Représentation informatique des objets mathématiques

Emmanuel Beffara **Guillaume François**

Janvier 2018

Résumé

L'informatique ne traite pas les objets idéaux abstraits des mathématiques mais leur représentation symbolique. Modéliser les objets mathématiques, même ceux apparemment les plus simples, impose des contraintes et va parfois contre l'intuition. Cette question de la représentation doit être un pré-requis de l'enseignement de l'informatique. Dans cet atelier, on a abordé cette question en partant du sujet de la représentation des nombres, de la nécessité de travailler avec des valeurs approchées et de ce que cela impose dans la modélisation de tous les problèmes faisant intervenir les nombres réels, qu'il s'agisse de géométrie, de suites numériques ou de fonctions. La question de la représentation donne de nouvelles façons de comprendre les objets mathématiques et suggère des activités enrichissantes tant pour l'enseignement de l'informatique en tant que tel que pour celui des mathématiques. Les participants à cet atelier ont été invités à venir avec leur ordinateur sur lequel un outil de programmation est installé (Scratch, Python, ...). Dans cet article nous donnerons les solutions en Python et en Scratch.

1. Deux exemples introductifs

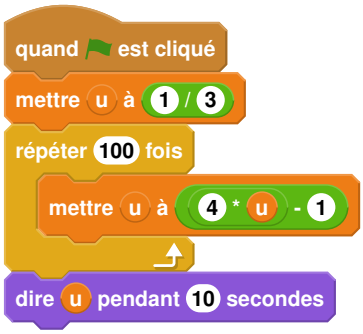
On introduit le sujet de la représentation des nombres au moyen de deux exercices. Le premier prend la forme d'un calcul de suite récurrente :

> **Exercice**
> On veut calculer u_{100} avec (u_n) définie par $u_0 = 1/3$ et $u_{n+1} = 4u_n - 1$ pour tout n.
> Écrire un programme permettant de calculer u_{100}.

Écrire un programme qui calcule le terme voulu consiste simplement à faire répéter la formule de récurrence 100 fois, on ne cherche pas à comprendre le comportement de la suite pour faire un calcul plus intelligent. On obtient les codes suivants en Python et en Scratch :

```python
def recurrence(n):
    u = 1.0/3
    for i in range(n):
        u = 4*u - 1
    return u

print recurrence(100)
```



0	0.333333333333	24	0.328125	48	−1.4660155037 e+12
1	0.333333333333	25	0.3125	49	−5.86406201480 e+12
2	0.333333333333	26	0.25	50	−2.34562480592 e+13
3	0.333333333333	27	0.0	51	−9.38249922369 e+13
4	0.333333333333	28	−1.0	52	−3.75299968948 e+14
5	0.333333333333	29	−5.0	53	−1.50119987579 e+15
6	0.333333333333	30	−21.0	54	−6.00479950316 e+15
7	0.333333333333	31	−85.0	55	−2.40191980126 e+16
8	0.333333333332	32	−341.0	56	−9.60767920506 e+16
9	0.333333333328	33	−1365.0	57	−3.84307168202 e+17
10	0.333333333314	34	−5461.0	58	−1.53722867281 e+18
11	0.333333333256	35	−21845.0	59	−6.14891469124 e+18
12	0.333333333023	36	−87381.0	60	−2.45956587649 e+19
13	0.333333332092	37	−349525.0	61	−9.83826350598 e+19
14	0.333333328366	38	−1398101.0	62	−3.93530540239 e+20
15	0.333333313465	39	−5592405.0	63	−1.57412216096 e+21
16	0.33333325386	40	−22369621.0	64	−6.29648864383 e+21
17	0.333333015442	41	−89478485.0	65	−2.51859545753 e+22
18	0.333332061768	42	−357913941.0	66	−1.00743818301 e+23
19	0.33332824707	43	−1431655765.0	67	−4.02975273205 e+23
20	0.333312988281	44	−5726623061.0	68	−1.61190109282 e+24
21	0.333251953125	45	−22906492245.0	69	−6.44760437128 e+24
22	0.3330078125	46	−91625968981.0	...	...
23	0.33203125	47	−3.66503875925 e+11	100	−2.97343269314 e+43

TABLE 1 – Valeurs calculées pour la suite du premier exemple

À la grande surprise des stagiaires, on trouve $-2,9734326931374163\,\text{e+}43$. Ce résultat est d'autant plus surprennent que, selon la définition donnée, cette suite devrait être constante ! Si nous affichons les cent premières valeurs, nous obtenons les valeurs données dans la table **??**.

Pourquoi le calcul sur machine nous donne-t-il un tel résultat ?

On s'intéresse ensuite à un calcul qui ne fait intervenir aucune forme d'algorithmique :

> **Exercice**
> Faire effectuer le calcul suivant : $0,1 + 0,2$.

