Lycée Louis-Le-Grand, Paris

Cours d'informatique commune

MPSI 4

Alain TROESCH

Version du:

4 juin 2015

Table des matières

1 Matériel et logiciels **5**

I Éléments d'architecture d'un ordinateur . . . 5
- I.1 Modèle de Von Neumann . . . 5
- I.2 Mémoires . . . 8
- I.3 Le processeur (CPU, Central Process Unit) . . . 10

II Codages . . . 13
- II.1 Bases de numération . . . 13
- II.2 Codage des entiers . . . 15
- II.3 Développement en base b d'un réel . . . 17
- II.4 Codage des réels (norme IEEE 754) . . . 19
- II.5 Problèmes liées à la représentation non exacte des réels . . . 21

III Circuits logiques . . . 23
- III.1 Portes logiques . . . 23
- III.2 Un exemple développé : l'addition sur n bits . . . 25
- III.3 Décodeurs d'adresse . . . 26
- III.4 Circuits bascules et mémoires . . . 27

IV Systèmes d'exploitation . . . 30
- IV.1 Qu'est-ce qu'un système d'exploitation ? . . . 30
- IV.2 Arborescence des fichiers . . . 32
- IV.3 Droits d'accès . . . 33

V Langages de programmation . . . 34
- V.1 Qu'est-ce qu'un langage de programmation ? . . . 34
- V.2 Niveau de langage . . . 34
- V.3 Interprétation et compilation . . . 35
- V.4 Paradigmes de programmation . . . 36

2 Les bases de la programmation en Python **37**

I Python dans le paysage informatique . . . 37

II Les variables . . . 38
- II.1 Affectation . . . 39
- II.2 Affichage . . . 39
- II.3 Type et identifiant . . . 39

III Objets et méthodes . . . 40
- III.1 Les nombres . . . 40
- III.2 Les booléens et les tests . . . 41

III.3 Les listes . . . 41
III.4 Les ensembles . . . 44
III.5 Les tuples . . . 44
III.6 Les chaînes de caractères . . . 45
III.7 Les itérateurs . . . 47
III.8 Conversions de types . . . 47
IV Structuration d'un programme . . . 48
IV.1 Notion de programme . . . 48
IV.2 Les fonctions . . . 50
IV.3 Les structures conditionnelles . . . 50
IV.4 Les structures itératives . . . 51
IV.5 La gestion des exceptions . . . 52
V Modules complémentaires . . . 55
V.1 Utilisation d'un module . . . 55
V.2 Le module `math` . . . 56
V.3 Le module `numpy` (calcul numérique) . . . 57
V.4 Le module `scipy` (calcul scientifique) . . . 58
V.5 Le module `matplotlib` (tracé de courbes) . . . 58
V.6 Autres modules (`random`, `time`, textttsqlite3,...) . . . 59
VI Lecture et écriture de fichiers . . . 60

3 Variables informatiques **61**
I Notion de variable informatique . . . 61
II Structures de données . . . 63
III Mutabilité ; cas des listes . . . 67

4 Algorithmique élémentaire **73**
I Algorithmes . . . 73
I.1 Définition . . . 73
I.2 Le langage . . . 74
I.3 Les structures élémentaires . . . 74
I.4 Procédures, fonctions et récursivité . . . 78
II Validité d'un algorithme . . . 79
II.1 Terminaison d'un algorithme . . . 80
II.2 Correction d'un algorithme . . . 83
III Exercices . . . 86

5 Complexité **89**
I Introduction . . . 89
II Complexité en temps (modèle à coûts fixes) . . . 90
II.1 Première approche . . . 90
II.2 Simplification du calcul de la complexité . . . 92
II.3 Amélioration de la complexité de l'exemple donné . . . 93
III Complexité dans le meilleur ou le pire des cas, en moyenne . . . 94
III.1 Complexité dans le meilleur et le pire des cas . . . 94
III.2 Complexité en moyenne . . . 96
III.3 Algorithmes randomisés . . . 96
IV Limitations du modèle à coûts fixes . . . 97
V Complexité en mémoire, ou en espace . . . 98
VI Étude de quelques algorithmes de recherche . . . 99
VI.1 Recherche du maximum d'une liste . . . 99
VI.2 Recherche d'un élément dans une liste . . . 99

VI.3 Recherche dans un liste triée . . . 100
VI.4 Autres algorithmes . . . 101

6 Calculs d'intégrales 103
I La méthode des rectangles . . . 104
II La méthode des trapèzes . . . 106
III La méthode de Simpson . . . 108
IV La méthode de Monte-Carlo . . . 109

7 Résolution numérique d'équations 111
I Dichotomie . . . 111
II Méthode de la fausse position (HP) . . . 113
III Méthode de la sécante (HP) . . . 115
IV Méthode de Newton . . . 117
V Le problème de la dérivation numérique (HP) . . . 118

8 Résolution numérique d'équations différentielles 121
I Méthode d'Euler . . . 122
II Notion d'ordre d'une méthode . . . 124
III Méthode de Runge-Kutta, HP . . . 126

9 Résolution numérique de systèmes linéaires 131
I Structures de données adaptées en Python . . . 131
II Rappels sur la méthode du pivot de Gauss . . . 135
III Décomposition LU . . . 137
IV Problèmes de conditionnement . . . 138

10 Bases de données relationnelles 141
I Environnement client / serveur . . . 141
I.1 Le serveur informatique . . . 141
I.2 Le client . . . 143
I.3 Architecture 3-tiers . . . 143
I.4 Une autre architecture . . . 144
II Bases de données . . . 144
II.1 Présentation intuitive . . . 144
II.2 Dépendances et redondances . . . 151
III Algèbre relationnelle . . . 153
III.1 Schéma relationnel et relation . . . 153
III.2 Clés . . . 154
III.3 Schéma de base de donnée . . . 157
IV Exercices . . . 158

11 SQL : Création d'une BDD et requêtes 159
I Création d'une base de données . . . 160
I.1 Création des tables . . . 160
I.2 Entrée des données . . . 161
II Interrogation d'une BDD (Requêtes) . . . 162
II.1 Requêtes simples . . . 162
II.2 Sous-requêtes . . . 167
II.3 Constructions ensemblistes . . . 169
II.4 Jointure . . . 171

1 Matériel et logiciels

Dans ce chapitre, nous décrivons de façon schématique le fonctionnement d'un ordinateur, ainsi que certaines limitations intrinsèques.

I Éléments d'architecture d'un ordinateur

Nous commençons par décrire le matériel informatique constituant un ordinateur, et permettant son fonctionnement. Notre but est de donner une idée rapide, sans entrer dans le détail logique du fonctionnement du processeur (portes logiques) et encore moins dans le détail électronique caché derrière ce fonctionnement logique (amplificateurs opérationnels, transistors etc.)

I.1 Modèle de Von Neumann

Pour commencer, interrogeons-nous sur la significtion-même du terme « informatique »

Définition 1.1.1 (Informatique)

Le mot *informatique* est une contraction des deux termes *information* et *automatique*. Ainsi, l'informatique est la science du traitement automatique de l'information.

Il s'agit donc d'appliquer à un ensemble de données initiales des règles de transformation ou de calcul déterminées (c'est le caractère automatique), ne nécessitant donc pas de reflexion ni de prise d'initiative.

Définition 1.1.2 (Ordinateur)

Un ordinateur est une concrétisation de cette notion.

Il s'agit donc d'un appareil concret permettant le traitement automatique des données. Il est donc nécessaire que l'ordinateur puisse communiquer avec l'utilisateur, pour permettre l'entrée des données initiales, la sortie du résultat du traitement, et l'entrée des règles d'automatisation, sous la forme d'un programme. Le modèle le plus couramment adopté pour décrire de façon très schématique le fonctionnement d'un ordinateur est celui décrit dans la figure 1.1, appelé *architecture de Von Neumann*. Dans ce schéma, les flèches représentent les flux possibles de données.

Note Historique 1.1.3 (von Neumann)

John von Neumann (János Neumann) est un scientifique américano-hongrois (Budapest, 1903 - Washington, D.C., 1957). Ses domaines de recherche sont très variés, de la mécanique quantique aux sciences économiques, en passant par l'analyse fonctionnelle, la logique mathématique et l'informatique. Il contribue au projet Manhattan,

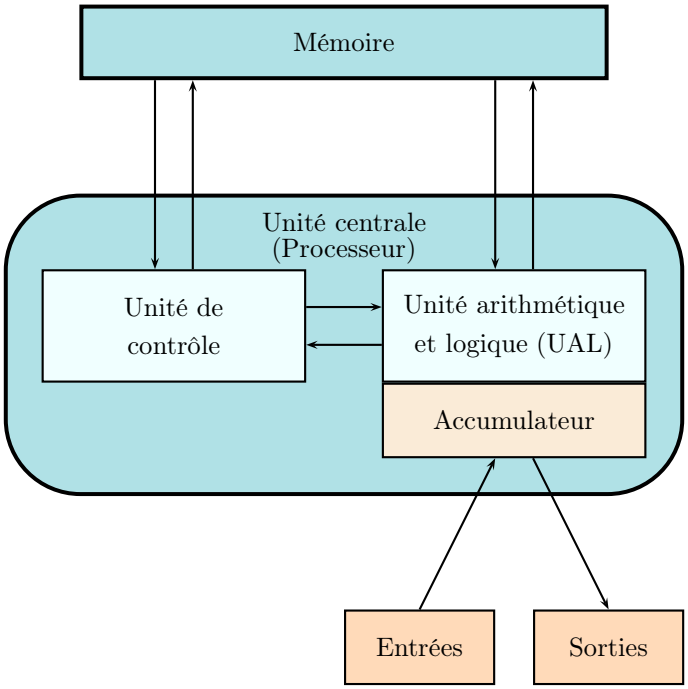


FIGURE 1.1 – Modèle d'architecture de Von Neumann

et notamment à l'élaboration de la bombe A, puis plus tard de la bombe H. Son nom reste attaché à la description de la structure d'un ordinateur, en 1945. C'est sur ce schéma qu'ont ensuite été élaborés les premiers ordinateurs. Si les ordinateurs actuels sont souvent beaucoup plus complexes, leur schéma grossier reste cependant très proche du schéma de l'architecture de von Neumann.

Note Historique 1.1.4 (architecture de von Neumann)

Le schéma d'un ordinateur (architecture de von Neumann) a été donné en 1945 par John von Neumann, et deux collaborateurs dont les noms sont injustement restés dans l'oubli : John W. Maukly et John Eckert. John von Neumann lui-même attribue en fait l'idée de cette architecture à Alan Turing, mathématicien et informaticien britannique dont le nom reste associé à la notion de calculabilité (liée à la machine de Turing), ainsi qu'au décryptage de la machine Enigma utilisée par les nazis durant la seconde guerre mondiale.

- **Entrées - sorties**
  Les entrées et sorties se font au moyen de périphériques spéciaux destinés à cet usage.
  * Les périphériques d'entrée permettent à un utilisateur d'entrer à l'ordinateur des données, sous des formats divers : clavier, souris, scanner, webcam, manettes de jeu etc.
  * Les périphériques de sortie permettent de restituer des informations à l'utilisateur. Ils sont indispensables pour pouvoir profiter du résultat du traitement de l'information : écran, imprimante, hauts-parleurs, etc.
  * Certains périphériques peuvent parfois jouer à la fois le rôle d'entrée et de sortie, comme les écrans tactiles. Ils peuvent aussi avoir des fonctions non liées aux ordinateurs, comme certaines photocopieuses, dont l'utilisation essentielle ne requiert pas d'ordinateur, mais qui peuvent aussi faire office d'imprimante et de scanner.
- La **mémoire** permet le stockage des données et des logiciels (programmes) utilisés pour les traiter. Ce stockage peut être :
  * définitif (mémoire morte, ou ROM, inscrite une fois pour toute, et non modifiable, à moins d'interventions très spécifiques),

  - temporaire à moyen et long terme (stockage de données et logiciels que l'utilisateur veut garder, au moins momentanément)
  - temporaire à court terme (données stockées à l'initiative du processeur en vue d'être utilisées ultérieurement : il peut par exemple s'agir de résultats intermédiaires, de piles d'instructions etc.)

  L'architecture de von Neumann utilise le même type de mémoire pour les données et les programmes, ce qui permet la modification des listes d'instructions (elle-mêmes pouvant être gérées comme des données). Ce procédé est à l'origine des boucles.

  Nous reparlerons un peu plus loin des différents types de mémoire qui existent.
- Le **processeur** est le cœur de l'ordinateur. C'est la partie de l'ordinateur qui traite l'information. Il va chercher les instructions dans un programme enregistré en mémoire, ainsi que les données nécessaires à l'exécution du programme, il traduit les instructions (parfois complexes) du programme en une succession d'opérations élémentaires, exécutées ensuite par les unités de calcul (UAL et unité de calcul flottant). Il interagit aussi éventuellement avec l'utilisateur, suivant les instructions du programme. Nous étudierons un peu plus loin le processeur de façon un peu plus précise, sans pour autant entrer dans les détails logiques associés aux traductions et aux exécutions.
- Le transfert des données (les flèches dans le schéma de la figure 1.1) se fait à l'aide de cables transportant des impulsions électriques, appelés **bus**.
  - Un bus est caractérisé :
    - par le nombre d'impulsions électriques (appelées **bit**) qu'il peut transmettre simultanément. Ce nombre dépend du nombre de conducteurs électriques parallèles dont est constitué le bus. Ainsi, un bus de 32 bits est constitués de 32 fils conducteurs pouvant transmettre indépendamment des impulsions électriques.
    - par la fréquence des signaux, c'est-à-dire le nombre de signaux qu'il peut transmettre de façon successive dans un temps donné. Ainsi, un bus de 25 MHz peut transmettre 25 millions d'impulsions sur chacun de ses fils chaque seconde.

    Ainsi, un bus de 32 bits et 25 MHz peut transmettre $25 \cdot 10^6 \cdot 32$ bits par seconde, soit $800 \cdot 10^6$ bit par seconde, soit environ 100 Mo (mégaoctet) par seconde (un octet étant constitué de 8 bit). Le « environ » se justifie par le fait que les préfixes *kilo* et *méga* ne correspondent pas tout-à-fait à 10^3 et 10^6 dans ce cadre, mais à $2^{10} = 1024$ et $2^{20} = 1024^2$.
  - Les bus peuvent donc transmettre les données à condition que celles-ci soient codées dans un système adapté à ces impulsions électriques. La base 2 convient bien ici (1 = une impulsion électrique, 0 = pas d'impulsion électrique). Ainsi, toutes les données sont codées en base 2, sous forme d'une succession de 0 et de 1 (les bits). Ces bits sont souvent groupés par paquets de 8 (un octet). Chaque demi-octet (4 bits) correspond à un nombre allant de 0 à 15, écrit en base 2. Ainsi, pour une meilleure concision et une meilleure lisibilité, les informaticiens amenés à manipuler directement ce langage binaire le traduisent souvent en base 16 (système hexadécimal, utilisant les 10 chiffres, et les lettres de a à f). Chaque octet est alors codé par 2 caractères en hexadécimal.
  - Les bus se répartissent en 2 types : les *bus parallèles* constitués de plusieurs fils conducteurs, et permettant de transmettre 1 ou plusieurs octets en une fois ; et les *bus séries*, constitués d'un seul conducteur : l'information est transmise bit par bit.

