = len(L) # la longueur de la liste
    return s/n, s2/n - (s/n)**2 # la fonction renvoie deux valeurs

(m,v) = moyenneVariance([12,343,11,14,43,62])
```

Il existe une façon très commode d'écrire des fonctions sans nommage, comme en mathématiques quand on écrit $x \longmapsto x^2 - 3x + 5$. On utilise en Python la notation `lambda x : x*x - 3*x + 5`.

Imaginons qu'on ait écrit une fonction `minimumLocal` de recherche d'un minimum d'une fonction sur un intervalle.

On pourra évaluer `minimumLocal(lambda x : x*x - 3*x + 5,0,6)` sans avoir besoin de définir et nommer une fonction par `def` …

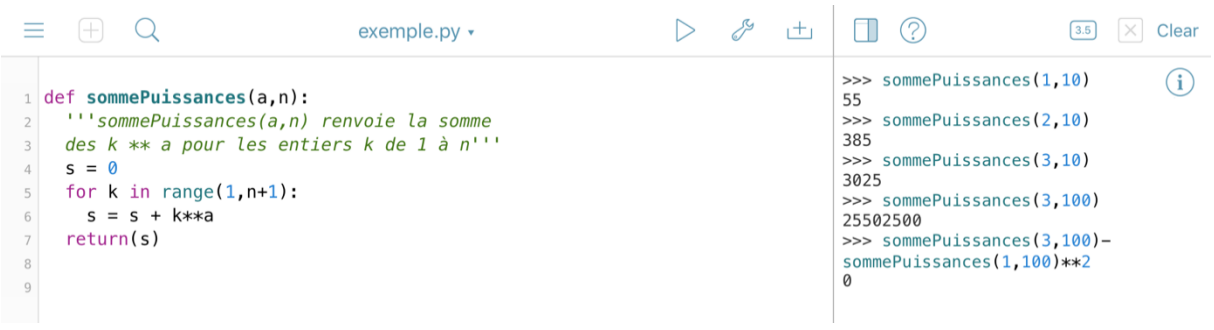
Le mode console

Il est important de distinguer très clairement la conception des algorithmes et leur utilisation. Les élèves écriront des fonctions qui pourront être enregistrées dans des fichiers, appelés scripts ou modules, qu'on peut développer, enrichir et réutiliser lors de séances successives.

Ces modules peuvent être exécutés dans un mode console, afin de tester et mettre au point les algorithmes et les programmes. L'exécution d'un module comportant la définition d'une fonction `f` ne produit aucun affichage particulier dans la console, même si la définition a été prise en compte : on peut maintenant, dans la console, faire des appels du type `f(2)` et la console affiche la valeur renvoyée par la fonction.

On notera que les notions d'entrées-sorties (fonctions `input` et `print`) ne sont pas développées dans ce document : elles ne relèvent pas de la pensée algorithmique et l'accent mis par le programme sur la notion de fonction permet de s'en libérer complètement.

La copie d'écran suivante montre, à gauche, un module dont l'exécution est lancée par un clic sur le bouton ▷ ; à droite on voit la console, où on exécute quelques instructions pour tester la correction du module. Les commandes de la console sont annoncées par l'invite `>>>`. En revanche, le module est un simple fichier texte, qui pourrait être retravaillé dans n'importe quel éditeur de texte.

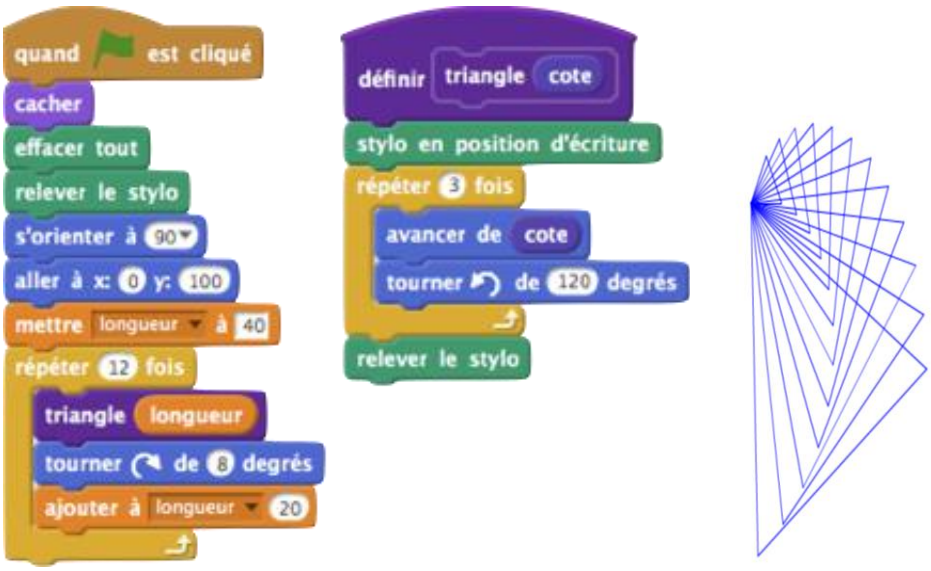


De Scratch à Python

Les élèves de seconde ont suivi au collège un enseignement d'algorithmique et de programmation, dans le cadre des mathématiques et de la technologie. En mathématiques, à l'aide de Scratch, ils ont utilisé des boucles, des instructions conditionnelles.

En classe de seconde, le passage de Scratch à Python peut être immédiat ou progressif, suivant les choix pédagogiques de l'enseignant. Les deux langages comportent, au-delà des différences évidentes de forme, des similitudes qui facilitent la transition.

Voici un exemple : un tracé géométrique en Scratch, et son analogue en Python.



L'utilisation de la bibliothèque `turtle` de Python permet d'écrire un programme très similaire.

```
import turtle

def figure():

    turtle.hideturtle()
    turtle.clear()
    turtle.up()
    turtle.setheading(0)
    turtle.goto(0,100)
    longueur = 40
    for i in range(12):
        triangle(longueur)
        turtle.right(8)
        longueur = longueur + 20

def triangle(cote):
    turtle.down()
    for i in range(3):
        turtle.forward(cote)
        turtle.left(120)
    turtle.up()
```

Il suffit de taper `figure()` dans la console pour lancer l'exécution.

Exemples de situations

Ce document est une ressource pour la formation des professeurs.
Il ne prétend aucunement prescrire telle ou telle pratique pédagogique dans les classes.
Il présente quelques exemples de situations qui peuvent inspirer les professeurs. Il appartient à chaque enseignant, dans le cadre de sa liberté pédagogique, de construire des activités à mener en classe, en s'inspirant au besoin des situations décrites dans ce document.
Les exemples proposés sont de niveaux de difficulté différents, tant du point de vue mathématique que du point de vue algorithmique. Les concepts mathématiques utilisés relèvent du programme de seconde ou des programmes du cycle terminal. Certains algorithmes proposés peuvent être écrits différemment. Certains peuvent être simplifiés, d'autres peuvent donner lieu à des prolongements.
Les modalités de mise en œuvre peuvent être variées : utilisation ponctuelle d'heures dédoublées pour un travail sur machine, utilisation de tablettes ou classes mobiles, vidéo-projection en classe entière, travail sur papier ou au tableau, intégration à des séances d'accompagnement personnalisé, mini-projets pouvant être réalisés en groupe…

Statistiques descriptives

Cette situation peut donner lieu à des activités en classe de seconde.

Les statistiques descriptives sont travaillées depuis le cycle 4. Le tableur constitue un outil important, mais le recours à la programmation présente deux grands intérêts :

- Comprendre et manipuler la définition des concepts : créer une fonction moyenne nécessite d'avoir compris la définition de l'indicateur.
- Manipuler de grandes séries qui peuvent être issues de données réelles, et qui sont bien plus facilement manipulables que sur un tableur.

Pour calculer la moyenne d'une série, on somme ses éléments grâce à une boucle `for x in serie` :

```
def moyenne(serie):
    n = len(serie)
    s = 0
    for x in serie:
        s = s + x
    return s/n
```

Pour déterminer la médiane, le plus simple est de commencer par trier les termes de la série en ordre croissant, grâce à l'instruction `serie.sort()`, qui modifie la liste `serie` en la triant, et de choisir ensuite le terme médian, selon la parité de l'effectif.

Quand on divise deux entiers `a` et `b`, `a % b` et `a//b` renvoient respectivement le reste et le quotient dans la division euclidienne alors que `a/b` renvoie le quotient décimal.

On rappelle que le symbole `=` est réservé à l'affectation, et doit être distingué du symbole `==` qui réalise le test d'égalité.

Attention : en Python, on indexe à partir de 0, donc `serie[0]` est le premier terme de la série, `serie[1]` le deuxième, etc.

