Géométrie et informatique

Un TP d'info-géométrie peut être un bon moyen de faire écrire aux élèves des fonctions simples, et de leur faire réviser les formules de base. En outre, il leur demandera, comme toujours en informatique en fait, de réfléchir au sens précis des notations : ne pas mélanger la distance AB, le segment $[AB]$, le vecteur $\overrightarrow{AB}$... Enfin, c'est un bon moyen d'introduire les tableaux : un point ou un vecteur pourra être représenté par ses coordonnées, qui pourront être rentrées dans un tableau de deux ou trois cases.

Pour cette séance comme pour chaque séance de cette formation, je propose de procéder en trois étapes :

1. Programmer les fonctions suggérées ;
2. S'il y lieu, comparer les diverses propositions et discuter de la version la plus claire et efficace, à ce stade dans l'absolu (c'est-à-dire indépendamment du niveau des élèves) ;
3. Discuter en pratique de ce qu'il est possible de présenter aux élèves, et de quelle manière.

1 Fonctions simples

Je propose ici de représenter un point, ainsi qu'un vecteur, par le tableau de ses coordonnées. Par exemple, `A=[1,2]`, ou `v=[0,3]` définissent le point A de coordonnées $(1, 2)$, et le vecteur $\vec{v}$ de coordonnées $(0, 3)$ dans un·e certain·e repère/base orthonormé·e.

Pour accéder à l'abscisse de A, on tape alors `A[0]`, et pour l'ordonnée `A[1]`.

Remarque : Il n'y aura pas de différence entre un vecteur et un point dans cette représentation. Il serait cependant possible en Python de le faire (créer des « classes » !), et donc de faire en sorte que Python déclenche une erreur si par exemple un élève essaie d'additionner deux points.

Remarque : Python dispose aussi d'un type pour les couples (plus généralement les n-uplets). En gros, il suffit de remplacer les crochets par des parenthèses. Par exemple `A=(1,2)` définit cette fois A comme le *couple* $(1, 2)$. Pour ce que nous allons faire aujourd'hui, il n'y a pas de différence entre les n-uplets et les tableaux. (Plus de détails sur demande !)

Commençons par programmer les fonctions suivantes :

1. Norme d'un vecteur ;
2. Le vecteur entre deux points ;
3. Distance entre deux points ;
4. Tester si un triangle est rectangle via Pythagore ; *Nombreux commentaires ici, cf corrigé.*
5. Somme et produit par un scalaire. **N.B.** Que renvoie `u+v` si `u` et `v` sont deux vecteurs, donc ici deux tableaux de deux nombres ?
6. Déterminer la nature d'un triangle (rectangle, isocèle, équilatéral) ?
7. Produit scalaire. Tester si un triangle est rectangle via le produit scalaire ;
8. Calculer un centre de gravité.

2 Pour ceux qui s'ennuient : nombres complexes et dessin du dragon

On peut représenter un point ou un vecteur du plan également par un nombre complexe. Il se trouve que Python est un des rares langages de programmation qui propose en natif un type `complex` pour les nombres complexes. Résumé de syntaxe :

- Le nombre i tel que $i^2 = -1$ s'obtient en tapant `1j`. Plus généralement, pour tous flottant a et b, `a+bj` représente le complexe $a + ib$.
- Les opérations classiques `* + - / ...` s'utilisent comme pour les flottants.
- On peut récupérer parties réelle et imaginaire par les attributs `real` et `imag`. Par exemple `z.real` renvoie la partie réelle de z.

On propose ci-dessous de tracer une fractale connue sous le nom de « dragon ».

1. Soient A et B deux points d'affixe z_A et z_B. Soit C tel que ABC est un triangle rectangle isocèle en C direct. Calculer l'affixe z_C de C.

2. Programmer la fonction `calcule_zc` correspondante.
3. La fonction `dragon` est alors définie comme ceci :
   - Pour tous points A et B, `dragon(0,A,B)` est le segment $[AB]$.
   - Pour tous points A et B et tout entier $n \in \mathbb{N}^*$, soit C comme dans la question 1, alors `dragon(n,A,B)` est la concaténation de `dragon(n-1,A,C)` et de `dragon(n-1,B,C)`.

   Programmer la fonction `dragon`. On rappelle qu'on représentera les points par leur affixe. Pour gagner du temps, on fournit une fonction `traceDepuisC` qui prend en argument une liste de nombres complexes et qui trace la suite de segments correspondante. Vous êtes vivement encouragés à regarder comme cette fonction fonctionne.

Attention : l'ordre des points compte. `dragon(n,A,B)` est différent de `dragon(n,B,A)`.

Lorsqu'on rassemble les deux morceaux, il faudra renverser une des deux listes pour qu'ils se raboutent correctement. Utiliser la fonction `rev`.

3 Égalité de deux flottants

On s'en sera rendu compte en testant certaines des fonctions ci-dessus : tester l'égalité de deux flottants n'a pas grand sens. Deux nombres flottants ont une probabilité presque nulle d'être égaux. En fait, certains langages de programmation ne permettent tout simplement pas de tester l'égalité de deux flottants !

Tester par exemple : `0.1*0.1 == 0.01`

`1.1+2.2`

Pour mieux comprendre ce genre de phénomène, il faut avoir conscience qu'un flottant est enregistré sous forme « scientifique en base 2 », c'est-à-dire sous la forme (pour un processeur en 64 bits) :

$$\pm 0{,}b_1 b_2 \cdots_{52} \times 2^e$$

où e est un entier codé sur 11 bits.

Dans l'exemple précédent, 0,1 est un nombre décimal qui ne peut être écrit de manière exacte en base 2. Ainsi, l'ordinateur enregistre une valeur approchée de ce nombre, et les erreurs d'arrondi arrivent très vite.

Remarque : Pour afficher le nombre réellement enregistré à la place de 0,1, on peut utiliser la commande `decimal.Decimal(0.1)` après avoir chargé la bibliothèque `decimal`.

Autre exemple classique :

- Écrire une fonction pour résoudre une équation de degré 2.
- La tester sur les équations $x^2 + 0{,}2x + 0{,}01 = 0$ puis $x^2 + 1{,}4x + 0{,}49 = 0$.

On notera que tout devient plus clair si on illustre en dessinant la parabole et ses points d'intersection avec l'axe des abscisses. Au passage, le premier exemple est intéressant pour illustrer la notion de racine double.

Pour aller plus loin : tester les deux bouts de code suivant :

```
def test1():
    x=0.5
    compte=0
    while x!=2*x:
        x*=2
        compte+=1
    return compte
```

```
def test2():
    x=0.5
    compte=0
    while 0.5+x != 0.5:
        x/=2
        compte+=1
    return compte
```

Remarque :

- `(2**(-53)+1)-1 != 2**(-53)+(1-1)` : l'addition des flottants n'est pas associative.
- De même, la multiplication n'est pas distributive sur l'addition.
- Par contre, l'addition et la multiplication sont commutatives.

En termes savants, l'ensemble des flottants muni des opérations implémentées par Python n'est pas un sous-corps de $(\mathbb{R}, +, \cdot)$.