    Paradoxalement, il est parfois plus intéressant d'utiliser des bus séries. En effet, puisqu'un bus série utilise moins de conducteur qu'un bus parallèle, on peut choisir, pour un même prix, un conducteur de bien meilleure qualité. On obtient alors, au même coût, des bus séries pouvant atteindre des débits égaux, voire supérieurs, à des bus parallèles.
  - Un ordinateur utilise des bus à 3 usages essentiellement :
    - le bus d'adresse, dont la vocation est l'adressage en mémoire (trouver un endroit en mémoire). C'est un bus unidirectionnel.
    - les bus de données, permettant la transmission des données entre les différents composants. Ce sont des bus bidirectionnels.
    - les bus de contrôle, indiquant la direction de transmission de l'information dans un bus de données.
- La carte-mère est le composant assurant l'interconnexion de tous les autre composants et des pé-

riphériques (*via* des ports de connection). Au démarrage, elle lance le BIOS (Basic Input/Output ystem), en charge de repérer les différents périphériques, de les configurer, puis de lancer le démarrage du système *via* le chargeur d'amorçage (boot loader).

I.2 Mémoires

Nous revenons dans ce paragraphe sur un des composants sans lequel un ordinateur ne pourrait rien faire : la mémoire.
La mémoire est caractérisée :

- par sa taille (nombre d'octets disponibles pour du stockage). Suivant le type de mémoire, cela peut aller de quelques octets à plusieurs Gigaoctets ;
- par sa volatilité ou non, c'est-à-dire le fait d'être effacée ou non en absence d'alimentation électrique.
- par le fait d'être réinscriptible ou non (mémoire morte, mémoire vive).

Nous énumérons ci-dessous différents types de mémoire qu'on peut rencontrer actuellement. Du fait de l'importance de la mémoire et des besoins grandissants en capacité de mémoire, les types de mémoire sont en évolution constante, aussi bien par leur forme que par les techniques ou principes physiques utilisés. Nous ne pouvons en donner qu'une photographie instantanée, et sans doute déjà périmée et loin d'être exhaustive.

- **Mémoire morte (ROM, read-only memory)**
  Il s'agit de mémoire non volatile, donc non reprogammable. Ce qui y est inscrit y est une fois pour toutes, ou presque. Le BIOS, permettant le lancement de l'ordinateur, est sur la ROM de l'ordinateur. Il s'agit plus spécifiquement d'une EPROM (erasable programmable read-only memory). Comme son nom l'indique, une EPROM peut en fait être effacée et reprogrammée, mais cela nécessite une opération bien particulière (elle doit être flashée avec des ultras-violets).
- **Mémoire vive (RAM, random access memory)**
  * Le nom de la RAM provient du fait que contrairement aux autres types de stockages existant à l'époque où ce nom a été fixé (notamment les cassettes magnétiques), la lecture se fait par accès direct (random), et non dans un ordre déterminé. Le nom est maintenant un peu obsolète, la plupart des mémoires, quelles qu'elles soient, fonctionnant sur le principe de l'accès direct.
  * La mémoire vive est une mémoire volatile, utilisée par l'ordinateur pour le traitement des données, lorsqu'il y a nécessité de garder momentanément en mémoire un résultat dont il aura à se resservir plus tard. Elle est d'accès rapide, mais peu volumineuse. Elle se présente généralement sous forme de barrettes à enficher sur la carte-mère.
  * Physiquement, il s'agit de quadrillages de condensateurs, qui peuvent être dans 2 états (chargé = 1, déchargé = 0). Ces condensateurs se déchargent naturellement au fil du temps. Ainsi, pour garder un condensateur chargé, il faut le recharger (rafraîchir) à intervalles réguliers. Il s'agit du cycle de rafraichissement, ayant lieu à des périodes de quelques dizaines de nanosecondes. Par conséquent, en l'absence d'alimentation électrique, tous les condensateurs se déchargent, et la mémoire est effacée.
- **Mémoires de masse**
  Ce sont des mémoires de grande capacité, destinées à conserver de façon durable de grosses données (bases de données, gros programmes, informations diverses...) De par leur vocation, ce sont nécessairement des mémoires non volatiles (on ne veut pas perdre les données lorsqu'on éteint l'ordinateur !). Par le passé, il s'agissait de bandes perforées, puis de cassettes, de disquettes etc. Actuellement, il s'agit plutôt de disques durs, de bandes magnétiques (fréquent pour les sauvegardes régulières), de CD, DVD, ou de mémoires flash (clé USB par exemple).
- **Mémoires flash**
  Les mémoires flash (clé USB par exemple) que nous venons d'évoquer ont un statut un peu particulier. Techniquement parlant, il s'agit de mémoire morte (EEPROM : electrically erasable programmable read-only memory), mais qui peut être flashée beaucoup plus facilement que les EPROM, par un processus purement électrique. Ce flashage fait partie du fonctionnement même de ces mémoires, ce qui permet de les utiliser comme des mémoires réinscritptibles et modifiables à souhait.

Une caractéristique très importante de la mémoire est son temps d'accès, qui représente un facteur limitant du temps de traitement de données. Ce temps d'accès est bien entendu dépendant du type de mémoire, ainsi que de sa position par rapport au processeur : même si elle est très grande, la vitesse de circulation des impulsions électriques n'est pas nulle, et loin d'être négligeable dans la situation présente. mais le processeur lui-même et les composantes auquels il est rattaché subissent le même principe de miniaturisation en vue de l'augmentation de l'efficacité. Ainsi, il y a physiquement peu de place près du processeur.

Par ailleurs, le temps d'accès dépend beaucoup de la technologie utilisée pour cette mémoire. Les mémoires électroniques, composées de circuits bascules (ou circuits bistables), sont rapides d'accès, tandis que les mémoires à base de transistors et de condensateurs (technologie usuelle des barrettes de RAM) sont plus lentes d'accès (temps de charge + temps de latence dû à la nécessité d'un rafraichissement périodique, du fait de la décharge naturelle des condensateurs). Les mémoires externes nécessitant un processus de lecture sont encore plus lentes (disques durs, CD...)

Pour cette raison, les mémoires les plus rapides sont aussi souvent les moins volumineuses, et les plus onéreuses (qualité des matériaux + coût de la miniaturisation), le volume étant d'autant plus limité que d'un point de vue électronique, un circuit bistable est plus volumineux qu'un transistor.

On représente souvent la **hiérarchie des mémoires** sous forme d'un triangle (figure 1.2)

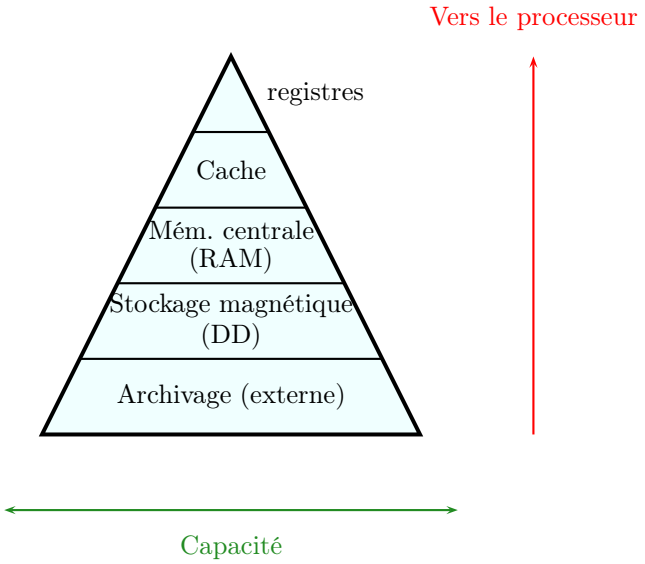


Figure 1.2 – Hiérarchie des mémoires

Les registres sont des mémoires situées dans le processeur, ne dépassant souvent pas une dizaine d'octets, et la plupart du temps à usages spécialisés (registres d'entiers, de flottants, d'adresses, compteur ordinal indiquant l'emplacement de la prochaine instruction, registres d'instruction...). Les données stockées dans ces registres sont celles que le processeur est en train d'utiliser, ou sur le point de le faire. Le temps d'accès est très court. Ces mémoires sont le plus souvent constitués de circuits bascule (voir plus loin)

La mémoire cache se décompose souvent en deux parties, l'une dans la RAM, l'autre sur le disque dur. Pour la première, il s'agit de la partie de la RAM la plus rapide d'accès (SRAM). La mémoire cache est utilisée pour stocker momentanément certaines données provenant d'ailleurs ou venant d'être traitées, en vue de les rapprocher, ou les garder près de l'endroit où elles seront ensuite utiles, afin d'améliorer par la suite le temps d'accès. Ainsi, il arrive qu'avant de parvenir au processeur, les données soient d'abord rapprochées sur la mémoire cache.

I.3 Le processeur (CPU, Central Process Unit)

C'est le cœur de l'ordinateur. C'est lui qui réalise les opérations. Dans ce paragraphe, nous nous contentons d'une description sommaire du principe de fonctionnement d'un processeur. Nous donnerons ultérieurement un bref aperçu de l'agencement électronique (à partir de briques électroniques élémentaires, traduisant sur les signaux électroniques les fonctions booléennes élémentaires) permettant de faire certaines opérations. Nous n'entrerons pas dans le détail électronique d'un processeur.

Composition d'un processeur

Le processeur est le calculateur de l'ordinateur. Il lit les instructions, les traduit en successions d'opérations élémentaires qui sont ensuite effectuées par l'unité arithmétique et logique (UAL), couplée (actuellement) à une unité de calcul en virgules flottantes (format usuellement utilisé pour les calculs sur les réels). Il est constitué d' :

- une **unité de traitement**, constituée d'une UAL (unité arithmétique et logique), d'un registre de données (mémoire destinée aux données récentes ou imminentes), et d'un accumulateur (l'espace de travail). Le schéma global est celui de la figure 1.3.

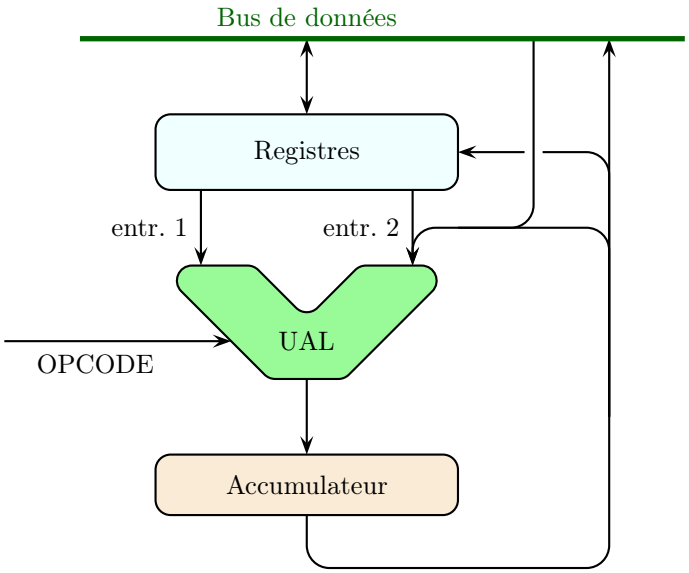


FIGURE 1.3 – Schéma d'une unité de traitement

Ce schéma correspond au cas d'une machine à plusieurs adresses (l'UAL peut traiter simultanément des données situées à plusieurs adresses). Il existe également des machines à une seule adresse : la deuxième donnée nécessaire aux calculs étant alors celle stockée dans l'accumulateur. Cela nécessite de décomposer un peu plus les opérations élémentaires (notamment dissocier le chargement de l'accumulateur du calcul lui-même).

Le code de l'opération (OPCODE) correspond à une succession de bits indiquant la nature de l'opération à effectuer sur l'entrée, ou les entrées (par exemple, un certain code correspond à l'addition, un autre à la multiplication, d'autres aux différents tests booléens etc.)

Une instruction élémentaire doit donc arriver à l'unité de traitement sous la forme d'un code (une succession de bits, représentés par des 0 et des 1, correspondant en réalité à des impulsions électriques), donnant d'une part le code de l'opération (OPCODE), d'autre part les informations nécessaires pour trouver les entrées auquelles appliquer cette opération (la nature de ces informations diffère suivant le code de l'opération : certains opérations demandent 2 adresses, d'autres une seule adresse et la donnée d'une valeur « immédiate », d'autres encore d'autres formats). Ce codage des instructions diffère également d'un processeur à l'autre. Par exemple, dans un processeur d'architeture MIPS, l'opération est codée sur 6 bits. L'opération 100000 demande ensuite la donnée de 3 adresses a_1, a_2 et a_3. Cette

opération consiste en le calcul de $a_2 + a_3$, le résultat étant ensuite stocké à l'adresse a_1 :

100000	a_1	a_2	a_3

$\$a_1 \leftarrow \$a_2 + \$a_3$

Dans cette notation, $\$a$ représente la valeur stockée à l'adresse a. L'opération 001000 est également une opération d'addition, mais ne prenant que deux adresses a_1 et a_2, et fournissant de surcroît une valeur i directement codée dans l'instruction (valeur immédiate). Elle réalise l'opération d'addition de la valeur i à la valeur stockée à l'adresse a_2, et va stocker le résultat à l'adresse a_1 :

001000	a_1	a_2	i

$\$a_1 \leftarrow \$a_2 + i$

L'UAL est actuellement couplée avec une unité de calcul en virgule flottante, permettant le calcul sur les réels (ou plutôt leur représentation approchée informatique, dont nous reparlerons plus loin). Un processeur peut avoir plusieurs processeurs travaillant en parallèle (donc plusieurs UAL), afin d'augmenter la rapidité de traitement de l'ordinateur ;

- une **unité de contrôle**, qui décode les instructions d'un programme, les transcrit en une succession d'instructions élémentaires compréhensibles par l'unité de traitement (suivant le code évoqué précédemment), et envoie de façon bien coordonnée ces instructions à l'unité de traitement. La synchronisation se fait grâce à l'horloge : les signaux d'horloge (impulsions électriques) sont envoyés de façon régulière, et permettent de cadencer les différentes actions. De façon schématique, la réception d'un signal d'horloge par un composant marque l'accomplissement d'une étape de la tâche qu'il a à faire. Les différentes étapes se succèdent donc au rythme des signaux d'horloge. Une unité de contrôle peut être schématisée à la manière de la figure 1.4.