On peut calculer ce résultat avec Python de façon interactive, simplement en tapant l'expression dans l'interpréteur. En Scratch, il suffit de faire dire le résultat du calcul :

```
>>> 0.1 + 0.2
0.30000000000000004
>>>
```

Le résultat affiché par Python n'est pas exactement ce que l'on pouvait espérer. Dans le cas de Scratch, le résultat est plus subtil : certaines versions vont dire « 0.3 », mais si on met le résultat du même calcul dans une variable et que l'on interroge ensuite la valeur de la variable, on obtiendra la valeur affichée par Python.

2. Les nombres entiers

Pour expliquer ces phénomènes, il faut s'intéresser de près à la représentation des nombres, en commençant par les nombres entiers, d'abord les naturels puis les relatifs.

2.1. Les bases

Notre numération est décimale, c'est-à-dire que nous utilisons une écriture en base 10, dont l'alphabet est constitué des dix chiffres que tout le monde connaît :

$$\{\ 0,\ 1,\ 2,\ 3,\ 4,\ 5,\ 6,\ 7,\ 8,\ 9\ \}$$

Pour représenter un nombre au-delà de 9, il faut utiliser plusieurs chiffres. En partant de la droite, on a un chiffre des unités, un chiffre des dizaines, un chiffre des centaines, et ainsi de suite, de sorte que la valeur associée à un chiffre dans l'écriture d'un nombre est la valeur du chiffre multipliée par une puissance de 10 qui correspond à sa position. Pour cette raison, on parle donc de *numération de position*.

Dans les ordinateurs, les nombres sont représentés en binaire, c'est-à-dire en base 2, parce que c'est le plus simple à mettre en œuvre techniquement. Tout ce qui change, c'est que l'on a maintenant des puissances de 2 et un alphabet réduit à deux chiffres :

$$\{\ 0,\ 1\ \}$$

Dans le jargon des informaticiens, on parle souvent de « bits » au lieu de « chiffres », c'est équivalent (à ceci près que le mot « bit », contraction de l'anglais « *binary digit* », n'est utilisé que pour la base 2). Par exemple, on a

$$171 = \mathbf{1}\times 2^7 + \mathbf{0}\times 2^6 + \mathbf{1}\times 2^5 + \mathbf{0}\times 2^4 + \mathbf{1}\times 2^3 + \mathbf{0}\times 2^2 + \mathbf{1}\times 2^1 + \mathbf{1}\times 2^0$$

donc le nombre 171 s'écrit « 1010 1011 » en base 2 ; il s'écrit sur 8 bits. Les premiers nombres entiers s'écrivent

$$0,\ 1,\ 10,\ 11,\ 100,\ 101,\ 110,\ 111,\ \ldots$$

À propos des microprocesseurs, on parle souvent d'architecture à 32 bits, 64 bits, etc. Il s'agit simplement de la taille des nombres entiers que le processeur peut traiter le plus efficacement. En effet, on ne peut pas coder tous les nombres entiers si on a un nombre fixé de chiffres : en codant sur n bits, on ne peut représenter que les 2^n premiers entiers, de 0 à $2^n - 1$.

Sur 8 bits, 87 se code : 0101 0111.
Sur 16 bits, 87 se code : 0000 0000 0101 0111.
Sur 4 bits, on ne peut pas coder 87.

Si on veut représenter un entier plus grand que le nombre de bits qu'utilise l'ordinateur, il est nécessaire de le décomposer en blocs, ce qui rend les calculs plus coûteux. Python et Scratch calculent implicitement avec des nombres entiers de taille quelconque mais ce n'est pas le cas dans des langages de plus bas niveau (par exemple C, Java, C# etc) où le type des entiers a effectivement une précision limitée.

2.2. Changements de base

Le changement de base (ou parle aussi de *transcodage* ou de *conversion de base*) est l'opération qui permet de passer de la représentation d'un nombre exprimé dans une base à la représentation du même nombre exprimé dans une autre base.

Pour convertir un nombre entier en binaire, on doit donc le décomposer en une somme de puissances de 2. Pour cela :

— On part du nombre à décomposer.
— On le divise par 2 et on note le reste de la division (c'est soit 1 soit 0).

— On fait la même chose avec le quotient, et on met de nouveau le reste de coté.
— On réitère la division, et ce jusqu'à ce que le quotient soit 0.
— L'écriture en binaire apparaît avec la suite des restes : le premier chiffre est le dernier reste non nul (forcément 1 si le nombre de départ n'était pas 0), on obtient les chiffres suivant en parcourant les restes du dernier jusqu'au premier.