```
def mediane(serie):
    n = len(serie)
    serie.sort()
    if n%2 == 0:
        return (serie[n//2]+serie[n//2 - 1])/2
    else:
        return serie[n//2]
```

Pour tirer au hasard une série d'entiers entre 1 et 50, on peut utiliser la fonction `randint` de la bibliothèque *random*. On commence par ouvrir la bibliothèque par l'instruction import `random`, l'appel de fonction est alors `random.randint(1,50)`.

S'inspirant de l'écriture mathématique d'un ensemble en compréhension, par exemple

$$\{ k^2 + 2,\ k \in [\![0,15]\!] \},$$

Python propose une syntaxe utile pour la création d'une liste en compréhension :

```
[ k**2 + 1 for k in range(16) ].
```

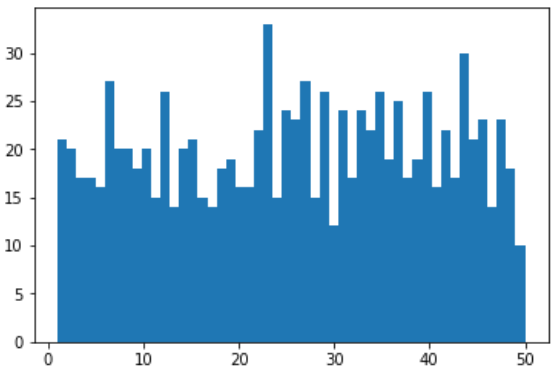
On peut ainsi utiliser l'expression `[random.randint(1,50) for i in range(1000)]` pour créer la liste simulant un échantillon de taille 1000 pour la loi uniforme sur $[\![1,50]\!]$.

On ouvre de même le module de tracé de la bibliothèque *matplotlib* par l'instruction `import matplotlib.pyplot as plt` de sorte qu'on peut utiliser l'abréviation `plt.hist` pour la fonction de tracé d'un histogramme. Le deuxième argument de cette fonction précise le nombre de classes de l'histogramme.

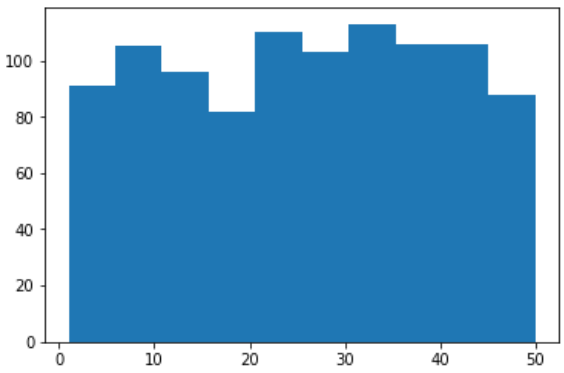
```
import matplotlib.pyplot as plt
import random
# on crée une série de 1000 entiers aléatoires entre 1 et 50
L = [random.randint(1,50) for i in range(1000)]
```

La commande `plt.show()` permet d'afficher une figure, après qu'on a exécuté toutes les commandes de tracé qu'on souhaitait.

```
plt.hist(L,50)
plt.show()
```




```
plt.hist(L,10)
plt.show()
```



Algorithme d'Euclide

L'arithmétique ne constitue pas une partie du programme de seconde en tant que telle, mais la division euclidienne et les nombres premiers ont été rencontrés au cycle 4 du collège. L'arithmétique permet de faire travailler les élèves sur les propriétés des nombres et la logique, et de les entraîner au raisonnement à travers l'utilisation des nombres entiers. Les trois exemples qui suivent permettent d'illustrer ces idées.

On considère deux entiers naturels non nuls a et b. Supposons par exemple $a \geq b$.

On écrit la division euclidienne : $a = bq + r$ où le reste r vérifie $0 \leq r < b$.

Si un entier d divise à la fois a et b, alors il divise également b et $r = a - bq$.

Inversement si un entier d divise à la fois b et r, alors il divise également $a = bq + r$ et b.

Autrement dit : les diviseurs communs à a et b sont exactement les mêmes que les diviseurs communs à b et r. L'idée de l'algorithme est donc de remplacer le couple (a, b) par le couple $(a', b') = (b, r)$ et d'itérer jusqu'à ce que le reste soit nul : les diviseurs communs du couple de départ sont les diviseurs communs du couple obtenu à chaque étape. Comme les diviseurs communs à a' et 0 sont les diviseurs de a', le plus grand d'entre eux, soit a' lui-même, est le PGCD du couple de départ.

Remarquons que si $a < b$, la division euclidienne s'écrit $a = b \times 0 + a$ de sorte que la première étape de la boucle intervertit a et b.

```
def euclide(a,b) :
    assert(a>0 and b>0)
    while b!=0:
        a,b = b,a%b
        # a%b est le reste dans la division euclidienne de a par b
    return a
```

La boucle s'arrête à coup sûr car la suite des deuxièmes éléments des couples consécutifs est une suite strictement décroissante d'entiers naturels.

On a utilisé l'instruction `assert(…)` qui permet de vérifier que les valeurs données en argument à la fonction vérifient bien les hypothèses prévues. Dans le cas contraire, Python interrompra immédiatement l'exécution en affichant un message d'erreur.

Changement de base de numération

On se propose d'écrire une fonction `ecritureBinaire` qui prend en argument un entier naturel n non nul et qui renvoie la liste des bits de son écriture en base 2, ainsi que la fonction réciproque `ecritureDecimale`.

Par exemple, la liste des bits de l'écriture binaire de 23 est `[1,0,1,1,1]` car $23 = 16 + 4 + 2 + 1 = 2^4 + 2^2 + 2^1 + 2^0$.

L'écriture binaire de n est $[b_p, b_{p-1}, \dots, b_2, b_1, b_0]$ où chaque bit b_i vaut 0 ou 1 quand on peut écrire : $n = \sum_{k=0}^{p} b_k 2^k$. Ainsi $b_0 = 1$ si et seulement si n est impair, et $[b_p, b_{p-1}, \dots, b_2, b_1]$ est l'écriture décimale du quotient entier de n par 2.

On en déduit l'écriture de la fonction `ecritureBinaire`. L'instruction `L = []` crée une liste vide. `L.append(x)` ajoute un élément `x` en fin de la liste `L`. Enfin `L.reverse()` retourne la liste.

```
def ecritureBinaire(n):
  L = []
    while n>0:
        L.append(n%2)  # n%2 est le reste dans la division par 2
        n = n//2       # n//2 est le quotient
    L.reverse()
    return L
```

L'appel `ecritureBinaire(23)` renvoie la liste `[1,0,1,1,1]`.

La fonction réciproque s'obtient de façon analogue :

```
def ecritureDecimale(L):
    n = 0
    for x in L:
        n = 2*n + x
    return n
```

L'appel `ecritureDecimale([1,0,1,1,1])` renvoie l'entier `23`.

Logarithme entier en base 2

Il s'agit en fait de calculer le nombre de bits de l'écriture binaire d'un entier naturel non nul n.

```
def nombreBits(n) :
    l = 1
    while n>1 :
        n,l = n//2,l+1
    return l
```

L'appel `nombreBits(23)` renvoie ainsi `5`.

On peut pareillement calculer la somme des bits d'un entier.

```
def sommeBits(n):
    s=0
    while n>0:
        s = s + n%2
        n = n//2
    return s
```

L'appel `sommeBits(23)` renvoie ainsi `4`.

Test de primalité

Cette situation peut donner lieu à des activités en classe de seconde.

Soit n un entier supérieur ou égal à 2. Si n n'est pas premier, et si d est son plus petit diviseur supérieur ou égal à 2, on peut écrire $n = d \times q$ avec $d \leq q$ puisque on a choisi le plus petit diviseur d ; on constate alors que $d \leq \sqrt{n}$.

Pour tester si n est premier, il suffit donc de tester sa divisibilité par les entiers plus petits ou égaux à sa racine carrée.

On obtient le programme suivant en Python.

On a utilisé l'instruction `assert(n>=2)` pour vérifier l'hypothèse faite sur l'argument.

La fonction renvoie une valeur booléenne, c'est-à-dire ou bien `True` ou bien `False`.

On remarque que dès qu'on trouve un diviseur d on peut renvoyer `False` : l'instruction `return` permet à la fois de renvoyer le résultat attendu et d'interrompre l'itération.

```
def estPremier(n):
    assert(n>=2)
    d = 2
    while d*d <= n:
        if n%d == 0:
            return False
        d = d +1
    # on n'a trouvé aucun diviseur
    return True
```

Décomposition en produit de facteurs premiers

Cette situation peut donner lieu à des activités en classe de seconde.

En s'inspirant de la fonction précédente, on peut écrire une fonction qui renvoie une liste de facteurs premiers dont le produit est le nombre n fourni en argument.

```
def factorisation(n):
    L = []
    d = 2
    while n>1:
        if n%d == 0:
            # on a trouvé un diviseur !
            while n%d == 0:
                # on le factorise autant que possible et
                # on l'ajoute à chaque fois dans la liste
                L.append(d)
                n = n//d
        d = d + 1
    return L
```

L'indentation est ici tout à fait cruciale.

L'appel `factorisation(2018)` renvoie `[2, 1009]` alors que l'appel `factorisation(2016)` renvoie `[2, 2, 2, 2, 2, 3, 3, 7]`. Un exercice un peu plus difficile consiste à renvoyer une liste de couples (p, e) où e est l'exposant du facteur premier p dans la décomposition.

`factorisationPlus(2016)` doit renvoyer `[(2,5),(3,2),(7,1)]`.

Longueur d'un arc de courbe

Cette situation peut donner lieu à des activités en classe de seconde.

Les fonctions et leur représentation constituent un des éléments essentiels de la culture mathématique que les élèves doivent acquérir au lycée. L'activité ici proposée constitue un détour original pour faire travailler ces notions, en dehors des classiques tracés de courbe. Elle constitue aussi une ouverture vers les problèmes d'approximation numérique.

On considère une fonction continûment dérivable f définie sur un intervalle $[a, b]$, et on cherche une valeur approchée de la longueur ℓ de la courbe représentative de f.

Pour cela, on subdivise l'intervalle en n petits intervalles $[x_i, x_{i+1}]$ de même longueur $x_{i+1} - x_i = \frac{b-a}{n}$, avec $x_0 = a$, $x_n = b$. Notons $A_i$le point de coordonnées $(x_i, f(x_i))$ de la courbe.

On approche alors sur $[x_i, x_{i+1}]$ la courbe par le segment $[A_i A_{i+1}]$.

On prend comme valeur approchée de ℓ la somme des longueurs de ces segments. Quand n tend vers l'infini, on peut prouver que la somme de ces longueurs tend vers ℓ.

