

04\_info\_2601172236

Synthèse  
**PYTHON 3**

CPGE TSI

JEAN DECROOCQ

# TABLE DES MATIÈRES

<b>1</b>	<b>Fondements</b>	<b>3</b>
1.1	Variables et types . . . . .	3
1.2	Fonctions . . . . .	3
1.3	Algorithmique et programmation . . . . .	4
<b>2</b>	<b>Tableaux Numpy ///</b>	<b>5</b>
2.1	Pour commencer . . . . .	5
2.2	Les tableaux numpy . . . . .	5
2.3	Opérations sur les tableaux numpy . . . . .	5
<b>3</b>	<b>Compléments sur les listes Python et chaînes de caractères</b>	<b>7</b>
3.1	Listes Python . . . . .	7
3.2	Chaînes de caractères . . . . .	8
<b>4</b>	<b>Dictionnaires ///</b>	<b>9</b>
4.1	Création et modification d'un dictionnaire . . . . .	9
4.2	Commandes principales . . . . .	9
<b>5</b>	<b>Modules et lecture de fichiers ///</b>	<b>10</b>
5.1	Les modules . . . . .	10
5.2	Lecture d'un fichier texte de données . . . . .	10
5.3	Tracé des courbes correspondantes . . . . .	11
<b>6</b>	<b>Algorithmes Gloutons et Dichotomiques ///</b>	<b>12</b>
6.1	Algorithmes gloutons et optimisation . . . . .	12
6.2	Algorithmes dichotomiques . . . . .	12
<b>7</b>	<b>Récursivité ///</b>	<b>14</b>
7.1	Principe des fonctions récursives . . . . .	14
7.2	Analyse des fonctions récursives . . . . .	14
<b>8</b>	<b>Matrices de pixels et images</b>	<b>16</b>
8.1	Formats de représentation d'image . . . . .	16
8.2	Manipulation des matrices de pixels . . . . .	16
8.3	Transformation d'images . . . . .	16
8.4	Modification par convolution : filtrage . . . . .	17
8.5	Détection de contours . . . . .	17
<b>9</b>	<b>Annexe</b>	<b>19</b>
9.1	Compléments . . . . .	19

# 1 FONDEMENTS

## 1.1 Variables et types

- Une variable constitue une référence vers un objet stocké en mémoire. L'affectation, réalisée via le signe `=`, lie un identifiant à une valeur. Le typage est dynamique : la nature de la variable est déterminée par la valeur qu'elle reçoit au moment de l'exécution.
- Les nombres entiers (`int`) supportent les opérations classiques `+`, `-`, `*`, `/` ainsi que la division euclidienne `//`, le modulo `%` et l'exponentiation `**`.
- Les nombres flottants (`float`) représentent les réels approximativement pour certaines valeurs décimales. Ils peuvent s'écrire en notation scientifique : `1.5e4` pour  $1,5 \times 10^4$ .
- Les booléens (`bool`) ne peuvent prendre que deux états : `True` ou `False`. Ils sont le résultat d'opérations de comparaison (`==`, `!=`, `<`, `>=`) et se combinent à l'aide des connecteurs logiques `not`, `and` et `or`.
- Les chaînes de caractères (`str`) sont des séquences immuables. On ne peut modifier un caractère directement (`s[0] = 'a'` lève une erreur). La concaténation s'effectue avec `+` et la répétition avec `*`.
- Les listes (`list`) sont des séquences mutables, ordonnées et hétérogènes, définies entre crochets. L'accès aux éléments se fait par un indice entier.

```
1 L = [10, 20, 30]
2 L[0] = 5           # Modification : L vaut [5, 20, 30]
3 dernier = L[-1]   # Accède au dernier élément (30)
```

- Il est possible de convertir une valeur d'un type vers un autre (casting).
  - `int(x)` : Convertit en entier.  
Un flottant et tronqué : `int(3.9)` donne `3`, `int(-2.7)` donne `-2`.  
Une chaîne doit représenter un entier : `int("42")` donne `42`, sinon une `ValueError` est levée : `int("3.5")` échoue.
  - `float(x)` : Convertit en flottant : `float("1e-4")` donne `0.0001`.
  - `str(x)` : Transforme n'importe quel objet en sa représentation textuelle, ce qui est indispensable pour l'affichage ou la concaténation.
  - `bool(x)` : Convertit en booléen. En Python, la plupart des valeurs sont considérées comme vraies (`True`), sauf les valeurs "vides" ou nulles qui valent `False` : le nombre `0`, `0.0`, la chaîne vide `" "`, les listes vides `[]`, et la valeur `None`.