Par exemple, la décomposition du nombre 87 en base 2 donne :

$$\begin{aligned} 87 &= 2 \times 43 + 1 \\ 43 &= 2 \times 21 + 1 \\ 21 &= 2 \times 10 + 1 \\ 10 &= 2 \times \ 5 + 0 \\ 5 &= 2 \times \ 2 + 1 \\ 2 &= 2 \times \ 1 + 0 \\ 1 &= 2 \times \ 0 + 1 \end{aligned}$$

$$87 = \mathbf{1} \times 2^6 + \mathbf{0} \times 2^5 + \mathbf{1} \times 2^4 + \mathbf{0} \times 2^3 + \mathbf{1} \times 2^2 + \mathbf{1} \times 2^1 + \mathbf{1} \times 2^0$$

Ainsi 87 s'écrit 101 0111 en base 2. De façon plus concise, on note

$$\overline{87}^{10} = \overline{1010111}^{2}.$$

Inversement, considérons une écriture en binaire, par exemple $\overline{10011}^{2}$. Pour calculer sa valeur, il suffit de multiplier chaque chiffre par la puissance de 2 correspondant à sa position et d'additionner les résultats. On obtient :

$$\mathbf{1} \times 2^4 + \mathbf{0} \times 2^3 + \mathbf{0} \times 2^2 + \mathbf{1} \times 2^1 + \mathbf{1} \times 2^0 = 19$$

donc

$$\overline{10011}^{2} = \overline{19}^{10}.$$

Programmer ces opérations de conversion est un exercice intéressant (et très classique) d'algorithmique et de programmation.

Exercice
Écrire un programme permettant de convertir des nombres entiers de la base 10 à la base 2 et inversement.

Pour mettre en œuvre les conversions dans les deux sens, et en fait de n'importe quelle base vers n'importe quelle autre base, il suffit d'appliquer les méthodes décrites plus haut, en remplaçant 2 par la base choisie. Ainsi, pour décomposer un nombre en base b, on calcule la suite des restes successifs par division par b ; l'opération prend en entrée un nombre entier et donne en sortie une liste de chiffres. Pour recomposer un nombre à partir de son écriture en base b on multiplie les chiffres par les puissances successives de b, l'opération prend en entrée une liste de chiffres et donne en sortie un nombre entier. On donne ici une solution possible en Python, Scratch est moins adapté pour ce genre de calcul mais permettrait de le faire en utilisant les variables de listes.

```
def decompose(nombre, base):
    chiffres = []
    while nombre != 0:
        chiffres.append(nombre % base)
        nombre = nombre // base
    chiffres.reverse()
    return chiffres

def compose(chiffres, base):
    nombre = 0
    for c in chiffres:
        nombre = nombre * base + c
    return nombre

def change_base(chiffres, base1, base2):
    return decompose(compose(chiffres,
       base1), base2)
```

Ici, une remarque s'impose : on pourrait croire que `decompose(n,b)` suffit à convertir `n` de la base 10 à la base `b` et que `compose(L,b)` suffit à convertir la liste de chiffres `L` de la base `b` à la base 10 :

```
>>> decompose(87, 2)
[1, 0, 1, 0, 1, 1, 1]
>>> compose([1, 0, 0, 1, 1], 2)
19
```

Il est important de comprendre que ce n'est pas le cas. En effet, une conversion avec la base 10 a bien eu lieu, mais ce n'est pas notre programme qui l'a faite. Dans le premier cas, c'est le fait d'écrire `87` qui a déclenché une conversion, faite par l'interpréteur Python avant d'exécuter notre fonction. Dans le deuxième cas, notre fonction produit un nombre entier et c'est à nouveau l'interpréteur Python qui la convertit en base 10 pour pouvoir l'afficher. Les entiers utilisés par la machine n'ont rien de décimal, la seule particularité de la base 10 est que c'est celle que nous utilisons pour parler et pour écrire, parce que nous avons 10 doigts pour compter !

2.3. Les opérations sur les entiers naturels

Les opérations de changement de base dans l'exercice précédent urtilisent les opérations arithmétiques sur les entiers naturels : addition, multiplication, division euclidienne. Comment ces opérations sont-elles effectivement programmées ? Il suffit de poser les opérations, comme on apprend à le faire à l'école primaire, mais en le faisant en base 2.

Par exemple, on a $\overline{150}^{10} = \overline{1001\,0110}^{2}$ et $\overline{85}^{2} = \overline{101\,0101}^{2}$. En posant l'addition en binaire de ces deux nombres, on a :

$$\begin{array}{cccccccccc} & & & 1 & & 1 & & & & \leftarrow \text{retenues} \\ & & 1 & 0 & 0 & 1 & 0 & 1 & 1 & 0 \\ + & & 0 & 1 & 0 & 1 & 0 & 1 & 0 & 1 \\ \hline & & 1 & 1 & 1 & 0 & 1 & 0 & 1 & 1 \end{array}$$

Ici le calcul des retenues se fait en base 2 : si on additionne 1 et 1, comme en base 2 on a $\overline{1}^2 + \overline{1}^2 = \overline{10}^2$, il faut poser 0 et retenir 1.