```
from math import *
def distance(x1,y1,x2,y2):
    return sqrt((x1-x2)**2 + (y1-y2)**2)

def longueurCourbe(f,a,b,n):
    longueur = 0
    x1,y1 = a,f(a)
    h = (b-a)/n
    for i in range(n):
        x2 = x1 + h
        y2 = f(x2)
        longueur = longueur + distance(x1,y1,x2,y2)
        x1,y1 = x2,y2
    return longueur
```

La commande `from math import *` permet d'utiliser de façon aisée les fonctions mathématiques usuelles, comme la racine carrée `(sqrt)`, le cosinus `(cos)`, etc. La commande `help('math')` permet d'avoir la liste des fonctions disponibles avec leur description.

Si on avait seulement ouvert la bibliothèque *math* avec la commande `import math`, il faudrait écrire `math.sqrt` au lieu de `sqrt`.

On remarque que Python admet sans difficulté de prendre en argument une fonction.

On peut tester la fonction sur le quart de cercle unité, de longueur $\frac{\pi}{2}$, en utilisant la notation `lambda x : expression` pour dénoter une fonction sans la nommer.

L'appel `2*longueurCourbe(lambda x: sqrt(1-x**2),0,1,10)` renvoie `3.1415832833677633`, alors que l'appel `2*longueurCourbe(lambda x: sqrt(1-x**2),0,1,1000)` renvoie `3.141591493294789`.

Résolution approchée d'une équation par dichotomie

Cette situation peut donner lieu à des activités en classe de seconde.

La méthode de dichotomie constitue un procédé dont la compréhension et la mise en œuvre peuvent être particulièrement délicates pour les élèves. Le détour par l'algorithmique permet de « faire

fonctionner » la méthode et de l'observer en acte. Cette activité peut se décliner de façons multiples pour les élèves. Il peut être simplement question de relier les blocs d'instructions aux éléments correspondant dans le texte qui décrit l'algorithme.

Soit f une fonction continue sur un intervalle I.

Soit a et b deux points de I tels que $a < b$ et $f(a) \times f(b) < 0$.

On sait qu'il existe au moins une solution de l'équation $f(x) = 0$ sur l'intervalle l'intervalle $[a, b]$. Le principe de l'algorithme de dichotomie consiste à considérer le point $c = \frac{a+b}{2}$. Si $f(a) \times f(c) \leq 0$, on sait qu'il existe une solution sur l'intervalle $[a, c]$. Sinon, on a $f(b) \times f(c) \leq 0$, et il existe une solution sur l'intervalle $[c, b]$. Ainsi, à chaque étape on passe d'un intervalle contenant une solution à un intervalle de longueur moitié contenant une solution.

On itère le procédé jusqu'à obtenir un intervalle de longueur inférieure à la précision requise.

```
def dichotomie(f,a,b,epsilon=0.0001):
    assert(f(a)*f(b) < 0 and a < b)
    while b-a > epsilon:
        c = (a+b)/2
        if f(a)*f(c) <= 0:
            a,b = a,c
        else:
            a,b = c,b
    return (a+b)/2
```

La commande `assert(…)` permet de s'assurer que l'on est bien dans les hypothèses requises.

On remarque que dans la définition de la fonction, une valeur par défaut est fournie pour `epsilon`. Cela signifie que l'appel `dichotomie(f,a,b)` est équivalent à l'appel `dichotomie(f,a,b,0.0001)`. Pour imposer une précision différente de la valeur par défaut, il suffit de la préciser `dichotomie(f,a,b,0.0000001)` par exemple.

Après avoir ouvert la bibliothèque *math* par la commande `from math import *` on peut par exemple demander `dichotomie(cos,0,2,0.01)` qui renvoie `1.57421875` alors que l'appel `dichotomie(cos,0,2)` utilise la valeur par défaut de epsilon et renvoie `1.570770263671875`.

Un ordinateur ne travaille pas avec des nombres réels, mais avec des *flottants*, c'est-à-dire un sous-ensemble des nombres décimaux dont la précision est limitée par des contraintes liées au codage en mémoire.

C'est ainsi qu'en Python, le test d'égalité `3+10**(-16)==3` s'évalue en `True` alors que le test `3+10**(-15)==3` s'évalue en `False`. On retiendra qu'il faut éviter de tester l'égalité entre deux flottants, et préférer la recherche d'une précision donnée. En revanche, bien sûr, il n'y a aucun problème à comparer deux nombres entiers.

Stabilisation des fréquences

En probabilités, on est amené à utiliser des générateurs pseudo-aléatoires, proposés par la bibliothèque *random*, qu'on ouvre avec la commande `import random`.

Les fonctions d'usage le plus fréquent sont :

- `random.random()` qui renvoie un nombre réel (pseudo)-aléatoire de l'intervalle semi-ouvert `[0,1[` ;
- `random.randint(a,b)` qui renvoie un nombre entier (pseudo)-aléatoire compris entre `a` et `b` (bornes incluses) ;
- `random.choice(L)` qui renvoie un élément tiré au sort de la liste `L`.

Considérons l'expérience aléatoire qui consiste à jeter un dé équilibré et à définir ainsi la variable aléatoire X : si le dé donne 1, $X = 1$; si le dé donne 5 ou 6, $X = 4$; sinon $X = 2$.

```
def experience():
    de = random.randint(1,6)
    # on tire au hasard un nombre entier parmi 1,2,3,4,5,6
    if de < 2:
        X = 1
    elif de < 5:
        X = 2
    else:
        X = 4
    return X
```

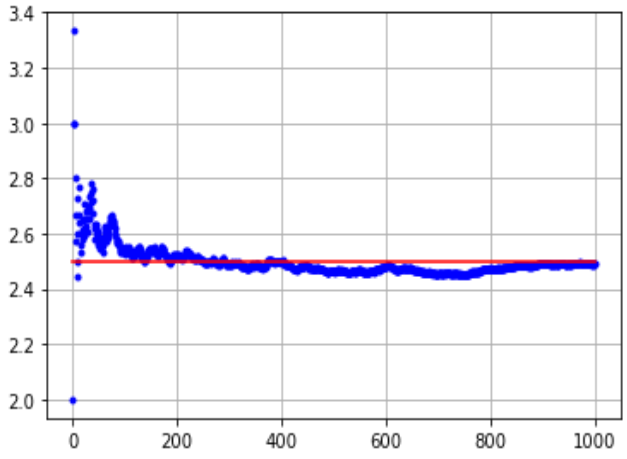
On souhaite observer la moyenne des valeurs prises par la variable aléatoire sur un nombre quelconque d'expériences, et plus précisément l'évolution de cette moyenne en fonction du nombre d'expériences.

On utilise la bibliothèque *matplotlib.pyplot* pour donner une représentation graphique de cette évolution. Il est pratique de l'ouvrir en lui donnant un nom abrégé, par la commande `import matplotlib.pyplot as plt`.

Les différentes commandes graphiques travaillent sur une même figure, qu'on fait apparaître par l'instruction `plt.show()`. On peut ainsi tracer une courbe, dessiner une grille, ajouter une légende, etc. sur une même figure.

