

# Exercices de Python

par RAYMOND MOCHÉ

ancien élève de l'École normale d'instituteurs de Douai et de l'École normale supérieure de Saint Cloud

ancien professeur de l'Université des Sciences et Technologies de Lille  
et des universités de Coïmbre et de Galatasaray

[http ://gradus-ad-mathematicam.fr](http://gradus-ad-mathematicam.fr)

# Table des matières

<b>1</b>	<b>Liste des Énoncés</b>	<b>3</b>
<b>2</b>	<b>Fonctions, boucles, instructions conditionnelles</b>	<b>14</b>
2.1	[*] Pour commencer : des calculs. . . . .	16
2.2	[*] Pour commencer : graphes simples. . . . .	17
2.3	[*] Représentations graphiques avec Python . . . . .	21
2.4	[*] Courbes définies paramétriquement . . . . .	27
2.5	[*] Polygones réguliers . . . . .	29
2.6	[*] Détermination graphique de $\sqrt{2}$ . . . . .	33
2.7	[*] Zéros d'une fonction par dichotomie. . . . .	36
2.8	[*] Programmer des fonctions : valeur absolue, volume du cône . . . . .	39
2.9	[*] Trinôme du second degré . . . . .	41
2.10	[**] Minimum de 2, 3, 4 nombres (instructions conditionnelles) . . . . .	43
2.11	[***] Alignement de 3 points . . . . .	46
2.12	[*] Affectation de données . . . . .	48
2.13	[*] Algorithme d'Euclide. . . . .	51
<b>3</b>	<b>Listes</b>	<b>53</b>
3.1	[**] Modélisation du jeu du lièvre et la tortue à l'aide de listes Python . . . . .	55
3.2	[*] Simuler $n$ lancers d'un dé équilibré, sous forme de liste . . . . .	57
3.3	[*] Maximum d'une liste de nombres . . . . .	58
3.4	[*] Ranger une liste de nombres . . . . .	60
3.5	[*] Permutation aléatoire d'une liste. . . . .	63
3.6	[*] Une liste de listes . . . . .	64
3.7	[**] Écrire toutes les permutations d'une liste donnée $L$ . . . . .	66
<b>4</b>	<b>Matrices-lignes</b>	<b>68</b>
4.1	[**] Maximum et minimum d'une matrice-ligne . . . . .	70
4.2	[***] Permutations aléatoires d'une matrice-ligne. . . . .	73
<b>5</b>	<b>Matrices (matrices (n,m))</b>	<b>76</b>
5.1	[*] Matrice croix (fantaisie). . . . .	77
5.2	[*] Matrice Zorro (fantaisie) . . . . .	78
5.3	[*] Transformations aléatoires de matrices . . . . .	79
5.4	[*] Calculer la trace d'une matrice . . . . .	81
5.5	[**] Ce carré est-il magique ? . . . . .	84
5.6	[***] Transposer une matrice . . . . .	88
5.7	[****] Écrire toutes les permutations de $(1, \dots, n)$ . . . . .	92
5.8	[****] Combien y a-t-il de carrés magiques d'ordre 3 ? . . . . .	96
5.9	[***] Construction des premiers flocons de Koch avec <code>plot</code> . . . . .	98

<b>6</b>	<b>La tortue de Python</b>	<b>101</b>
6.1	[*] Médiannes concourantes d'un triangle . . . . .	102
6.2	[*] Illustration du théorème de Pythagore . . . . .	104
6.3	[*] La tortue et les représentations graphiques. . . . .	107
6.4	[***] Algorithme de Bresenham . . . . .	108
6.5	[***] Construction itérative des triangles de Sierpinski . . . . .	112
<b>7</b>	<b>Algorithmes récursifs</b>	<b>115</b>
7.1	[*] Calculer récursivement le maximum d'une famille finie de nombres . . . . .	117
<b>8</b>	<b>Arithmétique, compter en nombres entiers</b>	<b>119</b>
8.1	[***] PGCD de p entiers positifs. . . . .	120
8.2	[****] Crible d'Ératosthène . . . . .	122
8.3	[***] Factoriser un entier en produit de nombres premiers . . . . .	125
	<b>Bibliographie</b>	<b>126</b>

# Chapitre 1

## Liste des Énoncés

---

### Chapitre 2 : Fonctions, boucles, instructions conditionnelles

---

Énoncé n° 2.1 [\*] : Pour commencer : des calculs.

Quelle est la valeur de  $\pi$  ? de  $e$  ? de  $\pi \cdot e$  ? Calculer  $\frac{\sin(3) \cdot e^2}{1 + \tan\left(\frac{\pi}{8}\right)}$ .

---

**Mots-clefs** : constantes pré-définies  $e$ ,  $\pi$ , calcul numérique, module `math`, fonctions `sin` et `tan`.

---

Énoncé n° 2.2 [\*] : Pour commencer : graphes simples.

- 1 - Tracer le segment de droite d'extrémités (1,3) et (15,-2), dans un repère orthonormé.
- 2 - Tracer le graphe de la fonction `sinus` quand la variable varie de 1 à 15 radians.

---

**Mots-clefs** : Instructions simples de tracer des graphes, commandes `linspace(a,b,n)` et `array([a,...,x])` du module `numpy`, fonctions vectorisées.

---

Énoncé n° 2.3 [\*] : Représentations graphiques avec Python.

Pour tracer un graphique, Python en calcule des points consécutifs et les relie par un segment. Habituellement, on obtient ainsi une bonne approximation de la courbe à tracer quand il y a suffisamment de points car alors, nos yeux ne font plus la différence.

On se propose d'étudier des approximations du graphe de la fonction `sin()` lorsque la variable  $x$  varie sur l'intervalle  $[0, \pi]$ .

- 1 - Tracer l'approximation du graphe obtenue en utilisant seulement ses points d'abscisse 0,  $\pi/2$  et  $\pi$ .
- 2 - Pour  $n$  valant successivement 3, 6, 10, 20, tracer l'approximation du graphe obtenue en calculant les points dont les abscisses sont données par la commande `linspace(0,pi,n+1)`.

---

**Mots-clefs** : Représentation graphique d'une fonction, repère orthonormé, commande `axis('equal')`, autres commandes graphiques.

---

Énoncé n° 2.4 [\*] : Courbes définies paramétriquement.

Les représentations graphiques des fonctions sont un cas particulier de courbes définies paramétriquement : si  $\mathbf{f}$  et  $\mathbf{g}$  sont deux fonctions définies sur un même intervalle  $I$ , le point  $M(t)$  de coordonnées  $x=f(t)$  et  $y=g(t)$  se déplace et engendre une courbe  $(C)$ , quand  $t$  décrit l'intervalle  $I$ . On peut de cette manière tracer de nombreuses courbes qui ne sont pas des représentations graphiques de fonctions.

À l'aide du module `matplotlib` de Python, tracer la courbe  $(C)$  d'équations paramétriques  $x = \cos(3t)$ ,  $y = \sin(2t)$ .

**Mots-clefs :** Tracés de courbes, `matplotlib`.

**Énoncé n° 2.5** [\*] : Polygones réguliers.

Le plan étant rapporté à un repère orthonormé,  $(C)$  désigne le cercle centré à l'origine des axes et de rayon 1,  $n$  un entier au moins égal à 2.

**1** - [Polygones réguliers convexes] Pour construire un polygone régulier à  $n$  côtés inscrit dans  $(C)$ , on place successivement les points  $A_k$ ,  $k = 0, 1, \dots, n$  de coordonnées  $(\cos(k \cdot \frac{2\pi}{n}), \sin(k \cdot \frac{2\pi}{n}))$ <sup>a</sup> et on joint chaque point au suivant par un segment de droite. On obtient un polygone à  $n$  côtés<sup>b</sup>.

**1.a** - Programmer une fonction `n --> polygone(n)` qui pour tout entier  $n \geq 2$  donné retourne le polygone à  $n$  côtés décrit ci-dessus<sup>c</sup>.

**1.b** - Décrire les polygones `polygone(3)` et `polygone(4)`.

**2** - [Polygones réguliers convexes ou étoilés] Soit  $n$  un entier au moins égal à 3 et  $r$  un entier vérifiant  $1 \leq r \leq n - 1$ . On recommence le tracé décrit à la question précédente en remplaçant l'angle  $\frac{2\pi}{n}$  par l'angle  $\frac{2\pi r}{n}$ . On obtient évidemment une ligne polygonale fermée, notée `polygone2(n,r)` ci-dessous.

**2.a** - Cas  $n = 3$  : décrivez `polygone2(3,1)` et `polygone2(3,2)`.

**2.b** - [Tracés] Programmer une fonction qui, étant donné  $n$  et  $r$ , retourne les polygones `polygone2(n,1), ..., polygone2(n,n-1)`.

**2.c** - À titre d'exemple, tracer les polygones obtenus pour  $n = 9$  puis  $n = 10$ .

<sup>a</sup>. Il est clair que  $A_0 = A_n$

<sup>b</sup>. appelé polygone régulier convexe

<sup>c</sup>. `polygone(2)` se réduit évidemment au segment  $[-1, 1]$  - un diamètre - de l'axe des abscisses.

**Mots-clefs :** tracés avec Python, cercle trigonométrique, fonctions sinus et cosinus, modules `math`, `numpy`, `matplotlib`.

**Énoncé n° 2.6** [\*\*] : Détermination graphique de  $\sqrt{2}$ .

Déterminer une valeur approchée de  $\sqrt{2}$  comme l'abscisse de l'intersection du graphe de la fonction  $x \rightarrow x^2$ ,  $x$  variant de 0 à 2 et du segment de droite parallèle à l'axe des abscisses d'ordonnée 2 obtenu de même en faisant varier  $x$  de 0 à 2. Plus précisément,

1 - déterminer une telle valeur approchée quand on approxime le graphe de  $x \rightarrow x^2$  par une ligne polygonale à  $n = 10$  côtés (voir l'exercice 2.3)

2 - puis quand on l'approxime par une ligne polygonale à  $n = 1000$  côtés.

**Mots-clefs :** `matplotlib.pyplot`.

**Énoncé n° 2.7** [\*] : Zéros d'une fonction par dichotomie.

Soit  $f$  une fonction usuelle définie sur un intervalle  $[a, b]$  telle que  $f(a) \cdot f(b) < 0$ . On est sûr que  $f$  s'annule au moins une fois entre  $a$  et  $b$  en un point  $p$  que l'on voudrait déterminer, appelé zéro de  $f$ . Comme approximation de  $p$ , on peut proposer le milieu  $m$  de  $[a, b]$ . Dans ce cas, l'erreur d'approximation est majorée par  $\frac{b-a}{2}$ .

On est sûr aussi que l'une des inégalités  $f(a) \cdot f(m) \leq 0$  ou  $f(m) \cdot f(b) < 0$  est vraie, ce qui montre que  $f$  s'annule dans l'intervalle  $]a, m]$  dans le premier cas ou dans l'intervalle  $]m, b[$  dans le deuxième cas. Comme nouvelle approximation de  $p$ , on peut maintenant choisir le milieu de  $[a, m]$  ou de  $[m, b]$  suivant le cas. C'est un progrès car l'erreur d'approximation est maintenant majorée par  $\frac{b-a}{2^2}$ . Et on peut ré-itérer ce procédé.

1 - Étant donné un entier  $n \geq 0$ , programmer une fonction qui retourne une approximation de  $p$  avec une erreur d'approximation majorée par  $10^{-n}$ .

2 - Trouver une approximation de  $\sqrt{2}$  avec une précision de  $10^{-6}$ .

3 - Trouver un zéro de la fonction  $f$  définie sur l'intervalle  $[0, \pi]$  par :

$$f(x) = x - 1 \quad \text{si } x \leq 1, \quad = (x - 1) \sin\left(\frac{1}{x - 1}\right) \quad \text{sinon,}$$

à  $10^{-6}$  près.

---

**Mots-clefs** : zéros d'une fonction, dichotomie, programmation d'une fonction, d'une fonction définie par morceaux, boucle **tant que**, instructions conditionnelles **if ... else: ...**

---

**Énoncé n° 2.8** [\*] : Programmer des fonctions : valeur absolue, volume du cône.

- 1 - Programmer une fonction qui retourne la valeur absolue de tout nombre.
- 2 - Programmer une fonction qui retourne le volume de tout cône droit à base circulaire, de hauteur  $h$  cm et de rayon  $r$  cm.

---

**Mots-clefs** : programmer une fonction, instruction conditionnelle **if: ...else: ...**, importer une constante ou une fonction d'un module de Python.

---

**Énoncé n° 2.9** [\*] : Trinôme du second degré.

Résoudre l'équation du second degré (E) :  $ax^2 + bx + c = 0$  ( $a \neq 0$ )

---

**Mots-clefs** : Fonction `sqrt()`, à importer du module `math`, sortie d'une fonction par `return` ou `print()`, instruction conditionnelle **if ...:elif ...:else:...**

---

**Énoncé n° 2.10** [\*\*] : Minimum de 2, 3, 4 nombres (instructions conditionnelles).

- 1.a - Programmer une fonction qui retourne le minimum de 2 nombres.
- 1.b - Programmer une fonction qui retourne le minimum et le maximum de 2 nombres.
- 2 - Programmer une fonction qui retourne le minimum de 3 nombres.
- 3 - Programmer une fonction qui retourne le minimum de 4 nombres.

---

**Mots-clefs** : Programmation de fonctions, instruction conditionnelle **if ...:else ...:**, **return**, négation d'une proposition logique, importation de constantes et de fonctions pré-définies.

---

**Énoncé n° 2.11** [\*\*\*] : Alignement de 3 points.

On se donne, à l'aide de leurs coordonnées, 3 points distincts  $A$ ,  $B$  et  $C$  d'un plan rapporté à un repère orthonormé. On appelle  $d_1$ ,  $d_2$  et  $d_3$  les distances de  $B$  à  $C$ , de  $A$  à  $C$  et de  $A$  à  $B$ . On peut supposer que  $d_1 \leq d_2 \leq d_3$ .

1 - Démontrer que les points sont alignés si et seulement si  $d_3 = d_1 + d_2$ .

2 - Sans utiliser la fonction racine carrée, programmer un algorithme qui retourne, à partir de la liste des coordonnées de  $A$ ,  $B$  et  $C$ , "Les points sont alignés" ou "Les points ne sont pas alignés", suivant le cas.

---

**Mots-clefs :** Calculer avec seulement des additions et des multiplications, transformer un problème en un problème équivalent. Commentaires : tester l'égalité de 2 nombres.

---

**Énoncé n° 2.12** [\*] : Affectation de données

Soit  $f$  une fonction définie sur  $\mathbb{R}$  par  $f(x) = x^2 - 5x + 3$  et  $k$  un nombre réel. On souhaite déterminer tous les entiers  $n$  compris entre deux entiers donnés  $a$  et  $b$  ( $a < b$ ) tels que  $f(n) < k$ .

1 - -10 est-il solution de  $f(n) < 5$ ? Même question pour 0.

2 - Écrire un script Python appelant un nombre et affichant "oui" s'il est solution de  $f(n) < 5$ , "non" sinon.

3 - Modifier le script pour appeler  $k$  et afficher successivement tous les entiers solutions de  $f(n) < k$  sur l'intervalle  $[-10, 10]$ .

4 - Modifier le script pour demander aussi  $a$  et  $b$ .

---

**Mots-clefs :** Affecter des données à un script (commandes **input()** et **eval()**), définir une fonction, instruction conditionnelle **if ...:...else ...:...**, boucle **pour**, conditions de type booléen, variables booléennes, masques, fonction **print()**, itérateur **range**.

---

**Énoncé n° 2.13** [\*] : Algorithme d'Euclide

1 - Programmer une fonction qui, étant donnés deux entiers positifs **a** et **b**, retourne leur PGCD calculé selon l'algorithme d'Euclide.

2 - En déduire le calcul de leur PPCM.

---

**Mots-clefs :** reste et quotient de la division euclidienne (**%** et **//**), algorithme d'Euclide, boucle **tant que**, importation de fonctions, module **math**.

---

## Chapitre 3 : Listes

---

**Énoncé n° 3.1** [\*\*] : Modélisation du jeu du lièvre et la tortue à l'aide de listes Python

On lance un dé équilibré. Si le 6 sort, le lièvre gagne. Sinon la tortue avance d'une case et on rejoue. La tortue gagne si elle parvient à avancer de **n** cases (**n** ≥ 1 donné).

On demande de modéliser ce jeu et de le simuler à l'aide d'une fonction bâtie sur des listes Python.

---

**Mots-clefs :** liste, Python, modélisation, mathématiques, probabilités, simulation.

---

**Énoncé n° 3.2** [\*] : Simuler **n** lancers d'un dé équilibré, sous forme de liste

En utilisant la fonction `randint` du module `random`, simuler, sous forme de liste, `n` lancers d'un dé équilibré.

---

**Mots-clefs** : liste, module `random`, commandes `randint` et `append`, boucle `pour`.

---

**Énoncé n° 3.3** [\*] : Maximum d'une liste de nombres

- 1 - Calculer, à l'aide d'une fonction à programmer, le maximum `M` d'une liste `L` de nombres, de longueur `l` comme suit :
- (a) On pose `M=L[0]`.
  - (b) Ensuite, si `l ≥ 2`, `i` prenant successivement les valeurs `1, ..., l-1`, on pose `M=L[i]` si `L[i] > M`.
- 2 - Application : calculer le maximum d'une liste de 10, puis de 10 000 nombres réels.
- 3 - Calculer le minimum `m` de `L` en utilisant l'algorithme du calcul ci-dessus.

---

**Mots-clefs** : instruction conditionnelle `if ...`, importation de fonction, convertisseur de type `list`, méthode `L.append()` d'une liste `L`, commande `len()`<sup>1</sup>, primitives `min()` et `max()`.

---

**Énoncé n° 3.4** [\*] : Ranger une liste de nombres

- Une liste de nombres `L` est donnée.
- 1 - à l'aide de la primitive `min()`, ranger cette liste dans l'ordre croissant (le premier terme `m` de la liste à calculer sera le minimum de `L`, le suivant sera le minimum de la liste `L` dont on aura retiré `m`, etc).
- 2 - Ranger `L` dans l'ordre décroissant.

---

**Mots-clefs** : primitives `min()` et `max()`, boucle `pour`, méthodes `list.sort()` et `list.reverse()`, fonction `len()` (pour les listes), primitive `sorted()`.

---

**Énoncé n° 3.5** [\*] : Permutation aléatoire d'une liste.

- Programmer une fonction qui retourne sous forme de liste une permutation aléatoire de la liste `L = [1, 2, ...,n]`.
- Méthode imposée : on commencera par tirer au hasard un élément de `L` qui deviendra le premier élément de la permutation aléatoire recherchée et on l'effacera de `L`. Puis on répètera cette opération jusqu'à ce que `L` soit vide.

---

**Mots-clefs** : boucle `pour`, fonction `randint()` du module `random`, primitive `len()`, `L.append(x)`, fonction `del()`, `list(range())`.

---

**Énoncé n° 3.6** [\*] : Une liste de listes.

- Programmer une fonction qui, pour tout entier naturel  $n \geq 2$ , retourne la liste dont le premier élément est la liste `[1, ..., n]`, le second élément la liste `[2, ...,n,1]` et ainsi de suite jusqu'à la liste `[n,1, ...,n-1]`.

---

**Mots-clefs** : `list(range(p,q))`, boucle `pour`, remplacement d'un élément d'une liste par un autre, concaténation de 2 listes.

---

**Énoncé n° 3.7** [\*\*] : Écrire toutes les permutations d'une liste

---

1. `len(L)` retourne la longueur de la liste `L`.



**1** - Étant donné une liste `L` à `n` éléments et une liste `l`, programmer une fonction notée `aïda( , )` de sorte que la commande `aïda(L,l)` retourne une liste de `n+1` listes dont la première est `l+L`<sup>a</sup>, la dernière `L+l`, les listes intermédiaires étant obtenues en insérant `l` dans la liste `L` entre les éléments `L[i]` et `L[i+1]`, `i` variant de 0 à `n-2`.

**2** - `L` étant une liste non vide de longueur `n`, programmer, à l'aide de la fonction `aïda( , )`, une fonction notée `permutliste()` de sorte que la commande `permutliste(L)` retourne la liste des `n!` listes dont chacune est une permutation différente de la liste `L`.

<sup>a</sup>. Concaténée des listes `L` et `l`.

**Mots-clefs** : concaténation de listes, extraction d'une sous-liste d'une liste.

## Chapitre 4 : Matrices-lignes

**Énoncé n° 4.1** [<sup>\*\*</sup>] : Maximum et minimum d'une matrice-ligne

**1** - Calculer le maximum d'une matrice-ligne de nombres `M` en procédant comme suit :

- (a) - on choisit arbitrairement l'un de ces nombres que l'on note `ma` et on efface les nombres  $\leq ma$ .
- (b) - s'il n'en reste plus, `ma` est le maximum recherché.
- (c) - sinon, on recommence (a) et (b).

**2** - Calculer le minimum `mi` de `M`.

**Mots-clefs** : matrices, masque, maximum et minimum, fonctions primitives `max` et `min`, boucle `tant que`, définition d'une fonction, appel d'une nouvelle fonction.

**Énoncé n° 4.2** [<sup>\*\*\*</sup>] : Permutations aléatoires d'une matrice-lignes.

**1** - Programmer une fonction qui retourne une permutation aléatoire de la matrice à une dimension `V = array([1, 2, ...,n])` sous forme de matrice à une dimension.  
Méthode imposée : On commencera par tirer au hasard un élément de `V` qui deviendra le premier élément de la permutation aléatoire `W` recherchée et on l'effacera de `V`. Puis on répètera cette opération tant que `V` n'est pas vide.

**2** - Utiliser la fonction précédente pour fabriquer des permutations aléatoires de n'importe quelle matrice à une dimension `M`.

**Mots-clefs** : fonctions `array()`, `arange()`, `hstack()`, `delete()`, `len()`, `shuffle()` du module `numpy`, `randint()`, `rand()` du module `numpy.random()`, échange d'éléments et extraction d'une sous-matrice d'une matrice à une dimension, `size`, méthode des matrices à une dimension, conversion d'une matrice à une dimension en une liste.

## Chapitre 5 : Matrices ou matrices (n,m)

**Énoncé n° 5.1** [<sup>\*</sup>] : Matrices croix (fantaisie).

Écrire la matrice carrée `M` de taille `2n+1` comportant des 1 sur la  $(n+1)^{\text{ième}}$  ligne et la  $(n+1)^{\text{ième}}$  colonne et des 0 ailleurs.

**Mots-clefs** : module `numpy` : fonctions `np.zeros(( , ), int)`, `np.ones(( , ), int)`, concaténation horizontale et verticale de matrices (respectivement `np.concatenate(( , , ),axis = 1)` et `np.concatenate((`

, , ),axis = 0)).

---

**Énoncé n° 5.2** [\*] : Matrice Zorro (fantaisie)

Écrire la matrice  $M$  de taille  $n$ , définie comme la matrice carrée de taille  $n$  comprenant des 1 sur la première ligne, sur la deuxième diagonale et sur la dernière ligne et des 0 partout ailleurs.

**Mots-clefs** : module `numpy` : fonctions `np.zeros((n, m),int)`, `np.ones((m, ),int)`, `np.arange(p,q,r)`, extraction d'un bloc d'éléments d'une matrice : `M[m1,m2]`.

---

**Énoncé n° 5.3** [\*] : Transformations aléatoires de matrices.

1 - Programmer une fonction qui, pour tout entier  $n$  positif, retourne la matrice carrée dont la première ligne est 1, 2, ...,  $n$ , la suivante  $n+1$ ,  $n+2$ , ...,  $2n$ , et ainsi de suite jusqu'à la dernière ligne <sup>a</sup>.

2 - à l'aide de la fonction `shuffle()` de `numpy.random`, effectuer une permutation aléatoire équiprobable des éléments d'une matrice quelconque donnée  $M$  et afficher la matrice obtenue.

<sup>a</sup>. qui est donc  $n(n-1)+1$ ,  $n(n-1)+2$ , ...,  $n*n$

---

**Mots-clefs** : fonctions `arange()`, `reshape()` du module `numpy`, fonction `shuffle()` du module `numpy.random`, commande `B = A.flatten()`.

---

**Énoncé n° 5.4** [\*] : Calculer la trace d'une matrice.

Soit  $M$  une matrice à  $n$  lignes et  $m$  colonnes.

1 - Calculer la somme  $t1$  de ses éléments dont l'indice de ligne est égal à l'indice de colonne.

2 - Calculer la somme  $t2$  de ses éléments dont l'indice de ligne  $i$  et l'indice de colonne  $j$  vérifient  $i+j=\min(n,m)-1$ , les lignes et les colonnes étant numérotées à partir de 0).

**Mots-clefs** : boucle `pour` ; module `numpy` : extraire un bloc d'éléments d'une matrice, changer l'ordre des lignes ou des colonnes d'une matrice, fonctions `shape()`, `diag()`, `sum()`, `trace()` ; module `numpy.random` : fonction `rand()`, module `copy` : fonction `deepcopy`.

---

**Énoncé n° 5.5** [\*\*] : Ce carré est-il un carré magique ?

Convenons qu'une matrice carrée  $M$  d'ordre  $n$  est un carré magique d'ordre  $n$  si ses éléments sont les entiers de 1 à  $n^2$  disposés de sorte que leurs sommes sur chaque ligne, sur chaque colonne et sur les deux diagonales soient égales. La valeur commune de ces sommes est appelée constante magique de  $M$ .

1 - S'il existe un carré magique d'ordre  $n$ , combien vaut sa constante magique ?

2 - `LuoShu = array([[4,9,2],[3,5,7],[8,1,6]])` est-il un carré magique ? Si oui, quelle est sa constante magique ?

3 - Même question pour `Cazalas = array([[1,8,53,52,45,44,25,32],[64,57,12,13,20,21,40,33],[2,7,54,51,46,43,26,31],[63,58,11,14,19,22,39,34],[3,6,55,50,47,42,27,30],[62,59,10,15,18,23,38,35],[4,5,56,49,48,41,28,29],[61,60,9,16,17,24,37,36]])`.

4 - Même question pour `Franklin = array([[52,61,4,13,20,29,36,45],[14,3,62,51,46,35,30,19],[53,60,5,12,21,28,37,44],[11,6,59,54,43,38,27,22],[55,58,7,10,23,26,39,42],[9,8,57,56,41,40,25,24],[50,63,2,15,18,31,34,47],[16,1,64,49,48,33,32,17]])`.

---

**Mots-clefs :** `V.size`, `M.sum(axis=1)`, `M.sum(axis=0)`, `trace(M)`, `M[arange(n-1,-1,-1), arange(0,n)]` (extraction d'une matrice 1-dimensionnelle de  $M$ ), `hstack((,))`, `min()`, `max()`.

---

**Énoncé n° 5.6** [\*\*\*] : Transposer une matrice

Programmer une fonction qui, étant donné une matrice numérique  $A$ , retourne la matrice  $B$  dont la première colonne est la première ligne de  $A$ , la seconde la deuxième ligne de  $A$ , etc.  $B$  s'appelle la transposée de  $A$ .

---

**Mots-clefs :** Boucle pour imbriquée dans une autre, commande `range`, module `numpy`, fonctions `shape()` (dimension d'une matrice), `zeros()`, `vstack()` et `hstack()` (empilements de matrices), `transpose()`, méthode (de l'objet matrice) `.T`.

---

**Énoncé n° 5.7** [\*\*\*\*] : Écrire toutes les permutations de  $(1, \dots, n)$ .

1 -  $M$  désignant une matrice quelconque d'entiers à `li` lignes et `co` colonnes,  $n$  un entier quelconque, programmer une fonction qui retourne la matrice `Sortie` qui empile verticalement les `co+1` matrices obtenues en adjoignant à  $M$  une colonne  $A$  à `li` lignes dont tous les éléments sont égaux à  $n$ ,  $A$  étant placée d'abord devant  $M$ , puis entre la première et deuxième colonne de  $M^a$ , etc, jusqu'à ce que  $A$  devienne sa dernière colonne. La matrice obtenue aura `co+1` colonnes et `(n+1).li` lignes.

2 - Écrire toutes les permutations de  $(1, \dots, n)$ .

<sup>a</sup> ce qui revient à décaler  $A$  d'un cran vers la droite ou d'échanger les deux premières colonnes

---

**Mots-clefs :** appel de modules et de fonctions, fonctions `ones()`, `zeros()`, `hstack()` du module `numpy`, méthode `shape()` des matrices, fonction `clock()` du module `time`, échange de colonnes d'une matrice, extraction d'une sous-matrice d'une matrice.

---

**Énoncé n° 5.8** [\*\*\*\*] : Carrés magiques d'ordre 3.

Combien y a-t-il de carrés magiques d'ordre 3 ?

---

**Mots-clefs** : permutations, appeler une fonction, commandes `factorial()`, `clock()`, matrices à une (commandes `min` et `max`, `reshape()`) et deux dimensions (extraction d'une ligne, somme sur les lignes et les colonnes, `trace()`), conversion d'une matrice à une dimension en une matrice à deux dimensions, listes (commande `append()`).

---

**Énoncé n° 5.9** [\*\*\*] : Construction des premiers flocons de Koch avec `plot`.

Un polygone  $K_0$  étant donné sous la forme d'une matrice à deux lignes (ligne 1 : ligne des abscisses des sommets, ligne 2 : ligne des ordonnées des sommets), programmer une fonction qui retourne le polygone  $K_n$ ,  $n \geq 1$  ainsi défini :

Polygone  $K_1$  : le polygone  $K_1$  est construit à partir de  $K_0$  en remplaçant le premier côté  $AB$  de  $K_0$  par la ligne brisée  $AMGNB$ ,  $M$  étant au premier tiers de  $AB$  à partir de  $A$ ,  $N$  étant au second tiers,  $G$  étant positionné à droite de  $AB$  quand on va de  $A$  vers  $B$ , de sorte que le triangle  $MGN$  soit équilatéral, puis en faisant de même avec le deuxième côté de  $K_0$  et ainsi de suite jusqu'au dernier côté de  $K_0$ .

Polygone  $K_n$  : Si  $n \geq 2$ , on construit de même  $K_2$  à partir de  $K_1$ , puis  $K_3$  à partir de  $K_2$  et ainsi de suite jusqu'à  $K_n$ .

Si  $K_0$  est un hexagone régulier,  $K_n$  s'appelle  $n^{\text{ième}}$  flocon de Koch.

---

**Mots-clefs** : Définir et appeler une fonction; module `numpy` : fonctions vectorisées, matrices; module `matplotlib`.

---

## Chapitre 6 : La tortue de Python

---

**Énoncé n° 6.1** [\*] : Médiannes concourantes d'un triangle

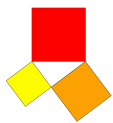
Vérifier expérimentalement que les médianes d'un triangle sont concourantes avec la tortue de Python

---

**Mots-clefs** : commandes `goto()`, `up()` et `down()`, `hideturtle()`, coordonnées du milieu d'un segment.

---

**Énoncé n° 6.2** [\*] : Illustration du théorème de Pythagore avec la tortue de Python

Tracer le dessin suivant à l'aide de la tortue :  Le triangle est rectangle. On pourra placer son hypoténuse sur l'axe des abscisses, l'angle droit en-dessous, les longueurs des côtés étant 60, 80 et 100.

---

**Mots-clefs** : Déplacements en avant et en arrière, rotations de la tortue, polygones en couleur, remplissage de polygones.

---

**Énoncé n° 6.3** [\*] : La tortue et les représentations graphiques

Tracer l'arc de sinusé entre 0 et  $\pi$  avec la tortue.

---

**Mots-clefs** : tortue, sinusé, ligne polygonale.

---

**Énoncé n° 6.4** [\*\*\*] : Algorithme de Bresenham

- 1 - Pour simplifier, disons qu'un écran d'ordinateur est un quadrillage de pixels, chacun d'eux pouvant être allumé ou éteint. Comment représenter un segment sur un tel écran ?

2 - Application : Tracer un triangle ainsi que ses médianes. Sont-elles concourantes ?
- 

**Mots-clefs** : tortue, fonction partie entière (mathématiques).

---

**Énoncé n° 6.5** [\*\*\*] : Construction itérative des triangles de Sierpinski

Les triangles de Sierpinski sont en fait des carrelages d'un triangle équilatéral de base, par exemple le triangle  $S_0$  de sommets  $(0, 0)$ ,  $(a, 0)$  et  $(\frac{a}{2}, \frac{a\sqrt{3}}{2})$  où  $a$  est un certain nombre de pixels, par exemple  $a = 200$ .

Le premier triangle de Sierpinski  $S_1$  s'obtient en coloriant le triangle équilatéral dont les sommets sont les milieux des côtés de  $S_0$ . C'est donc un carrelage de  $S_0$  formé de 4 triangles équilatéraux de côtés de longueur  $\frac{a}{2}$ , 3 triangles étant blancs, le triangle central étant coloré.

Et ainsi de suite : ayant tracé  $S_i$ , on obtient  $S_{i+1}$  en remplaçant chaque triangle blanc par 4 triangles, en utilisant comme précédemment les milieux de ses côtés.

Le problème est de programmer les déplacements de la tortue de sorte qu'elle dessine  $S_n$ .

---

**Mots-clefs** : Listes, algorithme itératif (non récursif), tortue.

---

## Chapitre 7 : Algorithmes récursifs

---

**Énoncé n° 7.1** [\*] : Calculer récursivement le maximum d'une famille finie de nombres

Calculer récursivement le maximum d'un ensemble fini  $E$  de nombres, en remarquant que le maximum de  $E$  est le maximum d'un terme de  $E$  et du maximum des autres éléments.

---

**Mots-clefs** : algorithme récursif, matrice-ligne, liste, commandes `len(L)` et `M.size`, instruction conditionnelle, extraction d'éléments d'une matrice-ligne ou d'une liste, fonction `normal()` du module `numpy.random`, convertisseur de type `list()`.

---

## Chapitre 8 : Arithmétique, compter en nombres entiers

---

**Énoncé n° 8.1** [\*\*\*] : PGCD de  $p$  entiers positifs

- 1 - Programmer une fonction qui, étant donnés  $p$  entiers positifs  $a_1, \dots, a_p$  dont le minimum est noté  $m$ , retourne la matrice à une dimension de leurs diviseurs communs en procédant de la manière suivante : retirer de la suite  $1, \dots, m$  les nombres qui ne sont pas des diviseurs de  $a_1$ , puis ceux qui ne sont pas des diviseurs de  $a_2$ , etc.

2 - En déduire leur PGCD.
- 

**Mots-clefs** : module `numpy`, fonctions `arange()`, `size()`, `sort()`, masque de matrice, `where()`, reste de la division euclidienne, extraction des éléments d'une matrice (`m, i`) dont les indices sont donnés, fonction pré-définie `min()`, module `math`, fonction `gcd` de ce module.

---

**Énoncé n° 8.2** [\*\*\*\*] : Crible d'Ératosthène

- |   |
|---|
| <p>1 - Programmer une fonction qui, pour tout entier <math>n \geq 2</math> donné, retourne les nombres premiers compris entre 2 et <math>n</math>, rangés dans l'ordre croissant.</p> <p>2 - Programmer une fonction qui, étant donné un nombre entier <math>n \geq 2</math>, retourne suivant le cas "<math>n</math> est premier" ou "<math>n</math> n'est pas premier".</p> |
|---|

---

**Mots-clefs** : effacer des éléments d'une matrice à une dimension à l'aide d'un masque.

---

**Énoncé n° 8.3** [\*\*\*] : Factoriser un entier en produit de nombres premiers.

Programmer une fonction qui, étant donné un entier $n \geq 2$ , retourne sa factorisation en produit de nombres premiers <sup>a</sup> .
---

---

*a.* On peut utiliser la fonction `erato()` de l'exercice 8.2.

---

**Mots-clefs** : appel d'une fonction, boucles `pour` et `tant que`.

---



## Chapitre 2

# Fonctions, boucles, instructions conditionnelles

---

Liste des exercices :

Énoncé n° 2.1 [\*] : Pour commencer : des calculs.