Dans le cas où il reste une retenue après le dernier chiffre, le résultat s'écrit sur un chiffre de plus que les nombres dont on est parti :

$$\begin{array}{cccccccccc} & 1 & 0 & 0 & 1 & 0 & 1 & 1 & 0 \\ + & 0 & 1 & 1 & 1 & 0 & 1 & 0 & 1 \\ \hline \mathbf{1} & 0 & 0 & 0 & 0 & 1 & 0 & 1 & 1 \end{array}$$

Ici on a deux entiers qui peuvent s'écrire sur 8 bits ($\overline{150}^{10} = \overline{1001\,0110}^2$ et $\overline{117}^{10} = \overline{111\,0101}^2$) mais leur somme ($\overline{267}^{10} = \overline{1\,0000\,1011}^2$) nécessite 9 bits. Quand on utilise un langage (ou un type de nombre) qui a une précision limitée, cela implique qu'on n'obtiendra pas le résultat attendu : en calculant sur 8 bits, on ne gardera que les 8 derniers chiffres et le résultat ne sera pas 267 mais $\overline{0000\,1011}^2$, c'est-à-dire $\overline{11}^{10}$. Dans ce cas on parle de phénomène de *débordement* (en anglais *overflow*), cela peut être source d'erreurs quand on n'en a pas conscience.

2.4. Codage des nombres entiers relatifs

L'écriture dans une base permet de représenter les nombres entiers naturels, c'est-à-dire positifs ou nuls. Pour représenter les entiers relatifs, il faut prendre en compre le signe. En pratique, il y a princialement deux façons de procéder : utiliser un *bit de signe* ou représenter en *complément à* 2.

Dans les deux cas, le bit le plus à gauche est utilisé pour représenter le signe du nombre, en utilisant 0 pour le positif et 1 pour le négatif. La différence se situe dans la représentation de la valeur absolue :

En binaire signé, les autres bits codent la valeur absolue du nombre. Un petit inconvénient est qu'il y a deux représentations de zéro : +0 et −0.

En complément à 2, on représente le nombre relatif modulo 2^n, s'il y a n bits (donc en incluant le bit de signe) :
- un nombre positif est codé de la même façon qu'en binaire pur,
- un nombre négatif a est codé par la représentation de $a + 2^n$.

Quelques éléments à titre de comparaison :

	bit de signe	complément à 2
codage de −87 sur 8 bits	1010 1001	1101 0111
codage de −1 sur 8 bits	1000 0001	1111 1111
entier minimum sur n bits	$-(2^n - 1)$	-2^n
entier maximum sur n bits	$2^n - 1$	$2^n - 1$

Chaque représentation a ses avantages et inconvénients. En général, quand on travaille avec un nombre de bits fixé (comme avec les entiers de base du processeur), c'est le complément à 2 qui est utilisé, parce que les opérations sont faciles à mettre en œuvre : comme tout se passe modulo 2^n, on peut calculer les additions, soustractions et multiplications sans se préoccuper de signe. La représentation avec bit de signe a des opérations un peu plus complexes mais est plus adaptée quand le nombre de bits utilisé est variable (comme c'est le cas avec les entiers illimités de Python et Scratch).

	simple précision	double précision
taille totale	32 bits	64 bits
dont signe	1 bit	1 bit
dont exposant	8 bits	11 bits
dont mantisse	23 bits	52 bits
minimum dénormalisé	2^{-149}	2^{-1074}
soit environ	$1,401298 \times 10^{-45}$	$4,940656 \times 10^{-324}$
minimum normalisé	2^{-126}	2^{-1022}
soit environ	$1,175494 \times 10^{-38}$	$2,225074 \times 10^{-308}$
maximum	$2^{128} - 2^{105}$	$2^{1024} - 2^{972}$
soit environ	$3,402823 \times 10^{+38}$	$1,797693 \times 10^{+308}$
précision de la mantisse	2^{-23}	2^{-52}
soit environ	$1,192093 \times 10^{-7}$	$2,220446 \times 10^{-16}$

TABLE 2 – Précision des nombres à virgule flottante dans la norme IEEE 754

Techniquement, la représentation d'un nombre négatif en complément à deux revient à inverser chaque bit de la représentation de sa valeur absolue puis à ajouter 1 au résultat. Avec n bits, on code tous les nombres entre -2^{n-1} et $2^{n-1} - 1$.

3. Les nombres à virgue flottante

La virgule flottante est la méthode la plus courante pour représenter de nombres pas forcément entiers. Il s'agit en fait d'une variation sur ce que l'on appelle habituellement la *notation scientifique*. Par exemple, en notation scientifique, le nombre $293752,393$ s'écrit $2,93752393 \times 10^5$ et le nombre $0,0004511$ s'écrit $4,511 \times 10^{-4}$. Plus généralement, on écrit un nombre x sous forme

$$x = \pm m \times b^e$$

où

- $\pm$ est le signe,
- m est la *mantisse* (un nombre de l'intervalle $[1, b[$),
- b est la *base* (un entier positif, 10 pour la notation scientifique usuelle, 2 pour une représentation en binaire),
- e est l'*exposant* (un entier relatif).

La mantisse contient donc les chiffres *significatifs*, c'est un nombre qui s'écrit avec un seul chiffre à gauche de la virgule.

