Informatique et algorithmique avec le logiciel Python en CPGE scientifique

BÉGYN Arnaud

9 août 2015

Table des matières

1 Cours de première année **7**

1 Structures de données en Python . . . 7

1.1 Nombres . . . 7

1.1.1 Nombres en base 10 et base 2 . . . 7

1.1.2 Représentation des nombres . . . 8

1.1.3 Opérations sur les nombres . . . 9

1.2 Les booléens et les tests . . . 10

1.3 Listes . . . 10

1.3.1 Exemples de listes . . . 10

1.3.2 Listes définies par compréhension . . . 11

1.3.3 Indices, longueur et intervalles . . . 12

1.3.4 Opérateurs sur les listes . . . 13

1.3.5 Modification d'une liste . . . 13

1.3.6 Sémantique de pointeur . . . 14

1.3.7 Autres fonctions et opérations pratiques . . . 14

1.4 Chaînes de caractères . . . 15

2 Programmation . . . 16

2.1 Variables et affectations . . . 16

2.2 Les structures conditionnelles . . . 20

2.3 Structures itératives . . . 21

2.4 Fonctions . . . 22

2.5 Terminaison et correction d'un algorithme . . . 23

2.5.1 Terminaison . . . 23

2.5.2 Correction . . . 24

3 Modules . . . 25

3.1 Généralités . . . 25

3.2 Math . . . 26

3.3 Numpy . . . 26

3.3.1 Manipuler des listes et des matrices . . . 26

3.3.2 Quelques outils d'algèbre linéaire . . . 32

3.3.3 Tableaux aléatoires . . . 35

3.3.4 Polynômes . . . 36

3.4 Random . . . 38

3.5 Matplotlib (tracé de courbes) . . . 38

3.6 Scipy (calcul scientifique) . . . 42

3.7 Autres modules (time, MySQLdb, sqlite3,...) . . . 45

4 Algorithmes de première année . . . 46

4.1 Recherche d'un élément dans un tableau/liste. . . . 46

4.2 Recherche naïve d'un mot dans une chaîne de caractères . . . 47

4.3 Recherche du maximum dans un tableau/liste de nombres. 47
4.4 Calcul de la moyenne, de la variance des valeurs d'un tableau/liste de nombres 48
4.5 Recherche par dichotomie dans un tableau/liste trié 48
4.6 Recherche par dichotomie d'un zéro d'une fonction continue 49
4.7 Méthode de Newton 50
4.8 Calcul de valeur approchées d'intégrales sur un segment par la méthode des rectangles ou des trapèzes 51
4.9 Résolution d'une équation différentielle ordinaire : méthode ou schéma d'Euler 53
4.10 Résolution d'un système linéaire inversible : méthode du pivot de Gauss 57
5 Lire et écrire dans un fichier 61
6 Bases de données 63
6.1 Généralités 63
6.2 Requêtes simples 64
6.3 Constructions ensemblistes 68
6.4 Requêtes composées 70

2 Cours de deuxième année 73
1 Piles 73
1.1 Généralités 73
1.2 Primitives 74
1.3 Files 75
2 Récursivité 76
2.1 Généralités 76
2.2 Exemples plus évolués 77
2.3 Pile d'exécution d'une fonction récursive 78
2.4 Terminaison et correction d'une fonction récursive 79
2.5 Complexité d'un algorithme récursif 81
2.6 Version récursive des algorithmes vus en première année 83
2.6.1 Recherche d'un élément dans une liste triée 83
2.6.2 Algorithme de dichotomie 84
3 Algorithmes de tri 84
3.1 Généralités sur les algorithmes de tri par comparaisons 84
3.2 Tri par insertion (insertion sort) 85
3.2.1 Présentation de l'algorithme 85
3.2.2 Complexité 86
3.3 Tri rapide (quick sort) 88
3.3.1 Présentation de l'algorithme 88
3.3.2 Complexité 91
3.4 Tri fusion (merge sort) 93
3.4.1 Présentation de l'algorithme 93
3.4.2 Complexité 96
3.5 Comparaison empirique des trois algorithmes de tri 98
3.6 Recherche de la médiane dans un tableau 99

3 Exercices de première année 101
1 Structure de données 101
1.1 Représentation des nombres 101
1.2 Listes 102

1.3 Chaînes de caractères 102
2 Modules 103
3 Algorithmique 103
3.1 Algorithmique des tableaux/listes et des chaînes de caractères 103
3.2 Méthode d'Euler 104
3.3 Algorithme de Gauss-Jordan 106
4 Lire et écrire dans un fichier 107
4.1 Requêtes SQL 107

4 Exercices de seconde année 113
1 Piles 113
2 Récursivité 115
3 Tris 119
4 Autres 120
4.1 Stéganographie 120
4.2 Algorithme de Dijkstra 124
4.3 Équation de la chaleur 128

Chapitre 1

Cours de première année

1 Structures de données en Python

1.1 Nombres

1.1.1 Nombres en base 10 et base 2

En informatique, l'information est réprésentée par des séquences de 0 et de 1. Les symboles 0 et 1 sont appelés *booléens*, *chiffre binaires*, ou encore *bit* (acronyme de *binary digits*).

Les nombres que nous manipulons sont habituellement représentés en base 10 : intuivement, pour dénombrer des objets, on les regroupe en paquets de 10, 100, 1000 etc…Par exemple, 2015 signifie $2 \times 1000 + 0 \times 100 + 1 \times 10 + 5$, donc 2 paquets de 1000, aucun paquet de 100, un paquet de 10 et un reste de 5 unités.

En informatique, les nombres sont représentés en base 2. Il est relativement aisé de passer de la base 10 à la base 2, et réciproquement.

Pour passer de la base 10 à la base 2. On effectue des divisions euclidienne par 2 du nombre n représenté en base 10, tant que le quotient n'est pas égal à 0. Si les restes successifs sont notés $r_0, r_1, \ldots, r_q$ alors :

$$n = \overline{r_q r_{q-1} \ldots r_1 r_0}^2$$

✎ **Exemple.** $2014 = 2 * 1007 + ⓪$
$1007 = 2 * 503 + ①$
$503 = 2 * 251 + ①$
$251 = 2 * 125 + ①$
$125 = 2 * 62 + ①$
$62 = 2 * 31 + ⓪$
$31 = 2 * 15 + ①$
$15 = 2 * 7 + ①$
$7 = 2 * 3 + ①$
$3 = 2 * 1 + ①$
$1 = 2 * \boxed{0} + ①$
Donc $n = \overline{11111011110}^2$.

Pour passer de la base 2 à la base 10. Si en base 2, $n = \overline{r_q r_{q-1} \ldots r_1 r_0}^2$, alors en base 10 :

$$n = r_q \times 2^q + r_{q-1} \times 2^{q-1} + \cdots + r_1 \times 2^1 + r_0 \times 2^0$$

✎ **Exemple.** Si en base 2, $n = \overline{100111010}^2$, alors en base 10 : $n = 2^8 + 2^5 + 2^4 + 2^3 + 2^1 = 314$.

1.1.2 Représentation des nombres

Les nombres *entiers relatifs* sont représentés en Python sur 32 ou 64 bits (cela dépend de l'ordinateur sur lequel on travaille !).

S'ils sont représentés sur n bits (avec $n = 32$ ou 64), on dispose de 2^n nombre binaires possibles, et en Python sont représentés tous les entiers compris entre -2^{n-1} et $2^{n-1} - 1$ (ce qui fait bien 2^n au total).

Si x est un entier compris entre 0 et $2^{n-1} - 1$, on le représente par sa représentation en base 2. Elle comportera au plus $n-1$ chiffres, et donc le premier bit du bloc de n bits sera égal à 0.

✎ **Exemple.** Sur 8 bits, $x = 23$ est représenté par sa représentation binaire $\overline{00010111}^2$, puisque $0 \leq 23 \leq 2^8 - 1 = 127$.

Si x est un entier compris entre -2^{n-1} et -1, on le représente par la représentation en base 2 du nombre $x + 2^n$. Ce nombre est compris entre $2^n - 2^{n-1} = 2^{n-1}$ et $2^n - 1$. Il n'y a donc pas de conflit avec le cas précédent et le premier bit est égal à 1.

✎ **Exemple.** Sur 8 bits, $x = -128$ est représenté par $\overline{10000000}^2$ puisque $-2^7 = -128 \leq x \leq -1$, et $x + 2^8 = 128$ est représenté en binaire par $\overline{10000000}^2$.

Pour les raisons données ci-dessus, le premier bit est appelé *bit de signe*.

Les nombres entiers forment le type `int`. Lorsqu'on manipule des nombres trop grand (ie plus grand que 2^n pour une représentation sur n bits), on parle de *dépassement de capacité*. En Python 2.x, un nouveau type a été créé, appelé `long` (pour *long int*). En Python 3.x, les types `int` et `long` sont fusionnés, on ne perçoit donc pas de dépassement de capacité.

Pour les nombres réels, le choix de la représentation est beaucoup plus délicat. On montre simplement que tout réel x a une unique écriture de la forme $\varepsilon \times m \times 2^n$ avec $\varepsilon = \pm 1$, $m \in [1,2[$ et $n \in Z$.

Ainsi pour stocker x en mémoire il suffire de stocker ε ,m et n. ε et n sont des entiers , respectivement appelés *signe* et *exposant* de x, donc on procède comme ci-dessus (en étant limité par les dépassements de capacités).
m, appelé *mantisse*, est un nombre réel de l'intervalle $[1,2[$. On va stocker sa représentation en binaire qui s'écrit $\overline{1,a_1a_2a_3\ldots a_p}^2 = \frac{1}{2^0} + \frac{a_1}{2^1} + \frac{a_2}{2^2} + \frac{a_3}{2^3} + \cdots + \frac{a_p}{2^p}$ c'est-à-dire un nombre à virgule avec le chiffre 1 à gauche de la virgule. Comme certains nombres réels ont un nombre infini de chiffres après la virgule, ils ne seront donc *pas représentables en machine*. En pratique, le chiffre à gauche de la virgule étant toujours 1, on ne stocke que les chiffres après la virgule.

Plus précisément on utilise souvent la norme IEEE 754.
• *Simple précision.* On utilise 32 bits : 1 bit de signe, 8 bits pour l'exposant (-126 à 127) et 23 bits de mantisse (avec 1 bit implicite).
• *Double précision.* On utilise 64 bits : 1 bit de signe, 11 bits pour l'exposant (-1022 à 1023) et 52 bits de mantisse (avec 1 bit implicite).

Les nombres ainsi représentés sont appelés *flottants* et forment en Python le type `float`. Un nombre entier peut être du type `int` ou du type `float` : `7` est du type `int` et `7.` est du type `float`.

⚠ Il faudra prendre garde à deux aspects du type float :

1) Certains nombres peuvent avoir une écriture décimale *finie* mais une écriture binaire *infinie*, c'est-à-dire que ce sont des nombres simples d'un point de vue mathématique et pourtant non représentable en machine ! (Ce sont les nombres non dyadiques : tous les irrationnels et certains rationnels)

✎ **Exemple.** $0.4 = \sum_{n=0}^{+\infty}\left(\frac{1}{2^{4n+2}} + \frac{1}{2^{4n+3}}\right) = \overline{0,01100110011001100\ldots}^{2}$ a une représentation binaire infinie, et n'est donc pas représentable en machine.

2) Il existe un plus petit nombre strictement positif représentable en machine, et tout nombre non nul et inférieur à ce nombre sera considéré comme nul ! On ne pourra donc jamais tester l'égalité stricte entre deux flottants, mais plutôt tester si la différence en valeur absolue entre ces deux flottants est inférieure à une constante bien choisie (de l'ordre 10^{-10} la plupart du temps).

Pour mesurer l'espace mémoire on utilise comme unité l'*octet* qui est un bloc de 8 bits. $2^{10} = 1024$ octets donnent un *kilo-octets* (ko), 1024 kilo-octets donnent un *méga-octets* (Mo) et 1024 méga-octets donnent un *giga-octets* (Go). Ainsi un flottant sur 32 bits occupe 4 octets en mémoire, et sur 64 bits il occupe 8 octets.

Remarquer que le passage de l'unité au kilo ne se passe pas comme pour les unités traditionnelles : $1000 = 10^3$ est remplacé par $1024 = 2^{10}$.

1.1.3 Opérations sur les nombres

Trois classes essentiellement (entiers, flottants, complexes).

Les complexes s'écrivent sous la forme `a + bj`. Le réel `b` doit être spécifié, même si `b=1`, et accolé à la lettre `j`. Par exemple `1+1j` désigne le complexe $1 + i$.

Opérations sur les nombres :

```
x + y # somme de deux entiers, réels ou complexes
x * y # produit de deux entiers, réels ou complexes
x - y # différence de deux entiers, réels ou complexes
x / y # division entière (quotient de la division euclidienne)
      # s'applique à des entiers ou des réels. Appliqué à des
      # entier, retourne un 'int', sinon un 'float'
x % y  # modulo (reste de la division euclidienne)
x ** y  # x puissance y
abs(x)  # valeur absolue
int(x)  # partie entière de l'écriture décimale (ne correspond pas
        # à la partie entière mathématique si x < 0:
        # par exemple int(-2.2)=-2.
x.conjugate()  # retourne le conjugué du nombre complexe x
```

```
x.real  # partie réelle
x.imag  # partie imaginaire
```

1.2 Les booléens et les tests

Les deux éléments de la classe des booléens sont `True` et `False` (avec les majuscules).

Opérations sur les booléens :

```
x & y   ou   x and y   # et
x | y   ou   x or y    # ou
x ^ y   # ou exclusif
not x   # négation de x
```

Toute opération valable sur des entiers l'est aussi sur les booléens : le calcul se fait en prenant la valeur 0 pour `False`, et la valeur 1 pour `True`.

La plupart du temps, les booléens sont obtenus au moyen de tests. Tests :

```
x == y    # égalité (la double égalité permet de distinguer
          # syntaxiquement de l'affectation)
x < y     # infériorité stricte
x > y     # supériorité stricte
x <= y    # infériorité large
x >= y    # supériorité large
x != y    # différent (non égalité)
x in y    # appartenance (pour les listes, ensembles, chaînes de caratères)
x is y    # identité (comparaison des identifiants de x et y)
```

1.3 Listes

Une liste permet de manipuler une séquence d'objets.

1.3.1 Exemples de listes

Une liste est représentée entre crochets, et ses éléments sont séparés par des virgules :

```
[1, 10, 2, 3, 2, 2]
```

Une liste peut contenir des objets de n'importe-quel type. Par exemple, des chaînes de caractères :

```
['a', 'b', 'toto']
```

On peut aussi mélanger des objets de types différents dans la même liste :

```
[1, 'a', 2, 3, 3.14159, True, False, 1]
```

On peut même avoir une liste qui contient une liste :

```
[1, [2, 3], 'toto']
```

Une liste peut être vide :

```
[]
```

Les éléments d'une liste peuvent contenir des expressions complexes, faisant appel à des variables ou des fonctions :

```
a = 2
[(2 + 3) * a, a * 2]
```

1.3.2 Listes définies par compréhension

On peut aussi définir une liste *par compréhension* avec la syntaxe :

```
[ fonction(x) for x in liste if test(x) ]
```

Exemples :

```
>>> [2*x for x in range(10)]
        [0, 2, 4, 6, 8, 10, 12, 14, 16, 18]
>>> [x for x in range(10) if x>5]
        [6, 7, 8, 9]
>>> [2*x for x in range(10) if x>5]
        [12, 14, 16, 18]
>>> [2*x for x in range(10) if 2*x>5]
        [6, 8, 10, 12, 14, 16, 18]
>>> [x for x in range(10) if x>5 and x<8]
        [6, 7]
>>> [x+1 for x in [2*x for x in range(10)]]
        [1, 3, 5, 7, 9, 11, 13, 15, 17, 19]
>>> [2*x for x in [x+1 for x in range(10)]]
        [2, 4, 6, 8, 10, 12, 14, 16, 18, 20]
>>> [i*2 for i in range(10)]
        [0, 2, 4, 6, 8, 10 ,12 ,14 ,16 ,18 ,20]
>>> [i for i in range(10) if i % 2 == 0]
        [0, 2, 4, 6, 8]
```

Plus compliqué, obtenir tous les nombres premiers inférieur ou égaux à 50 :

```
>>> noprimes = [j for i in range(2, 8) for j in range(i*2, 50, i)]
>>> primes = [x for x in range(2, 50) if x not in noprimes]
>>> primes
        [2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47]
```

1.3.3 Indices, longueur et intervalles

On accède aux éléments individuels d'une liste en indiquant leur indice entre crochets. La numérotation des éléments commence à zéro. Par exemple :

```
>>> a = [1, 7, 5]
>>> a[0]
        1
>>> a[2]
        5
```

On peut utiliser des indices négatifs pour compter à partir de la fin de la liste :

```
>>> a = [1, 7, 5]
>>> a[-1]
        5
>>> a[-3]
        1
```

La fonction `len` donne la longueur d'une liste :

```
>>> liste = [3, 7, 6]
>>> len(liste)
        3
```

On peut extraire un sous-ensemble des éléments d'une liste en spécifiant un intervalle d'indices de la façon suivante : `liste[min:max]`. Les indices conservés sont ceux qui sont supérieurs ou égaux à `min`, et inférieurs strictement à `max` :

```
>>> liste = [1, 4, 1, 5, 9]
>>> liste[1:3]
        [4, 1]
>>> liste[2:2]
        []
```

`min` ou `max` peut être omis. Dans ce cas, tout se passe comme si `min` était `0`, et `max` était `len(liste)` :

```
>>> liste[2:]
        [1, 5, 9]
>>> liste[:2]
        [1, 4]
>>> liste[:]
        [1, 4, 1, 5, 9]
```

Les indices négatifs peuvent être utilisés dans les intervalles aussi :

```
>>> liste[:-1]
        [1, 4, 1, 5]
```

On peut aussi préciser un pas avec l'instruction `liste[min:max:pas]`. Par exemple pour avoir les éléments d'indices pairs :

```
>>> liste[::2]
        [1, 1, 9]
```

1.3.4 Opérateurs sur les listes

De la même manière que les chaînes, les listes peuvent être concaténées avec l'opérateur +, et multipliées par un entier :

```
>>> [1, 2] + [2]
        [1, 2, 2]
>>> [1, 2] * 3
        [1, 2, 1, 2, 1, 2]
```

On peut comparer deux listes avec l'opérateur == :

```
>>> [1, 2] == [2, 3]
        False
>>> [1, 2] == [1, 2]
        True
```

1.3.5 Modification d'une liste

On peut affecter une valeur à un élément d'une liste :

```
>>> a = [1, 7, 5]
>>> a[1] = 0
>>> a
        [1, 0, 5]
```

On peut aussi affecter une liste à un intervalle :

```
>>> a = [1, 7, 5]
>>> a[0:2] = [2, 3]
>>> a
        [2, 3, 5]
```

Lors d'une affectation à un intervalle, il n'est pas nécessaire que les listes soient de longueur identique :

```
>>> a = [1, 7, 5]
>>> a[0:2] = [2, 3, 4]
>>> a
        [2, 3, 4, 5]
>>> a[0:2] = []
>>> a
        [4, 5]
```

La méthode `append` permet d'ajouter un élément à la fin de la liste :

```
>>> a = [1, 4, 5]
>>> a.append(0)
>>> a
        [1, 4, 5, 0]
```

1.3.6 Sémantique de pointeur

L'affectation d'une liste à une variable ne crée pas une copie de la liste. Voici un exemple de programme qui illustre ce phénomène :

```
>>> a = [1, 7, 5]
>>> b = a
>>> a[1] = 0
>>> b
        [1, 0, 5]
```

On voit que la modification de la liste `a` affecte aussi la liste `b`. Cela peut se représenter de la façon suivante dans un tableau d'exécution :

a	b	L1
L1		[1, 7, 5]
	L1	[1, 0, 5]

Pour faire une nouvelle copie de la liste, il est nécessaire d'utiliser une expression comme ci-dessous :

```
>>> a = [1, 7, 5]
>>> b = a[:] # b = a + [], ou b = list(a) ou b = [ x for x in a ] fonctionnent aussi
>>> a[1] = 0
>>> b
        [1, 7, 5]
```

Ce qui donne :

a	b	L1	L2
L1		[1, 7, 5]	
	L2		[1, 7, 5]
			[1, 0, 5]

1.3.7 Autres fonctions et opérations pratiques

On peut trier une liste « sur place » avec la méthode `sort` :

```
>>> a = [1, 5, 3, 2]
>>> a.sort()
>>> print(a)
        [1, 2, 3, 5]
```

Si on souhaite conserver la liste a intacte, et créer une nouvelle liste, on peut utiliser la fonction `sorted` :

```
>>> a = [1, 5, 3, 2]
>>> b = sorted(a)
>>> print(b)
        [1, 2, 3, 5]
>>> print(a)
        [1, 5, 3, 2]
```

L'appartenance à une liste peut se tester avec l'opérateur `in` :

```
>>> 1 in [3, 4, 5]
        False
>>> 4 in [3, 4, 5]
        True
```

Le minimum et le maximum peuvent se trouver avec les fonctions min et max :

```
>>> min([3, 1, 4, 1, 5, 9, 2])
        1
>>> max([3, 1, 4, 1, 5, 9, 2])
        9
```

`del` permet de supprimer un élément d'une liste :

```
>>> a = [0, 1, 2, 3, 4]
>>> del a[2]
>>> a
        [0, 1, 3, 4]
```

Notez qu'on peut faire la même opération en affectant une liste vide :

```
>>> a = [0, 1, 2, 3, 4]
>>> a[2:3] = []
>>> a
        [0, 1, 3, 4]
```

1.4 Chaînes de caractères

Les chaînes de caractères peuvent se manipuler de manière quasi identique aux listes, à la différence près qu'il est impossible de modifier une chaîne de caractères sur place. Par exemple :

```
>>> c = "bonjour"
>>> c[0]
        b
>>> c[0:3]
        bon
>>> len(c)
        7
```

On peut aussi délimiter une chaîne de caractères par des apostrophes `' '` ou pas des triples guillements `""" """`.

Si on cherche à modifier la chaîne, il se produit des erreurs :

```
>>> c[0] = "B"
        Traceback (most recent call last):
        File "<stdin>", line 1, in <module>
        TypeError: 'str' object does not support item assignment
>>> c.append("!")
        Traceback (most recent call last):
        File "<stdin>", line 1, in <module>
        AttributeError: 'str' object has no attribute 'append'
```

Mais on peut obtenir un effet similaire en construisant une nouvelle chaîne :

```
>>> c = "B" + c[1:]
>>> c
        Bonjour
```

L'opérateur d'appartenance in permet aussi de tester les sous-chaînes :

```
>>> "x" in c
        False
>>> "Bon" in c
        True
```

Attention : `in` ne fonctionne pas de cette façon pour les listes.

```
>>> [1, 2] in [1, 2, 3]
        False
>>> [1, 2] in [[1, 2], 3]
        True
```

2 Programmation

2.1 Variables et affectations

En Python , les variables bénéficient d'un typage dynamique : le type est détecté automatiquement lors de la création de la variable par affectation. Il n'est donc pas nécessaire de déclarer la variable avant de l'utiliser (comme dans la plupart des autres langages).
L'affectation est l'action consistant à donner une valeur à une variable.

```
a = 2   # affectation
```

L'affectation d'une valeur à une variable se fait avec l'égalité. La ligne ci-dessus donne la valeur 2 à la variable `a`, ceci jusqu'à la prochaine modification.

Une affectation peut prendre en argument le résultat d'une opération :

```
a = 2 + 4
```

Ce calcul peut même utiliser la variable a elle-même :

```
a = a + 4*a*a
```

Dans ce cas, le calcul du membre de droite est effectué d'abord, et ensuite seulement l'affectation. Ainsi, le calcul est effectué avec l'ancienne valeur de a. Il faut en particulier qu'une valeur ait déjà été attribuée à a. Par exemple, si a a initialement la valeur 2, la ligne ci-dessus attribue la valeur 18 à a.

Il existe des raccourcis pour certaines opérations de ce type :

```
a += 2   # Équivaut à a = a + 2
a -= 1   # Équivaut à a = a - 1
a *= 3   # Équivaut à a = a * 3
a /= 2   # Équivaut à a = a / 2 (en Python 2.7 division entière si possible)
a %= 3   # Équivaut à a = a % 3 (reste modulo 3)
a **= 4  # Équivaut à a = a ** 4 (puissance)
etc.
```

Pour afficher la valeur d'une variable en ligne de commande, il suffit de taper le nom de la variable après l'invite de commande :

```
>>> a = 2
>>> a
        2
```

Faire un affichage du contenu d'une variable lors de l'exécution d'un programme nécessite la fonction `print()`, qui affiche une chaîne de caractères. On peut aussi l'utiliser en ligne de commande :

```
>>> print(a)
        2
```

Cette fonction réalise une conversion automatique préalable du contenu de la variable en chaîne de caractères, car la fonction `print()` prend toujours en argument une chaîne de caractères.

Il est possible d'effectuer *simultanément* plusieurs affectations de variables (de même type ou non) :

```
>>> a, b, c = 5, 'bonjour! ', 3.14
                  # ici a reçoit 5, b reçoit 'bonjour! ', c reçoit 3.14
```

On peut initialiser plusieurs variables avec une même valeur en utilisant des = successifs.

```
>>> x = y = z = 1  # initialise les trois variables x,y,z à la valeur 1
>>> x, y, z
        (1, 1, 1)
>>> a, b = c, d = 3, 8     # pose a=c=3, et b=d=8
>>> (a, b) = (c, d) = (3, 8)     # idem, mais c'est plus lisible comme ça
>>> a, b, c, d
        (3, 8, 3, 8)
```

L'affectation simultanée est un bon moyen d'échanger le contenu de deux variables :

```
>>> x, y = 3, 7    # on donne à x la valeur 3, à y la valeur 7
>>> x, y = y, x    # on échange les valeurs de x et y
>>> [x, y]
        [7, 3]
```

On peut bien sûr effectuer toute permutation sur un nombre quelconque de variables.

```
>>> a, b, c, d, e = 1, 2, 3, 4, 5
>>> a, b, c, d, e = b, c, d, e, a   # permutation circulaire sur a, b, c, d, e
>>> a, b, c, d, e
        (2, 3, 4, 5, 1)
```

Comme dans toute évaluation, l'expression qui suit le signe = est évaluée en premier (donc avant que la première des affectations ne soit réalisée).

```
>>> u, v = 2, 3    # ici u reçoit 2, et v reçoit 3
>>> u, v = v*v, u*u    # à gauche de =, lire la séquence 9, 4
>>> u, v    # donc ne pas croire qu'on a effectué u=v²=9 puis v=u²=81
        (9, 4)
```

Dans un langage qui ne gère pas l'affectation simultanée, l'échange des valeurs de deux variables x et y nécessite une troisième variable temporaire z :

```
>>> z=x
>>> x=y
>>> y=z
```

Chaque variable possède un *identifiant* (l'adresse mémoire associée), et un *type* (la nature de l'objet stocké dans la variable). L'identifiant change à chaque réaffectation, le type peut changer lui-aussi. Les méthodes associées à un objet peuvent le modifier sans changement d'identifiant.

```
type(a)    # affiche le type (la classe) de la variable ou de l'objet
id(a)      # affiche l'identifiant (l'adresse en mémoire)
```

Les principaux types (hors modules complémentaires) :

```
int        # Entiers, de longueur non bornée
float      # Flottants (réels)
complex    # Nombres complexes
bool       # Booléens (True / False)
list       # Liste
tuple      # n-uplets
str        # Chaînes de caractères (string)
function   # Fonctions
```

Par exemple :

```
>>> type(87877654)
        <class 'int'>
>>> type(1.22)
        <class 'float'>
>>> type(True)
        <class 'bool'>
>>> type([1,2])
        <class 'list'>
>>> type('abc')
        <class 'str'>
>>> type(lambda x:x*x)
        <class 'function'>
```

Il existe d'autres types dans des modules spécifiques, en particulier le type `array` du module `numpy`, pour les tableaux et les matrices.

Certains objets d'un type donné peuvent être convertis en un autre type compatible. Conversions de type :

```
int(x)      # Convertit un objet x en nombre entier.
float(a)   # Convertit l'entier a en réel flottant
complex(a)   # Convertit l'entier ou le flottant a en complexe
str(x)   # Convertit l'objet x de type quelconque en
         # une chaîne de caractères.
list(x)     # Convertit un l'objet x en liste.
tuple(x)     # Convertit un l'objet x en liste.
```

Par exemple :

```
>>> int('24')
        24
>>> float(2)
        2.0
>>> complex(2)
        (2+0j)
>>> str([2,3,1])
        '[2, 3, 1]'
>>> str(77+2)
        '79'
>>> list((1,2,3))
        [1, 2, 3]
>>> list('ens')
        ['e', 'n', 's']
>>> tuple('abc')
        ('a', 'b', 'c')
```

2.2 Les structures conditionnelles

Une seule structure à connaître ici :

```
if test1:
    instructions1
elif test2:
    instructions2
elif test3:
    instructions3
...
else:
    instructions4
```

Les clauses `elif` (il peut y en avoir autant qu'on veut), ainsi que `else` sont facultatives.

La structure est à comprendre comme suit :

- Les `intructions1` sont effectuées uniquement si le `test1` est `True`
- Les instructions `instructions2` sont effectuées uniquement si le `test1` est `False` et le `test2` est `True`
- Les instructions `instructions3` sont effectuées uniquement si le `test1` et le `test2` sont `False` et le `test3` est `True`
- ...
- Les instructions `instructions4` sont effectuées dans tous les autres cas restants.

Le mot `elif` est à comprendre comme une abréviation de `else if`. Ainsi, la structure précédente est en fait équivalente à une imbrication de structures conditionnelles simples :

```
if test1:
    instructions1
else:
    if test2:
        instructions2
    else:
        if test3:
            instructions3
        else:
            instructions4
```

⚠ N'oubliez pas les double-points, et faites attention à l'indentation !

Les tests peuvent être remplacés par toute opération fournissant une valeur booléenne. On peut en particulier combiner plusieurs tests à l'aide des opérations sur les booléens. Par ailleurs, Python autorise les inégalités doubles (par exemple un test `2 < x < 4`).

2.3 Structures itératives

Il faut distinguer deux types de structures itératives ; celles où le nombre d'itération est *connu dès le début* (itération sur un ensemble fixe de valeurs), et celles où l'arrêt de l'itération est *déterminée par un test.*

La structure itérative `for` permet d'itérer sur un nombre fini de valeurs. Elle s'utilise avec un objet itérable quel qu'il soit. La boucle est alors répétée pour toute valeur fournie par l'itérateur. L'objet itérable étant fixé, le nombre d'itérations est connue dès le départ.

Boucle for, pour un nombre d'itérations connu d'avance :

```
for i in objet_iterable:
    instructions    # qui peuvent dépendre de i
```

Par exemple :

```
for i in [1,2,5]:
    print(i)
```

va afficher les 3 valeurs `1`, `2` et `5`.

Le classique `for i = 1 to n` (ou similaire) qu'on trouve dans la plupart des langages se traduit alors par :

```
for i in range(1,n+1):
    instructions
```

Les *itérateurs* sont des objets égrénant des valeurs les unes après les autres en les retournant successivement. On peut utiliser ces valeurs successives grace à l'instruction `for`.

L'itérateur le plus utile pour nous sera `range`, énumérant des entiers dans un intervalle donné.

```
range(a,b,p)  # retourne les entiers de a à b-1, avec un pas p
range(a,b)    # de même, avec la valeur par défaut p=1
range(b)      # de même, avec la valeur par défaut a=0
```

Ainsi, `range(b)` permet d'énumérer les entiers de `0` à `b-1`.

Boucles portant sur des objets itérables :

```
for i in range(n):        # exécute l'instruction instr pour toute valeur de
    instr                 # i entre 0 et n-1

for i in liste:           # exécute l'instruction instr pour toute valeur (non
     instr                # nécessairement numérique) de i dans la liste liste

for i in chaine:          # exécute l'instruction instr pour tout caractère
     instr                # i de la chaîne de caractère chaîne.
```

La boucle while permet d'obtenir un arrêt de l'itération par un test. Il s'agit en fait plutôt d'une condition de continuation, puisqu'on itère la boucle tant qu'une certaine condition est vérifiée.

```
while cond:    # Boucle while, pour l'arrêt des itérations par un test
    instructions
```

Tant que la condition `cond` est satisfaite, l'itération se poursuit.

Remarquez qu'une boucle `for` traditionnelle (de 1 à n) peut se traduire aussi par une boucle `while` :

```
i = 1
while i <= n:
    instructions
    i +=1
```

⚠ Remarquer que l'incrémentation `i+=1` n'est pas géré automatiquement (contrairement à la boucle `for`).

En revanche, ce n'est pas le cas de la boucle `for` sur d'autres objets itérables : on ne peut pas systématiquement avoir recours à une boucle `while`.

2.4 Fonctions

Une fonction est un morceau de programme que l'on isole, et qu'on peut faire dépendre de paramètres. Cela permet de :

- Dégager le coeur même du programme de toute technique en isolant cette technique dans des fonctions (clarté et lisibilité du code)
- Pouvoir répéter une certaine action à plusieurs endroits d'un même programme, ou même dans des programmes différents (en définissant des fonctions dans des fichiers séparés qu'on importe ensuite)
- Rendre certains actions plus modulables en les faisant dépendre de paramètres.

Une fonction prend en paramètres un certain nombre de variables ou valeurs et retourne un objet (éventuellement None), calculé suivant l'algorithme donné dans sa définition. Il peut aussi agir sur les variables ou les périphériques.

La syntaxe générale de la définition d'une fonction est la suivante :

```
def nom_de_la_fonction(x,y):
    """Description"""
    instructions
return résultat
```

La chaîne de caractères en raw string (c'est-à-dire entre les `"""`) est facultative. Elle permet de définir la description qui sera donnée dans la page d'aide associée à cette fonction. Les instructions sont les différents étapes permettant d'arriver au résultat. L'instruction spécifique commençant par return permet de retourner la valeur de sortie de la fonction. Si cette ligne n'est pas présente, la valeur de sortie sera None (pas de sortie). Cela peut être le cas pour une fonction dont le but est simplement de faire un affichage.

Voici un exemple :

```
def truc(x,y):
    """Que peut bien calculer cette fonction?"""
    while x >= y:
    x -= y
return x
```

Par exemple, en ajoutant la ligne `print(truc(26,7))` et en exécutant le programme, on obtient la réponse `5`.

Si on veut savoir ce que calcule cette fonction, on peut rajouter la ligne `help(truc)`. L'exécution du programme nous ouvre alors la page d'aide de la fonction `truc` :

```
Help on function truc in module __main__:
truc(x, y)
Que peut bien calculer cette fonction?
(END)
```

ce qui en l'occurrence ne va pas vous aider beaucoup. Il va falloir faire marcher vos neurones...

Enfin, notons qu'on peut décrire une fonction ponctuellement (sans la définir globalement) grâce à l'in- struction `lambda` :

```
lambda x: fonction(x)      # désigne la fonction en la variable x définie
                           # par l'expression fonction(x)
```

Cela permet par exemple de donner une fonction en paramètre, sans avoir à la définir globalement par `def`.

Pour qu'une fonction donne plusieurs résultats en sortie, il suffit des les donner dans une liste :
`return` $[x_0, x_1, ..., x_n]$

2.5 Terminaison et correction d'un algorithme

2.5.1 Terminaison

Dans un algorithme *non récursif* correctement structuré, et sans erreur de programmation (par exemple modification manuelle d'un compteur de boucle for), les seules structures pouvant amener une non terminaison d'un algorithme sont les boucles `while`.

On prouve la terminaison d'une boucle en exhibant un *variant de boucle*, c'est-à-dire une quantité v définie en fonction des variables $(x_1, ..., x_k)$ constituant l'état de la machine, et de n, le nombre de passages effectués dans la boucle et telle que :

- v ne prenne que des valeurs entières ;
- la valeur de v en entrée de boucle soit toujours positive ;
- v prenne des valeurs strictement décroissantes au fur et à mesure des passages dans la boucle. Ainsi, si v_n désigne la valeur de v au n-ième passage dans la boucle, si les passages n et $n+1$ ont lieu, on a $v_n < v_{n+1}$.

Le *principe de descente infini* permet alors d'affirmer que :

1. Si, pour une boucle donnée, on peut exhiber un variant de boucle, alors le nombre de passages dans la boucle est finie.
2. Si, pour un algorithme donné, on peut exhiber, pour toute boucle de l'algorithme, un variant de boucle, alors l'algorithme s'arrête en temps fini.

⚠ Il faut bien justifier la validité du variant de boucle pour toute valeur possible d'entrée.

La terminaison assure un arrêt théorique de l'algorithme. En pratique, un deuxième paramètre entre en jeu : le temps de réponse. Ce temps de réponse sans être infini, peut être très grand (certains algorithmes exponentiels peuvent avoir un temps de réponse de l'ordre de milliards d'années pour des données raisonnablement grandes !). Évidemment dans ce cas, même si l'algorithme s'arrête en théorie, c'est d'assez peu d'intérêt.

Prenons comme exemple l'algorithme d'Euclide qui calcule le PGCD deux entiers relatifs :

```
1 def euclide(a,b):
2     while b>0:
3         a,b = b,a%b
4     return a
```

La variable $|b|$ est un variant de boucle, ce qui prouve la terminaison de cet algorithme.

2.5.2 Correction

La terminaison d'un algorithme n'assure pas que le résultat obtenu est bien le résultat attendu, c'est- à-dire que l'algorithme répond bien à la question posée initialement. L'étude de la validité du résultat qu'on obtient fait l'objet de ce paragraphe. On parle de l'étude de la *correction* de l'algorithme.