Énoncé n° 2.2 [\*] : Pour commencer : graphes simples.

Énoncé n° 2.3 [\*] : Représentations graphiques avec Python.

Énoncé n° 2.4 [\*] : Courbes définies paramétriquement.

Énoncé n° 2.5 [\*] : Polygones réguliers.

Énoncé n° 2.6 [\*\*] : Détermination graphique de  $\sqrt{2}$ .

Énoncé n° 2.5 [\*\*] : Polygones réguliers.

Énoncé n° 2.7 [\*] : Zéros d'une fonction par dichotomie.

Énoncé n° 2.8 [\*] : Programmer des fonctions : valeur absolue, volume du cône.

Énoncé n° 2.9 [\*] : Trinôme du second degré.

Énoncé n° 2.10 [\*\*] : Minimum de 2, 3, 4 nombres (instructions conditionnelles).

Énoncé n° 2.11 [\*\*\*] : Alignement de trois points

Énoncé n° 2.12 [\*] : Affectation de données

Énoncé n° 2.13 [\*] : Algorithme d'Euclide

---

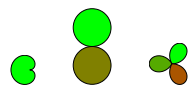
Ce recueil d'exercices avec Python est écrit à l'aide de l'environnement **Spyder** de Python et de sa console interactive **iPython**. La plupart des graphes y sont tracés à l'aide du package de tracés 2d **matplotlib** et de son interface **pylab**, l'exception étant l'usage de la tortue (**turtle**)<sup>1</sup>.

Représenter une fonction graphiquement est compliqué car les tracés peuvent être l'objet de nombreuses options.

Les quelques exercices sur les graphes qui suivent sont des modèles de référence, pour débiter. Ils se compliquent très vite mais il ne faut pas s'inquiéter. En effet, il existe une excellente documentation sur **matplotlib** en français, avec beaucoup d'exemples, à savoir [20]. On peut en extraire les paquets de lignes de code les plus utilisées pour régler les graphiques. Il suffit de savoir qu'ils existent, de comprendre à quoi ils servent et de consulter [20] quand c'est nécessaire. C'est ce que nous avons fait. Bien entendu, on peut aussi se référer à [19].

---

1. pour les tracés qui s'y prêtent





## 2.1 [\*] Pour commencer : des calculs.

Quelle est la valeur de  $\pi$  ? de  $e$  ? de  $\pi \cdot e$  ? Calculer  $\frac{\sin(3) \cdot e^2}{1 + \tan(\frac{\pi}{8})}$ .

**Mots-clefs** : constantes pré-définies  $e$ ,  $\pi$ , calcul numérique, module `math`, fonctions `sin` et `tan`.

Dans Python, il y a des constantes pré-définies, comme  $e$  et  $\pi$ , des fonctions standard (utilisables immédiatement)<sup>2</sup> et une foule d'autres fonctions qui sont regroupées par thème dans des bibliothèques de fonctions ou **modules**. Par exemple, `sin`, `tan` sont des fonctions du module `math`. Si dans un même script, on veut se servir de la constante  $e$  et de la fonction `sin`, il faudra les importer, en important tout le module `math` comme ceci :

```
from math import *
```

ou mieux<sup>3</sup>, en important seulement  $e$  et `sin`, comme cela :

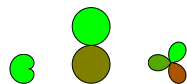
```
from math import e, sin
```

Dans l'exemple qui suit, on a oublié d'importer  $\pi$ , ce qui provoque un message d'erreur :

```
from math import e, sin
e
Out[2] : 2.718281828459045
sin(pi/6)
Traceback (most recent call last):
  File "<ipython-input-3-dbaab9b2f48b>", line 1, in <module>
    sin(pi/6)
NameError: name 'pi' is not defined
```

Nous ne soucions pas du format des nombres dans cette prise de contact<sup>4</sup>. Les règles de calcul sont les règles usuelles. Le problème posé est trivial. Comme il y a peu à écrire, on peut travailler directement dans la console<sup>5</sup>. Ouvrons-en une nouvelle :

```
from math import e, pi, sin, tan
pi
Out[2] : 3.141592653589793
e
Out[3] : 2.718281828459045
pi*e
Out[4] : 8.539734222673566
sin(3)*e**2/(1+tan(pi/8))
Out[5] : 0.7373311103636621
```



2. En anglais, les "built-in functions", voir [13].

3. pour économiser du temps, de l'espace-mémoire, etc.

4. Nous utilisons donc les réglages de Python par défaut.

5. Normalement, on écrit dans l'interpréteur, qui est un traitement de texte adapté à Python. On écrit plus vite, les retraits - très importants en programmation Python - se font automatiquement, on peut corriger facilement, etc.

## 2.2 [\*] Pour commencer : graphes simples.

- 1 - Tracer le segment de droite d'extrémités (1,3) et (15,-2), dans un repère orthonormé.
  - 2 - Tracer le graphe de la fonction `sinus` quand la variable varie de 1 à 15 radians.

**Mots-clefs :** Instructions simples de tracer des graphes, commandes `linspace(a,b,n)` et `array([a,...,x])` du module `numpy`, fonctions vectorisées, interface `pylab` de `matplotlib`.

**Solution :** La première question de cet exercice<sup>6</sup> peut paraître débile. En fait, elle est fondamentale car tout graphe se réduit à une succession de segments de droite, autrement dit à une ligne polygonale. En effet, pour tracer un graphe, **Python** calcule les coordonnées de points successifs du graphe et relie les points consécutifs par un segment (voir 2.3). Si ces points sont assez nombreux, les imperfections de nos yeux font que l'on voit une courbe lisse et non une ligne polygonale. C'est même plus compliqué : **Python**<sup>7</sup> ne trace pas vraiment des segments de droite mais des alignements de pixels le long des segments à tracer, voir [22]. Heureusement, grâce à la mauvaise qualité de notre vision, tout s'arrange.

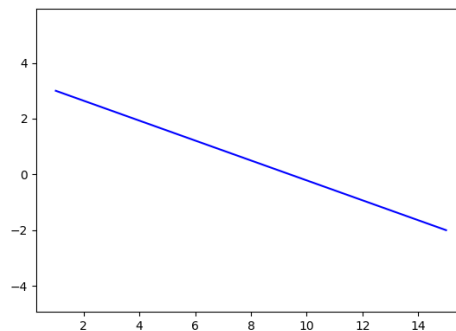
1 - Voici un script simple solution du problème :

```
import numpy as np
import pylab as plt
plt.figure()# Création d'une figure.
x = np.array([1,15])# x est une matrice (n, ).
y = np.array([3,-2])# y idem.
plt.plot(x,y,color='blue')# Le tracé sera bleu.
plt.axis('equal')# Le repère sera orthonormé.
plt.savefig("at34s1so")# La sauvegarde sera le fichier 'at34s1so.png'.
plt.show()# Sinon, la figure n'apparaît pas.
```

### Commentaires :

1 - `x` et `y` sont des 'matrices à une dimension'<sup>8</sup>, notées la plupart du temps 'matrice(n, )' dans ce manuel. On a évidemment envie de les appeler des vecteurs. Dans Python, il existe des matrices à une, deux, *etc* dimensions. La dénomination de matrice à une dimension est très perturbante. `x` ne comprend ici que les abscisses de l'origine et de l'extrémité du segment à tracer, de même `y` avec les ordonnées.

2 - Les commandes `color='blue'`, `axis('equal')` et `savefig("at34s1so")` ne sont pas indispensables.



### Vers le graphe d'une fonction 'quelconque' :

La droite qui porte le segment tracé ayant pour équation  $y = -5/14 * x + 47/14$ , nous allons tracer le graphe de la fonction  $f(x) = -5/14 * x + 47/14$  entre ses points d'abscisses 1 et 15. Nous faisons donc le même

6. qui a pour but de fournir des modèles de représentations graphiques

7. ainsi que tous les autres logiciels de calcul

8. le nombre de leurs éléments

travail que précédemment mais nous sommes maintenant dans un cadre beaucoup plus général :

```
import numpy as np
import pylab as plt
plt.figure()# Création d'une figure.
def f(x):
    return -5/14*x+47/14# Définition d'une fonction.
x = np.linspace(1,15,2)# x —> np.array([1,15]).
y = f(x)# y —> np.array([3,-2]). La fonction f est vectorisée.
plt.plot(x,y,color='blue')# Le tracé sera bleu.
plt.axis('equal')# Le repère sera orthonormé.
plt.savefig("at34s2so")# —> fichier at34s2so.png
plt.show()# Pour montrer la figure.
```

Ce script est un modèle à retenir. Il va pouvoir s'adapter à la représentation graphique de fonctions quelconques.

#### Précisions supplémentaires sur le script ci-dessus :

- On constate que  $f$  est définie par les instructions

```
def f(x):
    return -5/14*x+47/14
```

- La commande `linspace(a,b,n)` du module `numpy` produit  $n$  nombres dont  $a$  et  $b$ , qui partagent l'intervalle  $[a,b]$  en  $n-1$  intervalles de même longueur<sup>9</sup>. Elle est constamment utilisée quand on fabrique un graphe.
- Si  $x=\text{linspace}(a,b,n)$ ,  $f(x)$  produit les valeurs de  $f$  aux  $n$  points de  $x$  sous la forme d'une matrice à une dimension<sup>10</sup>.

#### Vers une figure de meilleure qualité (voir [20])

La figure obtenue ci-dessus est très sommaire. On peut repérer les abscisses sur la graduation du bas, les ordonnées sur la graduation de gauche, mais les axes de coordonnées n'apparaissent pas. Pour corriger cela, on peut effacer les bords du cadre à droite et en haut et faire glisser verticalement l'axe gradué du bas et horizontalement l'axe gradué de gauche pour les faire passer par le point  $(0,0)$ . On obtient ainsi les axes de coordonnées. On place aussi les graduations sous l'axe des abscisses et à gauche de l'axe des ordonnées. Enfin, on fait varier  $x$  de  $-0.5$  à  $15.5$ . On obtient le script suivant, qui améliore le précédent :

```
import numpy as np
import pylab as plt

plt.figure()# paramètres par défaut : graphique de 8x6 pouces
# avec une résolution de 80 points par pouce.

def f(x):
    return -5/14*x+47/14

x = np.linspace(1,15,2)# x est une matrice à une dimension.
y = f(x)# y est une matrice à une dimension.

plt.plot(x,y,color='blue')# Le tracé sera bleu.

plt.axis('equal')# Le repère sera orthonormé.
plt.xlim(-0.5, 15.5)# Limites de variation de x.
```

9. Par exemple `linspace(1,15,2)` produit `array([1,15])`.

10. On dit que  $f$  est une fonction vectorisée.

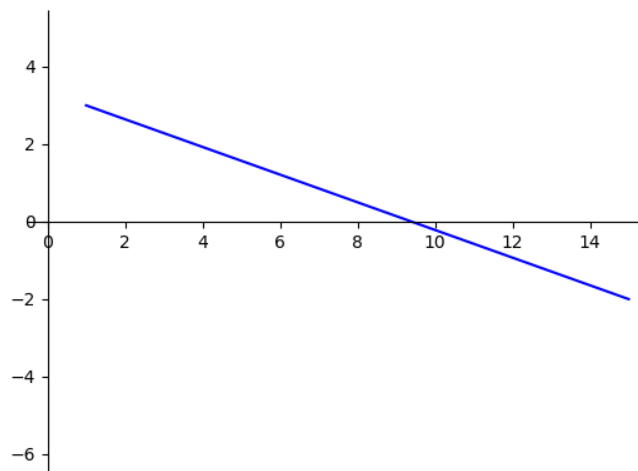
```
plt.ylim(-3.5, 2.5)# Limites de variation de y.

ax = plt.gca()# Accès aux outils de pylab pour les axes.
ax.spines[ 'right' ].set_color( 'none' )
ax.spines[ 'top' ].set_color( 'none' )
ax.xaxis.set_ticks_position( 'bottom' )
ax.spines[ 'bottom' ].set_position(( 'data',0))
ax.yaxis.set_ticks_position( 'left' )
ax.spines[ 'left' ].set_position(( 'data',0))

plt.savefig("at34s5so")

plt.show()
```

Voilà ce que l'on obtient :



Cette figure est plus satisfaisante que la précédente.

**2** - Il suffit de remplacer, dans le script précédent, la fonction **f** par **sin**. Voici successivement le script utilisé et le graphe obtenu :

```
import numpy as np
import pylab as plt

plt.figure()

x = np.linspace(1,15,100)
y = np.sin(x)
plt.plot(x,y,color='blue')

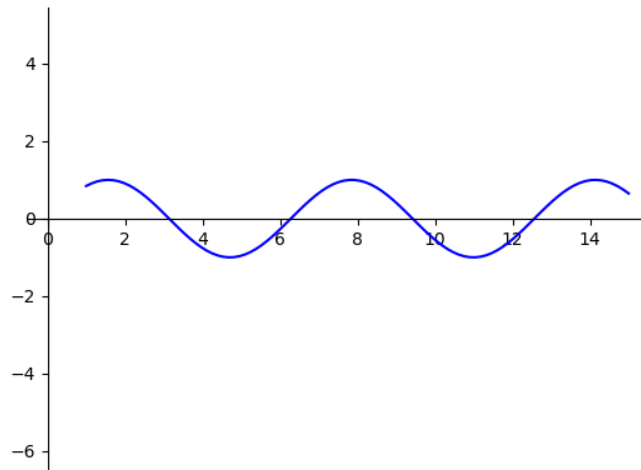
plt.axis('equal')

plt.xlim(-0.5, 15.5)
plt.ylim(-3.5, 2.5)

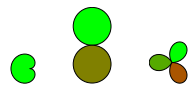
ax = plt.gca()
ax.spines[ 'right' ].set_color( 'none' )
ax.spines[ 'top' ].set_color( 'none' )
```

```
ax.xaxis.set_ticks_position('bottom')
ax.spines['bottom'].set_position(('data',0))
ax.yaxis.set_ticks_position('left')
ax.spines['left'].set_position(('data',0))

plt.savefig("at34s3so")
plt.show()
```



Bien sûr, on fera mieux plus tard.



## 2.3 [\*] Représentations graphiques avec Python

Pour tracer un graphique, Python en calcule des points consécutifs et les relie par un segment. Habituellement, on obtient ainsi une bonne approximation de la courbe à tracer quand il y a suffisamment de points, car alors nos yeux ne font plus la différence.

On se propose d'étudier des approximations du graphe de la fonction  $\sin()$  lorsque la variable  $x$  varie sur l'intervalle  $[0, \pi]$ .

1 - Tracer l'approximation du graphe obtenue en utilisant seulement ses points d'abscisse  $0$ ,  $\pi/2$  et  $\pi$ .

2 - Pour  $n$  valant successivement 3, 6, 10, 20, tracer l'approximation du graphe obtenue en calculant les points dont les abscisses sont données par la commande `linspace(0,pi,n+1)` <sup>a</sup>.

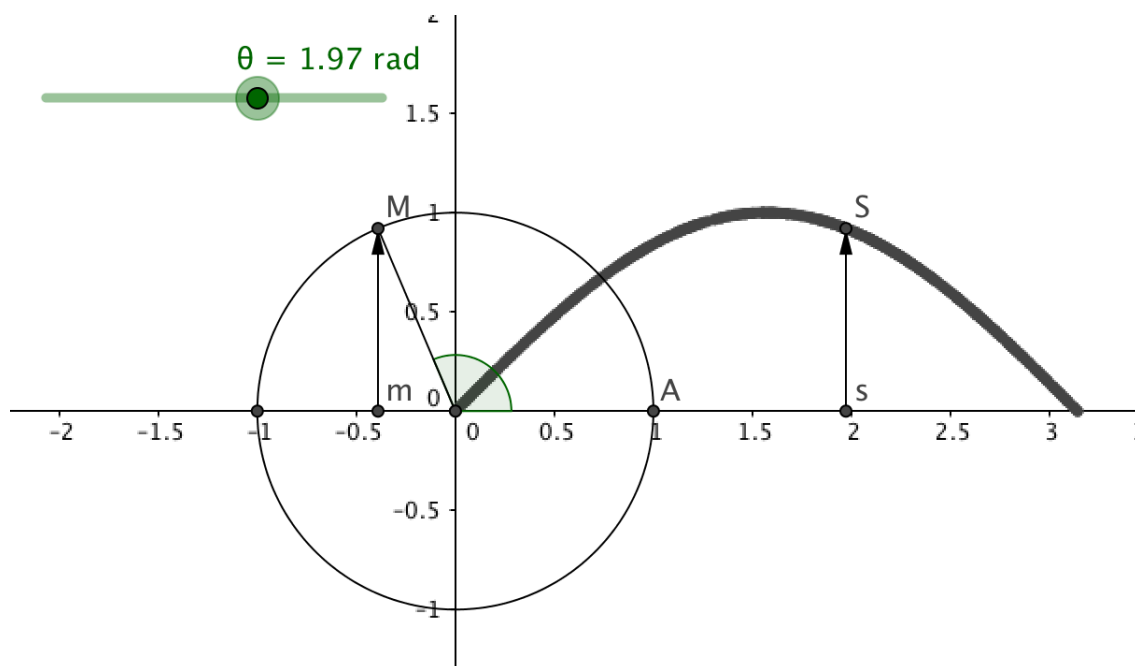
a. Cette représentation graphique est une ligne polygonale à  $n$  côtés.

**Mots-clefs :** Représentation graphique d'une fonction, repère orthonormé (commande `axis('equal')`), autres commandes graphiques, fonctions vectorisées.

Bien sûr, il est impossible de représenter graphiquement une fonction parce que, sauf cas particulier, une représentation graphique a une infinité de points <sup>11</sup>. C'est pourquoi les logiciels de calcul se contentent de représenter ces graphiques par des lignes polygonales <sup>12</sup>. Cet exercice peut être considéré comme une suite de l'exercice 2.2.

### Avant de commencer

On peut représenter simplement une sinusoïde à l'aide d'un logiciel de géométrie dynamique <sup>13</sup>. Le point  $M$  est repéré sur le cercle trigonométrique par l'angle de vecteurs  $\theta = (\overrightarrow{OA}, \overrightarrow{OM})$ . Les coordonnées de  $M$  sont  $(\cos \theta, \sin \theta)$ . Le point  $S$  de coordonnées  $(\theta, \sin \theta)$  engendre la sinusoïde quand  $\theta$  parcourt le segment  $[0, \pi]$ .



### Solution

1 - On peut utiliser le script suivant :

- 
- 11. Un ordinateur ne peut exécuter une infinité de commandes.
  - 12. dont les côtés sont eux-mêmes des successions de pixels, ce qui n'apparaîtra pas dans cet exercice
  - 13. L'image ci-dessous a été créée à l'aide de GeoGebra.

```

from math import pi
import numpy as np
import pylab as plt

plt.figure()

x = np.array([0, pi/2, pi]) # ou x = np.linspace(0, pi, 3).
y = np.sin(x)
plt.plot(x, y)

xx = np.linspace(0, pi, 100)
yy = np.sin(xx)
plt.plot(xx, yy)

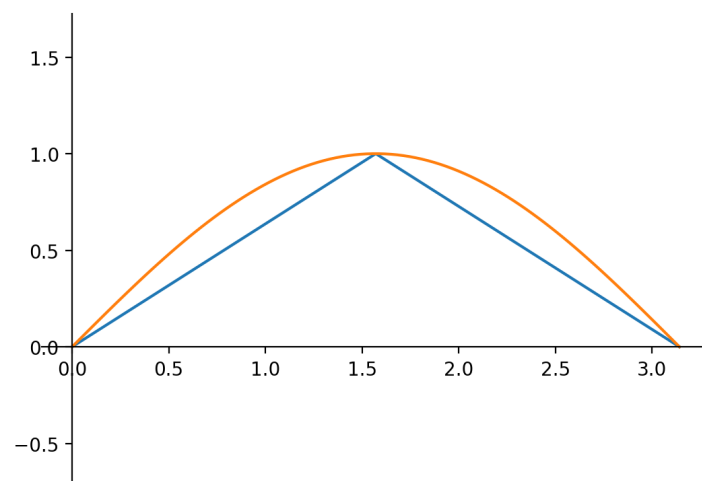
plt.axis('equal')

ax = plt.gca()
ax.spines['right'].set_color('none')
ax.spines['top'].set_color('none')
ax.spines['bottom'].set_position(('data', 0))
ax.spines['left'].set_position(('data', 0))
ax.xaxis.set_ticks_position('bottom')
ax.yaxis.set_ticks_position('left')

plt.savefig('atls1f1')
plt.show()

```

Les calculs de sinus ci-dessus se font non pas avec la fonction `sinus()` du module `math` mais avec la fonction `np.sinus()` qui est la fonction sinus vectorisée, c'est à dire la fonction sinus du module `numpy`, voir les remarques sur les fonctions vectorisées à la fin. Ce script<sup>14</sup> est, parmi les approximations raisonnables de la sinusoïde, la plus simple possible.



Mais ce n'en est pas une bonne approximation : la ligne polygonale bleue ne fait pas vraiment penser à la sinusoïde. Néanmoins, ce n'est pas franchement mauvais.

---

14. Pour tout ce qui concerne le module `matplotlib`, voir [20]. Python dispose d'autres modules graphiques que nous n'utilisons pas.

2 - Le script suivant définit une fonction appelée `sinpolygone(n)` qui retourne comme ci-dessus deux tracés sur la même fenêtre graphique : la représentation graphique de `sin` ainsi que la ligne polygonale  $(0,0) \rightarrow (\pi/n, \sin(\pi/n)) \rightarrow (2\pi/n, \sin(2\pi/n)) \rightarrow \dots \rightarrow (\pi, \sin(\pi))$  qui est censée en être une approximation.

```
from math import pi
import numpy as np
import pylab as plt

def sinpolygone(n): # n est le nombre de côtés de la ligne polygonale.
    plt.figure()

    x=np.linspace(0,pi,n+1)
    y=np.sin(x)
    plt.plot(x,y,color='green',label="sinpolygone")

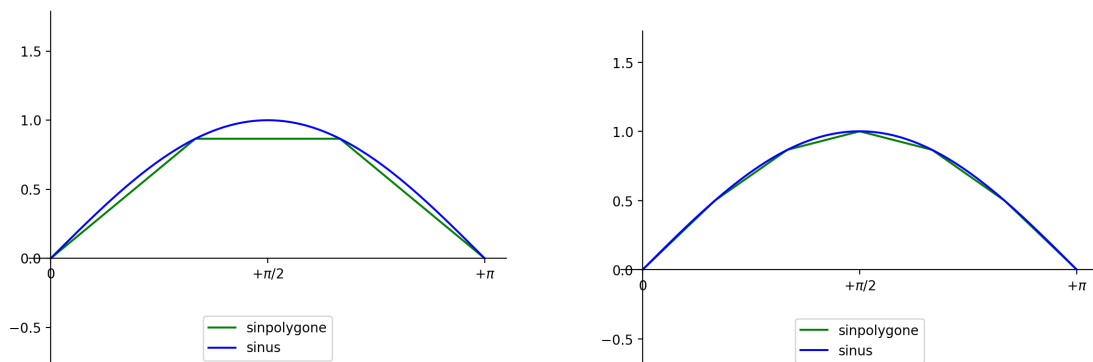
    xx=np.linspace(0,pi,100)
    yy=np.sin(xx)
    plt.plot(xx,yy,color='blue',label="sinus")

    plt.axis('equal')

    ax = plt.gca()
    ax.spines['right'].set_color('none')
    ax.spines['top'].set_color('none')
    ax.xaxis.set_ticks_position('bottom')
    ax.spines['bottom'].set_position(('data',0))
    ax.yaxis.set_ticks_position('left')
    ax.spines['left'].set_position(('data',0))

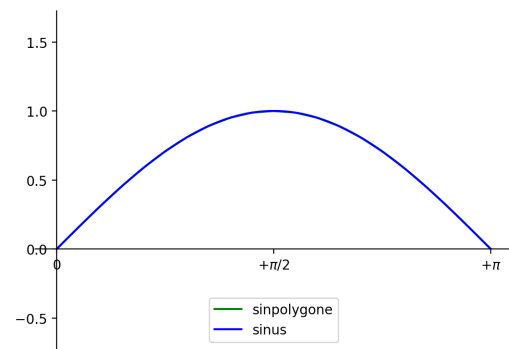
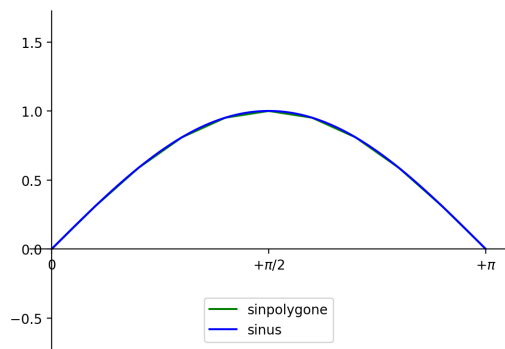
    plt.legend(loc='upper_left')
    plt.savefig('at1s4f1')
    plt.show()
```

Nous avons regroupé ci-dessous en un tableau à 2 lignes et 2 colonnes l'approximation de la sinusoïde obtenue pour  $n = 3$  (3 côtés), puis  $n = 6, 10, 20$  (de gauche à droite et de haut en bas), en tapant `sinpolygone(3)`, *etc*, dans la console interactive <sup>15</sup>.

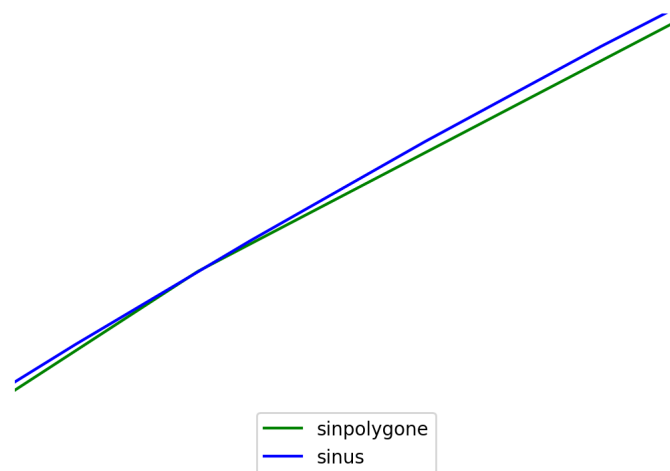


15. Les tracés ont été disposés à l'aide de **Latex**.





On constate que pour  $n=20$ , on ne distingue plus la sinusoïde de la ligne polygonale. C'est évidemment une impression fautive : voici un agrandissement d'une partie de ce dernier tracé (capture d'écran) :



### Compléments sur les tableaux de graphes :

Nous aurions pu faire le tableau des 4 graphes à l'aide de la commande `subplot` de `matplotlib`, par exemple avec le script suivant :

```
from math import pi
import numpy as np
import pylab as plt

plt.figure()

A=np.array([3,6,10,20])
l=['green','red','magenta','blue']

for i in range(1,5):
    xx=np.linspace(0,pi,100)
    yy=np.sin(xx)
    plt.subplot(2,2,i)
    plt.plot(xx,yy,color='black')
    x=np.linspace(0,pi,A[i-1]+1)
    y=np.sin(x)
    plt.plot(x,y,color=l[i-1],label="Approximation du sinus")
```

```
plt.axis('equal')

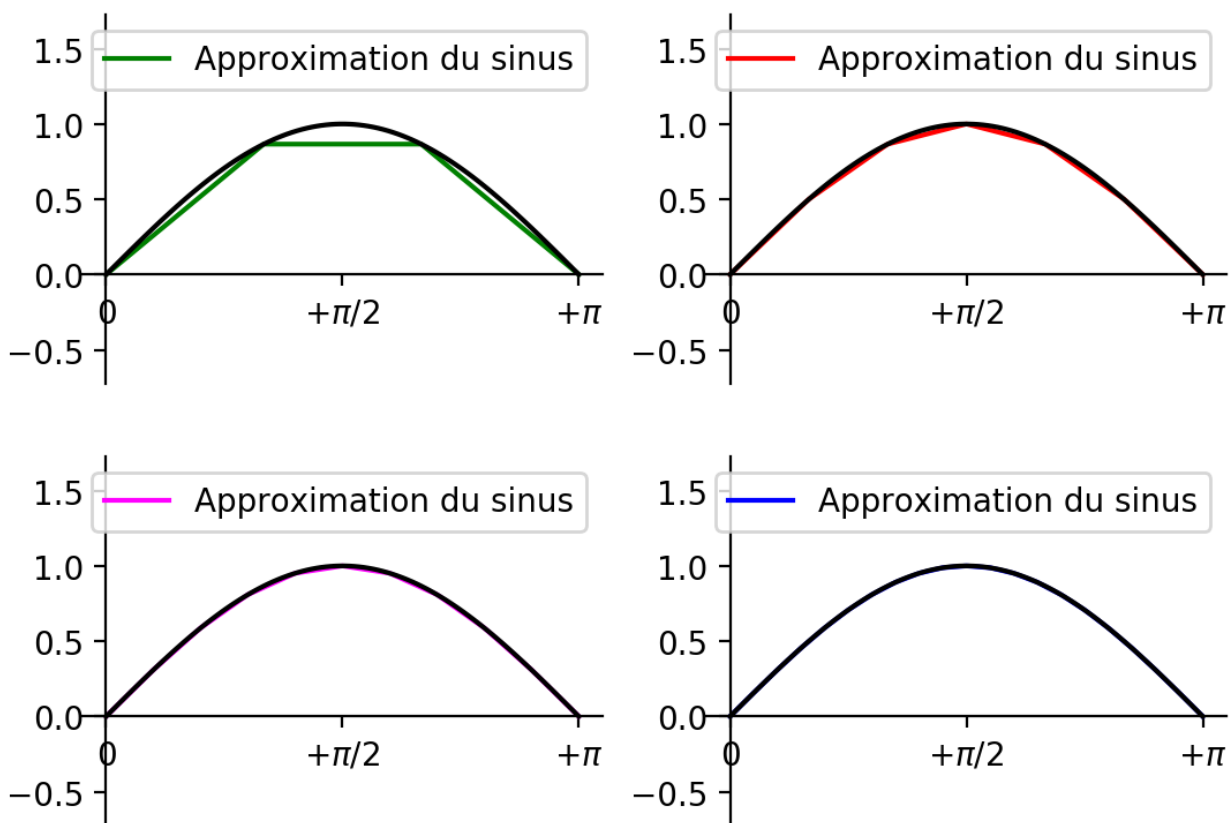
plt.xticks([0, pi/2, pi], [r'$0$', r'$+\pi/2$', r'$+\pi$'])

ax = plt.gca()
ax.spines['right'].set_color('none')
ax.spines['top'].set_color('none')
ax.spines['bottom'].set_position(('data',0))
ax.spines['left'].set_position(('data',0))
ax.xaxis.set_ticks_position('bottom')
ax.yaxis.set_ticks_position('left')

plt.legend(loc = 'upper_right')
plt.show()

plt.savefig('atls2f1')
```

Voici le tableau de tracés obtenu en exécutant le script précédent :



#### Remarques sur les fonctions vectorisées :

Voir les exemples commentés ci-dessous :

```
from math import (pi, sin)
sin(pi/4, 2*pi/3)
Message d'erreur % On a tenté ici de prendre les sinus des éléments
```

```

% d'un 2-uplet. Échec car sin est une fonction d'une variable.

sin([pi/4, 2*pi/3])
Message d'erreur % On a tenté ici de prendre les sinus des éléments
% d'une liste de 2 éléments. Échec car sin est une fonction d'une variable.

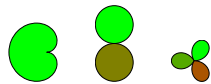
from numpy import array
sin(array([pi/4, 2*pi/3]))
Message d'erreur % On a tenté ici de prendre les sinus des éléments
% d'une matrice-ligne à 2 éléments. Échec car sin est
% une fonction d'une variable.

import numpy as np
% On utilise maintenant la fonction sinus vectorisée.
% Tentons de prendre les sinus des éléments d'une matrice-ligne à 2 éléments.
np.sin(array([pi/4, 2*pi/3]))
Out[9]: array([0.70710678, 0.8660254 ])
% On obtient une matrice-ligne à 2 éléments.

np.sin([pi/4, 2*pi/3])
Out[10]: array([0.70710678, 0.8660254 ])
% On peut prendre le sinus des éléments d'une liste. On obtient
% une matrice-ligne à 2 éléments.

% On ne peut pas prendre le sinus des éléments d'un t-uplet.

```



## 2.4 [\*] Courbes définies paramétriquement

Les représentations graphiques des fonctions sont un cas particulier de courbes définies paramétriquement : si  $f$  et  $g$  sont deux fonctions définies sur un même intervalle  $I$ , le point  $M(t)$  de coordonnées  $x=f(t)$  et  $y=g(t)$  se déplace et engendre une courbe  $(C)$ , quand  $t$  décrit l'intervalle  $I$ . On peut de cette manière tracer de nombreuses courbes qui ne sont pas des représentations graphiques de fonctions.

À l'aide du module `matplotlib` de Python, tracer la courbe  $(C)$  d'équations paramétriques  $x = \cos(3t)$ ,  $y = \sin(2t)$ .

**Mots-clefs :** Tracés de courbes, `matplotlib`.

### Solution

Cet exercice n'est pas du tout hors sujet. Les tracés demandés se font exactement comme les représentations graphiques usuelles (voir l'exercice 2.2). C'est l'occasion d'introduire des courbes nouvelles mignonnes comme celle que propose l'énoncé.

On obtient cette courbe en faisant varier  $t$  dans l'intervalle  $[-\pi, \pi]$ <sup>16</sup>. En faisant varier  $t$  dans l'intervalle  $[-\pi, \pi]$ , le point  $(-1, 0)$  est obtenu deux fois<sup>17, 18</sup>. On utilisera le script standard suivant :

```
from math import pi
import numpy as np
import pylab as plt

def fantaisiel():
    plt.figure()

    t = np.linspace(-pi, pi, 1001)
    x = np.cos(3*t)
    y = np.sin(2*t)

    plt.plot(x, y, color='blue')

    plt.axis('equal')

    ax = plt.gca()
    ax.spines['right'].set_color('none')
    ax.spines['top'].set_color('none')
    ax.xaxis.set_ticks_position('bottom')
    ax.spines['bottom'].set_position(('data', 0))
    ax.yaxis.set_ticks_position('left')
    ax.spines['left'].set_position(('data', 0))

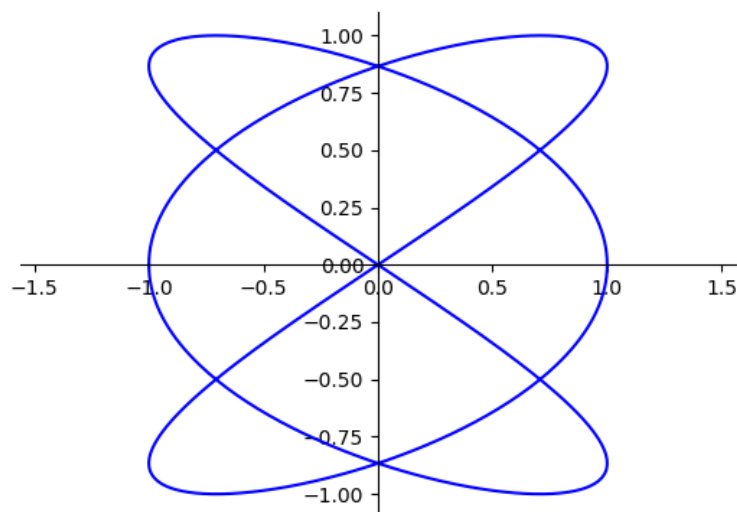
    plt.savefig('carlfig1')
    plt.show()
```

qui produit la figure suivante :

16. parce que les fonctions  $\sin$  et  $\cos$  sont périodiques de période  $2\pi$ .

17. En fait, il suffit de faire varier la variable dans l'intervalle  $[0, \frac{\pi}{2}]$  et d'ajouter à l'arc obtenu ses symétriques par rapport à l'axe des abscisses, puis par rapport à l'axe des ordonnées. Ce genre de considérations est ici à éviter.