Pour choisir une représentation informatique des nombres, il faut fixer une précision pour la mantisse (le nombre de chiffres significatifs) et une précision pour l'exposant. Le standard IEEE 754 définit deux précisions qui permettent un codage sur 32 et 64 bits (et qui correspondent respectivement aux types `float` et `double` en C), détaillés en table 2. De nos jours, vue la puissance du matériel, le simple précision n'est quasiment plus utilisée et c'est la *double précision* qui est utilisée par défaut, c'est notamment ce qu'emploient Scratch et Python.

Pour des raisons techniques, on emploie des conventions particulières :

— l'exposant n'est pas en complément à 2 mais décalé de 127 (en simple précision) ou 1023 (en double précision), ce qui permet de comparer les exposants comme s'il s'agissait de nombres entiers non signés ;
— pour la mantisse, on ne note que les chiffres après la virgule, puisque le chiffre avant la virgule est forcément un 1 si on est en base 2.

À celà s'ajoutent quelques raffinements, par exemple un exposant de 0 (qui correspondrait à -127 ou -1023 avec le décalage) est utilisé pour représenter le nombre 0 est des nombres très proches de 0 (dits *dénormalisés*) ; un exposant de 255 (respectivement 2047) est utilisé pour représenter les infinis et des valeurs non définies (dans le jargon, *not a number* ou *NaN*). En dehors de ces cas particuliers, s'il y a par exemple 23 bits de mantisse, il y a 23 chiffres significatifs en binaire, ce qui correspond à une précision de l'ordre $1,2 \times 10^{-7}$, soit environ 7 chiffres significatifs en décimal.

Par exemple, en simple précision, considérons le code suivant :

$$\underbrace{0}_{s}\ \underbrace{10000001}_{e}\ \underbrace{11100000000000000000000}_{m}$$

On a donc :

— Signe : 0, donc le nombre est positif.
— Exposant : $\overline{10000001}^2 = 129$ donc $e = 129 - 127 = 2$.
— Mantisse : $m = \overline{1,11100000000000000000000}^2 = 1,875$

donc ce code représente le nombre qui, en décimal, s'écrit

$$+1,875 \times 2^2 = 7,5.$$

Le nombre décimal $45,75$, si on le décompose en somme de puissances de 2, s'écrit

$$\begin{aligned} 45,75 &= 2^5 + 2^3 + 2^2 + 2^0 + 2^{-1} + 2^{-2} \\ &= \overline{101101,11}^2 = \overline{1,0110111}^2 \times 2^5 \end{aligned}$$

donc on a la décomposition suivante :

— Signe : 0 car le nombre est positif,
— Exposant : $e = 5$, codé par $5 + 127 = 132 = \overline{10000100}^2$,
— Mantisse : $\overline{01101110000000000000000}^2$,

donc le code de $45,75$ en virgule flottante qimple précision est

$$0\ 10000100\ 01101110000000000000000$$

Bien sûr, tous les nombres ne peuvent pas s'écrire sous cette forme. En d'autres termes, *il y a énormément de trous*. Par exemple, l'ensemble des nombres positifs à virgule flottante sur 8 bits (avec 3 d'exposant et 4 de mantisse) est réparti comme ceci sur la droite réelle :

Exercice
Voici un calcul fait par Python :

```
>>> 0.1 + 0.2
0.30000000000000004
>>>
```

En utilisant le codage en virgule flottante, expliquer le résultat donné.

Nous avons $\overline{0,1}^{10} \approx \overline{0,00011011001100110011...}^{2}$. Le nombre décimal 0, 1 n'a pas d'écriture exacte en binaire, comme de nombreux autres nombres. C'est exactement le même phénomène que pour l'écriture $1/3 = 0,333333333\ldots$ en décimal : il n'y a pas d'écriture exacte.

> Le calcul en virgule flottante impose de faire du calcul approché, donc *l'égalité n'a pas de sens* pour ces nombres !

4. Les objets géométriques

Quand on veut programmer avec des objets géométriques, il faut en choisir une représentation. En général, cela revient à choisir des paramètres qui permettent de les définir. Par exemple :

- un point du plan est représenté par 2 nombres, ses coordonnées dans un repère fixé par convention ;
- un segment est représenté par 2 points du plan, c'est-à-dire par 4 nombres ;
- un cercle peut être représenté de différentes façons : un point et un rayon (donc 3 nombres), 3 points non alignés (donc 6 nombres), un diamètre (donc un segment, c'est-à-dire 4 nombres), etc.

De même, on peut se demander quelle pourrait être une représentation pertinente pour une droite, un triangle, une parabole, etc.