Reprenons l'exemple de l'algorithme d'Euclide. Cet algorithme a pour objet le calcul du pgcd de a et b. La preuve mathématique classique de la correction de l'algorithme repose sur la constatation suivante : si r est le reste de la division euclidienne de a par b, alors $a \wedge b = b \wedge r$. La preuve mathématique de ce point ne pose pas de problème.

On constate alors que la quantité $w = a \wedge b$ prend toujours la même valeur à l'entrée d'une boucle (correspondant aussi à la valeur à la sortie de la boucle précédente). C'est le fait d'avoir une valeur constante qui nous assure que la valeur initiale recherchée $a \wedge b$, est égale à $a \wedge b$ pour les valeurs finales obtenues pour a et b. Or, en sortie de boucle $b = 0$, donc $a \wedge b = a$. Ainsi, le pgcd des valeurs initiales de a et b est égal à la valeur finale de a.

De façon générale, l'étude de la correction d'un algorithme se fera par la recherche d'une quantité *invariante* d'un passage à l'autre dans la boucle. On appellera donc *invariant de boucle* une quantité w dépendant des variables $x_1, \ldots, x_k$ en jeu ainsi éventuellement que du compteur de boucle n, telle que :

- la valeur prise par w en entrée d'itération, ainsi que la valeur qu'aurait pris w en cas d'entrée dans l'itération suivant la dernière itération effectuée, soit constante.
- Si le passage initial dans la boucle est indexé 0 et que la dernière itération effectuée est indexée $n-1$, l'égalité $w_0 = w_n$ permet de prouver que l'algorithme retourne bien la quantité souhaitée.

Dans le cas de l'algorithme d'Euclide, la quantité $w = a \wedge b$ est un invariant de boucle : il est constant, et à la sortie, puisque $b = 0$, il prend la valeur a. Ainsi, la valeur retournée par l'algorithme est bien égal à $a \wedge b$, pour les valeurs initiales de a et b, passées en paramètres.

3 Modules

3.1 Généralités

De nombreuses fonctions sont définies dans des modules spécialisées ; cela permet de ne pas encombrer la mémoire de la définition de plein de fonctions dont on ne se servira pas. On peut se contenter de charger (importer) les modules contenant les fonctions utilisées (ou même selectionner les fonctions qu'on importe dans chaque module). De cette manière, on peut définir un très grand nombre de fonctions, sans pour autant alourdir l'ensemble.

Pour utiliser une fonction `fonct` d'un module `mod`, il faut importer cette fonction, ou le module entier avant l'utilisation de cette fonction. Cela peut se faire de 3 manières.
• Import simple du module :

```
import mod

#Cette instruction permet de faire savoir à l'ordinateur qu'on
#utilise le module mod. On peut alors utilier les fonctions
#de ce module en les faisant précéder du préfixe mod (nom du
#module):

mod.fonct()
```

L'utilisation de préfixes permet d'utiliser des fonctions ayant éventuellement même nom et se situant dans des modules différents. Certains noms de module sont longs, ou seront utilisés très souvent. On peut, au moment de l'import du module, créer un alias, qui permettra d'appeler les fonctions de ce module en donnant comme préfixe l'alias au lieu du nom complet :

```
import mod as al

al.fonct()
```

• Import d'une fonction d'un module :

```
from mod import fonct

# Cette instruction permet d'importer la définition de la fonction
# fonct, qui peut alors être utilisée sans préfixe:

fonct()

# les autres fonctions du module ne peuvent pas être utilisées, même
# avec le préfixe.
```

Attention, si une fonction du même nom existe auparavant (de base dans Python, ou importée précédem- ment), elle sera écrasée.

• Import de toutes les fonctions d'un module :

```
from mod import *

# Cette instruction permet d'importer la définition de toutes les
# fonctions du module

fonct()

# les autres fonctions du module peuvent être utilisées de même.
```

⚠ Attention aussi aux écrasements possibles. Par ailleurs, l'import est plus long à effectuer, et d'une gestion plus lourde, si le module est gros.

3.2 Math

Le module `math` offre les fonctions et les constantes mathématiques usuelles, résumées dans ce tableau :

Python	mathématiques
`math.factorial(n)`	$n!$
`math.fabs(x)`	$\|x\|$
`math.exp(x)`	rme^x
`math.log(x)`	$\ln(x)$
`math.log10(x)`	$\log_{10}(x)$
`math.pow(x, y)`	x^y
`math.sqrt(x)`	$\sqrt{x}$
`math.sin(x)`	$\sin(x)$
`math.cos(x)`	$\cos(x)$
`math.tan(x)`	$\tan(x)$
`math.asin(x)`	$\arcsin(x)$
`math.acos(x)`	$\arccos(x)$
`math.atan(x)`	$\arctan(x)$
`math.pi`	π
`math.e`	e

3.3 Numpy

Dans toute la suite on supposera qu'on a effectué :

```
import numpy as np
```

3.3.1 Manipuler des listes et des matrices

`numpy` propose un type `numpy.ndarray` qui offre les fonctionnalités des listes de Python, mais aussi des routines de calcul efficaces. Dans la suite nous appellerons « tableau numpy » toute instance du type `numpy.ndarray` (ou plus simplement « tableau »). Attention : contrairement aux listes, les éléments d'un tableau `numpy` doivent tous être de même type.

• Création de tableaux monodimensionnels

La fonction `np.array` transforme une liste Python en tableau `numpy` :

```
>>> u = [1, 4, 6, 2]
>>> type (u)
        <type 'list'>
>>> t = np.array (u)
>>> t
        array([1, 4, 6, 2])
>>> type (t)
        <type 'numpy.ndarray'>
```

La fonction `np.zeros` crée un tableau composé uniquement de zéros, le nombre d'éléments étant passé en paramètre. Par défaut les éléments sont de type `float` mais on peut le modifier avec le paramètre facultatif `dtype`.

```
>>> np.zeros(5)
        array([ 0., 0., 0., 0., 0.])
>>> np.zeros(5, dtype = int)
        array([0, 0, 0, 0, 0])
```

La fonction `np.ones` fonctionne de la même manière que `np.zeros` sauf que les éléments sont tous égaux à 1.

```
>>> np.ones(3, dtype=int)
        array([1, 1, 1])
>>> 6*np.ones(4)
        array([ 6., 6., 6., 6.])
```

La fonction `np.arange(debut,fin,pas)` fonctionne comme `range` mais renvoie un objet de type `array` :

```
>>> x=np.arange(0,1,0.3)
>>> x
        array([ 0.,  0.3,  0.6,  0.9])
```

• <u>Création de tableaux bidimensionnels</u>

La fonction `np.array` permet de transformer une matrice Python en tableau bidimensionnel `numpy` :

```
>>> np.array([[1, 2], [3, 4]])
        array([[1, 2],
               [3, 4]])
```

Les fonctions `np.zeros` et `np.ones` fonctionnent de façon similaire au cas monodimensionnel, en renvoyant une matrice quand le paramètre est un couple d'entiers. Attention à la double paire de parenthèses.

```
>>> np.zeros ((2,4))
        array([[ 0., 0., 0., 0.],
               [ 0., 0., 0., 0.]])
```

La fonction `np.eye` prend en paramètre un seul entier, et renvoie la matrice identité : les coefficients de la diagoanle principale sont égaux à 1, les autres à 0.

```
>>> np.eye(3)
        array([[ 1., 0., 0.],
               [ 0., 1., 0.],
               [ 0., 0., 1.]])
```

La fonction `np.diag` prend en paramètre une liste $[t_0, \ldots, t_{n-1}]$ (ou un tableau `numpy`) et renvoie la matrice diagonale ayant les éléments t_i sur la diagonale et des zéros ailleurs :

```
>>> np.diag([5, 7, 2])
        array([[5, 0, 0],
               [0, 7, 0],
               [0, 0, 2]])
```

• Accès à un élément

Accès à un élément d'un tableau monodimensionnel :

```
>>> t=np.array([3,7,1])
>>> t[2]
        1
>>> t[-1]=20
>>> t
        array([ 3, 7, 20])
```

Accès à un élément d'un tableau bidimensionnel :

```
>>> m = np.array([[3, 8, 1], [2, 7, 3]])
>>> m[1] # ligne 1
         array([2, 7, 3])
>>> m[1,2] = 50 # ligne 1 colonne 2
>>> m
         array([[ 3, 8, 1],
                [ 2, 7, 50]])
```

• Opérateurs vectorialisés

Chaque fonction mathématique usuelle possède une fonction `numpy` qui lui est homonyme et qui calcule la même chose sur les objets de type `float`. Par exemple il existe `math.sin` et `np.sin`. L'avantage des fonctions `numpy`, c'est que si `a` est un tableau $\mathtt{a} = \mathtt{array}([\mathtt{a}_0, \ldots, \mathtt{a}_{n-1}])$, l'appel `f(a)` renvoie le tableau $\mathtt{array}[\mathtt{f}(\mathtt{a}_0), \ldots, \mathtt{f}(\mathtt{a}_{n-1})]$. Ce mécanisme s'applique aussi aux matrices. Exemple :

```
>>> x=np.array([3,4])
>>> np.exp(x)
        array([ 20.08553692, 54.59815003])
>>> import math
>>> math.exp(x)
        TypeError: only length-1 arrays can be converted to Python scalars
```

Lorsque la fonction f ne dispose pas d'équivalent `numpy` (par exemple si elle a été créée par l'utilisateur), on peut la vectorialiser grâce à l'instruction `np.vectorize(f)`. Exemple :

```
>>> x=np.array([3,4])
>>> np.vectorize(bin)(x)
        array(['0b11', '0b100'],
              dtype='|S5')
```

Quand les paramètres sont des tableaux `numpy`, les opérations usuelles (+, −, ∗, /, ∗∗) effectuent les opérations coefficient par coefficient.

```
>>> a = np.array([[1, 2], [3, 4]])
>>> b = np.eye(2)
>>> a
        array([[1, 2],
              [3, 4]])
>>> b
        array([[ 1., 0.],
               [ 0., 1.]])
>>> a+b
        array([[ 2., 2.],
               [ 3., 5.]])
>>> a-b
        array([[ 0., 2.],
               [ 3., 3.]])
>>> a*b
        array([[ 1., 0.],
               [ 0., 4.]])
>>> a/b
        __main__:1: RuntimeWarning: divide by zero encountered in divide
        array([[ 1., inf],
               [ inf, 4.]])
>>> a**2
        array([[ 1, 4],
               [ 9, 16]])
```

Il est possible aussi d'effectuer une opération du type constante + tableau `numpy`, dans ce cas l'opération se fait coefficient par coefficient. Ceci existe aussi avec les autres opérations.

```
>>> 3+a
        array([[4, 5],
               [6, 7]])
>>> 3*a
        array([[ 3, 6],
               [ 9, 12]])
```

Attention aux homonymies d'opérateurs :

— L'opérateur + de `numpy` n'est pas du tout le même que celui des listes Python. Avec les listes Python cet opérateur effectue une *concaténation*, avec les tableaux `numpy` il s'agit d'une addition terme à terme.

— Les opérateurs `*`, `/`, `**` de `numpy` ne correspondent pas du tout aux opérations matricielles mathématiques. Ils effectuent simplement un produit (ou une division ou une mise à la puissance) coefficient par coefficient. Pour les opérations matricielles mathématiques, se référer ci–dessous à la section « quelques outils d'algèbre linéaire ».

• Slicing, vues, copies

Il est possible de sélectionner une sous-matrice en ne gardant que certaines lignes consécutives et/ou certaines colonnes consécutives.

Un intervalle d'indices est d'une des formes suivantes :
— `debut:fin:pas` (les indices entre `debut` inclus et `fin` exclu, séparés par la valeur `pas`) ;
— `debut:fin` (tous les indices entre `debut` inclus et `fin` exclu) ;
— `debut:` (tous les indices depuis `debut` inclus) ;
— `:fin` (tous les indices jusqu'à fin exclu) ;
— `:` (tous les indices) ;
— `i` (seulement l'indice `i`).

La syntaxe `matrice[intervalle d'indices de lignes, intervalle d'indices de colonnes]` permet de créer une sous-matrice correspondant aux lignes et aux colonnes sélectionnées.

```
>>> m = np.array([[1, 2, 5, 3], [4, 7, 2, 4], [3, 1, 2, 6]])
>>> m
        array([[1, 2, 5, 3],
                [4, 7, 2, 4],
                [3, 1, 2, 6]])
>>> m[::2,:]   # lignes d'indices pairs
        array([[1, 2, 5, 3],
               [3, 1, 2, 6]])
>>> m[1:,1:3]   # lignes 1 et suivantes, colonnes 1 et 2
        array([[7, 2],
                [1, 2]])
>>> m[:,2]   # colonne 2
        array([5, 2, 2])
```

Une sous-matrice obtenue ainsi (méthode dite de *slicing*) s'appelle une vue. Les coefficients ne sont pas recopiés, et l'indexage se fait selon la géométrie de la sous-matrice :

```
>>> a = m[1:, 1:]
>>> a
        array([[7, 2, 4],
               [1, 2, 6]])
>>> a[0,0] = 20
>>> m
        array([[ 1, 2, 5, 3],
               [ 4, 20, 2, 4],
               [ 3, 1, 2, 6]])
>>> a[:,:2] = np.eye(2)
```

```
>>> a
        array([[1, 0, 4],
               [0, 1, 6]])
>>> m
        array([[1, 2, 5, 3],
               [4, 1, 0, 4],
               [3, 0, 1, 6]])
```

On constate dans l'exemple précédent qu'une vue peut servir de membre de gauche d'une affectation. Si on veut une duplication des éléments d'une matrice `numpy`, on utilise la fonction `np.copy` :

```
>>> a = np.copy(m[1:, 1:])
>>> a[0, 0] = 20
>>> m
        array([[1, 2, 5, 3],
               [4, 1, 0, 4],
               [3, 0, 1, 6]])
```

• `shape, reshape, concatenate`

L'attribut `shape` d'un tableau `numpy` est un n-uplet où n est le nombre d'entrées du tableau. Si le tableau est monodimensionnel, l'attribut `shape` est un 1-uplet contenant le nombre d'éléments. Si le tableau est bidimensionnel, l'attribut `shape` est un couple (nombre de lignes, nombre de colonnes).

La méthode `reshape` renvoie un nouveau tableau, dont la géométrie est un uplet passé en paramètre. Le remplissage se fait ligne par ligne (si la matrice de départ ne possède pas assez d'éléments pour la géométrie passée en paramètre, la méthode `reshape` réutilise les éléments du début ; si la matrice de départ en possède trop la méthode `reshape` n'utilise pas tous les éléments).

```
>>> import numpy as np
>>> vecteur = np.array([1,5,2,7,3,8])
>>> vecteur.shape
        (6,)
>>> matrice = np.concatenate((vecteur, vecteur+1, 2*vecteur))
>>> matrice
        array([1, 5, 2, 7, 3, 8, 2, 6, 3, 8, 4, 9, 2, 10, 4, 14, 6, 16])
>>> matrice.shape
        (18,)
>>> p = matrice.reshape(2,9)
>>> p
        array([[ 1, 5, 2, 7, 3, 8, 2, 6, 3],
               [ 8, 4, 9, 2, 10, 4, 14, 6, 16]])
>>> p.shape
        (2, 9)
```

3.3.2 Quelques outils d'algèbre linéaire

• Produit matriciel

Syntaxe de l'appel : `numpy.dot(matrice A, matrice B)`
Valeur de retour : le produit matriciel $A.B$

⚠ L'opérateur $*$ effectue une multiplication terme à terme des éléments des deux matrices, et n'a donc rien à voir avec un produit matriciel.

Exemple :

```
>>> from numpy.linalg import dot
>>> dot([[1,2],[3,4]], np.eye(2))   # syntaxe correcte du produit matriciel
        array([[1., 2.],
               [3., 4.]])
>>> [[1,2],[3,4]] * np.eye(2)   # Faux (produit terme a terme)
        array([[1., 0.],
               [0., 4.]])
```

• Rang d'une matrice

Syntaxe de l'appel : `numpy.rank(matrice A)`
Valeur de retour : le rang de la matrice A

Exemple :

```
>>> import numpy as np
>>> a=np.arange(15).reshape(3,5)
>>> a
        array([[ 0,  1,  2,  3,  4],
               [ 5,  6,  7,  8,  9],
               [10, 11, 12, 13, 14]])
>>> np.rank(a)
        2
```

• Inverse d'une matrice

Syntaxe de l'appel : `numpy.linalg.inv(matrice A)`
Valeur de retour : A^{-1} (provoque une erreur si A n'est pas inversible).

Exemple :

```
>>> from numpy.linalg import inv
>>> inv([[1,2],[3,4]])
        array([[-2. , 1. ],
               [ 1.5, -0.5]])
```

Comme d'habitude avec les logiciels de calcul scientifique, il faut d'abord savoir si la matrice est inversible pour l'inverser, ou encore rester critique vis à vis du résultat retourné. L'exemple suivant est caractéristique.

```
>>> import numpy as np
>>> a=np.arange(16).reshape(4,4)
>>> a
        array([[ 0,  1,  2,  3],
               [ 4,  5,  6,  7],
               [ 8,  9, 10, 11],
               [12, 13, 14, 15]])
>>> np.rank(a)  # la matrice n'est pas inversible
        2
>>> np.linalg.inv(a)
         array([[  9.00719925e+14,  -4.50359963e+14,  -1.80143985e+15,
                   1.35107989e+15],
                [ -2.40191980e+15,   2.70215978e+15,   1.80143985e+15,
                  -2.10167983e+15],
                [  2.10167983e+15,  -4.05323966e+15,   1.80143985e+15,
                   1.50119988e+14],
                [ -6.00479950e+14,   1.80143985e+15,  -1.80143985e+15,
                   6.00479950e+14]])
```

Les valeurs très grandes laissent tout de même planer un certain soupçon.

• Résolution d'un système de Cramer

Syntaxe de l'appel : `numpy.linalg.solve(matrice A, vecteur B)`
Valeur de retour : le vecteur X solution de $AX = B$

Exemple, résolution de $\begin{cases} x+2y=2 \\ 3x+4y=8 \end{cases}$

```
>>> from numpy.linalg import solve
>>> solve([[1,2],[3,4]], [2,8])
        array([ 4., -1.])
```

Il en est de même que pour l'inversion de matrice : il faut savoir rester critique.

```
>>> import numpy as np
>>> a=np.arange(16,dtype=float).reshape(4,4)
>>> a
        array([[  0.,   1.,   2.,   3.],
               [  4.,   5.,   6.,   7.],
               [  8.,   9.,  10.,  11.],
               [ 12.,  13.,  14.,  15.]])
>>> b=np.arange(4,dtype=float)
>>> b[1]=5
>>> b
        array([ 0.,  5.,  2.,  3.])
>>> x=np.linalg.solve(a,b)
>>> np.dot(a,x)
        array([ -2.,   2.,   6.,  10.])
```

• Déterminant

Syntaxe de l'appel : `numpy.linalg.det(matrice A)`
Valeur de retour : le déterminant de A, de type `float`

Exemple :

```
>>> from numpy.linalg import det
>>> det([[1,2],[3,4]])
        -2.0000000000000004
```

• Trace

Syntaxe de l'appel : `numpy.trace(matrice A)`
Valeur de retour : la trace de A

Exemple :

```
>>> np.trace([[1,2],[3,4]])
        5
```

• Puissance d'une matrice

Syntaxe de l'appel : `numpy.linalg.matrix_power(matrice A, entier n)`
Valeur de retour : A^n

Exemple :

```
>>> from numpy.linalg import matrix_power
>>> matrix_power([[1,2],[3,4]], -2)
        array([[ 5.5 , -2.5 ],
        [-3.75, 1.75]])
```

• Transposée d'une matrice

Syntaxe de l'appel : `numpy.transpose(matrice A)`
Valeur de retour : ${}^tA$

Exemple :

```
>>> np.transpose([[1,2],[3,4]])
        array([[1, 3],
               [2, 4]])
```

Autre possibilité :

```
>>> A = np.array([[1,2],[3,4]])
>>> A.T
        array([[1, 3],
               [2, 4]])
```

• Outils de deuxième année

`numpy.linalg.eigvals` permet d'obtenir les valeurs propres d'une matrice.

`numpy.linalg.eig` permet d'obtenir les valeurs propres et les vecteurs propres d'une matrice.

Exemple :

```
>>> import nupy as np
>>> a=np.array([2,4,6,8],float).reshape(2,2)
>>> a
        array([[ 2.,  4.],
               [ 6.,  8.]])
>>> np.linalg.eigvals(a)
        array([ -0.74456265,  10.74456265])
>>> np.linalg.eig(a)
        (array([ -0.74456265,  10.74456265]), array([[-0.82456484, -0.41597356],
              [ 0.56576746, -0.90937671]]))
```

Le premier argument représente les valeurs propres, tandis que le deuxième retourne la matrice des vecteurs propres en colonne (une colonne est un vecteur propre).
`numpy.poly(A)` donne le polynôme caractéristique de la matrice `A`.

```
>>> import numpy as np
>>> M=np.array([1,2,3,4]).reshape(2,2)
>>> M
        array([[1, 2],
               [3, 4]])
>>> np.poly(M)
        array([ 1., -5., -2.])
```

c'est-à-dire le polynôme $\chi_M(X) = X^2 - 5X - 2$.

3.3.3 Tableaux aléatoires

le sous-module `numpy.random` définit un certain nombre de fonctions de génération aléatoire de nombres, suivant la plupart des lois classiques. Nous n'en citons que la loi uniforme, utiliser l'aide `help(numpy.random)` pour découvrir les autres en cas de besoin. On suppose `numpy.random` importé sous le nom `npr` :

```
npr.randint(a)    # génère aléatoirement un entier de 0 à a-1 inclus,
                  # de façon uniforme
npr.randint(a,b)   # génère aléatoirement un entier de a à b-1 inclus,
                   # de façon uniforme
npr.random(a,b,n)    # génère un tableau 1 x n d'entiers aléatoirement
                     # choisis uniformément entre a et b-1
npr.random()    # génère aléatoirement un réel de [0,1[ de façon uniforme
npr.sample(n)    # génère un array 1 ligne n colonnes rempli aléatoirement
                 # par des réels de [0,1[ choisis uniformément
npr.sample(tuple)    # de même pour un tableau multidimentionnel de
                     # taille définie par le tuple
```

La fonction `binomial` permet de simuler une variable aléatoire suivant une loi binomiale de paramètres n et p. Elle permet donc également de simuler une variable aléatoire suivant une loi de Bernoulli de paramètres p en prenant simplement $n = 1$. Cette fonction prend un troisième paramètre optionnel qui correspond, comme pour les fonctions précédentes, au nombre de valeurs à obtenir.

```
>>> npr.binomial(10, 0.3, 7)
        array([2, 2, 2, 2, 2, 4, 3])
>>> npr.binomial(1, 0.6, 20)
        array([0, 1, 1, 1, 1, 0, 1, 1, 0, 0, 0, 1,
               1, 1, 1, 0, 1, 1, 1, 1])
```

Les fonctions `geometric` et `poisson` fonctionnement de la même manière pour les lois géométrique ou de Poisson.

```
>>> npr.geometric(0.5, 8)
        array([1, 1, 3, 1, 3, 2, 5, 1])
>>> npr.poisson(4, 15)
        array([5, 2, 3, 4, 6, 0, 5, 3,
               1, 5, 1, 5, 9, 4, 6])
```

3.3.4 Polynômes

Le sous-module `numpy.polynomial.polynomial` permet de travailler avec des polynômes :

```
>>> from numpy.polynomial import Polynomial
```

Pur créer un polynôme, il faut lister ses coefficients par ordre de degré croissant. Par exemple, pour le polynôme $X^3 + 2X - 3$:

```
>>> p = Polynomial([-3, 2, 0, 1])
```

On peut alors utiliser cette variable comme une fonction pour calculer, en un point quelconque, la valeur de la fonction polynôme associée. Cette fonction peut agir également sur un tableau de valeurs, elle calcule alors la valeur de la fonction polynôme en chacun des points indiqués.

```
>>> p(0)
       -3.0
>>> p([1, 2, 3])
        array([ 0., 9., 30.])
```

L'attribut `coef` donne accès aux coefficients ordonnés par degré croissant ; ainsi `p.coef[i]` correspond au coefficient du terme de degré i. La méthode `degree` renvoie le degré du polynôme alors que `roots` calcule ses racines.

```
>>> p.coef
        array([-3., 2., 0., 1.])
>>> p.coef[1]
        2.0
>>> p.degree()
        3
>>> p.roots()
        array([-0.5-1.6583124j, -0.5+1.6583124j,
              1.0+0.j])
```

La méthode `deriv` renvoie un nouveau polynôme, dérivé du polynôme initial. Cette méthode prend en argument facultatif un entier positif indiquant le nombre de dérivations à effectuer. De la même manière la méthode `integ` intègre le polynôme, elle prend un paramètre optionnel supplémentaire donnant la constante d'intégration à utiliser, ce paramètres peut être une liste en cas d'intégration multiple ; les constantes d'intégration non précisées sont prises égales à zéro.

```
>>> p.deriv().coef
        array([ 2., 0., 3.])
>>> p.deriv(2).coef
        array([ 0., 6.])
>>> p.deriv(5).coef
        array([-0.])
>>> p.integ().coef
        array([ 0. , -3. , 1. , 0. , 0.25])
>>> p.integ(1, 2).coef    # intégrer une fois avec la constante 2
        array([ 2. , -3. , 1. , 0. , 0.25])
>>> p.integ(2, [1, 2]).coef     # intégrer deux fois
        array([ 2., 1., -1.5, 0.33333333, 0.0.05])
```

Les opérateurs `+`, `-`, `*` permettent d'additionner, soustraire et multiplier des polynômes. Ils fonctionnent égale- ment entre un polynôme et un scalaire. L'opérateur `**` permet d'élever un polynôme à une puissance entière positive.

```
>>> a = Polynomial([1, 2, 1])
>>> b = Polynomial([5, 3])
>>> p = 2*a * b + Polynomial([-7, 2])
>>> p.coef
        array([ 3., 28., 22., 6.])
>>> (p**2).coef
        array([9., 168., 916., 1268., 820., 264., 36.])
```

L'opérateur `/` permet de diviser un polynôme par un scalaire. Pour diviser deux polynômes il faut utiliser l'opérateur `//` qui renvoie le quotient ; l'opérateur `%` calcule le reste.

```
>>> (p / 2).coef
        array([ 1.5, 14. , 11. ,3. ])
>>> q = p // a
>>> r = p % a
```

```
>>> q.coef
        array([ 10., 6.])
>>> r.coef
        array([-7., 2.])
>>> (q * a + r).coef
        array([ 3., 28., 22., 6.])
```

3.4 Random

Pour créer un nombre aléatoire, on peut aussi utiliser le module `random`. Exemple :

```
>>> import random
>>> random.randint(10,20)   #un entier >=10 et <=20
        19
>>> a = random.uniform(10,20)   # un flottant entre 10 et 20 (20 est inclu ou non)
>>> a
        14.549546974966514
>>> round(a,4)   #arrondir a la 4eme decimale
        14.5495
>>> random.choice(['beau','laid','waow'])    #choisir dans une liste
        'laid'
```

3.5 Matplotlib (tracé de courbes)

Ce module donne un certain nombre d'outils graphiques, notamment pour le tracé de courbes. Nous utiliserons plus souvent le sous-module `matplotlib.pyplot`, que nous supposerons importé sous l'alias `plt`.

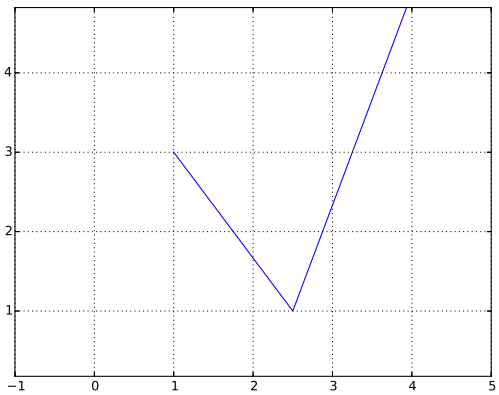
```
plt.plot(L1,L2, label='nom')    # trace la ligne brisée entre les points
                                # dont les abscisses sont dans la liste L1
                                # et les ordonnées dans la liste L2
                                # Le label sert à identifier la courbe
plt.title("titre")    # Définit le titre du graphe (affiché en haut)
plt.grid()     # Affiche un grille en pointillés pour
               # une meilleure lisibilité
plt.legend(loc = ...)    # Affiche une légende (associant à chaque
                         # courbe son label)
                         # loc peut être un nombre (1,2,3,4,5,...)
                         # plaçant la légende à un endroit précis
                         # loc peut aussi être plus explicitement:
                         # 'upper left' etc.

plt.savefigure('nom.ext')   # Sauvegarde le graphe dans un fichier de ce nom
                            # L'extention définit le format
                            # Voir l'aide pour savoir les formats acceptés
plt.show()    # Affichage à l'écran de la figure
plt.imshow(T)   # Affiche une image constituée de points dont
```

```
# les couleurs sont définies par les valeurs du
# tableau T de dimension 3 (à voir comme un
# tableau de pixels RGB)
```

• Tracés de lignes brisées et options de tracés

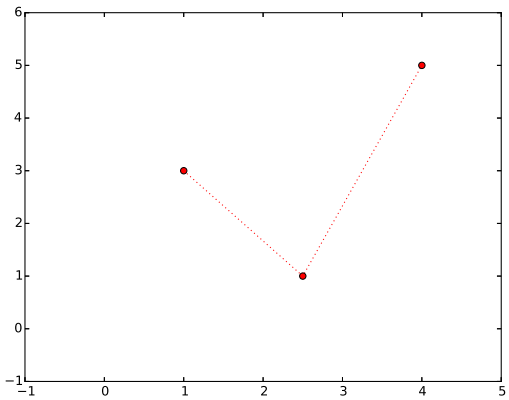
Pour tracer une ligne brisée, on donne la liste des abscisses et la liste des ordonnées puis on effectue le tracé. La fonction `axis` permet de définir la fenêtre dans laquelle est contenue le graphique. L'option equal permet d'obtenir les mêmes échelles sur les deux axes. Les tracés relatifs à divers emplois de la fonction plot se superposent. La fonction `plt.clf()` efface les tracés contenus dans la fenêtre graphique.

```
1 x = [1., 2.5, 4.]
2 y = [3., 1., 5.]
3 plt.axis('equal')
4 plt.plot(x, y)
5 plt.axis([-1., 5., -1., 6.])
6 plt.grid()
7 plt.show()
```



La fonction `plot` admet de nombreuses options de présentation. Le paramètre `color` permet de choisir la couleur (`'g'` : vert, `'r'` : rouge, `'b'` : bleu). Pour définir le style de la ligne, on utilise `linestyle` (`'-'` : ligne continue, `'--'` : ligne discontinue, `':'` : ligne pointillée). Si on veut marquer les points des listes, on utilise le paramètre `marker` (`'+'`, `'.'`, `'o'`, `'v'` donnent différents symboles).

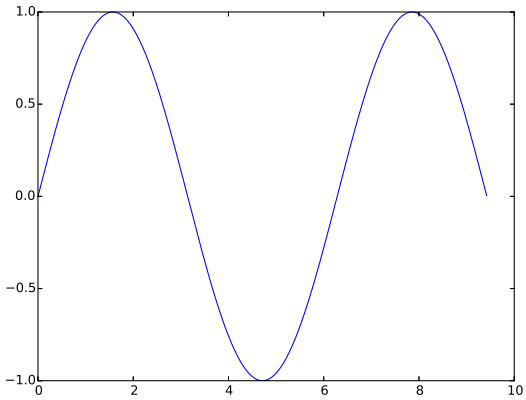
```
1 x = [1., 2.5, 4.]
2 y = [3., 1., 5.]
3 plt.axis([-1., 5., -1., 6.])
4 plt.plot(x, y, color='r',
     linestyle=':', marker='o')
5 plt.show()
```



• Tracés de fonction

Pour tracer la courbe représentative d'une fonction, on définit une liste d'abscisses puis on construit la liste des ordonnées correspondantes. L'exemple ci-dessous trace la courbe de $x \longmapsto \sin x$ sur $[0, 3\pi]$.

```
import math
import numpy as np

def f(x):
    return math.sin(x)

X = np.arange(0, 3*np.pi,
   0.01)
Y = [ f(x) for x in X ]
plt.plot(X, Y)
plt.show()
```



Une autre solution consiste à utiliser la fonction `vectorize` du module `numpy` qui permet de transformer une fonction scalaire en une fonction capable de travailler avec des tableaux. Il est cependant beaucoup plus efficace d'utiliser directement des fonctions universelles.

```
def f(x):
    return math.sin(x)

f = np.vectorize(f)
```

Il est à noter que les opérateurs python (`+`, `-`, `*`, etc.) peuvent s'appliquer à des tableaux, ils agissent alors terme à terme. Ainsi la fonction `f` définie ci-dessous est une fonction universelle, elle peut travailler aussi bien avec deux scalaires qu'avec deux tableaux et même avec un scalaire et un tableau.

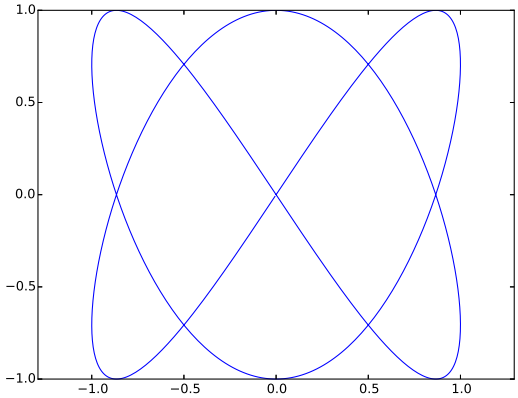
```
def f(x, y) :
    return np.sqrt(x**2 + y**2)
```

```
>>> f(3, 4)
        5.0
>>> f(np.array([1, 2, 3]), np.array([4, 5, 6]))
        array([ 4.12310563, 5.38516481, 6.70820393])
>>> f(np.array([1, 2, 3]), 4)
        array([ 4.12310563, 4.47213595, 5.])
```

• Tracés d'arcs paramétrés

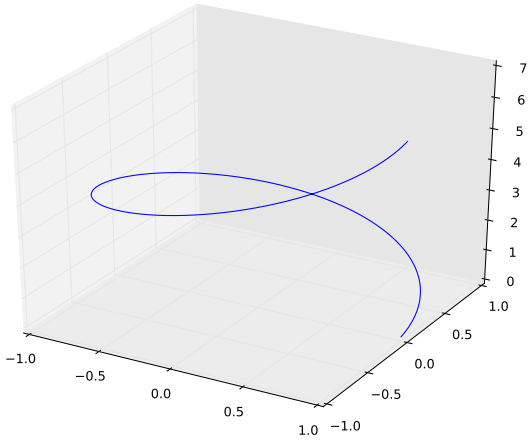
Dans le cas d'un arc paramétré plan, on définit d'abord la liste des valeurs données au paramètre puis on construit la liste des abscisses et des ordonnées correspondantes. On effectue ensuite le tracé.

```python
import numpy as np

def x(t) :
    return np.sin(2*t)

def y(t) :
    return np.sin(3*t)

T = np.arange(0, 2*np.pi, 0.01)
X = x(T)
Y = y(T)
plt.axis('equal')
plt.plot(X, Y)
plt.show()
```



Voici un exemple de tracé d'un arc paramétré de l'espace.

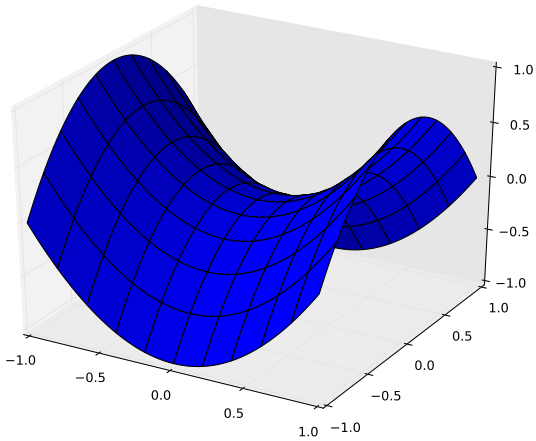
```python
from mpl_toolkits.mplot3d import Axes3D

ax = Axes3D(plt.figure())
T = np.arange(0, 2*np.pi, 0.01)
X = np.cos(T)
Y = np.sin(T)
Z = T
ax.plot(X, Y, T)
plt.show()
```



- <u>Tracé de surfaces</u>

Pour tracer une surface d'équation $z = f(x, y)$, on réalise d'abord une grille en (x, y) puis on calcule les valeurs de z correspondant aux points de cette grille. On fait ensuite le tracé avec la fonction `plot_surface`.