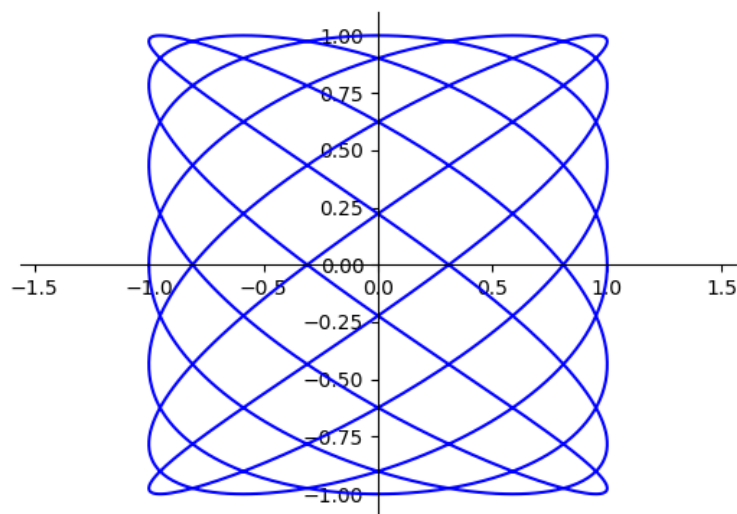
18. On peut aussi faire varier  $t$  dans l'intervalle  $[0, 2\pi]$ . Alors c'est le point  $(1, 0)$  qui est obtenu deux fois.



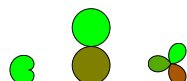
C'est en réalité une ligne polygonale fermée à 1000 côtés.

Remarques et questions faciles :

- Pourquoi cette courbe n'est-elle pas la représentation graphique d'une fonction ?
- Les élèves doivent savoir parcourir ce tracé quand  $t$  croît de  $-\pi$  à  $\pi$ , identifier les points qui correspondent à des valeurs remarquables de  $t$ .
- Ils pourront aussi remplacer les fonctions  $\cos(3t)$  et  $\sin(2t)$  par les fonctions  $\cos(nt)$  et  $\sin(pt)$ ,  $n$  et  $p$  étant des entiers  $> 0$  pouvant varier. Par exemple, pour  $n = 7$  et  $p = 5$ , on obtient :



- Qu'obtient-on quand  $n = p$  ? Combien de fois la courbe est-elle parcourue quand  $t$  décrit l'intervalle  $[-\pi, \pi]$  ? <sup>19</sup>




---

19. On obtient le cercle de centre  $(0, 0)$  et de rayon 1 parcouru  $n$  fois.

## 2.5 [\*] Polygones réguliers

Le plan étant rapporté à un repère orthonormé,  $(\mathcal{C})$  désigne le cercle centré à l'origine des axes et de rayon 1,  $n$  un entier au moins égal à 2.

**1** - [Polygones réguliers convexes] Pour construire un polygone régulier à  $n$  côtés inscrit dans  $(\mathcal{C})$ , on place successivement les points  $A_k$ ,  $k = 0, 1, \dots, n$  de coordonnées  $(\cos(k \cdot \frac{2\pi}{n}), \sin(k \cdot \frac{2\pi}{n}))$ <sup>a</sup> et on joint chaque point au suivant par un segment de droite. On obtient un polygone à  $n$  côtés<sup>b</sup>.

**1.a** - Programmer une fonction `n --> polygone(n)` qui pour tout entier  $n \geq 2$  donne le polygone à  $n$  côtés décrit ci-dessus<sup>c</sup>.

**1.b** - Décrire les polygones `polygone(3)` et `polygone(4)`.

**2** - [Polygones réguliers convexes ou étoilés] Soit  $n$  un entier au moins égal à 3 et  $r$  un entier vérifiant  $1 \leq r \leq n - 1$ . On recommence le tracé décrit à la question précédente en remplaçant l'angle  $\frac{2\pi}{n}$  par l'angle  $\frac{2\pi r}{n}$ . On obtient évidemment une ligne polygonale fermée, notée `polygone2(n,r)` ci-dessous.

**2.a** - Cas  $n = 3$  : décrivez `polygone2(3,1)` et `polygone2(3,2)`.

**2.b** - [Tracés] Programmer une fonction qui, étant donné  $n$  et  $r$ , retourne les polygones `polygone2(n,1)`, ..., `polygone2(n,n-1)`.

**2.c** - À titre d'exemple, tracer les polygones obtenus pour  $n = 9$  puis  $n = 10$ .

a. Il est clair que  $A_0 = A_n$

b. appelé polygone régulier convexe

c. `polygone(2)` se réduit évidemment au segment  $[-1, 1]$  - un diamètre - de l'axe des abscisses.

**Mots-clefs** : tracés avec Python, cercle trigonométrique, fonctions sinus et cosinus, modules `math`, `numpy`, `matplotlib`.

Cet exercice illustre et utilise le fait que Python ne sait tracer que des lignes polygonales. On profite ici de ce défaut.

### Solution

**1** - Comme  $n = 2$  conduit à un cas dégénéré, nous supposons dorénavant que  $n \geq 3$ .

**1.a** - Le script suivant suffit :

```
from math import pi
import numpy as np
import pylab as plt
def polygone(n):
    theta = np.linspace(0, 2*pi, n+1)
    x = np.cos(theta)
    y = np.sin(theta)
    plt.figure()# Création d'une figure.
    plt.axis('equal')# Repère orthonormé.
    plt.plot(x,y,color='blue')# Le polygone sera bleu.
    a = plt.show()# Montrer la figure.
    return a
```

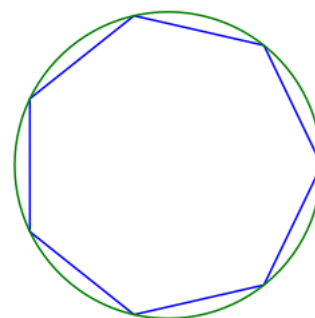
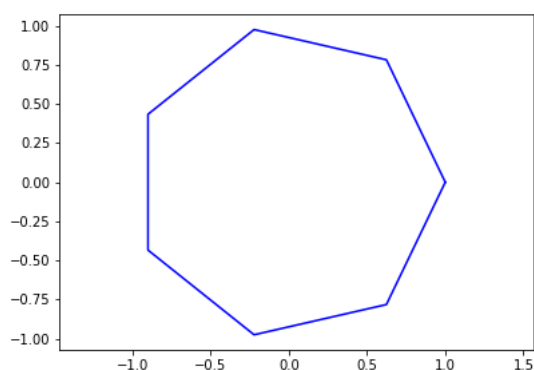
On peut préférer un tracé plus joli. La fonction `polygone()` modifiée ci-dessous trace en plus en vert le `polygone(100)` que notre vue ne peut distinguer du cercle  $(\mathcal{C})$  et efface le cadre et ses graduations. On pourrait vérifier que le soi-disant cercle  $(\mathcal{C})$  est bien une ligne polygonale à l'aide de plusieurs grossissements.

```

from math import pi
import numpy as np
import pylab as plt
def polygone(n) :
    theta = np.linspace(0,2*pi,n+1)
    x = np.cos(theta)
    y = np.sin(theta)
    eta = np.linspace(0,2*pi,101)# Polygone de 100 côtés identifié à (C).
    xx = np.cos(eta)
    yy = np.sin(eta)
    plt.figure()# Création d'une figure.
    plt.axis('equal')# Repère orthonormé.
    plt.plot(x,y,color='blue')# Le polygone sera bleu.
    plt.plot(xx,yy,color='green')# Le cercle sera vert.
    xticks([])# Pas de graduation en abscisses.
    yticks([])# Pas de graduation en ordonnées.
    ax = gca()
    ax.spines['right'].set_color('none')# Fait disparaître le
# bord droit du cadre.
    ax.spines['left'].set_color('none')# Fait disparaître le
# bord gauche du cadre.
    ax.spines['top'].set_color('none')# Fait disparaître le
# bord supérieur du cadre.
    ax.spines['bottom'].set_color('none')# Fait disparaître le
# bord inférieur du cadre.
    a = plt.show()# Montrer la figure.
    return a

```

Les instructions de tracé de courbes sont empoisonnantes. Il est conseillé de les réunir toutes dans un fichier commenté dans lequel on pourra piocher suivant les besoins. Voici les tracés obtenus pour  $n = 7$  avec le premier état, puis l'état définitif de la fonction `polygone()` :



**1.b** - `polygone(3)` est le triangle équilatéral de sommet  $(1,0)$  inscrit dans  $(C)$ , `polygone(4)` le carré de sommet  $(1,0)$  inscrit dans  $(C)$ .

**2** - Comme précédemment,  $A_n = A_0$ . Le polygone ainsi obtenu est noté ci-dessous `polygone2(n,r)`. Il est possible, selon le cas, que l'on revienne plus tôt en  $A_0$ .

**2.a** - `polygone2(3,1)` est le triangle équilatéral de sommet  $(1,0)$  inscrit dans  $(C)$ , de même `polygone2(3,2)`, mais la construction est différente : le premier est décrit dans le sens trigonométrique, le second dans le sens inverse.

**2.b** - Voici une fonction qui convient :  $(n,p,q) \longrightarrow \text{polygone2b}(n,p,q)$  produit les `polygone(n,1)`, ...,

polygone(n,n-1) disposés en un tableau à p lignes et q colonnes<sup>20, 21</sup> (p et q désignant 2 entiers >0 tels que p.q = n-1) et sauvegarde ce tableau.

```
from math import pi
import numpy as np
import pylab as plt

def polygone2b(n,p,q):# p et q désignent 2 entiers >0 tels que p.q = n-1.
    plt.figure(figsize=(8,8), dpi=80)# Création d'une figure.
    for r in range(1,n):
        subplot(p,q,r)
        plt.axis('equal')# Repère orthonormé.
        theta = np.linspace(0,2*pi,n+1)
        x = np.cos(theta*r)
        y = np.sin(theta*r)
        eta = np.linspace(0,2*pi,101)
        # Un polygone à 100 côtés suffit pour représenter (C).
        xx = np.cos(eta)
        yy = np.sin(eta)
        ax = gca()
        ax.spines['right'].set_color('none')# Fait disparaître le
        # bord droit du cadre.
        ax.spines['left'].set_color('none')# Fait disparaître le
        # bord gauche du cadre.
        ax.spines['top'].set_color('none')# Fait disparaître le
        # bord supérieur du cadre.
        ax.spines['bottom'].set_color('none')# Fait disparaître le
        # bord inférieur du cadre.
        plt.plot(x,y,color='blue')# Le polygone sera bleu.
        plt.plot(xx,yy,color='green')# Le cercle sera vert.
        xticks([])# Pas de graduation en abscisses.
        yticks([])# Pas de graduation en ordonnées.

    a = plt.show()# Montrer la figure.
    savefig("polygone2b1(n,p,q)")
    return a
```

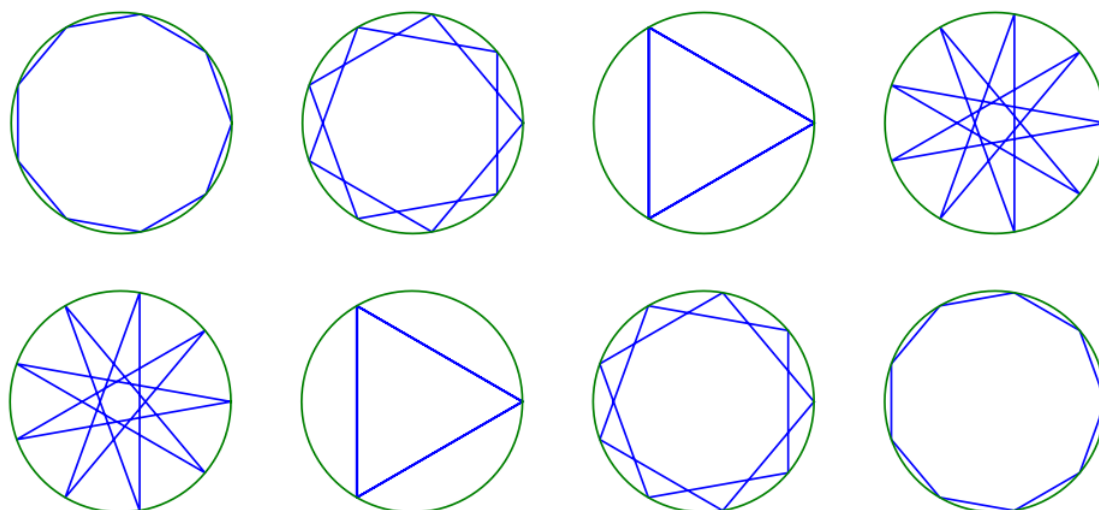
---

20. Cela complique le script, en faisant intervenir les commandes `subplot(a,b,c)`. On peut aussi le simplifier en produisant les polygones un par un.

21. En supprimant les commandes d'effacement des axes et des graduations, on allège considérablement ce script.



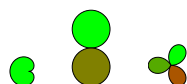
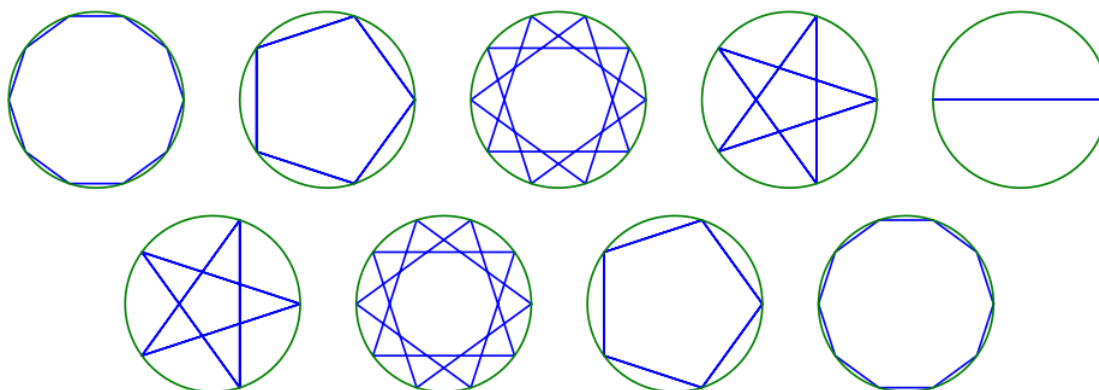
**2.c** - Voici le tableau de polygones obtenu pour  $n = 9$  :



On constate que

- $\text{polynome2b}(9,2)=\text{polynome2b}(9,7)$
- $\text{polynome2b}(9,4)=\text{polynome2b}(9,5)$
- $\text{polynome2b}(9,3)$  est un triangle équilatéral.

Voici le tableau des polygones obtenu pour  $n = 10$  :



## 2.6 [\*] Détermination graphique de $\sqrt{2}$

Déterminer une valeur approchée de  $\sqrt{2}$  comme l'abscisse de l'intersection du graphe de la fonction  $x \rightarrow x^2$ ,  $x$  variant de 0 à 2 et du segment de droite parallèle à l'axe des abscisses d'ordonnée 2 obtenu de même en faisant varier  $x$  de 0 à 2. Plus précisément,

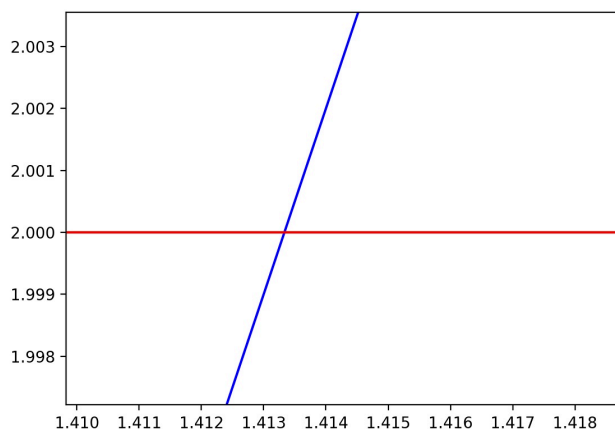
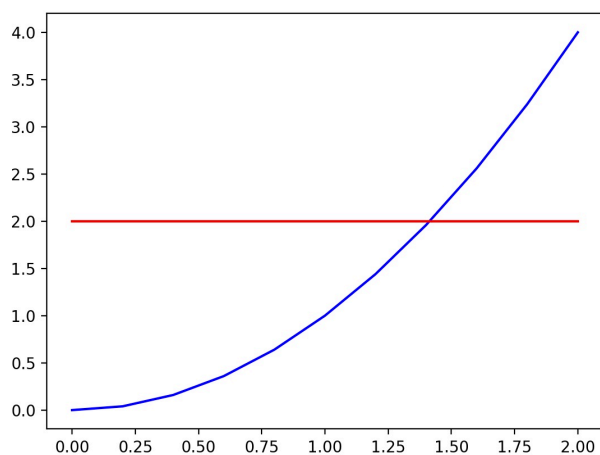
- 1 - déterminer une telle valeur approchée quand on approxime le graphe de  $x \rightarrow x^2$  par une ligne polygonale à  $n = 10$  côtés (voir les exercices 2.1 ou 2.3),
- 2 - puis quand on l'approxime par une ligne polygonale à  $n = 1000$  côtés.

**Mots-clefs :** Interface pylab de matplotlib.

**Solution :** Comme on n'a pas besoin d'un repère orthonormé, on peut laisser faire Python. Le script ci-dessous donne des tracés peu sophistiqués mais suffisants :

```
import numpy as np
import pylab as plt
plt.figure()# Création d'une figure.
n = eval(input('n= '))# Nombre de segments de la ligne polygonale
# qui représentera le graphe de  $x \rightarrow x^2$  pour  $0 \leq x \leq 2$ .
x = np.linspace(0,2,n+1)# x est une matrice (n, ).
y = x*x# y est une matrice (n, ).
plt.plot(x,y,color='blue')# Le tracé sera bleu.
x1 = np.array([0,2])
y1 = np.array([2,2])
plt.plot(x1,y1,color='red')# Segment de droite horizontale d'ordonnée 2.
plt.show()# Montrer la figure.
```

Quand on exécute ce script, on répond 10 à la question  $n =$ <sup>22</sup> sur la console interactive. On obtient ainsi la première fenêtre graphique ci-dessous. Elle contient le graphe ( $G$ ) de la fonction  $x \rightarrow x^2$ , le segment de droite horizontale d'ordonnée 2 ainsi que des graduations sur deux axes du cadre. On peut deviner (au début) que c'est bien une ligne polygonale, donc une approximation relativement peu précise de ( $G$ ). Lire l'abscisse du point A<sup>23</sup> intersection de ( $G$ ) et du segment rouge est malaisé. On peut juste dire, à vue, que  $\sqrt{2}$  se trouve entre 1.25 et 1.5. En s'aidant du pointeur placé juste sur A<sup>24</sup>, on peut proposer 4.4125<sup>25</sup>.

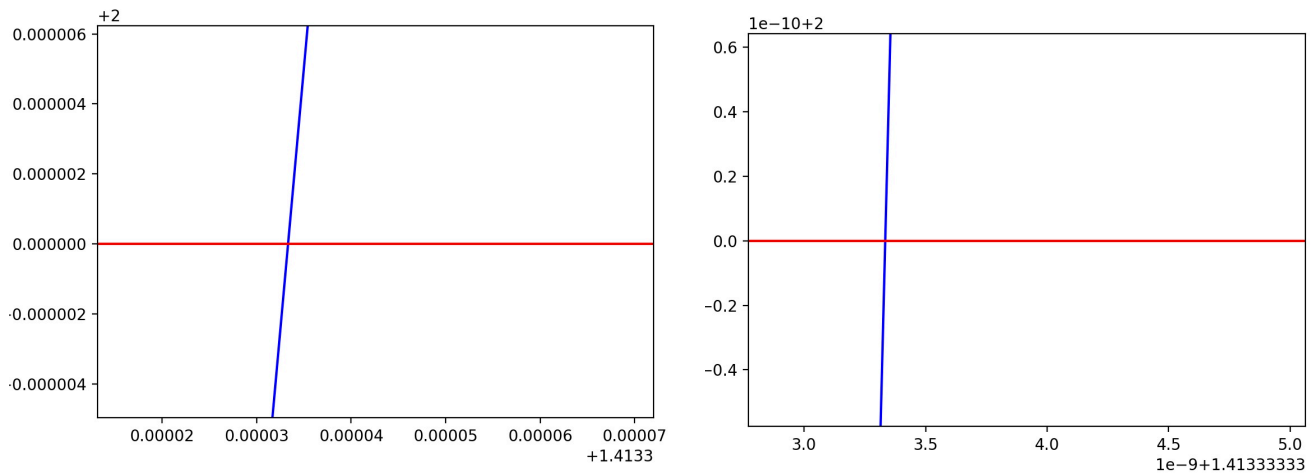


22. Commande `n = eval(input('n = '))`.

23. Le point A n'est pas noté sur la figure, mais c'est possible.

24. Les coordonnées du pointeur s'affichent en bas de la fenêtre graphique.

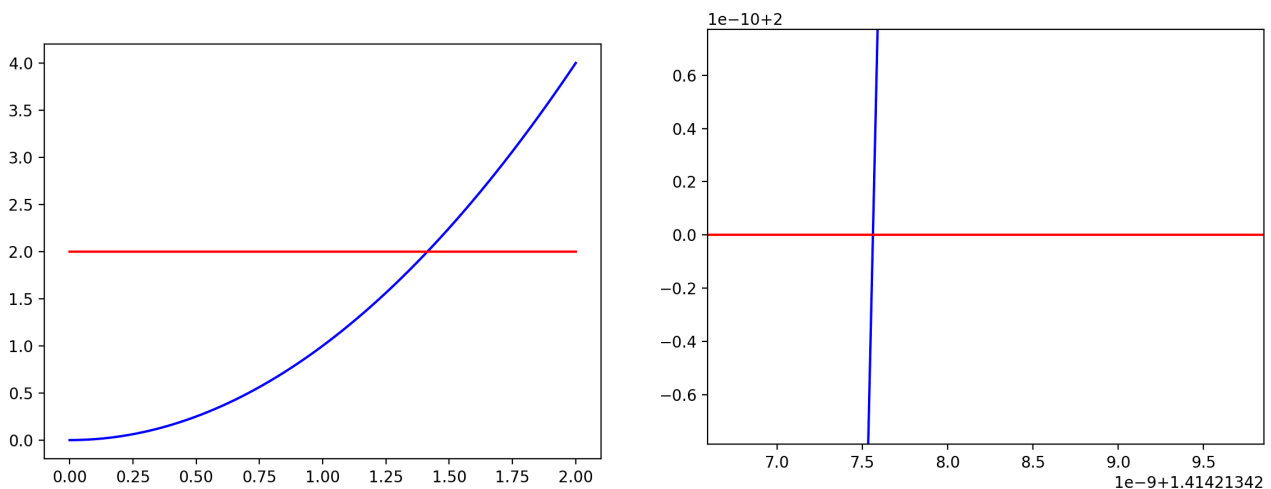
25. On pourrait s'arrêter ici et passer au cas  $n = 1000$  en procédant de même. Cela suffirait pour montrer que la qualité de l'approximation de  $\sqrt{2}$  dépend beaucoup de la qualité de l'approximation de ( $G$ ) et terminerait l'exercice. Ce serait suffisant, très rapide et éviterait toutes les explications oiseuses qui suivent.



Grossissons ce qui nous intéresse. Pour cela, il suffit d'activer le grossissement (cliquer sur l'icône de la loupe de la fenêtre graphique puis sélectionner la zone à grossir). On peut répéter cela plusieurs fois en gardant le grossissement activé. Cela donne de nouvelles fenêtres graphiques comme celles que nous avons reproduites ci-dessus. La portion de  $(G)$  que l'on voit (en bleu) ressemble de plus en plus à un segment de droite vertical<sup>26</sup>. Grâce à la dernière figure, on peut proposer comme valeur approchée de  $\sqrt{2}$ , à vue, le nombre 1.413333333 (ou n'importe quel nombre très voisin) ou utiliser l'abscisse du pointeur placé en A sur la figure. Cette approximation de  $\sqrt{2}$  n'est pas très bonne. En effet,

```
from math import sqrt
sqrt(2)
Out[6] : 1.4142135623730951
```

2 - Recommençons les mêmes opérations en donnant à  $n$  la valeur 1000. Les mêmes étapes que précédemment conduisent à :



On peut maintenant proposer comme approximation de  $\sqrt{2}$  la valeur 1.41421342755, qui est une bien meilleure approximation de  $\sqrt{2}$  que la précédente.

### Fin de la solution

#### Compléments et commentaires :

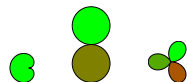
- On peut vouloir travailler en repère orthonormé et peut-être aussi vouloir que le point  $(0,0)$  soit l'origine des axes, *etc.* En fait, outre que c'est du travail inutile<sup>27</sup>, on perd la mise en forme automatique de la fenêtre graphique qui fait ressembler de plus en plus la portion de  $(G)$  obtenue à un segment presque vertical, ce qui facilite considérablement l'évaluation à vue de l'abscisse de A. Aussi, en repère orthonormé, on utilisera

26. Cela se produit si les zones rectangulaires choisies pour le grossissement ont une hauteur petite par rapport à la largeur.

27. `matplotlib` est plutôt complexe.

le pointeur.

- Cet exercice peut être l'occasion de reparler des causes d'erreur : ici,
  - on se sert de notre vue, notamment pour positionner le pointeur,
  - une ligne polygonale a été substituée à  $(G)$ ,
  - les lignes utilisées ont une certaine épaisseur (qui peut être réglée),
  - les calculs sont approchés (même élever une valeur décimale de  $x$  au carré entraîne une erreur d'approximation parce que le calcul est fait en base 2 et que des troncations interviennent).



## 2.7 [\*] Zéros d'une fonction par dichotomie.

Soit  $f$  une fonction usuelle définie sur un intervalle  $[a, b]$  telle que  $f(a) \cdot f(b) < 0$ . On est sûr que  $f$  s'annule au moins une fois entre  $a$  et  $b$  en un point  $p$  que l'on voudrait déterminer, appelé zéro de  $f$ . Comme approximation de  $p$ , on peut proposer le milieu  $m$  de  $[a, b]$ . Dans ce cas, l'erreur d'approximation est majorée par  $\frac{b-a}{2}$ .

On est sûr aussi que l'une des inégalités  $f(a) \cdot f(m) \leq 0$  ou  $f(m) \cdot f(b) < 0$  est vraie, ce qui montre que  $f$  s'annule dans l'intervalle  $]a, m]$  dans le premier cas ou dans l'intervalle  $]m, b[$  dans le deuxième cas. Comme nouvelle approximation de  $p$ , on peut maintenant choisir le milieu de  $[a, m]$  ou de  $[m, b]$  suivant le cas. C'est un progrès car l'erreur d'approximation est maintenant majorée par  $\frac{b-a}{2^2}$ . Et on peut ré-itérer ce procédé.

1 - Étant donné un entier  $n \geq 0$ , programmer une fonction qui retourne une approximation de  $p$  avec une erreur d'approximation majorée par  $10^{-n}$ .

2 - Trouver une approximation de  $\sqrt{2}$  avec une précision de  $10^{-6}$ .

3 - Trouver un zéro de la fonction  $f$  définie sur l'intervalle  $[0, \pi]$  par :

$$f(x) = x - 1 \quad \text{si } x \leq 1, \quad = (x - 1) \sin\left(\frac{1}{x-1}\right) \quad \text{sinon,}$$

à  $10^{-6}$  près.

**Mots-clefs** : zéros d'une fonction, dichotomie, programmation d'une fonction, d'une fonction définie par morceaux, boucle **tant que**, instructions conditionnelles **if ...: ... else: ...**.

Par fonction usuelle, on entend en réalité fonction continue<sup>28</sup> parce que si une fonction continue prend des valeurs opposées aux extrémités d'un intervalle, elle s'annule dans cet intervalle.

### Solution

1 - La fonction `manon()` suivante convient :

```
def manon(f, a, b, n):# la fonction non spécifiée f apparaît  
    # comme un paramètre de la fonction manon().  
    while b - a > 2/(10**n):  
        x = (a + b)/2  
        if f(a)*f(x) <= 0:  
            b = x  
        else:  
            a = x  
    return x
```

2 - On cherche maintenant une approximation de  $\sqrt{2}$  qui est par définition l'unique zéro de la fonction  $f(x) = x^2 - 2$  dans l'intervalle  $[0, 2]$ . Si on calcule sur une console interactive vierge, on commencera par importer la fonction `manon()` et on définira la fonction `f` :

```
from manon import manon  
def f(x):  
    return x**2-2  
a = manon(f, 0, 2, 6)  
print(a)
```

28. Néanmoins, il y a des fonctions usuelles qui ne sont pas continues, les histogrammes par exemple.

L'exécution de ce script conduit à :

```
runfile('Chemin_du_script_précédent')
1.4142131805419922
```

On est sûr que les 5 premières décimales sont exactes. On pourra comparer ce résultat avec celui de l'exercice 2.6.

3 - On admettra que la fonction  $f$  définie dans cette question est continue<sup>29</sup>. Elle a la particularité d'admettre une infinité de zéros, à savoir 1 et toutes les valeurs de  $x$  dans  $]1, \pi]$  qui annulent  $\sin(\frac{1}{x-1})$ , soit les nombres  $1 + \frac{1}{\pi}, 1 + \frac{1}{2 \cdot \pi}, 1 + \frac{1}{3 \cdot \pi}, \text{etc.}$  On peut se demander quel zéro de  $f$  la fonction `manon()` va donner.

Voici le script utilisé. Il comprend la définition d'une fonction par morceaux.

```
from manon import manon
from math import sin
def f(x):
    if x <= 1:
        return x-1
    else:
        return (x-1)*sin(1/(x-1))
a = manon(f,0,2,6)
print(a)
```

Exécution du script :

```
runfile('Chemin_du_script')
0.9999980926513672
```

C'est 1 que l'on approxime. En choisissant  $n = 10$ , on obtient 0.999999998835847 (à  $10^{10}$  près).

### Fin de l'exercice

Compléments : graphe de  $f$

Ce graphe est intéressant :

```
import numpy as np
from math import pi
import pylab as plt

def f(x):
    if x <= 1:
        return x - 1
    else:
        return (x-1)*np.sin(1/(x-1))

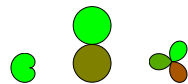
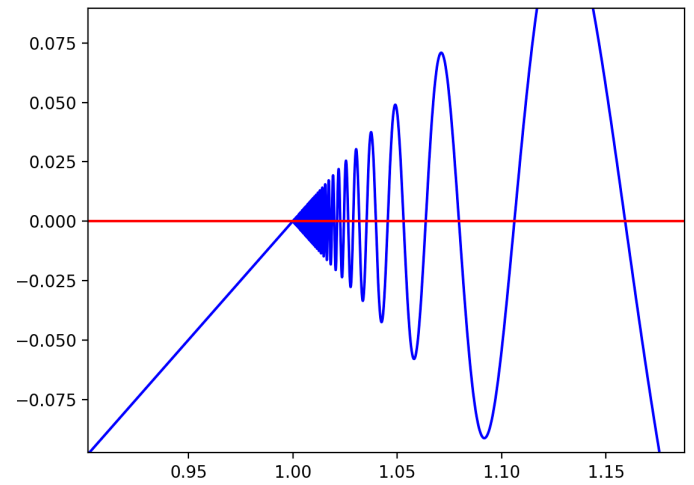
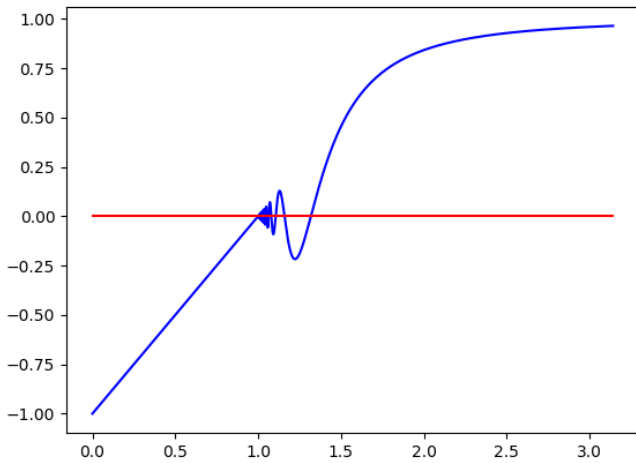
x3a = np.array([0,1])
y3a = np.array([-1,0])# pour le premier morceau de f.
x3b = np.linspace(1+(pi-1)/100000,pi,100000)
y3b = (x3b-1)*np.sin(1/(x3b-1))# pour le second morceau de f.
u3 = np.array([0,pi])
v3 = np.array([0,0])# axe des abscisses.
plt.figure()
```

---

29. Elle est clairement continue dans les intervalles  $[0, 1]$  et  $]1, \pi]$ . La continuité à droite au point 1 résulte de l'inégalité :  $\forall x > 1, \quad |f(x)| \leq x$ .

```
plt.plot(x3a,y3a,color='blue')
plt.plot(x3b,y3b,color='blue')
plt.plot(u3,v3,color='red')
plt.savefig('manonfig')
plt.show()
```

Voici le graphe obtenu ainsi qu'un grossissement au voisinage du point  $(1,0)$  :



## 2.8 [\*] Programmer des fonctions : valeur absolue, volume du cône

- 1 - Programmer une fonction qui retourne la valeur absolue de tout nombre.
- 2 - Programmer une fonction qui retourne le volume de tout cône droit à base circulaire, de hauteur  $h$  cm et de rayon  $r$  cm.

**Mots-clefs** : programmer une fonction, instruction conditionnelle `if: ...else: ...`, importer une constante ou une fonction d'un module de Python.

### Solution

1 - Il est difficile de trouver un exercice plus simple ! Appelons `vabsolue` cette fonction. Nous la définissons par le script :

```
# La fonction vabsolue retourne la valeur absolue de tout nombre x.
def vabsolue(x):
    if x >= 0:
        return x
    else:
        return -x
```

Exemple : calculons la valeur absolue de -2.17 (nombre réel) et de -125 (nombre entier), après avoir chargé `vabsolue` dans la console interactive<sup>30</sup> :

```
runfile('Chemin_de_la_fonction_vabsolue')
# Le chemin détaillé s'inscrit automatiquement quand on exécute le script.
vabsolue(-2.17)
Out[2]: 2.17
vabsolue(-125)
Out[3]: 125
```

On obtient un nombre réel dans le premier cas, un entier dans le second. Bien sûr, pour calculer une valeur absolue, on utilise habituellement la fonction pré-définie `abs(x)`, voir [13].

2 - La formule qui donne le volume  $v$  du cône, ici en  $\text{cm}^3$ , est  $v = \frac{\pi r^2 h}{3}$ . Programmer une fonction de deux variables ne pose pas de problème :

```
# La fonction volcone retourne le volume d'un cône droit
# à base circulaire de rayon r cm et de hauteur h cm.
from math import pi
def volcone(r,h):
    return (pi*r**2*h)/3
```

Remarquons que l'on a chargé la valeur de la constante  $\pi$  du module `math`<sup>31</sup>.