On observe donc qu'un objet donné peut être paramétré de plusieurs façons. Ainsi, on peut désigner une droite du plan de différentes façons qui correspondent à différents choix de représentation :

- la droite passant par $(2, 1)$ et $(3, 4)$
- la droite de pente 3 passant par $(4, 7)$
- la droite d'équation $y = 3x - 5$

Pour un choix de représentation donné, il peut encore y avoir plusieurs choix possibles des paramètres qui donnent le même objet :

- la droite passant par $(2, 1)$ et $(3, 4)$
- la droite passant par $(1, -2)$ et $(5, 10)$
- la droite passant par $(3, 4)$ et $(2, 1)$

Reconnaître que deux représentations désignent le même objet n'est pas toujours évident. On avait aperçu ce phénomène dans la représentation des entiers avec un bit de signe : il y avait $+0$ et -0 qui désignent en fait le même entier ; on pouvait régler cette ambiguïté en posant une convention simple. Quand les objets sont plus complexes, la situation se reproduit et il n'est pas toujours facile (ni même possible) de poser une convention simple. Reconnaître si deux représentations désignent le même objet est alors un calcul qui nécessite une compréhension mathématique des objets.

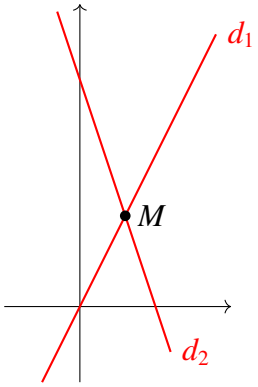
Indépendamment de cette question du choix des paramètres, on retrouve en géométrie la question de l'approximation. En effet, la géométrie du plan est basée sur les nombres réels

mais ceux-ci ne peuvent pas être utilisés directement, il faut aussi les représenter. En général on utilise des nombres à virgule flottante, donc :

— l'ensemble des valeurs possibles est fini ;
— selon le choix du repère, on ne représente pas les mêmes points ;
— les objets géométriques sont approchés.

L'emploi nécessaire d'approximations implique effectivement que le choix d'un repère du plan change l'ensemble des points qu'il est possible de représenter de façon exacte.

Exercice



Calculer le point d'intersection des droites d_1 passant par $(0,0)$ et $(1,2)$ et d_2 passant par $(0,3)$ et $(1,0)$.

$$d_1 : y = 2x$$
$$d_2 : y = 3 - 3x$$

Vérifier le résultat avec Python et Geogebra.

L'objet de cet exercice n'est pas de trouver des équations des droites (c'est pourquoi on peut même les donner directement), ni même de résoudre mathématiquement le système d'équations obtenu : il est très facile de voir que la solution est le point de coordonnées $(3/5; 6/5)$, soit $(0,6; 1,2)$. Le point important est la vérification par le calcul en utilisant les équations, ainsi en Python on obtient :

```
>>> x, y = 0.6, 1.2
>>> y == 2 * x
True
>>> y == 3 - 3 * x
False
```

ce qui laisse penser que le point obtenu ne vérifie pas l'équation de la droite (d_2), donc qu'il ne se trouve pas dessus. En réalité, on a à nouveau affaire à un problème d'arrondi :

```
>>> 3 - 3 * x - y
2.220446049250313e-16
```

On peut remarquer qu'il s'agit exactement de la précision de la mantisse dans les nombres à virgule flottante de 64 bits : l'arrondi ne porte que sur le dernier chiffre significatif en binaire. Il s'agit en fait d'une simple variation sur l'exercice introductif sur $0,1 + 0,2$, reformulé dans un contexte géométrique.

Exercice

Voici l'exercice 15 p 335 du manuel *Hyperbole 2de - conforme aux aménagements de programme 2017* :

Qu'en pensez-vous ?

L'exemple du *testeur de Pythagore* est un grand classique des nouveaux programmes. Au vu des remarques précédentes, il devrait être clair que ce genre d'exemple est à prendre avec

précaution : du fait des erreurs d'arrondi, il y aura facilement des *faux négatifs* si l'on fait le test par égalité. Ainsi, en Python, considérons la fonction suivante :

```
def H(a, b, c):
    if a > b and a > c:
        a, c = c, a
    elif b > a and b > c:
        b, c = c, b
    return a * a + b * b == c * c
```

La manipulation des quatre premières lignes consiste à s'assurer que `c` est la plus grande des trois valeurs données, donc la longueur du côté qui serait l'hypothénuse. Avec cette fonction, on pourra avoir de mauvais résultats :

```
>>> H(5, 12, 13)
True
>>> H(0.5, 1.2, 1.3)
False
```

Il est probable que les outils de géométrie dynamique donneront moins de faux négatifs parce qu'ils testent les égalités avec un certaine tolérance, *à ε près*. Cela permet d'obtenir les résultats attendus dans les cas courants pour ce genre d'exemple mais la contrepartie est qu'il y aura de faux positifs, pour des triangles « presque » rectangles.