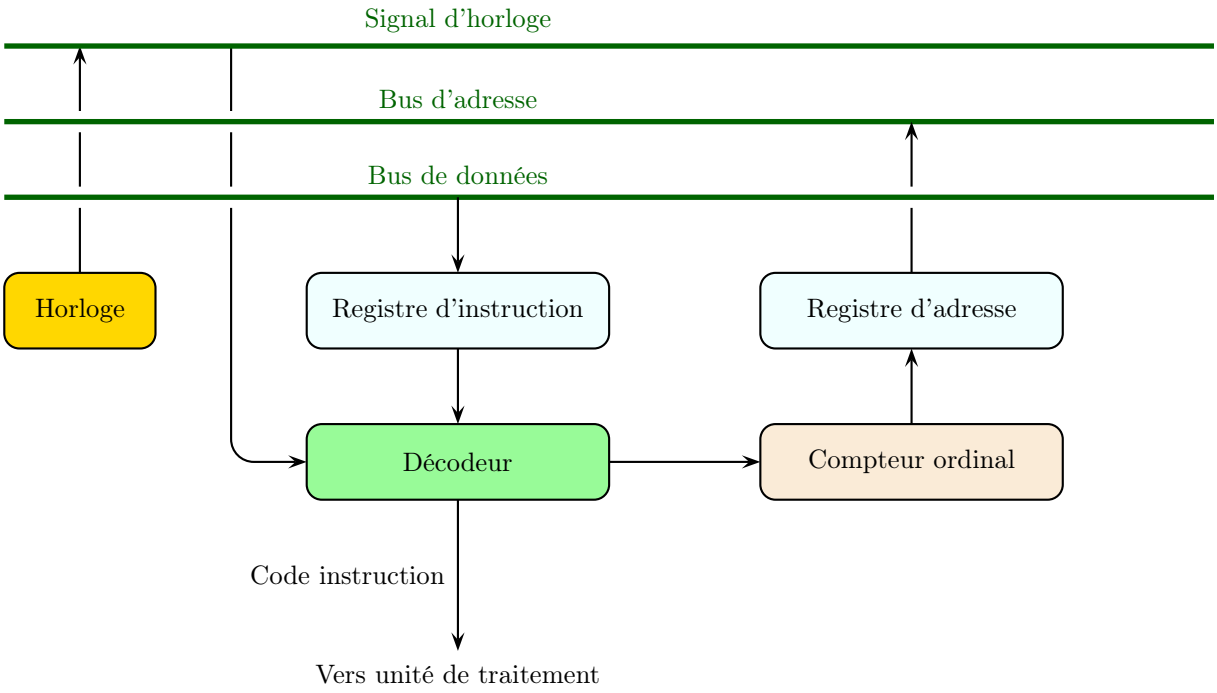


Figure 1.4 – Schéma d'une unité de contrôle

Le registre d'instruction contient l'instruction (non élémentaire) en cours, le compteur ordinal contient ce qu'il faut pour permettre de trouver l'adresse de l'instruction suivante, et le registre d'adresse contient l'adresse de l'instruction suivante.

Le décodeur décode l'instruction contenue dans le registre d'adresse et la traduit en une succession d'instructions élémentaires envoyées à l'unité de traitement au rythme de l'horloge. Lorsque l'instruction est entièrement traitée, l'unité va chercher l'instruction suivante, dont l'adresse est stockée dans le

registre d'adresse, et la stocke dans le registre d'instruction. Le décodeur actualise le compteur ordinal puis l'adresse de l'instruction suivante.

Décomposition de l'exécution d'une instruction

On peut schématiquement décomposer l'exécution d'une instruction par l'unité de traitement en 4 étapes :
- la réception de l'instruction codée (provenant de l'unité de contrôle) (FETCH)
- Le décodage de cette instruction ; les différentes données sont envoyées aux différents composants concerné (en particulier l'OPCODE est extrait et envoyé à l'UAL)
- L'exécution de l'instruction, correspondant au fonctionnement de l'UAL (EXECUTE)
- l'écriture du résultat dans une mémoire (WRITEBACK)

Si nous avons 4 instructions à exécuter successivement, cela nous donne donc 16 étapes successives :

Étape	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
Instr. 1	FE	DE	EX	WR												
Instr. 2					FE	DE	EX	WB								
Instr. 3									FE	DE	EX	WB				
Instr. 4													FE	DE	EX	WB

On se rend compte que l'UAL n'est pas utilisée à son plein régime : elle n'est solicitée qu'aux étapes 3, 7, 11 et 15 (EXECUTE). En supposant les instructions indépendantes les unes des autres (donc ne nécessitant pas d'attendre le résultat de la précédente pour pouvoir être exécutée), et en négligeant les dépendances éventuelles liées aux problèmes d'adresses, on peut imaginer le fonctionnent suivant, dans lequel on n'attend pas la fin de l'instruction précédente pour commencer le traitement de la suivante :

Étape	1	2	3	4	5	6	7
Instr. 1	FE	DE	EX	WR			
Instr. 2		FE	DE	EX	WB		
Instr. 3			FE	DE	EX	WB	
Instr. 4				FE	DE	EX	WB

On dit que le processeur est muni d'un **pipeline** s'il est muni d'un dispositif permettant d'effectuer ces opérations en parallèle de façon décalée. Évidemment, un pipeline doit aussi permettre de gérer les problèmes de dépendance, nécessitant parfois la mise en attente d'une instruction pour attendre le résultat d'une autre. Dans cet exemple, nous avons un pipeline à **4 étages** (c'est le nombre d'étapes dans la décomposition de l'exécution de l'instruction, correspondant aussi au nombre d'instructions maximal pouvant être traitées simultanément. Des décompositions plus fines peuvent permettre d'obtenir des pipelines constitués d'un nombre plus important d'étages (une dizaine).

Par ailleurs, certains processeurs sont munis de plusieurs cœurs (plusieurs unités de traitement), permettant l'exécution simultanée d'instructions. Ainsi, toujours en supposant les 4 instructions indépendantes, pour un processeur muni de 2 cœurs à pipeline à 4 étages, on obtient le tableau suivant :

Étape	1	2	3	4	5
Instr. 1	FE 1	DE 1	EX 1	WR 1	
Instr. 2	FE 2	DE 2	EX 2	WB 2	
Instr. 3		FE 1	DE 1	EX 1	WB 1
Instr. 4		FE 2	DE 2	EX 2	WB 2

On parle dans ce cas d'architecture en parallèle. Le parallélisme demande également un traitement spécifique des dépendances.

Puissance de calcul d'un processeur

La puissance d'un processeur est calculée en FLOPS (Floating Point Operation Per Second). Ainsi, il s'agit du nombre d'opérations qu'il est capable de faire en une seconde sur le format flottant (réels).

Pour donner des ordres de grandeur, voici l'évolution de la puissance sur quelques années références :

- 1964 : 1 mégaFLOPS (10^6)
- 1997 : 1 téraFLOPS (10^{12})
- 2008 : 1 pétaFLOPS (10^{15})
- 2013 : 30 pétaFLOPS

On estime que l'exaFLOPS (10^{18}) devrait être atteint vers 2020.

Évidemment, ces puissances correspondent aux super-calculateurs. La puissance des ordinateurs personnels suit une courbe bien inférieure. Par exemple, en 2013, un ordinateur personnel était muni en moyenne d'un processeur de 200 gigaFLOPS, ce qui est comparable à la puissance des super-calculateurs de 1995.

II Codages

Toute information étant codée en impulsions électriques (c'est-à-dire dans un système à 2 états : impulsion ou absence d'impulsion), le codage naturel des données doit se faire en binaire. Nous faisons dans ce paragraphe quelques rappels sur les bases de numération, puis nous indiquons plus précisément les normes de codage des entiers et des réels.

II.1 Bases de numération

Soit b un entier supérieur ou égal à 2.

Théorème 1.2.1 (existence et unicité de la représentation en base b)

Soit $N \in \mathbb{N}$. Il existe une unique suite $(x_n)_{n\in\mathbb{N}^*}$ d'entiers de $[\![0, b-1]\!]$, tous nuls à partir d'un certain rang, tels que :

$$N = \sum_{n=0}^{+\infty} x_n b^n.$$

Définition 1.2.2 (représentation d'un entier dans une base)

On dit que l'écriture de N sous la forme de la somme

$$N = \sum_{n=0}^{+\infty} x_n b^n,$$

où les x_n sont nuls à partir d'un certain rang, et vérifient $x_n \in [\![0, b-1]\!]$, est la représentation en base b de N. Si k est le plus grand entier tel que $x_k \neq 0$, on écrira cette représentation sous la forme synthétique suivante :

$$N = \overline{x_k x_{k-1} \dots x_1 x_0}^b.$$

Il convient de bien distinguer l'entier N de sa représentation sous forme de caractères. Il en est de même des x_i, qui dans la somme représente des valeurs (des entiers), alors que dans la notation introduite, ils représentent des caractères (les chiffres de la base de numération). De fait, lorsque $b \leqslant 10$, on utilise pour ces caractères les chiffres de 0 à $b-1$, tandis que si $b > 10$, on introduit au-delà du chiffre 9 des caractères spécifiques (souvent les premières lettres de l'alphabet). Ainsi, usuellement, les chiffres utilisés pour la représentation des entiers en base 16 (hexadécimal), très utilisée par les informaticiens, sont les 10 chiffres numériques de 0 à 9, et les 6 premières lettres de l'alphabet, de A à F.
Il faut bien comprendre la notation sous forme d'une juxtaposition de caractères (il ne s'agit pas du produit des x_i). Par exemple, $\overline{1443}^5$ représente l'entier

$$N = 3 + 4 \times 5 + 4 \times 5^2 + 1 \times 5^3 = 248.$$

Une même succession de caractères représente en général des entiers différents si la base b est différente. Ainsi $\overline{1443}^6$ représente l'entier

$$N = 3 + 4 \times 6 + 4 \times 6^2 + 1 \times 6^3 = 387.$$

En revanche, la notation $\overline{1443}^4$ n'a pas de sens, puisque seuls les caractères 0, 1, 2 et 3 sont utilisés en base 4.

Notation 1.2.3

Une représentation d'un entier N sous forme de juxtaposition de chiffres sans mention de la base (et sans surlignage) désignera suivant le contexte la numération en base 10 (numération usuelle en mathématiques) ou en base 2 (dans un contexte informatique). On s'arrangera pour que le contexte lève toute ambiguïté.

On remarquera que le seul cas d'ambiguïté dans ce contexte est le cas où seuls les chiffres 0 et 1 sont utilisés.

Méthode 1.2.4 (Convertir d'une base b à la base 10)

Soit N représenté sous la forme $\overline{x_k x_{k-1} \dots x_0}^b$. On trouve l'entier N en effectuant le calcul :

$$N = \sum_{i=0}^{k} x_i b^i.$$

Pour ce calcul, consistant en l'évaluation d'un polynôme, on utilisera de préférence l'algorithme de Hörner, basé sur la factorisation suivante :

$$N = x_0 + b(x_1 + b(x_2 + \cdots + b(x_{k-1} + bx_k))).$$

Ainsi, initialisant à x_k, on trouve N en répétant l'opération consistant à multiplier par b et ajouter le chiffre précédent.
Ce calcul peut être effectué en base 10, en convertissant les chiffres x_i en leur analogue en base 10, ainsi que b, puis en utilisant les algorithmes usuels de calcul des sommes et produits en base 10 (poser les opérations).

Remarque 1.2.5

Cette méthode permet en théorie de convertir tout entier d'une base b vers une base c. Les règles arithmétiques du calcul d'une somme et d'un produit se généralisent en effet en base quelconque.

Exemple 1.2.6

Convertir $\overline{342}^5$ en base 6 et en base 2, sans passer par la base 10.

Cependant, le manque d'habitude qu'on a des calculs en base différente de 10 font qu'en pratique, on préfère souvent faire une étape par la base 10, de sorte à n'avoir à utiliser des algorithmes de calculs que sur des numérations en base 10. Ainsi, on commence par convertir de la base b à la base 10, puis de la base 10 à la base c, en utilisant cette fois la méthode suivante, permettant de faire cette étape en n'utilisant que la numération en base 10.

Méthode 1.2.7 (Convertir de la base 10 à une base b)

Soit N (dont on connait la représentation en base 10). Pour trouver la représentation de N en base b, on effectue les divisions euclidiennes de N, puis des quotients successifs, par b, jusqu'à obtenir un

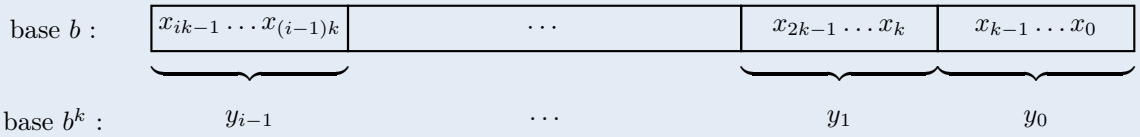
quotient nul. Les restes successifs sont les chiffres de N dans la numération en base b, dans l'ordre croissant des poids (le premier reste est le chiffre des unités).

Exemple 1.2.8

Convertir 2014 en base 7, en base 2.

Méthode 1.2.9 (cas particulier : convertir d'une base b à la base b^k)

Il suffit pour cela de regrouper k par k les chiffres de la représentation de N en base b, en commençant par les unités, et de traduire en base 10 les nombres dont les représentations en base b sont données par les groupements de k chiffres. Ces nombres sont dans $[\![0, b^k - 1]\!]$, et donnent les chiffres de la représentation de N en base b^k :



(dans cette représentation, on complète éventuellement la tranche la plus à gauche par des 0).
On peut également faire la démarche inverse sur le même principe.

Exemples 1.2.10

1. Donner la représentation hexadécimale de 2014, en utilisant la représentation binaire calculée ci-dessus.
2. Donner la représentation en base 2 de $\overline{af19b}^{16}$.

Ce dernier principe est très largement utilisé par les informaticiens, pour passer du binaire à l'hexadécimal et réciproquement.

II.2 Codage des entiers

Dans un système où le nombre de bits destinés à la représentation d'un entier est constant (langages traditionnels), seule une tranche de l'ensemble des entiers peut être codée. Plus précisément, avec n bits, on peut coder une tranche de longueur 2^n, par exemple $[\![0, 2^n - 1]\!]$, ou $[\![-2^{n-1}, 2^{n-1} - 1]\!]$, ou toute autre tranche de la même longueur suivant la convention adoptée. Nous nous bornerons à l'étude de cette situation où le nombre n de bits est une constante, même si en Python, le langage que nous utiliserons par la suite, le nombre n de bits alloués à la représentation d'un entier varie suivant les besoins (les entiers sont aussi grand qu'on veut en Python).

Pour pouvoir considérer des entiers de signe quelconque, de façon la plus équilibrée possible, on cherche à donner un codage des entiers de l'intervalle $[\![-2^{n-1}, 2^{n-1} - 1]\!]$. On pourrait imaginer différents codages, ayant chacun leurs avantages, mais aussi leurs inconvénients ; par exemple :

- coder la valeur absolue sur 7 bits, par la numération en binaire, et attribuer la valeur 0 ou 1 au 8^e bit suivant le signe ; il faudrait ensuite corriger bien artificiellement le doublon du codage de 0 et l'absence du codage de -2^{n-1}.
- Compter les nombres à partir de -2^{n-1}, à qui on attibue la valeur initiale 0 (autrement dit, considérer le code binaire de $x + 2^{n-1}$)

Ces deux codages sont simples à décrire et à calculer, mais s'avèrent ensuite assez peu commodes dans l'implémentation des calculs élémentaires.