## 1.2 Fonctions

- Une fonction encapsule un bloc d'instructions afin de le rendre réutilisable. Elle est définie par le mot-clé `def`, suivi de son nom et de ses paramètres entre parenthèses. Le bloc d'instructions constituant le corps de la fonction doit impérativement être indenté.
- L'instruction `return` interrompt l'exécution de la fonction et renvoie la valeur spécifiée à l'endroit où la fonction a été appelée. Une fonction sans instruction `return` (ou avec un `return` vide) renvoie implicitement la valeur `None`.

```
1 def f(x, a):
2     y = x ** 2 + a
3     return y
```

- Les variables définies à l'intérieur d'une fonction ont une portée locale : elles n'existent que durant l'exécution de celle-ci et sont détruites ensuite. Elles ne peuvent pas modifier directement une variable globale immuable, sauf usage explicite (et déconseillé) du mot-clé `global`.

### 1.3 Algorithmique et programmation

- L'instruction conditionnelle dirige le flot d'exécution. Si la condition du `if` est fausse, le programme teste les éventuels `elif` successifs. Si aucune condition n'est vérifiée, le bloc `else` est exécuté.

```
1 if x > 0:  
2     signe = 1  
3 elif x < 0:  
4     signe = -1  
5 else:  
6     signe = 0
```

- La boucle bornée `for` est utilisée pour parcourir un itérable (liste, chaîne, tuple) ou réaliser un nombre d'itérations déterminé. On l'associe souvent au générateur `range(start, stop, step)`.
  - `range(5)` génère : 0, 1, 2, 3, 4.
  - `range(1, 5)` génère : 1, 2, 3, 4.
  - `range(0, 10, 2)` génère : 0, 2, 4, 6, 8.
- La boucle non bornée `while` répète un bloc d'instructions tant qu'une condition booléenne reste vraie.

```
1 n = 0  
2 while n < 10:  
3     print(n)  
4     n = n + 1 # Indispensable pour que la condition devienne fausse
```

## 2 TABLEAUX NUMPY ///

### 2.1 Pour commencer

- Le module `numpy` est essentiel pour le calcul scientifique nécessitant la manipulation de grandes quantités de données. Il repose sur trois principes d'efficacité : le stockage sous forme de tableaux `ndarray`, la limitation des copies mémoire, et l'utilisation de fonctions vectorialisées pour éviter les boucles.
- Ces optimisations imposent des contraintes : les tableaux sont constitués d'éléments de même type (homogènes, souvent `float` ou `int64`) et leur taille est fixée à la création (on ne peut pas augmenter la taille comme pour une liste).
- Le chargement du module se fait traditionnellement via l'instruction : `import numpy as np`.

### 2.2 Les tableaux numpy

- La création basique se fait via `np.array()` à partir d'une liste. L'attribut `.dtype` indique le type commun (ex : `int64`), différent des listes Python.
- Pour les vecteurs (1D), on utilise `np.arange(start, stop, step)` qui accepte des pas flottants, ou `np.linspace(start, stop, num)` pour obtenir `num` points équitablement répartis (bornes incluses).
- Des matrices spéciales sont prédéfinies : `np.ones()`, `np.zeros()`, `np.eye()` (identité) et `np.diag()`. Le module `np.random` permet de générer des tableaux aléatoires (uniformes, binomiaux, géométriques...). La lecture de fichiers CSV se fait avec `np.genfromtxt`.
- Un tableau possède des attributs clés : `dtype`, `size` (nombre d'éléments), `shape` (tuple des dimensions) et `ndim`. L'attribut `shape` est mutable : on peut redimensionner un tableau via la méthode `reshape()`.
- Le stockage en mémoire est contigu. L'attribut `dtype` est immuable pour garantir la cohérence de l'espace mémoire.
- Le *slicing* (tranchage) s'étend à plusieurs dimensions (ex : `T[:, 0]` pour la première colonne). Attention : le slicing renvoie une vue et non une copie. Modifier une tranche modifie le tableau d'origine. Pour copier, il faut utiliser `.copy()`.
- Les masques (indexation booléenne) permettent de filtrer un tableau. Si `b` est un tableau de booléens de même format que `a`, alors `a[b]` extrait les éléments où `b` est `True`.