Raisonner à ε près ne résout pourtant pas le problème de fond du calcul approché : cela permet de déterminer quand des quantités sont à peu près égales, mais en déduire une égalité stricte n'est pas pertinent sans avoir plus de précision sur le problème considéré, et il faut choisir ε de façon adaptée à la situation. Dans le cas du testeur de Pythagore, sur des nombres courants entrés à la main dans le programme, on peut prédire qu'une différence très petite par rapport à 1 sera forcément une erreur d'arrondi :

```
def H(a, b, c):
    if a > b and a > c:
        a, c = c, a
    elif b > a and b > c:
        b, c = c, b
    return abs(a * a + b * b - c * c) < 1e-10
```

Obtenir un faux négatif ou un faux positif avec cette version est bien sûr possible mais il faut employer des valeurs qui ne correspondent pas aux ordres de grandeur attendus :

```
>>> H(3, 4, 5)
True
>>> H(3e200, 4e200, 5e200)
False
>>> H(3, 4, 6)
False
>>> H(3e-200, 4e-200, 6e-200)
True
```

Il est utile ici d'insister sur le sens algorithmique de cet exercice et de la solution naturelle. L'*algorithme* naturel avec un test par égalité est parfaitement correct si l'on considère que les calculs peuvent être faits sans approximation. C'est le passage au *programme* utilisant la virgule flottante qui ne satisfait pas cette supposition et rend donc le programme incorrect par rapport

à la question posée. Utiliser ce programme avec des données exactes (comme des entiers ou des rationnels sans limite de taille) ne poserait pas le même problème. Cependant, utiliser ce programme avec uniquement des nombres entiers revient plutôt à identifier des *triplets pythagoriciens* qu'à tester si des triangles sont rectangles.

En annexe nous vous proposons une façon de réécrire cet énoncé sous cette forme.

5. Les fonctions

En mathématiques, une fonction f est simplement la donnée d'un domaine de définition D et, pour chaque élément $x \in D$, d'une image $f(x)$ qui peut être définie de nombreuses façons différentes :

— par une formule explicite,
— par une propriété caractéristique (équation, équation différentielle),
— par un tableau de valeurs (mais que vaut la fonction en dehors des valeurs données ?),
— par une courbe (mais comment connaître les valeurs exactement ?).

En informatique, une fonction est uniquement un *procédé de calcul*, qui produit de façon effective une *représentation* d'une image pour chaque *représentation* d'un antécédent.

```
def f(x):
    y = 0
    for n in range(10):
        y = y + x**n / n
    return y
```

On ne peut utiliser une fonction qu'en l'évaluant en différents points.

Notons que dans le cadre du calcul formel, on peut manipuler une fonction dans sa globalité, en manipulant une formule qui la définit, mais c'est un tout autre sujet : cela ne correspond pas à la notion habituelle de fonction au sens informatique.

> **Exercice**
> Le théorème des valeurs intermédiaire affirme :
>
> > Soit f une fonction définie, continue et strictement croissante sur un intervalle $[a;b]$ tel que $f(a) \leq 0$ et $f(b) \geq 0$. Alors l'équation $f(x) = 0$ a une unique solution dans l'intervalle $[a;b]$.
>
> Écrire un programme Python qui trouve cette solution.

Il s'agit à nouveau d'un grand classique : la recherche de zéro d'une fonction par dichotomie, en utilisant le théorème des valeurs intermédiaires.

L'algorithme consiste à reproduire la démarche de la démonstration du théorème : répéter l'opération consistant à considérer le milieu c de l'intervalle $[a;b]$, évaluer la fonction en c puis, selon le signe de $f(c)$, recommencer avec l'intervalle $[a;c]$ ou l'intervalle $[c;b]$. Il est nécessaire de fixer une condition d'arrêt, puisqu'il n'y a aucune raison que la suite des valeurs de c atteigne le zéro de la fonction f. Un choix naturel consiste à nouveau à raisonner « à ε près » :

```
def dichotomie(f, a, b):
    # on suppose f(a)<0 et f(b)>0
    while True:
        c = (a + b) / 2
        fc = f(c)
        if fc == 0:
```

```
        return c
    elif fc > 0:
        b = c
    else:
        a = c
    if b - a < epsilon
        return c
```

On obtiendra ainsi une approximation de la solution à ε près (à condition de remplacer `epsilon` par une valeur explicite). Reste bien sûr à choisir une valeur pertinente pour ε, ce qui n'a rien d'évident. Une variation, si on calcule en virgule flottante, consiste à exploiter le fait que la précision est finie, en remplaçant les deux dernières lignes par la condition suivante :

```
if c == a or c == b:
    return c
```

En effet, à force de réduire l'intervalle $[a; b]$, les bornes étant des nombres à virgule flottante de précision finie et fixée, on obtiendra en fin de compte deux nombres consécutifs. Cet variante donnera donc une approximation dont la précision est celle du type utilisé.

Annexe

Exercice

On donne l'algorithme suivant :

...
Saisir dans l'ordre croissant trois nombres entiers A, B, C
Affecter à X la valeur de $A^2 + B^2$
Affecter à Y la valeur de C^2
Si $X = Y$
  Alors
    Afficher ...
  Sinon
    Afficher ...
Fin Si

1. Compléter les trois lignes incomplètes de l'algorithme précédent.
2. (a) Calculer les valeurs de X et Y pour $A = 7$, $B = 9$ et $C = 12$.
   (b) Quel est le résultat affiché à la sortie de l'algorithme dans ce cas ?
   (c) Écrire le script en Python correspondant à cet algorithme, et le tester.
3. Donner d'autres valeurs de A, B et C qui satisfont le test de sortie de l'algorithme.