Le codage adopté généralement est le codage appelé codage en complément à 2, correspondant grossièrement à une mise en cyclicité des entiers de -2^{n-1} à $2^{n-1}-1$, donc à un codage modulo 2^n. On commence à énumérer les entiers depuis 0 jusqu'à $2^{n-1}-1$, puis, les codages suivants en binaire vont correspondre aux entiers de -2^{n-1} à -1. Plus précisément :

Définition 1.2.11 (Codage en complément à 2)

Dans ce codage sur n bits des entiers de $[\![-2^{n-1}, 2^{n-1}-1]\!]$:
- le code binaire d'un entier $k \in [\![0, 2^{n-1}-1]\!]$ est sa représentation binaire
- le code binaire d'un entier $k \in [\![-2^{n-1}, -1]\!]$ est la représentation binaire de $k + 2^n = 2^n - |k|$.

Remarque 1.2.12

On distingue facilement par ce codage les entiers de signe positifs (leur premier bit est 0) des entiers de signe négatif (leur premier bit est 1). On dit que le premier bit est le bit de signe.

L'intérêt de ce codage est la facilité d'implémentation des algorithmes usuels de calcul (somme, produit...). En effet, ce codage évite d'avoir à distinguer dans ces algorithmes différents cas suivant le signe des opérandes : l'algorithme usuel pour la somme (et les autres également), correspondant à poser l'addition, est valable aussi bien pour les entiers négatifs, en posant la somme de leur représentation binaire en complément à 2, à condition :
- d'oublier une éventuelle retenue qui porterait sur un $n+1$-ième bit
- que le résultat théorique de l'opération soit bien compris dans l'intervalle $[\![-2^{n-1}, 2^{n-1}-1]\!]$.

En effet, les nombres théoriques et leur représentation binaire sont congrues modulo 2^n. De plus, l'oubli des retenues portant sur les bits non représentés (au delà du n^e) n'affecte par le résultat modulo 2^n du calcul. Ainsi, le résultat obtenu (sur le codage binaire, puis sur l'entier codé ainsi) est égal, modulo 2^n, au résultat théorique. Cette congruence est une égalité si on sait que le résultat théorique est dans $[\![-2^{n-1}, 2^{n-1}-1]\!]$, intervalle constitué de représentants de classes de congruences deux à deux distinctes.

Exemple 1.2.13

Pour un codage à 8 bits, illustrer ce qui précède pour le calcul de $-45+17$, de $-45+65$ et de $100+100$.

Certains langages gèrent les **dépassements de capacité** (le fait que le résultat sorte de la tranche initialement allouée) en attribuant des octets supplémentaires pour le codage du résultat (c'est le cas de Python), d'autres ne le font pas (comme Pascal par exemple). Dans ce cas, le résultat obtenu n'est correct que modulo 2^n. Ainsi, pour des entiers standard de type `integer` en Pascal, l'opération $32767+1$ renvoie le résultat -32768. Le programmeur doit être conscient de ce problème pour prendre les mesures qui s'imposent.

Pour terminer ce paragraphe, observons qu'il est possible d'obtenir très simplement le codage en complément à 2 d'un nombre $m < 0$, connaissant le codage de sa valeur absolue $|m|$. En effet, le codage de m est le développement binaire de $2^n - |m| = (2^n - 1) - |m| + 1$. Or, $2^n - 1$ est le nombre qui, se représente en binaire par n chiffres tous égaux à 1. Par conséquent, effectuer la soustraction $(2^n-1)-|m|$ est particulièrement simple : elle n'engendre aucune retenue, et consiste simplement à changer tous les 0 en 1 et réciproquement dans la représentation binaire de $|m|$. Par exemple :

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

Il suffit ensuite de rajouter 1 au résultat obtenu, ce qui se fait, par reports successifs des retenues, en transformant le dernier bloc $011\ldots1$ en $100\ldots0$. Il convient ici de noter qu'un tel bloc existe toujours

(éventuellement uniquement constitué du terme 0), car sinon, le nombre obtenu est $111\dots1$, ce qui correspond à $|m| = 0$, ce qui contredit l'hypothèse $m < 0$.
On résume cela dans la méthode suivante :

Méthode 1.2.14 (Codage en complément à 2 de $-|m|$)
Pour trouver le codage en complément à 2 de $-|m|$, on échange le rôle des chiffres 0 et 1 dans la représentation binaire de $|m|$, puis on ajoute 1, ce qui revient à changer le dernier bloc $01\dots1$ (éventuellement réduit à 0) en $10\dots0$.
Réciproquement si un codage correspond à un nombre négatif (*i.e.* le bit de signe est 1), on trouve le codage de la valeur absolue en effectuant la démarche inverse, c'est-à-dire en remplaçant le dernier bloc $10\dots0$ par $01\dots1$, puis en échangeant les 0 et les 1.

On peut remarquer que cela revient aussi à changer les 0 en 1 et réciproquement, sauf dans le dernier bloc $10\dots0$ (algorithme valable dans les deux sens)

II.3 Développement en base b d'un réel

La notion de représentation en base b d'un entier se généralise aux réels, à condition d'accepter des sommes infinies de puissances négatives de b. Ces sommes infinies sont des sommes de séries à termes positifs, à considérer comme la limite de sommes finies. Nous obtenons :

Théorème 1.2.15 (Développement en base b d'un réel)
Soit $x \in \mathbb{R}_+^*$. Il existe $n \in \mathbb{Z}$ et des entiers $x_i \in [\![0, b-1]\!]$, pour tout $i \in]\!]-\infty, n]\!]$, tels que $x_n \neq 0$ et

$$x = \sum_{i=-\infty}^{n} x_i b^i = \lim_{N \to -\infty} \sum_{i=N}^{n} x_i b^i.$$

De plus :
- *si pour tout $m \in \mathbb{N}$, $b^m x \notin \mathbb{N}$, alors ce développement est unique (c'est-à-dire que n et les x_i sont uniques)*
- *s'il existe $m \in \mathbb{N}$ tel que $b^m x \in \mathbb{N}$, alors il existe exactement deux développements, l'un « terminant » (vers $-\infty$) par une succession infinie de 0, l'autre terminant par une infinité de $b-1$.*

Notation 1.2.16 (Représentation du développement en base b)
Soit $x = \sum_{i=-\infty}^{n} x_i b^i$ un développement en base b de x. Si $n \geqslant 0$, on note :

$$x = \overline{x_n x_{n-1} \dots x_0, x_{-1} x_{-2} \dots}^b.$$

Si $n < 0$, on pose $x_0 = x_{-1} = \dots = x_{n+1} = 0$, et on écrit

$$x = \overline{x_0, x_{-1} x_{-2} \dots}^b = \overline{0, 0 \dots 0 x_n x_{n-1} \dots}^b.$$

De plus, s'il existe $m \leqslant 0$ tel que pour tout $k < m$, $x_k = 0$, on utilisera une représentation bornée, en omettant l'ultime série de 0 :

$$x = \overline{x_n x_{n-1} \dots x_0, x_{-1} x_{-2} \dots x_m}^b.$$

Enfin, si $b = 10$ (ou $b = 2$ dans un contexte informatique sans ambiguïté), on omettra la barre et la référence à b dans la notation.

Terminologie 1.2.17 (développement décimal, développement dyadique)

On parlera de développement décimal plutôt que de développement en base 10. De même, on parlera de développement dyadique plutôt que de développement en base 2. On rencontre aussi parfois la notion de développement triadique.

Exemples 1.2.18

1. En base 10, on a l'égalité entre 1 et $0,999999\ldots$
2. En base 3, on a l'égalité entre les réels $1,0210000000\ldots$ et $1,0202222222\ldots$

Terminologie 1.2.19 (développement propre)

On dira que le développement en base b est propre s'il ne termine pas par une série de chiffres $b-1$.

Ainsi, d'après ce qui précède, tout nombre réel admet une unique décomposition propre en base b.

Proposition 1.2.20

Avec la notation précédente, si les x_i, pour $i<0$, ne sont pas tous égaux à $b-1$ (c'est le cas en particulier si la décomposition est propre), $\overline{x_n x_{n-1}\ldots x_0}^b$ est la représentation en base b de la partie entière de x.

Ainsi, pour trouver le développement en base b de x, on peut utiliser la méthode suivante :

Méthode 1.2.21 (Développement en base b d'un réel)

Soit $x \in \mathbb{R}_+^*$ (le cas $x=0$ étant trivial, et le cas $x \in \mathbb{R}_-^*$ s'obtenant en rajoutant au développement de $|x|$ un signe $-$). Pour trouver un développement en base b de x, on peut décomposer le calcul en 2 étapes :

- trouver la représentation en base b de la partie entière $\lfloor x \rfloor$ de x, à l'aide des méthodes exposées précédemment. Cette représentation s'écrit sous la forme
$$\lfloor x \rfloor = \overline{x_n \ldots x_0}^b;$$
- trouver le développement décimal de la partie décimale $\{x\}$ de x, par la méthode exposée ci-dessous. Ce développement s'écrit sous la forme
$$\{x\} = \overline{0, x_{-1}x_{-2}\ldots}^b.$$

Le développement en base b de x est alors :

$$x = \overline{x_n \ldots x_0, x_{-1}x_{-2}\ldots}^b.$$

Pour trouver le développement propre en base b de $\{x\}$, on peut partir de la remarque évidente suivante : la multiplication par b consiste en le décalage à droite d'un cran de la virgule. Ainsi, x_{-1} est la partie entière de $b\{x\}$, puis reprenant la partie décimale de cette expression et la remultipliant par b, on obtient x_{-2} en prenant à nouveau la partie entière du résultat. On énonce :

Méthode 1.2.22 (Trouver le développement en base b d'un réel $x \in [0,1[$)

Soit $x \in [0,1[$. On a alors

$$x = \overline{0, x_{-1}x_{-2}\ldots}^b,$$

où :

$$x_{-1} = \lfloor b\{x\} \rfloor, \qquad x_{-2} = \lfloor b\{b\{x\}\} \rfloor, \qquad x_{-3} = \lfloor b\{b\{b\{x\}\}\} \rfloor \qquad \text{etc.}$$

Ainsi, on trouve les chiffres successifs du développement en répétant l'opération consistant à multiplier la partie décimale obtenue précédemment par b et en en prenant la partie entière.

Exemples 1.2.23

1. Trouver le développement dyadique de $0,7$
2. De même pour $0,1$.
3. De même pour $3,84375$.

On peut toujours écrire un nombre réel x (en base 10) sous la forme $y \times 10^k$, où $y \in [1, 10[$ et $k \in \mathbb{Z}$. Il s'agit de la notation scientifique usuelle. Ici, l'exposant k correspond au rang du premier chiffre non nul de x. De la même façon :

Proposition/Définition 1.2.24 (Notation scientifique en base b)

Soit $b \in \mathbb{N}$, $b \geqslant 2$. Soit $x \in \mathbb{R}_+^*$. Il existe un unique entier $k \in \mathbb{Z}$ et un unique réel $y \in [1, b[$ tel que $x = y \times b^k$.
Ainsi, il existe un unique $k \in \mathbb{Z}$, et d'uniques entiers y_i de $[\![0, b-1]\!]$, $i \in]\!-\infty, 0]\!]$, non tous égaux à $b-1$ à partir d'un certain rang, tels que

$$x = \overline{y_0, y_{-1}y_{-2}\dots}^{b} \times b^k \qquad \text{et} \qquad y_0 \neq 0.$$

Il s'agit de la notation scientifique en base b.

Remarque 1.2.25

Si $b = 2$, le chiffre y_0 est nécessairement égal à 1. Le stockage d'un nombre réel sous ce format pourra se dispenser du bit correspondant à y_0, tout déterminé.

II.4 Codage des réels (norme IEEE 754)

Le stockage des nombres réels ne peut pas se faire de façon exacte en général. En effet, même en ne se donnant pas de limite sur le nombre de bits, on sera limité par la capacité de la mémoire (finie, aussi grande soit-elle) de l'ordinateur. Or, entre deux réels distincts, aussi proches soient-ils, il existe une infinité non dénombrable de réels. La mémoire de l'ordinateur est incapable de décrire de façon exacte ce qui se trouve entre 2 réels distincts, quels qu'ils soient.
Ainsi, augmenter la capacité de stockage selon les besoins est vain, pour le stockage des réels. Si certains réels peuvent se stocker facilement sur un nombre fini de bits de façon cohérente, la plupart nécessiteraient une place infinie. Ainsi, par convention, on se fixe un nombre fini de bits, en étant conscient que ce qu'on codera sur ces bits ne sera qu'une valeur approchée du réel considéré. Il faudra alors faire attention, dans tous les calculs effectués, à la validité du résultat trouvé, en contrôlant, par des majorations, l'erreur dûe aux approximations de la représentation informatique des réels. On peut dans certains situations obtenir une divergence forte entre le calcul effectué par l'ordinateur et le calcul théorique. Nous en verrons un exemple frappant en TP lors de l'étude de l'algorithme d'Archimède pour le calcul de π.

Définition 1.2.26 (Représentation des réels sur 32 bits)

Le stockage des réels en norme IEEE 754 se fait sous la forme scientifique binaire :

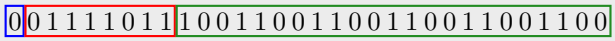
$$x = \pm \overline{1, \underbrace{x_{-1}x_{-2}\dots}_{\text{mantisse}}}^{2} \times 2^k.$$

Sur les 32 bits, 1 bit est réservé au stockage du signe, 8 au stockage de l'exposant et 23 au stockage de la mantisse. Le 1 précédant la virgule n'a pas besoin d'être stocké (valeur imposée) :

- Le bit de signe est 0 si $x \geqslant 0$ et 1 si $x < 0$.
- L'exposant k peut être compris entre -126 et 127. Les 8 bits destinés à cet usage donnent la représentation binaire de $127 + k$, compris entre 1 et 253. Les valeurs binaires 00000000 et 11111111 sont interdites (reservées à d'autres usages, correspondant à des représentations non normalisées)
- La mantisse est représentée de façon approchée par son développement dyadique à 23 chiffres après la virgule (donc les x_i, $i < -1$).

Exemples 1.2.27

- La représentation sur 32 bits de $0,1$ est :



- Donner la représentation sur 32 bits de $-5890,3$
- Quel est le nombre représenté par :

?

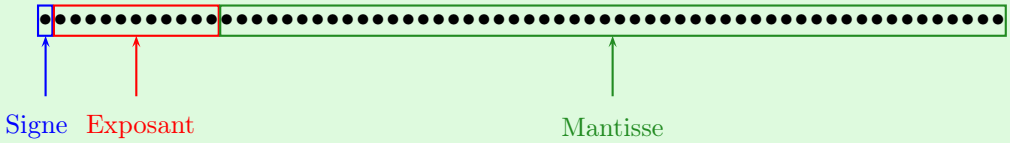
Remarque 1.2.28 (Valeurs extremales, précision pour un codage sur 32 bits)

- La valeur maximale est alors $\overline{1.111111111\ldots}^2 \times 2^{127} \simeq 2^{128} \simeq 3,403 \times 10^{38}$.
- La valeur minimale est $2^{-126} \simeq 1,75 \times 10^{-38}$.
- La précision de la mantisse est de l'ordre de $2^{-23} \simeq 1,192 \times 10^{-7}$.