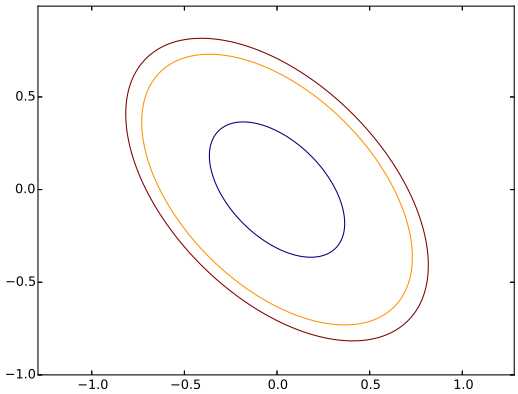
```python
import numpy as np
from mpl_toolkits.mplot3d import Axes3D

ax = Axes3D(plt.figure())

def f(x,y) :
    return x**2 - y**2

X = np.arange(-1, 1, 0.02)
Y = np.arange(-1, 1, 0.02)
X, Y = np.meshgrid(X, Y)
Z = f(X, Y)
ax.plot_surface(X, Y, Z)
plt.show()
```



• Tracé de lignes de niveau

Pour tracer des courbes d'équation $f(x,y) = k$, on fait une grille en x et en y sur laquelle on calcule les valeurs de f. On emploie ensuite la fonction `contour` en mettant dans une liste les valeurs de k pour lesquelles on veut tracer la courbe d'équation $f(x,y) = k$.

```
import numpy as np

def f(x,y):
    return x**2 + y**2 + x*y

X = np.arange(-1, 1, 0.01)
Y = np.arange(-1, 1, 0.01)
X, Y = np.meshgrid(X, Y)
Z = f(X, Y)
plt.axis('equal')
plt.contour(X, Y, Z,
    [0.1,0.4,0.5])
plt.show()
```



3.6 Scipy (calcul scientifique)

Le module `scipy` regroupe un certain nombre de fonctions de calcul scientifique (algorithmes classiques de calcul numérique).

Voici les principales fonctions.

```
scipy.integrate.quad(f,a,b)     # intégrale de f de a à b
scipy.optimize.newton(f,a,g)     # résolution d'équation par la méthode de Newton
                                 # initialisée au point a
                                 # g doit être égal à f'
scipy.integrate.odeint(f,y0,T)     # Résolution de l'ED y'=f(y,t)
                                   # y0 est la valeur initial (t minimal)
                                   # T est un tableau des temps auquels
                                   # calculer les valeurs de y
```

• Résolution approchée d'équations

Pour résoudre une équation du type $f(x) = 0$ où f est une fonction d'une variable réelle, on peut utiliser la fonction `fsolve` du module `scipy.optimize`. Il faut préciser la valeur initiale x_0 de l'algorithme employé par la fonction `fsolve`. Le résultat peut dépendre de cette condition initiale.

```
import scipy.optimize as resol

def f(x) :
    return x**2 - 2
```

```
>>> resol.fsolve(f, -2.)
          array([-1.41421356])
>>> resol.fsolve(f, 2.)
          array([ 1.41421356])
```

Dans le cas d'une fonction `f` à valeurs vectorielles, on utilise la fonction `root`. Par exemple, pour résoudre le système non linéaire $\begin{cases} x^2 - y^2 = 1 \\ x + 2y - 3 = 0 \end{cases}$

```
def f(v):
    return v[0]**2 - v[1]**2 - 1, v[0] + 2*v[1] - 3
```

```
>>> sol = resol.root(f, [0,0])
>>> sol.success
        True
>>> sol.x
        array([ 1.30940108, 0.84529946])
>>> sol=resol.root(f, [-5,5])
>>> sol.success
        True
>>> sol.x
        array([-3.30940108, 3.15470054])
```

• Calcul approché d'intégrales

La fonction `quad` du module `scipy.integrate` permet de calculer des valeurs approchées d'intégrales. Elle renvoie une valeur approchée de l'intégrale ainsi qu'un majorant de l'erreur commise. Cette fonction peut aussi s'employer avec des bornes d'intégration égales à $+\infty$ ou $-\infty$.

```
import as np
import scipy.integrate as integr

def f(x) :
    return np.exp(-x)
```

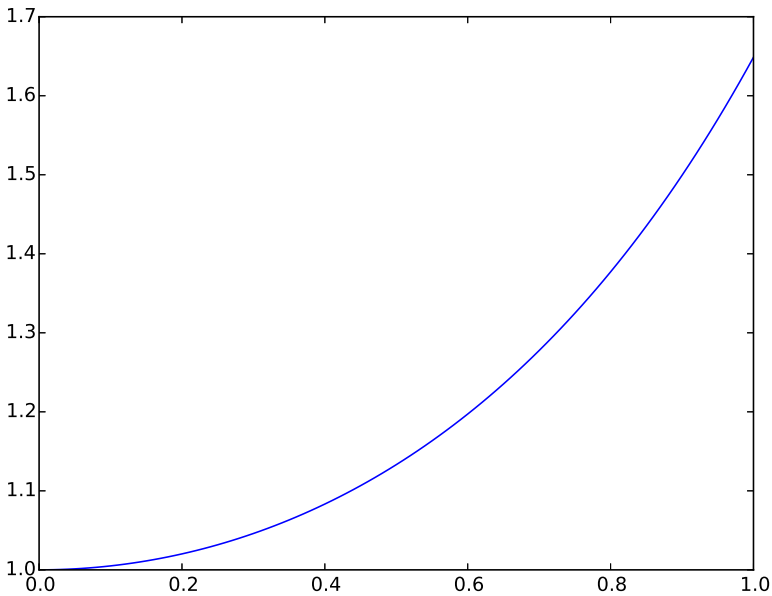
```
>>> integr.quad(f, 0, 1)
        (0.6321205588285578, 7.017947987503856e-15)
>>> integr.quad(f, 0, np.inf)
        (1.0000000000000002, 5.842607038578007e-11)
```

• Résolution approchées d'équations différentielles

Pour résoudre une équation différentielle $x' = f(x,t)$, on peut utiliser la fonction `odeint` du module `scipy.integrate`. Cette fonction nécessite une liste de valeurs de `t`, commençant en t_0 , et une condition initiale x_0. La fonction renvoie des valeurs approchées (aux points contenus dans la liste des valeurs de `t`) de la solution `x` de l'équation différentielle qui vérifie $x(t_0) = x_0$. Pour trouver des valeurs approchées sur $[0,1]$ de la solution $x'(t) = tx(t)$ qui vérifie $x(0) = 1$, on peut employer le code suivant.

```
import numpy as np
import scipy.integrate as integr
import matplotlib.pyplot as plt

def f(x, t) :
    return t*x
```

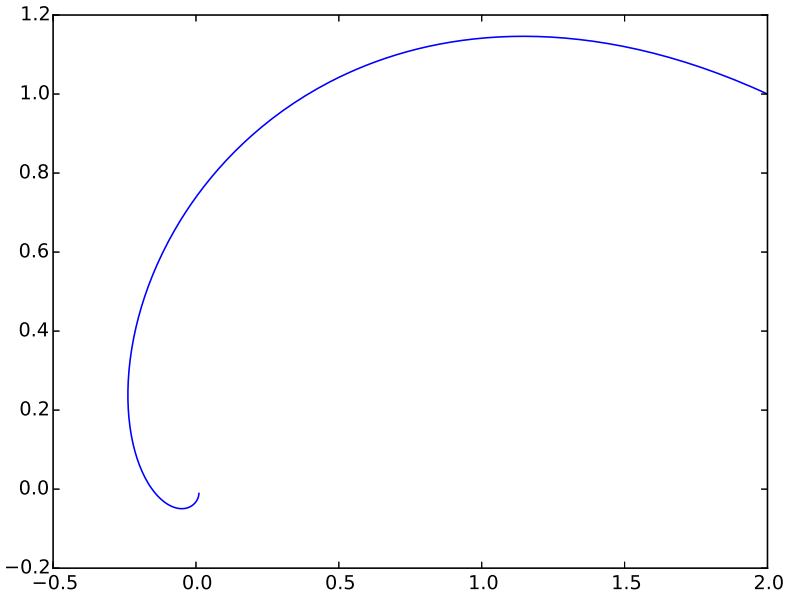
```
T = np.arange(0, 1.01, 0.01)
X = integr.odeint(f, 1, T)
X[0]
array([ 1.])
X[-1]
array([ 1.64872143])
plt.plot(T,X)
plt.show()
```



Si on veut résoudre, sur $[0,5]$, le système différentiel $\begin{cases} x'(t) = -x(t) - y(t) \\ y'(t) = x(t) - y(t) \end{cases}$ avec la condition initiale $x(0) = 2$, $y(0) = 1$ le code devient le suivant.

```
def f(x, t):
    return np.array([-x[0]-x[1], x[0]-x[1]])
```

```
>>> T = np.arange(0, 5.01, 0.01)
>>> X = integr.odeint(f, np.array([2.,1.]), T)
>>> X[0]
        array([ 2., 1.])
>>> plt.plot(X[ :,0], X[ :,1])
>>> plt.show()
```



Pour résoudre une équation différentielle scalaire d'ordre 2 de solution x, on demandera la résolution du système différentiel d'ordre 1 satisfait par $X(t) = \begin{pmatrix} x(t) \\ x'(t) \end{pmatrix}$.

3.7 Autres modules (time, MySQLdb, sqlite3,...)

Le module `time` permet d'accéder à l'horloge. On s'en servira essentiellement pour mesurer le temps d'exécution d'une fonction :

```
time.time()     # donne une valeur d'horloge correspondant à
                # l'instant en cours en seconde.
```

Cette fonction sera utilisée comme suit :

```
from time import time
debut = time()
instructions
temps_execution = time() - debut
```

Le module `MySQLdb` permet d'effectuer des requêtes dans une base de données au format `SQL`. La syntaxe est la suivante :

```
import MySQLdb as sql
Base = sql.connect(host='***',user='***',passwd='***',db='geographie')
         # connexion au serveur MySQL et à la base de données
requete='SELECT longitude, latitude, zmax FROM \
           communes WHERE num_departement <= 95'
Base.query(requete)  # exécution de la requête
resultat = Base.use_result()
res = resultat.fetch_row(maxrows=0)  # enregistrement du résultat
Base.close()   # fermeture de la connexion
```

Le résultat de la requête est contenu dans la variable `res` qui est un tuple de tuples : chaque ligne de réponse de la requête est donnée par un tuple ; un tuple contient toutes les réponses. On peut parcourir les réponses avec une boucle for :

```
for ligne in res:
print res
```

Le module `sqlite3` permet lui d'effectuer des requêtes dans une base de données au format `SQLite3`. La syntaxe est la suivante :

```
import sqlite3 as lite
connection = lite.connect('base.db')
           # crée un objet de connexion vers la base base.db
cur = connection.cursor()
           # crée un curseur qui permettra des modifications de la base
cur.execute("requête SQL")
            # Exécute l'instruction donnée, dans la syntaxe SQL,
            # dans la base pointée par le curseur
res=cur.fetchall()   # enregistre le résultat
```

Le résultat de la requête est contenu dans la variable `res` qui est une liste de tuples : chaque ligne de réponse de la requête est donnée par un tuple ; une liste contient toutes les réponses. On peut parcourir les réponses avec une boucle for :

```
for ligne in res:
print res
```

4 Algorithmes de première année

4.1 Recherche d'un élément dans un tableau/liste.

Pour rechercher un élément `x` dans une liste `t`, l'idée est de parcourir `t` de gauche à droite, jusqu'à obtenir l'élément `x`. En sortie on donne le *premier indice* `i` tel que `t[i]=x`, s'il existe, et sinon la fonction renvoie `False`.

```
1 def recherche(t,x):
2     n=len(t)
3     i=0
4     while i<n and t[i]!=x: # évaluation paresseuse de and
5         i+=1
6     if i<n:
7         return i
8     else:
9         print False
```

Si on s'autorise les sorties de boucle avec l'instruction `return`, on peut utiliser une boucle `for` :

```
def recherche(t,x):
    n=len(t)
    for i in range(n):
        if t[i]==x:
            return i # on arrête l'algorithme
    return False # cette ligne n'est atteinte que si x n'
        appartient pas à t
```

Pour évaluer les performances de cet algorithme, notons c_n le nombre de tests effectués pour une liste `t` de taille n. Il est clair que $c_n = 1$ dans le meilleur des cas (`t[0]=x`), et que $c_n = n$ dans le pire des cas (`x` est le dernier élément).

4.2 Recherche naïve d'un mot dans une chaîne de caractères

L'idée est la même que dans l'algorithme précédent : on parcourt la chaîne `chaine` de gauche à droite, en testant à chaque fois si le mot `mot` est trouvé. Si c'est le cas, on renvoie l'indice du première caractère de `mot` dans `chaine`, et `False` sinon.

```
def recherche2(chaine,mot):
    n=len(chaine)
    p=len(mot)
    if p>n:
        return False  # mot trop long pour chaine
    else:
        i=0
        while i+p<n+1 and chaine[i:i+p]!=mot: # évaluation
            paresseuse de and
            i+=1
        if i+p==n+1:
            return False
        else:
            return i
```

Avec une boucle for qu'on peut quitter avec `return` :

```
def recherche2(chaine,mot):
    n=len(chaine)
    p=len(mot)
    if p>n:
        return False  # mot trop long pour chaine
    else:
        for i in range(n-p+1):
            if chaine[i:i+p]==mot:
                return i # provoque une sortie de boucle
        else:
            return False
```

4.3 Recherche du maximum dans un tableau/liste de nombres.

On parcourt une fois la liste `tableau` de gauche à droite, en mémorisant le plus grand élément rencontré dans une variable auxiliaire `maxi`. À la fin, la variable `maxi` est égale au plus grand élément de la liste.

```
def maximum(t):
    maxi=t[0]
    n=len(t)
    for k in range(n):
        if t[k]>maxi: # le k-ième plus grand élément est plus grand que les k-1 premiers
            maxi=t[k] # il devient le nouveau maximum
    return maxi
```

Pour évaluer les performances de cet algorithme, notons c_n le nombre de tests effectués pour une liste `t` de taille n. Il est clair que $c_n = n$ dans tous les cas.

4.4 Calcul de la moyenne, de la variance des valeurs d'un tableau/liste de nombres

On rappelle que si $\mathtt{t} = [\mathtt{t}_0, \ldots, \mathtt{t}_{n-1}]$ est un tableau de nombres, sa *moyenne* est définie par :

$$m = \frac{1}{n}\sum_{k=0}^{n-1} t_k$$

et sa *variance* par :

$$v = \frac{1}{n}\sum_{k=0}^{n-1} (t_k - m)^2$$

On en déduit les programmes :

```
def moyenne(t):
    m=0
    n=len(t)
    for k in range(n):
        m+=t[k]
    return m/float(n) # Python 2.7
```

et :

```
def variance(t):
    v=0
    m=moyenne(t)
    n=len(t)
    for k in range(n):
        v+=(t[k]-m)**2
    return v/float(n) # Python 2.7
```

4.5 Recherche par dichotomie dans un tableau/liste trié

Nous avons déjà donné un algorithme de recherche d'un élément dans une liste, de complexité dans le pire des cas $c_n = \mathcal{O}(n)$. Si nous disposons d'une liste `t` triée dans l'ordre croissant, nous pouvons accélérer la recherche en procédant par *dichotomie* : on divise le tableau en deux, on identifie dans quelle partie se trouve l'élément cherché, puis on recommence à diviser ce sous-tableau en deux et etc...

```
def recherche_dicho(t,x):
    n=len(t)
```

```
    a=0
    b=n-1
    if t[a]>x or t[b]<x: # cas triviaux où x n'est pas dans t
        return False
    else:
        while b-a>0:    # on cherche dans t[a:b]
            c=(a+b)/2 # Python 2.7
            if t[c]==x: # x est au milieu
                return c # on a trouvé x !!
            elif t[c]>x: # x est dans la partie gauche
                b=c-1
            else:    # x est dans la partie droite
                a=c+1
```

On peut montrer qu'après k passage dans la boucle, on a $b-a \leq n/2^k$. On en déduit que dans le pire des cas, la complexité (en ne comptant que les comparaisons) est $c_n = \mathcal{O}\big(\ln(n)\big)$. On a donc nettement amélioré les performances de l'algorithme ! Mais il ne faut as perdre de vue que cette méthode ne fonctionne pas si les données ne sont pas préalablement triées...

4.6 Recherche par dichotomie d'un zéro d'une fonction continue

On souhaite trouver $x \in [a,b]$ tel que $f(x) = 0$, pour une fonction f continue sur $[a,b]$.

Le principe de la *dichotomie* est d'encadrer de plus en plus finement un zéro possible, sa présence étant détectée par un changement de signe de la fonction, supposée continue (l'existence du zéro étant alors assuré par le théorème des valeurs intermédiaires) :

- On initialise l'encadrement par deux valeurs $a_0 < b_0$ telles que f change de signe entre a_0 et b_0, c'est-à-dire $f(a_0)f(b_0) \leq 0$.
- On coupe l'intervalle en deux en son milieu. Puisqu'il y a changement de signe sur l'intervalle, il y a changement de signe sur l'une des deux moitiés. On conserve la moitié correspondante.
- On continue de la sorte en coupant à chauqe étape l'intervalle en deux, en gardant la moitié sur laquelle il y a un changement de signe. On continue jusqu'à ce que l'intervalle obtenu soit de longueur suffisamment petite pour donner une valeur approchée du zéro à la marge d'erreur souhaitée.

On en déduit le programme python où les arguments d'entrée sont la fonction `f`, l'intervalle de recherche `[a,b]`, et la précision `eps` :

```
def dichotomie(f,a,b,eps):
    assert f(a)*f(b)<=0
    while b-a>eps:
        c=(a+b)/2.   # Python 2.7
        if f(a)*f(c)<=0:
            b=c
        else:
            a=c
    return b
```

Par exemple on peut en déduire une approximation de $\sqrt{2}$ à la précision 10^{-3} :

```
>>> dichotomie(lambda x:x**2-2,0,2,0.001)
        1.4150390625
```

Les points forts de cet algorithme sont : sa simplicité, sa convergence assurée, et sa vitesse raisonnable. Ses points faibles sont : la nécessité d'un encadrement préalable du zéro, et le fait que s'il y a plusieurs zéros dans l'intervalle on n'en obtient qu'un. De plus l'algorithme fait de nombreux appels à `f` ce qui peut ralentir les calculs si la fonction est complexe.

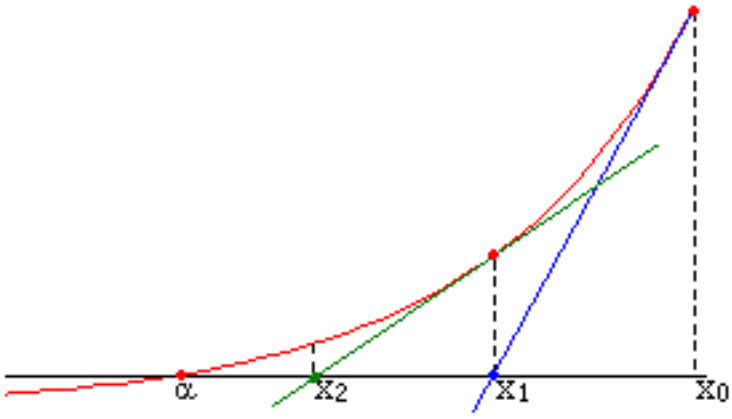
4.7 Méthode de Newton

La méthode de Newton bien plus rapide. On suppose f dérivable :

- On part d'une valeur initiale x_0.
- On construit x_1 comme l'intersection de la tangente en x_0 et de l'axe des abscisses.
- On itère cette construction.

On a alors, pour tout $n \in \mathbb{N}$:

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)}$$



On arrête l'algorithme dès que la différence entre deux termes consécutifs est assez petite.

```
1 def newton(f, fprime, x0, eps):
2     u=x0
3     v=u-f(u)/fprime(u)
4     while abs(v-u)>eps:
5         u, v = v, v-f(v)/fprime(v)
6     return v
```

Par exemple :

```
>>> newton(lambda x:x**2-2,lambda x:2*x,1.,0.0001)
        1.4142135623746899
```

⚠ La convergence est extrêmement rapide mais cet algorithme est un cauchemar pour l'informaticien :

- On peut rencontrer des divisions par zéro.
- La terminaison n'est pas assurée.
- Même si un résultat est renvoyé, il peut être éloigné d'un zéro de f.

4.8 Calcul de valeur approchées d'intégrales sur un segment par la méthode des rectangles ou des trapèzes

Le but est de calculer une valeur approchée de l'intégrale $\int_a^b f(x)\,\mathrm{d}x$ d'une fonction f continue sur un segment $[a,b]$.

On peut utiliser la *méthode des rectangles* : on calcule une valeur approchée de l'intégrale en réalisant une somme de surfaces de rectangles. Le domaine d'intégration est découpé en intervalles $[x_k, x_{k+1}]$ et on fait comme si la fonction restait constante sur chaque intervalle.

Sur chaque intervalle, on réalise ainsi l'approximation suivante :

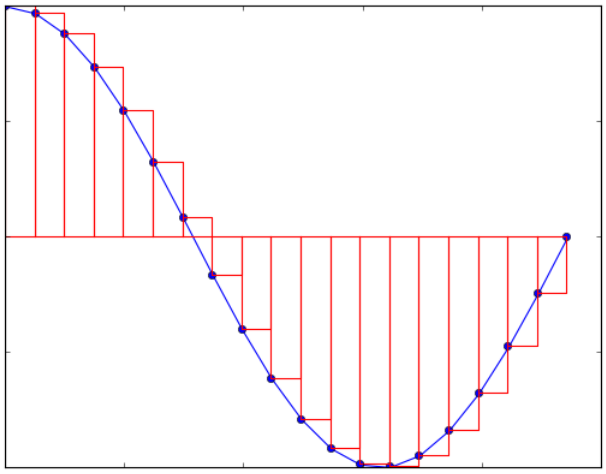
$$\int_{x_k}^{x_{k+1}} f(t)\,\mathrm{d}t \approx (x_{k+1} - x_k) f(x_k)$$

donc si la subdivision choisie est $x_0 = a < x_1 < \cdots < x_n = b$ alors $\int_a^b f(x)\,\mathrm{d}x$ est approximée par :

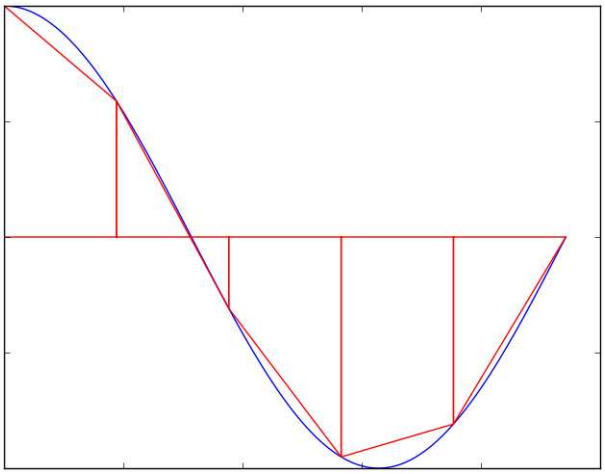
$$\sum_{k=0}^{n-1} (x_{k+1} - x_k) f(x_k)$$

Dans le cas d'une subdivision régulière, on a $x_k = a + k\dfrac{b-a}{n}$ et l'approximation devient :

$$\frac{b-a}{n} \sum_{k=0}^{n-1} f\left(a + k\frac{b-a}{n}\right)$$



La *méthode des trapèzes*, comme son nom l'indique, consiste à remplacer les rectangles par des trapèzes.



L'approximation est alors la suivante :

$$\int_a^b f(x)\,\mathrm{d}x \approx \sum_{k=0}^{n-1}(x_{k+1}-x_k)\frac{f(x_k)+f(x_{k+1})}{2}$$

et donc dans le cas d'une subdivision régulière :

$$\frac{b-a}{2n}\sum_{k=0}^{n-1}\left[f\left(a+k\frac{b-a}{n}\right)+f\left(a+(k+1)\frac{b-a}{n}\right)\right]$$

Pour éviter de nombreux appels à f on peut arranger la formule sous cette forme :

$$\frac{b-a}{n}\left[\frac{f(a)+f(b)}{2}+\sum_{k=1}^{n-1}f\left(a+k\frac{b-a}{n}\right)\right]$$

On en déduit les programmes Python :

```
1 def rectangles(f,a,b,n):
2     """ approximation de l'intégrale de f entre a et b avec n
          rectangles """
3     I=0
4     for k in range(n):
5         I += (b-a) * f( a+k*(b-a)/float(n) ) / float(n)    #
              Python 2.7
6     return I
```

et :

```
1 def trapezes(f,a,b,n):
2     """ approximation de l'intégrale de f entre a et b avec n
          trapèzes """
3     I=(b-a)*( f(a) + f(b) )/(2.*n)     # Python 2.7
4     for k in range(1,n):
5         I += (b-a) * f( a+k*(b-a)/float(n) ) / float(n)    #
              Python 2.7
6     return I
```

Intuitivement la précision sera meilleure pour de grandes valeurs de n. De plus c'est la méthode des trapèzes qui devrait converger le plus rapidement. On peut le vérifier sur un exemple :

```
>>> for n in range(20,201,20):
...     print rectangles(lambda x:x**2,0,3,n) , trapezes(lambda x:x**2,0,3,n)
...
            8.33625       9.01125
            8.6653125     9.0028125
            8.77625       9.00125
            8.831953125   9.000703125
            8.86545       9.00045
            8.8878125     9.0003125
            8.90380102041 9.00022959184
            8.91580078125 9.00017578125
            8.92513888889 9.00013888889
            8.9326125     9.0001125
```

4.9 Résolution d'une équation différentielle ordinaire : méthode ou schéma d'Euler

Étant donnée une équation différentielle écrite sous la forme :

$$\forall x \in I, \quad u'(x) = f(x, u(x))$$

et une initialisation $u(x_0) = y_0$, on souhaite calculer une approximation de la fonction u sur l'intervalle I.
La question initiale qui peut se poser est alors la façon de représenter la fonction solution u. Il serait vain de penser pouvoir en donner une expression par des fonctions usuelles, et une représentation purement numérique ne pourra être complète (puisqu'on ne peut renvoyer qu'un nombre fini de valeurs). On pourrait distinguer deux situations :

- On est intéressé par une description globale de toute la fonction. Dans ce cas, on peut choisir un pas p suffisamment petit, et calculer les valeurs de u à intervalles réguliers de pas p. On peut compléter ensuite par interpolation linéaire (ou représenter par un graphe, qui lui-même sera tracé par interpolation linéaire, par exemple sous Python)
- On n'est intéressé que par la valeur en un point x. Dans ce cas, on fait de même qu'au point précédent pour l'intervalle $[x_0, x]$, au lieu de l'intervalle global. On obtient alors une valeur approchée de $u(x)$ (sans avoir besoin de faire d'interpolation linéaire).

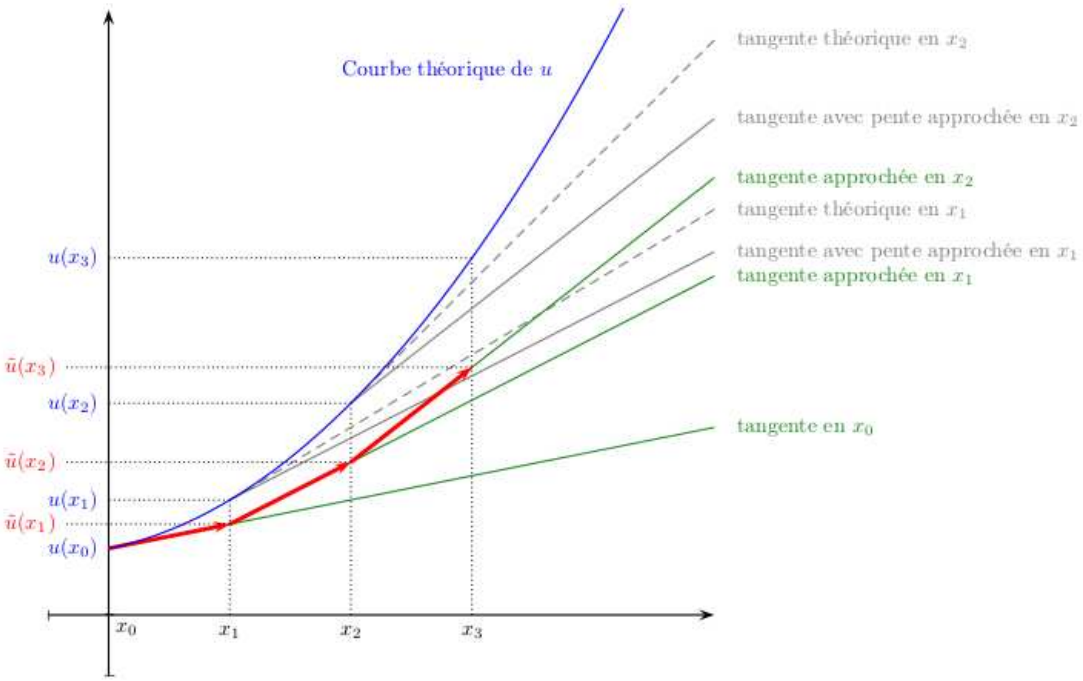
Dans les deux cas, l'idée est la même : elle consiste à progresser par petits pas, et à calculer à chaque étape une valeur approchée au pas suivant à partir des valeurs déjà obtenues.

Ainsi, partant d'un intervalle $[a, b]$, où $a = x_0$, on subdivise celui-ci régulièrement en posant les points intermédiaires $x_k = a + k\dfrac{b-a}{n}$, pour $k \in [\![0, n]\!]$. On cherche ensuite des approximations des $u(x_k)$.

Remarquer que le problème de la convergence vers la solution exacte se pose essentiellement pour les valeurs lointaines de x_0. En effet les approximations aux points successifs étant calculées à l'aide des approximations précédentes, les erreurs s'ajoutent de pas en pas : plus on s'éloigne de x_0 , plus l'approximation obtenue va être mauvaise.

Comme toujours en analyse numérique, les questions qui se posent sont les suivantes :

- Rechercher un algorithme donnant une erreur acceptable, en un temps raisonnable
- Contrôler précisément l'erreur
- Trouver un compris entre temps de calcul et précision.



La méthode d'Euler est basée sur une idée très simple :

- L'équation différentielle $u' = f(x,u)$ et la connaissance de $u(x_0)$ nous fournissent la connaissance de $u'(x_0)$
- On approche alors localement la courbe de u par sa tangente. Pour un pas suffisamment petit, on peut considérer que cette approximation est valable sur tout l'intervalle $[x_0, x_1]$.
- On en déduit alors une valeur approchée $\tilde{u}(x_1)$ de $u(x_1)$.
- On fait comme si cette valeur approchée était la valeur exacte : en utilisant l'équation différentielle, on en déduit une valeur approchée $\tilde{u}'(x-1)$ de $u'(x_1)$.
- On approche à nouveau la courbe de u par sa tangente, elle-même approchée par la droite calculée à l'aide des valeurs approchées calculées de $u(x_1)$ et $u'(x_1)$. C'est cette droite qui approche la tangente qu'on prend comme nouvelle approximation pour le calcul de $u(x_2)$.
- On continue de la sorte.

La construction est illustrée par la figure précédente. On se rend vite compte qu'on accumule les erreurs et que pour des valeurs éloignées de x_0, l'approximation risque de ne pas être très bonne.

Conformément aux explications précédentes, la méthode d'Euler consiste donc à approcher $u(x_k)$ par la valeur $u_k = \tilde{u}(x_k)$, calculée par la récurrence suivante :

$$u_0 = y_0 \qquad \text{et} \qquad \forall k \in [\![0, n-1]\!],\ u_{k+1} = f(x_k, u_k) \times (x_{k+1} - x_k) + u_k$$

En notant $h = \dfrac{b-a}{n}$ le pas constant de la subdivision, on peut réécrire la relation de récurrence :

$$\forall k \in [\![0, n-1]\!], \quad u_{k+1} = h.f(x_k, u_k) + u_k$$

Le code Python est le suivant :

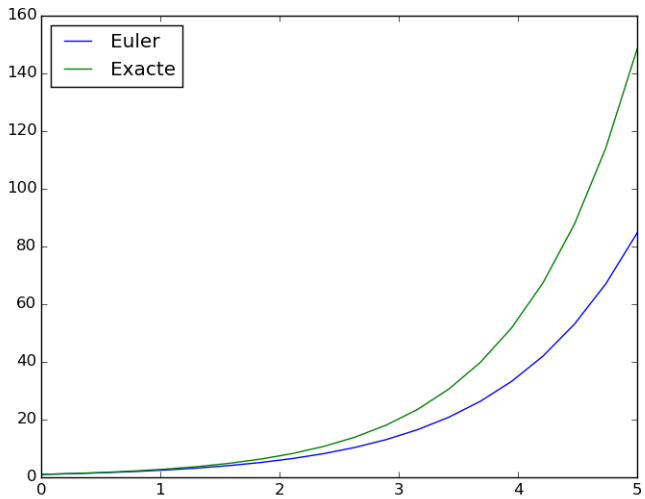
```
def Euler(f,a,b,y0,n):
    """ résolution approchée de y'=f(t,y) sur [a,b] avec y(a)=y0
        """
    y=[0]*n      # initialisation du vecteur y
    t=[ a+k*float(b-a)/(n-1) for k in range(n)]   # initialisation
        des piquets de temps
    y[0]=y0    # condition initiale
    for k in range(1,n):
        y[k]=y[k-1]+(b-a)/float(n-1)*f(t[k-1],y[k-1])    # Python
            2.7
    return t,y
```

En sortie on récupère le vecteur des piquets de temps, et le vecteur des valeurs approchées de y.

Par exemple appliquons cette méthode pour le problème de Cauchy $\begin{cases} y' = y \\ y(0) = 1 \end{cases}$ sur l'intervalle $[0,5]$, et comparons la solution approchée et la solution exacte.

```
>>> [t,y]=EulerTP(lambda t,y:y , 0, 5, 1, 20)
>>> from math import exp
>>> z=[ exp(x) for x in t ]
>>> import matplotlib.pyplot as plt
>>> plt.plot(t,y,label='Euler')
>>> plt.plot(t,z,label='Exacte')
>>> plt.legend(loc='upper left')
>>> plt.show()
```

On voit sur la figure suivante que la solution approchée diverge au cours du temps.



Dans certains cas, on peut avoir besoin d'utiliser cette méthode pour des fonctions à valeurs *vectorielle* (c'est-à-dire à valeurs dans $\mathbb{R}^2$, $\mathbb{R}^3$ ...). La méthode d'Euler est la même, la fonction Python est donc la même mais il faut prendre garde à une chose : l'opérateur + n'effectue pas *l'addition* de deux liste mais leur *concaténation* !

Pour utiliser des fonctions vectorielles, il faut donc ne pas utiliser des variables de type `list` mais des variables du type `array` fourni par le module `numpy` (dans le cas l'opérateur + correspond bien à l'addition de deux tableaux).

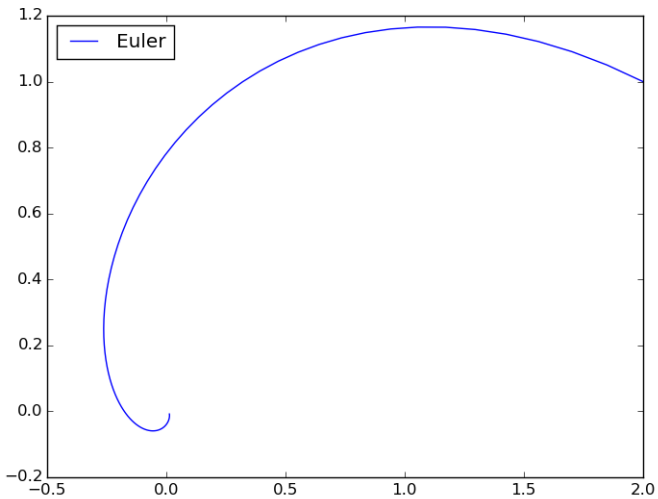
Par exemple, si on veut résoudre sur $[0,5]$ le système différentiel $\begin{cases} x_1'(t) = -x_1(t) - x_2(t) \\ x_2'(t) = x_1(t) - x_2(t) \end{cases}$ avec la condition initiale $x_1(0) = 2$, $x_2(0) = 1$, on considère que la fonction $y = (x_1, x_2)$ est solution de $y' = f(t, y)$ où $f(t, \alpha, \beta) = (-\alpha - \beta, \alpha - \beta)$. Le code devient le suivant :

```
import numpy as np

def f(t,y):
    return np.array([-y[0]-y[1], y[0]-y[1]])

def Euler_vect(f,a,b,y0,n):
    y=np.zeros((n,2))      # cette ligne a changé
    t=[ a+k*float(b-a)/(n-1) for k in range(n)]
    y[0]=y0
    for k in range(1,n):
        y[k]=y[k-1]+(b-a)/float(n-1)*f(t[k-1],y[k-1])
    return t,y
```

En sortie, le tableau y a deux colonnes : la première donne les valeurs approchées de x_1 et la seconde celles de x_2. Les lignes correspondent aux piquets de temps.

```
>>> import matplotlib.pyplot as plt
>>> [t,y]=Euler_vect(f , 0, 5,  np.array([2.,1.]), 100)
>>> plt.plot(y[:,0],y[:,1],label='Euler')
>>> plt.legend(loc='upper left')
>>> plt.show()
```

On a tracé les points $(x_1(t), x_2(t))$ en fonction du temps t.