Objectif

— Mettre en évidence le problème de comparaison des flottants.

Pré-requis

— La structure conditionnelle.

— Les différentes façons de lire une donnée :
`A=input("A=")`, `A=int(input("A="))` et `A=float(input("A="))`.

Analyse a priori

Une réponse attendue pour la première question est :

Ce programme sert à savoir si un triangle est rectangle ou non
Si $X = Y$
    Alors
        Afficher "Ce triangle est rectangle"
    Sinon
        Afficher "Ce triangle n'est pas rectangle"
Fin Si

L'enseignant questionnera les élèves sur le fait que l'on a choisi des nombres entiers pour A, B et C. Le but du dialogue qui s'établira dans la classe est de se questionner sur la différence entre les nombres décimaux et les flottants, et de faire évoluer leurs réponses vers la réponse suivante :

Ce programme sert à savoir si l'on a un triplet pythagoricien ou non
Si $X = Y$
    Alors
        Afficher "Ce triplet est pythagoricien."
    Sinon
        Afficher "Ce triplet n'est pas pythagoricien."
Fin Si

Analyse a posteriori

Travail sur les fonctions

La plupart des élèves ont écrit le script suivant à quelques petites erreurs près :

```
# Ce programme sert à savoir si un triangle est rectangle ou non.
print("Vous allez saisir trois nombres entiers A, B et C dans
   l'ordre croissant")
A=int(input("A="))
B=int(input("B="))
C=int(input("C="))
X=A**2+B**2
Y=C**2
if X==Y:
    print("Le triangle est rectangle")
else:
    print("Le triangle n'est pas rectangle")
```

Une élève a essayé d'utiliser une fonction :

```
def ex8(A,B,C):
    # Cette fonction sert à savoir si un triangle est rectangle.
    print("Vous allez saisir trois nombres entiers A, B et C dans
       l'ordre croissant")
    A=int(input("A="))
    B=int(input("B="))
    C=int(input("C="))
    X=A**2+B**2
    Y=C**2
    if X==Y:
        print("Le triangle est rectangle")
    else:
        print("Le triangle n'est pas rectangle")
    return
```

Nous en avons profité pour préciser qu'en Python, une fonction peut avoir zéro, un ou plusieurs paramètre(s), et peut retourner quelque chose ou pas (pas de `return`).

D'où les deux scripts suivants :

```
def ex8():
    # Cette fonction sert à savoir si un triangle est rectangle.
    print("Vous allez saisir trois nombres entiers A, B et C dans
       l'ordre croissant")
    A=int(input("A="))
    B=int(input("B="))
    C=int(input("C="))
    X=A**2+B**2
    Y=C**2
    if X==Y:
        print("Le triangle est rectangle")
    else:
        print("Le triangle n'est pas rectangle")
```

ET...

```
def ex8():
    # Cette fonction sert à savoir si un triangle est rectangle.
    print("Vous allez saisir trois nombres entiers A, B et C dans
       l'ordre croissant")
    A=int(input("A="))
    B=int(input("B="))
    C=int(input("C="))
    X=A**2+B**2
    Y=C**2
    if X==Y:
        print("Le triangle est rectangle")
    else:
        print("Le triangle n'est pas rectangle")
        Z="Le triangle est rectangle"
    else:
        Z="Le triangle n'est pas rectangle"
    print(Z)
```

Travail sur les flottants

Après que tous les élèves ont fait tourner leur script, je leur ai demandé s'ils pouvaient l'utiliser avec n'importe quel triangle. Certains élèves ont répondu que non, car si les côtés des triangles sont des nombres décimaux, on ne peut pas utiliser le programme. Des élèves ont proposé un changement d'unité pour se ramener à des nombres entiers, d'autres ont proposé de réécrire le programme pour qu'il fonctionne avec des nombres flottants (`A=float(input("A="))`).

Je leur ai alors demandé de taper `0,1+0,2` dans leur fenêtre Pyton IDLE et d'observer. Nous avons ensuite expliqué la réponse donnée par l'ordinateur en abordant l'écriture en base 2. Je leur ai dit que $0,1$ n'a pas de valeur exacte en binaire. Beaucoup de nombres décimaux n'ont pas de valeur exacte en binaire, ce qui fait qu'il n'est pas possible de comparer deux nombres décimaux en informatique.

Après explication de ce qu'est un triplet pythagoricien, nous avons repris notre script et l'avons finalisé en écrivant :

```
# Ce programme sert à savoir si trois nombres entiers forment un triplet
   pythagoricien.
print("Vous allez saisir trois nombres entiers A, B et C dans
   l'ordre croissant")
A=int(input("A="))
B=int(input("B="))
C=int(input("C="))
X=A**2+B**2
Y=C**2
if X==Y:
    print("C'est un triplet pythagoricien")
else:
    print("Ce n'est pas un triplet pythagoricien")
```