Ainsi, cette représentation nous donne des valeurs réelles avec environ 8 chiffres significatifs en notation décimale (en comptant le premier). Cette précision est souvent insuffisante, surtout après répercution et amplification des erreurs. On utilise de plus en plus fréquemment la norme IEEE 754 avec un codage sur 64 bits, permettant une augmentation des extrêmes et de la précision. C'est le cas de Python que nous utiliserons.

Définition 1.2.29 (Représentation des réels sur 64 bits)

Le stockage des réels en norme IEEE 754 sur 64 bits se fait sur le même principe que sur 32 bits avec les modifications suivantes : sur les 64 bits, 1 bit est réservé au stockage du signe, 11 au stockage de l'exposant et 52 au stockage de la mantisse.



- Le bit de signe est 0 si $x \geqslant 0$ et 1 si $x < 0$.
- L'exposant k peut être compris entre -1022 et 1023. Les 11 bits destinés à cet usage donnent la représentation binaire de $1023 + k$, compris entre 1 et 2047. Les valeurs binaires 00000000000 et 11111111111 sont interdites (reservées à d'autres usages, correspondant à des représentations non normalisées)

- La mantisse est représentée de façon approchée par son développement dyadique à 52 chiffres après la virgule.

Remarque 1.2.30 (Valeurs extremales, précision pour un codage sur 64 bits)

- La valeur maximale est alors $\overline{1.111111111\ldots}^2 \times 2^{1023} \simeq 2^{1024} \simeq 1,797 \times 10^{308}$.
- La valeur minimale est $2^{-1022} \simeq 2,225 \times 10^{-308}$.
- La précision de la mantisse est de l'ordre de $2^{-52} \simeq 2,22 \times 10^{-16}$.

On obtient donc, avec le bit implicite, une précision correcte à 16 ou 17 chiffres significatifs. Cela explique que Python (par exemple) retourne les résultats réels en donnant 16 chiffres après la virgule.

Définition 1.2.31 (Nombres dénormalisés)

- La valeur 11111111111 de l'exposant permet le codage des nombres infinis (NaN, not a nomber). Nous ne parlons pas de cet aspect des choses.
- La valeur 00000000000 de l'exposant permet de coder des nombres un peu plus petits que la borne trouvée ci-dessous, mais en diminuant la précision (nombres normalisés). Dans ce cas, le bit implicite n'est plus considéré égal à 1, mais à 0. Un codage :

  1 00000000000 0001000000000000000000000000000000000000000000000000

  correspond par exemple au nombre $-2^{-1022} \times \overline{0,0001}^2 = -2^{1026}$.

Le plus petit nombre qu'on puisse alors obtenir est $2^{-1022-52} = 2^{-1074}$. Python renvoie `5.e-324` pour cette valeur (on a perdu toute la précision derrière la virgule). Le calcul de 2^{-1075} renvoie 0.

II.5 Problèmes liées à la représentation non exacte des réels

Il faut faire attention essentiellement à trois problèmes :

1. **Problèmes d'arrondi** La plupart des nombres n'étant pas représentés de façon exacte, les approximations faites peuvent s'ajouter dans les calculs. Le caractère approché des résultats des calculs peut se constater sur un exemple très simple :

   ```
   >>> 0.3-0.2-0.1
   -2.7755575615628914e-17
   ```

   Ces petites erreurs s'additionnent, et peuvent devenir grandes si on effectue un grand nombre de calculs :

   ```
   >>> S = 0
   >>> for i in range(1000):
   ...     S += 0.1
   ...
   >>> S
   99.9999999999986
   >>> S-100
   -1.4068746168049984e-12
   ```

2. **Problème d'absorption**

   Ce problème se produit lors de la somme de deux réels n'ayant pas du tout le même ordre de grandeur. Si le plus petit des deux a un ordre de grandeur inférieur à la précision du plus grand (donc de l'ordre de grandeur du dernier chiffre significatif), le résultat de la somme admet la même représentation que la valeur la plus grande. On dit que la petite valeur a été absorbée par la grande. Cette situation se produit dès lors que $\frac{x}{y} > 2^{52}$. Par exemple, comparons les 2 calculs suivants :

```
>>> 2**100 + 1 - 2**100
1
>>> 2**100 + 1. - 2**100
0.0
```

La première ligne de calcul se fait avec des entiers (aussi grands qu'on veut en Python). Il n'y a dans ce cas pas de problème. La seconde ligne représente le même calcul avec des réels (le point après le 1 force le calcul à se faire en type réel). Le 1 a été absorbé.

Nous illustrons le fait que le rapport de grandeurs limite est 2^{52} en remarquant que 1 est absorbé face à 2^{53}, mais pas 2 :

```
>>> 2**53 + 1. - 2**53
0.0
>>> 2**53 + 2. - 2**53
2.0
```

On peut alors imaginer des algorithmes finissant (rapidement) en théorie, mais ne s'arrêtant pas en pratique à cause de ce problème. On en donne d'abord une version avec des entiers (tapé en ligne de commande) :

```
>>> S=0
>>> i=2**53
>>> while i < 2**53 + 2:
...     i += 1
...     S += 1
...
>>> S
2
```

Le comportement est celui qu'on attend. Voici la version réelle du même algorithme :

```
>>> S=0
>>> i=2**53
>>> while i < 2**53 + 2:
...     i += 1.
...     S += 1
...
^CTraceback (most recent call last):
  File "<stdin>", line 1, in <module>
KeyboardInterrupt
>>> S
30830683
```

On est obligé d'interrompre le calcul avec `Ctrl-C`. En demandant ensuite la valeur de S, on se rend compte qu'il est passé un grand nombre de fois dans la boucle. Le fait de travailler en réels a comme conséquence que la valeur de i considérée est toujours la même, car le 1 qu'on lui ajoute est absorbé.

3. **Problème de cancellation**

   Il s'agit du problème inverse. Lorsqu'on effectue la différence de deux nombres de même ordre de grandeur, il se peut que de nombreux bits de la représentation se compensent. Par exemple, si seuls les 2 derniers bits diffèrent, le premier bit du résultat sera de l'ordre de grandeur de l'avant-dernier bit des deux arguments, le second bit significatif sera de l'ordre de grandeur du dernier bit des arguments, et les suivants n'ont pas de signification, car provenant de résidus de calculs approchés.

Leur valeur exacte nécessiterait de connaître les chiffres suivants dans le développement des deux arguments. Nous illustrons ceci sur l'exemple suivant :

```
>>> def f(x):
...     return (x+1)**2-x**2-2*x-1
...
>>> f(1000000000)
0
>>> f(1000000000.)
-1.0
>>> f(10000000000)
0
>>> f(10000000000.)
-2049.0
>>> f(100000000000)
0
>>> f(100000000000.)
-905217.0
>>>
```

Dans cet exemple, la fonction f est identiquement nulle. Le calcul est correct s'il est effectué avec des entiers, mais plus du tout lorsqu'on passe aux réels.

4. **Comparaisons par égalité**
   Aux trois problèmes précédents, nous ajoutons le quatrième, qui est en fait dérivé du premier (problème de l'inexactitude de la représentation des réels). Comparer par une égalité 2 réels, surtout s'ils sont issus de calculs, n'a pas beaucoup de pertinence, car même s'ils sont égaux en théorie, il est probable que cela ne le soit pas en pratique. Ainsi, en reprenant l'exemple précédent, on obtient le résultat suivant, assez surprenant, pour un test très simple :

```
>>> 0.3==0.2+0.1
False
```

On y remédie souvent en s'autorisant une petite inexactitude, ce qui oblige à remplacer l'égalité par un encadrement :

```
>>> abs(0.3-0.2-0.1)<1e-15
True
```

III Circuits logiques

Du fait de la circulation de l'information sous forme d'impulsions électiques, comme nous l'avons dit plus haut, la base de représentation privilégiée est la base 2. Mais comment en pratique effectuer les calculs élémentaires sur ces représentations en base 2 ? Autrement dit, comment, à partir d'impulsions électriques correspondant à deux arguments, récupérer les impulsions électriques correspondant, par exemple, à la somme ?

Le but de cette section est d'apporter quelques éléments de réponse, basés sur l'étude des portes logiques, sans entrer dans le détail électronique de ces portes, et en se contentant d'un exemple d'opération, le cas de l'addition.

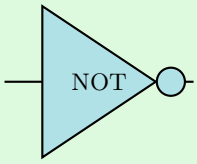
III.1 Portes logiques

Nous admettons l'existence de dispositifs électriques simples permettant de réaliser les opérations suivantes :

Définition 1.3.1 (Les portes logiques)

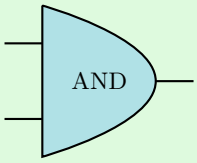
Dans les diagrammes ci-dessus, les entrées sont représentées à gauche, la sortie à droite ;

1. La porte NOT (NON) :



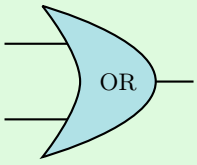
   La sortie vaut 1 si et seulement si l'entrée vaut 0.

2. La porte AND (ET) :



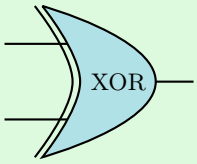
   La sortie vaut 1 si et seulement si les deux entrées valent 1 simultanément.

3. La porte OR (OU non exclusif) :



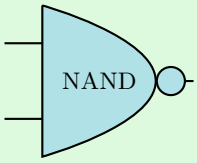
   La sortie vaut 1 si et seulement si l'une au moins des deux entrées vaut 1.

4. La porte XOR (OU exclusif) :



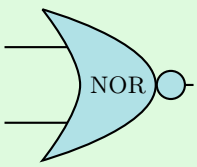
   La sortie vaut 1 si et seulement si une et une seule des deux entrées vaut 1.

5. La porte NAND (NON ET) :



   La sortie vaut 1 si et seulement si les deux entrées ne sont pas toutes les deux égales à 1, donc l'une au moins vaut 0.

6. La porte NOR (NON OU) :



   La sortie vaut 1 si et seulement si aucune des deux entrées ne vaut 1, donc si les deux valent 0.

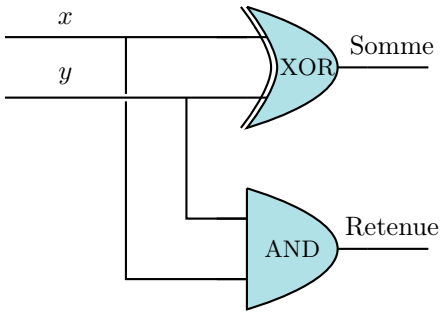
III.2 Un exemple développé : l'addition sur n bits

Le point de départ est la remarque suivante : l'addition de deux bits peut facilement se traduire à l'aide des portes logiques. En effet, en base 2, on a, en forçant le résultat à tenir sur 2 bits :

$$0+0=00 \qquad 0+1=1+0=01 \qquad 1+1=10.$$

Ainsi, le chiffre des unités vaut 1 si et seulement si un et un seul des deux bits additionnés est égal à 1 : il s'agit de l'opération booléenne XOR. De même, le chiffre des 2-aines est égal à 1 si et seulement si les deux bits sont tous deux égaux à 1 : il s'agit de l'opération booléenne AND. Ainsi, on obtient le bit en cours sous forme d'une opération booléenne, ainsi qu'un bit de retenue, sous forme d'une autre opération booléenne, ces deux opérations étant directement données par l'utilisation de portes logiques élémentaires.

Nous obtenons ainsi le schéma d'un *demi-additionneur*, additionnant deux bits x et y, et renvoyant un bit de somme et un bit de retenue :



Nous voyons dès lors comment faire pour sommer des nombres de 2 bits x_1x_0 et y_1y_0, en donnant le résultat sous forme d'un nombre à deux bits z_1z_0, et d'une retenue r_1 (figure 1.5)

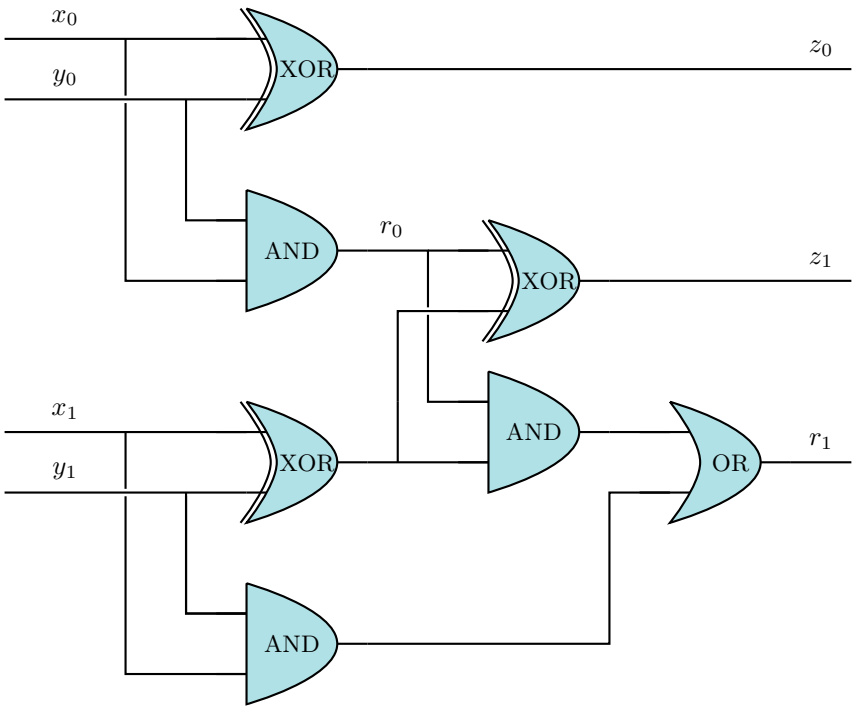


Figure 1.5 – Schéma logique d'un additionneur à 2 bits

On comprend alors comment continuer en branchant en série des demi-additionneurs sur le même principe, de sorte à obtenir un additionneur sur n bits. Remarquez qu'on obtiendra au bout un bit de retenue, qui donner la validité du résultat : si le résultat ne reste pas dans l'intervalle d'entier considéré (donné par

le nombre de bits), la retenue va valoir 1. Si elle vaut 0 en revanche, c'est qu'il n'y a pas de dépassement de capacité.

III.3 Décodeurs d'adresse

Définition 1.3.2 (Décodeur d'adresses)
Un décodeur d'adresses sur n bits est un assemblage électronique à n entrées et 2^n sorties, qui pour un signal d'entrée $(a_0, \cdots, a_{n-1}) \in \{0,1\}^n$ ressort une impulsion uniquement sur la ligne de sortie dont la numérotation correspond au nombre $\overline{\alpha_{n-1} \dots a_1 a_0}^2$. Typiquement , l'entrée correspond au codage d'une adresse, la sortie correspond à la selection de cette adresse parmi toutes les adresses possibles.