```
def evolutionMoyenne(experience,nExperiences):
    s = experience()
    n = 1
    L = [s]     # moyenne sur 1 expérience
    while n < nExperiences:
        n = n+1
        s = s + experience()
        L.append(s/n)  # on ajoute la moyenne sur n expériences
    plt.plot(list(range(1,nExperiences+1)),L,'b.')
    plt.plot([1,nExperiences],[2.5, 2.5],'r-')
    plt.grid()
    plt.show()
```

L'instruction `plt.plot(listeX,listeY,motif)` représente graphiquement des points dont la liste des abscisses est `listeX`, la liste des ordonnées `listeY`. L'argument `motif` permet de préciser la couleur (b pour bleu, r pour rouge) et le type de tracé (`.` pour des points isolés, `-` pour une ligne continue).



Ainsi on a tracé en rouge un segment horizontal qui correspond à la valeur autour de laquelle se stabilise la moyenne : c'est l'espérance, égale à 2.5, de la variable aléatoire X.

C'est ainsi que l'exécution de la commande `evolutionMoyenne(experience,1000)` permet d'obtenir le graphique ci-contre (évidemment, à chaque appel on aura un résultat légèrement différent car les tirages aléatoires seront différents, mais on observera la stabilisation de la moyenne autour de l'espérance).

Aiguille de Buffon

Cette situation peut donner lieu à des activités en classe de seconde.

On considère l'expérience consistant à jeter une aiguille sur un parquet et à observer si l'aiguille traverse la frontière entre deux lattes du parquet.

Dans notre modèle, les lattes seront considérées comme des bandes parallèles de largeur unité, et on repère le plan de sorte que les frontières des lattes sont les droites d'équations $y = k$, où k est un entier relatif. L'aiguille est de longueur ℓ.

On tire au hasard la position d'une extrémité de l'aiguille, obtenant des coordonnées (x, y). En fait x n'a aucune importance dans notre modèle. Pour l'ordonnée, on peut se ramener à tirer y dans l'intervalle $[0,1[$, la situation étant invariante par translation de vecteur $(0, k)$ pour tout entier relatif k.

Une fois choisie cette extrémité de l'aiguille, il reste à déterminer l'angle que fait l'aiguille avec la direction (Oy). Cet angle θ sera tiré au hasard dans l'intervalle $\left[-\frac{\pi}{2}, \frac{\pi}{2}\right[$ à l'aide de l'expression `random.random()*pi - pi/2`.

L'ordonnée de la deuxième extrémité de l'aiguille sera alors $y' = y + \ell \times \sin\theta$.

L'aiguille traverse la frontière entre deux lattes si et seulement si les parties entières de y et y' sont distinctes.

La fonction partie entière est en Python la fonction `floor` de la bibliothèque *math*.

On écrit donc la fonction suivante pour modéliser l'expérience.

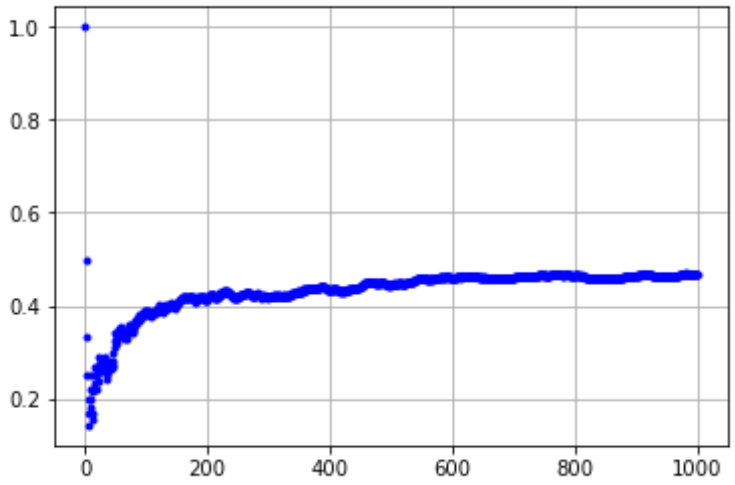
```
def buffon(l):
    y1 = random.random()
    theta = random.random()*pi - pi/2
    y2 = y1 + l*sin(theta)
    if floor(y1) != floor(y2):
        return 1
    else:
        return 0
```

En Python, le test d'égalité s'écrit `==` et le test contraire `!=`.

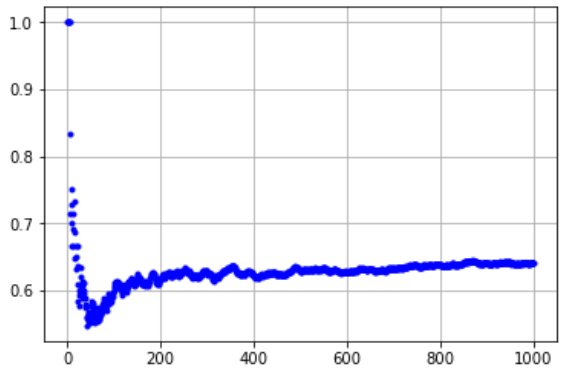
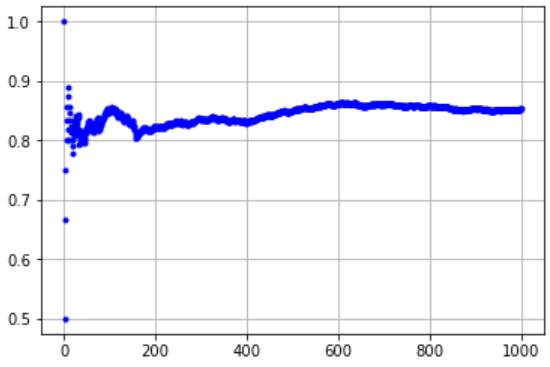
On peut vérifier la stabilisation vers la probabilité que l'aiguille traverse la frontière entre deux lattes en utilisant la fonction `evolutionMoyenne` précédente.

La notation `lambda : buffon(0.7)` permet de représenter une fonction sans argument, requise par `evolutionMoyenne`.

L'appel `evolutionMoyenne(lambda : buffon(0.7),1000)` produit la figure suivante.



Voici les figures obtenues avec des longueurs d'aiguille respectivement égales à 1 et 2.5.

$l=1$	$l=2.5$
	

On peut chercher à étudier comment la probabilité autour de laquelle se stabilisent les fréquences de succès dépend de la longueur ℓ de l'aiguille.

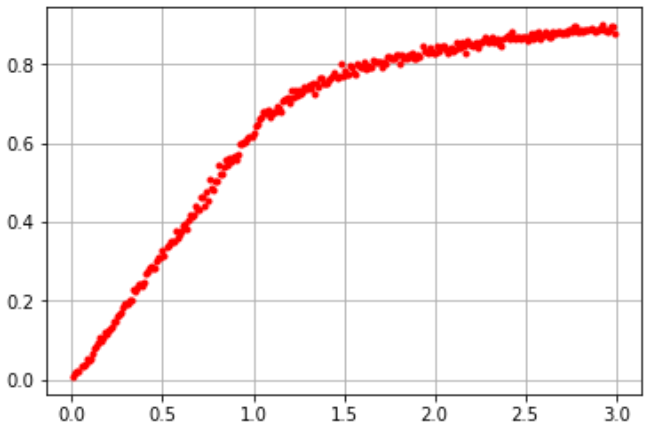
Pour cela on va identifier la probabilité avec la fréquence au bout de 5000 expériences.

```
def probaBuffon(l):
    s = 0
    for i in range(5000):
        s = s + buffon(l)
    return s/5000
```

On peut alors taper les lignes suivantes dans la console :

```
x = [t/100 for t in range(1,300)]
y = [probaBuffon(t) for t in x]
plt.plot(x,y,'r.')
plt.grid()
plt.show()
```

et on obtient la figure suivante.



On observe que la probabilité dépend linéairement de ℓ tant que $\ell \leq 1$, puis que la courbe change d'aspect, une asymptote d'équation $y = 1$ apparaissant : en effet, quand la longueur de l'aiguille tend vers l'infini, il est clair que la probabilité qu'elle coupe la frontière entre deux lattes tend vers 1.

On peut démontrer que le coefficient directeur de la partie linéaire est $\frac{2}{\pi}$ et qu'ensuite la probabilité vaut

$$1-\frac{2}{\pi}\left(\arcsin\frac{1}{\ell}-\left(\ell-\sqrt{\ell^2-1}\right)\right),$$

mais c'est évidemment plus difficile.

Intervalle de fluctuation d'une fréquence au seuil de 95 %

Cette situation peut donner lieu à des activités en classe de seconde.

On considère un échantillon de taille n constitué des résultats de n répétitions indépendantes de la même expérience, qui a une probabilité p de succès, et on relève la fréquence f du succès de l'expérience.

```
def echantillon(p,n):
    '''simule n répétitions indépendantes d'une expérience
    dont le succès est de probabilité p,
    et renvoie la fréquence des expériences réussies'''
    succes = 0
    for i in range(n):
        if random.random() <= p:
            succes = succes+1
    f = succes /n
    return f
```

On a utilisé ici un type particulier de commentaire, `'''commentaire'''` qui peut s'étendre sur plusieurs lignes, qui suit immédiatement la ligne `def … (…)` : et qui permet d'expliquer l'objectif d'une fonction. Ce commentaire sert d'aide pour l'utilisateur, qui peut le retrouver en évaluant `help(echantillon)`.

Une évaluation `echantillon(0.2,10000)` peut renvoyer 0.1981, par exemple (le résultat est évidemment aléatoire).

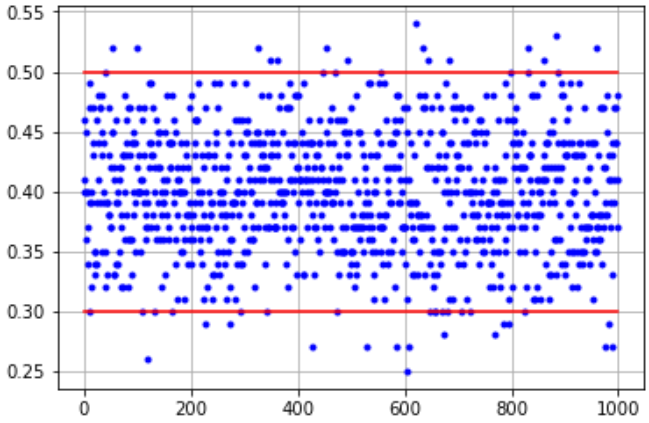
Le programme de mathématiques invite à observer avec quelle probabilité la fréquence f se retrouve dans l'intervalle de fluctuation $\left[p-\frac{1}{\sqrt{n}},p+\frac{1}{\sqrt{n}}\right]$.