Le cas des fonctions vectorielles englobe aussi le cas des équations différentielles d'ordre deux, qu'on peut ramener à l'ordre un par *vectorialisation*. Par exemple, considérons l'équation $x'' + x = 0$ avec les conditions initiales $x(0) = 0$ et $x'(0) = 1$. La fonction vectorielle $y = (x, x')$ vérifie l'équation différentielle d'ordre un $y' = f(t, y)$ avec $f(t, \alpha, \beta) = (\beta, -\alpha)$. On définit en Python la fonction f :

```
def f(t,y):
    return np.array([y[1], -y[0]])
```

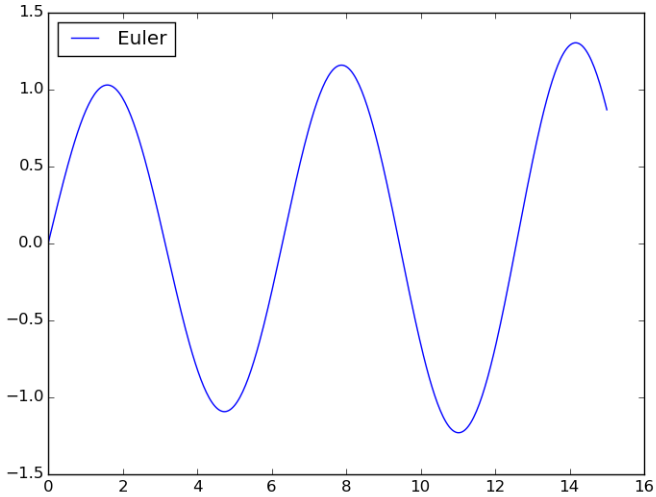
Ensuite on utilise la méthode d'Euler vectorielle :

```
>>> [t,y]=Euler_vect(f , 0, 15,  np.array([0.,1.]), 400)
```

En sortie, le tableau y a deux colonnes : la première donne les valeurs approchées de x et la seconde celles de x'. Les lignes correspondent aux piquets de temps.

```
>>> import matplotlib.pyplot as plt
>>> plt.plot(t,y[:,0],label='Euler')
>>> plt.legend(loc='upper left')
>>> plt.show()
```

On a trace la courbe représentative de la fonction $t \longmapsto x(t)$. On voit que l'approximation diverge puisque la solution exacte est la fonction sinus, et devrait donc rester entre -1 et 1.



4.10 Résolution d'un système linéaire inversible : méthode du pivot de Gauss

Les systèmes triangulaires sont très simples à résoudre.

$$\left\{\begin{array}{rcl} 2x+2y-3z & = & 2 \\ y-6z & = & -3 \\ z & = & 4 \end{array}\right. \iff \left\{\begin{array}{rcl} 2x+2y-3z & = & 2 \\ y & = & 6z-3=21 \\ z & = & 4 \end{array}\right.$$

$$\iff \left\{\begin{array}{rcl} x & = & \frac{1}{2}(-2y+3z+2)=-14 \\ y & = & 21 \\ z & = & 4 \end{array}\right.$$

Dès maintenant, il faut bien comprendre le rôle de l'équivalence, qui assure que le système initial et le système final ont les mêmes solutions.

La résolution d'un système linéaire quelconque passera systématiquement par une première étape pour le mettre sous forme triangulaire en gardant l'équivalence avec le système initial. Si une équation « disparaît » ou bien fournit une contradiction, alors on peut en conclure que :

- Si l'équation $0 = 0$ apparaît, on peut l'éliminer du système en gardant l'équivalence.
- Si l'équation $0 = \beta$ apparaît (avec $\beta \neq 0$), le système initial n'admet pas de solution

Pour un système de Cramer, ces situations ne se présenteront pas.

La résolution d'un système triangulaire sera la dernière étape (la plus simple et rapide) de la résolution d'un système linéaire. On l'appelle souvent *phase de remontée* : on résout les équations de bas en haut, en substituant aux inconnues les valeurs trouvées dans les lignes inférieures.

Pour mettre un système sous forme triangulaire, on va réaliser des *transvections*, c'est-à-dire des opérations élémentaires de la forme suivante : « ajouter tant de fois telle équation à telle autre », afin d'éliminer une inconnue dans des équations... mais pas dans le désordre !

On va décrire ces opérations pour un système 3×3 d'inconnues (x, y, z) dans une situation favorable, en notant L_1, L_2 et L_3 les trois équations en jeu. On suppose que le coefficient en x de la première équation, appelé a, est non nul. On va s'en servir comme *pivot* pour éliminer les autres occurrences de x. Si on note b et c les coefficients en x des deuxième et troisième lignes, le système constitué des équations L_1, $L_2' = L_2 - \dfrac{b}{a}L_1$ et $L_3' = L_3 - \dfrac{c}{a}L_1$ est alors équivalent au premier et ne fait apparaître x que dans la première équation. En supposant que le coefficient en y de la nouvelle deuxième ligne L_2', appelé d, est non nul (c'est alors le nouveau pivot) et en notant e celui de y dans la nouvelle troisième ligne L_3', le système constitué des lignes L_1, L_2' et $L_3'' = L_3' - \dfrac{e}{d}L_2'$ est équivalent au premier système et triangulaire : on est ramené à un cas qu'on sait traiter.

Lors de la première étape, on ne touche pas à la première ligne. De même, à la deuxième étape, on ne touche ni à la première ni à la deuxième ligne, et ainsi de suite.

Dans l'exemple qui suit, on adopte une notation classique : pour dire qu'on change la seconde ligne L_2 en $L_2' = L_2 + \alpha L_1$, on préférera noter $L_2 \longleftarrow L_2 + \alpha.L_1$. Cela signifie qu'on appelle désormais L_2 ce que désignait auparavant $L_2 + \alpha L_1$. Après cinq opérations de ce type, on parlera donc toujours de L_8 plutôt que de L_8''''.

$$\begin{cases} \mathbf{2}x + 2y - 3z = 2 \\ -2x - y - 3z = -5 \\ 6x + 4y + 4z = 16 \end{cases} \underset{L_3 \leftarrow L_3 - 3L_1}{\overset{L_2 \leftarrow L_2 + L_1}{\Longleftrightarrow}} \begin{cases} 2x + 2y - 3z = 2 \\ \mathbf{1}y - 6z = -3 \\ -2y + 13z = 10 \end{cases}$$

$$\overset{L_3 \leftarrow L_3 + 2L_2}{\Longleftrightarrow} \begin{cases} 2x + 2y - 3z = 2 \\ y - 6z = -3 \\ z = 4 \end{cases}$$

$$\Longleftrightarrow \quad \ldots$$

Dans les cas moins favorables, on peut rencontrer en cours de résolution d'un système 3×3 ces trois problèmes :

- Le pivot n'est pas là où on veut : si, à la première étape, le coefficient en x de la première ligne est nul, on peut échanger la première équation avec la deuxième ou la troisième. De même, si à la seconde étape, le coefficient en y (futur pivot) est nul, on peut échanger la deuxième équation avec la troisième, mais pas la première (se souvenir qu'on veut arriver à un système triangulaire : il ne faut pas faire réapparaître x dans les deux dernières équations).
- Il n'y a plus de pivot en une variable : si tous les coefficients en x sont nuls (c'est rare : cela revient à dire que x n'apparaît pas dans le système...), on peut prendre y ou z comme première variable. De même, si après la première étape, y n'apparaît ni dans la deuxième ni dans la troisième équation, on peut prendre z comme deuxième inconnue.
- Il n'y a plus de pivot : cela signifie que les membres de gauche des équations restantes sont nuls. Selon que les membres de droite correspondants sont nuls ou pas, ces équations vont disparaître ou bien rendre le système incompatible.

Les deux dernières situations ne se produiront pas sur un système de Cramer.

Pour la recherche d'un pivot, il faudrait tester la nullité des coefficients sur une colonne. . .mais tester la nullité d'un flottant n'a pas de sens en informatique ! On préfère donc chercher sur la colonne en cours le coefficient le plus élevé en valeur absolue (ou module). Cela s'appelle la *méthode du pivot partiel*.

On fait ici l'hypothèse que le système initial est de Cramer. Il est important de noter que les opérations réalisées vont introduire des systèmes équivalents au premier, qui demeureront donc des systèmes de Cramer.

Comme signalé plus haut, on veut éliminer des variables dans les équations successives. On va donc faire en sorte qu'après k étapes, pour tout i entre 1 et k, la i-ième variable ait disparu de toutes les équations du système à partir de la $(i+1)$-ième. Ce sera l'invariant de boucle.

Ainsi, après la $(n-1)$-ième étape, le système sera bien sous forme triangulaire.

Dans le pseudo-code qui suit, on résout le système $Ax = y$. La ligne L_i désigne à la fois les coefficients de A (qui sont dans une matrice, un tableau bidimensionnel) et les seconds membres, qui sont dans une matrice colonne y. Les indexations de tableaux vont de 0 à $n-1$ comme en Python :

Pour i de 0 à $n-2$ **faire**
  Trouver j entre i et $n-1$ tel que $|a_{j,i}|$ soit maximale.

Échanger L_i et L_j (coefficients de la matrice **et** membres de droite).
Pour k de $i+1$ à $n-1$ **faire**
$L_k \longleftarrow L_k - \frac{a_{k,i}}{a_{i,i}} L_i$
Fin Pour
Fin Pour

Rechercher j entre i et n tel que $|a_{j,i}|$ soit maximale (puis échanger deux lignes) a deux objectifs : d'une part s'assurer que le coefficient en position (i,i) sera différent de 0 (c'est essentiel pour pouvoir pivoter) et, d'autre part, minimiser les erreurs numériques dans la suite du calcul.

Arrivé ici, le système est sous forme triangulaire et il n'y a plus qu'à « remonter », via des substitutions. Le résultat est mis dans un tableau x et il s'agit donc de calculer :

$$x_i = \frac{1}{a_{i,i}} \left(y_i - \sum_{k=i+1}^{n-1} a_{i,k} x_k \right)$$

Pour i de $n-1$ à 0 **faire**
Pour k de $i+1$ à $n-1$ **faire**
$y_i \longleftarrow y_i - a_{i,k} x_k$
Fin Pour
$x_i \longleftarrow \frac{y_i}{a_{i,i}}$
Fin Pour

Dans les programmes Python, les matrices seront représentés par des listes de listes. Par exemple la matrice 2×3 suivante :

$$M = \begin{pmatrix} 0 & 1 & 2 \\ 3 & 4 & 5 \end{pmatrix}$$

peut être définie en Python par le tableau `m` ci-après :

```
m = [[0, 1, 2], [3, 4, 5]]
```

On accède à l'élément $M_{i,j}$ avec l'expression `m[i][j]`.

Comme les listes sont modifiées en place, les fonctions ne renverront rien.

Le premier programme Python prend en entrée une matrice `A` et un indice `i`. Il doit renvoyer un indice `j` $\geq$ `i` tel que $|a_{j,i}|$ soit maximale :

```
1 def chercher_pivot(A, i):
2     n = len(A)          # le nombre de lignes
3     j = i            # la ligne du maximum provisoire
4     for k in range(i+1, n):
5         if abs(A[k][i]) > abs(A[j][i]):
6             j = k           # un nouveau maximum provisoire
7     return j       # en faisant bien attention à l'indentation :-)
```

Les échanges de lignes se passent de commentaires :

```
1 def echange_lignes(A, i, j):
2     nc = len(A[0])
3     for k in range(nc):
4         A[i][k], A[j][k] = A[j][k], A[i][k]
```

Enfin viennent les transvections. Comme pour la fonction précédente, la matrice fournie en entrée est modifiée en place, la fonction ne renvoyant rien. En particulier, l'appel se fera via `transvection_ligne(A,i,j,mu)`, et non via `A = transvection_ligne(A,i,j,mu)` :

```
def transvection_ligne(A, i, j, mu):
    """ L_i <- L_i + mu.L_j """
     nc = len(A[0])      # le nombre de colonnes
     for k in range(nc):
         A[i][k] += mu * A[j][k]
```

Il reste à recoller les morceaux dans la fonction principale :

```
def resolution(A, Y):
    """Résolution de A.X=Y; A doit etre inversible"""
     n = len(A)
     # Mise sous forme triangulaire
     for i in range(n):
         j = chercher_pivot(A, i)
         if j > i:
             echange_lignes(A, i, j)
             echange_lignes(Y, i, j)
         for k in range(i+1, n):
             x = A[k][i] / float(A[i][i])
             transvection_ligne(A, k, i, -x)
             transvection_ligne(Y, k, i, -x)
     # Phase de remontée
     X = [0.] * n
     for i in range(n-1, -1, -1):
         somme=0.
         for j in range(i+1,n):
             somme+=A[i][j]*X[j]
         X[i] = (Y[i][0]-somme) / A[i][i]
     return X
```

⚠ Les variables `A` et `Y` sont modifiées en place par la fonction précédente. Si on souhaite les garder intactes, il faut en faire une copie dès le début du programme...

Testons sur un exemple :

```
>>> resolution([[2,2,-3],[-2,-1,-3],[6,4,4]],[[2],[-5],[16]])
        [-14.000000000000036, 21.000000000000046, 4.000000000000007]
```

Il est clair que la complexité de l'algorithme, en prenant en compte les comparaisons, affectations et opérations arithmétiques, vérifie $c_n = \Gamma(n^3)$.

5 Lire et écrire dans un fichier

On rassemble ci-dessous les commandes python utiles :

1. Connaître le répertoire de travail :

```
>>> import os
>>> os.getcwd()
    '/home/arno/Dropbox/PSI/Python/Cours_Arnaud'
```

Changer de répertoire :

```
>>> os.chdir('../DM/DM1')
>>> os.getcwd()
    '/home/arno/Dropbox/PSI/Python/DM/DM1'
```

2. Ouvrir un fichier : `monFichier=open('adresse du fichier', 'r')`
   — Le 'r' indique qu'on ouvre le fichier en lecture (« read »).
   — Pour l'ouvrir en écriture, remplacer par `'w'`.
   ⚠ Attention, ceci écrase tout fichier existant ! Si le fichier n'existe pas , il est créé.
   Pour ouvrir un fichier en mode « ajout de donnée », mettre `'a'` au lieu de `'w'`. De même, si le fichier n'existe pas, il est créé.
   — Remplacez `monFichier` par le nom sous lequel vous voulez appeler votre fichier dans votre programme.
   — Quand vous avez fini, fermez le fichier grâce à `monFichier.close()`.

3. Lire la ligne courante (si ouvert en mode lecture) `monFichier.readline()`
   Lors de l'ouverture du fichier, on est au début du fichier. À chaque utilisation de cette commande, on passe à la ligne suivante.
   NB : le type de donnée récupérée est toujours du texte. Il faudra éventuellement le convertir en entier ou flottant.

4. *Astuce python* : pour parcourir toutes les lignes du fichier vous pouvez utiliser une boucle « pour » automatique :

   ```
   for ligne in monFichier.readlines():
   ```

   La variable `ligne` prendra comme valeurs successives toutes les lignes du fichier.
   ⚠ Attention au `s` à la fin de `readlines`.
   En fait ceci fonctionne comme pour parcourir une liste (ex : `for i in [1,4,7,12,3]`) ou une chaîne de caractère (ex : `for lettre in 'bonjour'`).

5. Rajouter une ligne à la fin du fichier (si ouvert en mode écriture) :

   ```
   monFichier.write(la_ligne_à_écrire)
   ```

   Si on veut préciser que la prochaine écriture se fera sur un autre ligne, il faut que la chaîne de caractères `la_ligne_à_écrire` se termine par le caractère « fin de ligne » `\n`.

6. Retirer le caractère « fin de la ligne » à la fin d'une chaîne de caractère : `maChaine[:-2]` ou `maChaine.rstrip('\n')`

7. Découper une chaîne de caractère selon les virgules : `maChaine.split(',')`
   Par exemple :

   ```
   >>> 'abc,def,f'.split(',')
           ['abc', 'def', 'f']
   ```

   Cette technique fonctionne avec n'importe quel autre caractère que la virgule ','. Par défaut le caractère est l'espace `' '`.

6 Bases de données

6.1 Généralités

Il s'agit de stocker des données de façon adaptée, de sorte à :

- pouvoir facilement faire des recherches
- prendre le moins de place possible en mémoire.

Un *attribut* d'une base de donnée est une des caractéristiques dont on souhaite enregistrer la valeur, lors d'une entrée. Ainsi, il s'agit du nom des différentes colonnes.

On représente souvent une relation sous forme d'une table, comme nous l'avons fait précédemment. Ainsi, le *schéma relationnel* définit les noms des colonnes d'une table, et le type des entrées de chaque colonne, alors que la *relation* est l'ensemble des données entrées dans la table. Un *enregistrement* (ou une *valeur*) correspond à une ligne de la table. Le *cardinal* de la relation est le nombre de lignes dans la table.

Une *clé primaire* est un attibut tel que les valeurs prises par cet attribut déterminent toutes les autres valeurs de l'enregistrement (donc déterminent toute les lignes).

Nous utiliserons dans la suite l'exemple suivant de tables.

OEUVRES				
IdOeuvre	Compositeur	Oeuvre	Date	Type
COU 1-1	COU 1	Messe des couvents	1690	messe
BAC 1-1	BAC 1	Variations Goldberg	1740	variations
BAC 1-2	BAC 1	L'art de la fugue	1750	recueil
BAC 1-3	BAC 1	Le clavier bien tempéré I	1722	recueil
BAC 1-4	BAC 1	Messe en si mineur	1748	messe
MOZ 1-1	M0Z 1	Don Giovanni	1787	opéra
MOZ 1-2	MOZ 1	Die Zauberflöte	1791	opéra
MOZ 1-3	MOZ 1	Concerto pour flûte et harpe	1778	concerto
MOZ 1-4	MOZ 1	Messe du couronnement	1779	messe
MOZ 1-5	MOZ 1	Requiem	1791	messe
BEE 1-1	BEE 1	Symphonie n° 5	1808	symphonie
BEE 1-2	BEE 1	Symphonie n° 9	1824	symphonie
BEE 1-3	BEE 1	Quatuor n° 13	1825	quatuor
BEE 1-4	BEE 1	Grande Fugue	1825	quatuor
BEE 1-5	BEE 1	Sonate n° 29	1818	sonate
BEE 1-6	BEE 1	Sonate pour violon et piano n° 5	1801	sonate

NATIONALITÉ		
IdNat	IdComp	pays
1	COU 1	France
2	BAC 1	Allemagne
3	MOZ 1	Autriche
4	BEE 1	Allemagne
5	BEE 1	Autriche
6	HAE 1	Allemagne
7	HAE 1	Angleterre
8	STR 1	Russie
9	STR 1	France
10	BAC 2	Allemagne
11	BAC 2	Italie
12	BAC 2	Angleterre

COMPOSITEURS				
IdComp	Nom	Prénom	naissance	mort
COU 1	Couperin	François	1668	1733
BAC 1	Bach	Johann Sebastian	1685	1750
MOZ 1	Mozart	Wolfgang Amadeus	1756	1791
BEE 1	Beethoven	Ludwig (von)	1770	1827
HAE 1	Haendel	Georg Friedrich	1685	1759
STR 1	Stravinski	Igor	1882	1971
BAC 2	Bach	Johann Christian	1735	1782

SOLISTES		
IdSol	IdOeuvre	instrument soliste
1	COU 1-1	clavecin
2	BAC 1-1	clavecin
3	BAC 1-2	clavecin
4	MOZ 1-3	flûte
5	MOZ 1-3	harpe
6	BEE 1-5	piano
7	BEE 1-6	violon
8	BEE 1-6	piano

INSTRUMENTS	
instrument	famille
orgue	claviers
clavecin	claviers
flûte	bois
harpe	cordes
piano	claviers
violon	cordes

INSTRUMENTATION		
IdInstr	IdOeuvre	formation intrumentale
BAC 1-4-1	BAC 1-4	orchestre
BAC 1-4-2	BAC 1-4	choeur
BAC 1-4-3	BAC 1-4	chanteurs
MOZ 1-1-1	MOZ 1-1	orchestre
MOZ 1-1-2	MOZ 1-1	choeur
MOZ 1-1-3	chanteurs	
MOZ 1-2-1	MOZ 1-2	orchestre
MOZ 1-2-2	MOZ 1-2	choeur
MOZ 1-2-3	MOZ 1-2	chanteurs
MOZ 1-3-1	MOZ 1-3	orchestre
MOZ 1-4-1	MOZ 1-4	orchestre
MOZ 1-4-2	MOZ 1-4	choeur
MOZ 1-4-3	MOZ 1-4	chanteurs
MOZ 1-5-1	MOZ 1-5	orchestre
MOZ 1-5-2	MOZ 1-5	choeur
MOZ 1-5-3	MOZ 1-5	chanteurs
BEE 1-1-1	BEE 1-1	orchestre
BEE 1-2-1	BEE 1-2	orchestre
BEE 1-2-2	BEE 1-2	choeur
BEE 1-2-3	BEE 1-2	chanteurs
BEE 1-3-1	BEE 1-3	quatuor à cordes
BEE 1-4-1	BEE 1-4	quatuor à cordes

De façon naturelle, on peut choisir les clés primaires suivantes :

- `IDOEUVRE` pour la table `OEUVRES`
- `IDNAT` pour la table `NATIONALITÉ`
- `IDCOMP` pour la table `COMPOSITEURS`
- `IDSOL` pour la table `SOLISTES`
- `IDINSTR` pour la table `INSTRUMENTATION`
- `INSTRUMENT` pour la table `INSTRUMENT`

Nous abordons dans ce cours l'aspect principal de l'utilisation d'une base de donnée, à savoir la rédaction de requêtes en langage `SQL` afin d'extraire d'une base de donnée les informations qui nous intéressent. Pour rédiger une requête, il est indispensable de connaître précisément la structure de la base (les noms des tables, les noms des attributs, les clés primaires). Ainsi, la première étape est souvent de rechercher et comprendre cette structure.

Ensuite, les requêtes sont toutes effectuées à l'aide de l'instruction `SELECT ... FROM ... WHERE ...` ; et de constructions algébriques effectuées sur les tables, en particulier de jointures (`JOIN ON`). Une requête consiste en une demande d'extraction d'un ensemble d'attributs, vérifiant une certaine propriété.

- Les attributs demandés peuvent être pris dans différentes tables (nécessite des jointures ou produits cartésiens)
- Les conditions peuvent porter sur les attributs demandés, ou sur d'autres attributs, des mêmes tables, ou non.
- La réponse consiste en l'ensemble des données des attributs selectionnés, tels que les entrées (complètes) associées vérifient les conditions souhaitées.

6.2 Requêtes simples

Une *requête simple* est une requête portant sur l'extraction d'attributs d'une même table, la condition d'extraction s'exprimant uniquement à l'aide des attributs de cette même table.

Ainsi, tout se passe dans une unique table. L'instruction incontournable est :

```
SELECT Att1, Att2, ...
    FROM nom_table
    WHERE conditions ;
```

La sélection SELECT est à voir comme une projection sur un sous-ensemble des colonnes (on ne conserve que certaines colonnes), alors que la condition WHERE est à voir comme une sélection d'un sous-ensemble de lignes (on ne garde que les lignes vérifiant les conditions requises).

La clause WHERE est optionnelle. Si on ne l'utilise pas, il s'agit de l'extraction des colonnes souhaitées, sans restriction :

```
SELECT instrument FROM instrument ;
```

retourne l'ensemble des instruments solistes présents dans la base, à savoir :

```
'luth'
'violoncelle'
'violon'
'viole de gambe'
'clavecin'
'piano'
'orgue'
'flûte'
'hautbois'
'baryton'
'soprano'
'clarinette'
'cor'
'basson'
'haute-contre'
```

L'instruction suivante :

```
SELECT instrument FROM instrument WHERE famille = 'claviers' ;
```

renvoie quand à elle uniquement les instruments à clavier :

```
'clavecin'
'piano'
'orgue'
```

Remarquez que pour obtenir l'ensemble des familles d'instruments, on formule assez logiquement la requête :

```
SELECT famille FROM instrument ;
```

On reçoit la réponse suivante, peu satisfaisante :

```
'cordes pincées'
'cordes frottées'
'cordes frottées'
'claviers'
'claviers'
'claviers'
'bois'
'bois'
'voix'
'voix'
'bois'
'cuivre'
'bois'
'voix'
```

Ainsi, SELECT ne supprime pas les redondances dans les réponses. On peut le forcer à le faire en ajoutant l'option DISTINCT :

```
SELECT DISTINCT famille FROM instrument ;
```

On obtient :

```
'cordes pincées'
'cordes frottées'
'claviers'
'bois'
'voix'
'cuivre'
```

Enfin, on peut remplacer la liste des attributs par * si on veut selectionner l'ensemble des attributs de la table :

```
SELECT * FROM compositeur WHERE naissance = 1685 ;
```

On obtient :

```
'BAC 1',  'Bach',       'Johann Sebastian',  1685,  1750
'HAE 1',  'Haendel',    'Georg Friedrich',   1685,  1759
'SCA 1',  'Scarlatti',  'Domenico',          1685,  1757
```

Voici la liste des tests au programme pour exprimer la condition WHERE :

```
= > < != <= >=     # comparaisons sur des nombres ou chaînes de caractère.
```

Par ailleurs, on peut utiliser des opérations sur les attributs, soit pour exprimer des tests, soit pour former de nouvelles colonnes. Former de nouvelles colonnes se fait en indiquant dans la clause SELECT les opérations à faire à partir des autres attributs pour créer cette colonne.

```
+ * - /       # Opérations arithmétiques usuelles
```

Voici quelques exemples. Retourner Nom Prénom pour les compositeurs morts entre 1820 et 1830 :

```
SELECT nom, prenom FROM compositeur WHERE mort >= 1820 AND mort <= 1830 ;
```

On obtient :

```
'Beethoven',  'Ludwig (Van)'
'Schubert',   'Franz'
```

Donner noms des compositeurs nés avant 1700 :

```
SELECT nom FROM compositeur WHERE naissance < 1700 ;
```

On obtient :

```
'Couperin',  'François'
'Bach',      'Johann Sebastian'
'Haendel',   'Georg Friedrich'
```

Certaines opérations mathématiques portant sur l'ensemble des valeurs d'un attribut sont possibles. Ce sont les *fonctions agrégatives* :

```
COUNT(*)      # nombre de lignes
COUNT(ATTR)      # nombre de lignes remplies pour cet attribut
AVG(ATTR)      # moyenne
SUM(ATTR)      # somme
MIN(ATTR)      # minimum
MAX(ATTR)      # maximum
```

Les fonctions agrégatives servent à définir de nouvelles colonnes, mais ne peuvent pas être utilisées dans une condition. Par exemple, nom et âge du composteur mort le plus vieux :

```
SELECT nom, MAX(mort - naissance) FROM compositeur ;
```

Réponse :

```
'Ysaÿe', 93
```

Les fonctions agrégatives peuvent s'utiliser avec l'instruction `GROUP BY Att HAVING Cond` permettant de calculer les fonctions agrégatives sur des paquets de données prenant la même valeur pour l'attribut `Att`, en ne gardant que les lignes vérifiant la condition `Cond`.

Par exemple, date de la première sonate pour des compositeurs avant N dans l'ordre alphabétique :

```
SELECT compositeur, oeuvre, MIN(date) FROM oeuvre
GROUP BY compositeur, type HAVING (compositeur < 'N') AND (type = 'sonate') ;
```

Enfin, notons qu'il est possible d'ordonner les résultats par ordre croissant (numérique ou alphabétique) grâce à `ORDER BY`. Par exemple, compositeurs nés entre 1870 et 1890, par ordre de naissance :

```
SELECT nom, naissance, mort FROM compositeur
WHERE naissance BETWEEN 1870 AND 1890 ORDER BY naissance ;
```

6.3 Constructions ensemblistes

Lorsqu'on veut extraire des attributs de plusieurs tables, il faut utiliser des constructions ensemblistes permettant de combiner plusieurs tables. Nous introduisons ici l'union, l'intersection et le produit cartésien..

C'est à partir de cette dernière construction qu'on construira les « jointures « permettant des requêtes complexes sur plusieurs tables.

L'*intersection* de deux extractions de deux table peut se faire à condition que les attributs extraits des deux tables soient de même format. Le résultat est alors l'ensemble des valeurs de cet (ou ces) attribut commun aux deux tables. Chacune des deux tables peut elle-même être le résultat d'une extraction précédente.
La syntaxe utilise `INTERSECT`. Sur un exemple, identifiants de compositeurs nés entre 1700 et 1800 et ayant écrit une sonate :

```
SELECT idcomp FROM compositeur WHERE naissance>=1700 AND naissance<=1800
INTERSECT
SELECT DISTINCT compositeur FROM oeuvre WHERE type = 'sonate' ;
```

Une intersection peut souvent se reexprimer plus simplement avec une sous-requête et une opération booléenne `AND`. Cela peut être efficace lorsqu'on veut croiser deux tables complètes (par exemple deux tables de clients de deux filiales) :

```
SELECT * FROM table1
INTERSECT
SELECT * FROM table2
```

L'*union* de deux tables est possible si les attributs sont en même nombre et de même type. Il s'agit des enregistrements présents dans l'une ou l'autre de ces tables. L'union ne conserve que les attributs distincts.

Par exemple pour fusionner deux tables :

```
SELECT * FROM table1
UNION
SELECT * FROM table2
```

Ici encore, il est souvent possible de remplacer avantageusement une union par une opération booléenne `OR`.
La dernière opération ensembliste est le *produit cartésien*. Le produit cartésien de deux tables $\mathscr{R}_1$ et $\mathscr{R}_2$ est l'ensemble des $n+p$-uplets $(x_1,\dots,x_n,y_1,\dots,y_p)$, pour toutes les $(x_1,\dots,x_n)\in\mathscr{R}_1$ et $(y_1,\dots,y_p)\in\mathscr{R}_2$, sans préoccupation de concordance des attributs de $\mathscr{R}_1$ et $\mathscr{R}_2$ qui pourraient être reliés.

Le produit cartésien se fait en extrayant simultanément les colonnes des deux tables :

```
SELECT * FROM tab1, tab2 ;
```

On peut faire le produit cartésien d'extractions de deux tables. Dans ce cas, on liste les attributs à garder après `SELECT`, en précisant dans quelle table l'attribut se trouve par une notation suffixe. Si l'attribut (sous le même nom) n'apparaît pas dans les deux tables, on peut omettre la notation suffixe (il n'y a pas d'ambiguïté) :

```
SELECT tab1.Att1, Att2, tab2.Att3 FROM tab1, tab2 ;
```

Dans cet exemple, l'attribut `Att2` est supposé n'exister que dans l'une des deux tables.

Pour éviter des lourdeurs d'écriture (dans les conditions), et pouvoir référer plus simplement aux différents attributs (c'est utile aussi lorsqu'un attribut est obtenu par un calcul et non directement), on peut donner un alias aux attributs selectionnés :

```
SELECT tab1.Att1 AS B1, Att2*Att4 AS B2 , tab2.Att3 AS B3 FROM tab1, tab2 ;
```

Ici, on suppose que `Att2` et `Att4` sont deux attribus numériques. On pourra utiliser les valeurs des attributs de la nouvelle table via les alias `B1`, `B2`, et `B3`.

Le produit cartésien est une opération coûteuse et peu pertinente en pratique si elle n'est pas utilisée en parallèle avec une opération de sélection des lignes. En effet, on associe le plus souvent des lignes n'ayant rien à voir, par exemple une ligne consacrée à une oeuvre de Mozart et une ligne consacrée aux données personnelles de Stravinsky. Mais cette opération est le point de départ de la notion de *jointure*.

La technique de la jointure permet de former une table à l'aide d'une selection d'attributs provenant de deux tables différentes : plus précisément, une jointure de deux tables consiste à considérer le produit cartésien de ces deux tables (ou d'extractions), en identifiant deux colonnes (une de chaque table), de sorte à ne garder dans le produit cartésien que les n-uplets tels que les valeurs soient les mêmes pour ces deux attributs.

En d'autre terme, une jointure revient à une instruction de sélection sur le produit cartésien. Par exemple, pour sélectionner deux attributs de chaque table en faisant une jointure en identifiant l'attribut `T4` de la table 2 et l'attribut `T1` de la table 1 :

```
SELECT T1.Att1, T1.Att2, T3.Att3 FROM T1, T2 WHERE T1.Att1 = T2.Att4 ;
```

Un exemple concret :

```
SELECT nom, prenom, oeuvre FROM compositeur, oeuvre WHERE idcomp = compositeur
```

Résultat partiel :

```
('Weill', 'Kurt', 'Aufstieg und Fall der Stadt Mahagony')
('Weill', 'Kurt', 'Die Dreigroschenoper')
```

Une autre syntaxe parfois plus commode (notamment pour itérer le procéder en opérant des jointures successives) se fait avec `JOIN ... ON ...` :

```
SELECT nom, prenom, oeuvre FROM compositeur JOIN oeuvre ON idcomp = compositeur ;
```

On peut combiner cette jointure avec une sélection conditionnelle `WHERE`. Par exemple, oeuvres écrites par un compositeur à moins de 18 ans :

```
SELECT nom, prenom, oeuvre FROM compositeur JOIN oeuvre ON idcomp = compositeur
WHERE date - naissance <= 18 ORDER BY nom ;
```

On peut faire des jointures successives si on veut des attributs de tableaux plus éloignés, ou si on veut extraire des données de plus de deux tables :

```
SELECT nom, prenom, oeuvre.oeuvre, soliste.instrument
FROM compositeur JOIN oeuvre ON idcomp = compositeur
JOIN soliste ON soliste.oeuvre = idoeuvre
JOIN instrument ON instrument.instrument = soliste.instrument
WHERE famille = 'bois' ORDER BY nom ;
```

6.4 Requêtes composées

Dans ce paragraphe, on utilisera la table suivante :

Table eleve					
prenom	nom	filiere	numero	lycee_origine	note_bac
Mathilde	Dufour	PCSI	2	Calmette	18
Léa	Dupond	MPSI	2	Massena	14
Paul	Dugommier	PCSI	1	Massena	12
Mathilde	Dugommier	MPSI	1	Calmette	15
Clément	Durand	PCSI	1	Parc Imperial	13

On a déja vu `HAVING`, qui effectue une sélection en aval d'une agrégation. Par exemple, donner les moyennes par filière :

```
SELECT filiere,AVG(notebac) FROM eleves GROUP BY filiere ;
```

et ne garder que les moyennes strictement supérieures à 14.4 :

```
SELECT filiere,AVG(notebac) FROM eleve GROUP BY filiere HAVING AVG(notebac) > 14.4 ;
```

Voyons un exemple équivalent à celui plus haut, mais sans `HAVING` :

```
SELECT filiere,moy
FROM (SELECT filiere,AVG(notebac) FROM eleve GROUP BY filiere)
WHERE AVG(notebac)>14.4 ;
```

Ici, on effectue une première requête (à l'intérieur des parenthèses) produisant des lignes de la forme filiere, moyenne, puis on réutilise immédiatement la table produite dans une nouvelle requête. Celle-ci est équivalente à la première et produit le même résultat.

Il faut bien comprendre qu'appliquer des requêtes à une table produit une nouvelle table !

Un autre exemple est le suivant : quels sont les élèves ayant eu la plus haute note au bac ? Ici, il faut récupérer d'abord la plus haute note au bac, puis refaire une requête pour sélectionner les élèves ayant eu cette note. En `SQL`, on obtient :

```
SELECT * FROM eleve WHERE notebac=(SELECT MAX(notebac) FROM eleve) ;
```

Ici, on utilise l'identification entre une table à une ligne et une colonne et la valeur de cette case. Attention, on pourrait être tenté d'écrire quelque chose comme :

```
SELECT nom,prenom,MAX(notebac) FROM eleve ;
```

qui n'a pas vraiment de sens, mais fonctionne en SQL. Le résultat produit est invariablement une table avec une seule ligne : on obtient les nom et prénom d'un seul élève ayant eu la meilleure note au bac (avec cette note).

Pour conclure, retenir qu'on peut faire des sous-requêtes en les plaçant entre parenthèses, et qu'on peut utiliser comme valeur une table à une ligne et une colonne pour faire une sélection.

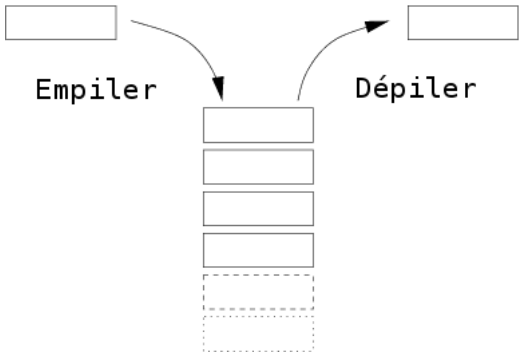
Chapitre 2

Cours de deuxième année

1 Piles

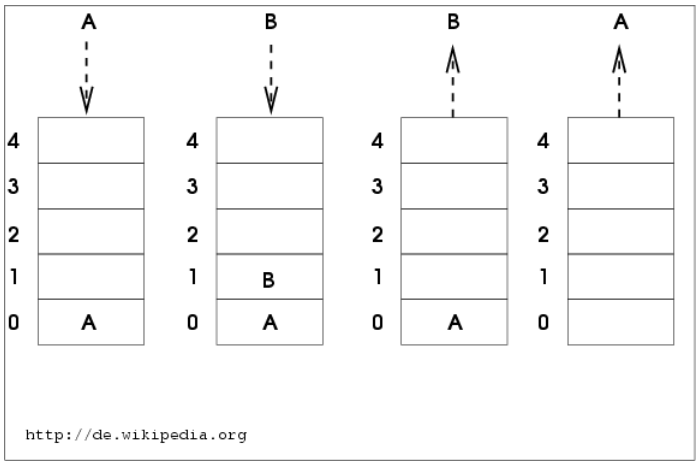
1.1 Généralités

En informatique, une *pile* (en anglais *stack*) est une structure de données fondée sur le principe » dernier arrivé, premier sorti » (ou LIFO pour « Last In, First Out »), ce qui veut dire que les derniers éléments ajoutés à la pile seront les premiers à être récupérés.



Le fonctionnement est celui d'une pile d'assiettes : on ajoute des assiettes sur la pile, et on les récupère dans l'ordre inverse, en commençant par la dernière ajoutée.

Sur la figure suivante on *empile* A puis B et ensuite on *dépile* dans l'ordre B, puis A.



Les Piles ont beaucoup d'applications :

— Dans un navigateur web, une pile sert à mémoriser les pages Web visitées. L'adresse de chaque nouvelle page visitée est empilée et l'utilisateur désempile l'adresse de la page précédente en cliquant le bouton « Afficher la page précédente « .