Exemple :

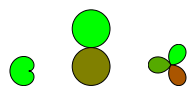
```
from math import pi
runfile('Chemin_de_la_fonction_volcone')
volcone(1.88,3.17)
Out[3]: 11.732851629089138
```

30. Nous supprimons systématiquement les lignes blanches pour gagner de la place.

31. On aurait pu importer le module `math` tout entier par `from math import *`.



```
pi*1.88*1.88*3.17/3 # Vérification (inutile) par calcul direct.  
Out[6] : 11.732851629089138
```



## 2.9 [\*] Trinôme du second degré

Résoudre l'équation du second degré (E) :  $ax^2 + bx + c = 0$  ( $a \neq 0$ )

**Mots-clefs :** Fonction `sqrt()`, à importer du module `math`, sortie d'une fonction par `return` ou `print()`, instruction conditionnelle `if ...:...elif ...:...else:...`

### Solution

La programmation de la solution doit envisager les 3 cas :  $b^2 - 4ac > 0$ ,  $b^2 - 4ac = 0$  et  $b^2 - 4ac < 0$ , d'où l'utilisation de l'instruction conditionnelle `if ...:...elif ...:...else:...`. Les scripts ci-dessous permettront aux professeurs de proposer des trinômes moins banals que les trinômes habituels à coefficients entiers dont la résolution est plus ou moins évidente. Les deux solutions se distinguent par leurs sorties : la fonction `trinome` (instruction `return`) retourne une liste, éventuellement vide<sup>32</sup>, que l'on peut utiliser dans des calculs ultérieurs ; la seconde `trinomebis` (instruction `print()`) donne les solutions de (E), s'il y en a, au moyen d'une phrase en français ; mais il ne semble pas qu'on puisse récupérer ces solutions pour des calculs ultérieurs. La sortie de `trinomebis` est de type `nonetype`.

```
# La fonction trinome ci-dessous retourne la liste des solutions de
# l'équation (E) : ax^2+bx+c=0 (où a non nul).
# L'utilisation de la fonction racine carrée impose qu'elle soit
# importée du module math.
from math import sqrt
def trinome(a,b,c):
    delta=b**2-4*a*c
    if delta > 0:
        x1 = (-b+sqrt(delta))/(2*a)
        x2 = (-b-sqrt(delta))/(2*a)
        return [x1,x2]# Liste de deux nombres.
    elif delta == 0:
        x12 = -b/(2*a)
        return [x12]# Liste d'un seul nombre.
    else:
        return []# Liste vide.
```

**Exemples :** On a demandé en plus le type de la solution et le calcul de la somme des carrés des racines, calcul sans autre intérêt que de montrer que les racines données par `return` peuvent être utilisées dans des calculs suivants.

```
runfile('Chemin_de_la_fonction_trinome')
L=trinome(1,2*(2+sqrt(3)),7+4*sqrt(3))# Exemple n°1.
L
Out[3]: [-3.732050807568877]# Solution unique de (E).
type(L)
Out[4]: list
L[0]**2# Calcul de la somme des carrés des solutions.
Out[5]: 13.928203230275509
L = trinome(-10,2*(2+sqrt(3)),7+4*sqrt(3))# Exemple n°2.
L
Out[7]: [-0.8645761419679951, 1.6109863034817706]# Les 2 solutions de (E).
type(L)
```

32. Il s'agit, suivant le cas, de 0 (liste vide), 1 ou 2 nombres séparés par une virgule et placés entre crochets

```

Out[8] : list
L[0]**2+L[1]**2# Calcul de la somme des carrés des solutions.
Out[9] : 3.3427687752661224
L = trinome(10,2*(2+sqrt(3)),7+4*sqrt(3))# Exemple n°3.
L
Out[11] : [ ]# (E) n'a pas de solution (liste vide).
type(L)
Out[12] : list

```

Remplaçons `return` par `print()` :

```

# La fonction trinomebis ci-dessous retourne les solutions de
# l'équation (E) :  $ax^2+bx+c=0$  (où  $a$  est un réel non nul).
# L'utilisation de la fonction racine carrée impose qu'elle soit
# importée du module math.
from math import sqrt
def trinomebis(a,b,c):
    delta=b**2-4*a*c
    if delta > 0:
        x1 = (-b+sqrt(delta))/(2*a)
        x2 = (-b-sqrt(delta))/(2*a)
        print(' (E) a deux solutions ',x1, ' et ',x2, '. ')
    elif delta == 0:
        x12 = -b/(2*a)
        print(' (E) a une solution : ',x12, '. ')
    else:
        print(" (E) n'a pas de solution. ")

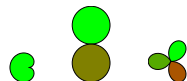
```

Exemples :

```

runfile('Chemin de la fonction trinomebis')
L = trinomebis(1,0,-2)
(E) a deux solutions  1.4142135623730951  et  -1.4142135623730951 .
type(L)
Out[3] : NoneType
L = trinomebis(1,2*(2+sqrt(3)),7+4*sqrt(3))
(E) a une solution :  -3.732050807568877 .
type(L)
Out[5] : NoneType
L = trinomebis(1,2,2)
(E) n'a pas de solution.
type(L)
Out[7] : NoneType

```



## 2.10 **[\*\*]** Minimum de 2, 3, 4 nombres (instructions conditionnelles)

- 1.a - Programmer une fonction qui retourne le minimum de 2 nombres.
  - 1.b - Programmer une fonction qui retourne le minimum et le maximum de 2 nombres.
  - 2 - Programmer une fonction qui retourne le minimum de 3 nombres.
  - 3 - Programmer une fonction qui retourne le minimum de 4 nombres.

**Mots-clefs** : Programmation de fonctions, instruction conditionnelle `if ...:...else ...:`, `return`, négation d'une proposition logique, importation de constantes et de fonctions pré-définies.

**Solution** : Quand on veut calculer le minimum d'une liste de nombres, on utilise bien évidemment la fonction pré-définie `min()` de Python, voir [13]. Ici, notre but est de calculer le minimum de 2, 3 ou 4 nombres sans utiliser cette fonction.

1.a - Cette question est élémentaire :

```
# La fonction min2nom retourne le minimum de 2 nombres a et b.
def min2nom(a,b) :
    if a <= b :
        return a
    else :
        return b
```

Application :

```
runfile('Chemin_de_la_fonction_min2nom')
min2nom(-7.17,-7.28)
Out[5] : -7.28
```

1.b - Ce n'est guère plus compliqué :

```
# minmax2nom retourne le minimum et le maximum de 2 nombres a et b.
def minmax2nom(a,b) :
    if a <= b :
        return [a,b]
    else :
        return [b,a]
```

Remarques :

- Le résultat apparaît sous la forme d'une **liste** de 2 nombres.
- Si `L` est une liste de nombres, `max(L) = - min(-L)`. Savoir calculer un minimum suffit pour calculer le maximum.

Application :

```
runfile('Chemin_de_la_fonction_minmax2nom')
minmax2nom(1.7,-3.84)
Out[7] : [-3.84, 1.7]
```

Si on remplace `[a,b]` et `[b,a]` par `a,b` et `b,a`, le résultat apparaîtra sous forme de **n-uplet**. Dans l'exemple ci-dessus, on obtiendra `(-3.84, 1.7)`<sup>33</sup>.

---

33. On peut sans dommage escamoter les notions de **liste** et de **n-uplet** (**tuple** pour Python) pour le moment.

3 - Ça se complique un peu car nous utilisons ci-dessous des instructions conditionnelles emboîtées. C'est aussi plus intéressant. Par exemple, on utilise

$$(a \geq b) \text{ ou } (a \geq c) \iff \neg (a < b \text{ et } a < c)$$

(la négation de "l'un ou l'autre", c'est "ni l'un, ni l'autre").

```
# La fonction min3nom retourne le minimum de 3 nombres a, b et c.
def min3nom(a,b,c) :
    if (a > b) or (a > c) :
        if b <= c :
            return b
        else :
            return c
    else :
        return a
```

On fait très attention, bien sûr, aux indentations.

Application : après avoir importé la valeur de `pi` du module `math`,

```
runfile('Chemin_de_la_fonction_min3nom')
from math import pi
min3nom(-2.1,3.2,-pi)
Out[10] : -3.141592653589793
```

3 - Le problème paraît plus difficile car il y aura plus d'instructions conditionnelles emboîtées ainsi que des difficultés logiques. Voici une solution commentée :

```
# min4nombis retourne le minimum de 4 nombres a, b, c et d.
def min4nombis(a,b,c,d) :
    if (a >= b) or (a >= c) or (a >= d) :# Le minimum est égal à b ou c ou d.
        if (b >= c) or (b >= d) :# Le minimum est égal à c ou d.
            if c >= d :
                return d
            else :
                return c
        else :
            return b
    else :
        return a
```

Cela repose sur l'équivalence logique :

$$(a \geq b) \text{ ou } (a \geq c) \text{ ou } (a \geq d) \iff \neg (a < b \text{ et } a < c \text{ et } a < d)$$

Application : dans l'exemple ci-dessous, nous devons importer la fonction `cos()` et la constante `pi` du module `math`.

```
runfile('Chemin_de_la_fonction_min4nombis')
from math import pi, cos
min4nombis(3.14,-2.71,1.4142,-cos(pi/6))
Out[3] : -2.71
```

C'est plus simple si l'on se permet d'utiliser la fonction `min3nom()`<sup>34</sup>, qu'il faudra alors importer<sup>35</sup> :

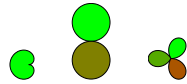
34. De même, on aurait simplifié le script définissant `min3nom()` en utilisant la fonction `min2nom()`.

35. si on ne l'a pas définie ou utilisée au cours de la même session

```
# min4nom retourne le minimum de 4 nombres a, b, c et d.  
from min3nom import min3nom  
def min4nom(a,b,c,d):  
    if a >= b:  
        return min3nom(b,c,d)  
    else :  
        return min3nom(a,c,d)
```

Application : reprenons l'exemple ci-dessus,

```
runfile('Chemin_de_la_fonction_min4nom')  
from math import cos, pi  
min4nom(3.14, -2.71, 1.4142, -cos(pi/6))  
Out[4] : -2.71
```



## 2.11 [\*\*\*] Alignement de 3 points

On se donne, à l'aide de leurs coordonnées, 3 points distincts  $A$ ,  $B$  et  $C$  d'un plan rapporté à un repère orthonormé. On appelle  $d_1$ ,  $d_2$  et  $d_3$  les distances de  $B$  à  $C$ , de  $A$  à  $C$  et de  $A$  à  $B$ . On peut supposer que  $d_1 \leq d_2 \leq d_3$ .

1 - Démontrer que les points sont alignés si et seulement si  $d_3 = d_1 + d_2$ .

2 - Sans utiliser la fonction racine carrée, programmer un algorithme qui retourne, à partir de la liste des coordonnées de  $A$ ,  $B$  et  $C$ , "Les points sont alignés" ou "Les points ne sont pas alignés", suivant le cas.

**Mots-clefs** : saisie de données, affectation de variables, additions et multiplications de nombres, problèmes logiques équivalents, équation d'une droite.

On peut supposer que  $0 < d_1 \leq d_2 \leq d_3$  parce que l'on peut échanger, si nécessaire, les lettres  $A$ ,  $B$  et  $C$ . L'échange éventuel de ces lettres est effectué ci-dessous par la commande `sorted([d1ca,d2ca,d3ca])`.

### Solution

Dans cet exercice, les calculs sont faciles. Les difficultés sont ailleurs.

- On peut s'appuyer sur un peu de géométrie : 2 cercles dont les centres sont distants de  $d$  et de rayons  $r_1$  et  $r_2$  plus petits que  $d$  sont tangents si et seulement si  $d = r_1 + r_2$ .
- On calcule facilement le carré d'une distance à l'aide d'additions et de multiplications<sup>36</sup>.
- La commande `ent = input('La liste des coordonnées de A, B et C est :')` définit `ent` comme une chaîne de caractères. Il faut ensuite l'évaluer à l'aide de la commande `ent = eval(ent)` qui la transforme en une liste de nombres.
- On sait bien que  $a = b$  n'est pas équivalent à  $a^2 = b^2$ . Pourtant, au moyen de 2 élévations au carré, dans ce cas particulier,  $d_1 + d_2 = d_3$  équivaut à  $4d_1^2d_2^2 = (d_3^2 - d_1^2 - d_2^2)^2$ <sup>37</sup>.

On en déduit une solution de notre problème sous forme de script :

```
ent = input('La liste des coordonnées de A, B et C est :')
ent = eval(ent)
d3ca = (ent[2] - ent[0])**2 + (ent[3] - ent[1])**2
# d3ca est le carré de la distance de A à B.
d2ca = (ent[0] - ent[4])**2 + (ent[1] - ent[5])**2
# d2ca est le carré de la distance de C à A.
d1ca = (ent[4] - ent[2])**2 + (ent[5] - ent[3])**2
# d1ca est le carré de la distance de B à C.
L=sorted([d1ca,d2ca,d3ca])
if 4*L[0]*L[1] == (L[2] - L[0] - L[1])**2 :
    print('Les points A, B et C sont alignés')
else :
    print('Les points A, B et C ne sont pas alignés')
```

Par exemple, exécutons ce script lorsque la liste des coordonnées de  $A$ ,  $B$  et  $C$  est égale à  $[-1, 3, 2, 0, 4, -2]$ , puis à  $[-4.74, 34.46, 0.17, 6.58, 5, -5.32]$ . On obtient, en travaillant directement dans la console interactive :

```
runfile('Chemin_du_script')
La liste des coordonnées de A, B et C est : [-1,3,2,0,4,-2]
Les points A, B et C sont alignés
```

36. alors que le calcul d'une distance n'est pas élémentaire, voir plus loin

37. Démontrer cette équivalence est en soi un exercice intéressant.

puis

```
runfile('Chemin_du_script')
La liste des coordonnées de A, B et C : [-4.74, 34.46, 0.17, 6.58, 5, -5.32]
Les points A, B et C ne sont pas alignés
```

Le premier essai est évident car  $B$  et  $C$  se trouvent manifestement sur la droite de pente  $-1$  qui passe par  $A$ . Pour le deuxième cas, il pourrait être utile de faire un graphe, c'est à dire de tracer le triangle  $ABC$ .

### Fin de la solution

#### Commentaires sur le test d'égalité de deux nombres réels

1 - Calculer une distance impose d'utiliser la fonction racine carrée qui appartient au module `math`. Il faudra l'importer. Voici ce qui arrive quand on l'oublie, si on tape `sqrt(2)` directement dans la console interactive :

```
sqrt(2)
Traceback (most recent call last):
  File "<ipython-input-9-40e415486bd6>", line 1, in <module>
    sqrt(2)
NameError: name 'sqrt' is not defined
```

La fonction `sqrt()` est déclarée inconnue ! Voilà ce qu'il faut faire :

```
from math import * # ou from math import sqrt
sqrt(2)
Out[2] : 1.4142135623730951
```

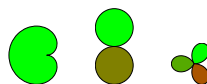
2 - Faisons un autre essai du script ci-dessus :

```
runfile('Chemin_du_script')
La liste des coordonnées de A, B et C est : [2.4, 0.197, -1.16, -0.2658,
7.7, 0.886]
Les points A, B et C ne sont pas alignés
```

Ce résultat est faux. Il est en effet facile de vérifier à la main que les points  $A$ ,  $B$  et  $C$  se trouvent sur la droite d'équation  $y=0.13x-0.115$ . Vérifions maintenant avec Python dans la console interactive :

```
0.13*2.4-0.115 == 0.197
Out[1] : True
0.13*(-1.16)-0.115 == -0.2658
Out[4] : True
0.13*7.7-0.115 == 0.886
Out[2] : False
0.13*7.7-0.115
Out[3] : 0.88600000000000001
```

On constate que pour Python,  $C$  n'est pas sur la droite  $(AB)$ . Cela provient du fait que Python calcule mal la quantité  $0.13*7.7-0.115$ . Il ne trouve pas  $0.886$  mais  $0.88600000000000001$ , ce qui est faux. Ce problème est imparable<sup>38</sup>. Habituellement, on se contente d'une quasi-égalité : on considère que deux nombres  $x$  et  $y$  sont égaux si  $|x - y| \leq \varepsilon = 2.220446049250313 * 10^{-16}$ . Ce problème apparaît presque toujours quand on veut tester l'égalité de deux nombres réels.



38. Voir [2], chapitre 4, 1 : « Les nombres en notation scientifique ».



## 2.12 [\*] Affectation de données

Soit  $f$  une fonction définie sur  $\mathbb{R}$  par  $f(x) = x^2 - 5x + 3$  et  $k$  un nombre réel. On souhaite déterminer tous les entiers  $n$  compris entre deux entiers donnés  $a$  et  $b$  ( $a < b$ ) tels que  $f(n) < k$ .

1 - -10 est-il solution de  $f(n) < 5$ ? Même question pour 0.

2 - Écrire un script Python appelant un nombre et affichant "oui" s'il est solution de  $f(n) < 5$ , "non" sinon.

3 - Modifier le script pour appeler  $k$  et afficher successivement tous les entiers solutions de  $f(n) < k$  sur l'intervalle  $[-10, 10]$ .

4 - Modifier le script pour demander aussi  $a$  et  $b$ .

**Mots-clefs :** Affecter des données à un script (commandes `input()` et `eval()`), définir une fonction, instruction conditionnelle `if ...:...else ...:`, boucle `pour`, conditions de type booléen, variables booléennes, masques, fonction `print()`, itérateur `range`.

### Solution <sup>39</sup>

1 - Directement dans la console interactive :

```
(-10)**2-5*(-10)+3 < 5
Out[1] : False
```

Ceci permet d'introduire les conditions de type booléen et les variables booléennes <sup>40</sup>. Ici, la variable booléenne prend la valeur `False`, ce qui signifie que l'inégalité  $f(-10) < 5$  est fausse.

2 - Le script ci-dessous convient :

```
n=eval(input('la valeur de n est :'))
if n**2 - 5*n + 3 < 5 :
    print('oui')
else :
    print('non')
```

Appliquons ce script <sup>41</sup> pour  $n = -4$  après l'avoir chargé dans la console de Python :

```
runfile('Chemin_du_script')
la valeur de n est : -4
non
```

Remarque : La commande `n=input('la valeur de n est :')` introduirait  $n$  comme une chaîne de caractères et non un nombre entier. Il faut l'«évaluer» à l'aide de la fonction `eval()`.

3 - Cette question est une répétition fastidieuse de la précédente. Les répétitions seront assurées par une boucle `pour`.

```
k=eval(input('la valeur de k est :'))
for n in range(-10,11) :
    if n**2 - 5*n + 3 < k :
        print(n)
```

39. On admettra que l'instruction `for n in range(a,b+1)` : où  $a$  et  $b$  sont des entiers tels que  $a < b$  produit successivement à les valeurs  $a, a+1, \dots, b$

40. Voir [2], chapitre 1, 7 La structure conditionnelle.

41. qui a été écrit dans l'interpréteur

Faisons l'essai pour  $k = 17$ . Ce qu'on obtient n'est pas très joli :

```
runfile('Chemin_du_script')
la valeur de k est :17
-1
0
1
2
3
4
5
6
```

4 - On a maintenant un problème à 3 paramètres :  $k$  (réel),  $a, b$  (entiers tels que  $a < b$ ).

```
k=eval(input('la_valeur_de_k_est_'))
a=eval(input('la_valeur_de_a_est_'))# par hypothèse, a est un entier.
b=eval(input('la_valeur_de_b_est_'))# par hypothèse, b est un entier.
for n in range(a,b+1):# On cherche les solutions dans [a,b].
    if n**2 - 5*n + 3 < k:
        print(n)
```

L'essai ci-dessous est fait pour  $k = -2.2$ ,  $a = -7$  et  $b = 22$ .

```
runfile('Chemin_du_script')
la valeur de k est : -2.2
la valeur de a est : -7
la valeur de b est :22
2
3
```

### Fin de la solution

#### Remarque et compléments :

Ce qui suit ne convient pas à des débutants en Python.

1 - On pourrait aussi traiter cet exercice à l'aide du signe du trinôme  $x^2 - 5x + 3 - k$ .

2 - On aurait pu rédiger cette solution à l'aide d'une fonction Python, voir la définition dans le script ci-dessous aux lignes 6 et 7.

3 - On peut illustrer l'exercice précédent à l'aide d'une représentation graphique :

```
import numpy as np
import pylab as plt

k=eval(input('La_valeur_de_k_est_'))#k est un nombre réel.
a=eval(input('La_valeur_de_a_est_'))# a est un entier.
b=eval(input('La_valeur_de_b_est_'))# b est un entier > a.

def f(x):
    return x**2 - 5*x + 3

abs=np.linspace(a,b,100)# [a,b] est partagé par 100 points équidistants
# dont a et b en 99 intervalles égaux.
ord=f(abs)# f est vectorisée.
plt.plot(abs,ord,color="blue")# Graphe de f entre a et b.
```

```
plt.plot([a,b],[k,k],color="red")# Graphe de  $y=k$  entre  $a$  et  $b$ .

plt.plot([a,b],[0,0],color="red")# Graphe de  $y=0$  entre  $a$  et  $b$ .

c=np.arange(a,b+1)#  $c$  est la matrice  $[a, a+1, \dots, b]$ .
d=f(c)#  $f$  est vectorisée.
np.masque=(d < k)

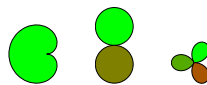
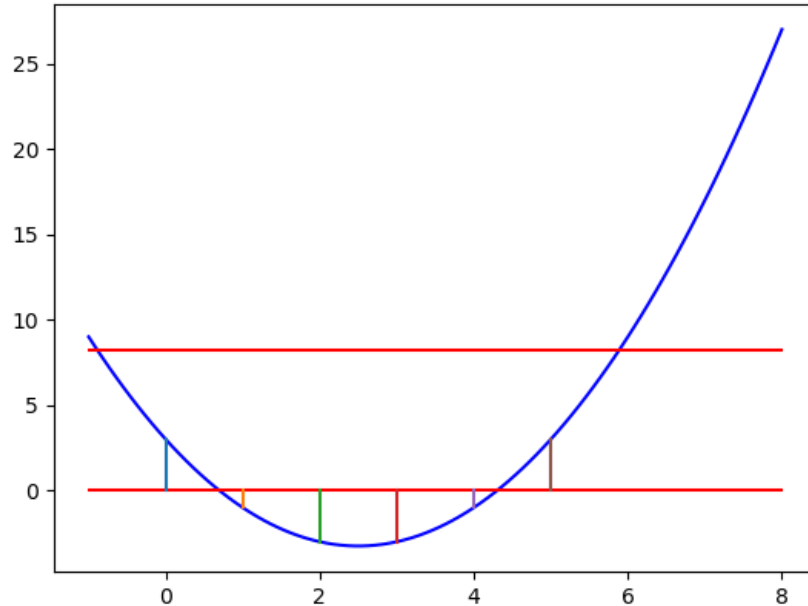
y=d[np.masque]
x=c[np.masque]
for i in x:
    plt.plot([i,i],[0,f(i)])# Graphe de  $x=i$  entre  $0$  et  $f(i)$ .

plt.savefig('sec2fig5.png')# Production d'une image au format png.
plt.show()# Tous les graphes sont affichés simultanément.
```

Application :

```
runfile('Chemin_du_script')
La valeur de k est : 8.21
La valeur de a est : -1
La valeur de b est : 8
```

Voici le graphe obtenu (image png produite par Python) :



## 2.13 [\*] Algorithme d'Euclide.

- 1 - Programmer une fonction qui, étant donnés deux entiers positifs **a** et **b**, retourne leur PGCD <sup>a</sup> calculé selon l'algorithme d'Euclide.
- 2 - En déduire le calcul de leur PPCM <sup>b</sup>.

*a.* plus grand diviseur commun  
*b.* plus petit multiple commun

**Mots-clefs** : reste et quotient de la division euclidienne (% et //), algorithme d'Euclide, boucle **tant que**, importation de fonctions, module **math**.

### Solution <sup>42</sup>

1 - La justification de l'algorithme d'Euclide est bien connue : si **r** désigne le reste de la division euclidienne de **a** par **b** <sup>43</sup>, l'ensemble des diviseurs communs de **a** et **b** est l'ensemble des diviseurs communs de **b** et **r**. Comme **r** < **b**, on tombe sur un problème plus simple si, au lieu de chercher l'ensemble des diviseurs communs de **a**, **b**, on cherche l'ensemble des diviseurs communs de **b**, **r**. Il est même possible que **r** = 0. Sinon, on recommence. On est sûr d'arriver à un reste nul. Or 0 étant divisible par tout entier positif **m**, l'ensemble des diviseurs communs de **n**, entier positif quelconque, et 0 est l'ensemble des diviseurs de **n**, le plus grand étant évidemment **n** lui-même. L'algorithme ci-dessous se trouve ainsi justifié : le PGCD de **a** et **b** est le dernier reste > 0.

```
def euclide(a,b):  
    while b > 0:  
        a,b = b,a%b  
    return a
```

Applications : On charge la fonction **euclide()** dans la console interactive en l'exécutant, si le listing précédent vient d'être rédigé dans l'interpréteur ou en l'important, s'il est conservé quelque part. Ci-après, on l'importe :

```
from euclide import *  
euclide(177,353)  
Out[2] : 1  
euclide(123456789,987654321)  
Out[3] : 9
```

Les calculs sont pratiquement instantanés : l'algorithme d'Euclide est très efficace. Bien sûr, Python fournit une fonction primitive qui fait la même chose : la fonction **gcd()** <sup>44</sup> :

```
from math import gcd  
math.gcd(177,353)  
Out[5] : 1  
math.gcd(123456789,987654321)  
Out[6] : 9
```

2 - Le calcul du PPCM de deux entiers positifs se déduit du calcul de leur PGCD parce que le produit du PPCM par le PGCD est égal au produit des deux nombres. En continuant le calcul ci-dessus dans la console interactive, on obtient :

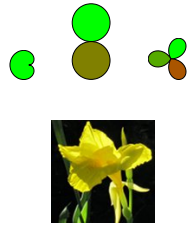
42. Le calcul du PGCD de **p** entiers positifs est repris à l'exercice 8.1.

43. Plus généralement, dans l'algorithme d'Euclide, **a** peut être un entier de signe quelconque, **b** un entier de signe quelconque non nul.

44. Le code source de la fonction **gcd()** du module **math**, voir [15] est exactement celui de la fonction **euclide()**.

```
print(177*353//1)
62481
print(123456789*987654321//9)
13548070123626141
```

Les PPCM recherchés sont respectivement égaux à 62481 et 13548070123626141.



# Chapitre 3

## Listes

---

Liste des exercices :

Énoncé n° 3.1 <sup>[\*\*]</sup> : Modélisation du jeu du lièvre et la tortue à l'aide de listes Python

Énoncé n° 3.2 <sup>[\*]</sup> : Simuler  $n$  lancers d'un dé équilibré, sous forme de liste

Énoncé n° 3.3 <sup>[\*]</sup> : Maximum d'une liste de nombres

Énoncé n° 3.4 <sup>[\*]</sup> : Ranger une liste de nombres

Énoncé n° 3.5 <sup>[\*]</sup> : Permutation aléatoire d'une liste.

Énoncé n° 3.6 <sup>[\*]</sup> : Une liste de listes

Énoncé n° 3.7 <sup>[\*\*]</sup> : Écrire toutes les permutations d'une liste

---

Ce chapitre contient des listes et des boucles **pour**.

Une liste est une séquence d'éléments rangés dans un certain ordre<sup>1</sup>. Ces éléments peuvent être de types différents<sup>2</sup>. Dans ce chapitre, nous considérerons essentiellement des listes de nombres. On dispose, pour manipuler des listes, d'un certain nombre d'instruments appelés **méthodes**<sup>3, 4</sup>.

Si l'on veut tester un algorithme, on ressent en général le besoin de disposer de longues listes de nombres que l'on engendre habituellement avec des générateurs de nombres aléatoires, pour l'unique raison que c'est bien pratique.

Sauf erreur, il n'existe pas de fonction qui retourne directement une liste de nombres aléatoires. Aussi, nous engendrerons d'abord des matrices-lignes de tels nombres, que nous convertirons ensuite en listes comme dans le script qui suit<sup>5, 6</sup> :

```
from numpy.random import normal
m = eval(input('m= ')) # moyenne de la loi normale (m quelconque)
ect = eval(input('ect= ')) # écart-type de la loi normale (ect > 0)
n = eval(input('n= ')) # longueur de la liste L (n entier >=1)
L = list(normal(m, ect, n))
```

---

1. Voir [2], pp. 20-27.

2. On en déduit que les listes ne sont pas faites pour les calculs.

3. Voir [2], p. 27 et les exercices qui suivent.

4. cf.[1], p. 219-221.

5. `list()` est un convertisseur de type

6. Les nombres aléatoires, dans cet exemple, sont engendrés suivant la loi normale de moyenne `m` et d'écart-type `ect` ; `n` est la longueur de la liste `L` à créer (fonction `normal()` du module `numpy.random`). Il suffit de savoir que `m` est un réel quelconque, `ect` un réel  $>0$ .

Exemples :

```
Exécution du script
m = 100
ect = 4
n = 10
L
Out[2] :
[98.336247781166918,
 101.84806811274237,
 96.381495198322625,
 92.589743231385569,
 104.57103790147788,
 106.85817698718772,
 101.40781668320082,
 96.726653061485123,
 91.331429567426639,
 102.87967202371736]
```

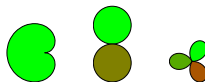
Calculons maintenant la durée de création d'une liste de longueur 1000000 de nombres aléatoires tirés suivant la loi normale de moyenne 5 et d'écart-type 10. Pour cela, commençons par ajouter au script précédent les instructions concernant la durée du calcul :

```
from time import clock
from numpy.random import normal
m = eval(input('m_=_'))# moyenne de la loi normale (m quelconque)
ect = eval(input('ect_=_'))# écart-type de la loi normale (ect > 0)
n = eval(input('n_=_'))# longueur de la liste L (n entier >=1)
a = clock()
L = list(normal(m, ect, n))
b = clock()
print(b-a)
```

Exécutons ce script :

```
m = 5
ect = 10
n = 1000000
0.10004399999999958
```

On constate que les calculs sont très rapides.



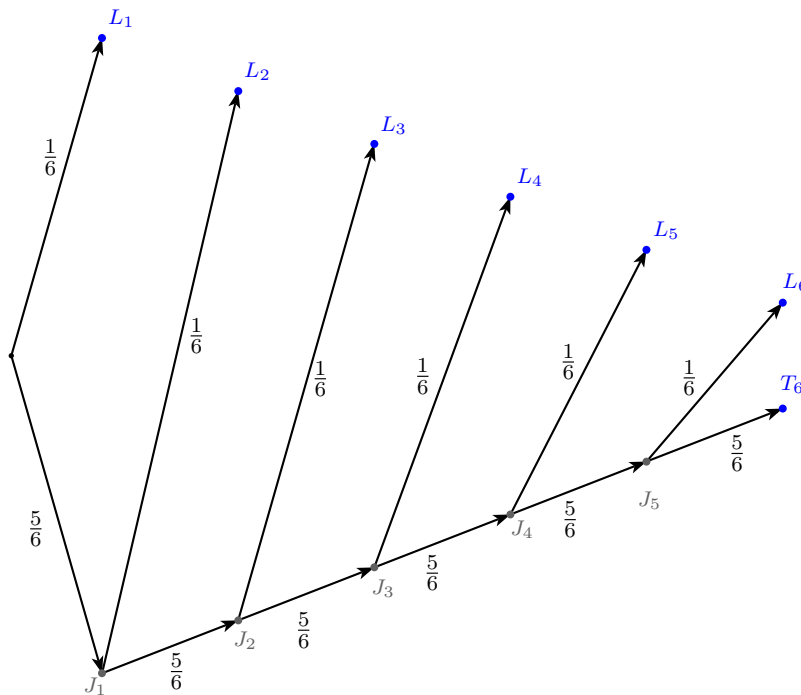
### 3.1 **[\*\*]** Modélisation du jeu du lièvre et la tortue à l'aide de listes Python

On lance un dé équilibré. Si le 6 sort, le lièvre gagne. Sinon la tortue avance d'une case et on rejoue. La tortue gagne si elle parvient à avancer de  $n$  cases ( $n \geq 1$  donné). On demande de modéliser ce jeu et de le simuler à l'aide d'une fonction bâtie sur des listes Python.

**Mots-clefs** : liste, Python, modélisation, mathématiques, probabilités, simulation.

Cet exercice très connu a peu d'intérêt calculatoire. C'est avant tout un exercice de modélisation que nous renvoyons à [24]. La meilleure modélisation se fait sous la forme du graphe ci-dessous, dessiné dans le cas  $n = 6$ , avec les notations suivantes :

- $J_i, i = 1, \dots, 5$  désigne l'événement : « Après le  $i^{\text{ème}}$  lancer, le jeu continue »,
- $L_i, i = 1, \dots, 6$  l'événement « Le lièvre gagne au  $i^{\text{ème}}$  lancer »,
- $T_6$  l'événement « La tortue gagne (nécessairement au  $6^{\text{ème}}$  lancer) ».



Le script repose sur ce graphe (qu'il est facile d'imaginer lorsque  $n$  est quelconque) :

```
# La fonction laftaine_listes() simule le jeu du lièvre et de la tortue
# à l'aide de listes Python.
from random import randint
def laftaine_listes(n):
    L = [] # Initialisation de la liste des faces obtenues.
    for i in range(n): # On sait qu'il y aura au plus n lancers du dé.
        de = randint(1,6) # Tirage d'un chiffre au hasard entre 1 et 6.
        L.append(de) # L sera la liste des chiffres tirés.
        if de == 6:
            p = str(len(L)) # Nombre des jets du dé écrit comme
```



```

        # une chaîne de caractères.
        print('Le dé a sorti la chaîne '+str(L)+' se terminant par 6.')
        return 'Donc le lièvre a gagné en '+p+' coups.'
    print("Le dé a sorti la chaîne "+str(L)+' de '+str(n)+' faces ne se
terminant pas par 6.')
    return 'La tortue a gagné.'

```

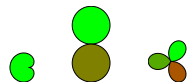
Exemples :

```

runfile('Chemin de la fonction laftaine_listes(n)')
laftaine_listes(1)
Le dé a sorti la chaîne [6] se terminant par 6.
Out[2]: 'Donc le lièvre a gagné en 1 coups.'
laftaine_listes(1)
Le dé a sorti la chaîne [4] de 1 faces ne se terminant pas par 6.
Out[3]: 'La tortue a gagné.'
laftaine_listes(1)
Le dé a sorti la chaîne [2] de 1 faces ne se terminant pas par 6.
Out[4]: 'La tortue a gagné.'
laftaine_listes(6)
Le dé a sorti la chaîne [1, 4, 1, 6] se terminant par 6.
Out[5]: 'Donc le lièvre a gagné en 4 coups.'
laftaine_listes(6)
Le dé a sorti la chaîne [6] se terminant par 6.
Out[6]: 'Donc le lièvre a gagné en 1 coups.'
laftaine_listes(6)
Le dé a sorti la chaîne [4, 2, 4, 2, 2, 2] de 6 faces ne se terminant
pas par 6.
Out[7]: 'La tortue a gagné.'
laftaine_listes(50)
Le dé a sorti la chaîne [1, 2, 1, 2, 2, 6] se terminant par 6.
Out[8]: 'Donc le lièvre a gagné en 6 coups.'
laftaine_listes(50)
Le dé a sorti la chaîne [4, 2, 6] se terminant par 6.
Out[9]: 'Donc le lièvre a gagné en 3 coups.'
laftaine_listes(50)
Le dé a sorti la chaîne [5, 2, 5, 1, 5, 4, 5, 2, 1, 2, 1, 3, 3, 5, 1,
5, 3, 1, 6] se terminant par 6.
Out[10]: 'Donc le lièvre a gagné en 19 coups.'

```

Dans [24], on trouvera des questions complétant cet exercice.



## 3.2 [\*] Simuler n lancers d'un dé équilibré, sous forme de liste

En utilisant la fonction `randint()` du module `random`, simuler, sous forme de liste, `n` lancers d'un dé équilibré.

**Mots-clefs** : liste, module `random`, fonction `randint()`, méthode `append()` de l'objet liste, boucle `pour`.

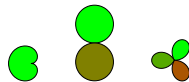
### Solution

`randint(a,b)` où `a` et `b`,  $a \leq b$  sont des entiers, retourne un nombre entier tiré au hasard dans l'intervalle `[a,b]`, extrémités comprises. Par exemple, `randint(1,6)` retourne un nombre tiré au hasard dans l'ensemble  $\{1, \dots, 6\}$  et par conséquent `randint(1,6)` simule le lancer d'un dé. Cela fournit le premier terme de la liste<sup>7</sup> recherchée. À l'aide d'une boucle `pour`, on ajoutera à cette liste les tirages suivants comme dans le script ci-dessous, à l'aide de la commande `append()`<sup>8,9</sup>. La fonction `lancersDe` est une solution du problème posé.

```
from random import randint
def lancersDe(n) :
    L=[] # Liste vide.
    for i in range(n): # Pour engendrer une liste de n termes.
        L.append(randint(1,6)) # randint est une fonction du module random.
    return(L)
# lancerDe(n) retourne une simulation d'une suite de n lancers d'un dé.
```

Exemple : simuler 15 lancers successifs d'un dé.

```
runfile('Chemin_de_la_fonction_lancersDe')
lancersDe(15)
Out[2] : [4, 4, 6, 4, 2, 6, 6, 3, 1, 5, 4, 4, 2, 1, 5]
```



7. Sur la notion de liste dans Python, voir [2], chapitre 1, section 9.

8. `append()` est une méthode de l'objet liste. Les méthodes usuelles des listes sont décrites dans [2], déjà cité.