```
def fluctuation(p,n):
'''renvoie    (a,b)    bornes    de    l'intervalle    de    fluctuation
au seuil de 95 % pour des échantillons de taille n
et une proportion p du caractère'''
   return(p-1/sqrt(n),p+1/sqrt(n))
On peut écrire la fonction suivante pour en avoir un aperçu graphique.
def grapheFluctuation(p,n,nbEchantillons):
    a,b = fluctuation(p,n)
    L = []
    for i in range(nbEchantillons):
        L.append(echantillon(p,n))
    plt.plot(list(range(0,nbEchantillons)),L,'b.')
    plt.grid()
    plt.plot([0,nbEchantillons-1],[a,a],'r-')
    plt.plot([0,nbEchantillons-1],[b,b],'r-')
    plt.show()
```

Ici p est la probabilité de succès d'une expérience, n est la taille de chaque échantillon et `nbEchantillons` le nombre d'échantillons sur lesquels on mène l'observation.

On trace en rouge les horizontales correspondant aux bornes de l'intervalle de fluctuation considéré, et chaque point bleu représente la fréquence obtenue pour un échantillon.

Voici ce que produit l'appel `grapheFluctuation(0.4,100,1000)`.



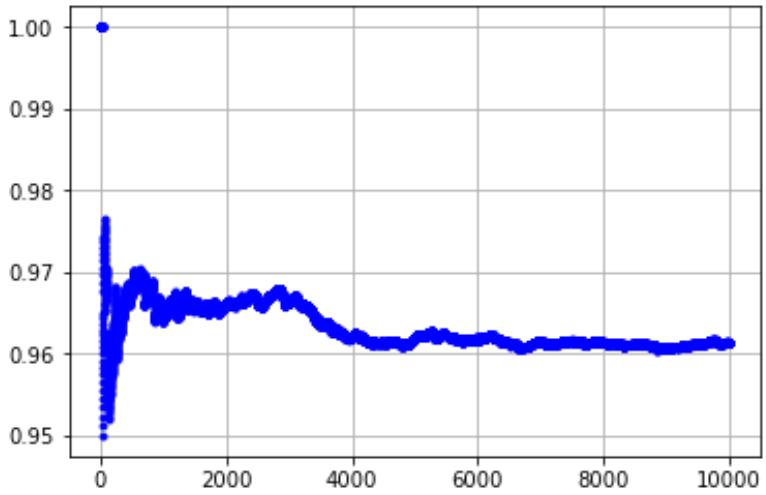
On peut aussi choisir d'évaluer la proportion des échantillons tels que la fréquence f observée est dans l'intervalle de fluctuation en fonction du nombre d'échantillons simulés.

On observe que cette proportion est rapidement supérieure à 0,95.

C'est ce que fait la fonction suivante, qui utilise un compteur d'échantillons corrects, c'est-à-dire tels que la fréquence correspondante est dans l'intervalle souhaité.

```
def grapheProportionDansIntervalleFluctuation(p,n,nbEchantillons):
    a,b = fluctuation(p,n)
    L = []
    corrects = 0
    for i in range(1,nbEchantillons):
        f = echantillon(p,n)
        if f >= a and f <= b:
            corrects = corrects + 1
        proportion = corrects/i
        L.append(corrects/i)
    plt.plot(list(range(1,nbEchantillons)),L,'b.')
    plt.grid()
    plt.show()
```

Voici le résultat de l'appel `grapheProportionDansIntervalleFluctuation(0.4,100,10000)`



Casser un bâton

Cette situation peut donner lieu à des activités en classe de seconde.

On considère la situation suivante : on prend un bâton, qu'on casse en 3 morceaux. L'expérience est réussie si les 3 morceaux constituent les côtés d'un triangle.

La condition à vérifier pour trois longueurs a, b, c est la conjonction :

$$a + b \geq c \text{ et } b + c \geq a \text{ et } c + a \geq b.$$

```
def correct(a,b,c):
    return a+b>=c and b+c>=a and c+a>=b
```

On peut considérer que le bâton a une longueur unité.

Il y a plusieurs façons de procéder : on tire au hasard deux réels de $]0,1[$, on note x le plus petit et y le plus grand. x et y sont les abscisses des points de casse. Les trois segments sont alors de longueurs x, $y - x$ et $1 - y$.

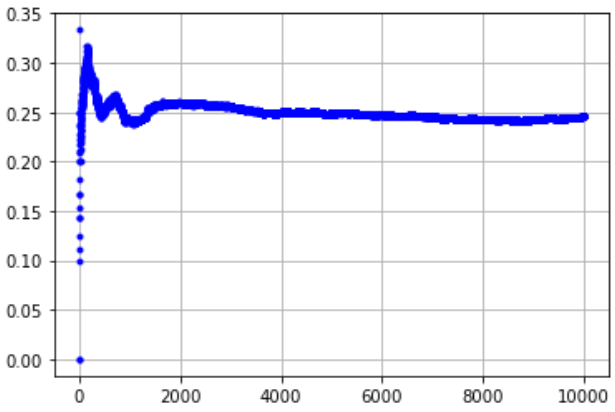
```
def casseBaton1():
    x = random.random()
    y = random.random()
    if x > y:
            x,y = y,x   # permet d'échanger les valeurs de x et y
    return correct(x,y-x,1-y)
```

La fonction `casseBaton1` renvoie `True` si l'expérience est réussie et `False` dans le cas contraire.

La fonction suivante permet d'observer la stabilisation de la fréquence des succès vers une probabilité égale à $\frac{1}{4}$ (c'est un exercice intéressant de le démontrer).

```
def chercheProbabilite(experience,nExperiences):
    n,succes = 0,0
    L = []
    while n < nExperiences:
        n = n+1
        if experience():
            succes = succes+1
        L.append(succes/n)
    plt.plot(list(range(1,nExperiences+1)),L,'b.')
    plt.grid()
    plt.show()
```

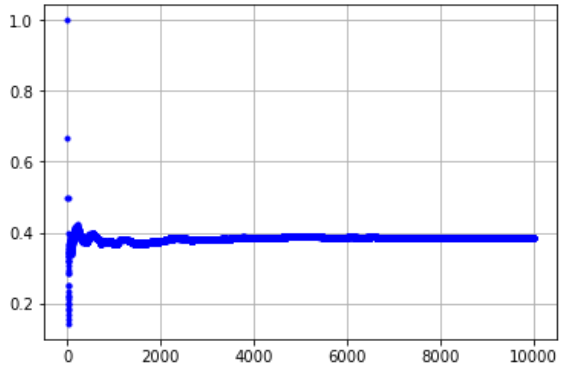
L'appel `chercheProbabilite(casseBaton1,10000)` produit la figure suivante.



Considérons un autre modèle de l'expérience : on casse le bâton une première fois (en tirant au hasard une abscisse x), puis on casse le plus gros morceau (en tirant à nouveau au hasard l'abscisse de casse).

```
def casseBaton2():
    x = random.random()
    if x > 1/2:    # le grand morceau est à gauche
        x,y = x * random.random(), x
    else:          # le grand morceau est à droite
        x,y = x, x + (1-x) * random.random()
    return correct(x,y-x,1-y)
```

L'appel `chercheProbabilite(casseBaton2,10000)` produit la figure suivante.



Quelle est la probabilité de succès ?

On pourrait encore envisager un autre modèle d'expérience : on casse le bâton une première fois, puis on choisit au hasard l'un ou l'autre morceau, qu'on casse à son tour.

Marche aléatoire à une dimension

On considère un mobile qui se déplace sur un axe à des tops d'horloge. Son abscisse initiale est $x = 0$. À chaque top d'horloge, son abscisse augmente ou diminue d'une unité, de façon équiprobable.

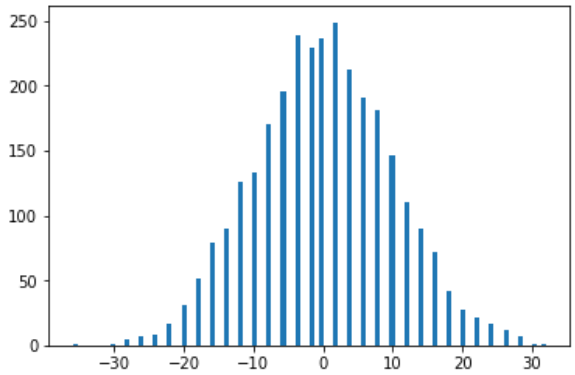
On peut simuler ce comportement ainsi :

```
def marche(nbPas):
    '''renvoie l'abscisse d'arrivée, partant de 0,
    après nbPas pas de +1 ou -1'''
    x = 0
    for i in range(nbPas):
        if random.random()<0.5:
            x = x + 1
        else:
            x = x - 1
    return x
```

On peut s'intéresser à la position du mobile après un certain nombre de pas. Chaque simulation est susceptible de donner une position d'arrivée différente. Pour mettre en évidence la distribution de probabilité de la position d'arrivée, on simule un grand nombre de marches, et on trace l'histogramme correspondant des positions d'arrivée.