### 2.3 Opérations sur les tableaux numpy

- Les opérations arithmétiques usuelles (`+`, `*`, `**...`) s'appliquent terme à terme. C'est la vectorisation.
- Il ne faut pas faire de copies inutiles de grands tableaux. Les opérations se font souvent au travers de vues.
- Les fonctions universelles s'appliquent aussi terme à terme. On peut vectoriser une fonction personnelle avec `np.vectorize`.
- Des méthodes statistiques sont disponibles : `max`, `min`, `sum`, `prod`, `mean` (moyenne arithmétique), `var` (variance), `std` (écart-type).

- Pour l'algèbre linéaire, le produit matriciel se fait avec `np.dot(a, b)`. Le produit scalaire canonique utilise `np.vdot` et le produit vectoriel `np.cross`. La transposée s'obtient avec `.T` ou `.transpose()`.
- Le sous-module `np.linalg` fournit des outils avancés : `solve` (résolution de système), `inv` (inverse), `det` (déterminant), `norm` (norme) et `eig` (éléments propres).
- La classe `Polynomial` (du module `numpy.polynomial`) permet de manipuler formellement des polynômes (racines, dérivées, primitives) définis par leurs coefficients.

### 3 COMPLÉMENTS SUR LES LISTES PYTHON ET CHAÎNES DE CARACTÈRES

#### 3.1 Tableaux et listes Python

- Les listes Python (`list`) sont des structures de données mutables, ordonnées et hétérogènes. Bien que souvent appelées "listes", elles sont implémentées sous forme de tableaux dynamiques contenant des références vers les objets.
- La construction d'une liste peut se faire par extension ou, de manière plus élégante et performante, par compréhension.

```
1 L1 = [0, 1, 4, 9, 16] # Par extension
2 L2 = [k**2 for k in range(5)] # Par comprehension (identique à L1)
3 L3 = [x for x in L1 if x % 2 == 0] # Avec filtrage (nombres pairs)
```

- Lorsqu'une liste contient une autre liste, on accède à ses éléments ainsi : `L[a][b]`.
- Le mécanisme de tranchage (*slicing*) permet d'extraire une sous-partie de la liste en créant une nouvelle liste (copie). La syntaxe générale est `L[début:fin:pas]`.
  - `L[i:j]` : sélectionne les éléments de l'indice *i* inclus à *j* exclu.
  - `L[i:]` : sélectionne de l'indice *i* jusqu'à la fin.
  - `L[:j]` : sélectionne du début jusqu'à l'indice *j* exclu.
  - `L[::-1]` : crée une copie renversée de la liste.
- Les listes étant mutables, elles disposent de méthodes agissant *in-place* (modifiant l'objet sans le renvoyer). Les plus courantes sont :
  - `L.append(x)` : ajoute l'élément *x* à la fin de la liste.
  - `L.pop()` : supprime et renvoie le dernier élément. `L.pop(i)` supprime l'élément à l'indice *i*.
  - `L.sort()` : trie la liste en place.
  - `L.reverse()` : inverse l'ordre des éléments en place.
- Voici quelques opérations usuelles :
  - `L1 + L2`, `L * n` : concaténation, répétition.
  - `len(L)` : longueur/nombre d'éléments.
  - `max(L)`, `min(L)` : maximum, minimum.
  - `sum(L)` : somme des éléments.
  - `x in L` : test d'appartenance.
- Un point de vigilance majeur concerne l'aliasing. L'instruction `L2 = L1` ne copie pas la liste, mais crée une seconde référence vers le même objet en mémoire :

```
1 L1 = [1, 2, 3]
2 L2 = L1
3 L2[0] = 99
4 print(L1) # Affiche [99, 2, 3], L1 est modifiée aussi !
```

Pour obtenir une copie indépendante, il faut utiliser le tranchage complet `L[:]` ou la méthode `L.copy()` :

```
1 L1 = [1, 2, 3]
2 L2 = L1[:]           # ou L2 = L1.copy()
3 L2[0] = 99
4 print(L1) # Affiche [1, 2, 3], L1 reste inchangée
```