— L'évaluation des expressions mathématiques en notation post-fixée (ou polonaise inverse) utilise une pile.

— La fonction « Annuler la frappe » (en anglais Undo) d'un traitement de texte mémorise les modifications apportées au texte dans une pile.

— Un algorithme de recherche en profondeur dans un graphe utilise une pile pour mémoriser les nœuds visités.

— Par exemple, on peut très simplement inverser les éléments contenus dans un tableau ou dans une chaîne de caractères (pour tester un palindrome) en utilisant une pile. Il suffit d'empiler les éléments sur une pile puis de reconstituer le tableau (ou la chaîne) inverse en désempilant les éléments.

Un des gros avantages des piles est qu'il n'est pas nécessaire de connaître à l'avance la taille de la pile, c'est-à-dire le nombre d'éléments qu'on va empiler.

1.2 Primitives

Pour manipuler les Piles, on dispose des fonctions suivantes, appelées *primitives*.

Trois primitives indispensables :

- *Empiler.* Ajoute un élément sur la pile. Terme anglais correspondant : « Push ».
- *Désempiler.* Enlève un élément de la pile et le renvoie. Terme anglais correspondant : « Pop ».
- *La pile est-elle vide ?* Renvoie vrai si la pile est vide, faux sinon. Terme anglais « isempty ».

D'autres primitives peuvent être obtenues à partir des 3 premières :

- *Nombre d'éléments de la pile.* Renvoie le nombre d'éléments dans la pile.
- *Quel est l'élément de tête ?* Renvoie l'élément de tête sans le dépiler. Terme anglais correspondant : « Peek ».
- *Vider la pile.* Désempiler tous les éléments. Terme anglais correspondant : « Clear ».
- *Dupliquer l'élément de tête* et *Échanger les deux premiers éléments.* Existe sur les calculatrices fonctionnant en notation polonaise inverse (style marque HP). Termes anglais correspondants : « Dup » et « Swap » respectivement.

En Python, on utilise des listes. On ne s'autorise aucune fonction exceptée les trois suivantes qui correspondent aux primitives indispensables : les méthodes `append()` et `pop()` et le test d'égalité à la liste vide.

```
1 pile=[5,3,4,5]
2 pile.append(1)
3 print pile
4     [5,2,4,5,1]
5 pile.pop()
6     1
7 print pile
```

```
8       [5,2,4,5]
9  pile==[]
10      False
```

Remarquer que la méthode `append()` renvoie `None` alors que la méthode `pop()` renvoie l'élément désempilé (ou une erreur si la pile est vide).

On peut ensuite programmer les autres primitives. On modélise ainsi des piles *non bornées*.

Dans certains cas, on peut n'utiliser que des *piles bornées* (limitation imposée par le logiciel, ou taille mémoire réduite). Dans ce cas, si $N \in \mathbb{N}^*$ est la taille limite d'une pile bornée, on utilise en Python une liste de taille N, qui peut s'initialiser par les instructions `pile=[0]*N` ou `pile=[ 0 for k in range(N) ]`.

Par convention le premier élément de la liste va donner la longueur de la pile, et les autres éléments seront les éléments de la pile.

✎ **Exemple.** Pour $N = 8$, `pile=[3,2,1,4,0,0,0]` est une pile de longueur 3.

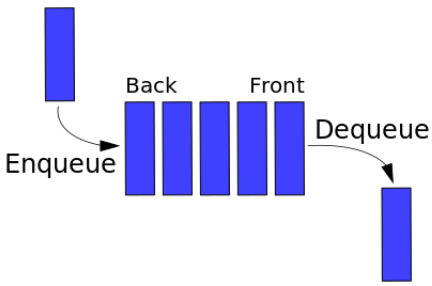
Les primitives indispensables se programment ainsi :

```
1  def push(pile,x):
2      global N
3      assert pile[0]<N
4      n=pile[0]
5      pile[n+1]=x
6      pile[0]+=1
7
8  def isempty(pile):
9      return pile[0]==0
10
11 def pop(pile):
12     assert not(isempty(pile))
13     n=pile[0]
14     pile[0]-=1
15     return pile[n]
```

1.3 Files

En informatique, une *file* (queue en anglais) est une structure de données basée sur le principe du »premier entré, premier sorti », en anglais FIFO (First In, First Out), ce qui veut dire que les premiers éléments ajoutés à la file seront les premiers à être récupérés.

Le fonctionnement ressemble à une file d'attente à la poste : les premières personnes à arriver sont les premières personnes à sortir de la file.



Les applications sont différentes de celles des piles :

— Mémoriser temporairement des transactions qui doivent attendre pour être traitées dans l'ordre d'arrivée.

— Les serveurs d'impression, qui doivent traiter les requêtes dans l'ordre dans lequel elles arrivent, et les insèrent dans une file d'attente (ou une queue).

— Certains moteurs multitâches, dans un système d'exploitation, qui doivent accorder du temps-machine à chaque tâche, sans en privilégier aucune.

— Pour créer toutes sortes de mémoires tampons (en anglais « buffers »).

2 Récursivité

2.1 Généralités

En informatique « pratique » les fonctions *récursives* sont simplement des fonctions dont le calcul nécessite d'invoquer la fonction elle-même. C'est l'analogue des formules de récurrences en mathématiques.

Les fonctions non récursives seront appelées fonctions *itératives*.

✎ **Exemple.** La fonction factorielle vérifie $0! = 1$ et $(n+1)! = (n+1) \times n!$. Elle se programme donc de manière itérative avec une boucle `for` :

```
1 def factorielle(n):
2     fact=1
3     for k in range(2,n+1):
4         fact=fact*k
5     return fact
```

ou une boucle `while` :

```
1 def factorielle2(n):
2     fact=1
3     k=2
4     while k<=n:
5         fact=fact*k
6         k+=1
7     return fact
```

On peut aussi transcrire la formule de récurrence en une fonction récursive :

```
1 def factorielle_rec(n):
2     if n==0 or n==1:
3         return 1
4     else:
5         return n*factorielle_rec(n-1)
```

⚠ Cette exemple montre les trois grands principes d'une fonction récursive :

- Une fonction récursive doit comporter une *close d'arrêt* qui consiste en la donnée d'un ou plusieurs cas de base : c'est le type de la sortie de cette close qui va déterminer le type de la sortie de la fonction.
- Une fonction récursive doit appeler la close d'arrêt au bout d'un nombre *fini* d'étapes (ce qui assure la terminaison de l'algorithme).

- Une fonction récursive *s'appelle elle-même*, mais avec différentes valeurs de (ou des) l'argument(s).

La structure générale est donc la suivante :

```
def fonction_recursive(paramètre):
    if ... :      # close d'arrêt
        return ...  # cette ligne détermine le type de la variable de sortie
    else:
        return ... # instructions contenant un appel à fonction_recursive
```

On remarque que l'instruction `return` doit être présente au moins deux fois : une fois pour la close d'arrêt, une fois pour l'appel récursif.

2.2 Exemples plus évolués

Il est possible de donner plusieurs closes d'arrêt grâce à l'instruction `elif` :

```
def factorielle_rec(n):
    if n==0:
        return 1
    elif n==1:
        return 1
    else:
        return n*factorielle_rec(n-1)
```

Dans l'exemple de la fonction `factorielle_rec`, l'appel du rang `n` lance l'appel du rang `n-1`, ce qui n'est pas sans rappeler le principe de « récurrence simple » en mathématiques. En fait, il est aussi possible d'appeler la fonction avec d'autres valeurs du paramètres ; ceci se rapproche alors du principe de « récurrence forte ».

Par exemple le calcul de x^n peut se programmer en version <u>itérative</u> :

```
def puissance(x,n):
    puiss=1
    for k in range(n)
        puiss*=x
    return puiss
```

mais aussi de façon <u>récursive</u> :

```
def puissance_rec(x,n):
    if n==0:
        return 1
    elif n==1:
        return x
    else:
        return x*puissance_rec(x,n-1)
```

Mais on peut accélérer un peu l'algorithme en utilisant les identités suivantes :

$$\begin{cases} x^{2k} &= \left(x^k\right)^2 \\ x^{2k+1} &= x\left(x^k\right)^2 \end{cases}$$

On obtient le programme <u>récursif</u> :

```
def speed_puissance_rec(x,n):
    if n==0:
        return 1
    else:
        r=speed_puissance_rec(x,n/2) # Python 2.7
        if n%2==0:
            return r*r
        else:
            return x*r*r
```

On voir sur cet exemple qu'une instruction du type `speed_puissance_rec(x,n/2)*speed_puissance_rec(x,n/2)` est trompeuse, puisque les deux appels ne vont pas se faire en même temps et beaucoup de calculs identiques vont être effectuées deux fois ; d'où l'intérêt de la variable auxiliaire `r`.

Il est aussi possible de définir des fonctions *mutuellement récursives*. Il suffit de les écrire l'une à la suite de l'autre :

```
def pair(n):
    if n==0:
        return True
    else:
        return impair(n-1)

def impair(n):
    if n==0:
        return False
    else:
        return pair(n-1)
```

Remarquons qu'en utilisant l'évaluation paresseuse (= retardée) des opérateurs `or` et `and`, on aurait pu écrire plus brièvement :

```
def pair(n):
    return n==0 or impair(n-1)

def impair(n):
    return n!=0 and pair(n-1)
```

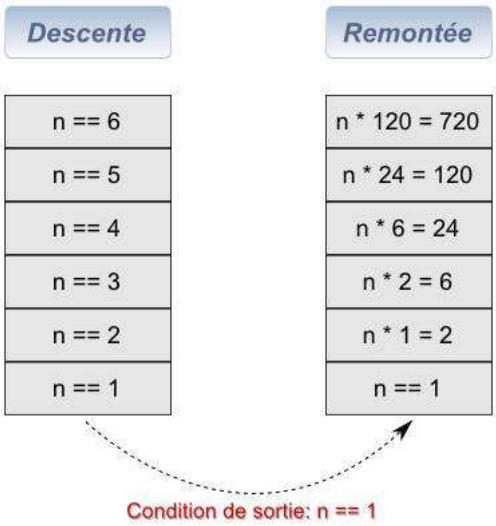
2.3 Pile d'exécution d'une fonction récursive

Les appels successifs d'une fonction récursive sont stockés dans une pile : c'est la *phase de descente*. Une fois que la close d'arrêt est demandée, les appels sont ensuite désempilés jusqu'à arriver à l'appel initial : c'est la *phase de remontée*.

Par défaut, Python limite à 1000 le nombre d'appels récursifs. Tout dépassement provoquera une erreur `RuntimeError: maximum recursion depth exceeded`. On peut augmenter la taille de la pile en utilisant le module `sys`. Par exemple, pour une pile de taille 2000 :

```
import sys
sys.setrecursionlimit(2000)
```

La figure suivante montre les phases de descente et de remontée pour le calcul de 6!.



On peut aussi visualiser l'exécution pas à pas sur le site :
`http://www.pythontutor.com/visualize.html#mode=edit`.

2.4 Terminaison et correction d'une fonction récursive

Reprenons la fonction factorielle récursive :

```
1 def factorielle_rec(n):
2     if n==0 or n==1:
3         return 1
4     else:
5         return n*factorielle_rec(n-1)
```

On peut remarquer que l'appel de `factorielle_rec(-1)` ne va s'arrêter qu'une fois dépassée la capacité de la pile d'exécution. On dit que la fonction ne *termine* pas dans tous les cas.

On peut corriger ce problème en « levant une exception » avec l'instruction `assert` :

```
1 def factorielle_rec(n):
2     assert n>=0, 'n doit être >=0'
3     if n==0 or n==1:
4         return 1
5     else:
6         return n*factorielle_rec(n-1)
```

Dans ce cas la fonction *termine* dans tous les cas : en effet, soit on lui donne un entier $n < 0$ et on tombe sur l'exception, soit on lui donne un entier $n \geq 0$, et dans ce cas les différents appels récursifs vont amener à un appel avec l'argument 1 (ou 0 si $n = 0$ au départ) qui fait partie de la close d'arrêt, et donc le programme termine aussi dans ce cas là.

On peut donc retenir la méthode générale suivante : pour montrer qu'un algorithme récursif termine, il faut montrer que, pour toutes valeurs de l'argument, on arrivera en un nombre fini d'étapes à un appel de la close d'arrêt.

⚠ Il existe des fonctions récursives pour lesquelles on ne sait pas démontrer la terminaison. Par exemple la fonction de Syracuse :

```
def syracuse(n):
    if n==0 or n==1:
        return 1
    else:
        if n%2==0:
            return syracuse(n/2) #Python 2.7
        else:
            return syracuse(3*n+1)
```

Pour éviter les valeurs de l'argument pour lesquelles l'algorithme de ne termine pas, on peut aussi utiliser les instructions `while` et `input` pour redemander une valeur à l'utilisateur (et on suppose qu'il le fera au bout d'un nombre fini d'essais...) :

```
def factorielle_rec(n):
    while n<0:
        n=input('n doit être positif, recommencez: ')
        n=int(n) # input donne une chaîne de caractères
    if n==0 or n==1:
        return 1
    else:
        return n*factorielle_rec(n-1)
```

Examinons maintenant le problème de la *correction* d'un algorithme récursif, c'est-à-dire de la preuve que l'algorithme effectue bien les tâches souhaitées.

La méthode est naturelle : on utilise une démonstration par le principe de récurrence (simple ou forte).

Par exemple, pour démontrer la correction de la fonction `factorielle_rec`, on considère le prédicat :

$$H_n : \quad \text{« la fonction } \texttt{factorielle\_rec(n)} \text{ donne en sortie } n! \text{ »}$$

Il est clair que H_0 est vraie : c'est la close d'arrêt. Supposons H_n vraie à un rang $n \in \mathbb{N}$ fixé. L'instruction `factorielle_rec(n+1)` exécute l'instruction `(n+1)*factorielle_rec(n)`. D'après l'hypothèse de récurrence H_n, l'instruction `factorielle_rec(n)` renvoie $n!$ et donc l'instruction `factorielle_rec(n+1)` renvoie $(n+1) \times n! = (n+1)!$: H_{n+1} est donc vraie. D'après le principe de récurrence, H_n est vraie pour tout $n \in \mathbb{N}$: la correction de l'algorithme est donc démontrée pour les entiers naturels (remarquer que nous ne sommes pas intéressés au cas $n < 0$).

Pour alléger la rédaction, peut aussi démontrer simultanément la *terminaison* et la *correction* de l'algorithme récursif. Par exemple, pour la fonction `factorielle_rec`, on considère le prédicat :

$$H_n : \quad \text{« la fonction } \texttt{factorielle\_rec(n)} \textit{ termine} \text{ et donne en sortie } n! \text{ »}$$

Il est clair que H_0 est vraie : c'est la close d'arrêt. Supposons H_n vraie à un rang $n \in \mathbb{N}$ fixé. L'instruction `factorielle_rec(n+1)` exécute l'instruction `(n+1)*factorielle_rec(n)`. D'après l'hypothèse de récurrence H_n, l'instruction `factorielle_rec(n)` *termine* et renvoie $n!$ et donc l'instruction `factorielle_rec(n+1)` *termine* et renvoie $(n+1) \times n! = (n+1)!$: H_{n+1} est donc vraie. D'après le principe de récurrence, H_n est vraie pour tout $n \in \mathbb{N}$: la *terminaison* et la correction de l'algorithme sont donc démontrées pour les entiers naturels.

⚠ Pour la fonction récursive `syracuse`, elle renvoie toujours 1 (pour un argument dans $\mathbb{N}$), mais personne n'a réussi à le démontrer pour le moment.

2.5 Complexité d'un algorithme récursif

Rappelons qu'étudier la complexité d'un algorithme, c'est évaluer le nombre d'opérations élémentaires qu'il effectue, ou encore son occupation mémoire lors de son exécution.

Si n est son argument, on note $C(n)$ le nombre d'opérations élémentaires effectuées, et $A(n)$ l'occupation mémoire. On cherche ensuite, en étudiant l'algorithme une relation de récurrence vérifiée par la suite $C(n)$, ou encore la suite $A(n)$. On essaye ensuite d'en déduire une expression explicite mais cela est rarement possible à cause de la difficulté des calculs. On cherche souvent à dominer $C(n)$ et $A(n)$ par une des suites de références (n^α, $\ln(n)$, $n!\ldots$).

Par exemple, pour la fonction `factorielle_rec`, on trouve $C(n) = 1 + C(n-1)$: en effet, on effectue une multiplication et l'appel récursif au rang $n-1$. On a donc $C(n) = n$. On obtient la même complexité que pour la version récursive.

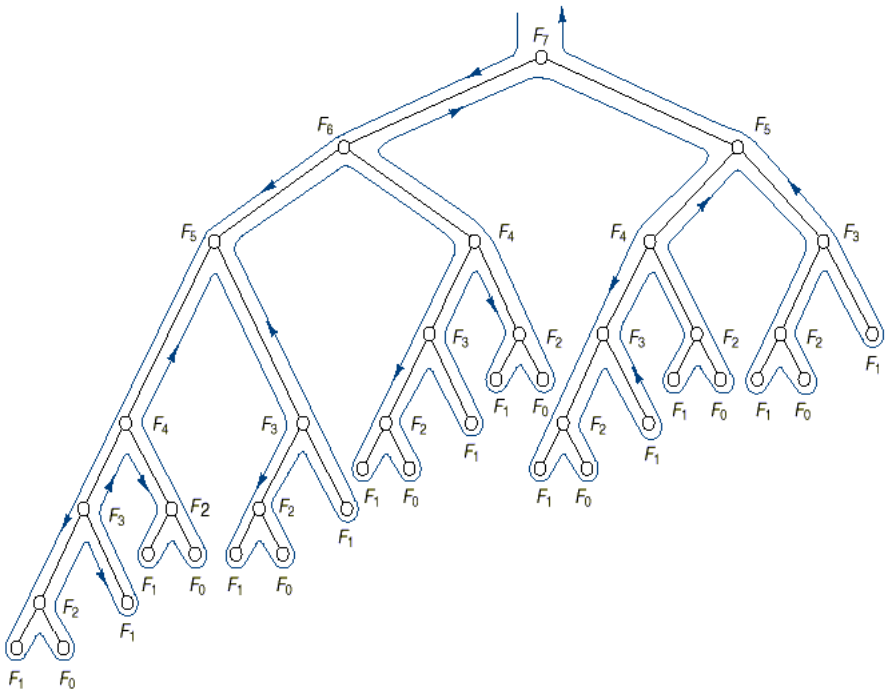
Donnons un exemple plus complexe : la suite de Fibonacci. Elle est définie par $u_0 = u_1 = 1$ et, pour tout $n \in \mathbb{N}$: $u_{n+2} = u_{n+1} + u_n$. La version itérative est la suivante :

```
def fibo(n):
    u=1 # u représente u_k
    v=1 #v représente u_{k+1}
    for k in range(n-1):
        u,v=v,u+v  # affectation simultanée en Python
    return v
\end{pyton}
La récurrence d'ordre $2$ oblige l'utilisation de deux variables
   \verb+u+ et \verb+v+, ce qui dégrade la lisibilité du
   programme. La version \underline{récursive} est plus simple:
\begin{python}
def fibo_rec(n):
    if n==0 or n==1:
        return 1
    else:
        return fibo_rec(n-1)+fibo_rec(n-2)
```

Si note $A(n)$ l'occupation mémoire de `fibo_rec(n)`, on trouve que $A(n) = A(n-1) + A(n-2)$. On montre alors qu'il existe un constante $C > 0$ telle que :

$$A(n) \underset{n\to+\infty}{\sim} C.\left(1+\frac{\sqrt{5}}{2}\right)^n$$

ce qui est très mauvais (pour la version itérative $A(n) = 2$ et $C(n) = n-1$). En effet, certains termes de la suite sont calculés un très grand nombre de fois (alors qu'une fois suffirait). On peut le visualiser sur la figure suivante, ou le calcul de u_6 a demandé de calculer 8 fois u_2.

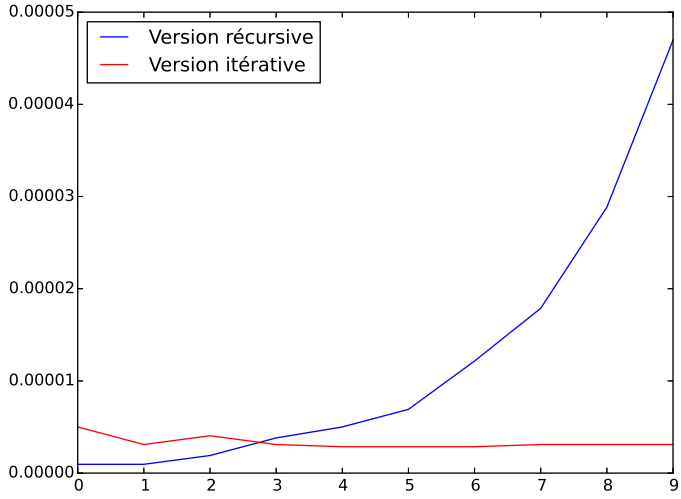


Pour résoudre ce problème, il faudrait enregistrer dans un tableau les termes déjà calculés : cette technique s'appelle la *mémoïsation*.

Grâce à l'instruction `time()` du module `time`, on peut comparer les vitesse d'exécution des versions itératives et récursives :

```
from __future__ import unicode_literals #accent dans les plots

from time import time

repetitions=10

iteratif=[]
for k in range(repetitions):
    t=time()
    fibo(k)  # on calcule les premiers termes de la suite
    t=time()-t
    iteratif.append(t)# on stocke les temps de calculs

recursif=[]
for k in range(repetitions):
    t=time()
    fibo_rec(k)
    t=time()-t
    recursif.append(t)

import matplotlib.pyplot as plt

abscisses=range(repetitions)
plt.plot(abscisses,recursif,color='b',label='Version récursive')
plt.plot(abscisses,iteratif,color='r',label='Version itérative')
plt.legend(loc=2) # en haut à gauche
```

On obtient :



2.6 Version récursive des algorithmes vus en première année

2.6.1 Recherche d'un élément dans une liste triée

On dispose d'une liste `l` de *de nombres déjà triée dans l'ordre croissant*. On veut déterminer si un élément `x` appartient à cette liste et donner la position d'un élément égal à `x` (ne pas perdre de vue qu'il peut en exister plusieurs).

```
1  def recherche(l,x):
2      if len(l)==0:   # close d'arrêt
3          return -1
4      else:
5          k=len(l)/2 # Python 2.7
6          if l[k]==x:   # x est au milieu de la liste
7              return k
8          elif l[k]>x: # on cherche à gauche
9              return recherche(l[:k],x)
10         else:
11             test=recherche(l[k+1:],x) # à droite
12             if test==-1: # non trouvé
13                 return -1
14             else: #trouvé
15                 return test+k+1 # correction de la position
```

Dans ce cas, on sait à l'avance si la recherche doit se faire dans la partie gauche ou dans la partie droite, contrairement au cas précédent.

Si on note $C(n)$ le nombre de tests booléens, dans le pire des cas, pour une liste de taille n, on a $C(n) = 4 + C(n/2)$ et donc $C(n) = \mathcal{O}\big(\log_2(n)\big)$. Cette fois on obtient une nette amélioration par rapport à la version itérative.

2.6.2 Algorithme de dichotomie

Le but est de trouver une solution approchée de l'équation $f(x) = 0$ sur un segment $[a, b]$. Pour cela on se base sur le théorème des valeurs intermédaires : si f est continue sur $[a, b]$, et si $f(a)$ et $f(b)$ sont de signes contraires (au sens large), alors f s'annule au moins une fois sur $[a, b]$.

L'algorithme consiste à couper le segment en son milieu $c = \dfrac{a+b}{2}$. Si $f(a)$ et $f(c)$ sont de signes contraires, on cherche la solution dans $[a, c]$, sinon on la cherche dans $[c, b]$. On réitère ensuite ce processus. Comme on divise à chaque fois l'intervalle en 2, on arrive au bout de n étapes à un intervalle $[a', b']$ de longueur $\dfrac{b-a}{2^n}$: a' et b' sont alors des valeurs approchées de la solution cherchée, avec une précision de $\dfrac{b-a}{2^n}$. Si on souhaite une précision de $\varepsilon > 0$ donnée, il suffit de choisir le nombre n d'itérations tel que $\dfrac{b-a}{2^n} < \varepsilon$.

L'algorithme s'écrit naturellement de manière récursive :

```
1  def dicho(f,a,b,eps):
2      assert f(a)*f(b)<=0, 'on ne sait pas si f s''annule'
3      if b-a<eps: # close d'arrêt
4          return a
5      else:
6          c=(a+b)/2. # Python 2.7
7          if f(a)*f(c)<0:
8              return dicho(f,a,c,eps) # recherche dans [a,c]
9          else:
10             return dicho(f,c,b,eps) # recherche dans [c,b]
```

Le nombre d'itérations est le même que pour la version itérative : c'est le plus petit entier n tel que $\dfrac{b-a}{2^n} < \varepsilon$.

3 Algorithmes de tri

Dans cette section, nous allons présenter plusieurs algorithmes permettant de trier une liste (ou un tableau) de nombres.

3.1 Généralités sur les algorithmes de tri par comparaisons

Nous donnons un résultat très général.

Théorème. Si c_n est le nombre de comparaisons effectuées par un algorithme de tri, pour trier une liste de n nombres, alors $n.\log_2(n) = \mathcal{O}(c_n)$.

Pour cette raison, un algorithme de tri sera dit *optimal* lorsque $c_n = \mathcal{O}\big(\log_2(n)\big)$.

⚠ Lorsqu'on ajoute des hypothèses sur les objets à trier, il existe des algorithmes plus rapides !

Donnons ensuite une notion qui va permettre de comparer les performances des différents algorithmes de tri : on dit qu'un tri est fait *en place* si il ne nécessite pas l'utilisation d'une liste (ou tableau) auxiliaire. Dans ce cas, la complexité spatiale (= occupation en mémoire) de l'algorithme est constante.

3.2 Tri par insertion (insertion sort)

3.2.1 Présentation de l'algorithme

C'est le tri naturellement utilisé pour trier des cartes : prendre les cartes (mélangées) une à une sur la table, et former une main en insérant chaque carte à sa place.

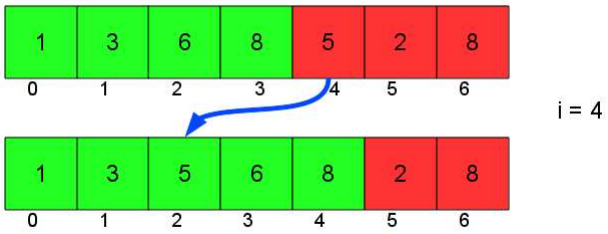
Pour une liste de nombres `t` le principe est le suivant :

• On parcourt la liste de gauche à droite.

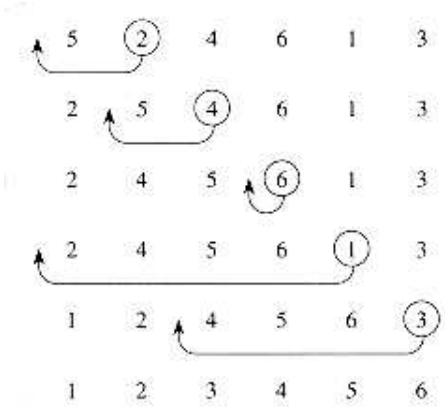
• Au moment où on considère le k-ème élément (noté `t[k-1]` en Python), les $k-1$ éléments qui le précédent sont déjà triés entre eux.

• On insère alors le k-ième élément dans la portion du tableau qui le précède (notée `t[:k]`), de telle sorte que les k premiers éléments soient à leur tour triés : pour cela on parcourt le tableau de droite à gauche et tant que les éléments sont plus grand que `t[k-1]`, on les décale d'une position vers la droite (ce qui conserve le tri précédent) ; dès qu'un élément plus petit est trouvé, on place juste à côté la valeur de `t[k-1]`.

La figure suivante donne un exemple d'insertion :



Et sur une liste qu'on trie entièrement :



Le code Python est le suivant :

```
def tri_insertion(t):
# attention une liste Python est un paramètre passé par référence
    for k in range(1,len(t)):
        j=k-1
        x=t[k]
        while j>=0 and t[j]>x:  # et non pas l[j]>x and j>=0
            t[j+1]=t[j] # décalage à droite
            j=j-1
        t[j+1]=x # insertion
# pas besoin de return t puisque t a été passée par référence
# la fonction ne renvoie rien en sortie
```

Par exemple :

```
>>> t=[5,2,4,6,1,3]
>>> tri_insertion(t)
>>> t
[1, 2, 3, 4, 5, 6]
```

À la place de `t` est un *paramètre passé par référence*, on dit parfois que `t` est une *variable modifiée en place*.

On remarque que le tri par insertion est tri fait *en place*, sa complexité spatiale est donc faible.

En utilisant la fonction `enumerate` on peut écrire une version peut-être plus lisible du programme précédent. L'instruction `for k,x in enumerate(t)` parcours la liste `t` de telle sorte que `k` prenne successivement les valeurs 0, 1, ..., `len(t)`, et simultanément `x` prend les valeurs `t[0]`, `t[1]`, ..., `t[len(t)]`.

```
def tri_insertion2(t):
    for k,x in enumerate(t): # x=t[k]
        j=k-1
        while j>=0 and t[j]>x:
            t[j+1]=t[j]
            j=j-1
        t[j+1]=x
```

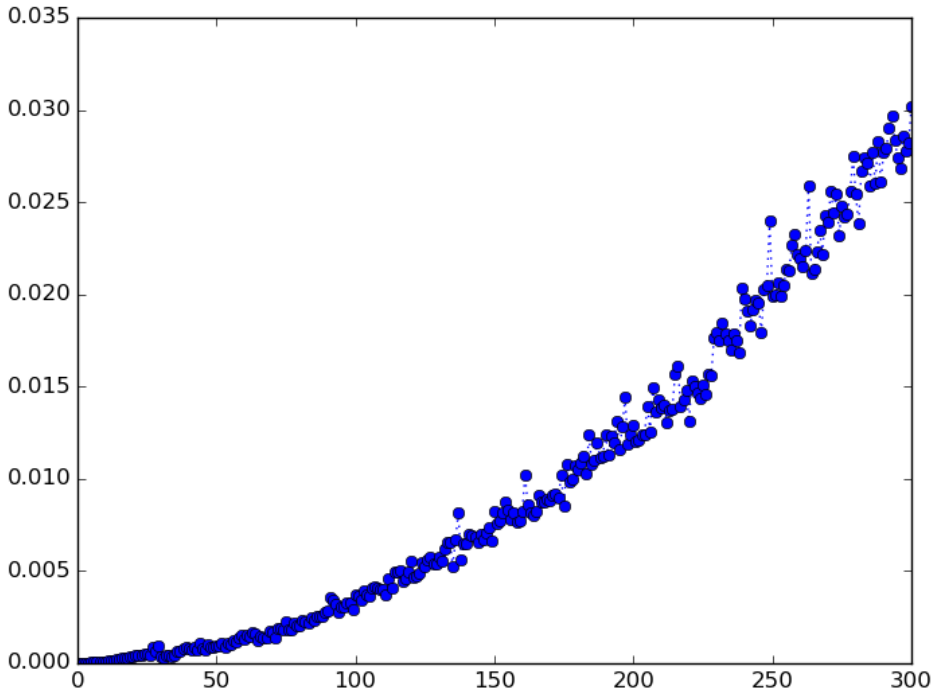
3.2.2 Complexité

Avec le module `time`, on peut évaluer empiriquement la complexité en regardant le temps de calcul pour trier des listes de nombres de taille variant de 1 à 300. Pour créer des liste aléatoires on utilise la fonction `randint` du sous-module `random` du module `numpy`.

```
from time import time
import numpy.random as rd

def complexite_insertion(n):
    ordonnees=[]
    for k in range(n+1):
        liste=rd.randint(0,k+1,k+1)
          # k+1 nombres entiers entre 0 et k
```

```
        temps=time()
        tri_insertion(liste)
        temps=time()-temps
        ordonnees.append(temps)
    abscisses=range(n+1)
    plt.plot(abscisses,ordonnees,linestyle=':',marker='o')
    plt.show()
```

On obtient la courbe suivante qui suggère une complexité en $\mathcal{O}(n^2)$, où n est la taille de la liste :



Nous allons donc évaluer la complexité de l'algorithme en fonction de la taille n d'une liste donnée en argument d'entrée.

Dans le *pire des cas*, la boucle `while` s'effectue jusqu'à ce que la variable j prenne la valeur -1, et le nombre de comparaisons est égal k. Avec la boucle `for` on obtient que le nombre total de comparaison est :

$$1+2+\cdots+k+\cdots+n-1=\frac{n(n-1)}{2}$$

Si on note C_n la complexité temporelle (en ne prenant en compte que les comparaisons) dans le pire des cas, on a donc :

$$C_n \underset{n\to+\infty}{\sim} \frac{n^2}{2}$$

Remarquez que cette situation correspond au cas d'un tableau trié au départ dans l'ordre décroissant.

Dans le *meilleur des cas*, la boucle `while` ne s'effectue pas, et le nombre de comparaisons est égal 1. Avec la boucle `for` on obtient que le nombre total de comparaison est :

$$1+1+\cdots+k+\cdots+1=n-1$$

Si on note c_n la complexité temporelle (en ne prenant en compte que les comparaisons) dans le meilleur des cas, on a donc :

$$c_n \underset{n\to+\infty}{\sim} n$$

Remarquer que cette situation correspond au cas d'un tableau déjà trié au départ dans l'ordre croissant.

Ce tri n'est donc pas optimal mais il est tout de même beaucoup utilisé. Empiriquement, il est établi qu'il est le plus rapide sur des données presque triées, ou sur des listes (ou tableaux) de petite taille.

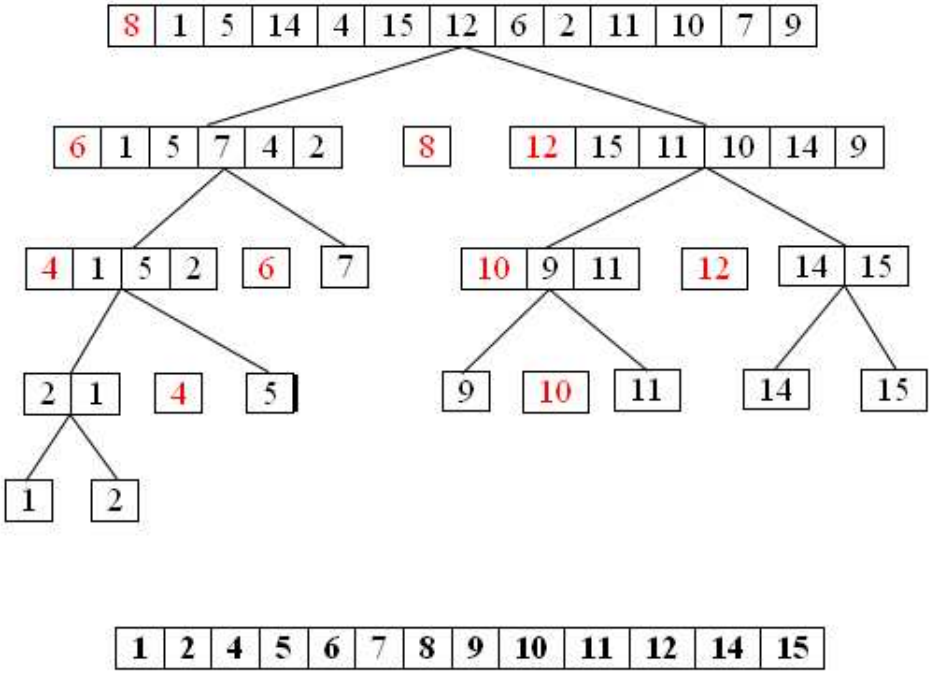
3.3 Tri rapide (quick sort)

3.3.1 Présentation de l'algorithme

On choisit dans la liste un élément particulier (en général le premier), noté `x`. On crée alors deux sous-tableaux : un contenant les éléments strictement inférieurs à `x`, et l'autre contenant les éléments supérieurs ou égaux à `x`. On relance le tri de manière récursive sur ces deux sous-tableaux qui deviennent donc des listes triées, et il ne reste plus qu'à les concaténer, en intercalant le pivot entre les deux. En effet, on remarque qu'on reforme ainsi la liste initiale mais sous forme triée.

Cet algorithme est un exemple d'une méthode générale utilisée en récursivité, appelée « diviser pour régner » : on divise le problème initial en deux sous-problèmes qu'on sait résoudre, et on revient ensuite au problème initial. Dans le cas du tri rapide, les appels récursifs sont lancés jusqu'à tomber sur des listes de taille 1 ou 0, qui sont déjà triées, et il ne reste plus qu'à les concaténer dans le bon ordre.

La figure suivante illustre ce tri sur un exemple :



Nous donnons une première version naïve :

```
def tri_rapide(t):
    if len(t)<=1: # close d'arrêt: tableau vide
        return t  # ou avec un seul élément
    else:
        pivot=t[0] # choix arbitraire du pivot
        plus_petits=[ x for x in t if x<pivot ]
        plus_grands=[ x for x in t[1:] if x>=pivot ]
        return tri_rapide(plus_petits)+[pivot]+tri_rapide(plus_grands)
            # concaténation et renvoi du résultat
```

Si on ne souhaite pas utiliser les listes par compréhension, on peut écrire :

```
def tri_rapide2(t):
    if len(t)<=1: # close d'arrêt: tableau vide
        return t  # ou avec un seul élément
    else:
        pivot=t[0] # choix arbitraire du pivot
        plus_petits=[]
        plus_grands=[]
        for x in l[1:]:
            if x<pivot:
                plus_petits.append(x)
            else:
                plus_grands.append(x)
        return tri_rapide(plus_petits)+[pivot]+tri_rapide(plus_grands)
            # concaténation et renvoi du résultat
```

Ce premier algorithme naïf illustre très bien la méthode utilisée, grâce aux liste `plus_petits`

et `plus_grands`. Mais il a un gros défaut : le tri ne se fait pas en place. Au cours de l'exécution il se crée beaucoup de tableaux auxiliaires et la complexité spatiale est alors très importante.