Nous nous contentons de décrire comment on peut construire des décodeurs d'adresses sur un petit nombre de bits à l'aide de portes logiques.

Décodeur d'adresses à 1 bit

Nous disposons d'une entrée a_0 et de deux sorties s_0 et s_1. La description qui précède amène le tableau de vérité suivant pour le décodeur d'adresse :

Entrée	Sortie	
a_0	s_0	s_1
0	1	0
1	0	1

La sortie s_0 correspond à non-a_0 et s_1 correspond à a_0. On obtient donc le circuit logique 1.6, réalisant le décodeur d'adresses sur 1 bit.

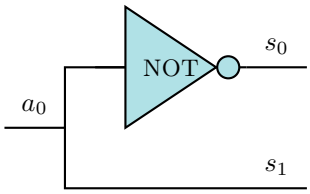


FIGURE 1.6 – Décodeur d'adresses sur 1 bit

Décodeur d'adresses sur 2 bits

La table de vérité du décodeur d'adresses sur 2 bits est :

Entrée		Sortie			
a_0	a_1	s_0	s_1	s_2	s_3
0	0	1	0	0	0
0	1	0	1	0	0
1	0	0	0	1	0
1	1	0	0	0	1

On constate que :

$$s_0 = \neg a_0 \wedge \neg a_1, \qquad s_1 = \neg a_0 \wedge a_1, \qquad s_2 = a_0 \wedge \neg a_1, \qquad s_3 = a_0 \wedge a_1.$$

On obtient alors le circuit du décodeur d'adresses sur 2 bits de la figure 1.7

Décodeur d'adresses sur n bits

De façon plus générale, on duplique chaque entrée, en appiquant une porte NOT à l'une des deux branches, puis on applique une porte AND (à n entrées) à toutes les sélections d'une des deux branches pour chacune des n entrées. Cela selectionne une unique sortie pour chacune des 2^n entrées possibles. En numérotant ces sorties de façon convenable, on obtient le décodeur d'adresse voulu.

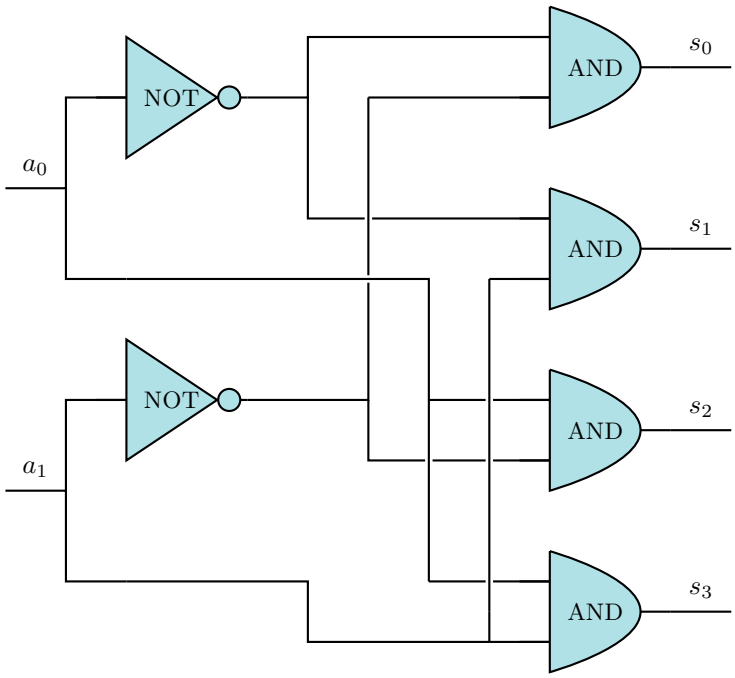


Figure 1.7 – Décodeur d'adresses sur 2 bits

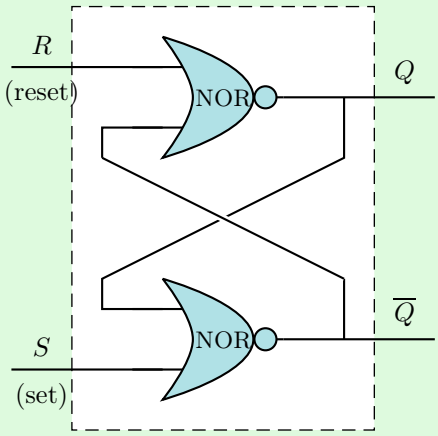
Ce circuit nécessite 2^n portes AND à n entrées. Chacune de ces portes est elle-même réalisable à l'aide de $n-1$ portes AND à 2 entrées (appliquées en cascade). On peut gagner en nombre de portes logiques en factorisant les opérations booléennes, ce qu'on illustre par la figure 1.8, pour un décodeur sur 3 bits. On peut noter qu'on peut obtenir des circuits de plus petite profondeur (nombre de composantes en série), donc plus performants, en utilisant un branchement du type « diviser pour régner » pour réaliser les portes AND à n entrées, mais cela augmente le nombre de composantes nécessaires. Comme souvent, il faut trouver le bon compromis entre coût et performance.

III.4 Circuits bascules et mémoires

Nous étudions dans ce paragraphe comment les portes logiques peuvent être utilisées en vue de mémoriser de l'information, grâce aux propriétés des circuits-bascule, aussi appelés circuits bistables.
Le schéma élémentaire à la base de ces mémoires est celui de la bascule RS minimale, dans lequel on utilise deux portes AND dont les sorties sont branchées sur une entrée de l'autre.

Définition 1.3.3 (Bascule RS minimale)

Le circuit bascule RS minimale, est le circuit dont le schéma électronique est le suivant :



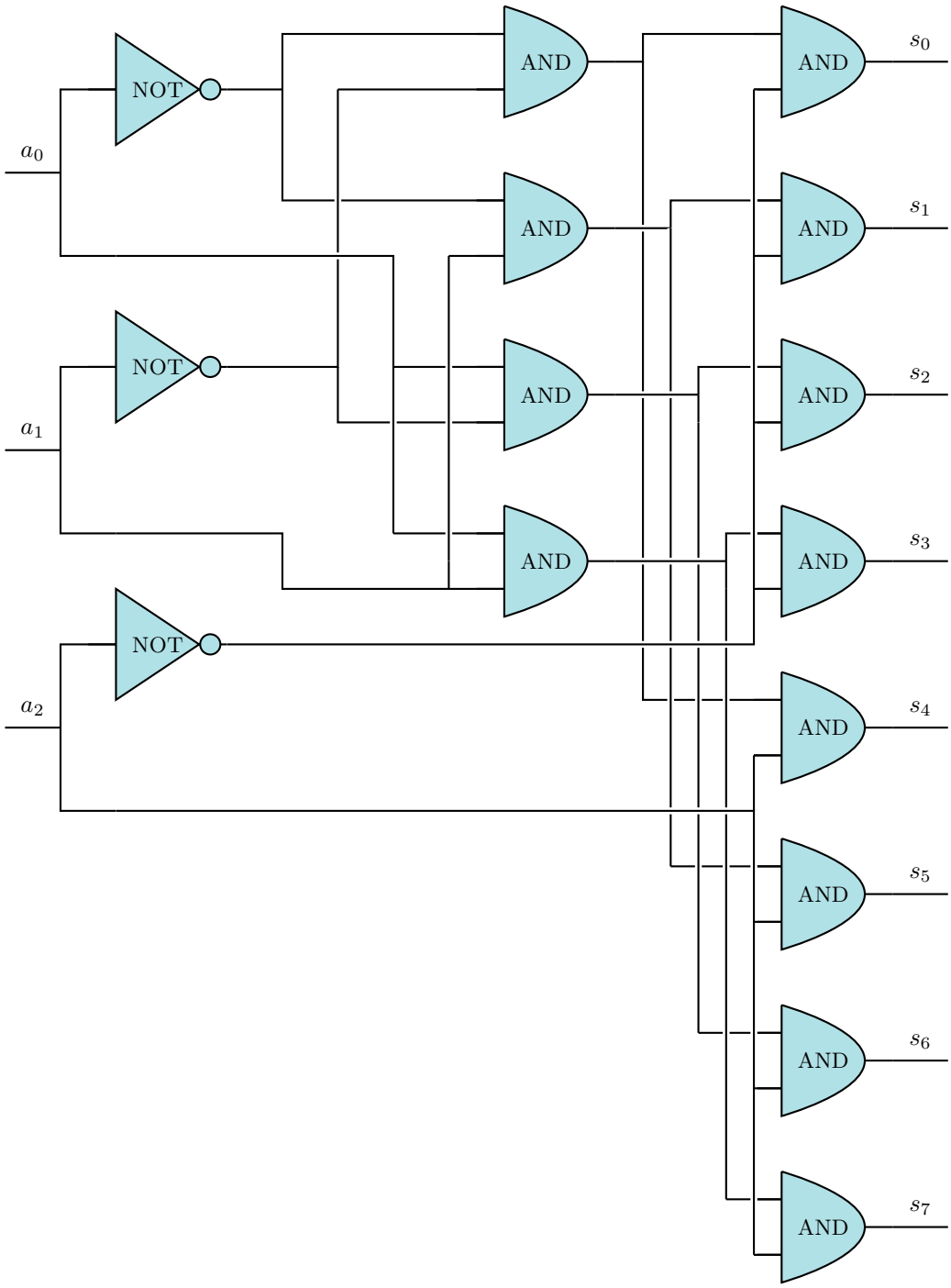
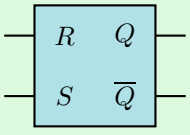


FIGURE 1.8 – Décodeur d'adresses sur 3 bits

Il sera noté par le schéma suivant :



Le principe de la bascule est le suivant :

- R et S ne peuvent pas prendre simultanément la valeur 1 (valeur interdite)

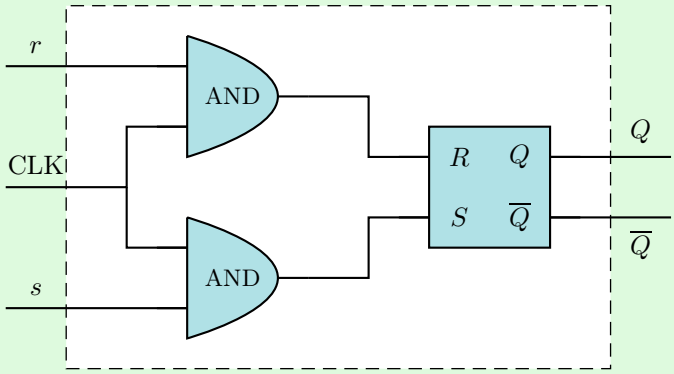
- Les deux sorties ne peuvent pas être simultanément égales à 0 (cela nécessiterait que les deux entrées soient égales à 1), ni toutes deux égales à 1 (si l'une est égale à 1, par définition d'une porte NOR, la seconde vaut 0). Ainsi, les deux sorties sont complémentaires (Q et $\overline{Q}$).
- Si $R = S = 0$, les deux positions $Q = 0$ et $Q = 1$ sont possibles (stables)
- Si $R = 1$, Q prend nécessairement la valeur 0 (bouton « reset » pour remettre à 0)
- Si $S = 1$, $\overline{Q}$ prend nécessairement la valeur 0, donc Q la valeur 1 (bouton « set », pour mettre la valeur 1)

Ainsi, une impulsion sur l'entrée R ou S permet de donner une valeur à Q, puis, en absence d'autre impulsion, la valeur de Q reste inchangée. On a donc mémorisé une valeur.

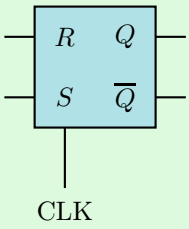
Pour que les mises à jour se fasse uniquement lors des signaux d'horloge, on peut brancher le signal d'horloge sur chacune des 2 entrées avec une porte AND. Ainsi, la bascule ne peut pas recevoir de signal en l'absence d'un signal d'horloge :

Définition 1.3.4 (Bascule synchrone)

La bascule synchrone est donnée par le schéma suivant :



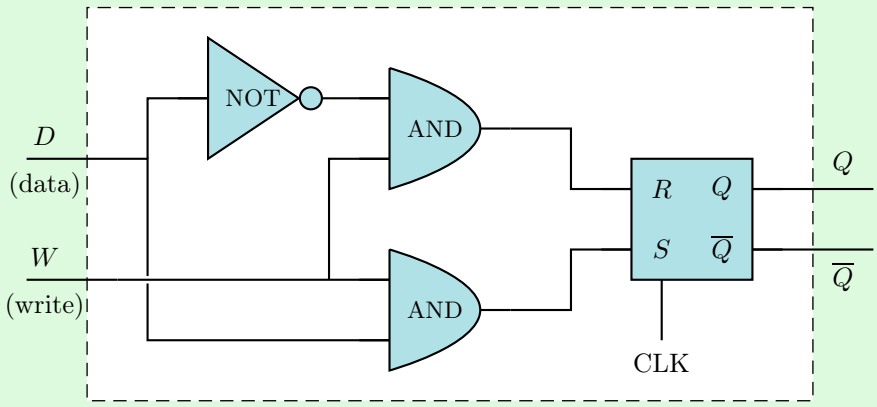
On notera une bascule synchrone par le schéma :



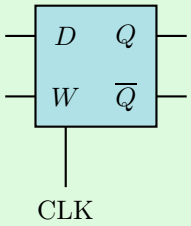
La mémorisation d'un bit D se fait alors à l'aide du schéma suivant :

Définition 1.3.5 (Bascule D)

Une bascule D est un dispositif électronique décrit par le schéma suivant :



Un tel dispositif sera noté :



Le fonctionnement de la bascule D est alors le suivant :

- En l'absence d'un signal W (write), les deux entrées de la bascule RS sont nulles, donc la sortie Q reste inchangée.
- En présence d'un signal W, les entrées de la bascule RS sont respectivement non-D et D. On vérifie sans peine qu'à tout signal d'horloge, la sortie Q sera égale à l'entrée D.

Ainsi :

Proposition 1.3.6 (Fonctionnement d'une bascule D)

Envoyer une donnée D accompagnée de l'instruction d'écriture W enregistre la donnée D dans Q. Cette donnée ne sera pas modifiée tant qu'il n'y aura pas d'autre signal d'écriture.

Nous donnons en figure 1.9 le schéma des circuits-mémoire (registres), basés sur l'utilisation des bascules D.

Lorsque le signal d'écriture W est donné, de façon synchrone, les bits $a_0, \ldots, a_{n-1}$ sont enregistrés par les n bascules D, jusqu'au prochain signal d'écriture. La ligne R (lecture) permet en cas de signal sur cette ligne (signal de lecture) de sortir les valeurs mémorisées par les bascules D lors du dernier enregistrement ressortent (sans signal de lecture, ces sorties sont nulles)

Ainsi, W est la ligne commandant l'écriture, R est la ligne commandant la lecture des données.