### 3.2 Chaînes de caractères

- Les chaînes de caractères (`str`) sont des séquences ordonnées de caractères Unicode. Contrairement aux listes, elles sont immuables : il est impossible de modifier un caractère par affectation directe (`ch[0] = 'a'` lève une erreur `TypeError`).
- On définit une chaîne de caractère en l'entourant de guillemets simples, doubles, ou trois guillemets simples ou doubles. L'utilisation de guillemets simple permet d'utiliser des guillemets doubles dans la chaîne et vice-versa.
- La syntaxe de tranchage (*slicing*) et les fonctions universelles de séquences (`len()`, `in`) fonctionnent exactement comme pour les listes. La conversion d'un objet en chaîne se fait via le constructeur `str()`.
- Le langage propose des méthodes spécifiques pour la manipulation de texte :
  - `s.find(motif)` : renvoie l'indice de la première occurrence du motif (ou -1 si absent).
  - `s.count(motif)` : compte le nombre d'occurrences du motif.
  - `s.strip()` : retire les espaces (et caractères invisibles) en début et fin de chaîne.
  - `s.replace(old, new)` : remplace toutes les occurrences de la sous-chaîne `old` par `new`.
- Deux méthodes sont essentielles pour passer du type `str` au type `list` et inversement :
  - `sep.join(liste)` : concatène les éléments d'une liste de chaînes en les séparant par la chaîne `sep`.
  - `s.split(sep)` : découpe la chaîne `s` en une liste de sous-chaînes, en utilisant `sep` comme délimiteur.

```
1 phrase = "Sciences du Numerique"
2 mots = phrase.split(" ") # ['Sciences', 'du', 'Numerique']
3 reconst = "-".join(mots) # "Sciences-du-Numerique"
```

# 4 DICTIONNAIRES ///

## 4.1 Cration et modication d'un dictionnaire

- Le type `dict` (dictionnaire ou table d’association) permet de stocker des couples cle-valeur. Contrairement aux listes indexes par des entiers, les dictionnaires sont indexes par des cles appartenant  un ensemble  $C$ , associes  des valeurs d’un ensemble  $V$ .
- Une contrainte fondamentale concerne les cles : elles doivent tre de type hachable (immuable), comme les `int`, `float`, `str` ou `tuple`. Les valeurs, en revanche, peuvent tre de n’importe quel type et sont mutables.
- L’intert majeur de cette structure (implemente par table de hachage) est l’efficacit : l’accès  une valeur via sa cle se fait en temps constant moyen  $O(1)$ , quelle que soit la taille du dictionnaire.
- La cration s’effectue via des accolades `{}` ou le constructeur `dict()`. On peut deфинir un dictionnaire par extension ou par comprension.

```

1 d1 = {} # Dictionnaire vide
2 d2 = { 'argent': 'silver', 'or': 'gold'} # Par extension
3 d3 = {x: x**2 for x in range(5)} # Par comprehension
4 # d3 vaut {0: 0, 1: 1, 2: 4, 3: 9, 4: 16}

```

- L’ajout ou la modication d’une entre utilise la syntaxe d’affectation `d[cle] = valeur`. Si la cle existe djà, l’ancienne valeur est crase (unicit des cles). Si elle n’existe pas, une nouvelle entre est cree.

## 4.2 Commandes principales

- La fonction `len(d)` renvoie le nombre de paires cle-valeur stockes.
- Le test d’appartenance `k in d` vriﬁe si la cle `k` est prente dans le dictionnaire (opration trs rapide). Attention, cela ne teste pas la presence des valeurs.
- Pour parcourir un dictionnaire, on dispose de trois methodes renvoyant des *vues* (objets iterables dynamiques) :
  - `d.keys()` : itere sur les cles (comportement par dfault d’une boucle `for k in d`).
  - `d.values()` : itere sur les valeurs.
  - `d.items()` : itere sur les couples (cle, valeur).

```

1 d = { 'pomme': 2, 'poire': 5}
2 for k, v in d.items():
3     print(k, "->", v) # Affiche "pomme -> 2" etc.

```

- Pour rcuprer une valeur, deux approches existent :
  - `d[k]` : renvoie la valeur associe  `k`, mais lve une erreur `KeyError` si la cle est absente.
  - `d.get(k)` : renvoie la valeur si elle existe, ou `None` (ou une valeur par dfault spcifie) sinon, sans provoquer d’erreur.
- La suppression d’une entre se fait via la methode `d.pop(k)`, qui supprime la cle `k` et renvoie la valeur associe. Si la cle n’existe pas, une erreur est leve.

# 5 MODULES ET LECTURE DE FICHIERS ///

## 5.1 Les modules

- Un module est une bibliothèque contenant un ensemble de fonctions et de variables pré-définies, permettant d'étendre les fonctionnalités natives de Python. Des modules standards comme `math`, `random` ou `time` sont disponibles immédiatement, tandis que d'autres comme `numpy` ou `matplotlib` doivent être installés.
- L'importation d'un module peut se faire de plusieurs manières. La méthode recommandée est l'import global, éventuellement avec un alias pour alléger l'écriture.

```

1 import math
2 y = math.cos(math.pi)
3
4 import numpy as np      # Alias standard
5 T = np.array([1, 2, 3])

```

- Il est possible d'importer spécifiquement certaines fonctions dans l'espace de nommage courant avec `from ... import ...`. En revanche, l'importation totale via `from module import *` est fortement déconseillée car elle pollue l'espace de noms et peut créer des conflits (aliasing) difficiles à déboguer.
- La fonction `help(module)` ou `help(fonction)` permet d'accéder à la documentation intégrée, décrivant les paramètres attendus et le type de retour.