Nous allons donner une version plus compliquée, mais pour laquelle le tri se fera en place : il faut réussir à partitionner selon le pivot, mais sans créer de nouvelle liste auxiliaire, la seule possibilité étant de permuter les éléments de la liste deux à deux.

Nous allons commencer par créer une fonction `partition(t)` qui va partitionner une liste `t` en prenant le premier élément `t[0]` comme pivot.

Pour cela, on parcourt la liste de gauche à droite (à partir de t[1]), de telle sorte que lorsqu'on arrive à l'élément `t[i]`, les i premiers éléments soient partitionnés, c'est-à-dire que :
• leur partie gauche jusqu'à un indice appelé `position` est formée des éléments plus petits (au sens strict) que le pivot,
• leur partie droite (après l'indice `position`) est formée des éléments plus grands (au sens large).

0	…	position	…	i	…	len(t)-1
pivot	<pivot	<pivot	≥ pivot	t[i]	?	

On compare alors `t[i]` avec le pivot :
— s'il est plus grand (au sens large), on ne fait rien ;
— s'il est plus petit (au sens strict), on le permute avec l'élément `t[position+1]` et on incrémente la variable `position` de +1.

Une fois arrivé à la fin de la liste, on permute les éléments `t[0]` et `t[position]` : le tableau est alors partitionné comme souhaité. On revoie en sortie la valeur de la variable `position`. Comme une liste (ou un tableau) est un paramètre passé par référence, elle est modifiée par la fonction comme s'il était une variable globale.

```
1  def partition(t):
2      assert t!=[],'on ne peut pas partitionner la liste vide'
3      pivot=t[0]
4      position=0
5      for i in range(1,len(t)):
6          if t[i]<pivot:
7              t[i],t[position+1]=t[position+1],t[i]
8              position+=1
9      if position!=0:
10         t[0],t[position]=t[position],t[0]
11     return position
```

Pour lancer le tri récursif, nous allons devoir appliquer cette fonction sur une portion de la liste, et non la liste entière. On adapte donc notre fonction pour qu'elle ne partitionne que la portion `t[a:b]` (formée des `t[i]` pour t allant de a à $b-1$). Le reste de la liste n'est pas modifié.

```
1  def partition(t,a,b):
2      assert a<b,'on ne peut pas partitionner la liste vide'
3      pivot=t[a]
```

```
    position=a
    for i in range(a+1,b):
        if t[i]<pivot:
            t[i],t[position+1]=t[position+1],t[i]
            position+=1
    if position!=a:
        t[a],t[position]=t[position],t[a]
    return position
```

On peut écrire le programme principal du tri rapide en place. Lui aussi doit pouvoir se lancer sur des portions de liste, il aura donc trois arguments comme la fonction `partition` précédente. En sortie, il ne renvoie rien mais la liste a été modifiée.

```
def tri_rapide2(t,a,b):
    if a<b:
        position=partition(t,a,b)
        tri_rapide2(t,a,position)
        tri_rapide2(t,position+1,b)
```

Pour trier toute la liste `t`, il faut lancer `tri_rapide2(t,0,len(t))`. Comme ce n'est pas très naturel, on peut créer une nouvelle fonction avec un seul argument d'entrée.

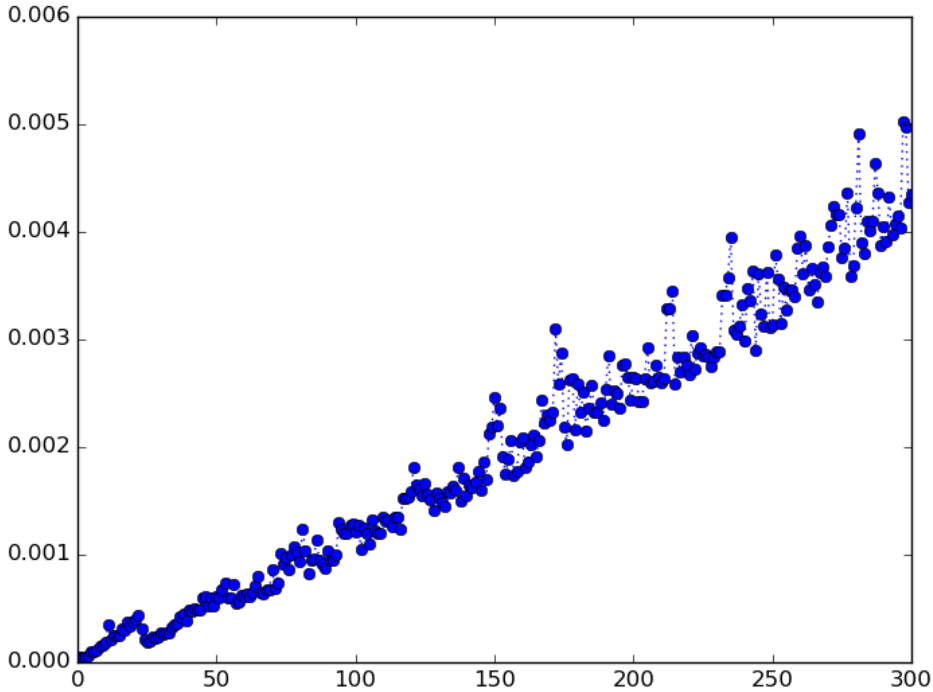
```
def quicksort(t):
    tri_rapide2(t,0,len(t))
```

3.3.2 Complexité

Comme pour le tri par insertion, on peut évaluer empiriquement la complexité en regardant le temps de calcul pour trier des listes de nombres de taille variant de 1 à 300.

```
from time import time
import numpy.random as rd

def complexite_qs(n):
    ordonnees=[]
    for k in range(n+1):
        liste=rd.randint(0,k+1,k+1)
          # k+1 nombres entiers entre 0 et k
        temps=time()
        quicksort(liste)
        temps=time()-temps
        ordonnees.append(temps)
    abscisses=range(n+1)
    plt.plot(abscisses,ordonnees,linestyle=':',marker='o')
    plt.show()
```

On obtient la courbe suivante qui suggère une complexité un peu plus que linéaire, où n est la taille de la liste :



Nous allons donc évaluer la complexité de l'algorithme en fonction de la taille n d'une liste donnée en argument d'entrée, et nous allons voir qu'elle n'est pas linéaire mais en $\mathcal{O}(n\log_2(n))$.

Dans le *pire des cas*, le résultat de la fonction `partition` place systématiquement le pivot en début ou en fin de tableau. Les appels récursifs se font donc sur un tableau vide et un tableau de taille $n-1$. Avant de lancers les appels récursifs, l'appel à la fonction `partition` effectue $n-1$ comparaisons entre les éléments du tableau.
Si on note C_n la complexité temporelle (en ne prenant en compte que les comparaisons) dans le pire des cas, on a donc :

$$C_n = n-1+C_{n-1}$$

On en déduit facilement que :

$$C_n = (n-1)+(n-2)+\cdots+1+0+C_0 = \frac{n(n-1)}{2}+C_0$$

donc :

$$C_n \underset{n\to+\infty}{=} \mathcal{O}\left(n^2\right)$$

Remarquer que cette situation correspond au cas d'un tableau déjà trié au départ (dans n'importe quel ordre).

Dans le *meilleur des cas*, le résultat de la fonction `partition` place systématiquement le pivot en milieu de de tableau. Les appels récursifs se font donc sur deux tableaux taille $\left\lfloor\frac{n}{2}\right\rfloor$. Avant de lancers les appels récursifs, l'appel à la fonction `partition` effectue $n-1$ comparaisons entre les éléments du tableau.
Si on note c_n la complexité temporelle (en ne prenant en compte que les comparaisons) dans le meilleur des cas, on a donc :

$$c_n = n-1+2.c_{\lfloor\frac{n}{2}\rfloor}$$

Les calculs suivant prouvent que :

$$c_n \underset{n\to+\infty}{=} \mathcal{O}\left(n.\log_2(n)\right)$$

Cas 1. On commence par le cas où n est une puissance de 2 : $n = 2^p$ pour $p \in \mathbb{N}$.
Pour simplifier les notations, posons $x_p = c_{2^p}$.
On a donc la relation de récurrence :

$$x_p = c_{2^p} = 2^p - 1 + 2.c_{2^{p-1}} = 2^p - 1 + 2.x_{p-1}$$

On en déduit que :

$$\begin{aligned} x_p &= \left(2^p - 1\right) + 2.\left(2^{p-1} - 1\right) + 2^2.\left(2^{p-2} - 1\right) + \cdots + 2^{p-1}.\left(2 - 1\right) + 2^p.x_0 \\ &= p.2^p - \frac{1-2^p}{1-2} + 2^p.x_0 \\ &= p.2^p + 1 - 2^p + 2^p.c_0 \end{aligned}$$

Cas 2. On revient au cas général. Pour cela, on fait l'hypothèse raisonnable que la suite $(c_n)_{n\in\mathbb{N}}$ est *croissante* (plus la taille des données à trier est important, plus le nombre de comparaison est important).
On pose $p = \lfloor \log_2(n) \rfloor + 1$ de sorte que $\log_2(n) \leq p$ et donc :

$$n < 2^p$$

On en déduit que :

$$c_n \leq c_{2^p} = p.2^p + 1 - 2^p + 2^p.c_0$$

mais on a aussi $p \leq \log_2(n) + 1$ donc :

$$c_n \leq \left(\log_2(n) + 1\right).2^{\log_2(n)+1} + 1 - 2^{\log_2(n)+1} + 2^{\log_2(n)+1}.c_0 = 2n\Big(\left(\log_2(n) + 1\right) - 1 + c_0\Big) - 1$$

et finalement :

$$c_n \underset{n\to+\infty}{=} \mathcal{O}\left(\log_2(n)\right)$$

Ce tri est donc optimal dans le meilleur des cas mais il ne l'est pas dans le pire des cas. Mais il se trouve qu'en pratique c'est l'algorithme le plus rapide, il est donc empiriquement le plus performant. Cette différence vient du fait que nous avons seulement dénombré les comparaisons entre éléments du tableau à trier, alors qu'il se passe beaucoup d'autre choses lors de l'exécution de l'algorithme (affectations etc...).

3.4 Tri fusion (merge sort)

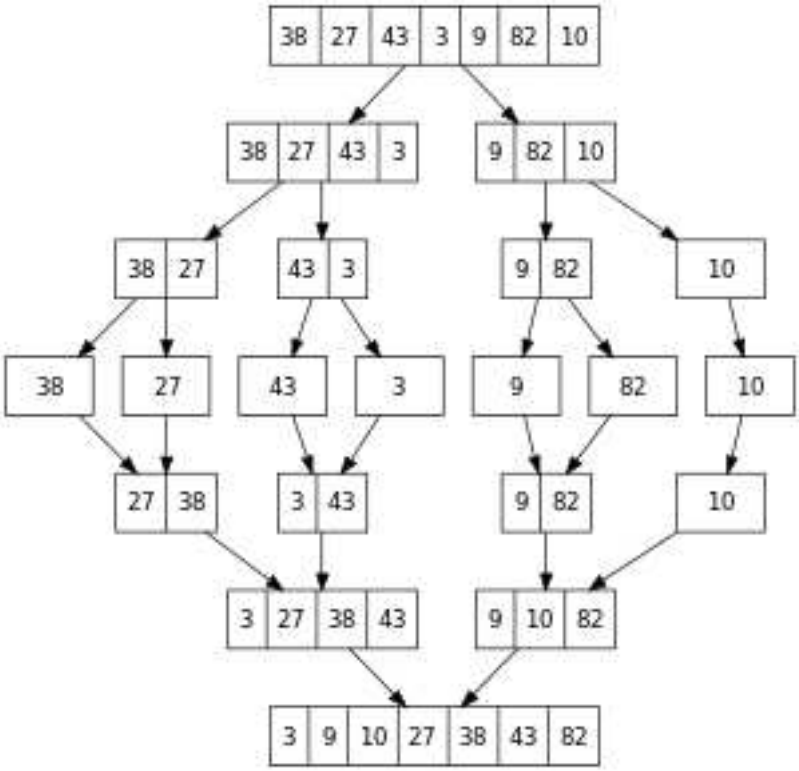
3.4.1 Présentation de l'algorithme

Nous allons encore créer un programme récursif utilisant la méthode « diviser pour régner ».

Pour le tri rapide, l'idée est de faire en sorte que les deux parties à concaténer donne directement le tableau trié (c'est tout l'intérêt de la fonction `partition`), le défaut étant qu'on ne coupe pas toujours en deux parties égales. Pour le tri fusion, le principe est différent. On

coupe le tableau en deux morceaux toujours égaux, qu'on trie, puis reste l'étape délicate appelée *fusion*. Elle consiste à fusionner ces deux tableaux de telle sorte qu'on obtienne un tableau trié.

La figure suivante illustre ce tri sur un exemple :



La fonction principale est la fonction `fusion` qui permet de « fusionner » deux tableaux triés en un tableau trié. Pour cela on procède manière récursive : le plus petit élément des deux tableaux est placé en première position du nouveau tableau, il reste ensuite à fusionner les deux tableaux privés de cet élément.

```
1  def fusion(t1,t2): # fusion de deux tableaux déjà triées
2      if t1==[]: # première close d'arrêt
3          return t2
4      if t2==[]: # deuxième close d'arrêt
5          return t1
6      if t1[0]<t2[0]:
7    # pour fusionner on place le plus petit élément des 2 tableaux
       en premier
8    # c'est le plus petit élément global tableaux triés
9          return [t1[0]]+fusion(t1[1:],t2) # appel récursif
10     else:
11         return [t2[0]]+fusion(t1,t2[1:]) # appel récursif
```

La fonction qui effectue le tri fusion est très simple :

```
1  def trifus(t):
2      if len(t)<=1:  # close d'arrêt
3          return t
4      else:
5          return fusion(trifus(t[:len(t)/2]),trifus(t[len(t)/2:]))
6    # en Python 2.7 / est la division euclidienne
```

On voir tout de suite l'avantage de cet algorithme : la lecture est très simple, et le fait de toujours couper le tableau en deux parties égales va limiter le nombre de comparaisons.

Par contre la complexité spatiale est très mauvaise, on utilise une fonction récursive qui appelle elle-même une fonction récursive, et les opérations ne sont faites pas en place.

On peut donc espérer améliorer l'algorithme en le programmant en place. Pour cela, si on dispose d'un tableau dont les deux moitiés sont triées, il faut être capable de les fusionner dans créer de tableau auxiliaire. Nous allons utiliser des *permutations circulaires*. Si on se donne un tableau $[x_1, x_2, \ldots, x_{n-1}, x_n]$, il devient après permutation circulaire :

$$[x_n, x_1, x_2, \ldots, x_{n-1}]$$

```
1 def perm_circ(t):
2     for k in range(len(t)-1,0,-1):
3         t[k],t[k-1]=t[k-1],t[k]
4  # permutation circulaire: le dernier élément devient le premier
5  # et les autres sont décalés vers la droite
```

Et on adapte pour lancer sur le sous-tableau `t[i:j+1]`.

```
1 def perm_circ(t,i,j):
2     for k in range(j,i,-1):
3         t[k],t[k-1]=t[k-1],t[k]
4  # permutation circulaire: le dernier élément devient le premier
5  # et les autres sont décalés vers la droite
```

Ensuite, supposons qu'on dispose d'un tableau `t` et d'un indice `m` dans $[\![0,$`len(t)`$]\!]$, de telle sorte que les sous-tableaux `t[:m]` et `t[m:]` soient triés. Pour réaliser la fusion, nous allons parcourir le tableau de gauche à droite de sorte que lorsqu'on arrive à l'élément `t[i]` :
- `t[:i]` est un sous-tableau d'un tableau trié `t[:j]` avec $j > i$ et $j \geq m$,
- chaque élément de `t[:i]` est inférieur ou égal à x,
- le tableau `t[j:]` est trié.

…	i	…	j	…	len(t)-1
≤t[j]	t[i]		t[j]	?	
trié			trié		

On compare alors `t[j]` avec `t[i]` :
- si `t[j]≥t[i]`, on incrément i de 1 ;
- si `t[j]<t[i]`, on effectue une permutation circulaire sur le tableau `t[i:j+1]`, puis on incrémente i et j de 1.

On arrête lorsqu'on atteint la fin de la liste, ie $j =$`len(t)`, ou lorsque $i = j$ puisque dans ce cas le tableau entier est déjà trié (les deux parties triées se concatènent en une partie triée) !

```
def fusion(t,m):
    # t[:m] et t[m:] sont supposés triés
    i=0  # prochain élément à traiter
    j=m
    while j<b and i<j:
        if t[j]>=t[i]:
            i+=1
        else:
            perm_circ(t,i,j)  # permutation circulaire
            i+=1
            j+=1
```

Et on adapte au cas d'un sous-tableau `t[a:b]`.

```
def fusion(t,a,m,b):
    # t[a:m] et t[m:b] sont supposés triés
    i=a  # prochain élément à traiter
    j=m
    while j<b and i<j:
        if t[j]>=t[i]:
            i+=1
        else:
            perm_circ(t,i,j)  # permutation circulaire
            i+=1
            j+=1
```

On peut écrire le programme principal du tri fusion en place. Lui aussi doit pouvoir se lancer sur des portions de liste, il aura donc trois arguments comme la fonction `fusion` précédente. En sortie, il ne renvoie rien mais la liste a été modifiée.

```
def trifus2(t,a,b):  # trie t[a:b]
    if b>a+1:
        m = (a+b)/2 # Python 2.7
        trifus2(T,a,m)
        trifus2(T,m,b)
        fusion(T,a,m,b)
```

Pour trier toute la liste `t`, il faut lancer `trifus2(t,0,len(t))`. Comme ce n'est pas très naturel, on peut créer une nouvelle fonction avec un seul argument d'entrée.

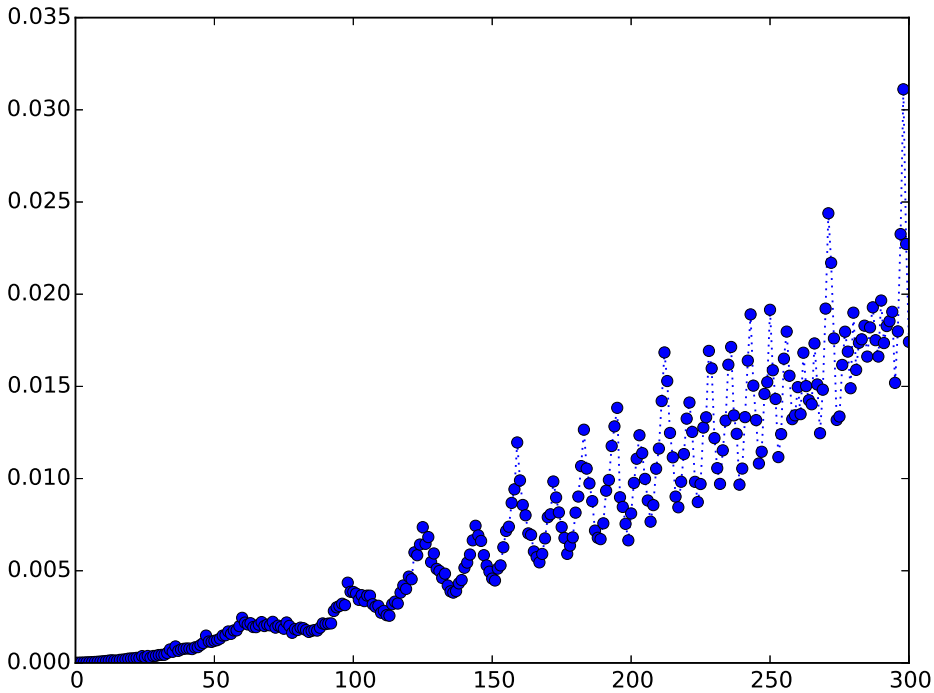
```
def tri_fusion(t):
    trifus2(t,0,len(t))
```

3.4.2 Complexité

On peut évaluer empiriquement la complexité en regardant le temps de calcul pour trier des listes de nombres de taille variant de 1 à 300.

```python
from time import time
import numpy.random as rd

def complexite_ms(n):
    ordonnees=[]
    for k in range(n+1):
        liste=rd.randint(0,k+1,k+1)
          # k+1 nombres entiers entre 0 et k
        liste=list(liste)
 # il ne faut pas de tableaux numpy mais des listes à cause du comportement de +
        temps=time()
        tri_fusion(liste)
        temps=time()-temps
        ordonnees.append(temps)
    abscisses=range(n+1)
    plt.plot(abscisses,ordonnees,linestyle=':',marker='o')
    plt.show()
```

On obtient la courbe suivante qui suggère la même complexité que pour le tri rapide : $\mathcal{O}\big(n.\log_2(n)\big)$.



Nous allons donc évaluer la complexité de l'algorithme en fonction de la taille n d'une liste donnée en argument d'entrée.

Dans le *pire des cas* (tableau trié l'envers), lors de l'appel de la fonction `fusion`, `j` varie de $\left\lfloor \frac{n}{2} \right\rfloor$ à $n-1$, et i prend ses valeurs entre 0 et $n-2$, donc au total l'algorithme effectue $\left\lfloor \frac{n}{2} \right\rfloor$

comparaisons. En gardant les notations des paragraphes précédents, on a donc :

$$C_n = \left\lfloor \frac{n}{2} \right\rfloor + 2.C_{\left\lfloor \frac{n}{2} \right\rfloor}$$

et les mêmes calculs que pour le tri fusion donnent :

$$C_n \underset{n\to+\infty}{=} \mathcal{O}\left(n.\log_2(n)\right)$$

Dans le *meilleur des cas*(tableau trié), `j` ne varie pas et i varie de 0 à $\left\lfloor \frac{n}{2} \right\rfloor - 1$, donc au total l'algorithme effectue $\left\lfloor \frac{n}{2} \right\rfloor$ comparaisons. En gardant les notations des paragraphes précédents, on a donc :

$$c_n = \left\lfloor \frac{n}{2} \right\rfloor + 2.c_{\left\lfloor \frac{n}{2} \right\rfloor}$$

et les mêmes calculs que pour le tri fusion donnent :

$$c_n \underset{n\to+\infty}{=} \mathcal{O}\left(n.\log_2(n)\right)$$

Ce tri est donc optimal ! Mais ce n'est vrai qu'en prenant en compte le nombre de comparaisons. La version présentée a l'avantage d'être en place, donc la complexité spatiale est constante. Par contre le nombre d'affectations est très grand, et il paraît peu rigoureux de ne pas en tenir compte.

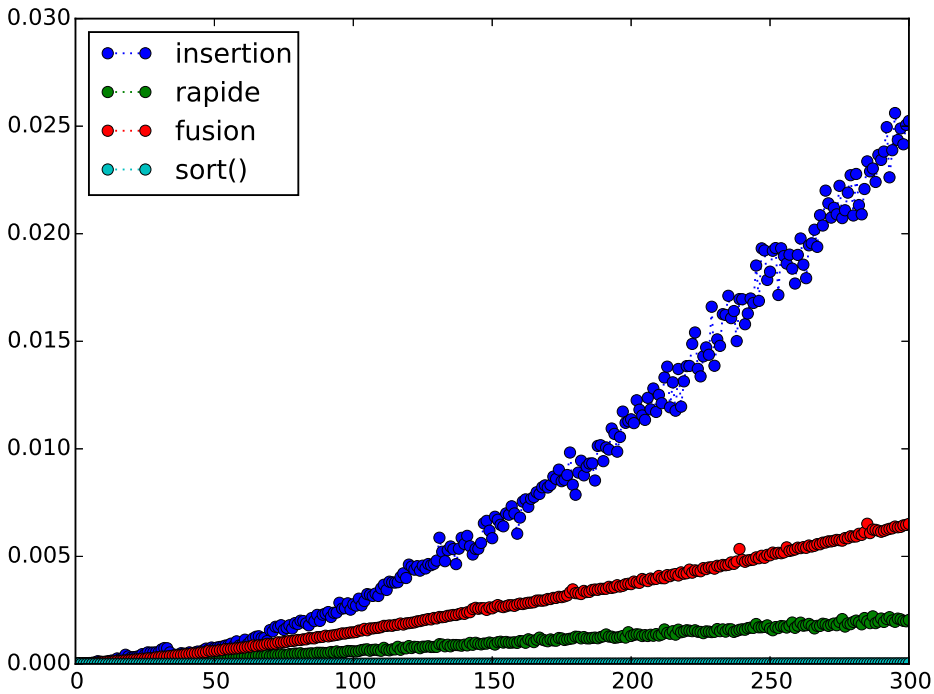
3.5 Comparaison empirique des trois algorithmes de tri

Nous allons comparer la rapidité des trois algorithmes de tri sur des tableaux aléatoires de taille comprise entre 0 et 300 éléments ; comme on ne s'intéresse pas à la complexité spatiale, nous utilisons les versions non en place qui sont plus simples. Python dispose aussi de la méthode `sort()` : si `t` est une liste alors l'instruction `t.sort()` trie la liste en place.

```
1  def comparaison_tri(n):
2      ordonnees=[]
3      for k in range(n+1):
4          liste=rd.randint(0,k+1,k+1)
5          t=time()
6          tri_insertion(liste)
7          t=time()-t
8          ordonnees.append(t)
9      abscisses=range(n+1)
10     plt.plot(abscisses,ordonnees,linestyle=':',marker='o',label='
           insertion')
11     ordonnees=[]
12     for k in range(n+1):
13         liste=rd.randint(0,k+1,k+1)
14         t=time()
15         tri_rapide(liste)
16         t=time()-t
17         ordonnees.append(t)
18     plt.plot(abscisses,ordonnees,linestyle=':',marker='o',label='
           rapide')
```

```
19      ordonnees=[]
20      for k in range(n+1):
21          liste=rd.randint(0,k+1,k+1)
22          t=time()
23          trifus(liste)
24          t=time()-t
25          ordonnees.append(t)
26      plt.plot(abscisses,ordonnees,linestyle=':',marker='o',label='
           fusion')
27      plt.legend(loc=2)
28      plt.show()
```

On obtient les courbes suivantes.



On retrouve qu'empiriquement le tri rapide est le meilleur en temps. La méthode `sort()` de Python paraît beaucoup plus efficace que les trois algorithmes présentés ; nous n'aborderons pas son fonctionnement (il faut simplement savoir qu'en pratique on mélange les algorithmes de tri).

3.6 Recherche de la médiane dans un tableau

Si on dispose d'un tableau `t=[t[0],t[1],...t[n-1]]` de nombres flottants de taille n, que l'on suppose *trié*, on définit la *médiane* de `t` ainsi :

- si n est impair, $n = 2p+1$. La médiane de `t` est l'élément `t[p]` avec $p = \left\lfloor \frac{n}{2} \right\rfloor$;
- si n est pair, $n = 2p$. Une médiane de `t` est n'importe quel élément de l'intervalle `[t[p-1],t[p][` avec $p = \left\lfloor \frac{n}{2} \right\rfloor$. On choisit souvent le milieu de cet intervalle : $\frac{t[p-1]+t[p]}{2}$.

Pour déterminer la médiane d'un tableau quelconque de nombres flottants, il suffit donc de le trier, puis d'appliquer la définition ci-dessus. La complexité minimale est donc celle de l'algorithme de tri, c'est-à-dire $\mathcal{O}\big(n.\log_2(n)\big)$ au minimum.

Il existe des algorithmes plus performants, appelés *algorithme de sélection*, qui permette de trouver le k-ième plus grand élément d'un tableau de nombre flottants, sans avoir à trier systématiquement tout le tableau. Leur complexité est en moyenne linéaire, ie en $\mathcal{O}(n)$.

Chapitre 3

Exercices de première année

1 Structure de données

1.1 Représentation des nombres

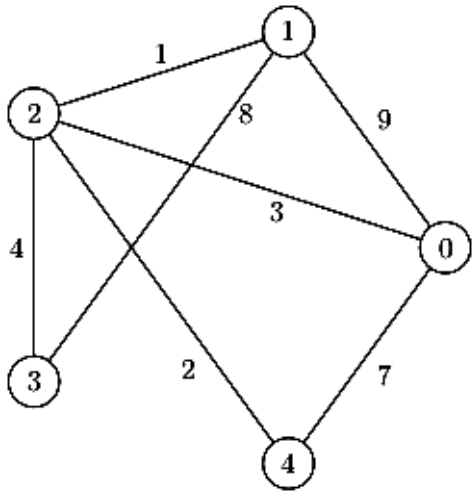
Exercice 1 (Suite récurrente linéaire instable) Dans cet exercice, on s'intéresse aux suites $(u_n)_{n\in\mathbb{N}}$ vérifiant la relation de récurrence linéaire :

$$\forall n \in \mathbb{N}, \qquad u_{n+2} = u_n - \frac{8}{3}u_{n+1} \qquad (R)$$

1. Soit u vérifiant (R). Exprimer, pour tout $n \in \mathbb{N}$, u_n en fonction de u_0 et u_1.
2. Donner une condition nécessaire et suffisante simple portant sur u_0 et u_1 pour que $(u_n)_{n\in\mathbb{N}}$ converge.
3. On s'intéresse dans un premier temps à la suite vérifiant (R) avec les conditions initiales $\begin{cases} u_0 = 1 \\ u_1 = 1 \end{cases}$
   (a) Écrire un programme prenant en entrée n et retournant la valeur de u_n (en utilisant la relation de récurrence, c'est-à-dire sans utiliser la formule établie à la première question).
   (b) Vérifier la solution obtenue en calculant puis représentant les 10 premiers termes (en reliant les points de coordonnées (n, u_n)).
4. On s'intéresse maintenant à la suite vérifiant (R) avec les conditions initiales $\begin{cases} u_0 = 3 \\ u_1 = 1 \end{cases}$
   (a) Écrire un programme prenant en entrée n et retournant la valeur de u_n (en utilisant la relation de récurrence, c'est-à-dire sans utiliser la formule établie à la première question).
   (b) Vérifier la solution obtenue en calculant puis représentant les 10 premiers termes.
   (c) Recommencer avec les 30 puis les 40 premiers termes.
   (d) Expliquer qualitativement le phénomène.
   (e) Plus précisément, expliquer pourquoi le phénomène est apparu pour ces valeurs de n.

1.2 Listes

Exercice 2 (Matrice d'adjacence d'un graphe) On considère le graphe G suivant, où le nombre situé sur l'arête joignant deux sommets est leur distance, supposée entière :



1. Construire la matrice $(M_{ij})_{0\leq i,j\leq 4}$, *matrice de distances* du graphe G, définie par : pour tous les indices i, j, M_{ij} représente la distance entre les sommets i et j, ou encore la longueur de l'arête reliant les sommets i et j.
   On convient que, lorsque les sommets ne sont pas reliées, cette distance vaut -1. La distance du sommet i à lui-même est, bien sûr, égale à 0.
2. Écrire une suite d'instructions permettant de dresser à partir de la matrice M la liste des voisins du sommet 4.
3. Écrire une fonction `voisins`, d'argument un sommet i, renvoyant la liste des voisins du sommet i.
4. Écrire une fonction `degre`, d'argument un sommet i, renvoyant le nombre des voisins du sommet i, c'est-à-dire le nombre d'arêtes issues de i.
5. Écrire une fonction `longueur`, d'argument une liste `L` de sommets de G, renvoyant la longueur du trajet décrit par cette liste `L`, c'est-à-dire la somme des longueurs des arêtes empruntées. Si le trajet n'est pas possible, la fonction renverra -1.

1.3 Chaînes de caractères

Exercice 3 (Cryptographie) Soit n un entier vérifiant $n \leq 26$. On souhaite écrire un programme qui code un mot en décalant chaque lettre de l'alphabet de n lettres.

Par exemple pour $n = 3$, le décalage sera le suivant :

avant décalage	a	b	c	d	...	...	x	y	z
après décalage	d	e	f	g	...	...	a	b	c

Le mot `oralensam` devient ainsi `rudohqvdp`.

1. Définir une chaîne de caractères contenant toutes les lettres dans l'ordre alphabétique (caractères en minuscule).

2. Écrire une fonction `decalage`, d'argument un entier n, renvoyant une chaîne de caractères contenant toutes les lettres dans l'ordre alphabétique, décalées de n, comme indiqué ci-dessus.
3. Écrire une fonction `indices`, d'arguments un caractère `x` et une chaîne de caractères `phrase`, renvoyant une liste contenant les indices de `x` dans phrase si `x` est une lettre de phrase et une liste vide sinon.
4. Écrire une fonction `codage` d'arguments un entier `n` et une chaîne de caractères `phrase`, renvoyant `phrase` codée avec un décalage de n lettres.
5. Comment peut-on décoder un mot codé ?

2 Modules

Exercice 4 (Algorithme du pivot de Gauss avec Numpy)

1. Écrire une fonction `pivot(m,j)` qui prend en entrée une matrice `m` (de taille $n \times p$) et un entier `j`, et donne en sortie un indice $i \geq j$ tel que $|m[i,j]| = \max\big(|m[j,j]|, |m[j+1,j]|, \ldots, |m[n-1,j]|\big)$.
   On rappelle qu'on choisit comme pivot en mathématiques un coefficient non nul de la colonne, mais qu'en informatique ceci est problématique, et on préfère donc choisir un coefficient maximum en valeur absolue.

2. Écrire une fonction `echange(m,i,j)` qui prend en entrée une matrice `m` et deux entiers `i` et `j`, et qui échange les lignes `i` et `j` dans la matrice `m`.
   Remarque : cette fonction ne donne rien en sortie.

3. Écrire une fonction `transvection(m,i,j,c)` qui prend en entrée une matrice `m`, deux entiers `i` et `j`, et un réel `c`, et qui effectue l'opération $L_i \longleftarrow L_i + c \times L_j$.
   Remarque : cette fonction ne donne rien en sortie.

4. Écrire une fonction `triangulaire(m)` qui prend en entrée une matrice `m` qui est transformée en une matrice triangulaire par l'algorithme de Gauss-Jordan.

5. En déduire une fonction `rang(m)` qui donne le rang d'une matrice `m`.
6. Pour finir écrire une fonction `resolution(A,B)` qui teste si le système linéaire $AX = B$ est de Cramer et le résout dans le cas échéant.

3 Algorithmique

3.1 Algorithmique des tableaux/listes et des chaînes de caractères

Exercice 5 (Recherche d'un élément dans une liste) Montrer la terminaison et la correction de l'algorithme de recherche d'un élément dans une liste.

Exercice 6 (Recherche d'un élément dans une liste : variantes) On souhaite rechercher un élément `x` dans une liste `t`.

1. Écrire une fonction `recherche_inverse(t,x)` qui détermine le *dernier indice* `i` tel que `t[i]=x`, et renvoie `i` s'il existe, `Flase` sinon.

2. Écrire une fonction `recherche_all(t,x)` qui détermine la *liste de tous les indices* `i` tel que `t[i]=x`, et la renvoie (si `x` n'appartient pas à `t`, la fonction doit renvoyer la liste vide `[]`).

Exercice 7 (Recherche du maximum dans une liste) Montrer la terminaison et la correction de l'algorithme de recherche du maximum dans une liste.

Exercice 8 (Recherche du maximum dans une liste : variante)

1. Donner une fonction `argmax(t)` qui donne le plus petit indice `i` tel que `t[i]` soit le maximum de `t`.
2. Donner une fonction `argmax2(t)` qui donne la liste de tous les indices `i` tels que `t[i]` soit le maximum de `t`.

3.2 Méthode d'Euler

Exercice 9 (Étude de l'oscillation d'un pendule par la méthode d'Euler) On considère un pendule simple oscillant dans un champ de pesanteur constant, composée d'une masse `m` fixée à l'extrémité d'une tige rigide de masse nulle, tournant sans frottement dans le plan vertical autour de son extrémité fixe. On note θ l'élongation angulaire du pendule, et on rappelle que θ vérifie l'équation :

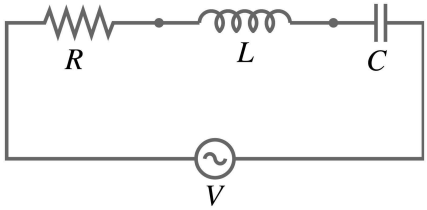
$$\theta'' + \frac{g}{\ell}\sin(\theta) = 0$$

où ℓ est la longueur de la tige et g l'accélération de la pesanteur.

On donne $\ell = 10\,cm$, $g = 9{,}81\,m.s^{-1}$ et on suppose qu'initialement le pendule a une vitesse nulle et fait un angle de $\frac{\pi}{2}$ avec la verticale.

1. Résoudre numériquement l'équation avec la méthode d'Euler. Représenter les courbes de θ et θ' en fonction du temps.
2. Vérifier la loi de conservation de l'énergie, c'est-à-dire que :
$$\frac{1}{2}\ell^2(\theta')^2 - g\ell\cos(\theta)$$
est constant au cours du temps.
3. Tracer sur un même graphe les différentes orbites de phase, c'est-à-dire les courbes $\big(\theta(t), \theta'(t)\big)$, pour les différentes positions initiales $\theta(0)$ choisie dans l'ensemble $\left\{k\frac{\pi}{10} \Big/ k \in [\![0,10]\!]\right\}$.
4. Recommencer l'exercice en utilisant la fonction `odeint` du module `scipy.optimize`.

Exercice 10 (Circuit RLC) On considère un circuit électrique simple constitué d'une source de tension V, d'une résistance R, d'une bobine L et d'une capacité C montées en série comme le montre le schéma suivant.