IV Systèmes d'exploitation

IV.1 Qu'est-ce qu'un système d'exploitation ?

Définition 1.4.1 (Système d'exploitation)

Un système d'exploitation est un ensemble de programmes permettant de gérer de façon conviviale et efficace les ressources de l'ordinateur. Il réalise un pont entre l'utilisateur et le processeur.

En particulier, le système d'exploitation :

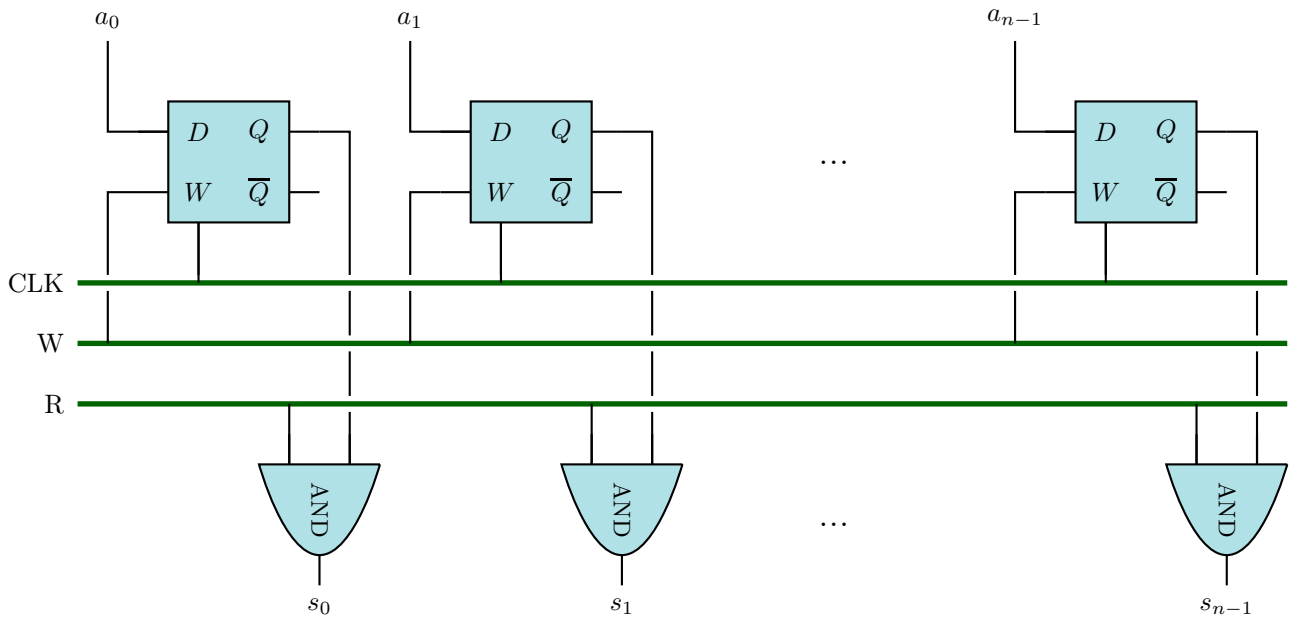


Figure 1.9 – Schéma d'un registre à n bits

- fournit une interface la plus conviviale possible, permettant le lancement d'applications diverses par l'utilisateur ;
- s'occupe de l'initialisation de l'ordinateur, ainsi que de la communication avec les périphériques ;
- propose une interface simple et intuitive pour le rangement des données (structure arborescente du système de fichier) : l'utilisateur n'a pas à se soucier lui-même de la localisation physique de ses fichiers sur l'espace mémoire, il les retrouve dans l'arborescence, en général logique et thématique, qu'il a lui-même élaboré ;
- gère les ressources de l'ordinateur.

En particulier, les systèmes d'exploitation actuels (parmi les plus connus, citons Linux, Windows, MacOS...) permettent l'exécution simultanée de plusieurs programmes, par le même utilisateur, ou même par différents utilisateurs. Chaque utilisateur doit dans ce cas avoir la sensation d'avoir l'ordinateur pour lui seul (machine virtuelle), ce qui nécessite une répartition des tâches assez élaborée.

Définition 1.4.2 (processus)
Un processus est une activité de l'ordinateur résultant de l'exécution d'un programme.

On peut imaginer plusieurs modes de fonctionnement :

- Les processus accèdent au processeur dans l'ordre de leur création, et utilisent ses ressources jusqu'à leur terminaison. Ainsi, les programmes sont exécutés les uns après les autres. Le temps de réponse dépendra beaucoup plus de la file d'attente que du temps d'exécution du programme lancé. Cela défavorise nettement les processus courts. On peut arranger un peu le système en donnant une priorité plus forte aux processus courts, mais malgré tout, ce système n'est pas satisfaisant.
- Méthode du tourniquet (ou balayage cyclique) : le processeur tourne entre les différents processus en cours. Chaque processus accède au processeur pour un temps donné, fixé à l'avance (le quantum), et au bout de ce temps, cède la place au processus suivant, même s'il n'est pas achevé. Il accèdera de nouveau au processeur lorsque celui-ci aura fait le tour de tous les processus actifs. Certains processus peuvent être mis en attente s'ils ont besoin de données d'autres processus pour continuer.
- Méthode du tourniquet multiniveau : c'est une amélioration du système précédent. On peut attribuer un niveau de priorité aux différents processus. Le processeur s'occupe d'abord des processus de priorité

la plus élevée, par la méthode du tourniquet, puis descend petit à petit les priorités. Un processus n'accède au processeur que s'il n'y a plus de processus en cours de niveau de priorité plus élevé.

C'est au système d'exploitation de gérer ce découpage des processus de sorte à ce que du point de vue de l'utilisateur, tout se passe comme s'il était seul, ou s'il ne lançait qu'une tâche (sur un ordinateur un peu plus lent). Évidemment, le quantum doit être assez petit pour que l'utilisation d'un logiciel semble continue à l'utilisateur, même si en réalité son exécution est entrecoupée un grand nombre de fois. On ne doit pas ressentir ce hâchage lors de l'utilisation.

Trois grands systèmes d'exploitation se partagent le marché grand public actuellement : Windows de Microsoft, Linux, dérivé de Unix, et MacOS de Apple, également construit sur un noyau Unix. Windows est réputé plus convivial que Linux dans sa présentation (interfaces sous forme de fenêtres, menus déroulants etc), l'utilisation de Linux se faisant davantage à l'aide de Shells (terminaux d'invites de lignes de commandes) : on y tape des instructions directement en ligne de commande plutôt qu'en utilisant la souris et des interfaces graphiques.

Les deux points de vue se défendent, mais dans des contextes différents. Pour un usage tout public et non professionnel, une interface graphique plaisante est souvent plus intuitive. Pour des utilisations plus poussées, la possibilité d'utiliser un terminal de commandes est plus efficace (rapidité de manipulation, possibilité de scripts élaborés etc.), mais nécessite un apprentissage et une habitude accrus. Il faut noter d'ailleurs qu'Unix (à l'origine de Linux) a été développé à des fins professionnelles et industrielles, puis les distributions de Linux se sont petit à petit adapté au grand public par l'ajout d'interfaces graphiques, mais restent tout de même tournées vers une activité professionnelle ; à l'inverse, Windows a été développé pour les ordinateurs personnels, donc pour un usage non professionnel : sa priorité est la convivialité, de sorte à rendre l'utilisation de l'ordinateur accessible à tous. Ce n'est que par la suite que Microsoft a adapté Windows à des utilisations professionnelles en entreprise. Ces deux grands systèmes d'exploitation ont donc eu une évolution opposée qui explique un peu leur philosophie de la présentation, même si les différences de convivialité et de souplesse d'utilisation tendent à s'amenuiser au fil des versions.

IV.2 Arborescence des fichiers

La gestion de la mémoire est un des autres grands rôles du système d'exploitation, aussi bien la mémoire vive que les mémoires de stockage. Lors de l'enregistrement d'un fichier, il recherche une place disponible sur l'espace mémoire de stockage (disque dur par exemple). Au fur et à mesure des enregistrements et effacements, ces espaces disponibles de stockage peuvent de moins en moins pratiques et de plus en plus fragmentés, ce qui nécessite parfois d'enregistrer un même fichier en plusieurs endroits non contigus (fragmentation). Plus le système est obligé de fragmenter les fichiers, plus l'utilisation de ces fichiers devient lente. C'est pourquoi il faut parfois réorganiser l'ensemble du disque dur, en déplaçant un grand nombre de données, pour regrouper ce qui va ensemble (défragmentation). Par exemple, Windows lance automatiquement la défragmentation du disque dur lorsque la situation devient critique.

Une telle gestion de la mémoire, où on range les données là où on trouve de la place n'est pas compatible avec une utilisation humaine directe et efficace. On ne peut pas demander à l'utilisateur de se souvenir que tel fichier qui l'intéresse est coupé en 3 morceaux rangés aux adresses $1552fab1$, $5ce42663$ et $4ceb5552$. L'utilisateur aurait tôt fait d'être complètement perdu.

Ainsi, le système d'exploitation propose à l'utilisateur une interface facile avec la mémoire : l'utilisateur ordonne son espace mémoire à sa convenance sous la forme d'une arborescence de dossiers (répertoires) dans lesquels il peut ranger des fichiers (figure 1.10). Cette arborescence doit être construite de façon logique, et correspond au rangement thématique qu'effectuerait l'utilisateur dans une armoire. Chaque noeud correspond à un dossier, chaque feuille correspond à un fichier (ou un dossier vide). Le contenu d'un dossier est l'ensemble de ses fils et de leur contenu. Un dossier peut contenir simultanément des fichiers et d'autres dossiers.

L'utilisateur peut se déplacer facilement dans son arborescence, et visualiser son contenu. Il n'a pas pour cela à se soucier des adresses physiques sur le disque, c'est le système d'exploitation qui se charge de

cela. D'ailleurs, la position d'un fichier dans l'arborescence n'a pas d'incidence sur l'adresse physique sur le disque : lorsqu'on déplace un fichier dans l'arborescence, on ne fait que modifier certains pointeurs d'adresse dans la description de l'arborescence, mais le fichier lui-même ne change pas de place physique sur le disque. Ainsi, déplacer un gros fichier est tout aussi rapide que déplacer un petit fichier, et quasi-instantané. Ce n'est évidemment pas le cas en cas de déplacement vers un autre support, ou en cas de copie.

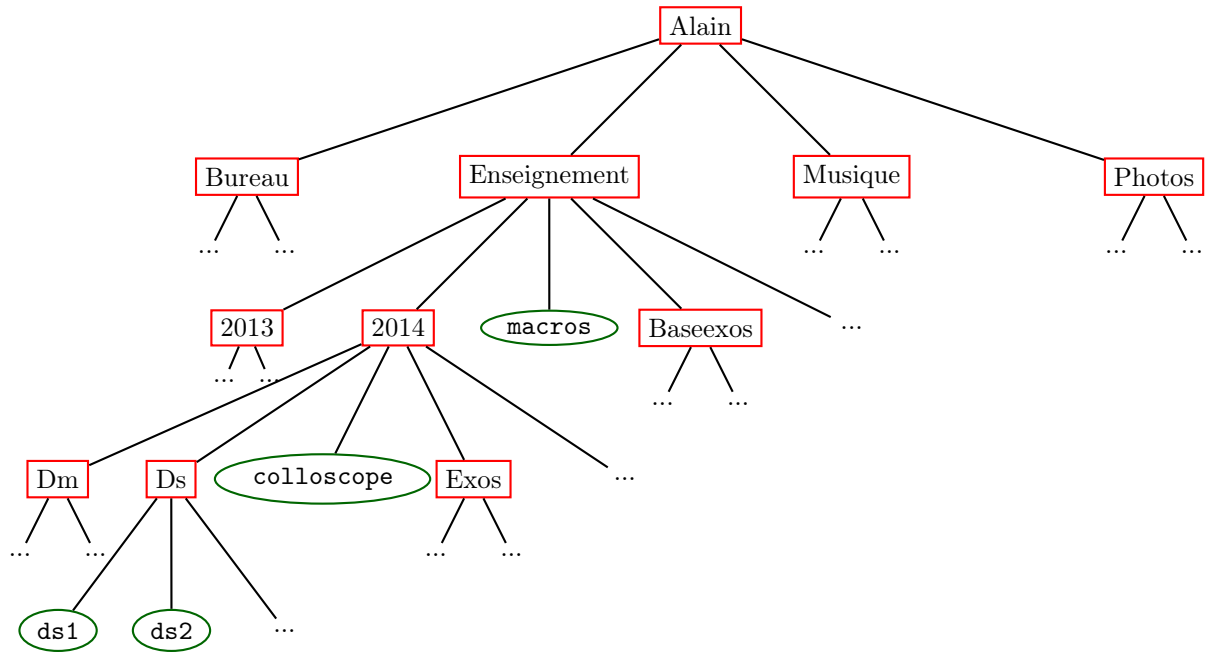


Figure 1.10 – Exemple d'arborescence de fichiers (simplifiée !)

IV.3 Droits d'accès

Avec l'évolution des réseaux locaux et d'internet, la protection des fichiers est importante : il faut pouvoir vous assurer que personne ayant accès au contenu de votre poste de travail (par exemple dans un réseau d'entreprise) ne pourra ouvrir ou exécuter vos fichiers sans votre autorisation. Pour cette raison, à chaque fichier et chaque dossier sont associés des droits d'accès. Ces droits se classent en 3 types :

- droit de lecture (`r`).
- droit d'écriture (`w`) : un fichier en droit d'écriture peut être modifié. Attention, il est possible de modifier des fichiers sans les ouvrir en lecture.
- droit d'exécution (`x`).

Ces droits se comprennent intuitivement pour les fichiers. Pour un répertoire, le droit de lecture permet de lister son contenu, le droit d'exécution permet d'accéder à ce répertoire, le droit d'écriture permet d'y ajouter ou d'en supprimer des fichiers.

La gestion des droits diffère selon les systèmes d'exploitation. Sous Linux par exemple, on peut visualiser les droits associés à un fichier par la commande `ls -l`, qui liste le contenu d'un répertoire et donne les propriétés essentielles de chaque fichier. À chaque fichier ou répertoire est associé un code du type `rwxrwxrwx`, certaines lettres pouvant être remplacées par un `-`, l'ensemble étant précédé d'une lettre `d` (pour *directory* si c'est un répertoire), d'une lettre `s` (si c'est un lien symbolique), ou d'un `-` (si c'est un fichier). Ce code est à comprendre de la sorte :

- Les lettres `rwx` représentent les droits en lecture, écriture et exécution respectivement.
- Si le droit est accordé, la lettre est apparente, sinon, elle est remplacée par `-`
- Le premier groupe `rwx` donne les droits pour l'utilisateur `u` (le propriétaire du fichier)

- Le second groupe `rwx` donne les droits pour le groupe `g` (défini par l'ingénieur système : une entreprise peut se diviser en plusieurs entités ; le groupe est l'entité dans laquelle se trouve le propriétaire du fichier)
- Le troisième groupe `rwx` donne les droits pour les autres `o`.
- `root` possède tous les droits.