```
def histogrammeMarche1d(nbPas,nbMarches):
    L = []
    for i in range(nbMarches):
        L.append(marche(nbPas))
    plt.hist(L,100)
    plt.show()
```

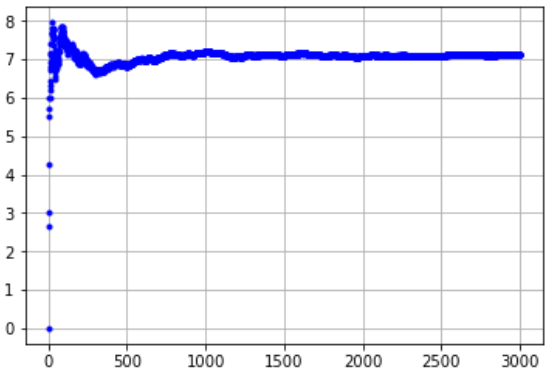
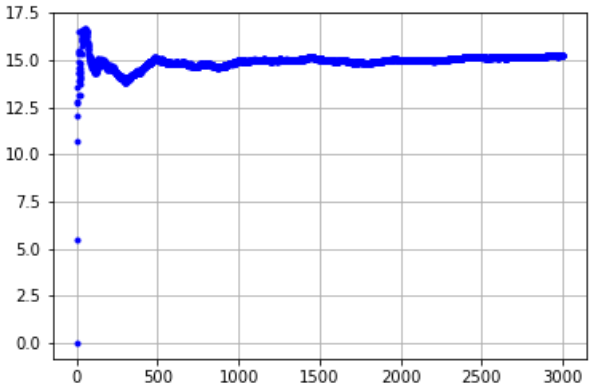
On obtient par exemple la figure suivante par l'appel `histogrammeMarche1d(100,3000)`.



On peut aussi s'intéresser au nombre de retours à la position de départ lors d'une marche.

```
def nbRetoursOrigine(nbPas):
    '''nombre de retours à l'origine après une marche'''
    x,r = 0,0
    for i in range(nbPas):
        if random.random()<0.5:
            x = x+1
        else:
            x = x-1
        if x==0:
            r = r+1
    return r
```

En utilisant la fonction `evolutionMoyenne` introduite dans l'activité « stabilisation des fréquences », on écrit, après avoir fixé un nombre de pas pour chaque marche, `evolutionMoyenne(lambda : nbRetoursOrigine(nbPas), 3000)` pour estimer l'espérance du nombre de retours à l'origine. On obtient les figures suivantes.

pour des marches de 100 pas	pour des marches de 400 pas
	

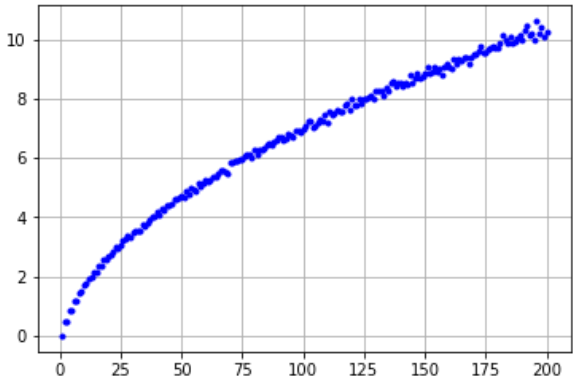
Cela donne envie de chercher comment évolue l'espérance du nombre de retours à l'origine d'une marche aléatoire en fonction du nombre de pas de la marche.

On approche pour cela l'espérance à la moyenne du nombre de retours sur 3000 marches, ce qui amène à écrire les fonctions suivantes.

```
def esperanceNbRetours(nbPas):
    n = 0
    for i in range(3000):
        n = n+nbRetoursOrigine(nbPas)
    return n/3000

def evolutionNbRetours(nbPasMaximum):
    x,L = [],[]
    for nbPas in range(1,nbPasMaximum+1):
        x.append(nbPas)
        L.append(esperanceNbRetours(nbPas))
    plt.plot(x,L,'b.')
    plt.grid()
    plt.show()
```

L'appel `evolutionNbRetours(200)` produit la figure suivante.



Urnes de Polyà

On considère une urne. On y introduit au départ r boules rouges et b boules bleues. Les boules sont indiscernables au toucher. À chaque étape, on tire au hasard une boule, on la remet dans l'urne en ajoutant une nouvelle boule de même couleur. On cherche à connaître la répartition des couleurs au cours du temps, selon les valeurs initiales de r et b.

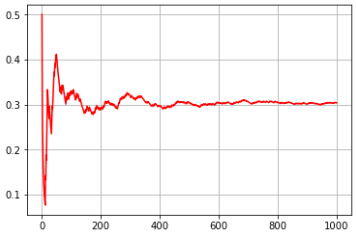
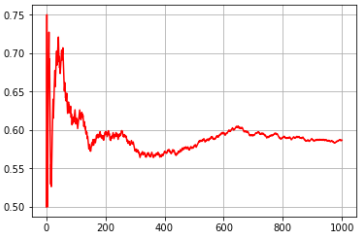
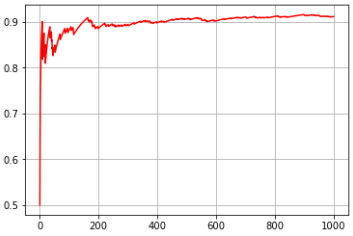
La simulation de plusieurs étapes se programme aisément. On introduit une liste L qui contiendra les proportions successives de boules rouges (la proportion de boules bleues étant toujours le complément à 1).

```
def urnesPolya(r,b,etapes):
    assert(r+b >= 1) # il faut au moins une boule au départ !
    L = [r/(r+b)]
    for i in range(etapes):
        n = random.randint(1,r+b)
        if n <= r:
            r,b = r+1,b
        else:
            r,b = r,b+1
        L.append(r/(r+b))
    x = list(range(0,etapes+1))
    plt.plot(x,L,'r-')
    plt.grid()
    plt.show()
```

On a utilisé la fonction `randint` de la bibliothèque *`random`* : `random.randint(a,b)` renvoie un entier aléatoire de l'intervalle fermé d'entiers $[\![a,b]\!]$. Pour chaque tirage, on imagine avoir numéroté les

boules, d'abord les rouges, puis les bleues. On choisit un numéro n de boule au hasard : s'il est plus petit que r, c'est qu'on a choisi une boule rouge, sinon c'est une boule bleue.

Voilà plusieurs figures produites par le même appel `urnesPolya(1,1,1000)`.

En partant de la même proportion initiale, en l'occurrence 50 %, on observe une stabilisation vers des proportions très différentes : environ 30 %, environ 60 %, environ 90 %.

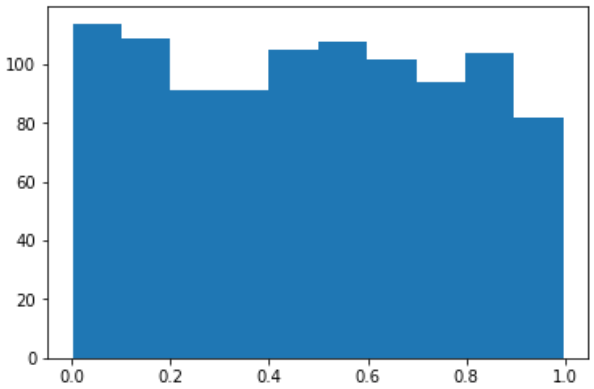
On aimerait avoir une idée de la distribution des proportions limites. Pour cela on va considérer la proportion obtenue au bout de 1000 tirages et approcher ainsi la proportion limite. On écrit donc la fonction suivante.