L'équation satisfaite par la tension u aux bornes de la bobine est :

$$\frac{d^2u}{dt^2}+\frac{R}{L}.\frac{du}{dt}+\frac{1}{LC}.u=\frac{d^2E}{dt^2}$$

1. À l'aide de la méthode d'Euler, tracer la courbe de la tension aux bornes de la bobine pour $R=30\,\Omega$, $L=870\,mH$, $C=630\,mF$ et $V(t)=10.\sin(2\pi.f.t)$, avec $f=100\,Hz$, sur l'intervalle de temps $[0,0.4]$.
2. Faire de même avec la fonction `odeint` du module `scipy`.

Exercice 11 (Équations de Lotka-Volterra) On veut étudier les équations de Lotka-Volterra formées d'un couple d'équations différentielles non linéaires du premier ordre utilisées pour modéliser l'évolution des populations de deux espèces, l'une étant la proie de l'autre.

$$\begin{cases} \dfrac{dx}{dt}(t)=x(t).\big(a-b.y(t)\big) \\ \dfrac{dy}{dt}(t)=-y(t).\big(c-d.x(t)\big) \end{cases}$$

$x(t)$ et $y(t)$ représentent respectivement le nombre d'individus des populations de proies et de prédateurs. La constante a est la taux de reproduction des proies sans prédateurs. La constante b est le taux de mortalité des proies du prédateurs. La constante c est le taux de mortalité des prédateurs en l'absence de proie, et d est le taux de reproduction des prédateurs en présence de proies.

On ne précise pas l'unité de temps et on prendra $a=1.1$, $b=0.01$, $c=1.2$ et $d=0.005$.

1. Dans cette question on utilise le schéma d'Euler.
   (a) Tracer sur un même graphe les courbes $\big(x(t),y(t)\big)$, où $x(t)$ est le nombre de proies à l'instant t et $y(t)$ le nombre de prédateurs, avec comme conditions initiales $\big(x(0),y(0)\big)$:

   $$(100,50)\quad(100,130)\quad(200,100)\quad(50,50)$$

   (b) Traer sur un même graphe l'évolution au cours du temps des populations de proies et de prédateurs pour la condition initiale $\big(x(0),y(0)\big)=(50,50)$ sur une période de 20 unités de temps.
2. Même questions en utilisant la fonction `odeint` du module `scipy`.

3.3 Algorithme de Gauss-Jordan

Exercice 12 (Matrices de Hilbert) Soit H_n la matrice de Hilbert d'ordre n, de coefficients $h_{ij}=\dfrac{1}{i+j-1}$ pour $1\leq i,j\leq n$.

1. Créer une fonction `hilbert` d'argument `n` qui renvoie la matrice de Hilbert d'ordre n sous forme de tableau `numpy`.
2. Inverser H_{10} et faire afficher le coefficient en bas à droite. Le résultat attendu (formel) est 44 914 183 600. Que constate-t-on ? Pour $n=20$, le coefficient correspondant devrait être 48 722 219 250 572 027 160 000.
   On peut démontrer que la matrice inverse de H_n est à coefficients dans $\mathbb{Z}$.

3. Résoudre le système $H_5X = Y$ avec $Y = \begin{pmatrix} -7.7 \\ -6. \\ 2.1 \\ -0.5 \\ 0. \end{pmatrix}$ puis $Y = \begin{pmatrix} -7.7 \\ -6. \\ 2.1 \\ -0.4 \\ 0. \end{pmatrix}$.

   Que peut-on en conclure ?

Exercice 13 (Matrices tridiagonales) On définit la matrice carrée d'ordre n :

$$V_n = \begin{pmatrix} 2 & -1 & & & 0 \\ -1 & \ddots & \ddots & & \\ & \ddots & \ddots & \ddots & \\ & & \ddots & \ddots & -1 \\ 0 & & & -1 & 2 \end{pmatrix}$$

Résoudre écrire une fonction `resolution(Y)` qui donne en sortie la solution du système linéaire $V_nX = Y$, en adaptant l'algorithme du pivot de Gauss à la forme particulière de V_n (forme tridiagonale).

4 Lire et écrire dans un fichier

Exercice 14 Objectif : créer un fichier de n lignes remplies aléatoirement et représentant les résultats des élèves d'une promotion. En récupérer le contenu, puis calculer la moyenne des élèves par genre.

1. Le fichier des résultats est constituée de 100 lignes. Chaque ligne est de la forme :

   ```
   identifiant :20155 note :12.55 sexe : F
   ```

   Voici comment est rempli le fichier :

   (a) Un élève est représenté par son identifiant : un nombre entier aléatoire entre 1000 et 60000.

   (b) La note est un nombre flottant choisi aléatoirement entre 0 et 20 et arrondi à une décimale comme 13.1.

   (c) Le sexe est déterminé aléatoirement : `H` ou `F`.

   Écrire une fonction `fill` qui prend en paramètres un nom de fichier comme `'resultat.txt'` et un nombre de lignes `n`. La fonction remplit le fichier de `n` lignes sur le modèle ci-dessus (on utilisera le module `random`).
   Voici un exemple d'appel :

   ```
   >>> fill('resultat.txt',100)
   ```

2. Écrire une fonction `file2list` qui prend en paramètres un nom de fichier rempli comme `resultat.txt`, en récupère le contenu sous forme d'une liste dont les éléments sont des listes de trois éléments :l'identifiant, la note, et le sexe. Exemple :

   ```
   >>> file2list('resultat.txt')
           [[18706, 14.1, 'F'],
            [53651, 14.3, 'F'],
            [59644, 13.5, 'H']]
   ```

3. Écrire une fonction `averageByGenre` qui prend en paramètres une liste de listes, comme celle retournée par la fonction précédente et un caractère `H` (homme) ou `F` (femme). La fonction retourne la moyenne des filles, ou la moyenne des garçons.

```
>>> liste=file2list('resultat.txt')
>>> averageByGenre(liste,'H')
        13.5
>>> averageByGenre(liste,'F')
        14.2
```

4.1 Requêtes SQL

Exercice 15 (Un peu de géographie) Cet exercice utilise le fichier `geographie.sql`.

Nous disposons de trois tables. Une table `communes` dont les attributs sont :

```
num_departement: numéro du département ;
nom: nom de la commune ;
canton : canton de la commune ;
population_2010: nombre d'habitants lors du recensement de 2010 ;
population_1999: nombre d'habitants lors du recensement de 1999 ;
surface: surface de la commune en km² ;
longitude: longitude du centre de la commune en GRD
          (peu importe l'unité, quand il s'agira de faire des dessins) ;
lattitude: lattitude du centre de la commune en GRD ;
zmin: hauteur minimum de la commune par rapport au niveau de l'eau;
zmax: hauteur maximum de la commune par rapport au niveau de l'eau.
```

On dispose aussi d'une table `departements` dont les attributs sont `num_departement`, `num_region` et `nom`, et d'une table `regions` dont les attributs sont `num_region` et `nom` (on rappelle que chaque commune appartient à un département, et que chaque département appartient à une région).

1. **Requêtes sans jointure.**
   (a) Combien d'habitants en France (en supposant que chaque habitant est rattaché à exactement une commune, ce qui est peut-être un postulat faux...) ?
   (b) Combien de communes sur l'île de la Réunion ?
   (c) Obtenez la liste des dix plus petites communes de France en termes de surface. *L'instruction* `ORDER BY Attribut LIMIT n` *affiche les valeurs d'attributs par ordre croissant en se limitant aux n premières valeurs.*
   (d) Obtenez la liste des dix communes de France les plus peuplées. *L'instruction* `ORDER BY Attribut DESC LIMIT n` *affiche les valeurs d'attributs par ordre décroissant en se limitant aux n premières valeurs.*
   (e) Quelles sont les 5 communes les plus densément peuplées de l'île de la Réunion ?
   (f) Obtenez la liste des numéros des départements, et pour chaque département correspondant, le nombre de communes.
   (g) Obtenez la liste des numéros des départements, et pour chaque département correspondant, la population totale. Ordonnez par population totale décroissante.

(h) Combien de communes françaises contiennent la lettre 'x' ou 'X' dans leur nom ?
L'instruction `WHERE Attribut LIKE 'chaine'` *teste si la chaîne de caractère* `Attribut` *est égale à* `'chaine'`. *Le caractère* `'%'` *représente une chaîne quelconque (même vide). Le test est insensible à la casse (= majuscule ou minusucule)*

(i) Combien de communes françaises contiennent les six voyelles 'a', 'e', 'i', 'o', 'u', 'y' (en majuscule ou minuscule) dans leurs noms ?

(j) Existe-t-elle une telle commune sur l'île de la Réunion ?

(k) Quelles sont les 15 communes de France dont l'écart entre l'altitude la plus haute, et l'altitude la plus basse, est le plus élevé ?

(l) Quelles sont les cinq communes de l'île de la Réunion où la population a le plus augmenté entre 1999 et 2010 ?

(m) Quelles sont les cinq communes de l'île de la Réunion où la population a le plus augmenté en pourcentage entre 1999 et 2010 ?
On affichera le nom des communes, l'augmentation effective, et l'augmentation en pourcentage.

2. **Jointures à deux tables.**

(a) Obtenez la liste des départements (les noms !), et pour chaque département correspondant, le nombre de communes.
Vous ordonnerez par nombre de communes décroissant.

(b) Obtenez la liste des départements (les noms !) des régions Auvergne et Aquitaine.

(c) Quelle est la population de chaque département (on veut les noms des départements en toutes lettres) ?
Les départements seront classés par population décroissante.

(d) Quelle est la densité de population de chaque département (on veut les noms des départements en toutes lettres et le classement par ordre de densité décroissante) ?

(e) Obtenez la liste des régions de France, ainsi que le nombre de départements composant la région, rangée par nombre de départements décroissant.

(f) Obtenez la liste des départements (les noms !) de plus de 1 200 000 habitants.

(g) Existe-t-il une commune de la région Aquitaine, dont le nom contient les six voyelles `'a'`, `'e'`, `'i'`, `'o'`, `'u'`, `'y'` (en majuscule ou minuscule) ?
Indice : l'Aquitaine est la région numéro 2.

(h) Obtenez (sans doublons) la liste des régions contenant des départements dont le nom commence par 'c' ou 'C'.

3. Jointures à trois tables.

(a) Obtenez la liste des régions de France, et pour chaque région, la population totale, ainsi que le nom et le nombre d'habitants de la commune la plus peuplée de la région.
Les résultats seront affichés par ordre décroissant de population totale.

(b) Obtenir la liste des régions (avec le nom en français), avec la population totale de chaque région, classées par ordre décroissant de population ?

(c) Densité de population de chaque région ?

(d) Quelles sont les 15 communes de France dont l'écart entre l'altitude la plus haute, et l'altitude la plus basse, est le plus élevé ? Cette fois, précisez le nom du département et le nom de la région pour chaque commune...

(e) Combien de communes dans la région Basse Normandie ?

4. **Sous-requêtes.**
Pour la clarté de la rédaction, on pourra présenter les requêtes sous la forme :

```
soustable = SELECT ...
SELECT ... FROM soustable WHERE ...
```

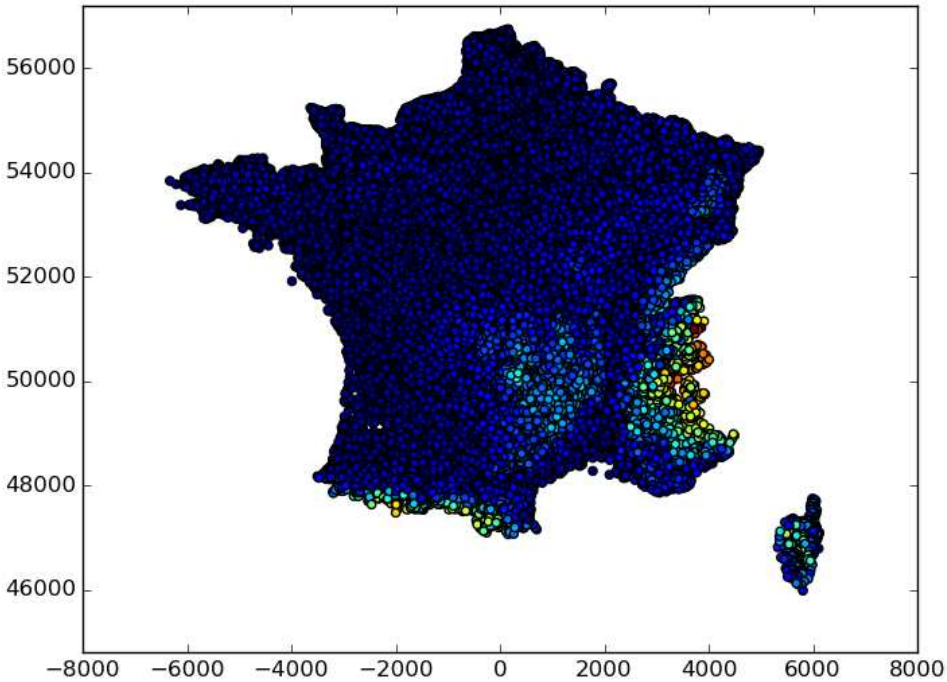
(a) Afficher la (ou les) commune(s) d'altitude maximale.

(b) Donner, pour chaque numéro de département, le nombre de communes dont la population en 1999 était supérieure à la population moyenne en 2010.

(c) Afficher le nom du département de métropole où se trouve la commune d'altitude la plus basse.
Utiliser trois requêtes imbriquées.

(d) Afficher le nom de la région où se trouve la commune d'altitude la plus basse.

5. **Un peu de dessin...**



La figure précédente est une représentation graphique des villes de métropole par des points dont la couleur dépend de l'altitude. Elle a été obtenue avec le programme suivant :

```
1 import numpy as np
2 from math import cos,pi
3 import matplotlib.pyplot as plt
4 import MySQLdb
```

```
5  Base = MySQLdb.connect(host='***',user='***',passwd='***',db
      ='geographie')
6  requete='SELECT longitude, latitude, zmax FROM communes
      WHERE num_departement <= 95'
7  Base.query(requete)
8  resultat = Base.use_result()
9  res = resultat.fetch_row(maxrows=0)
10 corr=cos(40*pi/180)
11 Lx=[]
12 Ly=[]
13 Lz=[]
14 for ligne in res:
15     Lx.append(float(ligne[0]))
16     Ly.append(float(ligne[1]))
17     Lz.append(float(ligne[2]))
18 plt.scatter(corr*np.array(Lx),Ly,c=Lz)
19 plt.axis('equal')
20 plt.show()
21 Base.close()
```

(a) Adapter le programme précédent pour qu'il regroupe les communes par canton.

(b) Représenter par des points de couleur l'attractivité des cantons français, représentée par l'augmentation relative du nom d'habitants entre 1999 et 2010 (points placés en fonction de longitude et latitude).

(c) Représenter par des points les 5000 plus grosses communes de métropole (points placés en fonction de longitude et latitude).

Chapitre 4

Exercices de seconde année

1 Piles

Exercice 16 (Fonctions sur les piles) Les *piles* seront modélisées par des listes, et on ne s'autorisera comme opérations que les méthodes `pop()`, `append()`, ainsi que la comparaison à la liste vide `[]`.
On rappelle que les variables de type `list` sont modifiées en place lorsqu'elles sont données en argument d'une fonction.
On pourra effectuer une copie d'une pile ainsi : `l2 = [ x for x in l ]` (et non `l2=l`).
Dans les questions suivantes, on n'utilisera que les opérations ci-dessus, ainsi que les fonctions programmées dans les questions précédentes.

1. Ecrire une fonction `isempty` qui renvoie un booléen indiquant si une pile est vide ou non.
2. Ecrire une fonction `card` qui renvoie le nombre d'éléments d'une pile.
3. Ecrire une fonction `peek` qui renvoie l'élément de tête sans le dépiler.
4. Ecrire une fonction `clear` qui vide une liste en dépilant tous ses éléments et ne renvoie rien.
5. Ecrire une fonction `dup` qui duplique l'élément du haut de la pile et ne renvoie rien.
6. Ecrire une fonction `swap` qui permute les deux éléments du haut de la pile et ne renvoie rien.
7. Ecrire une fonction `permut` qui effectue une permutation circulaire des éléments d'une pile (pour cela, on pourra utiliser une pile auxiliaire) et ne renvoie rien.
   Par exemple `[6,4,2,7,9]` devient `[9,6,4,2,7]`.

Exercice 17 (Notation polonaise inversée) La notation polonaise inversée (NPI) permet d'écrire un calcul sans aucune parenthèse. Pour cela on place en premier les *opérandes*, suivi des *opérateurs*.
Par exemple, le calcul `32+2*(12-3*(7-2))` se note `32 2 12 3 7 2 - * - * +`.
En python le calcul en NPI sera donné sous forme d'une *liste* :

```
[32,2,12,3,7,2,'-','*','-','*','+']
```

appelée `calcul`, où les opérandes sont des nombres de type `float`, et les opérateurs sont des chaînes de caractères. On se limitera aux opérations : addition, soustraction, multiplication et division.
Pour évaluer le calcul, on parcourt utilise une *pile* auxiliaire `pile_aux`. On parcourt la `liste` représentant le calcul de gauche à droite, élément par élément :

- si l'élement est un opérande, on l'empile dans la pile `pile_aux` ;
- si l'élément est un opérateur, on dépile les deux éléments du sommet de la pile `pile_aux`, on leur applique l'opérateur, on empile le résultat sur `pile_aux`.

A la fin, la pile auxiliaire ne contient plus qu'un élément qui est le résultat du calcul. Par exemple, pour le calcul `[32,2,12,3,7,2,'-','*','-','*','+']`, la variable `pile_aux` évolue de la façon suivante :

```
[]
[32]
[32, 2]
[32, 2, 12]
[32, 2, 12, 3]
[32, 2, 12, 3, 7]
[32, 2, 12, 3, 7, 2]
[32, 2, 12, 3, 5]
[32, 2, 12, 15]
[32, 2, -3]
[32, -6]
[26]
```

Ecrire une fonction `NPI` qui prend en entrée une liste représentant un calcul, et qui évalue ce calcul en utilisant l'algorithme présenté ci-dessus.
Pour tester si un élément de la liste est un opérande ou un opérateur, on pourra utiliser une variable `op = ['+','-','*','/']`.

Exercice 18 (Expressions bien parenthésées) Pour vérifier si une chaîne de caractères est bien parenthésée, c'est-à-dire qu'elle possède autant de parenthèses ouvrantes que de parenthèses fermantes et ceci dans le bon ordre, on utilise une pile.
L'algorithme consiste à parcourir la chaîne de caractères de gauche à droite :

- à chaque fois qu'on rencontre une parenthèse ouvrante, on empile une parenthèse fermante ;
- à chaque fois qu'on rencontre une parenthèse fermante :
  — si la pile est vide c'est que l'expression est mal parenthésée,
  — sinon on dépile une fois ;
- à la fin du parcours, il faut que la pile soit vide pour dire que l'expression est bien parenthésée.

Ecrire une fonction `parentheses` qui prend en entrée une chaîne de caractères, et qui renvoie en sortie un booléen indiquant si l'expression est bien parenthésée.

Exercice 19 (Tri d'une pile)

1. Ecrire une fonction `insertion` qui prend en entrée une pile `pile` triée dans l'ordre croissant et un nombre `x`, et qui insère `x` dans la pile de telle sorte qu'elle soit toujours triée dans l'ordre croissant.
   Pour cela on pourra utiliser un pile auxiliaire `buffer`.
2. En déduire un algorithme de tri en place d'une pile quelconque.
   Encore une fois, on pourra utiliser un pile auxiliaire `buffer`.

2 Récursivité

Exercice 20 (Coefficients binômiaux)

1. En utilisant la formule $\binom{n}{k} = \frac{n!}{k! \times (n-k)!} = \frac{n(n-1) \times (n-k+1)}{k!}$, écrire un algorithme itératif qui calcule les coefficients binômiaux.
2. En utilisant la formule de Pascal $\binom{n}{k} = \binom{n-1}{k-1} + \binom{n-1}{k}$, écrire une version récursive de cet algorithme.

Exercice 21 (PGCD) Si a et b sont deux entiers naturels (avec $b \neq 0$), et si la division euclidienne de a par b s'écrit $a = bq + r$ avec $r \in [\![0, b-1]\!]$, alors on montre que $\text{pgcd}(a, b) = \text{pgcd}(b, r)$.
En déduire un algorithme itératif et un algorithme récursif de calcul du pgcd.

Exercice 22 (Inversion récursive d'une liste) Écrire un programme récursif qui inverse l'ordre des éléments d'une liste.

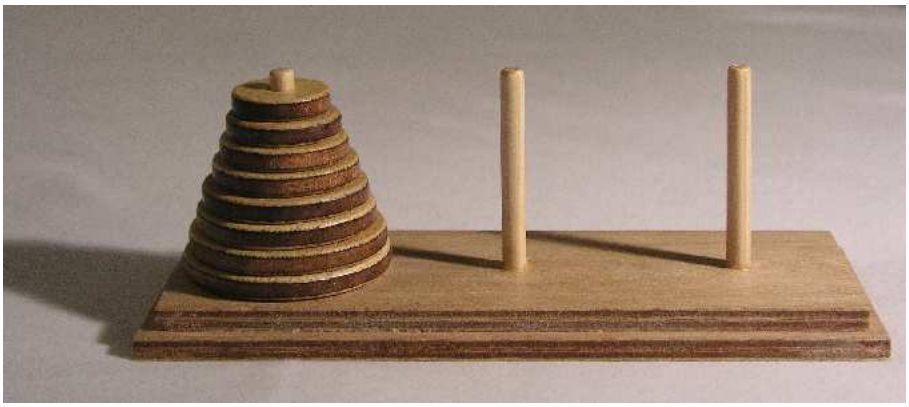
Exercice 23 (Suite de Fibonacci) La suite de Fibonacci est la suite $(u_n)_{n\in\mathbb{N}}$ définie par :

$$u_0 = u_1 = 1 \quad \text{et} \quad \forall n \in \mathbb{N},\ u_{n+2} = u_{n+1} + u_n$$

1. Écrire une fonction *itérative* qui calcule u_n pour n donné..
   Évaluer sa complexité spatiale et temporelle.
2. (a) Écrire une fonction *récursive* qui calcule u_n pour n donné..
   Évaluer sa complexité spatiale et temporelle.
   (b) Utiliser une variable globale `nb_recalculs` de type `list` à $n+1$ éléments, pour que `nb_recalculs[k]` soit égal au nombre de fois où u_k a été calculé lors du calcul de u_n ($k \in [\![0, n]\!]$).
3. Pour améliorer la fonction récursive utiliser une technique de *mémoïsation* : utiliser une variable globale `memo` type `list` à $n+1$ éléments, qui stocke les termes de la suite déjà calculés et les réutilise lors des appels récursifs.
   Évaluer sa complexité spatiale et temporelle.
4. Écrire une fonction qui calcule u_n, à l'aide d'une fonction récursive auxiliaire d'arguments d'entrée `(n,a,b)`, qui calcule successivement les couples (u_{n-1}, u_n), pour une suite $(u_n)_{n\in\mathbb{N}}$ telle que $u_0 = a$, $u_1 = b$ et $\forall n \in \mathbb{N}$, $u_{n+2} = u_{n+1} + u_n$.
   Évaluer sa complexité spatiale et temporelle.
5. Dans cette question on pose $u_{-1} = 0$.
   (a) Montrer que, pour tout $n \in \mathbb{N}^*$, $\begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}^n = \begin{pmatrix} u_n & u_{n-1} \\ u_{n-1} & u_{n-2} \end{pmatrix}$.
   En déduire que, pour tout $n \in \mathbb{N}^*$, $\begin{cases} u_{2n-1} = 2u_{n-1}u_n - u_{n-1}^2 \\ u_{2n} = u_{n-1}^2 + u_n^2 \end{cases}$.
   (b) En déduire une fonction récursive qui calcule le couple (u_{n-1}, u_n) pour n donné.
   Évaluer sa complexité spatiale et temporelle.

Exercice 24 (Tours de Hanoï) Le problème des tours de Hanoï est un jeu de réflexion imaginé par le mathématicien français Édouard Lucas, et consistant à déplacer des disques de diamètres différents d'une tour de « départ » à une tour d'« arrivée » en passant par une tour « intermédiaire », et ceci en un minimum de coups, tout en respectant les règles suivantes :

- on ne peut déplacer plus d'un disque à la fois,
- on ne peut placer un disque que sur un autre disque plus grand que lui ou sur un emplacement vide.



1. Proposer alors une fonction récursive `hanoi(n,dep,aux,arr)` qui affiche les déplacements à effectuer pour déplacer n disques du piquet `dep` vers le piquet `arr`, en utilisant le piquet intermédiaire `aux`.
2. Combien de déplacements sont nécessaires pour que les n disques passent tous du piquet de « départ » au piquet d'« arrivée » ?
3. Selon la légende imaginée par E. Lucas, le jeu a commencé en -1500 avant J.C., dans un temple à Hanoï, avec 64 disques d'or, et la fin du monde arrivera quand le jeu se termine (à raison d'une seconde par déplacement). Est ce vrai ? (on prendra 14 milliards d'années comme âge de l'univers)

Exercice 25 (Sudoku) Dans cette exercice, on se propose de résoudre les grilles de sudoku de manière récursive. Une grille de sudoku sera representée par une matrice 9×9 comme suit :

	9			2	6			8
	3							
	7					2	1	4
	1			7	8			
			6		3			
			4	5			3	
5	6	7					2	
							9	
8			3	1			5	

est representée par

```
M=[[0,9,0,0,2,6,0,0,8],
[0,3,0,0,0,0,0,0,0],
[0,7,0,0,0,0,2,1,4],
[0,1,0,0,7,8,0,0,0],
[0,0,0,6,0,3,0,0,0],
[0,0,0,4,5,0,0,3,0],
[5,6,7,0,0,0,0,2,0],
[0,0,0,0,0,0,0,9,0],
[8,0,0,3,1,0,0,5,0]]
```

1. Écrire une fonction `recherche_zero(M)` qui renvoie les coordonnées de la première case non remplie de la grille `M`, et qui renvoie `[10,10]` si toutes les cases sont remplies.
2. Écrire une fonction `test_ligne_colonne(M,i,j,val)` qui vérifie si on peut mettre la valeur `val` dans la case `[i,j]` de la grille `M` en vérifiant que `val` n'est pas déjà dans la ligne `i` ou la colonne `j`.
   Cette fonction renverra `True` ou `False`.

3. Écrire une fonction `test_sous_carre(M,i,j,val)` qui fait la même chose mais vérifie si `val` n'est pas présent dans le « sous-carré » qui contient la case `[i,j]`.
4. Compléter le code ci-dessous pour créer une fonction qui vérifie si on peut mettre la valeur `val` dans la case `[i,j]` de la grille `M` (vous n'avez le droit qu'à une ligne de code.)

```
def test_case(M,i,j,val):
    return ...
```

5. À l'aide des fonctions précédentes, écrire une fonction récursive `sudoku` qui va résoudre la grille de sudoku en testant toutes les valeurs possibles pour chaque case.
   La fonction se contentera d'afficher la grille solution. S'il y a plusieurs solutions on les affichera toutes (bien que dans la vraie règle cette grille serait déclarée insoluble)

Exercice 26 (Algorithme de Hörner) Soit $P = a_n X^n + a_{n-1} X^{n-1} + \cdots + a_0 \in \mathbb{R}[X]$ un polynôme, et x_0 un nombre réel.

1. Écrire un programme « naïf » qui calcul $P(x_0)$.
2. La méthode de Hörner consiste à améliorer ce résultat en effectuant le calcul comme suit :
$$P(x_0) = ((\ldots((a_n x_0 + a_{n-1})x_0 + a_{n-2})x_0 + \ldots)x_0 + a_1)x_0 + a_0.$$
   Proposer une version itérative et une version récursive de cet algorithme.
3. Comparer les complexités ces algorithmes (on ne prendra en compte que les multiplications).

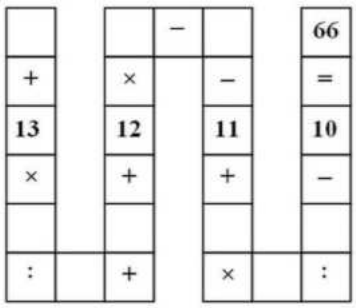
Exercice 27 (Recherche d'un caractère dans une chaîne)

1. Écrire une fonction récursive `search(c,s)` qui renvoie un booléen indiquant si c est dans s.
2. Améliorer `search` pour qu'elle renvoie la première position du caractère dans la chaîne, s'il y est, et le nombre -1 sinon.

Exercice 28 (Calcul d'un déterminant)

1. Écrire un programme récursif qui calcule le déterminant d'une matrice à l'aide du développement par rapport à la première colonne.
2. Montrer que la complexité est $\Gamma(n!)$.
3. Proposer un meilleur algorithme de calcul de déterminant.

Exercice 29 (Serpent vietnamien) Le but du jeu est de remplir les cases de la grille suivante, avec des chiffres de 1 à 9 (qu'il ne faut utiliser qu'une fois chacun), en suivant l'ordre des opérations, de façon à obtenir 66 comme résultat final :



1. Écrire une fonction récursive qui prend en entrée une liste `l` de nombres, et donne en sortie la liste de toutes les permutations de cette liste (donc si `l` est une liste de n nombres, la liste donnée en sortie est une liste de $n!$ listes).
2. En déduire un programme Python qui détermine toutes les solutions au jeu du serpent vietnamien (en sortie on pourra se contenter de n'afficher que le nombre total de solutions).

Exercice 30 (Algorithme de Strassen) Dans cet exercice, nous allons voir plusieurs façons de programmer le produit de deux matrices carrées d'ordre k. Pour manipuler les matrices on utiliser le module `numpy`.

1. Écrire un algorithme qui effectue le produit matriciel grâce à la formule appris dans le cours de mathématiques. Évaluer sa complexité.
2. On va essayer d'améliorer les performances.
   (a) Écrire une fonction `logdeux(k)` qui prend en entrée un entier naturel `k`, et renvoie en sortie le plus petit entier naturel n tel que $k \leq 2^n$.
   (b) Créer une fonction `newsize(m)` qui prend en paramètres une matrice carrée `m`, et retourne une nouvelle matrice qui est une copie de `m` à laquelle on ajoute des zéros pour en faire une matrice $2^n \times 2^n$.
   (c) Créer une fonction `one2four(m)` qui prend en paramètre une matrice carrée $2^n \times 2^n$ et renvoie les 4 sous-matrices usuelles de tailles $2^{n-1} \times 2^{n-1}$.
   (d) Créer une fonction `four2one((a,b,c,d))` inverse de la précédente.
   (e) En utilisant ce qui précède, créer une fonction récursive `produit(m1,m2)` qui prend en paramètres deux matrices carrées de même dimensions$2^n \times 2^n$ et les découpe en blocs comme ici :

   $$m_1' = \begin{pmatrix} A & B \\ C & D \end{pmatrix} \qquad \text{et} \qquad m_2' = \begin{pmatrix} A' & B' \\ C' & D' \end{pmatrix}$$

   La fonction calcule ensuite récursivement le produit $m_1' m_2'$ en utilisant le produit par blocs vu en cours.
   (f) Écrire le programme principal `mult(m1,m2)` qui prend en entrée deux matrices carrées `m1`, `m2`, les redimensionne en matrices de taille $2^n \times 2^n$, effectue leur produit matriciel avec la fonction `produit`, puis affiche le résultat sans les lignes et colonnes de zéros ajoutés par la fonction `newsize`.
   (g) Quelle est la complexité de cet algorithme ?
   (h) Pour diminuer cette complexité, Volker Strassen a mis en place la méthode suivante en 1968. Pour effectuer le produit $m_1' m_2'$, on pose :

   $$\begin{aligned} X_1 &= (A+D)(A'+D') \\ X_2 &= (C+D)A' \\ X_3 &= A(B'-D') \\ X_4 &= D(C'-A') \\ X_5 &= (A+B)D' \\ X_6 &= (C-A)(A'+B') \\ X_7 &= (B-D)(C'+D') \end{aligned}$$

et vérifie alors facilement que :

$$m_1' m_2' = \begin{pmatrix} X_1 + X_4 - X_5 + X_7 & X_3 + X_5 \\ X_2 + X_4 & X_1 - X_2 + X_3 + X_6 \end{pmatrix}$$

Transformer la fonction `mult(m1,m2)` en une fonction `strassen(m1,m2)` qui utilise la méthode de Strassen.
Quelle est la complexité de ce nouvel algorithme ?

3 Tris

Exercice 31 (Tri bulle)

1. On considère une liste `l` de n nombres.
   On la parcourt de gauche à droite, en comparant à chaque fois l'élément avec le suivant : s'ils ne sont pas dans l'ordre croissant, alors on les permute.
   À la fin du parcours de la liste, quelle particularité a la liste `l` ?
2. En déduire un nouvel algorithme de tri.
3. Déterminer sa complexité en moyenne, dans le pire des cas, et dans le meilleur des cas.
   On ne s'intéressera qu'au nombre de comparaisons.

Exercice 32 (Tri par sélection)

1. Écrire une fonction qui détermine le plus petit élément d'une liste de nombres et le place en première position.
2. En déduire une fonction (récursive ou non) qui trie en place une liste de nombres.
3. Déterminer sa complexité en moyenne, dans le pire des cas, et dans le meilleur des cas.
   On ne s'intéressera qu'au nombre de comparaisons.

Exercice 33 (Recherche de la médiane) La médiane d'un ensemble de n nombres est le nombre qui est plus grand que la moitié de ces nombres, et plus petit que l'autre moitié de ces nombres. Autrement dit, c'est le nombre qui se trouve au milieu du tableau si on trie celui-ci. Pour une liste **triée** de $2N+1$ nombres c'est donc le $(N+1)$-ième nombre, pour une liste **triée** de $2N$ nombres, une médiane est n'importe quel nombre compris entre le N-ième et le $(N+1)$-ième.

1. Proposer une fonction `mediane_tri` qui calcule une médiane d'une liste en utilisant un algorithme de tri.

Pour une liste quelconque, le mieux qu'on puisse faire est alors en $O(n \log n)$, même si, dans certains cas, le tri insertion donne une complexité en $O(n)$.
On ne va donc pas trier la liste, juste chercher sa médiane.
On implante ici la méthode Quickselect de Hoare qui a une complexité moyenne en $O(n)$, $O(n^2)$ dans le pire des cas.

2. Implémenter la fonction `partition(l,g,d)` du cours, qui transforme la sous-liste `l[g:d]` de telle sorte qu'à gauche de la valeur `l[g]` se trouve les éléments plus petits et à droite les éléments plus grands. En sortie elle donne la position du pivot `l[g]`.

3. On écrit une fonction récursive `select_rec(l,g,d,k)` qui prend en paramètres une liste l, deux indices gauche et droite, g, d ainsi qu'un indice k.
   Cette fonction retourne le k-ième élément de la liste dans le classement croissant (sans trier la liste).
   Pour ce faire, on partitionne d'abord la portion entre g et d de la liste et on note p, l'indice du pivot (donné par l'appel `partition(l,g,d)`) :
   (a) Si $p = k$, on a l'élément k de la liste.
   (b) Si $p < k$, l'élément k est à droite de p, on appelle `select_rec` sur la portion à droite du pivot.
   (c) Sinon, le k-ième élément est à gauche de p, on appelle `select_rec` sur la portion de la liste à gauche du pivot.

   La close d'arrêt est donnée par le cas où la sous-liste n'a qu'un seul élément.
   Comme pour le tri rapide en place, on écrit ensuite une fonction `select(l,k)` qui appelle `select_rec` pour obtenir le k-ième élément du tableau dans le classement croissant.
   Tester sur des exemples et remarquer que la liste n'a pas besoin d'être entièrement triée.
4. Pour une liste l trié de longueur N, la médiane est l'élément placé à proximité immédiate de l'indice $N/2$ (suivant la parité du nombre d'éléments) :
   (a) si N est impair, la médiane est $t[(N-1)/2]$,
   (b) si N est pair (et la liste non vide), la médiane est n'importe quel nombre strictement compris entre $t[N/2-1]$ et $t[N/2]$.

   En gardant ceci en mémoire, et en utilisant la fonction `select`, proposer une nouvelle implémentation de la fonction `mediane`.
   Pour cela faites attention au fait que si la liste `l` est triée, l'appel `select(l,k)` ne donne pas `l[k]` mais `l[k-1]` (car les listes sont indexées à partir de 0).

4 Autres

4.1 Stéganographie

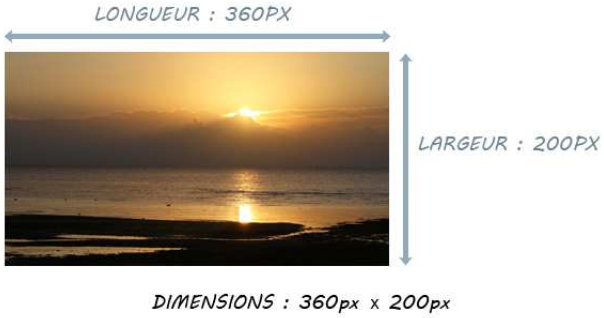
La stéganographie est l'art de la dissimulation : son objet est de faire passer inaperçu un message dans un autre message. Elle se distingue de la cryptographie, « art du secret », qui cherche à rendre un message inintelligible à autre que qui-de-droit. Pour prendre une métaphore, la stéganographie consisterait à enterrer son argent dans son jardin là où la cryptographie consisterait à l'enfermer dans un coffre-fort — cela dit, rien n'empêche de combiner les deux techniques, de même que l'on peut enterrer un coffre dans son jardin.