## 5.2 Lecture d'un fichier texte de données

- La gestion de l'encodage est primordiale lors de l'ouverture d'un fichier texte. Bien que la norme actuelle soit l'UTF-8 (compatible Unicode), des formats hérités (ASCII) ou propriétaires (Windows CP1252) persistent. Pour éviter les erreurs d'interprétation des accents, il convient de préciser systématiquement l'argument `encoding="utf-8"`.
- Le format CSV (*Comma Separated Values*) est un standard de fichier texte où les données sont séparées par un délimiteur (virgule, point-virgule ou tabulation).
- La lecture "naïve" en Python pur s'effectue avec l'instruction `with open(...) as f:`, qui garantit la fermeture du fichier après lecture. On utilise ensuite `f.readlines()` pour obtenir une liste de chaînes, qu'il faut nettoyer (`.strip()`) et découper (`.split()`).
- Le module `csv` simplifie cette démarche grâce à l'objet `csv.reader`, qui gère automatiquement le découpage selon un délimiteur donné. Cependant, les données lues restent des chaînes de caractères qu'il faut convertir explicitement (en `float` ou `int`).
- La méthode avancée (et recommandée pour le calcul scientifique) repose sur la fonction `np.loadtxt` de Numpy. Elle charge directement les données dans un tableau, gère la conversion de type, et permet de sélectionner les colonnes ou d'ignorer les en-têtes.

```

1 import numpy as np
2
3 # Chargement direct dans des tableaux (unpack=True)
4 temps, vitesse = np.loadtxt(
5     "data.csv",
6     delimiter=";",      # separateur
7     skiprows=1,         # ignore la 1ere ligne (en-tetes)
8     usecols=(0, 2),    # selectionne colonnes 0 et 2

```

```

9     encoding="utf-8",
10    unpack=True           # transpose pour separer les vecteurs
11 )

```

### 5.3 Tracé des courbes correspondantes

- La bibliothèque `matplotlib.pyplot` (importée usuellement sous l'alias `plt`) est l'outil standard pour la représentation graphique.
- La construction d'un graphique suit généralement une séquence ordonnée : création de la figure, tracé des données, ajout de métadonnées (titres, légendes, grille) et enfin affichage.

```

1 import matplotlib.pyplot as plt
2
3 plt.figure() # Creation d'une nouvelle fenetre
4 # Trace de la vitesse en fonction du temps
5 plt.plot.temps, vitesse, label="Vitesse (m/s)", color="blue")
6
7 plt.title("Evolution de la vitesse")
8 plt.xlabel("Temps (s)")
9 plt.ylabel("Vitesse (m/s)")
10 plt.grid()      # Affiche la grille
11 plt.legend()    # Affiche la legende definie dans plot()
12 plt.show()      # Bloque l'execution et affiche la fenetre

```

- D'autres types de tracés sont disponibles pour des besoins spécifiques : `plt.semilogx` et `plt.loglog` pour les échelles logarithmiques (Bode), `plt.scatter` pour les nuages de points, ou `plt.hist` pour les histogrammes.

# 6 ALGORITHMES GLOUTONS ET DICHOTOMIQUES ///

## 6.1 Algorithmes gloutons et optimisation

- Les algorithmes gloutons s'inscrivent dans le cadre de la résolution de problèmes d'optimisation. Un problème d'optimisation consiste à choisir, parmi un ensemble de solutions possibles, celle qui maximise un gain (fonction-objectif) ou minimise un coût (fonction-coût).
- L'ensemble des solutions qui respectent les contraintes imposées par le problème est appelé l'ensemble admissible. Une solution qui répond au critère d'optimisation (le meilleur score possible) est une solution globale.
- Quelques exemples classiques de problèmes d'optimisation :
  - Le problème du sac à dos (maximiser la valeur des objets emportés sous contrainte de poids).
  - Le problème du rendu de monnaie (minimiser le nombre de pièces pour atteindre une somme).
  - Le problème du voyageur de commerce (minimiser la distance pour visiter un ensemble de villes).
- Le principe d'un algorithme glouton est de faire, à chaque étape, le choix qui semble le meilleur localement (le choix optimal à l'instant  $t$ ), sans jamais revenir sur une décision prise, dans l'espoir que cette suite de choix locaux mène à l'optimum global. Notez que cela ne garantit pas toujours de trouver la solution optimale absolue, mais fournit souvent une solution approchée acceptable rapidement.