```
def uneExperiencePolya(r,b):
    for i in range(1000):
        if random.randint(1,r+b) <= r:
            r,b = r+1,b
        else:
            r,b = r,b+1
    return r/(r+b)
```

On peut maintenant réaliser plusieurs expériences et observer l'histogramme des proportions limites obtenues.

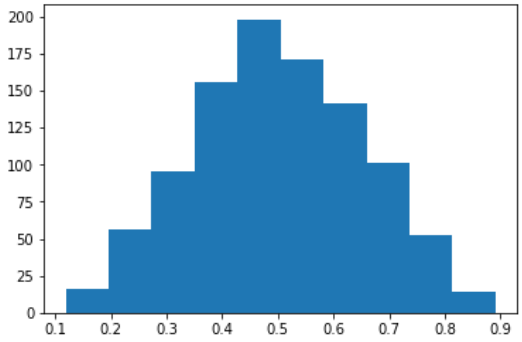
```
def experiencesPolya(r,b,nombre):
    L = []
    for i in range(nombre):
        L.append(uneExperiencePolya(r,b))
    plt.hist(L,10)
    plt.show()
```

L'appel `experiencesPolya(1,1,1000)` produit la figure suivante.

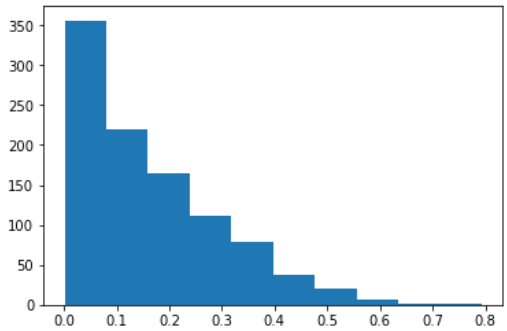


On peut démontrer que, partant d'une boule rouge et une boule bleue, toutes les proportions limites sont équiprobables. Mais il ne faudrait pas croire que cela soit vrai quand on part par exemple de 5 boules rouges et 5 boules bleues, ce qui fait pourtant la même proportion de 50 %.

En effet, l'appel `experiencesPolya(5,5,1000)` produit une figure très différente.



Moins inattendue peut-être, voici la figure obtenue en partant d'une distribution non équitable : l'appel `experiencesPolya(1,5,1000)` produit la figure suivante.



Réfraction d'un rayon lumineux

Cette situation peut donner lieu à des activités en classe de seconde.

On considère deux milieux d'indices de réfraction n_1 et n_2, un point $A(x_1, y_1)$ dans le premier milieu, un point $B(x_2, y_2)$ dans le deuxième. Les deux milieux sont séparés par une courbe dont on connaît une équation de la forme $y = f(x)$.

On cherche le point $M(x, f(x))$ de la séparatrice par lequel passe un rayon lumineux allant de A à B. Le principe de Fermat indique qu'il réalise le minimum (local) du chemin optique

$$\varphi(x) = n_1 \times AM + n_2 \times BM$$

Considérons une fonction φ qu'on va supposer pour simplifier continue et unimodale sur un intervalle $[a, b]$: elle atteint un minimum à l'abscisse c, elle décroît strictement sur $[a, c]$ et croît strictement sur $[c, b]$.

On se donne une abscisse x_0 de l'intervalle $[a, b]$ et une précision ε. On cherche à déterminer une valeur approchée à ε près de l'abscisse c du minimum de φ.

On choisit un pas $h > 0$ et on évalue $\varphi(x_0 - h)$. Si cette valeur est inférieure à $\varphi(x_0)$, on est sûr que $c < x_0$ et on peut recommencer en remplaçant x_0 par $x_0 - h$. On arrive finalement, à force de reculer, à une valeur x_0 telle que $\varphi(x_0 - h) > \varphi(x_0)$. On sait alors que $x_0 - h < c \leq x_0$.

Si on a pu se déplacer vers la gauche, on a atteint une valeur approchée de c à h près.

Sinon, il faut aller vers la droite en comparant $\varphi(x_0)$ à $\varphi(x_0 + h)$.

Remarquons que pour une valeur fixée de h, on se déplace soit vers la gauche, soit vers la droite.

Pour garantir une précision ε, il suffit d'itérer le procédé en remplaçant h par $\frac{h}{10}$ jusqu'à ce qu'on ait bien $h \leq \varepsilon$.

Voici le programme correspondant.

```
def chercheMinimum(phi,x,h,precision=0.00001):
    while h > precision:
        while phi(x-h) < phi(x):
            x = x - h
        while phi(x+h) < phi(x):
            x = x + h
        h = h/10
    return x
```

Le déroulement de cet algorithme est tel que, pour chaque valeur de h, une seule des deux boucles `while` intérieures est réellement active.

On a utilisé une valeur par défaut de la précision requise.

Le calcul du point de la séparatrice par lequel passe le rayon réfracté est maintenant facile à écrire.

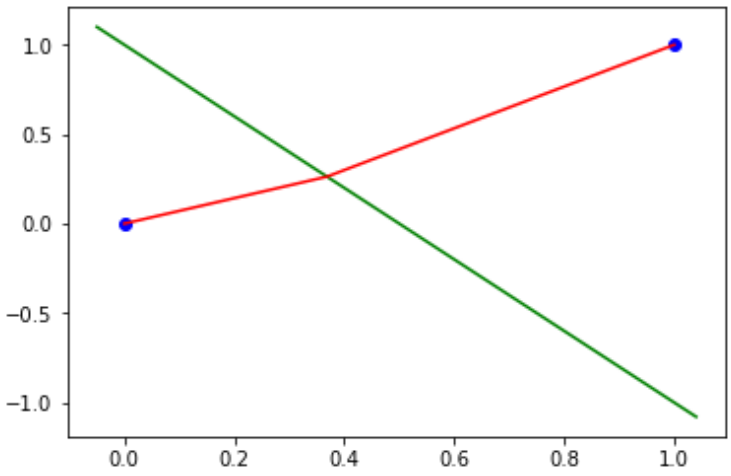
```
def distance(x1,y1,x2,y2):
    return sqrt((x1-x2)**2+(y1-y2)**2)

def refraction(f,x1,y1,x2,y2,n1,n2):
    assert(f(x1)>y1 and f(x2)<y2)
phi = lambda x:
        n1*distance(x1,y1,x,f(x))+n2*distance(x,f(x),x2,y2)
    x = chercheMinimum(phi,(x1+x2)/2,0.1)
    return (x,f(x))
```

Testons différents types de séparatrice, en faisant un dessin.

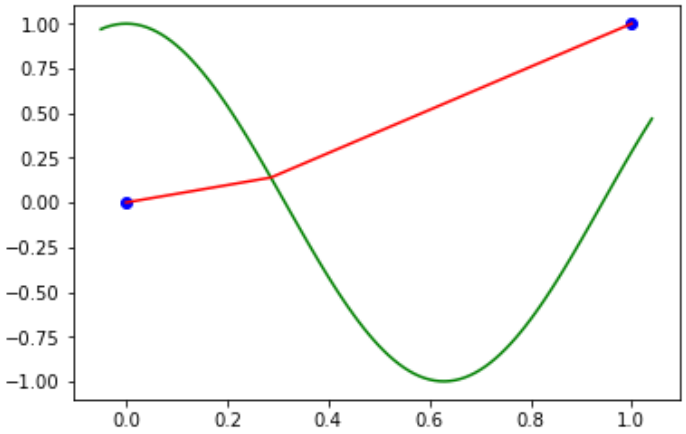
Envisageons d'abord le cas d'une séparatrice rectiligne.

```
separatrice = lambda x: 1-2*x
# on trace les points de départ et d'arrivée
plt.plot([0,1],[0,1],'bo') # gros points bleus
# tracé de la séparatrice elle-même
vx = [t/100 for t in range(-5,105)]
plt.plot(vx,[separatrice(t) for t in vx],'g-') # en vert (green)
# on trace ensuite le rayon lumineux (ligne brisée)
(x,y) = refraction(separatrice, 0,0, 1,1, 2.5,1)
plt.plot([0,x,1],[0,y,1],'r-') # en rouge (red)
plt.show()
```



Passons maintenant à un autre type de séparatrice.

```
separatrice = lambda x: cos(5*x)
plt.plot([0,1],[0,1],'bo')
vx = [t/100 for t in range(-5,105)]
plt.plot(vx,[separatrice(t) for t in vx],'g-')
(x,y) = refraction(separatrice, 0,0, 1,1, 2.5,1)
plt.plot([0,x,1],[0,y,1],'r-')
plt.show()
```



Recherche d'un minimum par une autre méthode

Cette situation peut donner lieu à des activités en classe de seconde.

Considérons une fonction φ continue et unimodale sur un intervalle $[\alpha, \beta]$, qui atteint un minimum à l'abscisse γ, décroît strictement sur $[\alpha, \gamma]$ et croît strictement sur $[\gamma, \beta]$. On cherche à déterminer γ à une précision ε fixée.

L'idée de l'algorithme est la suivante : on suppose donné un triplet (a, c, b) qui vérifie les trois conditions suivantes :

$$a < c < b$$
$$\varphi(a) > \varphi(c)$$
$$\varphi(c) < \varphi(b)$$

Dans ces conditions on est certain que $\gamma \in]a, b[$. On va remplacer le triplet (a, c, b) par un nouveau triplet (a', c', b') qui vérifie également les trois conditions ci-dessus et tel que $b' - a' < b - a$. En itérant suffisamment le procédé, on espère avoir $b - a < \varepsilon$ et ainsi obtenir une valeur approchée satisfaisante de γ.

Soit $u = \frac{a+c}{2}$ et $v = \frac{c+b}{2}$.

Cas 1 : si $\varphi(a) > \varphi(u)$ et $\varphi(u) < \varphi(c)$, on peut poser $(a', c', b') = (a, u, c)$.

La fonction étant unimodale, on ne peut pas avoir $\varphi(a) < \varphi(u)$. Donc dans le cas contraire du cas 1, on a $\varphi(a) > \varphi(u) > \varphi(c)$.

Cas 2 : si $\varphi(c) > \varphi(v)$ et $\varphi(v) < \varphi(b)$, on peut poser $(a', c', b') = (c, v, b)$.

Cas 3 : si $\varphi(u) > \varphi(c)$ et $\varphi(c) < \varphi(v)$, on peut poser $(a', c', b') = (u, c, v)$.

On vérifiera que l'on se retrouve nécessairement dans l'un ou l'autre de ces trois cas.

La programmation en Python ne pose pas de problème particulier : ici, c'est l'étude mathématique systématique des différents cas qui est difficile.