On peut utiliser Python pour cacher une image dans une autre. Commençons par quelques notions sur les images en informatique :

- une image est une représentation visuelle de quelque chose ;
- cette représentation est découpée en éléments carrés appelés *pixels* (**pi**cture **el**ement), chaque pixel étant composé d'une seule couleur ;
- un écran possède un nombre fixe de pixels défini comme étant le produit du nombre de pixels en largeur par le nombre de pixels en hauteur (c'est la *définition* d'écran, par exemple : 1024×768).

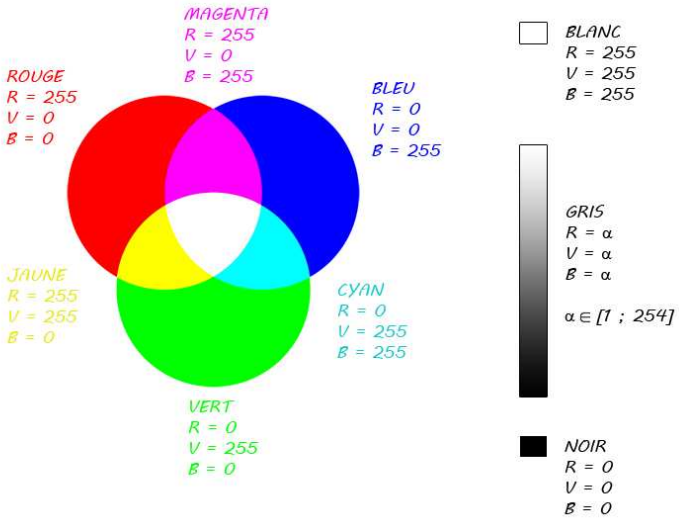
On peut définir la *taille* et la *dimension* d'une image :

— **Taille d'une image**. Espace mémoire qu'occupe un fichier image sur le disque dur, exprimée en octets. Exemple : 250 Ko.
— **Dimensions d'une image**. Espace visuel qu'occupe une image sur l'écran, représentées par un couple de valeurs longueur × largeur. L'unité utilisée ici sera le pixel. Exemple : 360 × 200.

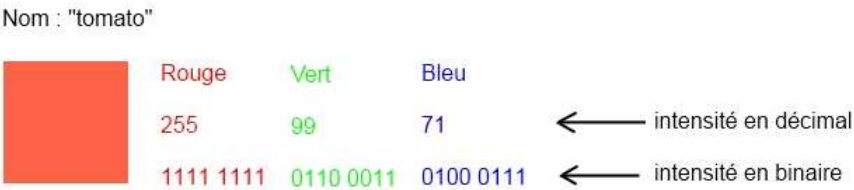


Pour représenter un couleur, on utilise une combinaison de couleurs primaires. Il existe plusieurs façons de le faire : ce sont les *modes colorimétriques*.

Nous nous intéresserons ici au mode RGB (Red-Green-Blue) ou RVB (Rouge-Vert-Bleu). Ce mode attribue à chaque pixel une intensité de rouge, vert et bleu. Cette intensité est un nombre compris entre 0 (intensité nulle) et 255 (intensité maximum). On peut représenter de cette façon $256^3 = 16777216$ couleurs différentes !

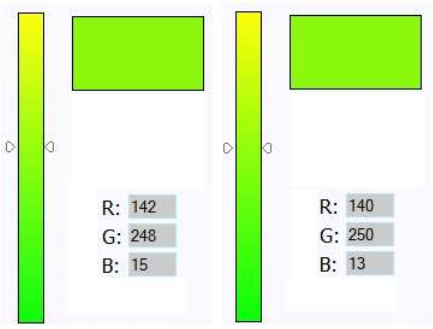


Chaque intensité étant une valeur comprise entre 0 et 255, il faut donc 8 bits (1 octet) pour coder une intensité en binaire ($2^8 = 256$).



Nous allons cacher notre image secrète (hidden image) à l'intérieur de chaque pixel de notre image originale (cover image). Pour simplifier, les dimensions de ces 2 images seront identiques. Chaque pixel de notre image originale sera légèrement modifié : plus précisément nous allons modifier les bits de poids faible de chaque pixel. C'est la méthode LSB : Least Significant Bit.

Sur l'exemple ci-dessous, les composantes R,G et B de ces deux couleurs diffèrent de ± 2. Pourtant à l'oeil nu on ne voit pas la différence.



Pour cacher un pixel B dans un pixel A, nous allons donc remplacer les 4 bits de *poids faible* du pixel A par les 4 bits de *poids fort* du pixel B.

Par exemple si :

$$A \left| \begin{array}{lcl} R & : & 39 = \overline{0010\ 0111}^2 \\ G & : & 88 = \overline{0101\ 1000}^2 \\ B & : & 165 = \overline{1010\ 0101}^2 \end{array} \right. \quad \text{et} \quad B \left| \begin{array}{lcl} R & : & 62 = \overline{0011\ 1110}^2 \\ G & : & 73 = \overline{0100\ 1001}^2 \\ B & : & 75 = \overline{0100\ 1011}^2 \end{array} \right.$$

on remplace le pixel A' par le pixel modifié :

$$A' \left| \begin{array}{lcl} R & : & \overline{0010\ 0011}^2 = 35 \\ G & : & \overline{0101\ 0100}^2 = 84 \\ B & : & \overline{1010\ 0100}^2 = 164 \end{array} \right.$$

ce qui transforme en , le pixel caché étant .

Inversement pour retrouver l'image caché, on fait l'opération inverse : on extrait les bits de *poids faible* du pixel reçu et on complète par quatre 0.

Dans l'exemple précédent, le pixel A' devient :

$$B' \left| \begin{array}{lcl} R & : & \overline{0011\ 0000}^2 = 48 \\ G & : & \overline{0100\ 0000}^2 = 64 \\ B & : & \overline{0100\ 0000}^2 = 64 \end{array} \right.$$

donc le pixel , une fois caché, est récupéré comme le pixel .

Une image de dimension $p \times n$ sera représentée par un tableau `numpy` à trois dimensions $n \times p \times 3$ (noter l'inversion de n et p). La troisième dimension correspond à la couleur (rouge,

vers ou bleu, indice 0, 1 ou 2). Les valeurs stockées dans ce tableau sont des entiers entre 0 et 255 : on imposera donc à ces valeurs d'être du type `uint8` de numpy (entiers non signés codés sur 8 bits).

Dans la suite on suppose qu'on a importé le module `numpy`.

1. La fonction `bin` donne la représentation binaire d'un entier. Les entiers entre 0 et 255 sont représentés sous la forme d'une chaîne de caractère `'0b...'` composés de 3 à 10 caractères.
   Pour appliquer l'algorithme de stéganographie, on a besoin qu'il soient représentés par des chaînes de 8 caractères, sans le préfixe `'0b'` (les premiers caractères étant éventuellement nuls).
   (a) En utilisant la concaténation `mot1+mot2` et l'extraction `mot[n:p]` (où n et p peuvent être négatifs), écrire une ligne de commande qui transforme une chaîne de 3 à 8 caractères `0b...` en une chaîne de 8 caractères exactement, sans le préfixe `0b`, les premiers caractères étant éventuellement nuls.
   Par exemple `'0b10100'` doit devenir `'00010100'`.

   (b) En déduire une fonction `tab2bin(tableau)` qui prend entrée une variable `tableau` qui est un tableau `numpy` de dimensions $n \times p \times 3$ d'entiers de type `uint8`, les transforme en binaire avec la fonction `bin`, puis les met sous la forme de chaînes de 8 caractères.
   On rappelle que si `f` est une fonction définie sur les nombres, alors `vectorize(f)` donne une fonction définie sur les tableaux `numpy`.
   De plus, `tableau.shape` donne `[n,p,3]`.

   (c) Inversement, ces chaînes de 8 caractères peuvent être transformées en entiers avec la fonction `int(.,2)`.
   Écrire une fonction `bin2tab(tableau)` qui prend entrée un tableau `numpy` de chaînes de 8 caractères représentant en binaire des entiers entre 0 et 255, crée un tableau `numpy` appelé `tableau2` dont les valeurs sont de type `uint8`, puis affecte à chaque case de `tableau2` l'entier représenté par la case correspondant de `tableau`. En sortie cette fonction retournera la variable `tableau2`.
   On rappelle que `zeros((n,p,3),dtype=uint8)` crée un tableau `numpy` de dimensions $n \times p \times 3$, dont les valeurs sont de type `uint8`.

2. (a) Donner une fonction `steganographie(tab_image,tab_secret)` qui :
   — prend en entrée deux tableaux `numpy` de dimensions $n \times p \times 3$, appelés `tab_image` et `tab_secret`, dont les cases sont des chaînes de 8 caractères ;
   — crée un tableau `numpy` de même dimension, dont les cases sont du type `S10` (chaînes de 10 caractères), appelé `tab_encode` ;
   — applique à chaque case de `tab_image` et `tab_secret` l'algorithme de stéganographie (pour cacher `tab_secret` dans `tab_image`), et enregistre le résultat dans la case correspondante de `tab_encode`.
   En sortie la fonction doit retourner le tableau `tab_encode`.
   On rappelle que `zeros((n,p,3),dtype='|S10')` crée un tableau `numpy` de dimensions $n \times p \times 3$, dont les valeurs sont de type `S10`.

   (b) Donner une fonction `steganographie_inverse(tab_image)` qui :
   — prend en entrée un tableau `numpy` de dimensions $n \times p \times 3$, appelé `tab_image`, dont les cases sont des chaînes de 8 caractères ;

— crée un tableau numpy de même dimension, dont les cases sont du type S10 (chaînes de 10 caractères), appelé tab_decode ;
— applique à chaque case de tab_image l'algorithme de décodage et enregistre le résultat dans la case correspondante de tab_decode

En sortie la fonction doit retourner le tableau tab_decode.

Nous allons maintenant utiliser le module matplotlib.pyplot pour manipuler des images au format PNG (attention ce dernier point est indispensable). Si image.png est le nom du fichier (donc une chaîne de caractères), les commandes :

```
tableau_float=imread('image.png')
tableau_int=uint8(255*tableau_float)
```

crée le tableau numpy de dimensions $n \times p \times 3$ associé (pour image.png de dimensions $p \times n$), sous le nom tableau_int.

Ensuite imshow(tableau_int) permet de visualiser l'image.

Inversement imsave('image2.png',tableau_int) crée l'image PNG associée au tableau et l'enregistre sous le nom image2.png.

3. (a) Donner une fonction image_encode(masque, secret) qui :
   — prend en entrée deux noms d'images masque et secret qui sont des chaînes de caractères ;
   — crée les tableaux d'entiers numpy tab_masque et tab_secret associés ;
   — les transforme en tableaux de chaînes de caractères comme indiqué à la question 1. ;
   — leur applique l'algorithme de stéganographie et enregistre le résultat dans une variable tab_encode ;
   — transforme ce tableau en tableau d'entiers comme à la question 1. et enregistre le résultat dans une variable encode ;
   — affiche l'image associée et l'enregistre dans le nouveau fichier encode.png.

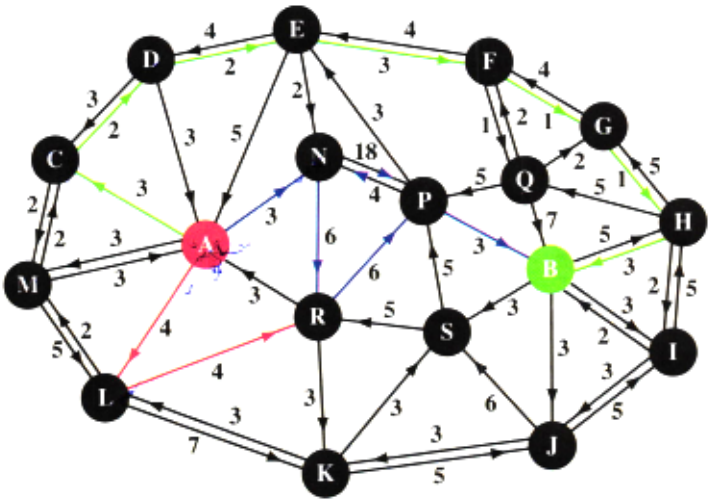
   Cette fonction ne renvoie rien en sortie.

   (b) Donner une fonction image_decode(image) qui :
   — prend en entrée un nom d'image image qui est une chaîne de caractères ;
   — crée le tableau d'entiers numpy tab_image associé ;
   — le transforme en tableaux de chaînes de caractères comme indiqué à la question 1. ;
   — lui applique l'algorithme de décodage et enregistre le résultat dans une variable tab_decode ;
   — transforme ce tableau en tableau d'entiers comme à la question 1. et enregistre le résultat dans une variable decode ;
   — affiche l'image associée et l'enregistre dans le nouveau fichier decode.png.

   Cette fonction ne renvoie rien en sortie.

4.2 Algorithme de Dijkstra

L'algorithme de Dijkstra permet de déterminer le plus court chemin d'un point à un autre. Les circuits possibles seront modélisés par un graphe orienté (les chemins peuvent parfois n'être parcouru que dans un seul sens) et pondéré (chaque chemin comporte un temps de trajet). Prenons par exemple le graphe suivant où on cherche le plus court temps de trajet de A à B :



Si on veut essayer tous les chemins possibles, on arrive très vite çà un temps colossal, et calculer un itinéraire sur un GPS pourrait prendre plus de 10^{30} années !

L'idée clé de Dijkstra est la suivante : si un chemin optimal de A vers B apsse par un sommer X, alors la partie de ce chemin de A à X est encore un itinéraire optimal de A vers X. En effet, s'il existait un chemin plus rapide de A vers X, il permettrait de réduire le trajet total de A vers B.

On associe alors, à chaque sommet X (autre que A) son *prédecesseur* $p(X)$, c'est-à-dire le sommet immédiatement précédent dans le chemin optimal menant de A à X. Dans l'exemple ci-dessus, on a $p(B) = H$, $p(H) = G$, et ainsi de suite jusqu'à $p(C) = A$ (le chemin optimal est indiqué en vert).

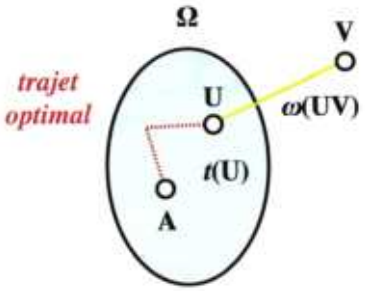
La fonction p permet de reconstruire, de proche en proche, pour chaque sommet X, un chemin optimal de A vers X. On note $t(X)$ le *temps de parcours* correspondant.

Au départ, on ne dispose pas des valeurs définitives de $t(X)$ et $p(X)$: on ne se contente que de valeurs provisoires, respectivement $\tau(X)$ et $\pi(X)$, données par le meilleur itinéraire de A vers X actuellement connu. Ces valeurs vont évoluer, par améliorations successives, vers les valeurs définitives.

Appelons Ω l'ensemble des sommets du graphe pour lesquels on sait que les fonctions t et p ont atteint leur valeurs définitives. L'algorithme consiste à faire grossir de proche en proche Ω en lui ajoutant, à chaque itération, un nouveau sommet.

Voici comment :

- Initialisation :
  - ⋆ au départ $\Omega = A$ avec $\tau(A) = t(A) = 0$ (le trajet optimal de A vers A prend un temps nul, par un chemin formé de zéro arête)) ;
  - ⋆ pour les sommets X différents de A, on ne connaît pas de chemin et on initialise donc $\tau(X)$ à $+\infty$ en convenant que, tant que $\tau(X)$ n'aura pas de valeur finie, $\pi(X)$ ne sera pas défini.
- Une itération se fait en deux étapes :
  - ⋆ D'abord, on considère toutes les arêtes UV dont l'origine U est dans Ω et l'extrémité v est à l'extérieur de $Omega$. Pour chacune d'elles, on connaît un trajet optimal de A vers U (car U est dans Ω). En prolongeant ce trajet jusqu'à V on a un temps total de parcours égal à $t(U) + \omega(UV)$, où $\omega(UV)$ est le temps de parcours de U à V (donné par le graphe).



Si cette somme est inférieure à $\tau(V)$, ce trajet pourrait être optimal vers V, et on le retient en posant $\tau(V) = t(U) + \omega(UV)$ et $\pi(V) = U$.
Ces valeurs restent toujours relatives au meilleur itinéraire actuellement connu (notons que V peut être atteint par plusieurs sommets...).

⋆ Dans une seconde étape, on choisit parmi les sommets X extérieurs à $Omega$ et tels que $\tau(X) < +\infty$ (et où $\pi(X)$ est donc défini), un X qui minimise $\tau(X)$. Le trajet optimal de A vers $\pi(X)$ (qui est dans Ω) suivi de l'arête joignant $\pi(X)$ à X est alors un trajet optimal vers X.
En effet, s'il existait un trajet plus rapide, en notant Y le prédécesseur dans ce dernier trajet, Y devrait être dans Ω (car Y a dû, lors d'une étape précédente, être inclus dans Ω), ce qui contredit la définition de X ($\tau(X)$ ne serait pas minimal).
Le sommet X ainsi trouvé est ensuite ajouté à l'ensemble Ω, et on pose $p(X) = \pi(X)$ et $t(X) = \tau(X)$.
Ceci termine l'itération.

- On recommence alors avec le nouvel ensemble Ω « grossi » qui, de proche en proche, finira aussi par contenir le point B qui nous intéresse (sauf l'arbre rend inacessible le point B partant de A), marquant la fin de l'algorithme.

L'algorithme de Dijkstra fournit l'itinéraire optimal de A vers B ainsi que les itinéraires optimaux pour tous les points qui, partant de A, sont plus rapides à atteindre que B.

Donnons les premières étapes sur l'exemple de l'énoncé :

- **Initialisation.** On initialise $\Omega = \{A\}$, $t(A) = 0$ et $\tau(X) = +\infty$ pour tout sommet X autre que A.

- **Première itération.** On considère les arêtes ayant leur origine en A, c'est-à-dire AC, AL, AM et AN, ce qui nous amène à poser :

$$\tau(C) = 3, \tau(L) = 4, \tau(M) = 3, \tau(N) = 3$$
$$\pi(C) = \pi(L) = \pi(M) = A$$

  Parmi ces quatre sommets (C, L, M et N), il faut en trouver un minimisant le temps de parcours : C par exemple.
  On pose donc $t(C) = 3$ et $p(C) = A$.
  Continuons l'algorithme avec $\Omega = \{A, C\}$, $t(A) = 0$, $t(C) = 3$, $p(C) = A$, $\tau(L) = 4$, $\tau(M) = 3$, $\tau(N) = 3$ et les autres à $+\infty$, $\pi(L) = \pi(M) = \pi(N) = A$.

- **Seconde itération.** On considère les arêtes ayant leur origine en A ou C, c'est-à-dire AL, AM, AN, CD et CM.
  La première étape de l'itération conduit à poser $\tau(D) = 6$, $\pi(D) = C$.
  Choisissons M. On pose donc $t(M) = 3$, $p(M) = A$.
  Poursuivons donc l'algorithme avec $\Omega = \{A, C, M\}$, $t(A) = 0$, $t(C) = 3$, $t(M) = 3$, $\tau(D) = 6$, $\tau(L) = 4$, $\tau(N) = 3$ et les autres à $+\infty$, $\pi(L) = \pi(N) = A$ et $\pi(D) = C$.

- **Troisième itération.** On considère les arêtes AL, AN, CD et ML. Les deux étapes conduisent à $\Omega = \{A, C, M, N\}$, $t(N) = 3$ et $p(N) = A$.

- **Quatrième itération.** Les arêtes à considérer sont AL, CD, ML, NP et NR. Le point suivant à être englobé dans Ω est L ave $t(L) = 4$ et $p(L) = A$.

Nous modéliserons le graphe orienté pondéré par sa *matrice des distances* : si il y a n sommets numérotés de 0 à $n-1$, on définit une matrice A d'ordre n (avec les lignes et colonnes numérotées à partir de 0 comme en Python) telle que a_{ij} donne le temps de parcours du sommet i vers le sommet j. Si on ne peut aller de i vers j, on pose $a_{ij} = 0$.
Les matrices seront représentés en Python par des listes de listes.

1. (a) Écrire la matrice des distances du graphe donné en exemple.

   (b) Elle est donnée dans le fichier `Exemple.txt`. Écrire un script Python qui permet de lire le fichier et d'enregistrer la matrice dans une variable `arbre` (commencer par visualiser le fichier pour comprendre sa strucutre)

2. Écrire une fonction `ind_min(liste,omega)` qui prend en entrée :
   - une liste `liste` de n nombres ;
   - une liste `omega` de nombres entre 0 et $n-1$;

   et donne en sortie l'indice `X` qui minimise `liste[k]` pour tous les `k` pris à l'extérieur de `omega`.

3. Dans cette question, on va écrire la fonction principale `dijkstra(arbre,A,B,sommets)` :
   - `arbre` est la matrice des distances ;
   - `A` est l'indice du sommet de départ ;
   - `B` est l'indice du sommet d'arrivée ;
   - `sommets` est la liste des noms de sommets : `sommets[k]` donne le nom du somme numéro `k`.

   En Python il existe un flottant qui représente $+\infty$: `infini=float('inf')`.

   (a) Initialiser les variables suivantes :
   - `n` comme le nombre de sommets ;
   - `Omega` comme une liste ne contenant que le sommet `A` ;
   - `tau` comme une liste de n nombres égaux à `infini` ;
   - `t` comme une liste de n zéros ;
   - `pi` et `p` comme deux listes de n fois `None`.

   (b) Écrire une boucle `while` effectuant l'algorithme de Dijkstra : à la fin `t[B]` donne le temps de parcours optimal de A vers B, et la trajet optimal est contenu dans la liste `p`.

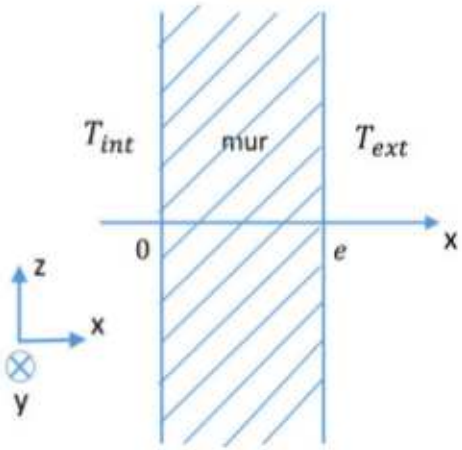
   (c) Écrire la fin du programme qui permet de récupérer dans `p` le trajet optimal et retourne en sortie de la fonction :
   - le parcours optimal sous forme d'une liste de noms de sommets ;
   - le temps de trajet optimal sous forme d'un flottant.

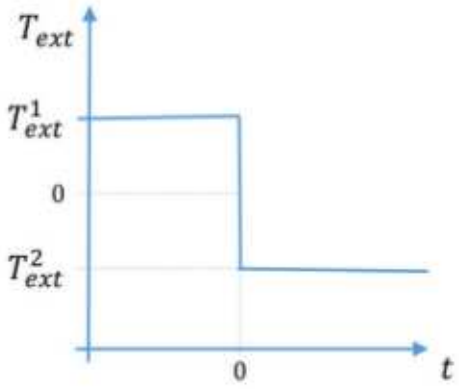
4. Tester avec l'exemple donné dans l'énoncé.

5. Que fait l'algorithme si B n'est pas accessible depuis A dans le graphe proposé ?

4.3 Équation de la chaleur

On étudie les transferts thermiques dans le mur d'une maison. La température à l'intérieur de la maison est constante dans le temps et égale à $T_{\text{int}} = 20\ °C$. Aux temps négatifs ($t < 0$), la température extérieure est égale à $T_{ext1} = 10\ °C$. À $t = 0$, elle chute brusquement à $T_{ext2} = -10\ °C$ et elle reste égale à cette valeur aux temps positifs ($t > 0$). On souhaite étudier l'évolution du profil de température dans le mur au cours du temps.





Le mur a une épaisseur $e = 40\ cm$. Les propriétés physiques du mur sont constantes : conductivité thermique $\lambda = 1{,}65\ W.m^{-1}.K^{-1}$, capacité thermique massique $c_p = 1\,000\ J.kg^{-1}.K^{-1}$, masse volumique $\rho = 2\,150\ kg.m^{-3}$.

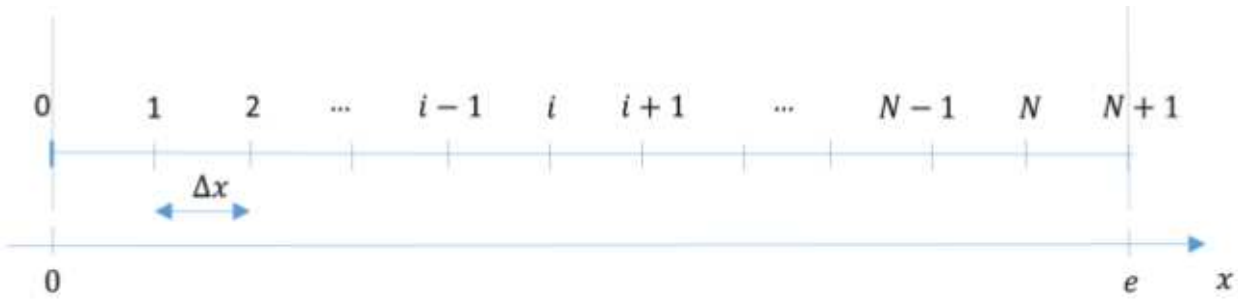
On suppose que la température dans le mur T ne dépend que du temps t et de la coordonnée x. L'équation qui décrit l'évolution de cette température est :

$$\alpha = \frac{\partial T}{\partial t} = \frac{\partial^2 T}{\partial x^2} \qquad (1)$$

où $\alpha = \rho * c_p/\lambda$. Les conditions aux limites sont :

$$\begin{aligned} &T(0,t) = T_{int} && \text{pour tout } t > 0 \\ &T(e,t) = T_{ext2} && \text{pour tout } t > 0 \\ &T(x,0) = \frac{T_{ext1} - T_{int}}{e}.x + T_{int} \end{aligned}$$

Pour résoudre le problème de manière numérique, on discrétise l'espace et le temps. On divise l'intervalle $[0, e]$, représentant l'épaisseur du mur, en $N+2$ points, numérotés de 0 à $N+1$, régulièrement espacés de Δx. Cette division est appelée « discrétisation ». La distance Δx est appelée le « pas d'espace ». À l'intérieur du mur (frontières intérieure et extérieure exclues) se trouvent donc N points. On cherche à obtenir la température en ces points particuliers à chaque instant.



On remarquera que $\Delta x = \dfrac{e}{N+1}$ et, pour $i \in [\![0, N+1]\!]$: $x_i = i * \Delta x$.

Le temps est discrétisé en $ItMax$ intervalles de durée Δt et on ne s'intéresse au profil de température qu'aux instants particuliers $t_k = k.\Delta t$. L'intervalle élémentaire de temps Δt est appelé le « pas de temps ».

On notera T_i^k la température $T(x_i, t_k)$, évaluée au point d'abscisse x_i à l'instant t_k. Ainsi le vecteur $(T_0^k, T_1^k, \ldots, T_{N+1}^k)$ représente le profil de température dans le mur à l'instant t_k.

1. **Méthode utilisant un schéma explicite.**

   (a) Justifier que l'équation (1) peut s'écrire sous la forme :

$$\begin{array}{l} T_i^{k+1} = r.T_{i-1}^k + (1-2r).T_i^k + r.T_{i+1}^k \quad \text{pour tout } i \in [\![1, N]\!] \\ T_0^k = T_{int} \\ T_{N+1}^k = T_{ext2} \end{array}$$

   où on a posé $r = \dfrac{\Delta t}{\alpha.(\Delta x)^2}$, et $k \in \mathbb{N}$.

   Cette équation est appelée *schéma d'Euler explicite*. Si on connaît la température en tous les points $x_1, x_2, \ldots, x_N$ à l'instant t_k, on peut calculer grâce à elle la température en tous les points à l'instant ultérieur t_{k+1}.

   On veut élaborer une fonction `schema_explicite(T0,alpha,e,Delta_t,Tint,Text2)` permettant de calculer la température en chaque point au cours du temps. La variable `T0` est un vecteur de dimension N, contenant les valeurs de la température aux points de discrétisation à l'instant initial. Les autres variables d'entrée sont les constantes définies dans l'énoncé. Au sein de la fonction, un algorithme calculera itérativement la température avec un nombre maximal d'itérations `ItMax=\numprint{2000}`. En sortie de la fonction, on récupèrera le nombre d'itérations réellement effectuées `nbIter`, et une matrice `T_tous_k` de dimensions $N \times ItMax$. Chaque colonne de cette matrice contient le vecteur $\mathbf{T^k}$ dont les éléments sont les valeurs de la température aux N points $x_1, \ldots, x_N$ (points à l'intérieur du mur) à l'instant k :

$$\mathbf{T^k} = \begin{pmatrix} T_1^k \\ T_2^k \\ \vdots \\ T_N^k \end{pmatrix} \qquad \text{et} \qquad \mathtt{T_tous_k} = \begin{pmatrix} T_1^1 & T_1^2 & \ldots & T_1^k & \ldots & T_1^{k-1} & T_1^k \\ T_2^1 & T_2^2 & \ldots & T_2^k & \ldots & T_2^{k-1} & T_2^k \\ \ldots & \ldots & \ldots & \ldots & \ldots & \ldots & \ldots \\ T_N^1 & T_N^2 & \ldots & T_N^k & \ldots & T_N^{k-1} & T_N^k \end{pmatrix}$$

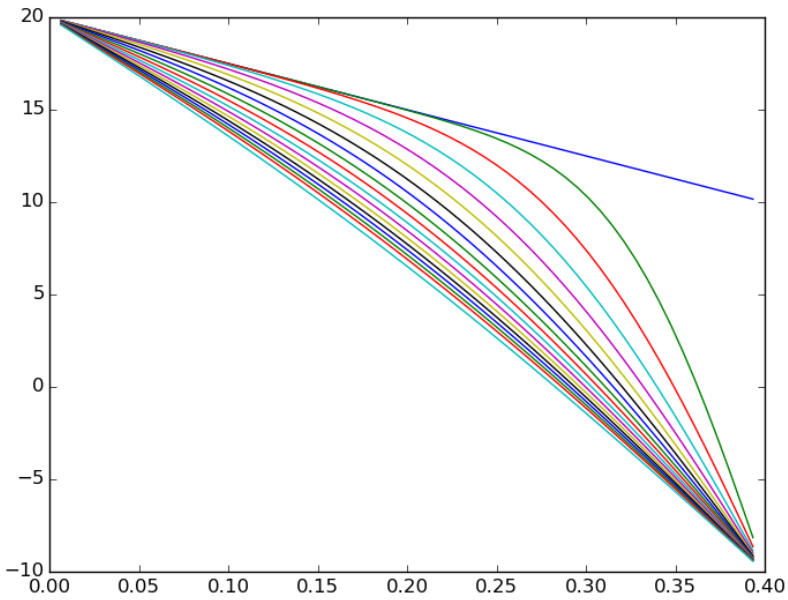
   On souhaite arrêter le calcul lorsque la température ne varie presque plus dans le temps. Dans ce but, on évaluera la norme de $\mathbf{T^k} - \mathbf{T^{k-1}}$ à chaque itération.

   (b) Écrire la fonction `schema_explicite(T0,alpha,e,Delta_t,Tint,Text2)` en suivant les indications suivantes :
   - définir les constantes `ItMax`, `N`, `Delta x` et `r` ;
   - écrire un test qui affiche une alerte si $r \geq 0.5$ (on admet que dans ce cas le schéma ne converge pas vers la solution) ;
   - créer avec la matrice `T_tous_k` de dimensions $N \times ItMax$ en la remplissant de zéros ;
   - remplacer la première colonne de `T_tous_k` par `T0` ;

- calculer le profil de température à l'instant $k = 1$ (ie $t = \Delta t$) en disinguant les cas $i = 1, 2 \leq i \leq N-1$ et $i = N$, et l'affecter à la deuxième colonne de `T_tous_k` ;
- élaborer une boucle permettant de calculer itérativement le profil de température aux instants $t_k = k.\Delta t$ (avec $k \geq 2$), boucle interrompue lorsque le la norme de $\mathbf{T^k} - \mathbf{T^{k-1}}$ deviendra inférieur à 10^{-2} ou lorsque le nombre d'itérations atteindra la valeur `ItMax` ;
- retourner en sortie le nombre d'itérations `nbIter` et la matrice `T_tous_k`.

(c) Pour $N = 60$ et $\Delta t = 25\,s$, faire afficher :
- le temps en heures au bout duquel le régime permanent est établi ;
- sur un même graphique le profil de température en fonction de `x` tous,les 100 pas de temps.

```
Le régime permanent est atteint au bout de 12.4375 heures.
```



2. **Méthode utilisant un schéma implicite.**
Le schéma explicité ne converge que si le pas de temps Δt est suffisamment faible par rapport au pas d'espace `Delta x`. Si l'on souhaite effectuer un calcul pour un temps physique long, beaucoup d'itérations seont nécessaires et le temps de calcul sera très long. C'est pourquoi on préfère d'autres types de schémas appelés implicites.
La dérivée partielle seconde par rapport à x de la température sera évaluée au point d'abscisse x_i et à l'instant $k+1$:

$$\frac{\partial^2 T}{\partial x^2}(x,t) \approx \frac{\partial^2 T}{\partial x^2}(x_i, t_{k+1})$$

et la dérivée partielle par rapport à t est évaluée au point d'abscisses x_i à l'instant k :

$$\frac{\partial T}{\partial t}(x,t) \approx \frac{\partial T}{\partial t}(x_i, t_{k+1})$$

(a) Justifier que l'équation (1) peut alors s'écrire sous la forme :

$$\begin{aligned} &T_i^k = -r.T_{i-1}^{k+1} + (1+2r).T_i^{k+1} - r.T_{i+1}^{k+1} \quad \text{pour tout } i \in [\![1, N]\!] \\ &T_0^k = T_{int} \\ &T_{N+1}^k = T_{ext2} \end{aligned}$$

où on a posé $r = \dfrac{\Delta t}{\alpha.(\Delta x)^2}$, et $k \in \mathbb{N}$.

Cette équation est appelée *schéma implicite* car la température à l'instant t_k est exprimée en fonction de la température à l'instant ultérieur t_{k+1}.
Le système d'équations ainsi obtenu peut être écrit sous la forme :

$$M.\mathbf{T^{k+1}} = \mathbf{T^k} + r.\mathbf{v}$$

où M est une matrice carrée $N \times N$, et $\mathbf{v}$ est un vecteur de taille N faisant intervenir les conditions aux limites.

(b) Préciser l'expression de la matrice `M` et l'expression du vecteur `v`.

À chaque pas de temps, il faut inverser le système matriciel :

$$M.\mathbf{T^{k+1}} = \mathbf{T^k} + r.\mathbf{v}$$

pour obtenir $\mathbf{T^{k+1}}$ à partir de $\mathbf{T^k}$.

Pour cela, on va utiliser l'algorithme de Thomas qui permet de résoudre un système matriciel tridiagonal de la forme :

$$M.\mathbf{u} = \mathbf{d}$$

où M est une matrice de dimensions $N \times N$ tridiagonale, et $\mathbf{u}$ et $\mathbf{d}$ des matrices colonnes de dimension N.

Dans cet algorithme, on calcule d'abord les coefficients suivants :

$$\begin{aligned} c'_1 &= \frac{c_1}{b_1} \\ c'_i &= \frac{c_i}{b_i - a_i c'_{i-1}} \quad \text{pour } i = 2, 3, \ldots, N-1 \end{aligned}$$

et

$$\begin{aligned} d'_1 &= \frac{d_1}{b_1} \\ d'_i &= \frac{d_i - a_i d'_{i-1}}{b_i - a_i c'_{i-1}} \quad \text{pour } i = 2, 3, \ldots, N \end{aligned}$$

Les inconnues u_1 ,u_2, ..., u_N sont alors obtenues par les formules :

$$\begin{aligned} u_N &= d'_N \\ u_i &= d'_i - c'_i u_{i+1} \quad \text{pour } i = N-1, N-2, \ldots, 2, 1 \end{aligned}$$

(c) En utilisant l'algorithme de Thomas, écrire une fonction `CalcTkp1(M,d)` qui permet de calculer le vecteur `u`.

(d) Écrire la fonction `schema_implicite(T0,alpha,e,Delta_t,Tint,Text2)` en suivant les indications suivantes :
- définir les constantes `ItMax`, `N`, `Delta x` et `r` ;
- créer avec la matrice `T_tous_k` de dimensions $N \times ItMax$ en la remplissant de zéros ;
- remplacer la première colonne de `T_tous_k` par `T0` ;
- calculer le profil de température à l'instant $k = 1$ (ie $t = \Delta t$) en disinguant les cas $i = 1$, $2 \leq i \leq N-1$ et $i = N$, et l'affecter à la deuxième colonne de `T_tous_k` ;

- élaborer une boucle permettant de calculer itérativement le profil de température aux instants $t_k = k.\Delta t$ (avec $k \geq 2$), boucle interrompue lorsque le la norme de $\mathbf{T^k} - \mathbf{T^{k-1}}$ deviendra inférieur à 10^{-2} ou lorsque le nombre d'itérations atteindra la valeur `ItMax` ;
- retourner en sortie le nombre d'itérations `nbIter` et la matrice `T_tous_k`.

(e) Pour $N = 60$ et $\Delta t = 25\,s$, faire afficher :
- le temps en heures au bout duquel le régime permanent est établi ;
- sur un même graphique le profil de température en fonction de `x` tous,les 100 pas de temps.

```
Le régime permanent est atteint au bout de 12.4444444444 heures.
```