## 6.2 Algorithmes dichotomiques

- La méthode dichotomique (du grec "couper en deux") est une stratégie de recherche efficace qui consiste à réduire de moitié l'espace de recherche à chaque étape. Elle s'applique principalement dans deux contextes : la recherche dans une liste **triée** et la recherche de racine d'une fonction monotone.
- **Recherche dans une liste triée** : On cherche un élément  $x_0$  dans une liste  $L$  triée. On compare  $x_0$  avec l'élément central de la liste. Si  $x_0$  est plus petit, on ne cherche que dans la moitié gauche ; s'il est plus grand, dans la moitié droite. On répète le processus jusqu'à trouver l'élément ou épuiser la liste.
- Cette méthode est beaucoup plus efficace qu'une recherche séquentielle : sa complexité est logarithmique ( $O(\log_2 n)$ ).

```

1 def recherche_dichotomie(L, x0):
2     """ Recherche x0 dans une liste L triee. Renvoie un boolean. """
3     g = 0                      # Indice gauche
4     d = len(L) - 1              # Indice droit
5     found = False
6
7     while g <= d and not found:
8         m = (g + d) // 2        # Indice milieu (division entiere)
9         if x0 == L[m]:
10             found = True
11         elif x0 < L[m]:
12             d = m - 1            # On cherche a gauche
13         else:

```

```

14         g = m + 1      # On cherche a droite
15
16     return found

```

- **Recherche de racine (Méthode de la bisection)** : Il s’agit de trouver une solution approchée de l’équation  $f(x) = 0$  sur un intervalle  $[a, b]$ . Le principe repose sur le Théorème des Valeurs Intermédiaires (TVI). Si  $f$  est continue et que  $f(a)$  et  $f(b)$  sont de signes opposés ( $f(a) \cdot f(b) < 0$ ), alors il existe au moins une racine dans l’intervalle.
- L’algorithme calcule le milieu  $m = \frac{a+b}{2}$ . Si  $f(a)$  et  $f(m)$  sont de signes contraires, la racine est dans  $[a, m]$ , sinon elle est dans  $[m, b]$ . On réduit ainsi la taille de l’intervalle par 2 à chaque itération jusqu’à ce que sa largeur soit inférieure à une précision  $\epsilon$  donnée.

```

1 def zero_dichotomie(f, a, b, epsilon):
2     """ Trouve une racine de f dans [a, b] avec precision epsilon """
3     val_g = a
4     val_d = b
5     while (val_d - val_g) > epsilon:
6         m = (val_g + val_d) / 2
7         if f(val_g) * f(m) <= 0:    # Changement de signe a gauche
8             val_d = m
9         else:                      # Changement de signe a droite
10            val_g = m
11    return (val_g + val_d) / 2

```

- La bibliothèque `scipy` propose une implémentation optimisée de cet algorithme via la fonction `bisect`.

```

1 import scipy.optimize as spo
2 # Recherche de la racine de f entre 1 et 2
3 racine = spo.bisect(f, 1, 2)

```

# 7 RÉCURSIVITÉ ///

## 7.1 Principe des fonctions récursives

- Une fonction est dite récursive si son corps contient un ou plusieurs appels à elle-même. C'est une méthode de résolution de problèmes consistant à décomposer un problème complexe en sous-problèmes de même nature mais de taille réduite.
- Pour qu'une fonction récursive soit valide et se termine, deux conditions sont impératives :
  - L'existence d'un ou plusieurs cas de base (ou conditions d'arrêt) qui sont traités sans appel récursif.
  - Une progression stricte des appels récursifs vers ce cas de base (souvent via un paramètre entier décroissant ou la taille d'une liste qui diminue).
- L'exemple canonique est le calcul de la factorielle  $n! = n \times (n - 1)!$  avec  $0! = 1$ .

```

1 def factorielle(n):
2     if n == 0:           # Cas de base
3         return 1
4     else:               # Appel récursif
5         return n * factorielle(n - 1)