9. Par exemple, directement dans la console :

```
L = [1,2,3,4]
L.append(5)
L
Out[11] : [1, 2, 3, 4, 5]
```

### 3.3 [\*] Maximum d'une liste de nombres

1 - Calculer, à l'aide d'une fonction à programmer, le maximum  $M$  d'une liste  $L$  de nombres, de longueur  $l$  comme suit :

(a) On pose  $M=L[0]$ .

(b) Ensuite, si  $l \geq 2$ ,  $i$  prenant successivement les valeurs  $1, \dots, l-1$ , on pose  $M=L[i]$  si  $L[i] > M$ .

2 - Application : calculer le maximum d'une liste de 10, puis de 10 000 nombres réels.

3 - Calculer le minimum  $m$  de  $L$  en utilisant l'algorithme du calcul ci-dessus.

**Mots-clefs** : instruction conditionnelle `if ...`, importation de fonction, convertisseur de type `list`, méthode `L.append()` d'une liste  $L$ , commande `len()`, primitives `min()` et `max()`.

#### Solution

1 - Nous proposons la fonction `atlas()` suivante, conforme à l'énoncé :

```
# atlas() retourne le maximum d'une liste de nombres.
def atlas(L):# L est une liste de nombres (au moins 1).
    for i in range(len(L)):# len(L) : longueur de L (notée l dans l'énoncé).
        if L[i]>L[0]:
            L[0] = L[i]
    return L[0]
```

Remarquons que lors d'une exécution d'`atlas()`, seul le premier terme de  $L$  est changé et ceci seulement dans le cas où il n'en est pas le maximum.

2 - On utilisera deux listes de longueur 10 et 10000 respectivement engendrées comme dans l'introduction de ce chapitre, cf. 3.

Cas d'une liste de 10 nombres

```
from atlas import atlas
from numpy.random import normal
L = list(normal(0,1,10))# On utilise ici la loi normale centrée réduite.
print(atlas(L))
```

On obtient après exécution :

```
1.56895495743
```

Cas d'une liste de 10 000 nombres

Il suffit de remplacer 10 par 10000 dans le script dont l'exécution paraît instantanée :

```
3.42225621142
```

On peut vérifier ce résultat. En effet, `max()` est une primitive de Python qui donne directement le maximum d'une liste de nombres ou d'une matrice. En continuant les calculs sur la console :

```
max(L)
Out[10]: 3.4222562114222046
```

3.1 - Normalement, on calcule le minimum de  $L$  à l'aide de la primitive `min()`<sup>10</sup> et de l'instruction `min(L)`.

3.2 - Une autre solution serait de programmer une fonction analogue de `atlas()` qui donnerait le minimum

10. `min()` est une primitive de Python qui donne directement le minimum d'une liste de nombres ou d'une matrice.

au lieu de donner le maximum (il suffit de remplacer > par < dans le script définissant `atlas()`).

**3.3** - On peut déduire le calcul du minimum d'une liste de nombres du calcul du maximum. En effet, le minimum de `n` nombres est l'opposé du maximum des opposés de ces nombres. C'est compliqué parce que `L` étant une liste, `- L` n'a pas de sens pour Python. On peut changer les signes des éléments de `L` un par un à l'aide d'une boucle `pour`. Dans le script ci-dessous, la liste `-L` est notée `Lm` :

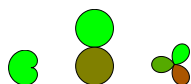
```
def hera(L) :  
    Lm = []  
    for i in range(len(L)) :  
        Lm.append(-L[i])  
    return Lm
```

Remarquons qu'on peut remplacer la commande `Lm.append(-L[i])` par la commande `Lm = Lm + [-L[i]]`<sup>11</sup> ou par la commande `Lm += [-L[i]]`.

**Application** : calculons le minimum de la liste `L` de 10000 nombres engendrée précédemment<sup>12</sup>,<sup>13</sup> :

```
from hera import hera  
-atlas(hera(L))  
Out[20] : -3.6212036474730676
```

Évidemment, les primitives `max()` et `min()` sont les fonctions que l'on utilise normalement.



---

11. Concaténation de listes.

12. continuation des calculs précédents

13. `atlas()` a déjà été importée.

### 3.4 [\*] Ranger une liste de nombres

Une liste de nombres  $L$  est donnée.

1 - À l'aide de la primitive `min()`, ranger cette liste dans l'ordre croissant (le premier terme  $m$  de la liste à calculer sera le minimum de  $L$ , le suivant sera le minimum de la liste  $L$  dont on aura retiré  $m$ , etc).

2 - Ranger  $L$  dans l'ordre décroissant.

**Mots-clefs :** primitives <sup>14</sup> `min()` et `max()`, boucle `pour`, méthodes `list.sort()` et `list.reverse()`, commandes `L[: p]` et `L[p :]` <sup>15</sup>, fonction `len()` (pour les listes), primitive `sorted()`.

#### Solution <sup>16</sup>

Pour ranger  $L$  dans l'ordre croissant puis dans l'ordre décroissant, on utilise normalement les méthodes `list.sort()` <sup>17, 18</sup> et `list.reverse()` <sup>19</sup>.

1 - L'énoncé suggère le procédé suivant :

- le premier terme de la future liste dans l'ordre croissant sera le minimum  $m$  de  $L$ ,
- le second sera le minimum de la liste obtenue en enlevant  $m$  de  $L$  (une seule fois si  $m$  y figure plusieurs fois)
- et ainsi de suite.

Cela se traduit par la fonction `aphrodite()` suivante qui retourne toute liste de nombres donnée après l'avoir rangée dans l'ordre croissant :

```
def aphrodite(L) :  
    Lcroissante = [] # [] : liste vide  
    for i in range(len(L)) : # i vaut 0 puis 1 ... et enfin len(L)-1  
        m = min(L)  
        Lcroissante.append(m) # Ajoute m à la fin de la liste Lcroissante.  
        L.remove(m) # Retire la première occurrence de m dans L.  
    return(Lcroissante)
```

$L$  perd un élément à chaque passage de la boucle. À la fin,  $L$  est vide. La fonction `aphrodite()` a été écrite dans l'éditeur de texte de l'environnement de programmation utilisé (Spyder). Pour ranger dans l'ordre croissant la liste  $L = [-2, 4.17, 8.79, -0.57, 0.19, -4.23]$ , exécutons `aphrodite()`, qui est ainsi chargée dans la console, ce qui se traduit par :

```
runfile('Chemin_de_la_fonction_aphrodite')
```

puis tapons :

```
L = [-2, 4.17, 8.79, -0.57, 0.19, -4.23]  
aphrodite(L)  
Out[3] : [-4.23, -2, -0.57, 0.19, 4.17, 8.79]  
L  
Out[4] : []
```

14. cf. [2], section 4, p. 37 : une primitive désigne une fonction de base fournie par Python.

15. listes des  $p$  premiers termes et des  $p$  derniers termes de la liste  $L$

16. Ceci est un exercice d'apprentissage. Il existe de nombreuses méthodes de tri.

17. remplace la liste par la même, ordonnée dans l'ordre croissant

18. ou la primitive `sorted()`

19. remplace la liste par la même, ordonnée dans l'ordre décroissant

On a vérifié que la valeur finale de L est [] et on a trouvé Lcroissante=[-4.23, -2, -0.57, 0.19, 4.17, 8.79].

On a déjà dit qu'en temps normal, on aurait utilisé la méthode `list.sort()` ou la primitive `sorted()` <sup>20</sup>. La valeur originale de L ayant été récupérée par un copier-coller, cela donne :

```
L = [-2,4.17,8.79,-0.57,0.19,-4.23]
L.sort()
L
Out[9]: [-4.23, -2, -0.57, 0.19, 4.17, 8.79]
L = [-2,4.17,8.79,-0.57,0.19,-4.23]
sorted(L)
Out[11]: [-4.23, -2, -0.57, 0.19, 4.17, 8.79]
```

**2** - Il suffit de modifier le script qui définit la fonction `aphrodite()` en remplaçant `min` par `max`.

### Remarque importante :

La liste L ci-dessus étant très courte et sans intérêt, remplaçons-la par une liste de 10 000 réels engendrée par la commande `reels(10000)` et ordonnons-la à l'aide de la fonction `aphrodite()` puis de la fonction `sorted()` :

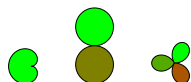
```
from reels import *
[L,Lo] = reels(10000)# Lo est une copie indépendante de L.
from aphrodite import *
Lcroissante = aphrodite(L)# Suite L rangée dans l'ordre croissant
# à l'aide de la fonction aphrodite.
Lcroissante[:4]# 4 plus petits termes de L.
Out[6]:
[-29.41317007426497,
 -28.747176136743192,
 -26.928746199250646,
 -25.74374951009743]
Lcroissante[9996:]# 4 plus grands termes de L.
Out[7]:
[40.68458999805904,
 41.64978402811625,
 41.821776781909804,
 43.184709110328164]

Lcroissante2 = sorted(Lo)# Suite L rangée dans l'ordre croissant
# à l'aide de la fonction sorted().
Lcroissante2[:4]# Vérification : 4 plus petits termes de L.
Out[9]:
[-29.41317007426497,
 -28.747176136743192,
 -26.928746199250646,
 -25.74374951009743]
Lcroissante2[9996:]# Vérification : 4 plus grands termes de L.
Out[10]:
[40.68458999805904,
 41.64978402811625,
 41.821776781909804,
 43.184709110328164]
```

---

20. Voir [2], chapitre 1, 9. Les listes, p.27.

La primitive `sorted()` est beaucoup plus rapide que la fonction `aphrodite()` à tel point que si on veut ranger une liste d'un million de nombres, par exemple, on sera très tenté d'interrompre le premier calcul tellement c'est long alors que le second dure à peu près une seconde.



### 3.5 [\*] Permutation aléatoire d'une liste.

Programmer une fonction qui retourne sous forme de liste une permutation aléatoire de la liste  $L = [1, 2, \dots, n]$ .

Méthode imposée : on commencera par tirer au hasard un élément de  $L$  qui deviendra le premier élément de la permutation aléatoire recherchée et on l'effacera de  $L$ . Puis on répètera cette opération jusqu'à ce que  $L$  soit vide.

**Mots-clefs** : boucle `pour`, fonction `randint()` du module `random`<sup>21</sup>, primitive `len()`, `L.append(x)`<sup>22</sup>, fonction `del()`, `list(range())`.

**Solution** : Le script est imposé par l'énoncé. Il suffit de connaître les commandes suivantes :

- `randint(a,b)`,  $a$  et  $b$  entiers tels que  $a \leq b$ , retourne un entier tiré au hasard dans l'intervalle  $[a, b]$ .
- `len(L)` retourne la longueur de la liste  $L$ ,
- `L.append(x)` modifie la liste  $L$  en lui ajoutant (à la fin) l'élément  $x$ .
- `list(range(n))` retourne  $[0, 1, \dots, n-1]$ .

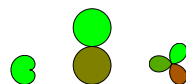
Cela qui permet de définir une fonction `alealiste` solution du problème comme suit :

```
from random import randint
def alealiste(L) :# L est une liste.
    alea = []
    n = len(L)
    for i in range(n) :
        j = randint(0,n-1-i)
        alea.append(L[j])
        del(L[j])
    return alea
```

Exemple : Produire une permutation aléatoire de la liste  $[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10]$  à l'aide de la fonction `alealiste()`.

```
L = list(range(11))
L
Out[2] : [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
from alealiste import *
alealiste(L)
Out[4] : [4, 2, 1, 7, 0, 9, 8, 5, 3, 10, 6]
```

Remarque importante : L'énoncé manque de précision puisque l'expression *permutation aléatoire* est très ambiguë. Le lecteur un peu au fait du Calcul des probabilités comprendra aisément qu'en modélisant comme il faut ce problème, on pourrait démontrer que toutes les permutations possibles de la liste `list(range(len(L))` sont équiprobables.



21. Le module `random` doit être distingué du module `numpy.random`.

22. `append()` est une méthode de l'objet liste.



## 3.6 [\*] Une liste de listes

Programmer une fonction qui, pour tout entier naturel  $n \geq 2$ , retourne la liste dont le premier élément est la liste  $[1, \dots, n]$ , le second élément la liste  $[2, \dots, n, 1]$  et ainsi de suite jusqu'à la liste  $[n, 1, \dots, n-1]$ .

**Mots-clefs :** `list(range(p,q))`, boucle `pour`, remplacement d'un élément d'une liste par un autre, concaténation de 2 listes.

### Solution

La fonction `liliste()` ci-dessous est une solution de ce problème simple : elle retourne la liste `K` qui est initialisée à `K = [1, ..., n]`. Ensuite, son premier terme : 1 est remplacé par la liste `L = [0, ..., n-1]` à l'aide de la commande `K[0] = L`. On fait de même pour les autres éléments de `K` après avoir, à chaque étape de la boucle `pour`, remplacé `L` par sa nouvelle version obtenue en mettant son premier terme à la fin<sup>23</sup>.

```
def liliste(n):
    K = list(range(n)) # --> K = [0, ..., n-1].
    L = list(range(1,n+1))
    K[0] = L # K = [[1, ..., n], 1, ..., n-1]
    for i in range(1,n):
        L = L[1:n]+[L[0]] # On met le premier terme de L à la fin.
        K[i] = L
    return K # K est une liste de n listes.
```

Exemple : Pour  $n = 8$  :

```
runfile('Chemin_de_la_fonction_liliste')
liliste(8)
Out[2]:
[[1, 2, 3, 4, 5, 6, 7, 8],
 [2, 3, 4, 5, 6, 7, 8, 1],
 [3, 4, 5, 6, 7, 8, 1, 2],
 [4, 5, 6, 7, 8, 1, 2, 3],
 [5, 6, 7, 8, 1, 2, 3, 4],
 [6, 7, 8, 1, 2, 3, 4, 5],
 [7, 8, 1, 2, 3, 4, 5, 6],
 [8, 1, 2, 3, 4, 5, 6, 7]]
```

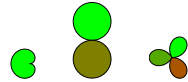
En modifiant légèrement la fonction `liliste()`, la liste `L` précédente peut devenir une liste quelconque :

```
def liliste2(L): # L est une liste quelconque.
    n = len(L) # Longueur de L.
    K = list(range(n)) # --> K = [0, ..., n-1].
    K[0] = L # K = [L, 1, ..., n-1]
    for i in range(1,n):
        L = L[1:n]+[L[0]] # On met le premier terme de L à la fin.
        K[i] = L
    return K # K est une liste de n listes.
```

23. Pour cela, on concatène `L[1:n]` avec `[L[0]]` qui est une liste à un élément et non avec `L[0]` qui, ici, est un nombre.

Exemple :

```
runfile('Chemin_de_la_fonction_liliste2')
L = [3, 'Oh', [8], 'Attila', 0.175]
liliste2(L)
Out[3] :
[[3, 'Oh', [8], 'Attila', 0.175],
 ['Oh', [8], 'Attila', 0.175, 3],
 [[8], 'Attila', 0.175, 3, 'Oh'],
 ['Attila', 0.175, 3, 'Oh', [8]],
 [0.175, 3, 'Oh', [8], 'Attila']]
```



### 3.7 **[\*\*]** Écrire toutes les permutations d'une liste donnée L

**1** - Étant donné une liste L à n éléments et une liste l, programmer une fonction notée `aïda( , )` de sorte que la commande `aïda(L,l)` retourne une liste de n+1 listes dont la première est `l+L`<sup>a</sup>, la dernière `L+l`, les listes intermédiaires étant obtenues en insérant l dans la liste L entre les éléments `L[i]` et `L[i+1]`, i variant de 0 à n-2.

**2** - L étant une liste non vide de longueur n, programmer, à l'aide de la fonction `aïda( , )`, une fonction notée `permutliste()` de sorte que la commande `permutliste(L)` retourne la liste des n! listes dont chacune est une permutation différente de la liste L.

a. Concaténée des listes L et l.

**Mots-clefs** : concaténation de listes, extraction d'une sous-liste d'une liste.

On suppose évidemment que la liste L n'est pas vide. La solution ci-dessous pour engendrer toutes les permutations d'une liste est analogue à celle de 5.7 où on engendrait toutes les permutations d'une matrice-ligne. Les outils changent, évidemment.

#### Solution

**1** - La fonction `aïda()` ci-dessous n'a guère qu'un intérêt technique. Elle correspond à la fonction `nabucco()` de 5.7 et se programme sans difficulté :

```
def aïda(L,l):
    Li=[]
    n=len(L)
    for i in range(0,n+1):
        M=L[:i]+l+L[i:]# M est une liste dont le seul élément est une liste.
        Li=Li+M
    return Li
```

Exemple :

```
L=['Anatole','Bonjour','au_revoir']
l=[0,0]
runfile('Chemin_de_la_fonction_aïda()')
aïda(L,l)
Out[4]:
[[0, 0, 'Anatole', 'Bonjour', 'au_revoir'],
 ['Anatole', 0, 0, 'Bonjour', 'au_revoir'],
 ['Anatole', 'Bonjour', 0, 0, 'au_revoir'],
 ['Anatole', 'Bonjour', 'au_revoir', 0, 0]]
```

**2** - À partir de la liste des permutations de `L[:i]`,  $1 \leq i < n$ , on obtient la liste des permutations de `L[:i+1]` à l'aide de la fonction `aïda( , )` et d'une boucle `pour`. Cela donne une fonction notée `permutliste()` qui retourne la liste des permutations de L :

```
# L étant une liste non vide de longueur n, la fonction permutliste() retourne
# une liste de n! listes, chacune d'elles étant une permutation différente
# de la liste L.
```

```
from aïda import aïda
def permutliste(L):
```

```

n=len(L)
perm=[[L[0]]]# Liste des permutations de la liste L[:1].
for i in range(1,n):# Passage de la liste des permutations de L[:i]
    # à la liste des permutations de L[:i+1].
    m=len(perm)
    Li=[]
    for j in range(m):
        Li=Li+aïda(perm[j],[L[i]])
    perm=Li
return perm

```

Exemple :

```

L=list(range(1,6))
L
Out[2]: [1, 2, 3, 4, 5]
runfile('Chemin_de_la_fonction_permutliste()')
permutliste(L)
Out[4]:
[[5, 4, 3, 2, 1],
 [4, 5, 3, 2, 1],
 [4, 3, 5, 2, 1],
 [1, 2, 5, 3, 4],
 [1, 2, 3, 5, 4],
 [1, 2, 3, 4, 5]]

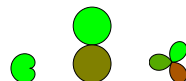
```

Autre exemple : <sup>24</sup>

```

L=['Belle_Marquise','_vos_beaux_yeux','_me_font_mourir_d'amour']
permutliste(L)
Out[6]:
[["_me_font_mourir_d'amour", '_vos_beaux_yeux', 'Belle_Marquise'],
 ['_vos_beaux_yeux', "_me_font_mourir_d'amour", 'Belle_Marquise'],
 ['_vos_beaux_yeux', 'Belle_Marquise', "_me_font_mourir_d'amour"],
 ["_me_font_mourir_d'amour", 'Belle_Marquise', '_vos_beaux_yeux'],
 ['Belle_Marquise', "_me_font_mourir_d'amour", '_vos_beaux_yeux'],
 ['Belle_Marquise', '_vos_beaux_yeux', "_me_font_mourir_d'amour"]]

```




---

24. Exemple extrait de l'article [26].

# Chapitre 4

## Matrices-lignes

---

Liste des exercices :

Énoncé n° 4.1 <sup>[\*\*]</sup> : Maximum et minimum d'une matrice-ligne.

Énoncé n° 4.2 <sup>[\*\*\*]</sup> : Permutations aléatoires d'une matrice-ligne.

---

Les matrices relèvent du module `numpy` de Python qu'il faut donc charger dans la console lorsqu'on en utilise. Malheureusement, `numpy` distingue deux sortes de matrices, ce que l'on ne fait pas en mathématiques <sup>1</sup> :

- les matrices-lignes ou matrices  $(m, )$  <sup>2</sup> ou matrices à une dimension, terme ambigu <sup>3</sup>, par exemple

```
import numpy as np
A = np.array([1,2,3])
A
Out[3]: array([1, 2, 3])
type(A)
Out[4]: numpy.ndarray
A.dtype
Out[5]: dtype('int64')# Les éléments de A sont des entiers codés en 64 bits.
A.shape# ou shape(A)
Out[6]: (3, )# A est bien une matrice (m, ).
```

- les matrices multidimensionnelles (on se limitera aux matrices à deux dimensions, voir le chapitre suivant). Ainsi dans la console, à la suite de l'exemple précédent :

```
B = np.array([[1.1,2.2,3.3],[4.4,5,6]])
B
Out[8]:
array([[ 1.1,  2.2,  3.3],
       [ 4.4,  5. ,  6. ]])
type(B)
Out[9]: numpy.ndarray
B.dtype
Out[10]: dtype('float64')
```

---

1. En mathématiques, au niveau élémentaire, les matrices sont de simples tableaux rectangulaires de nombres qui ont donc deux dimensions : le nombre des lignes  $n$  et le nombre des colonnes  $m$  résumées en  $(n,m)$ . Elles servent principalement à représenter des applications linéaires d'un espace vectoriel de dimension  $m$  dans un espace vectoriel de dimension  $n$ .

2.  $m$  est le nombre d'éléments de la ligne.

3. Les matrices-colonnes ne sont pas des matrices à une dimension, mais des matrices à deux dimensions pour `numpy`

```
B.shape  
Out[11] : (2, 3)
```

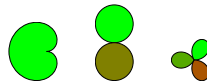
On peut convertir une matrice (m, ) en une matrice (1,m) comme suit :

```
C = A[np.newaxis]  
C  
Out[13] : array([[1, 2, 3]])# À comparer avec la sortie [3].  
type(C)  
Out[14] : numpy.ndarray  
C.shape  
Out[15] : (1, 3)# À comparer avec la sortie [6].
```

*Exemple* : une matrice-colonne est une matrice à deux dimensions.

```
from numpy import array  
A = array([[1],[2.25]],float)# Matrice de nombres à virgule flottante.  
A  
Out[2] :  
array([[1. ],  
       [2.25]])  
A.shape  
Out[3] : (2, 1)
```

Ce chapitre comprend seulement des exercices sur les matrices-lignes.



## 4.1 **[\*\*]** Maximum et minimum d'une matrice-ligne

1 - Calculer le maximum d'une matrice-ligne de nombres  $M$  en procédant comme suit :

- (a) - on choisit arbitrairement l'un de ces nombres, noté  $ma$ <sup>a</sup> et on efface les nombres  $\leq ma$ .
- (b) - s'il n'en reste plus,  $ma$  est le maximum recherché.
- (c) - sinon, on recommence (a) et (b).

2 - Calculer le minimum  $mi$  de  $M$ .

a. Ci-dessous,  $ma$  est le premier terme de  $M$ .

**Mots-clefs :** matrices-lignes, commande `len()`, masque, maximum et minimum, fonctions primitives `max()` et `min()`, boucle `tant que`, importation de fonctions, module `time`, fonction `clock()`, module `copy`, fonction `deepcopy()`.

### Solution

S'agissant de matrices, on aura besoin d'importer le module `numpy` ; on utilisera un masque de matrice<sup>4, 5</sup>, ce qui fait l'intérêt de cet exercice au demeurant très court.

1 - On peut utiliser le script commenté suivant qui définit une fonction notée `maxime()` qui, comme la primitive `max()`<sup>6</sup>, retourne le maximum d'une matrice-ligne de nombres donnée.

```
# La fonction maxime utilise le masque M > ma.
from numpy import *

def maxime(M) : # M est une matrice-ligne de nombres.
    while len(M)>=1 : # Tant que M n'est pas vide.
        ma = M[0] # Le choix de ma dans M est arbitraire.
        M = M[M>ma] # On ne conserve que les éléments de M > ma.
        # La longueur de M diminue à chaque passage de la boucle.
    return (ma)
```

La commande `M = M[M>ma]` ne retient de  $M$  que les nombres  $> ma$ , autrement dit efface les autres.

Exemple : Importons la fonction `maxime()` dans la console<sup>7, 8</sup>, puis cherchons le maximum de la matrice-ligne  $M$  retourné par la commande `M = randn(1000000)`<sup>9</sup> :

```
from maxime import *
from numpy.random import *
M = randn(1000000)
maxime(M)
Out[7] : 4.8898710392903109
```

Le calcul a été pratiquement instantané.

2 - Pour calculer le minimum d'une matrice `Mat`, on voudrait utiliser l'égalité

$\min(\text{Mat}) = -\max(-\text{Mat})$ , autrement dit,  $\min(\text{Mat}) = -\max(-\text{Mat})$ .

4. cf. [10], p. 17-18.

5. On verra que les masques de matrices sont très utiles.

6. `max()` fait mieux. En effet, `max()` retourne le maximum d'une matrice-ligne ou d'une liste de nombres alors que `maxime()` ne s'applique pas aux listes.

7. On la charge comme si c'était un module.

8. ce qui provoque aussi le chargement de `numpy`

9. `randn` est une fonction du module `numpy.random` qu'il faut aussi charger dans la console. Il est inutile de savoir que  $M$  est un échantillon de la loi normale centrée réduite de taille 1 000 000.

vraie pour n'importe quelle matrice **Mat**. Malheureusement, la valeur initiale de **M** a été perdue dans le calcul de **maxime(M)**. On aurait dû conserver cette valeur initiale la ré-utiliser dans le calcul du minimum. Cela se fait avec la fonction **deepcopy()** du module **copy**<sup>10</sup>.

Exemple : Engendrons une matrice-ligne de 1 000 000 de nombres flottants comme ci-dessus, puis nous calculons son maximum et son minimum ; enfin, faisons apparaître ces valeurs à l'aide de **print()**.

```
from maxime import *
from copy import deepcopy
from numpy.random import randn

M = randn(1000000)
Mbis = deepcopy(M)
ma = maxime(M)
mi = -maxime(Mbis)
print(ma, mi)
4.59811940548 -4.59811940548
```

### Fin de l'exercice

#### Compléments : comparaison des vitesses de **maxime()** et de **max()**.

Pour comparer les vitesses de **maxime()** et de la primitive **max()**, considérons le script suivant, qui donne les durées du calcul du maximum de la matrice-ligne **M** ci-dessous par **maxime()**, puis par **max()**. Pour fixer les idées, nous lançons une nouvelle session de calcul en tapant dans l'interpréteur :

```
from time import clock # Pour compter les durées de calcul.
from copy import deepcopy # Pour fabriquer une copie indépendante de M.
from maxime import * # Importation de la fonction maxime.
# Cette importation comprend le module numpy.

M=random.randn(1000000)
Mbis=deepcopy(M)
d=clock()
print('Le maximum de M par maxime est ',maxime(M))
f=clock()
print('Le temps de calcul par maxime est ',f-d)
d=clock()
print('Le maximum de Mbis par max est ',max(Mbis))# Autre calcul du maximum.
f=clock()
print('Le temps de calcul par max est ',f-d)
```

puis nous exécutons ce script :

```
runfile('Chemin_du_script')
Le maximum de M par maxime est 4.69931038606
Le temps de calcul par maxime est 0.0257899999999999758
Le maximum de Mbis par max est 4.69931038606
Le temps de calcul par max est 0.070354000000000003
```

La comparaison des durées de calcul est surprenante car la fonction **maxime()** paraît plus rapide que la fonction primitive de Python **max()**.

---

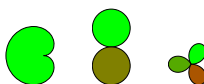
10. Voir [2], p. 23-25



Faisons un deuxième essai.  $M$  est maintenant un échantillon de taille 80000 de la loi uniforme sur l'intervalle d'entiers  $[-100000, 250000]$ <sup>11</sup>,<sup>12</sup>. On obtient :

```
runfile('Chemin_du_script_modifié')
Le maximum de M par maxime est 249999
La durée du calcul par maxime est 0.0018669999999999841
Le maximum de M par max est 249999
La durée du calcul par max est 0.008494999999999992
```

Cela semble confirmer la remarque précédente sur la vitesse de `maxime()` et de `max()`.



---

11. ce que l'on peut ignorer ; sur la fonction `randint` du module `numpy.random`, voir [2], p. 163.

12. Il suffit, dans le script précédent, de remplacer la commande `M=random.randn(100000)` par la commande `M=randint(-100000,250001,size = 80 000)`.

## 4.2 [\*\*\*] Permutations aléatoires d'une matrice-ligne.

1 - Programmer une fonction qui retourne une permutation aléatoire de la matrice à une dimension  $V = \text{array}([1, 2, \dots, n])$  sous forme de matrice à une dimension.

Méthode imposée : On commencera par tirer au hasard un élément de  $V$  qui deviendra le premier élément de la permutation aléatoire  $W$  recherchée et on l'effacera de  $V$ . Puis on répètera cette opération tant que  $V$  n'est pas vide.

2 - Utiliser la fonction précédente pour fabriquer des permutations aléatoires de n'importe quelle matrice à une dimension  $M$ .

**Mots-clefs** : fonctions `array()`, `arange()`, `hstack()`, `delete()`, `len()`, `shuffle()` du module `numpy`, `randint()`, `rand()` du module `numpy.random()`, échange d'éléments et extraction d'une sous-matrice d'une matrice à une dimension, `size`, méthode des matrices à une dimension, conversion d'une matrice à une dimension en une liste.

### Solution

1 - Les deux fonctions `orphise()` et `orphise1()` ci-dessous, solutions de la première question, ne se distinguent que par la façon d'effacer de  $V$  les éléments tirés au hasard successifs.

Solution n°1 :

```
import numpy as np
def orphise(n):# n est un entier >=1.
    W = np.array([],int)# W est une matrice d'entiers (m, ) vide.
    V = np.arange(n)# V est une matrice (m, ).
    for i in range(n):
        j = np.random.randint(0,len(V))# j est choisi au hasard,
        # équiprobablement, dans l'ensemble 0, ..., n-i-1.
        W = np.hstack((W,np.array([V[j]])))# On ajoute V[j] à la fin de W.
        V = np.delete(V,[j])# On efface V[j] de V.
    return W # W est une matrice (m, ) obtenue par une permutation aléatoire
# de 0, 1, ..., n-1.
```

Exemple :

```
from orphise import *
orphise(15)
Out[2]: array([ 7, 12,  8,  4,  5, 15,  1, 13, 11,  3,  2,  0, 14, 16,  6])
```

Solution n°2 :

```
import numpy as np
def orphise1(n):# n entier >=1.
    W = np.array([],int)# W est une matrice d'entiers (m, ) vide.
    V = np.arange(n)# V est une matrice (m, ).
    for i in range(n):
        j = np.random.randint(0,len(V))# j est choisi au hasard,
        # équiprobablement, dans l'ensemble 0, ..., n-i-1.
        W = np.hstack((W,np.array([V[j]])))# On ajoute V[j] à la fin de W.
        V = np.hstack((V[0:j],V[j+1:len(V)]))# On efface V[j] de V.
    return W # W est une matrice (m, ) obtenue par une permutation aléatoire
# de 0, 1, ..., n-1.
```

Exemple :

```
from orphise1 import *
orphise1(15)
Out[4] : array([ 0,  3,  6,  7, 15, 10,  1,  5, 16, 12, 13, 11,  2,  9,  8])
```

2 - On permute aléatoirement les éléments d'une matrice en permutant aléatoirement les indices de ces éléments. Cela donne la fonction `orphise2()` suivante qui, étant donné une matrice `M` à une dimension retourne une matrice à une dimension qui est une permutation de `M`.

```
from orphise import *
def orphise2(M) :# M est une matrice (m, ).
    m = M.size
    W = orphise(m)# W est une matrice (m, ).
    L = list(W)# Conversion de type, de "matrice à une dimension" vers "liste".
    Ma = M[L]
    return Ma
```

Exemple :

```
from orphise2 import *
M = numpy.random.rand(25, )# M est une matrice à une dimension à 25 éléments.
M# Pour voir.
Out[3] :
array([ 3.59103529e-03,  8.36588940e-01,  9.31836143e-01,
        5.97425051e-01,  3.98459887e-01,  2.46173892e-01,
        9.09341745e-01,  5.15681264e-01,  3.44637311e-01,
        1.21643146e-04,  4.67647036e-01,  4.90701074e-01,
        9.47681157e-01,  1.87804186e-01,  6.02391582e-01,
        1.06736664e-01,  8.00597715e-01,  2.76038296e-01,
        3.31099944e-02,  8.22724748e-01,  9.89534567e-01,
        6.37337017e-01,  4.26496185e-01,  4.95840654e-01,
        2.31432883e-01])
orphise2(M)
Out[4] :
array([ 3.59103529e-03,  5.97425051e-01,  5.15681264e-01,
        2.46173892e-01,  4.90701074e-01,  4.26496185e-01,
        9.47681157e-01,  1.06736664e-01,  3.44637311e-01,
        6.02391582e-01,  9.89534567e-01,  2.76038296e-01,
        8.36588940e-01,  8.22724748e-01,  2.31432883e-01,
        9.09341745e-01,  4.67647036e-01,  8.00597715e-01,
        1.87804186e-01,  3.98459887e-01,  6.37337017e-01,
        9.31836143e-01,  4.95840654e-01,  3.31099944e-02,
        1.21643146e-04])
```

### 3 - Solution Python des questions 1 et 2 :

- La fonction `shuffle()` rebat les cartes : la commande `shuffle(M)` remplace toute matrice à une dimension `M` par la matrice qui en est issue quand on permute aléatoirement équiprobablement les éléments de `M`. La valeur initiale de `M` est perdue.

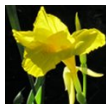
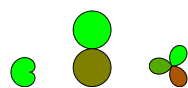
Exemple :

```
M = np.arange(7)
M
Out[4] : array([0, 1, 2, 3, 4, 5, 6])# Matrice à une dimension.
```

```
np.random.shuffle(M)
```

M

```
Out[6]: array([6, 0, 2, 3, 5, 1, 4])
```



## Chapitre 5

# Matrices (matrices (n,m))

---

Liste des exercices :

Énoncé n° 5.1 [\*] : Matrices croix (fantaisie).

Énoncé n° 5.2 [\*] : Matrice Zorro (fantaisie)

Énoncé n° 5.3 [\*] : Transformations aléatoires de matrices.

Énoncé n° 5.4 [\*] : Calculer la trace d'une matrice.

Énoncé n° 5.5 [\*\*] : Ce carré est-il magique ?

Énoncé n° 5.6 [\*\*\*] : Transposer une matrice

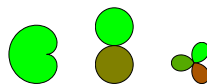
Énoncé n° 5.7 [\*\*\*\*] : Écrire toutes les permutations de  $(1, \dots, n)$ .

Énoncé n° 5.8 [\*\*\*\*] : Carrés magiques d'ordre 3.

Énoncé n° 5.9 [\*\*\*] : Construction des premiers flocons de Koch avec `plot`.

---

Les matrices (n,m) ou matrices à deux dimensions : le nombre de lignes `n` et le nombre de colonnes `m` sont les matrices des mathématiciens ! Le module `numpy` leur est consacré, voir [16].



## 5.1 [\*] Matrice croix (fantaisie).

Écrire la matrice carrée  $M$  de taille  $2n+1$  comportant des 1 sur la  $(n+1)^{\text{ième}}$  ligne et la  $(n+1)^{\text{ième}}$  colonne et des 0 ailleurs.