```
def chercheMinimum(phi,a,c,b,epsilon=0.0001):
    assert(phi(a)>phi(c) and phi(c)<phi(b) and a<c<b)
    while b-a > epsilon:
        u = (a+c)/2
        v = (c+b)/2
        if phi(a)>phi(u) and phi(u)<phi(c):
            a,c,b = a,u,c
        elif phi(u)>phi(c) and phi(c)<phi(v):
            a,c,b = u,c,v
        elif phi(c)>phi(v) and phi(v)<phi(b):
            a,c,b = c,v,b
    return c
```

L'appel `chercheMinimum(cos,2,3,4)` renvoie 3.1416015625 (précision par défaut de 10^{-4}).

L'appel `chercheMinimum(cos,2,3,4,0.0000001)` renvoie 3.141592651605606.

Détermination des coordonnées des sommets d'un triangle dont on connaît les longueurs des trois côtés

Cette situation peut donner lieu à des activités en classe de seconde.

On se donne trois longueurs a, b, c vérifiant les conditions $a + b > c$, $b + c > a$ et $c + a > b$.

On sait qu'il existe des triangles ABC tels que les longueurs de ses côtés soient $a = BC$, $b = CA$ et $c = AB$. Il est facile de construire un tel triangle à l'aide de GeoGebra, mais il est plus difficile de déterminer les coordonnées des trois sommets.

On conviendra que, dans un repère bien choisi, A est de coordonnées $(0,0)$, B est de coordonnées $(c, 0)$, et C est de coordonnées (x, y), avec $y > 0$. Il s'agit d'écrire une fonction qui renvoie le couple de coordonnées (x, y) pour les arguments (a, b, c) pour une précision ε fixée.

On va procéder par améliorations successives.

On renverra les coordonnées du point candidat C quand on aura obtenu

$$|AC - b| + |BC - a| \leq \varepsilon.$$

On part d'un point C placé un peu au hasard (en fait on a choisi ici de le placer de sorte que ABC soit un triangle équilatéral), et à chaque étape de l'algorithme, on va rapprocher le point de sa position idéale.

On commence par calculer la longueur $\ell = AC$. Sur la demi-droite $[AC[$ on peut placer le point D tel que $AD = b$, la distance désirée. On déplace alors le point C vers le point D, par exemple en prenant l'image de C par l'homothétie de centre D et de rapport $\frac{9}{10}$ (on pourrait choisir un autre rapport dans l'intervalle $]0,1[$) obtenant ainsi une nouvelle position C'.

On fait alors de même de l'autre côté, en posant $\ell' = BC'$, en choisissant E sur la demi-droite $[BC'[$ tel que $BE = a$. On déplace alors le point C' vers le point E, en prenant son image par l'homothétie de centre E et de rapport $\frac{9}{10}$ obtenant ainsi une nouvelle position C''.

On itère le procédé jusqu'à obtenir la précision requise.

On écrit tout d'abord une fonction qui renvoie les coordonnées de C' en fonction des coordonnées de A, des coordonnées de C et de la distance b. La même fonction permettra de déterminer les coordonnées de C'' en utilisant les arguments correspondants.

On a : $\overrightarrow{AD} = \frac{b}{\ell}\overrightarrow{AC}$ et $\overrightarrow{DC'} = \frac{9}{10}\overrightarrow{DC} = \frac{9}{10}\left(1 - \frac{b}{\ell}\right)\overrightarrow{AC}$ donc finalement $\overrightarrow{AC'} = \frac{1}{10}\left(9 + \frac{b}{\ell}\right)\overrightarrow{AC}$.

Ces formules se traduisent immédiatement dans le programme Python ci-dessous.

```
from math import *
import matplotlib.pyplot as plt

def distance(x1,y1,x2,y2):
    return sqrt((x1-x2)**2 + (y1-y2)**2)

def deplacement(xA,yA,xC,yC,b):
    l = distance(xA,yA,xC,yC)
    k = (9+b/l)/10
    x = xA + k * (xC-xA)
    y = yA + k * (yC-yA)
    return (x,y)
```

On écrit alors la fonction qui renvoie la position des trois points A, B, C pour une précision donnée.

```
def troisLongueurs(a,b,c,epsilon):
    assert(a+b>c and b+c>a and c+a>b)
    xA,yA = 0,0
    xB,yB = c,0
    # position de départ de C
    xC,yC = c/2,c*sqrt(3)/2
    while abs(distance(xA,yA,xC,yC) - b) +
          abs(distance(xB,yB,xC,yC) - a) > epsilon:
        xC,yC = deplacement(xA,yA,xC,yC,b)
        xC,yC = deplacement(xB,yB,xC,yC,a)
    return((xA,yA),(xB,yB),(xC,yC))
```

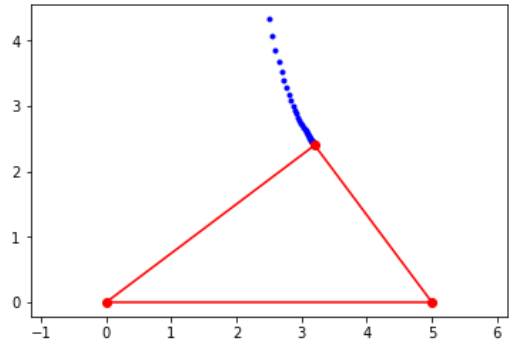
(Remarque la commande `while` doit être écrite sur une seule ligne, on l'a coupée pour des raisons de lisibilité.)

On réécrit cette fonction pour réaliser en même temps la figure et mettre en évidence les améliorations successives de la position du point C.

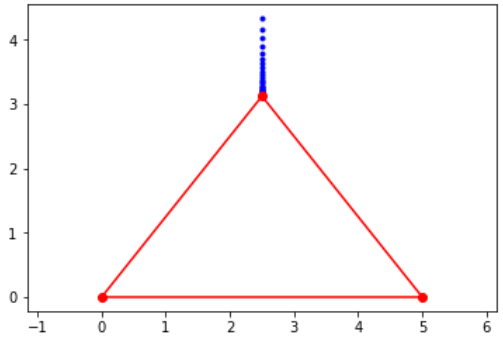
```
def troisLongueursGraphe(a,b,c,epsilon):
    assert(a+b>c and b+c>a and c+a>b)
    xA,yA = 0,0
    xB,yB = c,0
    plt.axis('equal')  # pour utiliser un repère orthonormé
    plt.plot([xA,xB],[yA,yB],'r-') # AB en rouge
    plt.plot([xA],[yA],'ro')   # gros point A rouge
    plt.plot([xB],[yB],'ro')          # gros point B rouge
    # position de départ de C
    xC,yC = c/2,c*sqrt(3)/2
    while abs(distance(xA,yA,xC,yC) - b) +
          abs(distance(xB,yB,xC,yC) - a) > epsilon:
        plt.plot([xC],[yC],'b.')   # point C temporaire bleu
        xC,yC = deplacement(xA,yA,xC,yC,b)
        xC,yC = deplacement(xB,yB,xC,yC,a)
    plt.plot([xA,xC,xB],[yA,yC,yB],'r-')  # AC et CB en rouge
    plt.plot([xC],[yC],'ro')   # gros point C final
    plt.show()
    return((xA,yA),(xB,yB),(xC,yC))
```

La figure suivante montre quelques exécutions de cette fonction (on ne montre que les graphiques produits, mais pas les coordonnées calculées).

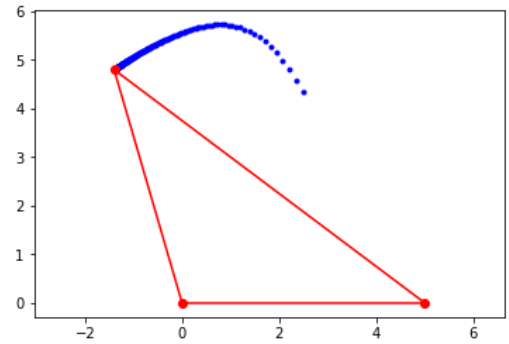
`troisLongueursGraphe(3,4,5,0.00001)`



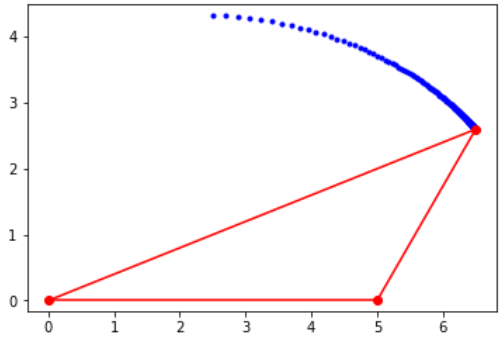
`troisLongueursGraphe(4,4,5,0.00001)`



`troisLongueursGraphe(8,5,5,0.00001)`



`troisLongueursGraphe(3,7,5,0.00001)`



Annexe

Installation de Python 3.6.x

Sur ordinateur

Le plus simple est sans doute d'installer Python 3.6.x depuis le site de référence : l'installation offre un environnement de programmation un peu spartiate mais efficace, nommé IDLE. On peut ajouter les bibliothèques de son choix, comme *matplotlib* par exemple, en tapant dans un terminal la commande `pip install matplotlib`.

On peut préférer la distribution pyzo, qui offre un environnement de programmation plus riche, nommé Conda, et qui préinstalle la plupart des bibliothèques usuelles. Les autres peuvent s'installer en tapant directement dans la console de pyzo la commande `conda install matplotlib` (par exemple).

Une alternative est d'installer Jupyter, qui permet de travailler dans un navigateur internet.

Toutes ces distributions sont gratuites et libres.

Sur une tablette Android

Il existe plusieurs applications disponibles sur la boutique GooglePlay, dont certaines sont gratuites. Une application souvent utilisée est QPython3.

Sur une tablette iOS

Il existe plusieurs applications disponibles sur AppStore, dont certaines sont gratuites.

L'application Pythoni3 est gratuite et répond aux besoins. L'application Pythonista3 coûte une dizaine d'euros mais est certainement une des plus pratiques.