```

## 7.2 Analyse des fonctions récursives

- **Gestion des appels (Pile d'exécution)** : Lorsqu'une fonction s'appelle elle-même, l'interpréteur suspend l'exécution courante et empile le contexte (variables locales, paramètres, adresse de retour) dans une structure de données appelée *pile d'exécution* (call stack). Lorsque le cas de base est atteint, les résultats sont renvoyés successivement ("dépilés") lors de la remontée. Si la profondeur de récursion est trop importante, on risque une erreur de type `RecursionError` (débordement de pile ou *stack overflow*).
- **Récursivité terminale** : Une fonction est dite récursive terminale si l'appel récursif est la toute dernière instruction exécutée (aucune opération n'est effectuée sur le résultat renvoyé par l'appel). Cette forme est théoriquement optimisable par le compilateur pour ne pas consommer de pile (ce n'est cependant pas le cas en Python standard).
- **Complexité** : La complexité temporelle se calcule souvent en établissant une relation de récurrence sur le nombre d'opérations  $C(n)$ . Par exemple, pour la factorielle,  $C(n) = C(n - 1) + O(1)$ , ce qui conduit à une complexité linéaire  $O(n)$ . Pour une dichotomie, la division de la taille du problème par 2 à chaque étape mène à une complexité logarithmique  $O(\log n)$ .
- **Récursif vs Itératif** : Tout programme récursif peut être transformé en version itérative (avec des boucles).
  - *Avantages du récursif* : Code souvent plus élégant, lisible et proche de la définition mathématique (ex : suites, structures arborescentes).
  - *Inconvénients* : Coût mémoire (pile) et surcoût temporel lié aux appels de fonctions. L'itératif est généralement plus efficace en Python.
- **Exemple : Suite de Fibonacci** : La définition mathématique est  $F_0 = 0, F_1 = 1$  et  $F_n = F_{n-1} + F_{n-2}$ . L'implémentation récursive "naïve" est extrêmement inefficace (complexité exponentielle) car elle recalcule de nombreuses fois les mêmes termes.

```

1 def fibo_rec(n):
2     if n < 2: return n
3     return fibo_rec(n-1) + fibo_rec(n-2)

```

Une version itérative (ou utilisant la mémoïsation) est indispensable pour calculer des termes de rang élevé.

# 8 MATRICES DE PIXELS ET IMAGES

## 8.1 Formats de représentation d'image

- Il existe deux modes de codage numérique. Le mode vectoriel décrit les formes par des propriétés mathématiques (zoom infini sans perte, ex : PDF, SVG). Le mode matriciel (ou *bitmap*) repose sur une grille de pixels (ex : PNG, JPEG).
- Une image matricielle se caractérise par sa définition (nombre de pixels), sa résolution (nombre de pixels par unité de longueur, en dpi ou ppp) et aussi par sa quantification des couleurs exprimée en bits par pixel (ex : noir et blanc equivaut à 1bpp ; 256 nuances de gris equivaut à 8bpp ; 256 nuances dans les trois composantes equivaut à 24bpp...).
- Généralement, la colorimétrie utilise le modèle RVB (Rouge, Vert, Bleu). Chaque couleur est codée sur un octet (de 0 à 255). Un pixel est ainsi un triplet  $(R, V, B)$ . Le noir correspond à  $(0, 0, 0)$  et le blanc à  $(255, 255, 255)$ .

## 8.2 Manipulation des matrices de pixels

- En Python, une image est traitée comme un tableau Numpy de dimension 3 (hauteur, largeur, composantes). On utilise `matplotlib.image` (alias `img`) pour la lecture et `pyplot` pour l'affichage.

```

1 import matplotlib.image as img
2 import matplotlib.pyplot as plt
3
4 im = img.imread("image.png") # Chargement et stockage dans la variable
5 plt.imshow(im) # Preparation de l'affichage
6 plt.show() # Affichage
7
8 im.shape # Envoi des 3 dimensions de l'image
9 print(im) # Affichage du contenu de la variable
10
11 image = np.zeros((1920, 1080, 3)) # Creation d'une image vide
12 image[342, 135] = [100, 50, 12] # Accès et modification d'un pixel

```

## 8.3 Transformation d'images

- Symétrie axiale

```

1 H, L, C = im.shape
2 im_sym = np.zeros((H, L, C))
3
4 # Methode iterative
5 for i in range(H):
6     for j in range(L):
7         im_sym[i, j] = im[i, L - 1 - j]
8
9 # Methode par slicing
10 im_sym = im[:, ::-1]

```

- Rotation ( $90^\circ$ )

```

1 im_rot = np.zeros((L, H, C)) # Dimensions inversees
2
3 # Methode iterative

```

```

4 for i in range(H):
5     for j in range(L):
6         im_rot[L - 1 - j, i] = im[i, j]