**Mots-clefs :** module `numpy` : fonctions `np.zeros(( , ), int)`, `np.ones(( , ), int)`, concaténation horizontale et verticale de matrices<sup>1</sup>.

Ceci est un exercice d'apprentissage des fonctions citées dans les mots-clefs ci-dessus.

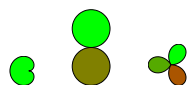
### Solution

C'est très facile :  $M$  est un assemblage de 7 matrices constituées soit de 0, soit de 1, objet de la fonction appelée `croix()` ci-dessous<sup>2</sup>.

```
import numpy as np
def croix(n):
    A = np.zeros((n,n),int)
    B = np.ones ((n,1),int)
    C = np.ones ((1,2*n+1),int)
    D = np.concatenate((A,B,A),axis = 1)
    E = np.concatenate((D,C,D),axis = 0)
    return E
```

Exemple : pour  $n = 8$ ,

```
runfile('Chemin_de_la_fonction_croix')
croix(8)
Out[15]:
array([[0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0],
       [0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0],
       [0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0],
       [0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0],
       [0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0],
       [0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0],
       [0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0],
       [0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0],
       [1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1],
       [0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0],
       [0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0],
       [0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0],
       [0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0],
       [0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0],
       [0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0],
       [0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0],
       [0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0]])
```



1. respectivement `np.concatenate(( , , ),axis = 1)` et `np.concatenate(( , , ),axis = 0)`

2. Comme l'énoncé ne le précise pas, on a supposé que ces 0 et 1 sont des nombres entiers ; sinon, remplacer `int` par `float`

## 5.2 [\*] Matrice Zorro (fantaisie)

Écrire la matrice  $M$  de taille  $n$ , définie comme la matrice carrée de taille  $n$  comprenant des 1 sur la première ligne, sur la deuxième diagonale et sur la dernière ligne et des 0 partout ailleurs.

**Mots-clefs :** module `numpy` : fonctions `np.zeros((n, m),int)`, `np.ones((m, ),int)`, `np.arange(p,q,r)`, extraction d'un bloc d'éléments d'une matrice : `M[m1,m2]`.

Étant donné une matrice  $M$ ,  $M[m1,m2]$ , où  $m1$  et  $m2$  sont des matrices  $(m, )$  de même longueur, retourne la matrice  $(m, )$  dont les éléments sont ceux de  $M$  dont les indices de ligne forment  $m1$  et les indices de colonne forment  $m2$ . Par exemple, `M[arange(n-1,-1,-1), arange(0,n)]` est la deuxième diagonale de  $M$ . Les matrices  $m1$  et  $m2$  peuvent être remplacées par des listes.

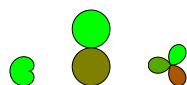
### Solution

Pour ce faire, on part d'une matrice carrée d'ordre  $n$  ne comprenant que des 0 entiers (on aurait pu les choisir flottants), puis on remplace ces 0 par des 1 sur la première et la dernière lignes ainsi que sur la deuxième diagonale. Cette fonction a été appelée `zorro()`.

```
import numpy as np
def zorro(n):
    M = np.zeros((n,n),int)
    M[0,:] = np.ones((n, ),int)
    M[n-1,:] = np.ones((n, ),int)
    M[np.arange(n-2,0,-1),np.arange(1,n-1,1)] = np.ones((1,n-2),int)
    return M
```

Exemple :  $n=13$

```
runfile('Chemin_de_la_fonction_zorro')
zorro(13)
Out[2]:
array([[1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1],
       [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0],
       [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0],
       [0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0],
       [0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0],
       [0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0],
       [0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0],
       [0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0],
       [0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0],
       [0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
       [0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
       [1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1]])
```



## 5.3 [\*] Transformations aléatoires de matrices

1 - Programmer une fonction qui, pour tout entier  $n$  positif, retourne la matrice carrée dont la première ligne est 1, 2, ...,  $n$ , la suivante  $n+1$ ,  $n+2$ , ...,  $2n$ , et ainsi de suite jusqu'à la dernière ligne <sup>a</sup>.

2 - À l'aide de la fonction `shuffle()` de `numpy.random`, effectuer une permutation aléatoire équiprobable des éléments d'une matrice quelconque donnée  $M$  et afficher la matrice obtenue.

<sup>a</sup>. qui est donc  $n(n-1)+1$ ,  $n(n-1)+2$ , ...,  $n*n$

**Mots-clefs :** fonctions `arange()`, `reshape()` du module `numpy`, fonction `shuffle()` du module `numpy.random`, commande `B = A.flatten()`, voir ci-dessous.

### Rappels :

- Étant donné une matrice à une dimension  $M$ , la commande `shuffle(M)` retourne une matrice à une dimension qui est une permutation de  $M$  choisie équiprobablement parmi toutes ses permutations possibles. La valeur initiale de  $M$  est perdue.
- La commande `B = A.flatten()` <sup>3</sup> retourne la matrice à une dimension  $B$  qui est en fait la matrice  $A$  écrite en ligne, les lignes successives de  $A$  étant écrites à la queue leu-leu.
- Si  $A$  est une matrice à une ou deux dimensions dont  $n$  est le nombre d'éléments, `A.reshape(p,q)` <sup>4</sup> où  $p$  et  $q$  sont des entiers tels que  $p*q = n$ , retourne une matrice à  $p$  lignes et  $q$  colonnes écrites dans l'ordre attendu <sup>5</sup>.

### Solution

1 - La fonction `papageno()` ci-dessous est une solution quasi évidente de la première question :

```
import numpy as np
def papageno(n):# n est un entier positif.
    A = np.arange(1,n*n+1)
    return A.reshape(n,n)
```

Exemple :

```
from papageno import *
papageno(125)
Out[16] :
array([[ 1, 2, 3, ..., 123, 124, 125],
       [126, 127, 128, ..., 248, 249, 250],
       [251, 252, 253, ..., 373, 374, 375],
       ...,
       [15251, 15252, 15253, ..., 15373, 15374, 15375],
       [15376, 15377, 15378, ..., 15498, 15499, 15500],
       [15501, 15502, 15503, ..., 15623, 15624, 15625]])
```

2 - La fonction `papagena()` ci-dessous est une solution plutôt facile de la deuxième question :

```
import numpy as np
from numpy.random import shuffle
```

3. `flatten()` est une méthode de l'objet `matrice`.

4. `reshape()` est une méthode de l'objet `matrice`.

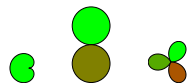
5.  $A$  ayant été écrite en ligne, les  $q$  premiers termes de cette ligne forment la première ligne de `A.reshape(p,q)`, etc.



```
def papagena(M) :
    N = M.flatten()
    shuffle(N)
    P = N.reshape(np.shape(M))
    return M,P
```

Exemple :

```
from papagena import *
M = (np.arange(1,26)).reshape(5,5)
papagena(M)
Out[3] :
(array([[ 1,  2,  3,  4,  5],
        [ 6,  7,  8,  9, 10],
        [11, 12, 13, 14, 15],
        [16, 17, 18, 19, 20],
        [21, 22, 23, 24, 25]]), array([[24, 20, 12,  8, 19],
        [18,  5,  4,  9, 23],
        [11, 16,  7, 22,  3],
        [ 2, 17, 25,  6,  1],
        [14, 10, 13, 21, 15]]))
```



## 5.4 [\*] Calculer la trace d'une matrice

Soit  $M$  une matrice à  $n$  lignes et  $m$  colonnes.

1 - Calculer la somme  $t1$  de ses éléments dont l'indice de ligne est égal à l'indice de colonne.

2 - Calculer la somme  $t2$  de ses éléments dont l'indice de ligne  $i$  et l'indice de colonne  $j$  vérifient  $i+j=\min(n,m)-1$ , les lignes et les colonnes étant numérotées à partir de 0).

**Mots-clefs :** boucle `pour` ; module `numpy` : extraire un bloc d'éléments d'une matrice, changer l'ordre des lignes ou des colonnes d'une matrice, fonctions `shape()`, `diag()`, `sum()`, `trace()` ; module `numpy.random` : fonction `rand()`, module `copy` : fonction `deepcopy`.

Si  $M$  est carrée ( $n=m$ ),  $t1$  est la somme de ses éléments diagonaux,  $t2$  la somme des éléments de sa deuxième diagonale. Si  $M$  n'est pas une matrice carrée, on se ramène immédiatement à ce cas. En effet, il est clair que  $t1$  et  $t2$  ne changent pas quand on remplace  $M$  par la sous-matrice de  $M$  formée par ses  $\min(n,m)$  premières lignes et ses  $\min(n,m)$  premières colonnes<sup>6</sup>.

### Solution

1 - Le script ci-dessous donne en fait 3 solutions de la première question, appliquées à un exemple.

- Afin de comparer ces solutions, nous avons ajouté le calcul des durées d'exécution.
- Au cours de la première solution, la matrice  $M$  est modifiée. Comme nous avons besoin de sa valeur initiale pour calculer les solutions suivantes, nous avons conservé cette valeur à l'aide de l'instruction `MM = deepcopy(M)`.
- Comme il est commode d'utiliser des matrices aléatoires<sup>7</sup> quand on a besoin d'une matrice pour tester un script, on a choisi ci-dessous une matrice  $M$  de taille (10000,120000) dont les éléments sont tirés au hasard suivant la loi uniforme sur l'intervalle  $[0,1]$ , indépendamment les uns des autres.

```
import numpy as np
from time import clock
M = np.random.rand(10000,120000)# Matrice à 1 200 000 000 éléments !
from copy import deepcopy
MM = deepcopy(M)# MM est une copie indépendante de M.
a = clock()
p = min(np.shape(M))# Minimum du 2-uplet np.shape(M).
M = M[0:p,0:p]# La valeur initiale de M est perdue. M est carrée.
t11 = M[0,0]# Premier terme de la diagonale de M.
for i in range(1,p):# i prend successivement les valeurs 1, ..., p-1
    t11 = t11 + M[i,i]
# t11 est la trace de M.
b = clock()
d11 = b-a
# Deuxième solution.
a = clock()
diag2 = MM[list(np.arange(0,10000,1)),list(np.arange(0,10000,1))].# Matrice
# à une dimension des éléments diagonaux de M
t12 = sum(diag2)# t12 est la trace de MM et de M.
b = clock()
d12 = b-a
# Troisième solution
a = clock()
t13 = np.trace(MM)# trace() est une fonction du module numpy.
```

6. commande `M=M[0:min(n,m),0:min(n,m)]`

7. parce qu'elles sont très faciles à engendrer

```
b = clock()
d13 = b-a
```

Commentaires : La solution n°1 est très simple car elle ne comporte qu'une boucle **pour**. La solution n°2 est plus sophistiquée car on y extrait les éléments de MM situés sur une ligne oblique<sup>8</sup>. La solution n°3 est fournie par Python : c'est la fonction **trace()** de **numpy**.

Suite des calculs, sur la console :

```
runfile('Chemin_du_script')
t11, t12, t13
Out[2] : (4993.9678710011149, 4993.9678710011158, 4993.9678710011158)
d11, d12, d13
Out[3] : (1.9125719999999973, 2.0644849999999977, 0.005915999999999144)
```

On voit que **trace()** est beaucoup plus rapide que les solutions 1 et 2. C'est cette fonction que l'on utilise normalement.

2 - À étant donné une matrice M, on peut la remplacer par la matrice carrée MM définie comme précédemment. La commande **N = MM[range(shape(MM)[0]-1, -1, -1), :]** retourne la matrice carrée N de même dimension que MM dont la première ligne est la dernière ligne de MM, etc<sup>9</sup>. Dans cette manipulation, les diagonales se sont échangées. En particulier, la première diagonale de N est la deuxième diagonale de MM dont la somme est donc **t2**. La matrice M étant donnée, le calcul de **t2** se résume donc maintenant à :

```
p = min(np.shape(M))
M = M[0:p, 0:p]
t2 = trace(M[range(p-1, -1, -1), :])
```

Exemple : dans une nouvelle console,

```
import numpy as np
M = np.random.rand(4, 6)
M
Out[3] :
array([[ 0.09844368,  0.36667626,  0.48009385,  0.87182384,  0.1740172 ,
         0.37642479],
       [ 0.38778403,  0.98906187,  0.61910549,  0.40341863,  0.59408361,
         0.18363526],
       [ 0.99841847,  0.1614101 ,  0.43666381,  0.59053185,  0.45606471,
         0.23099434],
       [ 0.33903413,  0.31377958,  0.12998992,  0.90110014,  0.19849328,
         0.84980818]])
p = min(np.shape(M))
M = M[0:p, 0:p]
M
Out[6] :
array([[ 0.09844368,  0.36667626,  0.48009385,  0.87182384],
       [ 0.38778403,  0.98906187,  0.61910549,  0.40341863],
       [ 0.99841847,  0.1614101 ,  0.43666381,  0.59053185],
       [ 0.33903413,  0.31377958,  0.12998992,  0.90110014]])
M = M[range(p-1, -1, -1), :]
M
```

8. commande **diag2 = MM[list(np.arange(0, 10000, 1)), list(np.arange(0, 10000, 1))]**

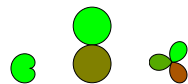
9. de même pour les lignes : la commande **P = MM[:, range(shape(MM)[1]-1, -1, -1)]** retourne la matrice P dont la première colonne est la dernière colonne de MM, etc

```

Out[8] :
array([[ 0.33903413,  0.31377958,  0.12998992,  0.90110014],
       [ 0.99841847,  0.1614101 ,  0.43666381,  0.59053185],
       [ 0.38778403,  0.98906187,  0.61910549,  0.40341863],
       [ 0.09844368,  0.36667626,  0.48009385,  0.87182384]])
t2 = np.trace(M)
t2
Out[10] : 1.9913735509512127
0.33903413+0.1614101+0.61910549+0.87182384
Out[11] : 1.9913735600000002

```

Le calcul à la main de la trace de M se fait avec les nombres affichés, qui sont arrondis, ce qui explique la légère différence avec t2.



## 5.5 **[\*\*]** Ce carré est-il magique ?

On appelle carré magique d'ordre  $n$  toute matrice carrée  $M$  d'ordre  $n$  dont les éléments sont les entiers de 1 à  $n^2$  disposés de sorte que leurs sommes sur chaque ligne, sur chaque colonne et sur les deux diagonales soient égales. La valeur commune de ces sommes est appelée constante magique de  $M$ .

1 - S'il existe un carré magique d'ordre  $n$ , combien vaut sa constante magique ?

2 - `LuoShu = array([[4,9,2],[3,5,7],[8,1,6]])` est-il un carré magique ? Si oui, quelle est sa constante magique ?

3 - Même question pour `Cazalas = array([[1,8,53,52,45,44,25,32],[64,57,12,13,20,21,40,33],[2,7,54,51,46,43,26,31],[63,58,11,14,19,22,39,34],[3,6,55,50,47,42,27,30],[62,59,10,15,18,23,38,35],[4,5,56,49,48,41,28,29],[61,60,9,16,17,24,37,36]])`.

4 - Même question pour `Franklin = array([[52,61,4,13,20,29,36,45],[14,3,62,51,46,35,30,19],[53,60,5,12,21,28,37,44],[11,6,59,54,43,38,27,22],[55,58,7,10,23,26,39,42],[9,8,57,56,41,40,25,24],[50,63,2,15,18,31,34,47],[16,1,64,49,48,33,32,17]])`.

**Mots-clefs :** `M.size`, `M.sum(axis=1)`, `M.sum(axis=0)`, `trace(M)`, `M[arange(n-1,-1,-1), arange(0,n)]` (voir ci-dessous), `concatenate(( , ),axis =)`, `min()`, `max()`, `shape()`, `reshape()`.

### Solution <sup>10</sup>

1 - La somme des éléments de ce carré est  $1 + 2 + \dots + n^2 = \frac{n^2(n^2+1)}{2}$ . Ce nombre est aussi  $n$  fois la constante magique <sup>11</sup>. La constante magique est donc  $\frac{n(n^2+1)}{2}$ .

**Solution n°1 des questions 2, 3 et 4** <sup>12</sup> : Ces questions sont faciles puisque, une matrice d'ordre  $n$  dont les éléments sont les entiers de 1 à  $n^2$  étant donnée, il suffit de calculer les sommes de ses éléments sur chaque ligne, chaque colonne et sur les deux diagonales et de les comparer. On pourra utiliser les commandes suivantes :

- `M.size`,  $M$  étant une matrice à une ou deux dimensions, retourne le nombre de ses éléments,
- `M.sum(axis=1)` retourne la matrice à une dimension des sommes des lignes de  $M$ ,
- `M.sum(axis=0)` retourne la matrice à une dimension des sommes des colonnes de  $M$ ,
- `trace(M)` retourne la somme des éléments de la diagonale principale de  $M$  <sup>13</sup>,
- `M[m1,m2]` où  $m1$  et  $m2$  sont des matrices  $(m, )$  de même longueur retourne la matrice  $(m, )$  dont les éléments sont ceux de  $M$  dont les indices de ligne forment  $m1$  et les indices de colonne forment  $m2$ . Les matrices  $m1$  et  $m2$  peuvent être remplacées par des listes. Par exemple, `M[arange(n-1,-1,-1), arange(0,n)]` est la deuxième diagonale de  $M$  <sup>14</sup> <sup>15</sup>,
- `np.concatenate((A,B),axis=0)` retourne la matrice obtenue en empilant verticalement les matrices  $A$  et  $B$  qui doivent avoir le même nombre de dimensions (soit matrices à une dimension, soit matrices à 2 dimensions) et le même nombre de colonnes. La commande `np.diag(M)[np.newaxis,:]` ajoute une dimension à `diag(M)`. C'est une matrice  $(1,n)$ .

10. Voir l'intéressant article [27] de Wikipedia sur les carrés magiques.

11. Il y a  $n$  lignes, la somme de chacune d'elles étant la constante magique.

12. Il est sans doute judicieux de se reporter directement à la solution n°2 des questions 2, 3 et 4 ci-dessous. Cette deuxième solution, quoique comportant des tests, se révèle en effet beaucoup plus rapide, car ces tests évitent de nombreux calculs.

13. `trace()` est une fonction du module `numpy`, voir l'exercice 5.4.

14. analogue de `diag(M)` qui retourne la première diagonale de  $M$

15. On peut aussi calculer la somme des éléments de la deuxième diagonale de  $M$  avec la commande : `trace(A[:,range(shape(A)[1]-1,-1,-1)])`. Voir l'exercice 5.4.

Cela donne la fonction `mascarille` suivante :

```
# M étant une matrice carrée d'ordre n dont les éléments
# sont les entiers de 1 à n*n, la fonction mascarille()
# retourne suivant le cas 'M est ou n'est pas un carré magique'
# ainsi, optionnellement (sans #), que le temps de calcul.
# Une autre option (##) est de retourner 1 si M est un carré magique, 0 sinon.
import numpy as np
from time import clock
def mascarille(M):
    #a = clock()
    n = M[:,0].size# Nombre de colonnes de M.
    MM = np.concatenate((M,M.T,np.diag(M)[np.newaxis,:],
                           np.diag(M[:,range(n-1,-1,-1))][np.newaxis,:]),axis=0)
    c = n*(n**2+1)//2# Constante magique.
    MMM = MM.sum(axis=1)
    #b = clock()
    if np.array_equal(MMM,c*np.ones((2*n+2,),int)):
        return "M est un carré magique"#,b-a##,1
    else:
        return "M n'est pas un carré magique"#,b-a##,0
```

Commentaires : Dans ce script, on empile donc verticalement les quatre matrices à deux dimensions `M`, `M.T`, `np.diag(M)[np.newaxis,:]` et `np.diag(M[:,range(n-1,-1,-1))][np.newaxis,:]`. On calcule les sommes de toutes les lignes de la matrice obtenue, qui produit une matrice à une dimension. On la compare avec la matrice à une dimension et `n` colonnes ne comportant que des `c`. On teste ensuite l'égalité de ces matrices.

Application aux matrices LuoShu, Cazalas et Franklin :

```
from mascarille import *
LuoShu = np.array([[4,9,2],[3,5,7],[8,1,6]])
mascarille(LuoShu)
Out[3]: 'M est un carré magique'
Cazalas = array([[1,8,53,52,45,44,25,32],
[64,57,12,13,20,21,40,33],
[2,7,54,51,46,43,26,31],
[63,58,11,14,19,22,39,34],
[3,6,55,50,47,42,27,30],
[62,59,10,15,18,23,38,35],
[4,5,56,49,48,41,28,29],
[61,60,9,16,17,24,37,36]])
mascarille(Cazalas)
Out[5]: 'M est un carré magique'
Franklin = array([[52,61,4,13,20,29,36,45],
[14,3,62,51,46,35,30,19],
[53,60,5,12,21,28,37,44],
[11,6,59,54,43,38,27,22],
[55,58,7,10,23,26,39,42],
[9,8,57,56,41,40,25,24],
[50,63,2,15,18,31,34,47],
[16,1,64,49,48,33,32,17]])
mascarille(Franklin)
Out[7]: "M n'est pas un carré magique"
```

On constate que les deux premiers carrés proposés sont magiques tandis que le troisième ne l'est pas, contrairement à ce qui est dit dans [27]. Regardons le détail des calculs :

```
Franklin.sum(axis=1)
Out[9]: array([260, 260, 260, 260, 260, 260, 260, 260])
Franklin.sum(axis=0)
Out[10]: array([260, 260, 260, 260, 260, 260, 260, 260])
n = Franklin[:,0].size
array([trace(Franklin),sum(Franklin[arange(n-1,-1,-1), arange(0,n)])])
Out[12]: array([228, 292])
```

On constate que les sommes des lignes et des colonnes de la matrice de Franklin sont égales à 260 alors que sa trace vaut 228 et la somme des éléments de la deuxième diagonale 292!

#### Solution n°2 des questions 2, 3 et 4 :

La définition de la fonction `marotte()` qui suit contient des tests qui permettent d'arrêter des calculs inutiles<sup>16</sup>. Ces calculs sont faits quand on utilise la fonction `mascarille()` ci-dessus.

```
# Si M est une matrice carrée dont les éléments sont les entiers de 1 à n*n,
# marotte(M) retourne 1 si M est un carré magique, 0 sinon.
# marotte() semble beaucoup plus rapide que mascarille().
import numpy as np
def marotte(M):
    n = np.shape(M)[0]
    c = n*(n*n+1)//2
    if (sum(np.diag(M))!= c) or (sum(np.diag(M[:,range(n-1,-1,-1)]))!= c):
        return 0
    for i in range(n-1):
        if (sum(M[i,:])!= c) or (sum(M[:,i])!= c):
            return 0
    return 1
```

#### Exemple :

Appliquons `marotte()` à une matrice `M` d'un million de termes<sup>17</sup> en faisant apparaître la durée du calcul :

```
import numpy as np
import marotte as mar
from time import clock
M = np.arange(1,1000001)# M = np.array([1, ...,1000000])
np.random.shuffle(M)#Brouillage aléatoire des éléments de M.
M = M.reshape(1000,1000)# M est transformée en matrice (1000,1000)
a = clock()# Déclenchement du chronomètre.
x = mar.marotte(M)
b = clock()# Arrêt du chrnomètre.
print(x,b-a)
```

Exécutons-le :

```
runfile('Chemin_du_script')
0 [[330450 195238 123356 ..., 805557 635024 471971]
 [ 43855 957950 912146 ..., 430652 269016 302098]
 [ 25366 835825 331316 ..., 972094 101682 346065]
 ...,

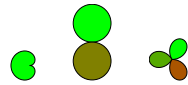
```

16. grâce à la commande `return`

17. `n = 1000`

```
[473576 666289 964539 ... , 219784 721260 588693]  
[396011 694839 760462 ... , 643871 620982 497566]  
[345506 86068 334800 ... , 811284 731600 125029]] 0.00019300000000003318
```

L'exécution de `marotte()` a duré moins de deux dix-millièmes de seconde. M n'est pas un carré magique.





## 5.6 **[\*\*\*]** Transposer une matrice

Programmer une fonction qui, étant donné une matrice à deux dimensions **A**, retourne la matrice **B** dont la première colonne est la première ligne de **A**, la seconde la deuxième ligne de **A**, etc. **B** s'appelle la transposée de **A**.

**Mots-clefs** : Boucle pour imbriquée dans une autre ; commande **range** ; module **numpy** : fonctions **np.shape()** (dimension d'une matrice), **np.zeros()**, **np.concatenate(( , ),axis = )** (empilements horizontaux et verticaux de matrices), **np.transpose()** ; méthode (de l'objet matrice) **.T**.

**Solution n°1** - Il y a évidemment des commandes de Python qui retournent la transposée d'une matrice à deux dimensions<sup>18, 19</sup> :

```
import numpy as np
A = np.array([[1.5,2,3,4,5],[6,7,8,9,10],[11,12,13,14,15]])
#A est une matrice (3,5) de nombres réels.
print(A)
Bt = np.transpose(A)
print(Bt)#Matrice (5,3).
BT = A.T
print(BT)# Matrice (5,3). Bt et BT sont égales à la transposée de A.
```

Exécutons ce script dans la console :

```
runfile('Chemin_du_script_précédent')
[[ 1.5  2.   3.   4.   5. ]
 [ 6.   7.   8.   9.  10. ]
 [ 11.  12.  13.  14.  15. ]]
[[ 1.5  6.  11. ]
 [ 2.   7.  12. ]
 [ 3.   8.  13. ]
 [ 4.   9.  14. ]
 [ 5.  10.  15. ]]
[[ 1.5  6.  11. ]
 [ 2.   7.  12. ]
 [ 3.   8.  13. ]
 [ 4.   9.  14. ]
 [ 5.  10.  15. ]]
```

Une matrice à deux dimensions qui n'a qu'une ligne est transformée en une matrice à deux dimensions qui n'a qu'une colonne et inversement. **transpose()** et **.T** n'agissent pas sur les matrices à une dimension, comme le montre l'exemple suivant :

```
import numpy as np
A = np.array([1,2,3,4,5])# A est une matrice (m, ) d'entiers.
print(A)
Bt = np.transpose(A)
print(Bt)# Bt est une matrice (5,1).
BT = A.T
print(BT)# Bt est une matrice (5,1).
```

18. **transpose()** est une fonction du module **numpy** qui retourne la transposée de toute matrice à deux dimensions.

19. **.T** est une méthode de l'objet matrice - à deux dimensions - qui transpose cette matrice.

Exécutons ce script dans la console :

```
runfile('Chemin_du_script_précédent')
[1 2 3 4 5]
[1 2 3 4 5]
[1 2 3 4 5]
```

**Solution n°2** - Programmons maintenant nous-mêmes une fonction transposant toute matrice  $A$  à deux dimensions. Pour cela, il suffit de remarquer que l'élément qui est sur la  $i^{\text{ème}}$  ligne et la  $j^{\text{ème}}$  colonne de la matrice transposée  $B$  est  $A[j,i]$  <sup>20</sup>. Si  $l$  et  $c$  désignent respectivement le nombre de lignes et le nombre de colonnes de  $A$  <sup>21</sup>,  $i$  courra de 1 à  $c$  et  $j$  courra de 1 à  $l$ . Il y aura donc une boucle pour imbriquée dans une boucle pour dans l'algorithme définissant la fonction appelée `transpoze()` ci-dessous :

```
import numpy as np
def transpoze(A):# A est une matrice à 2 dimensions données.
    a = type(A[0,0])# a est le type des éléments de A.
    (l,c) = np.shape(A)
    B = np.zeros((c,l),a)
    # B est une matrice de 0 de même type que les éléments de A.
    # La dimension de B est (c,l).
    for i in range(c):
        for j in range(l):
            B[i,j] = A[j,i]
    return B
```

Exemple :

```
import numpy as np
A = np.random.rand(3,5)
A
Out[3]:
array([[ 0.49253415,  0.8270516 ,  0.30327453,  0.11759094,  0.19154943],
       [ 0.2482007 ,  0.58941235,  0.55208151,  0.21272643,  0.23828777],
       [ 0.08465715,  0.0335026 ,  0.93107188,  0.96414524,  0.96205538]])
runfile('Chemin_de_la_fonction_transpoze')
A = transpoze(A)
A
Out[6]:
array([[ 0.49253415,  0.2482007 ,  0.08465715],
       [ 0.8270516 ,  0.58941235,  0.0335026 ],
       [ 0.30327453,  0.55208151,  0.93107188],
       [ 0.11759094,  0.21272643,  0.96414524],
       [ 0.19154943,  0.23828777,  0.96205538]])
transpoze(A)
Out[7]:
array([[ 0.49253415,  0.8270516 ,  0.30327453,  0.11759094,  0.19154943],
       [ 0.2482007 ,  0.58941235,  0.55208151,  0.21272643,  0.23828777],
       [ 0.08465715,  0.0335026 ,  0.93107188,  0.96414524,  0.96205538]])
```

On constate, ce qui était évident, que la transposée de la transposée de  $A$  est  $A$ .

**Solution n°3** - Dans l'espoir d'obtenir un algorithme plus rapide, essayons de manipuler directement les

---

20. commande `B[i,j] = A[j,i]`

21. donnés par la commande `shape(A)`, qui retourne le 2-uplet `(l,c)` ; `l = shape(A)[0]` ; `c = shape(A)[1]`

lignes ou les colonnes de A. On sait que si A est une matrice à deux dimensions et si nous en extrayons la première colonne (commande `A[:,0]`), on obtient une matrice (m, 1) qui peut être transformée en une matrice à deux dimensions (1,c) (commande `A[:,0][np.newaxis,:]`). Il reste à faire de même pour les autres colonnes de A et à empiler les l matrices (1,c) obtenues à l'aide de la fonction `concatenate((,),axis = 0)` de numpy<sup>22</sup>.

```
import numpy as np
def transpozze(A):# A est une matrice donnée à 2 dimensions.
    (l,c) = np.shape(A)
    B = A[:,0][np.newaxis,:]# Matrice à 2 dimensions réduite à une ligne
    # de l colonnes.
    for i in range(1,c):
        B = np.concatenate((B,A[:,i][np.newaxis,:]),axis = 0)
    return B# Matrice à c lignes et l colonnes : transposée de A.
```

Exemple :

```
runfile('Chemin_de_la_fonction_transpozze')
A = np.random.rand(4,6)
A
Out[3]:
array([[ 0.14375364,  0.41827656,  0.10152442,  0.05841431,  0.1026309 ,
         0.03954335],
       [ 0.79758356,  0.50179013,  0.95236493,  0.94198123,  0.54791014,
         0.83251176],
       [ 0.75258107,  0.26999299,  0.93020474,  0.46672265,  0.31675741,
         0.87933763],
       [ 0.0363524 ,  0.8232996 ,  0.13521136,  0.53129757,  0.38076294,
         0.17876873]])
transpozze(A)
Out[4]:
array([[ 0.14375364,  0.79758356,  0.75258107,  0.0363524 ],
       [ 0.41827656,  0.50179013,  0.26999299,  0.8232996 ],
       [ 0.10152442,  0.95236493,  0.93020474,  0.13521136],
       [ 0.05841431,  0.94198123,  0.46672265,  0.53129757],
       [ 0.1026309 ,  0.54791014,  0.31675741,  0.38076294],
       [ 0.03954335,  0.83251176,  0.87933763,  0.17876873]])
```

Supplément : Comparons sur un exemple les fonctions `transpose()`, `transpoze()` et `transpozze()`. On s'attend à ce que la première soit la plus rapide, suivie par la troisième. Pour cela, exécutons le script suivant dans une nouvelle console.

```
import numpy as np
import transpoze as z
import transpozze as zz
from time import clock
A = np.random.rand(1000,2500)
a = clock()
B = np.transpose(A)
b = clock()
d = b - a
```

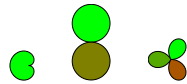
22. `np.concatenate((A,B),axis = 0)` empile verticalement les matrices à deux dimensions A et B pourvu qu'elles aient le même nombre de colonnes.

```
a = clock()
Bz = z.transpose(A)
b = clock()
dz = b - a
a = clock()
Bzz = zz.transpozze(A)
b = clock()
dzz = b - a
print(d, dz, dzz)
```

On obtient :

```
runfile('Chemin_du_script')
Reloaded modules: transpose, transpozze
0.0016400000000018622 0.68638099999999972 22.59821
```

`transpozze()` est la plus lente!



## 5.7 [\*\*\*\*] Écrire toutes les permutations de (1, ..., n).

1 - M désignant une matrice quelconque d'entiers à `li` lignes et `co` colonnes, `n` un entier quelconque, programmer une fonction qui retourne la matrice `Sortie` qui empile verticalement les `co+1` matrices obtenues en adjoignant à M une colonne A à `li` lignes dont tous les éléments sont égaux à `n`, A étant placée d'abord devant M, puis entre la première et deuxième colonne de M<sup>a</sup>, etc, jusqu'à ce que A devienne sa dernière colonne. La matrice obtenue aura `co+1` colonnes et `(n+1).li` lignes.

2 - Écrire toutes les permutations de (1, ..., n).

a. ce qui revient à décaler A d'un cran vers la droite ou d'échanger les deux premières colonnes

**Mots-clefs** : appel de modules et de fonctions ; module `numpy` : fonctions `ones(( , ), int)`, `zeros(( , ), float)`, `concatenate(( , ), axis = )`, échange de colonnes d'une matrice, extraction d'une sous-matrice ; méthode `shape()` des matrices ; fonction `clock()` du module `time`.

### Solution

1 - La fonction `nabucco()` définie ci-dessous<sup>23</sup> suit fidèlement l'énoncé de la question 1<sup>24</sup> :

```
# Fonction auxiliaire sans intérêt en soi.
# M est par hypothèse une matrice d'entiers à deux dimensions, n un entier.
def nabucco(M,n):
    import numpy as np
    (li ,co) = M.shape
    A = n*np.ones((li ,1),int)# Colonne de n à li lignes.
    Sortie = np.zeros(((co+1)*li ,co+1),int)# Initialisation de la future
    # sortie de nabucco().
    B = np.concatenate((A,M),axis=1)
    # B est une matrice à li lignes et co+1 colonnes obtenue
    # en mettant A devant M.
    Sortie[0:li ,:] = B# Début du calcul de la sortie de nabucco().
    # Le bloc de zéros de la valeur initiale de Sortie, porté par
    # les lignes 0, ..., li-1, est remplacé par B.
    for i in range(co):
        B[:,[i,i+1]] = B[:,[i+1,i]]# Echange des colonnes i et i+1 de B. En
        # clair, dans B, la colonne de n se déplace d'un cran vers la droite.
        Sortie[(i+1)*li:(i+2)*li ,0:co+1] = B# Le bloc de zéros de Sortie
        # porté par les lignes d'indices (i+1)*li à (i+2)*li-1 est remplacé
        # par B. A la fin, Sortie a (co+1).li lignes et co+1 colonnes.
    return Sortie
```

Voici une autre version de la fonction `nabucco`<sup>25</sup> :

```
# nabucco est une fonction auxiliaire sans intérêt en soi.
# M est par hypothèse une matrice d'entiers à deux dimensions, n un entier.
import numpy as np
def nabucco(M,n):
    (l ,c) = M.shape
```

23. script commenté

24. Elle sera utilisée pour définir la fonction `permut()` qui suit.

25. moins rapide que la première version de `nabucco()`

```

S = n*np.ones((1,c+1),int)
for i in range(c+1):
    A = n*np.ones((1,c+1),int)
    A[:,0:i] = M[:,0:i]
    A[:,i+1:c+1] = M[:,i:c]
    S = np.concatenate((S,A),axis = 0)
return S[1:1*(c+2),:]

```

Commentaires : La première valeur de **S** est une matrice de 1 à 1 lignes et **c+1** colonnes qu'il faudra retirer à la fin et qui permet d'empiler les matrices d'intérêt à l'aide de la fonction **np.concatenate()**.

Exemples :

Exemple n°1 :

```

runfile('Chemin_de_la_fonction_nabucco')
M = np.array([[1]])
nabucco(M,17)
Out[3]:
array([[17,  1],
       [ 1, 17]])

```

Exemple n°2 :

```

from nabucco import *
M = np.array([[1,2,3],[4,5,6],[7,8,9],[10,11,12],[13,14,15],[16,17,18]])
nabucco(M,-111)
Out[3]:
array([[ -111,    1,    2,    3],
       [ -111,    4,    5,    6],
       [ -111,    7,    8,    9],
       [ -111,   10,   11,   12],
       [ -111,   13,   14,   15],
       [ -111,   16,   17,   18],
       [    1, -111,    2,    3],
       [    4, -111,    5,    6],
       [    7, -111,    8,    9],
       [   10, -111,   11,   12],
       [   13, -111,   14,   15],
       [   16, -111,   17,   18],
       [    1,    2, -111,    3],
       [    4,    5, -111,    6],
       [    7,    8, -111,    9],
       [   10,   11, -111,   12],
       [   13,   14, -111,   15],
       [   16,   17, -111,   18],
       [    1,    2,    3, -111],
       [    4,    5,    6, -111],
       [    7,    8,    9, -111],
       [   10,   11,   12, -111],
       [   13,   14,   15, -111],
       [   16,   17,   18, -111]])

```

2 - Dans le script ci-dessous<sup>26</sup>, pour calculer toutes les permutations des entiers de 1 à  $n$  écrites en colonne<sup>27</sup>, on écrit toutes les permutations de 1, puis de (1, 2), puis de (1, 2, 3), *etc*<sup>28</sup>, ce que l'on peut représenter par :

$$1 \longrightarrow \begin{smallmatrix} 2 & 1 \\ 1 & 2 \end{smallmatrix} \longrightarrow \begin{smallmatrix} 3 & 2 & 1 \\ 3 & 1 & 2 \\ 2 & 3 & 1 \\ 1 & 3 & 2 \\ 2 & 1 & 3 \\ 1 & 2 & 3 \end{smallmatrix} \longrightarrow$$

et on s'arrête quand on a obtenu les permutations de  $(1, 2, \dots, n)$ .

```
# La fonction permut() associe, à tout entier positif n,
# la matrice à n! lignes et n colonnes des permutations de 1, ..., n,
# chaque ligne étant une permutation. Optionnellement, elle retourne aussi
# la durée du calcul.
import numpy as np
import nabucco as nab
from time import clock
def permut(n):
    a = clock()
    M = np.array([[1]]) # Initialisation de M.
    for i in range(1,n):
        M = nab.nabucco(M, i+1)
    b = clock()
    return M, b-a
```

Exemples :

```
runfile('Chemin_de_la_fonction_permut')
permut(10)
Out[2]:
(array([[10,  9,  8, ...,  3,  2,  1],
        [10,  9,  8, ...,  3,  1,  2],
        [10,  9,  8, ...,  2,  3,  1],
        ...,
        [ 1,  3,  2, ...,  8,  9, 10],
        [ 2,  1,  3, ...,  8,  9, 10],
        [ 1,  2,  3, ...,  8,  9, 10]]), 0.36113399999999984)
permut(11)
Out[3]:
(array([[11, 10,  9, ...,  3,  2,  1],
        [11, 10,  9, ...,  3,  1,  2],
        [11, 10,  9, ...,  2,  3,  1],
        ...,
        [ 1,  3,  2, ...,  9, 10, 11],
        [ 2,  1,  3, ...,  9, 10, 11],
        [ 1,  2,  3, ...,  9, 10, 11]]), 5.130067)
permut(12)
Out[4]:
(array([[12, 11, 10, ...,  3,  2,  1],
        [12, 11, 10, ...,  3,  1,  2],
        [12, 11, 10, ...,  2,  3,  1],
        ...,
```

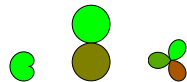
26. script commenté; la fonction définie a été appelée `permut()`

27. ce qui donnera une matrice à  $n!$  lignes et à  $n$  colonnes

28. à l'aide de la fonction `nabucco()`

```
[ 1,  3,  2, ..., 10, 11, 12],
[ 2,  1,  3, ..., 10, 11, 12],
[ 1,  2,  3, ..., 10, 11, 12]]) , 127.04567899999999) Populating
the interactive namespace from numpy and matplotlib
permut(13)
Kernel est mort, redémarré
```

On constate que les temps de calcul augmentent très vite avec **n** parce que la matrice `permut(n)` grandit très vite. Par exemple, `permut(12)` a `factorial(12) = 479 001 600` lignes et  $12 \times 12 = 144$  éléments. C'est beaucoup. À partir de **n=13**, il y a un dépassement de capacité de mémoire<sup>29</sup>.




---

29. `factorial(13) = 6 227 020 800` et `13*factorial(13) = 80 951 270 400`.



## 5.8 [\*\*\*\*] Combien y a-t-il de carrés magiques d'ordre 3 ?

Combien y a-t-il des carrés magiques d'ordre 3 ?

**Mots-clefs** : permutations, appeler une fonction ; module `math` : fonction `factorial()` ; module `time` : fonction `clock()` ; module `numpy` : matrices à une dimension (fonctions `min()`, `max()`, méthode `reshape()`), matrices à deux dimensions (extraction d'une ligne, somme sur les lignes et les colonnes, fonction `trace()`), conversion d'une matrice à une dimension en une matrice à deux dimensions ; listes : commande `append()`.

Cet exercice fait suite aux exercices 5.5 et 5.7.

### Solution :

- Nous allons engendrer toutes les matrices à  $n$  lignes et à  $n$  colonnes dont les éléments sont tous les entiers de 1 à  $n*n$ , à l'aide de la fonction `permut()`<sup>30</sup> de l'exercice 5.7<sup>31</sup>.
- puis nous les prendrons une par une. Quand nous tomberons sur un carré magique<sup>32</sup>, nous l'ajouterons à la liste des carrés magiques déjà trouvés (liste initialisée à `[]`) et nous ajouterons 1 à `c`, un compteur initialisé à 0.

L'algorithme suivant résout le problème posé. Cette fonction, appelée `cmagique()`, retourne aussi la durée du calcul et le nombre de carrés magiques trouvés. C'est donc un 3-uplet qui sera retourné.

```
# La fonction cmagique() retourne, pour tout entier positif n,
# le 3-uplet formé de la durée du calcul, de la liste A
# des carrés magiques d'ordre n et du nombre de ces carrés magiques.
import permut as per# Version de permut qui retourne
# la matrice des permutations.
import marotte as mar
from time import clock
from math import factorial
def cmagique(n) :
    a = clock()
    A = []# Liste vide. Initialisation de la liste des carrés magiques
    # d'ordre n.
    M = per.permut(n*n)# M est une matrice a (n*n)! lignes et
    # n*n colonnes.
    m = factorial(n*n)
    for i in range(m) :
        N = M[i]# M[i] est une matrice a une dimension à n*n colonnes
        # (ligne de M d'indice i).
        P = N.reshape(n, n)
        if mar.marotte(P) == 1 :
            A.append(P)
    b = clock()
    return b-a, A, len(A)# len(A) est la longueur de A.
```

`len(A)` est évidemment le nombre de carrés magiques d'ordre  $n$  trouvés.

**Exemples** : Calculons les carrés magiques d'ordre 1, 2, 3, ...<sup>33</sup>

30. commande `permut(n*n)`

31. Nous confondons bien entendu chacune de ces matrices avec la permutation correspondante de 1, ...,  $n*n$ .

32. identifié à l'aide de la fonction `marotte()` de l'exercice 5.5

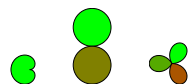
33. Les nombres de carrés magiques obtenus pourront être différents de ceux que donne [27] parce que cet article de Wikipedia compte comme identiques des carrés magiques qui se déduisent les uns des autres moyennant une transformation simple.

```

from cmagique import *
cmagique(1)
Out[2] : (0.00071200000000000459, [array([[1]])], 1)
cmagique(2)
Out[3] : (0.0053730000000000072, [], 0)
cmagique(3)
Out[4] :
(14.2448490000000002, [array([[4, 9, 2],
    [3, 5, 7],
    [8, 1, 6]]), array([[2, 9, 4],
    [7, 5, 3],
    [6, 1, 8]]), array([[4, 3, 8],
    [9, 5, 1],
    [2, 7, 6]]), array([[2, 7, 6],
    [9, 5, 1],
    [4, 3, 8]]), array([[8, 3, 4],
    [1, 5, 9],
    [6, 7, 2]]), array([[6, 7, 2],
    [1, 5, 9],
    [8, 3, 4]]), array([[8, 1, 6],
    [3, 5, 7],
    [4, 9, 2]]), array([[6, 1, 8],
    [7, 5, 3],
    [2, 9, 4]])], 8)
Populating the interactive namespace from numpy
and matplotlib
cmagique(4)
Kernel est mort, redémarré

```

Il y a donc 8 carrés magiques d'ordre 3 dont la liste est donnée à l'issue d'un calcul qui est long. À partir de  $n = 4$ , ces calculs deviennent impossibles (dans ce cas, la matrice  $M$  a 20 922 789 888 000 lignes et 16 colonnes, ce qui est beaucoup). De plus, la fonction `cmagique()` provoque de nombreux calculs inutiles.



## 5.9 [\*\*\*] Construction des premiers flocons de Koch avec plot

Un polygone K0 étant donné sous la forme d'une matrice à deux lignes (ligne 1 : ligne des abscisses des sommets, ligne 2 : ligne des ordonnées des sommets), programmer une fonction qui retourne le polygone Kn,  $n \geq 1$  ainsi défini :

Polygone K1 : le polygone K1 est construit à partir de K0 en remplaçant le premier côté AB de K0 par la ligne brisée AMGNB, M étant au premier tiers de AB à partir de A, N étant au second tiers, G étant positionné à droite de AB quand on va de A vers B, de sorte que le triangle MGN soit équilatéral, puis en faisant de même avec le deuxième côté de K0 et ainsi de suite jusqu'au dernier côté de K0.

Polygone Kn : Si  $n \geq 2$ , on construit de même K2 à partir de K1, puis K3 à partir de K2 et ainsi de suite jusqu'à Kn.

Si K0 est un hexagone régulier, Kn s'appelle n<sup>ième</sup> flocon de Koch.

**Mots-clefs** : Définir et appeler une fonction ; module `numpy` : fonctions vectorisées, matrices ; module `matplotlib`.

### Solution

La solution proposée ici se réduit à du calcul matriciel. On peut se reporter, pour le détail des calculs, à l'exercice analogue écrit pour *scilab* sur ce site, à l'adresse :

Accueil > Activités > Première S > Algorithmique > Le flocon de Koch sans la tortue.

La solution proposée comprend 3 fonctions.

- La fonction `coque()` ci-dessous retourne, un segment AB étant donné, la ligne brisée AMGN décrite dans l'énoncé. En option (`##`), elle peut retourner la ligne brisée AMGNB.

```
import numpy as np

def coque(S):# Le segment AB est donné sous la forme d'une matrice S
    # à 2 lignes et 2 colonnes.
    # Première ligne : abscisses de A et B.
    # Deuxième ligne : ordonnées de A et B.
    xA,xB,yA,yB = S[0,0],S[0,1],S[1,0],S[1,1]
    xM=2/3*xA+1/3*xB
    yM=2/3*yA+1/3*yB
    xN=1/3*xA+2/3*xB
    yN=1/3*yA+2/3*yB
    xG=(xA+xB)/2+(yB-yA)/(2*np.sqrt(3))
    yG=(yA+yB)/2+(xA-xB)/(2*np.sqrt(3));
    S1=np.array([[xA,xM,xG,xN],[yA,yM,yG,yN]])
    # S1 est la matrice à 2 lignes et 4 colonnes qui représente AMGN.
    ## S1=np.array([[xA,xM,xG,xN,xB],[yA,yM,yG,yN,yB]]).# Matrice à 2 lignes
    ## et 5 colonnes qui représente AMGNB.
    return(S1)
```

- Étant donné un polygone K0, la fonction `floconK1()` ci-dessous retourne un polygone K1 qui est issu de K0 dont tous les côtés ont successivement été transformés par `coque()`. Si K0 est un polygone régulier, K1 est le premier flocon de Koch.

```
# Un polygone K0 étant donné sous la forme d'une matrice à deux lignes
# (ligne 1, ligne des abscisses des sommets,
# ligne 2, ligne des ordonnées des sommets),
```

```
# la fonction floconK1() ci-dessous retourne le polygone K1
# décrit dans l'énoncé sous la forme d'une matrice à deux lignes.
```

```
import numpy as np
from coque import coque

def floconK1(K0):# K0 est une matrice à deux lignes.
    c = K0.shape[1] Nombre de colonnes de K0.
    K1 = np.zeros((2,4*(c-1)+1),dtype=float)
    for i in range(0,c-1):
        K1[:,4*i:4*(i+1)] = coque(K0[:,i:i+2])
    K1[:,4*(c-1)] = K0[:,c-1]
    return K1
```

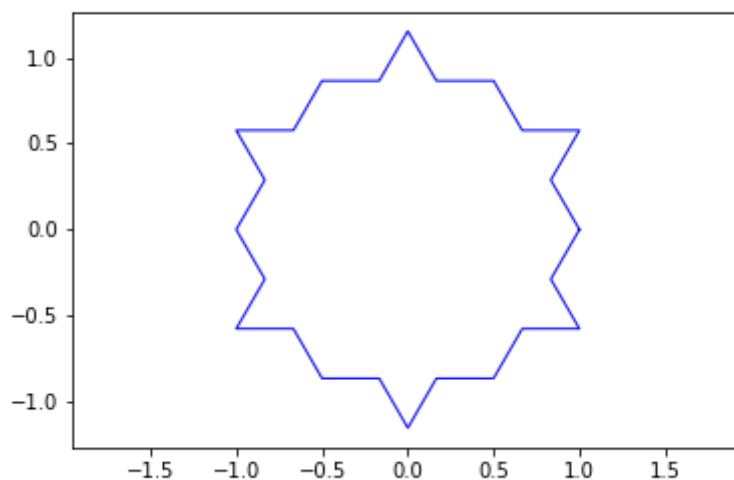
Tracé de K1 : On peut avoir la curiosité d'exhiber le premier flocon de Koch. Pour cela, il faut définir l'hexagone régulier K0 et ajouter des instructions graphiques, ce qui donne le script suivant :

```
from floconK1 import floconK1

# Définition de K0, hexagone régulier convexe.
theta = np.linspace(0,2*np.pi,7)
x = np.cos(theta)# x est une matrice-ligne à 7 éléments.
y = np.sin(theta)# y est une matrice-ligne à 7 éléments.
K0 = np.array([x,y])
K1 = floconK1(K0)

# Tracé de K1 :
import matplotlib.pyplot as plt
plt.figure()# Création d'une figure.
plt.axis('equal')
plt.plot(K1[0,:],K1[1,:], color="blue", linewidth=1.0, linestyle="-")
plt.savefig("K1",dpi=72)
```

ce qui donne :



- Comme on passe de K1 à K2 comme on est passé de K0 à K1, une simple boucle suffit pour obtenir Kn :

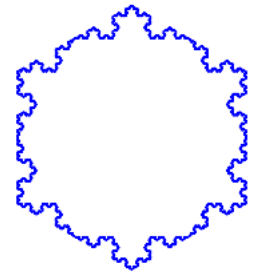
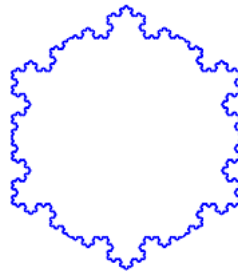
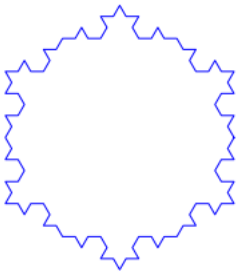
```
from floconK1 import floconK1
```

```

def floconKn(n) :# n est un entier >= 1.
    theta = np.linspace(0,2*np.pi,7)
    x = np.cos(theta)# x est une matrice-ligne à 7 éléments.
    y = np.sin(theta)# y de même.
    K0 = np.array([x,y])# K0 est un hexagone régulier.
    M = K0
    for p in range(1,n+1):
        M = floconK1(M)
    return M

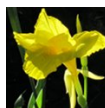
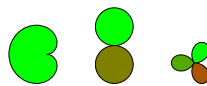
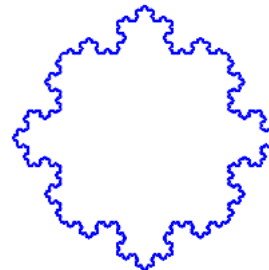
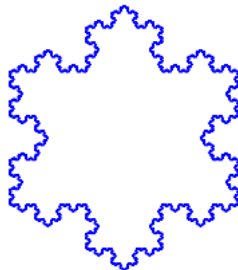
```

En ajoutant des instructions de tracé comme précédemment, voici les flocons K2, K5, K10 :



Le calcul de K10 est un peu long. Pour calculer les flocons suivants, on peut procéder pas à pas pour éviter des calculs inutiles : K11 = floconK1(K10), etc ...

Enfin, K0 peut être un polygone quelconque. Voici les tracés de K10 quand K0 est un triangle équilatéral puis un carré :



## Chapitre 6

# La tortue de Python

---

Liste des exercices :

Énoncé n° 6.1 [] : Médiannes concourantes d'un triangle

Énoncé n° 6.2 [\*] : Illustration du théorème de Pythagore avec la tortue de Python

Énoncé n° 6.3 [\*] : La tortue et les représentations graphiques

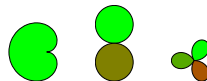
Énoncé n° 6.4 [\*\*\*] : Algorithme de Bresenham

Énoncé n° 6.5 [\*\*\*] : Construction itérative des triangles de Sierpinski

---

Le module `Turtle` offre peu d'intérêt. Il est par conséquent présenté dans les manuels avec une liste assez sommaire de commandes<sup>1</sup>. La documentation officielle, voir [21], fait quand même 43 pages qui ouvrent d'autres possibilités. Les tracés de polygones sont le domaine privilégié de la tortue.

La fenêtre graphique de la tortue est graduée en pixels<sup>2</sup>, les angles sont, par défaut, comptés en degrés.



---

1. Voir par exemple[1], 1 - Mise en route, 2. Tortue (Scratch avec Python).

2. Par défaut, sa largeur est 950 pixels, sa hauteur 800 pixels

## 6.1 [\*] Médianes concourantes d'un triangle

Vérifier expérimentalement que les médianes d'un triangle sont concourantes avec la tortue de Python

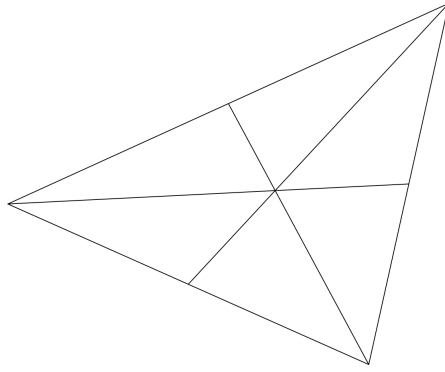
**Mots-clefs** : commandes `goto()`, `up()` et `down()`, `hideturtle()`, coordonnées du milieu d'un segment.

### Solution

C'est un exercice de premier contact avec la tortue, extrêmement simple. Le script suivant est commenté.

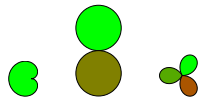
```
# La fonction médianes ci-dessous trace un triangle donné par
# les coordonnées de ses 3 sommets, puis trace ses 3 médianes.
import turtle as tu
def médianes(x1,y1,x2,y2,x3,y3):
    tu.reset()# Efface les dessins précédents, remet la tortue
    # dans sa position initiale, à savoir :
    # tu.position() —> (0.00,0.00)
    # tu.heading() —> 0.0
    # Tortue descendue.
    # tu.screenize(950,1000)# Taille standard de la toile à dessin.
    tu.speed(0)# vitesse maximum de la tortue.
    tu.hideturtle()# Cache la tortue.
    tu.up()# Tortue élevée.
    tu.goto(x1,y1)# Aller à (x1,y1).
    tu.down()# Tortue descendue.
    tu.goto(x2,y2)
    tu.goto(x3,y3)
    tu.goto(x1,y1)
    tu.up()
    tu.goto((x1+x2)/2,(y1+y2)/2)
    tu.down()
    tu.goto(x3,y3)
    tu.up()
    tu.goto((x2+x3)/2,(y2+y3)/2)
    tu.down()
    tu.goto(x1,y1)
    tu.up()
    tu.goto((x3+x1)/2,(y3+y1)/2)
    tu.down()
    tu.goto(x2,y2)
    tu.up()
    return
```

Voici ce que l'on obtient :



Le seul inconvénient rencontré est de bien choisir le triangle de manière à ce qu'il apparaisse tout entier dans la fenêtre graphique. C'est assez irritant. Heureusement, on peut aussi jouer sur la taille de la toile : la commande `tu.screenSize(950,1000)` imposerait la taille standard et est donc inutile. Mais on peut l'utiliser pour augmenter cette taille.

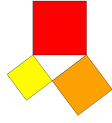
Bien sûr, cette vérification n'a aucune valeur, voir l'exercice [6.4](#), qui tient compte de la pixellisation des images.





## 6.2 [\*] Illustration du théorème de Pythagore

Tracer le dessin suivant à l'aide de la tortue :



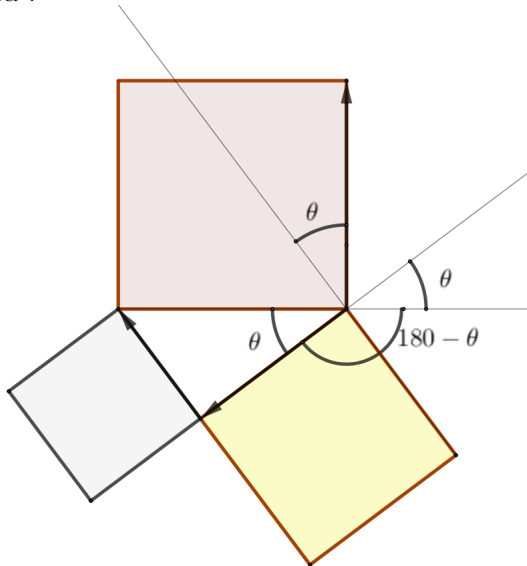
Le triangle est rectangle. On pourra placer son hypoténuse sur l'axe des abscisses, l'angle droit en-dessous, les longueurs des côtés étant 60, 80 et 100.

**Mots-clefs :** Déplacements en avant et en arrière, rotations de la tortue, polygones en couleur, remplissage de polygones.

### Solution

On pourrait dire que le dessin demandé est le logo du théorème de Pythagore. On le voit souvent. Ceci dit, cet exercice est seulement un exercice de maniement, une prise de contact de la tortue.

La relation  $60^2 + 80^2 = 100^2$  entre les longueurs des 3 côtés prouve que le triangle à construire est un triangle rectangle, d'après le théorème de Pythagore. On va seulement tracer les 3 carrés, d'abord le carré orange, puis le rouge et enfin le vert. Ces carrés sont parcourus dans le sens des flèches indiqué sur ce dessin GeoGebra :



Le sommet de l'angle droit ne peut pas, semble-t-il être obtenu sans calculer l'angle  $\theta$ , ce qui est l'objet des commandes<sup>3</sup>

```
from math import degrees, atan
theta = degrees(atan(3/4))
```

Voici le script utilisé :

```
import turtle as tu
from math import degrees, atan
tu.reset()
tu.up()
tu.goto(100,0)
theta = degrees(atan(3/4))
print(theta)
tu.rt(180-theta)
tu.down()
# Carré orange
tu.color('black','orange')# Couleur des côtés, couleur du remplissage.
tu.begin_fill()
```

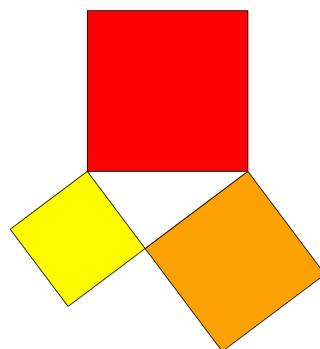
3. `degrees()` convertit les radians en degrés, `atan()`, fonction du module `math`, désigne la fonction `arctg`. Ici, on a besoin de l'arc (l'angle) dont la tangente est  $\frac{3}{4}$ , entre 0 et 90°. Le calcul montre que c'est  $\theta = 36.86989764584402$  degrés.

```

tu.fd(160)
tu.lt(90)
tu.fd(160)
tu.lt(90)
tu.fd(160)
tu.lt(90)
tu.fd(160)
tu.end_fill()
tu.rt(theta)
# Carré rouge
tu.color('black','red')
tu.begin_fill()
tu.fd(200)
tu.lt(90)
tu.fd(200)
tu.lt(90)
tu.fd(200)
tu.lt(90)
tu.fd(200)
tu.end_fill()
tu.lt(theta)
tu.backward(160)
tu.lt(90)
# Carré jaune
tu.color('black','yellow')
tu.begin_fill()
tu.fd(120)
tu.lt(90)
tu.fd(120)
tu.lt(90)
tu.fd(120)
tu.lt(90)
tu.fd(120)
tu.end_fill()
tu.hideturtle()

```

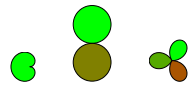
Voilà ce que l'on obtient :



#### Remarques :

- La relation  $60^2 + 80^2 = 100^2$  signifie que l'aire du carré rouge est la somme des aires des deux autres carrés.
- Avec **GeoGebra**, on peut faire beaucoup mieux. On peut, par exemple, tracer un triangle dont les côtés sont de longueur 3, 4 et 5, à la règle et au compas. On peut ensuite tracer les 3 carrés avec l'outil de traçage

des polygones réguliers. Enfin, on peut demander la valeur de l'angle qui paraît droit ainsi que les aires de ces carrés et constater la relation qui les lie. On constate ainsi expérimentalement que si les longueurs des côtés d'un triangle vérifient  $a^2 + b^2 = c^2$ , ce triangle est rectangle.



## 6.3 [\*] La tortue et les représentations graphiques.

Tracer l'arc de sinusoïde entre 0 et  $\pi$  avec la tortue.

**Mots-clefs** : tortue, sinusoïde, ligne polygonale.

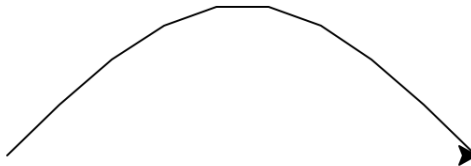
### Solution

Puisqu'un graphique est toujours une ligne polygonale, il est naturel d'essayer de tracer des graphiques à l'aide de la tortue.

Exemple : Pour tracer l'arc de sinusoïde entre 0 et  $\pi$ , utilisons le script suivant :

```
from math import pi, sin
import turtle as tu
tu.reset()
l=80
def sinustortue(n):# ligne polygonale à n côtés approchant la sinusoïde.
    tu.goto(0,0)
    #down()
    for i in range(1,n+1):
        x=i*pi/n
        tu.goto(x*l, sin(x)*l)
    tu.up()
```

Essais : En tapant `sinustortue(9)` dans la console (après avoir importé la fonction `sinustortue`), on obtient :

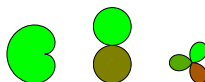


En tapant `sinustortue(1000)` dans la console :



On constate que ce tracé est très lent<sup>4</sup> et que l'image est clairement pixellisée. La première image l'est aussi, mais ça ne se voit pas. De plus, il manque beaucoup de choses à ces tracés (les axes, *etc*).

Conclusion : représenter graphiquement des fonctions à l'aide de la tortue est une mauvaise idée, sauf cas particulier.



4. à cause de la boucle "pour"

## 6.4 [\*\*\*] Algorithme de Bresenham

- 1 - Pour simplifier, disons qu'un écran d'ordinateur est un quadrillage de pixels, chacun d'eux pouvant être allumé ou éteint. Comment représenter un segment sur un tel écran ?  
2 - Application : tracer un triangle ainsi que ses médianes : sont-elles concourantes ?

**Mots-clefs** : tortue, fonction partie entière (mathématiques).

A priori, ce problème ne peut avoir de solution exacte : on ne peut pas dessiner un segment de droite avec des carrés !

L'algorithme de tracé de segment de Bresenham<sup>5</sup> résout le problème de la représentation pixellisée d'un segment sur un écran considéré comme un quadrillage constitués de carrés appelés pixels. Plus précisément, dans le plan rapporté à un repère orthonormé<sup>6</sup> est dessiné un quadrillage formé par les droites d'équation  $x = ka$ ,  $k \in \mathbb{Z}$  et  $y = la$ ,  $l \in \mathbb{Z}$ . Les carrés de ce quadrillage sont les pixels, de côté  $a$ .

Chaque pixel n'a que deux états possibles : il peut être allumé<sup>7</sup> ou éteint. La représentation pixellisée d'un segment donné  $[AB]$  sera donc nécessairement une réunion de pixels. Le but du jeu est qu'elle ressemble le plus possible au segment  $[AB]$ . On pourrait décréter que la meilleure représentation pixellisée est celle qui réunit tous les pixels qui contiennent au moins un point de  $[AB]$ . Cette idée n'est pas viable car elle ferait intervenir une infinité de pixels si  $[AB]$  est un vrai segment ( $A$  et  $B$  distincts). L'idée est donc de sélectionner quelques points de  $[AB]$  - notés dorénavant  $A = A_0, A_1, \dots, A_n = B$  - et de réunir les pixels qui les contiennent. Si ces points sont bien choisis, nous obtiendrons une représentation acceptable de  $[AB]$ . Le problème est donc de définir un critère de choix de  $A_0, A_1, \dots, A_n$ <sup>8</sup>.

### Solution

1 -  $C(x,y)$  désignant, pour tout point  $(x,y)$ , le pixel qui contient ce point, le sommet de  $C(x,y)$  qui se trouve en bas à gauche et qui, avec  $a$ , caractérise ce pixel, sera le point  $(k \cdot a, l \cdot a)$  tel que

$$k \cdot a \leq x < (k+1)a \quad \text{et} \quad l \cdot a \leq y < (l+1)a$$

autrement dit le point  $(\left[\frac{x}{a}\right] \cdot a, \left[\frac{y}{a}\right] \cdot a)$ <sup>9</sup>.

La longueur du segment  $[AB]$  est  $d = \sqrt{(x_B - x_A)^2 + (y_B - y_A)^2}$ . Si les points  $A_0, A_1, \dots, A_n$  sont choisis équidistants, la distance de deux points consécutifs sera  $\frac{d}{n}$ . Il est facile de voir que la condition  $\frac{d}{n} < a$  ou  $n > \frac{d}{a}$  implique que les pixels<sup>10</sup> associés à deux points consécutifs se touchent<sup>11</sup>, si bien que sous cette condition, la représentation de  $[AB]$  n'aura pas de trou. Enfin, il vaut mieux que  $n$  soit le plus petit possible, ce qui conduit à un critère de choix des points  $A_0, A_1, \dots, A_n$  qui paraît satisfaisant :

$$n - 1 \leq \frac{d}{a} < n \quad \text{ou} \quad n = \left\lceil \frac{d}{a} \right\rceil$$

et conduit à l'algorithme suivant :

```
import turtle as tu
from math import sqrt, floor
```

5. J. Bresenham, 1962, cf. [25]

6. Les coordonnées des points utilisés ci-dessous se rapportent à ce repère.

7. Ci-dessous, seuls les pixels allumés seront dessinés.

8. Sauf erreur, ce n'est pas la voie suivie par Bresenham, cf. [22].

9.  $x \rightarrow \left[ x \right]$  désigne la fonction partie entière.

10. qui sont des carrés fermés

11. Ils sont confondus ou ont un côté commun ou ont au moins un sommet commun

```

tu.reset()
tu.speed('fastest')
tu.hideturtle()

# Tracé du pixel contenant(x,y)
def carré(x,y,a):
    x=floor(x/a)*a
    y=floor(y/a)*a
    tu.up();tu.goto(x,y);tu.down()
    for i in range(2):
        tu.fd(a);tu.lt(90);tu.fd(a);tu.lt(90)

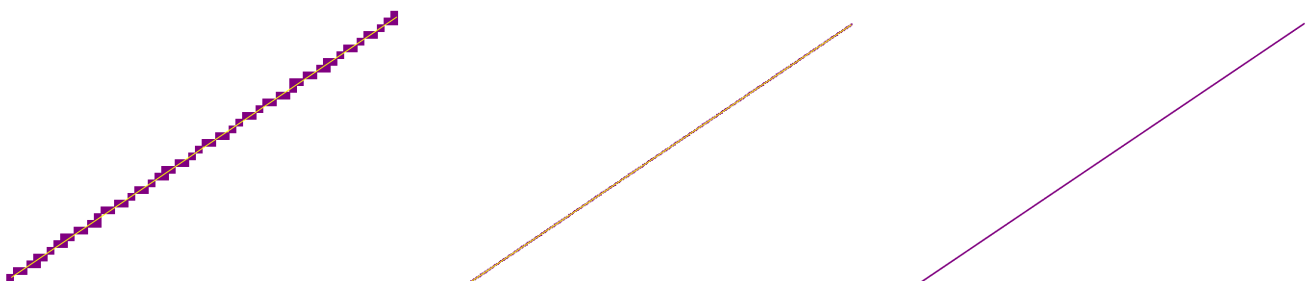
# Tracé des représentations pixellisées des segments en violet
def segment(xA,yA,xB,yB,a):
    d=sqrt((xB-xA)**2+(yB-yA)**2)
    n=floor(d/a)+1
    tu.color('purple')
    for i in range(n+1):
        tu.begin_fill()
        carré(xA+(xB-xA)*i/n,yA+(yB-yA)*i/n,a)
    tu.end_fill()

# 'Vrai' tracé du segment en jaune
def vraisegment(xA,yA,xB,yB):
    tu.color('yellow')
    tu.up();tu.goto(xA,yA);tu.down()
    tu.goto(xB,yB);tu.up();tu.home()

# Exemple 1 :
xA,yA,xB,yB,a=-100,-150,300,120,7
segment(xA,yA,xB,yB,a)
vraisegment(xA,yA,xB,yB)
# Exemple 2 :
# xA,yA,xB,yB,a=-100,-150,300,120,1
# segment(xA,yA,xB,yB,a)
# vraisegment(xA,yA,xB,yB)

```

On remarquera que l'algorithme proposé est très court. Il retourne les images suivantes, lorsque **a** prend successivement les valeurs 7, 1, 0.215<sup>12</sup> Cela illustre l'intérêt de fabriquer des écrans dont les pixels sont petits<sup>13</sup>.