```

- Passage en niveau de gris (par moyenne pondérée)

```

1 H, L, C = im.shape
2 im_gris = np.zeros((H, L)) # Tableau 2D (pas de 3eme dimension)
3
4 # Methode iterative
5 for i in range(H):
6     for j in range(L):
7         r, v, b = im[i, j] # Si C = 3
8         im_gris[i, j] = 0.2125*r + 0.7154*v + 0.0721*b # Norme 709 de la
9             CIE
10            # Sinon, (r+v+b)/3
11
12 # Affichage d'une image en niveau de gris
13 plt.imshow(im_gris, cmap='gray')

```

## 8.4 Modification par convolution : filtrage

- Le filtrage consiste à modifier la valeur d'un pixel en fonction de ses voisins grâce à une petite matrice.
- Le nouveau pixel est obtenu par un filtre (ou produit de convolution) permettant pour chaque pixel de modifier sa valeur en fonction des valeurs des pixels avoisinants, affectées de coefficients. Pour conserver la luminosité, on divise souvent le résultat par la somme des coefficients du filtre.
- Pour un pixel en bordure, certains voisins manquent. Une solution consiste à ignorer les bords de l'image (balayage de la ligne 1 à  $h - 1$ ).
- On peut appliquer un filtre sur une zone spécifique en utilisant une matrice masque (image binaire). Le résultat final est une combinaison :  $I_{finale} = M \cdot I_{floue} + (1 - M) \cdot I_{initiale}$ .
- Flou (Moyenneur)

```

1 n = 3
2 H, L, C = im.shape
3 b = n//2 # bordure
4
5 im_flou = np.zeros((H, L, C))
6
7 for i in range(b, H - b):
8     for j in range(b, L - b):
9         for c in range(C):
10             s=0
11             for k in range(n):
12                 for l in range(n):
13                     s = s + im[i+k-n//2, j+l-n//2, c]
14             im_flou[i, j, c]=s/(n*n)
15
16 return im_flou

```

## 8.5 Détection de contours

- La détection de contour repose sur l'identification des changements brutaux de couleur ou de contraste entre pixels voisins.

- On calcule pour chaque pixel une "distance" euclidienne par rapport à ses voisins directs de valeurs  $p_1$ ,  $p_2$ ,  $p_3$  et  $p_4$  (image en niveau de gris pour avoir une seule valeur). Une formule courante est la distance euclidienne des différences :  $d = \sqrt{(p_1 - p_3)^2 + (p_2 - p_4)^2}$ .
- On compare cette distance à une valeur seuil : si  $d > seuil$ , le pixel appartient à un contour et est tracé en blanc ; sinon, il est laissé en noir.

## 9 ANNEXE

### 9.1 Compléments

- Pour insérer des valeurs de variables au sein d'une chaîne de caractères, il suffit de placer la lettre `f` juste avant les guillemets et d'écrire les noms des variables (ou même des expressions) entre accolades `{}` à l'intérieur de la chaîne.

```
1 value = 10
2 print(f"Valeur mesurée : {value}") # Insertion simple
3 print(f"Résultat : {value + value/3}") # Opérations dans les accolades
```

- La fonction `input(message)` permet de mettre le programme en pause et de demander à l'utilisateur de saisir du texte au clavier. Le `message` est affiché pour guider l'utilisateur. La fonction `input` renvoie toujours une chaîne de caractères (`str`), même si l'utilisateur tape des chiffres. Il est donc impératif de convertir (caster) le résultat si l'on attend un nombre.

```
1 nom = input("Quel est votre nom ? ") # Renvoie un str
2 age_str = int(input("Quel est votre age ? ")) # Renvoie un int
```

- Le module `random` de la bibliothèque standard est utilisé pour générer des nombres aléatoires scalaires (uniques). Il s'importe via `import random`. Voici quelques fonctions notables :
  - `random.random()` : Renvoie un flottant  $x$  tel que  $0.0 \leq x < 1.0$  (avec beaucoup de digits).
  - `random.randint(a, b)` : Renvoie un entier  $n$  tel que  $a \leq n \leq b$  (bornes incluses!).
  - `random.choice(seq)` : Renvoie un élément choisi au hasard dans une séquence non vide (liste, chaîne...).
  - `random.shuffle(liste)` : Mélange les éléments d'une liste sur place (ne renvoie rien).

```
1 import random
2 de = random.randint(1, 6) # Simule un dé à 6 faces
3 piece = random.choice(["Pile", "Face"])
```

*Document en cours d'édition jusqu'en avril 2027.*