**2 - Application :** Nous utilisons le script suivant qui n'apporte rien de neuf.

12. L'instruction `vraisegment(xA,yA,xB,yB)` a été supprimée lors du dernier dessin.

13. Voir l'article [30] sur les pixels.

```

import turtle as tu
from math import sqrt, floor
tu.reset()
tu.speed('fastest')
tu.hideturtle()

# Tracé des pixels
def carré(x,y,a):
    x=floor(x/a)*a
    y=floor(y/a)*a
    tu.up();tu.goto(x,y);tu.down()
    for i in range(2):
        tu.fd(a);tu.lt(90);tu.fd(a);tu.lt(90)

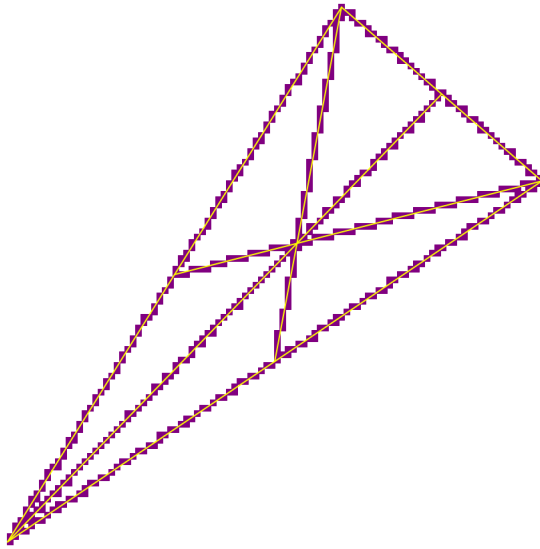
# Tracé de la représentation pixellisée du segment, en violet
def segment(xA,yA,xB,yB,a):
    d=sqrt((xB-xA)**2+(yB-yA)**2)
    n=floor(d/a)+1
    tu.color('purple')
    for i in range(n+1):
        tu.begin_fill()
        carré(xA+(xB-xA)*i/n,yA+(yB-yA)*i/n,a)
    tu.end_fill()

# Les médianes d'un triangle sont-elles concourantes ?
# Choisissons :
xA,yA,xB,yB,xC,yC,a=-100,-150,300,120,150,250,4
# Tracé de la représentation pixellisée du triangle ABC, en violet
segment(xA,yA,xB,yB,a)
segment(xA,yA,xC,yC,a)
segment(xB,yB,xC,yC,a)
# Calcul des milieux des côtés.
xM,yM,xN,yN,xP,yP=(xB+xC)/2,(yB+yC)/2,(xA+xC)/2,(yA+yC)/2,(xA+xB)/2,(yA+yB)/2
# Tracé des représentations pixellisées des médianes, en violet
segment(xA,yA,xM,yM,a)
segment(xB,yB,xN,yN,a)
segment(xC,yC,xP,yP,a)

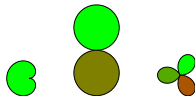
# 'Vrai' tracé d'un segment, en jaune
def vraiegment(xA,yA,xB,yB):
    tu.color('yellow')
    tu.up();tu.goto(xA,yA);tu.down()
    tu.goto(xB,yB);tu.up();tu.home()
# 'Vrai' tracé du triangle ABC, en jaune
vraiegment(xA,yA,xB,yB)
vraiegment(xA,yA,xC,yC)
vraiegment(xB,yB,xC,yC)
# 'Vrais' tracés des médianes, en jaune
vraiegment(xA,yA,xM,yM)
vraiegment(xP,yP,xC,yC)
vraiegment(xB,yB,xN,yN)

```

qui produit l'image que voici :



La représentation pixellisée du triangle et de ses médianes (partie violette du dessin) ne permet pas de conclure que les médianes d'un triangle sont concourantes. Le triangle lui-même et ses médianes ont été tracés en jaune. On constate que les médianes du triangle jaune sont concourantes mais c'est un leurre. En fait, si nos yeux étaient suffisamment bons, on verrait que la partie jaune est elle aussi pixellisée et on ne pourrait pas conclure.





## 6.5 [\*\*\*] Construction itérative des triangles de Sierpinski

Les triangles de Sierpinski sont en fait des carrelages d'un triangle équilatéral de base, par exemple le triangle  $S_0$  de sommets  $(0, 0)$ ,  $(a, 0)$  et  $(\frac{a}{2}, \frac{a\sqrt{3}}{2})$  où  $a$  est un certain nombre de pixels, par exemple  $a = 200$ .

Le premier triangle de Sierpinski  $S_1$  s'obtient en coloriant <sup>a</sup> le triangle équilatéral dont les sommets sont les milieux des côtés de  $S_0$ . C'est donc un carrelage de  $S_0$  formé de 4 triangles équilatéraux de côtés de longueur  $\frac{a}{2}$ , 3 triangles étant blancs, le triangle central étant coloré.

Et ainsi de suite : ayant tracé  $S_i$ , on obtient  $S_{i+1}$  en remplaçant chaque triangle blanc par 4 triangles, en utilisant comme précédemment les milieux de ses côtés.

Le problème est de programmer les déplacements de la tortue de sorte qu'elle dessine  $S_n$ .

---

<sup>a</sup>. en violet ci-dessous

**Mots-clefs :** Listes, algorithme itératif (non récursif), tortue.

### Solution

La solution qui suit repose sur le fait que l'on peut se contenter de colorier les triangles équilatéraux qui doivent être coloriés. Comme ils sont tous posés sur leur pointe, il suffit, pour identifier un tel triangle, de déterminer les coordonnées de la pointe et de connaître la longueur de ses côtés.

- Pour tracer  $S_1$ , on colorie le triangle de côté  $\frac{a}{2}$ , de pointe  $(\frac{a}{2}, 0)$ .
- Pour tracer  $S_2$ , on doit colorier 3 triangles supplémentaires de côté  $\frac{a}{2^2}$  dont les sommets sont faciles à déterminer. En effet, plus généralement, si on vient de dessiner  $S_i$  et si  $(x, y)$  est la pointe d'un triangle coloré de ce carrelage, de côté  $\frac{a}{2^i}$ , il faudra colorer dans chacun des 3 triangles blancs qui l'entourent un triangle coloré de côté  $\frac{a}{2^{i+1}}$ , leurs pointes étant successivement  $(x - \frac{a}{2^{i+1}}, y)$ ,  $(x + \frac{a}{2^{i+1}}, y)$  et  $(x, y + \frac{a\sqrt{3}}{2^{i+1}})$ . Ce qui donne l'algorithme suivant de tracé de  $S_n$  à l'aide de la tortue :

"""

*La fonction `—> sierpinski(n)` retourne le nième triangle de Sierpinski tracé par la tortue sur un triangle équilatéral de côté `a` pixels précisé ci-dessous. Cette fonction est itérative.*

"""

```
import turtle as tu
from math import sqrt
tu.reset()
tu.speed('fastest')
tu.color('purple')
tu.hideturtle()
```

```
a=200 #eval(input('a='))
```

```
for i in range(3):
    tu.forward(a)
    tu.left(120)
# Le triangle équilatéral de base a été tracé. La tortue est
# dans son état initial.
```

```
def tripinski(x,y,b):
    tu.up();tu.goto(x,y);tu.down()
    tu.begin_fill()
```

```

    tu.lt(60);tu.fd(b);tu.rt(60)
    tu.bk(b);tu.rt(60)
    tu.fd(b);tu.lt(60)
    tu.end_fill()
    tu.up();tu.goto(0,0);tu.down()
    return
# Cette fonction trace un triangle équilatéral de côté b
# reposant sur sa pointe en (x,y).
# Elle laisse la tortue dans son état initial.
h=a*sqrt(3)/2

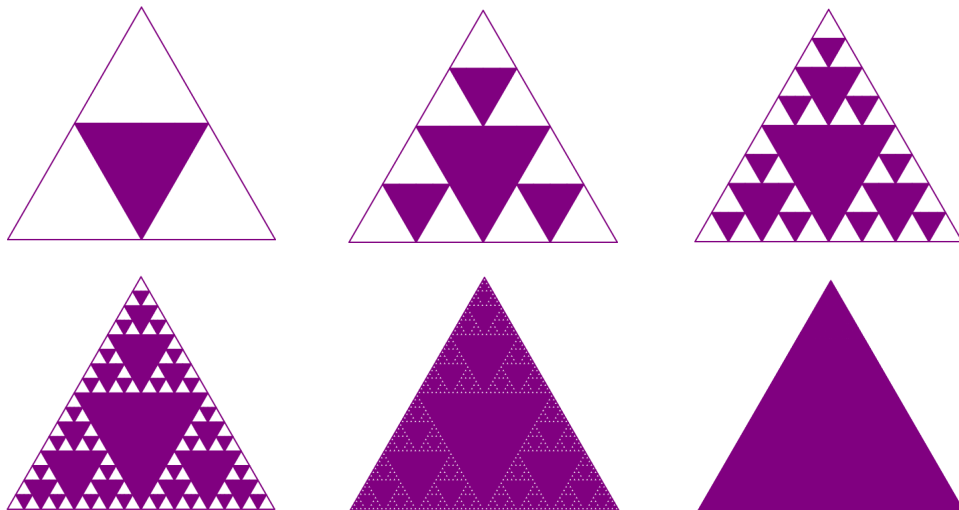
def sierpinski(n):
    L=[[a/2,0]]
    l=len(L)
    tripinski(L[0][0],L[0][1],a/2**1)
    # S1 a été tracé.
    for i in range(2,n+1):
        Li=[]
        for j in range(l):
            Li=Li+[[L[j][0]-a/2**i,L[j][1]],
                    [L[j][0]+a/2**i,L[j][1]],
                    [L[j][0],L[j][1]+h/2**(i-1)]]
        L=Li
        l=len(L)
        for j in range(l):
            tripinski(L[j][0],L[j][1],a/2**i)

    return
    # Sn a été tracé.

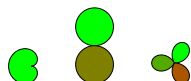
sierpinski(6)
# pour tracer le 6ème triangle de Sierpinski.

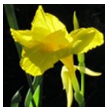
```

Voici les triangles de Sierpinski  $S_1$ ,  $S_2$ ,  $S_3$ ,  $S_4$ ,  $S_6$  et  $S_{10}$  :



Bien sûr,  $S_{10}$  contient  $3^{10}$  triangles blancs mais ils ne se voient pas.





# Chapitre 7

## Algorithmes récursifs

---

Liste des exercices :

**Énoncé n° 7.1** [\*] : Calculer récursivement le maximum d'une liste de nombres

---

L'approche récursive est un des concepts de base en informatique, cf. [26].

Exemple d'algorithme récursif : Le calcul de `factorielle(n)` se fait de manière spontanée sous la forme récursive, suggérée par la formule

$$\forall n \in \mathbb{N}^*, \quad n! = n \cdot (n-1)!$$

avec Python (ou scilab) ou tout logiciel qui englobe la récursivité<sup>1</sup>. C'est un exemple typique : on remarque que le calcul de `factorielle(n)` utilise `factorielle(n-1)` :

```
def factorielle(n):# n est un entier >=0.
    if n == 0:
        a = 1
    else :
        a = n*factorielle(n-1)
    return a
```

En exécutant cet algorithme récursif pour `n=3`, par exemple, Python devra calculer (phase de propagation) `3*factorielle(2)` Pour cela, il devra calculer `2*factorielle(1)`, puis `1*factorielle(0)`. Python utilisera alors le cas de base `factorielle(0) = 1`.

En remontant les calculs, il calculera ensuite successivement `factorielle(1)`, puis `factorielle(2)` et enfin `factorielle(3)`.

Exemples d'exécution de la fonction `factorielle()` :

```
runfile('Chemin_de_la_fonction_factorielle()')
factorielle(0)
Out[2] : 1
factorielle(10)
Out[4] : 3628800
factorielle(100)
Out[6] : 9332621544394415268169923885626670049071596826438162146859296389521759
999322991560894146397615651828625369792082722375825118521091686400000000000000
0000000000
```

Les calculs sont presque instantanés. Vérifions cette impression à l'aide d'un script :

---

1. Les premiers langages de programmation qui ont autorisé l'emploi de la récursivité sont LISP et Algol 60. Depuis, tous les langages de programmation généraux réalisent une implémentation de la récursivité (cf. [26]).

```

from time import process_time
from factorielle import factorielle
# Le fichier contenant factorielle() s'appelle factorielle.py.
a = process_time()
n = eval(input('n=='))
factorielle(n)
b = process_time()
print(b-a)

```

retourne la durée du calcul de  $n!$  <sup>2</sup>. Pour  $n = 2000$ , on obtient

```

runfile('Chemin_du_script')
n = 2000
0.016174000000000355

```

(en secondes). Par contre, le calcul de 3000 ! se révèle impossible :

```

n = 3000
RecursionError : maximum recursion depth exceeded in comparison

```

autrement dit, dans ce cas, le nombre de pas de propagation est trop grand.

Complément : Comparons les performances de l'algorithme récursif et de la simple boucle pour suivante <sup>3</sup> :

```

from time import process_time
a = process_time()
n = eval(input('n=='))
x = 1
for i in range(1,n+1):
    x = i*x
b = process_time()
print(b-a)

```

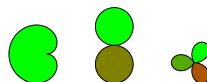
en calculant 2000 !, puis 3000 ! :

```

runfile('Chemin_du_script')
n = 2000
0.007579999999999992
runfile('Chemin_du_script')
n = 3000
0.014097999999999722

```

Le principal enseignement de cette comparaison est que l'on peut calculer des factorielles plus grandes avec la boucle parce que l'on n'est pas gêné par la limitation du nombre de pas de propagation.



2. Remarque : la fonction `clock()` du module `time` est devenue obsolète et doit être remplacée par la fonction `process_time()` ou la fonction `perf_counter()` du même module.

3. qu'on appelle algorithme itératif, par opposition à algorithme récursif

## 7.1 [\*] Calculer récursivement le maximum d'une famille finie de nombres

Calculer récursivement le maximum d'un ensemble fini  $E$  de nombres, en remarquant que le maximum de  $E$  est le maximum d'un terme de  $E$  et du maximum des autres éléments.

**Mots-clefs :** algorithme récursif, matrice-ligne, liste, commandes `len(L)` et `M.size`, instruction conditionnelle, extraction d'éléments d'une matrice-ligne ou d'une liste, fonction `normal()` du module `numpy.random`, convertisseur de type `list()` <sup>4</sup>.

### Solutions

Il suffit de mettre en forme l'énoncé. On distingue ci-dessous le cas où ces nombres sont donnés sous la forme d'une matrice-ligne  $M$  du cas où ils sont donnés sous la forme d'une liste  $L$ .

#### 1.a - Supposons d'abord que $E$ est une matrice-ligne notée $M$ . <sup>5</sup>

La fonction `maks()` définie ci-dessous convient. Elle sera testée plus loin.

```
def maks(M) :
    if M.size == 1 :
        a = M[0]
    else :
        a = max(M[0] , maks(M[1 :]))
        M = M[1 :]
    return a
```

#### 1.b - Supposons maintenant que $E$ est une liste notée $L$ .

Il suffit de remplacer dans le script précédent la commande `M.size` par la commande `len(L)`. Cette nouvelle fonction s'appelle `maksliste`.

```
def maksliste(L) :
    if len(L) == 1 :
        a = L[0]
    else :
        a = max(L[0] , maksliste(L[1 :]))
        L = L[1 :]
    return a
```

### 2 - Essais

#### 2.a - Supposons d'abord que $E$ est une matrice-ligne notée $M$ . Nous utiliserons le script suivant :

```
from time import process_time# clock() a été déclarée obsolète.
from maks import maks
# parce que le module contenant la fonction maks() a été appelé maks.py.
from numpy.random import normal

a = process_time()
n = eval(input('n= '))
M = normal(5,10,n)
# normal(5,10,n) produit une matrice-ligne de n éléments qui sont des nombres
# tirés au hasard suivant la loi normale de moyenne 5 et d'écart-type 10.
```

4. Voir le préambule du chapitre 3

5. matrice à une dimension

```
x = maks(M)
b = process_time()
print(x, b-a)
```

<sup>6</sup> et donnons à **n** successivement les valeurs 100, 1000 et 3000. Cela donne :

```
runfile('Chemin_du_script')
n = 100
24.601408302609787 0.005687999999999693
runfile('Chemin_du_script')
n = 1000
38.478588229729006 0.012920000000000265
runfile('Chemin_du_script')
n = 3000
RecursionError: maximum recursion depth exceeded in comparison
```

**2.b** - Supposons maintenant que **E** est une liste notée **L**. Modifions le script précédent en remplaçant **M** par **L** et **maks()** par **maksliste()** et exécutons-le pour **n** =100, 1000, 3000. Cela donne :

```
runfile('Chemin_du_script')
n = 100
33.03803592629363 0.01584000000000003
runfile('Chemin_du_script')
n = 1000
41.860708455607885 0.022264000000000284
runfile('Chemin_du_script')
n = 3000
RecursionError: maximum recursion depth exceeded while calling a Python object
```

**3** - Pourquoi les algorithmes récursifs ci-dessus sont à éviter ?

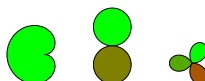
Parce qu'ils ne fonctionnent qu'avec de petits ensemble de nombres (on a vu que 3000 nombres, c'est déjà trop). Des algorithmes itératifs n'ont pas cet inconvénient. Exemple :

```
from time import process_time
import numpy as np
a=process_time()
M = np.random.normal(5,10,100000)
max = M[0]
for i in range(1,100000):
    if M[i] > max:
        max = M[i]
b=process_time()
print(max, b-a)
```

ce qui donne :

```
51.19422120633325 0.48645799999999983
```

Moins d'une demi-seconde pour calculer le maximum d'un million de nombres !




---

6. On affiche le maximum de **M** et la durée du calcul.

## Chapitre 8

# Arithmétique, compter en nombres entiers

Le calcul en nombres entiers concerne notamment l'arithmétique et les problèmes de dénombrement, donc le calcul des probabilités<sup>1, 2</sup>. Les calculs en nombres entiers donnent souvent lieu à des dépassements de capacité de la mémoire.

---

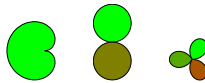
Liste des exercices :

Énoncé n° 8.1 [\*\*\*] : PGCD de  $p$  entiers positifs

Énoncé n° 8.2 [\*\*\*\*] : Crible d'Ératosthène

Énoncé n° 8.3 [\*\*\*] : Factoriser un entier en produit de nombres premiers.

---



---

1. dans le cas où il n'y a qu'un nombre fini d'issues équiprobables

2. Le module du Calcul des probabilités est le module `numpy.random`, voir [2], p.163.



## 8.1 **[\*\*\*]** PGCD de p entiers positifs.

1 - Programmer une fonction qui, étant donnés p entiers positifs  $a_1, \dots, a_p$  dont le minimum est noté  $m$ , retourne la matrice à une dimension de leurs diviseurs communs en procédant de la manière suivante : retirer de la suite  $1, \dots, m$  les nombres qui ne sont pas des diviseurs de  $a_1$ , puis ceux qui ne sont pas des diviseurs de  $a_2$ , etc.

2 - En déduire leur PGCD.

**Mots-clefs** : module `numpy`, fonctions `arange()`, `size()`, `sort()`, masque de matrice, `where()`, reste de la division euclidienne, extraction des éléments d'une matrice (`m,` ) dont les indices sont donnés, fonction pré-définie `min()`, module `math`, fonction `gcd` de ce module.

### Solution

Notons  $M$  la matrice à une dimension<sup>3</sup> dont les éléments sont  $a_1, \dots, a_p$ .

1 - Le script de la fonction `miris()` définie ci-dessous reproduit exactement l'énoncé. Les nombres qui restent, formant le dernier état de la matrice à une dimension  $L$ , sont les diviseurs communs de  $a_1, \dots, a_p$ .

```
# La fonction miris suivante a pour données une matrice à une
# dimension M de p nombres entiers >0. Elle retourne la matrice
# à une dimension de leurs diviseurs communs.
import numpy as np
def miris(M):
    M = np.sort(M)# M est rangé dans l'ordre croissant, ce qui ne modifie pas
    # le problème. La matrice M n'a pas besoin d'être ordonnée.
    m = min(M)
    p = np.size(M)# n est le nombre d'éléments de M.
    L = np.arange(1,m+1)# Les diviseurs communs sont à rechercher
    # dans L, une matrice à une dimension.
    for i in range(p-1,-1,-1):
        Mat = M[i]%L# Mat est la matrice à une dimension des restes dans
        # les divisions euclidiennes de M[i] par les éléments de L.
        masque = (Mat == 0)# Repérage de ces diviseurs de M[i].
        indices = np.where(masque)# Indices de ces diviseurs.
        L = L[indices]# On ne conserve que ces diviseurs.
    return L
```

Exemple n°1 :

```
from miris import *
M = np.array([114,747,1266])
miris(M)
Out[3]: array([1, 3])
type(miris(M))
Out[4]:
numpy.ndarray
```

114, 747 et 1266 ont donc 1 et 3 comme diviseurs communs. Le maximum de  $L$  est évidemment le PGCD des p entiers positifs considérés.

Exemple n°2 :

---

3. voir 4

```
M=np.array([2171,318215,41023,16179])
max(miris(M))
Out[5] : 1
```

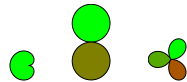
2171, 318215, 41023 et 16179 sont donc premiers dans leur ensemble.

**2** - Le PGCD est le plus grand diviseur commun, autrement dit **max**(L).

Exemple n°3 : Calculons le PGCD de 123 456 789 et 987 654 321 d'abord en utilisant la fonction **miris**() puis la fonction **gcd**() <sup>4</sup>, <sup>5</sup> :

```
from miris import *
M = array([123456789,987654321])
max(miris(M))
Out[4] : 9
from math import *
gcd(123456789,987654321)
Out[6] : 9
```

Pendant l'exécution de ce script, on peut constater que la fonction **gcd**() est beaucoup plus rapide que **max(miris( ))**.



---

4. **gcd**(), fonction du module **math**, retourne le PGCD de deux entiers positifs donnés.

5. **gcd**() repose, sauf erreur, sur l'algorithme d'Euclide, cf.[2.13](#)

## 8.2 [\*\*\*\*] Crible d'Ératosthène

- 1 - Programmer une fonction qui, pour tout entier  $n \geq 2$  donné, retourne les nombres premiers compris entre 2 et  $n$ , rangés dans l'ordre croissant.
- 2 - Programmer une fonction qui, étant donné un nombre entier  $n \geq 2$ , retourne suivant le cas "n est premier" ou "n n'est pas premier".

**Mots-clefs** : effacer des éléments d'une matrice à une dimension à l'aide d'un masque.

**1 - Rappels** :<sup>6</sup> Tout entier  $n \geq 2$  est appelé nombre premier s'il n'est divisible que par 1 et par lui-même<sup>7</sup>.

**Algorithme** : Pour trouver les nombres premiers  $\leq n$ , partons de la suite  $S = 2, 3, \dots, n$ .

- Conservons le minimum de  $S$ , soit 2 et supprimons ceux de ses multiples stricts 4, 6, etc qui se trouvent dans  $S$ .
- Re commençons cette manipulation avec le nombre suivant dans  $S$  jusqu'à ce que l'on ne supprime plus rien. On ne retire plus rien de  $S$  quand le premier multiple de  $p$  que l'on pourrait supprimer, c'est à dire  $p^2$ <sup>8</sup>, n'est pas dans  $S$ , ce qui équivaut à  $p^2 > n$ .
- Les nombres restants dans  $S$  sont les nombres premiers recherchés parce qu'ils ne sont pas des multiples de nombres plus petits. Ils sont rangés dans l'ordre croissant.

```
# erato(n) retourne les nombres premiers <= n sous forme de matrice (n, )
# ainsi, optionnellement, que la durée b - a du calcul et
# le nombre de nombres premiers trouvés S.size.
import numpy as np
from time import clock
def erato(n):# n est un entier >=2.
    #a = clock()
    S = np.arange(2,n+1)# S est une matrice (n, ).
    p,i = 2,0# 2 est le premier nombre premier, i est un compteur.
    while p**2 <= n:# La boucle commence à 2, premier nombre premier.
        T = S%p# Les restes des multiples de p dans la division par p
        # sont nuls.
        T[i] = p# En remplaçant 0 par p, on va récupérer p ensuite.
        masque = T > 0# Pour effacer les 0 (dans S, les multiples de p > p).
        S = S[np.where(masque)]# Effacement.
        i = i + 1
        p = S[i]# Nombre premier suivant.
    #b = clock()
    return S#, S.size#, b-a
```

**Application à quelques valeurs de n** : Ci-dessous, on a demandé à `erato(n)` de retourner les nombres premiers inférieurs ou égaux à  $n$  ainsi que la durée des calculs :

```
runfile('Chemin_de_la_fonction_erato()')
erato(2)
Out[2]: (array([2]), 1.4999999999432134e-05)
erato(3)
Out[3]: (array([2, 3]), 2.800000000036107e-05)
```

6. Le crible d'Ératosthène est supposé connu.

7. Par exemple, 2 est un nombre premier. On considère parfois que 1 est aussi un nombre premier.

8. qui ne peut pas avoir été effacé précédemment

```

erato(5)
Out[4] : (array([2, 3, 5]), 5.8000000000113516e-05)
erato(10)
Out[5] : (array([2, 3, 5, 7]), 7.599999999996498e-05)
erato(100)
Out[6] :
(array([ 2,  3,  5,  7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47, 53, 59,
        61, 67, 71, 73, 79, 83, 89, 97]), 0.00010900000000013677)
erato(1000)
Out[7] :
(array([ 2,  3,  5,  7, 11, 13, 17, 19, 23, 29, 31, 37, 41,
        43, 47, 53, 59, 61, 67, 71, 73, 79, 83, 89, 97, 101,
        103, 107, 109, 113, 127, 131, 137, 139, 149, 151, 157, 163, 167,
        173, 179, 181, 191, 193, 197, 199, 211, 223, 227, 229, 233, 239,
        241, 251, 257, 263, 269, 271, 277, 281, 283, 293, 307, 311, 313,
        317, 331, 337, 347, 349, 353, 359, 367, 373, 379, 383, 389, 397,
        401, 409, 419, 421, 431, 433, 439, 443, 449, 457, 461, 463, 467,
        479, 487, 491, 499, 503, 509, 521, 523, 541, 547, 557, 563, 569,
        571, 577, 587, 593, 599, 601, 607, 613, 617, 619, 631, 641, 643,
        647, 653, 659, 661, 673, 677, 683, 691, 701, 709, 719, 727, 733,
        739, 743, 751, 757, 761, 769, 773, 787, 797, 809, 811, 821, 823,
        827, 829, 839, 853, 857, 859, 863, 877, 881, 883, 887, 907, 911,
        919, 929, 937, 941, 947, 953, 967, 971, 977, 983, 991, 997]),
 0.0004569999999999297)
erato(10000000)
Out[8] : (array([ 2,  3,  5, ..., 9999971, 9999973, 9999991]),
7.436985)
erato(100000000)
Out[9] :
(array([ 2,  3,  5, ..., 99999959, 99999971, 99999989]),
153.03122199999999)
erato(500000000)
Out[10] :
(array([ 2,  3,  5, ..., 499999909, 499999931,
        499999993]), 2068.7021259999997)

```

Ce dernier essai dure près de 35 minutes. En demandant en plus le nombre de nombres premiers inférieurs ou égaux à 500 000 000, on découvre qu'il y en a 26 355 867.

**2** - Il suffit d'utiliser la fonction `erato()` ci-dessus : un entier `n` est premier si et seulement s'il figure dans la liste des entiers premiers inférieurs ou égaux à `n` ou si et seulement s'il est le maximum de cette liste.

```

from eratosthene import erato# Attention : on utilise erato(n)
# dans sa version qui retourne seulement la matrice à une dimension
# des nombres premiers <= n.
def japhet(n):
    return n == max(erato(n))

```

Application à quelques valeurs de `n` :

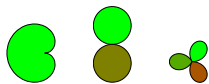
```

runfile('Chemin_de_la_fonction_japhet()')
Reloaded modules : eratosthene
japhet(5)
Out[2] : True

```

```
japhet(9999971)
Out[3] : True
japhet(123456789)
Out[4] : False
```

La fonction `japhet()` ci-dessus est ce qu'on appelle un test de primalité<sup>9</sup> Les tests de primalité basés sur le crible d'Ératosthène ne permettent pas de traiter de grands nombres. Il y a mieux, voir [\[23\]](#) par exemple.



---

9. Dire si un nombre entier donné est premier ou pas.

## 8.3 [\*\*\*] Factoriser un entier en produit de nombres premiers

Programmer une fonction qui, étant donné un entier  $n \geq 2$ , retourne sa factorisation en produit de nombres premiers <sup>a</sup>.

a. On peut utiliser la fonction `erato()` de l'exercice 8.2.

**Mots-clefs :** appel d'une fonction, boucles `pour` et `tant que`.

### Solution

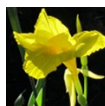
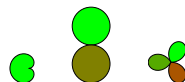
Étant donné un entier  $n \geq 2$ , `erato(n)` nous donne la matrice `(n, )` des nombres premiers  $\leq n$ . La fonction `beatris()` ci-dessous passe en revue ces nombres premiers et fait la liste de ceux qui divisent `n`. Un tel nombre premier apparaît dans la liste autant de fois que sa puissance la plus grande qui divise `n`.

```
from eratosthene import erato
def beatris(n):
    S = erato(n)
    L = []
    for i in range(S.size):
        while n%S[i] == 0:
            L.append(S[i])
            n = n//S[i]
    return L
```

Exemples :

```
runfile('Chemin_de_la_fonction_beatris')
beatris(24)
Out[6]: [2, 2, 2, 3]
beatris(79)
Out[7]: [79]
beatris(2018)
Out[8]: [2, 1009]
beatris(9999971)
Out[5]: [9999971]
beatris(123456789)
Out[3]: [3, 3, 3607, 3803]
3*3*3607*3803
Out[10]: 123456789
```

On voit donc, à la lecture de ces résultats, que  $24 = 2^3 \cdot 3$ , que 79 et 9 999 971 sont des nombres premiers et que  $123\,456\,789 = 3^2 \cdot 3607 \cdot 3803$ .



# Bibliographie

- [1] ARNAUD BODIN *Python au lycée*, Amazon Distribution (2018).  
Très bon manuel, bien écrit et accessible. Ce livre est téléchargeable à l'adresse :  
<http://www.exo7.emath.fr/cours/livre-python1.pdf>  
  
Les codes Python des exercices se trouvent à l'adresse :  
<https://github.com/exo7math/python1-exo7>
- [2] ALEXANDRE CASAMAYOU-BOUCAU, PASCAL CHAUVIN, GUILLAUME CONNAN *Programmation en Python pour les mathématiques*, 2<sup>ème</sup> édition, Dunod Éditeur (2016).  
Ce manuel est à peu près indispensable pour un professeur de mathématiques qui débute en Python.
- [3] GÉRARD SWINNEN *Apprendre à programmer avec Python 3*, Eyrolles Éditeur, 3<sup>ème</sup> édition (2016).  
Python est un langage de programmation général. Ce manuel ne vise pas particulièrement les professeurs de mathématiques.
- [4] WIKIBOOKS *Mathématiques avec Python et Ruby*.  
[https://fr.wikibooks.org/wiki/Mathématiques\\_avec\\_Python\\_et\\_Ruby](https://fr.wikibooks.org/wiki/Mathématiques_avec_Python_et_Ruby)
- [5] PATRICK FUCHS ET PIERRE POULAIN *Cours de Python*.  
<https://python.sdv.univ-paris-diderot.fr>  
  
Très bon cours de Python en français.
- [6] BERND KLEIN *Python course*.  
[https://www.python-course.eu/python3\\_course.php](https://www.python-course.eu/python3_course.php)  
  
Très bon cours de Python en anglais.

## Programmes de mathématiques au lycée

- [7] *Seconde, programme de mathématiques*  
[http://cache.media.education.gouv.fr/file/30/52/3/programme\\_mathematiques\\_seconde\\_65523.pdf](http://cache.media.education.gouv.fr/file/30/52/3/programme_mathematiques_seconde_65523.pdf)
- [8] *Seconde, mathématiques, Ressources pour la classe de Seconde*  
[http://cache.media.eduscol.education.fr/file/Programmes/17/8/Doc\\_ress\\_algo\\_v25\\_109178.pdf](http://cache.media.eduscol.education.fr/file/Programmes/17/8/Doc_ress_algo_v25_109178.pdf)

## Documentation sur des modules de Python

### Cours de Télécom ParisTech

Le cours de Télécom ParisTech, de Alexandre Gramfort & Slim Essid, est disponible aussi en version pdf. Il est intéressant :

<http://www.prepas.org/2013/Info/Liesse/Telecom/>

En détail :

- [9] *Introduction à Python*  
<http://www.prepas.org/2013/Info/Liesse/Telecom/1-Intro-Python.html>  
<http://www.prepas.org/2013/Info/Liesse/Telecom/1-Intro-Python.pdf>

- [10] *Numpy et matplotlib*  
<http://www.prepas.org/2013/Info/Liesse/Telecom/2-Numpy.html>  
<http://www.prepas.org/2013/Info/Liesse/Telecom/2-Numpy.pdf>
- [11] *Librairie d'algorithmes pour le calcul scientifique en Python*  
<http://www.prepas.org/2013/Info/Liesse/Telecom/3-Scipy.html>  
<http://www.prepas.org/2013/Info/Liesse/Telecom/3-Scipy.pdf>

#### **Autres sources de documentation sur les modules**

- [12] PYTHON : THE STANDARD PYTHON LIBRARY  
<https://docs.python.org/py3k/library/index.html>  
Liste complète des modules courants de Python. Celle liste permet d'accéder à la documentation sur les fonctions pré-définies de Python via ses modules.
- [13] *Built-in functions*  
<https://docs.python.org/3/library/functions.html>  
Documentation officielle en anglais sur les fonctions pré-définies de Python.
- [14] *Turtle graphics*  
<https://docs.python.org/fr/3/library/turtle.html>  
Documentation complète en anglais sur le module Turtle de Python.
- [15] *math - Mathematical functions*  
<https://docs.python.org/3/library/math.html>  
Documentation officielle en anglais sur le module «math».
- [16] *Numpy reference*  
<https://docs.scipy.org/doc/numpy/reference/index.html>  
Documentation officielle en anglais décrivant les fonctions, modules et objets de «Numpy».
- [17] *Numpy.random reference*  
<https://docs.scipy.org/doc/numpy/reference/routines.random.html>  
Documentation officielle sur le module «numpy.random» (Random sampling) : génération de nombres aléatoires, tirages aléatoires, différents types courants de lois de probabilité.
- [18] *random - Generate pseudo-random numbers*  
<https://docs.python.org/3/library/random.html>  
Documentation officielle en anglais sur le module «random».
- [19] *Matplotlib User's Guide*  
<http://matplotlib.org/users/>  
Documentation en anglais sur le module «Mathplotlib». On préférera [20].
- [20] NICOLAS P. ROUGIER : *Tutoriel Matplotlib*, sur le site «Developpez.com» :  
<http://python.developpez.com/tutoriels/graphique-2d/matplotlib/>  
Très bonne documentation traduite en français sur le module «Mathplotlib».
- [21] *Turtle Graphics*  
<https://docs.python.org/fr/3/library/turtle.html>  
Documentation complète officielle sur Python.

#### **Articles divers**

- [22] JEAN-MARC DUQUESNOY, PIERRE LAPÔTRE, RAYMOND MOCHÉ *Algorithme de Bresenham 1 et 2* :  
[http://gradus-ad-mathematicam.fr/Premiere\\_Algorithmique2.htm](http://gradus-ad-mathematicam.fr/Premiere_Algorithmique2.htm)
- [23] JULIEN ÉLIE *L'algorithme AKS ou Les nombres premiers sont de classe P*  
<http://www.trigofacile.com/maths/curiosite/primarite/aks/pdf/algorithme-aks.pdf>
- [24] RAYMOND MOCHÉ *Le lièvre et la tortue : jeu équitable ?*  
<https://gradus-ad-mathematicam.fr/activites/seconde/seconde-statistiques-probabilites-et-alg-le-lievre-et-la-tortue/>



- [25] WIKIPEDIA *Algorithme de tracé de segment de Bresenham*  
[https://fr.wikipedia.org/wiki/Algorithme\\_de\\_trac%C3%A9\\_de\\_segment\\_de\\_Bresenham](https://fr.wikipedia.org/wiki/Algorithme_de_trac%C3%A9_de_segment_de_Bresenham)
- [26] WIKIPEDIA *Algorithme récursif*  
[https://fr.wikipedia.org/wiki/Algorithme\\_r%C3%A9cursif](https://fr.wikipedia.org/wiki/Algorithme_r%C3%A9cursif)
- [27] WIKIPEDIA *Carré magique (mathématiques)*  
[https://fr.wikipedia.org/wiki/Carr%C3%A9\\_magique\\_\(math%C3%A9matiques\)](https://fr.wikipedia.org/wiki/Carr%C3%A9_magique_(math%C3%A9matiques))
- [28] WIKIPEDIA *Crible d'Ératosthène*  
[https://fr.wikipedia.org/wiki/Crible\\_d%27%C3%89ratosth%C3%A8ne](https://fr.wikipedia.org/wiki/Crible_d%27%C3%89ratosth%C3%A8ne)
- [29] WIKIPEDIA *Nombre premier*  
[https://fr.wikipedia.org/wiki/Nombre\\_premier#Autres\\_algorithmes](https://fr.wikipedia.org/wiki/Nombre_premier#Autres_algorithmes)
- [30] WIKIPEDIA *Pixel*  
<https://fr.wikipedia.org/wiki/Pixel>