Le propriétaire d'un fichier peut modifier les droits de ses fichiers. Pour des raisons historiques, ces modifications sont moins accessibles sous Windows que sous Linux, mais restent possibles. Sous Linux, il suffit d'utiliser la commande `chmod`, puis préciser en paramètre la lettre associée au groupe et les lettres associés aux droits qu'on veut ajouter (avec `+`) ou supprimer (avec `-`). Par exemple :

```
chmod go-r exemple.txt
```

supprime les droits de lecture pour les membres du groupe et pour les autres utilisateurs.

La fonction `chmod` peut aussi s'utilise en trascrivant les droits en binaire : les droits de chaque groupe fournissent un nombre binaire de 3 chiffres (0 pour un droit non accordé, 1 pour un droit accordé). Il s'agit donc d'un nombre entre 0 et 7. Ainsi, la donnée d'un nombre à 3 chiffres compris entre 0 et 7 (donc en base 8) suffit à décrire l'ensemble des droits. On peut utiliser ce nombre directement en paramètre de `chmod` pour redéfinir entièrement les droits. Par exemple

```
chmod 640 exemple.txt
```

attribuera les droits `rw-r-----`.

V Langages de programmation

V.1 Qu'est-ce qu'un langage de programmation ?

Le langage machine, c'est-à-dire le code binaire compris par le processeur, est assez peu accessible au commun des mortels. S'il fallait communiquer avec le processeur directement ainsi, seule une élite très restreinte pourrait faire de la programmation, d'autant plus que chaque processeur a son propre langage machine.

Un langage de programmation est un langage qui se place généralement à un niveau plus compréhensible par l'humain lambda, permettant, dans une syntaxe stricte, de décrire un algorithme par la donnée d'une succession d'instructions (des ordres). Cette succession d'instructions sera ensuite traduite en langage machine par un programme spécifique (le compilateur ou l'interpréteur)

Ainsi, un langage de programmation est à voir comme un langage intermédiaire permettant de communiquer indirectement avec le coeur opératoire de l'ordinateur.

Nous utiliserons cette année le langage de programmation `Python`, dans sa troisième version.

V.2 Niveau de langage

Comme vous le savez, il existe un grand nombre de langages de programmation. Pourquoi une telle variété ? Pourquoi certaines personnes ne jurent-elles que par un langage en particulier ? Y a-t-il réellement des différences fondamentales entre les différents langages ?

La réponse est OUI, de plusieurs points de vue. Nous abordons 3 de ces points de vue dans les 3 paragraphes qui viennent.

Le premier point de vue est celui du **niveau (d'abstraction) de langage**. Il s'agit essentiellement de savoir à quel point le langage de programmation s'éloigne du langage machine et des contraintes organisationnelles de la mémoire et des périphériques qui lui sont liées pour s'approcher du langage humain et s'affranchir de la gestion de l'accessoire pour se concentrer sur l'aspect algorithmique :

- Un **langage de bas niveau** est un langage restant proche des contraintes de la machine ; le but est davantage la recherche de l'efficacité par une gestion pensée et raisonnée des ressources (mémoire, périphérique...), au détriment du confort de programmation. Parmi les langages de bas niveau, on peut citer Assembleur ou C.
- Un **langage de haut niveau** est un langage s'approchant davantage du langage humain, et dégageant la programmation de toutes les contraintes matérielles qui sont gérées automatiquement (mémoire, périphériques...). L'intérêt principal est un confort de programmation et la possibilité de se concentrer sur l'aspect algorithmique. En revanche, on y perd nécessairement en efficacité : la gestion automatique des ressources ne permet pas de gérer de façon efficace toutes les situations particulières.

Ainsi, dans la vie pratique, les langages de haut niveau sont largement suffisants en général. C'est le cas en particulier pour la plupart des applications mathématiques, physiques ou techniques, pour lesquels le temps de réponse est suffisamment court, et qui sont amenées à être utilisées à l'unité et non de façon répétée.
Évidemment, dès que l'enjeu de la rapidité intervient (calculs longs, par exemple dans les problèmes de cryptographie, ou encore besoin de réactivité d'un système, donc pour tout ce qui concerne la programmation liée aux systèmes d'exploitations, embarqués ou non), il est préférable d'adopter un langage de plus bas niveau.

Il est fréquent dans des contextes industriels liés à l'efficacité de systèmes embarqués, que le prototypage se fasse sur un langage de haut niveau (travail des algorithmiciens), puis soit traduit en un langage de bas niveau (travail des programmeurs), souvent en C.

Python est un langage de très haut niveau d'abstration. Sa philosophie est de dégager l'utilisateur de toute containte matérielle, et même de se placer à un niveau où la plupart des algorithmes classiques (tri, recherche de motifs, algorithmes de calcul numérique...) sont déjà implémentés. Ainsi, Python propose un certain nombre de modules complémentaires facultatifs, proposant chacun un certain nombre d'outils algorithmiques dans un domaine précis. Python se place donc délibérément à un niveau où une grande partie de la technique est cachée.

Cela fournit un grand confort et une grande facilité de programmation, mais une maîtrise moins parfaite des coûts des algorithmes utilisant des fonctions prédéfinies.

V.3 Interprétation et compilation

Une autre grande différence entre les langages de programmation est la façon dont ils sont traduits en langage machine. C'est lors de cette traduction qu'est rajoutée toute la gestion de la basse-besogne dont on s'était dispensé pour un programme de haut-niveau. Ainsi, cette traduction est d'autant plus complexe que le langage est de haut-niveau.

Il existe essentiellement deux types de traduction :

- **La complilation.** Elle consiste à traduire entièrement un langage de haut niveau en un langage de bas niveau (souvent en Assembleur), ou directement en langage machine (mais cela crée des problèmes de portabilité d'une machine à une autre). Le programme chargé de faire cette traduction s'appelle le *compilateur*, et est fourni avec le langage de programmation. Le compilateur prend en argument le code initial, et retourne en sortie un nouveau fichier (le programme compilé), directement exploitable (ou presque) par l'ordinateur.
  - Avantages : une fois la compilation effectuée, le traitement par l'ordinateur est plus rapide (il repart de la source compilée). C'est donc intéressant pour un programme finalisé, voué à être utilisé souvent.
  - Inconvénients : Lors du développement, la compilation peut être lente, et nécessite que le programme soit syntaxiquement complet : elle nécessite donc de programmer étape entière par étape entière, sans pouvoir faire de vérifications intermédiaires.
- **L'interprétation.** Contrairement à la compilation, il ne s'agit pas de la conversion en un autre programme, mais d'une exécution dynamique. L'exécution est effectuée directement à partir du fichier

source : l'interpréteur lit les instructions les unes après les autres, et envoie au fur et à mesure sa traduction au processeur.
  * Avantages : il est inutile d'avoir un programme complet pour commencer à l'exécuter et voir son comportement. Par ailleurs, les environnements de programmation associés à ces langages permettent souvent la localisation dynamique des erreurs de syntaxes (le script est interprété et validé au moment-même où il est écrit). Enfin, l'exécution dynamique est compatible avec une exécution en console, instruction par instruction. C'est un atout majeur, notamment pour vérifier la syntaxe d'utilisation et le comportement d'une instruction précise.
  * Inconvénients : Le programme finalisé est plus lent à l'exécution, puisque la traduction part toujours du code initial, lointain du programme machine.

Python est un langage interprété, semi-compilé. On peut ainsi bénéficier d'une utilisation en console, ainsi que, suivant les environnements, d'une détection dynamique des erreurs de syntaxe. En revanche, l'exécution du programme commence par une compilation partielle, nécessitant une syntaxe complète.

V.4 Paradigmes de programmation

Un paradigme de programmation est un style de programmation déterminant de quelle manière et sous quelle forme le programmeur manipule des données et donne des instructions. Il s'agit en quelque sorte de la philosophie du langage. Il existe un grand nombre de paradigmes de programmation. Parmi les plus utilisés, citons les suivants :

- **La programmation impérative.** Le programme consiste en une succession d'instructions. L'exécution d'une instruction a pour conséquence une modification de l'état de l'ordinateur. L'état final de l'ordinateur fournit le résultat voulu.
- **La programmation orientée objet.** On agit sur des objets (des structures de données). Les objets réagissent à des messages extérieurs au moyen de méthodes. Ainsi, le gros de la programmation porte sur la définition de méthodes associées à des classes d'objets.
- **La programmation fonctionnelle.** On ne s'autorise pas les changements d'état du système. Ainsi, on ne fait aucune affectation. On n'agit pas sur les variables mais on exprime le programme comme un assemblage de fonctions (au sens mathématique).
- **La programmation logique.** C'est un paradigme basé sur les règles de la logique formelle. Il est notamment utilisé pour les démonstrations automatiques, ou pour l'intelligence artificielle.

Python est un langage hybride, adapté à plusieurs paradigmes. Essentiellement cette année, nous utiliserons Python en tant que langage impératif et orienté objet.

2 Les bases de la programmation en Python

Ce chapitre n'est pas un chapitre d'apprentissage du Python (pour cela, se reporter au tutoriel en ligne sur le site `https://docs.python.org/3/tutorial/index.html`, référence absolue en la matière). Il n'est pas destiné à être dispensé en cours, mais est plutôt à voir comme un aide-mémoire auquel se reporter tout au long de l'année pour les fonctionnalités les plus fréquemment utilisées de Python. Bien sûr, on ne peut pas être exhaustif, et l'aide en ligne, ainsi que le tutoriel, restent les références privilégiées pour aller au-delà de ce qui est exposé dans ce chapitre. Certains aspects évoqués dans ce chapitre seront développés dans les chapitres suivants.

En revenant à la page principale du site sus-cité `https://www.python.org/`, vous trouverez également différentes distributions de Python à télécharger et installer sur vos ordinateurs personnels (plus que conseillé !), notamment pour Windows ou MacOS. Pour une distibution Linux, renseignez-vous sur les forums ; il y a moyen de l'installer avec les moteurs d'installation (`yum install` pour Fedora par exemple). Sur certains systèmes, Python est installé par défaut (vérifier dans ce cas la version). Il existe des différentes notables entre les versions 2 et 3, les rendant incompatibles. Nous utiliserons la version 3 en TP. Choisissez donc la version 3 la plus récente.

Il peut également être utilise de télécharger un environnement de programmation (essentiellement un programme incluant dans un même environnement graphique un éditeur de code et un interpréteur permettant de taper des instructions en ligne, et de visualiser les résultats de vos programmes). Certains environnements sont généralistes (prévus pour la programmation dans plusieurs langages différents), comme Eclipse ; d'autres sont plus spécifiques à Python, comme IDLE, conçu par le concepteur de Python, ou Pyzo, que nous utiliserons en TP. Ce dernier est donc à privilégier sur vos ordinateurs personnels , afin de vous habituer. IDLE est plus minimaliste, pour ceux qui préfère la sobriété. Certains environnements viennent avec une distribution précise de Python (qu'il n'est dans ce cas pas nécessaire d'installer avant).

On peut aussi utiliser un éditeur de texte standard, comme `emacs`, puis lancer Python sur le fichier en ligne de commande. La plupart des éditeurs reconnaissent le langage Python et proposent une coloration syntaxique, mais ils ne disposent pas en revanche des aides syntaxiques lorsqu'on tape le nom d'une fonction.

I Python dans le paysage informatique

- Python est un langage hybride, adapté à la programmation impérative, tout en utilisant certaines facilités de la programmation objet, au travers de méthodes, applicables à des objets d'un certain type

(on parle de classe). Il existe la possibilité de créer de nouvelles classes d'objets, et d'y définir des méthodes. Nous n'explorerons pas ces possibilités cette année, même s'il s'agit d'un point de vue très important.

- Python est un langage de haut niveau. Il gère tout le côté matériel sans que le programmeur ait à s'en préoccuper. La seule tâche du programmeur est l'aspect purement algorithmique. Le haut niveau se traduit aussi par la possibilité d'utiliser des méthodes élaborées sur les objets (recherche d'éléments dans une liste, tri, etc), et d'importer des modules complémentaires spécialisés suivant les besoins, dans lesquels sont définis des fonctions complexes (calcul matriciel, résolutions d'équations différentielles, résolutions de systèmes linéaires, méthode de Newton...). Les méthodes et fonctions ainsi définies cachent une grande partie de la technique au programmeur, qui n'a donc qu'à se préoccuper de l'essentiel. L'avantage est une facilité de programmation, et une utilisation de méthodes le plus souvent optimisées (plus efficaces en général que ce que peut espérer un programmeur débutant). L'inconvénient est une connaissance beaucoup plus floue de la complexité des algorithmes écrits. Par ailleurs, ces fonctions et méthodes, ainsi que la gestion du matériel, sont optimisés pour la situation la plus générale possible, mais pas pour les situations particulières. C'est pour cela qu'un programme en langage de plus bas niveau (en C par exemple) est souvent plus efficace, s'il est bien conçu, qu'un programme en Python. Pour cette raison, il est fréquent en entreprise d'utiliser Python comme langage de prototypage, puis de traduire en C pour plus d'efficacité, une fois que l'algorithmique est en place. Cela permet de scinder l'écriture du code en 2 phases, la phase algorithmique, et la phase programmation. De la sorte, il est possible d'utiliser du personnel plus spécialisé (des concepteurs d'algorithmes d'une part, et des programmateur d'autre part, plus au fait du fonctionnement-même de l'ordinateur)
- Python est un langage semi-compilé. Il ne nécessite pas une compilation spécifique avant lancement. Il est compilé sur le tas, à l'exécution, de sorte à repérer certaines erreurs de syntaxe avant le lancement-même du programme, mais il ne ressort pas de fichier compilé utilisable directement. L'avantage est une certaine souplesse liée à la possibilité d'utiliser Python en ligne de commande (n'exécuter qu'une seule instruction dans un shell) : on peut ainsi tester certaines fonctions avant de les utiliser dans un programme, ce qui est parfois assez pratique. L'inconvénient est une lenteur d'exécution du programme définitif, puisqu'il faut le compiler à chaque utilisation : un langage compilé peut directement être exécuté à partir du code compilé, en assembleur, proche du langage machine, et est donc plus rapide à l'exécution. C'est une autre raison pour que Python servent plus au prototypage qu'à la finalisation, notamment pour tous les dispositifs nécessitant une réactivité assez importante (téléphones...)
- Contrairement à beaucoup de langages, Python est un langage dans lequel la présentation est un élément de syntaxe : les délimitations des blocs d'instruction se font par indentation (en pratique de 4 caractères), contrairement à de nombreux autres langages qui utilisent des débuts et fins de blocs du type `begin` et `end`. Ces indentations sont donc ici à faire de façon très rigoureuses pour une correction du programme. Une mauvaise indentation peut amener une erreur de compilation, voire un mauvais résultat (ce qui est plus dur à détecter). L'avantage est une plus grande clarté (imposée) du code par une délimitation très visuelle des blocs (même si cette indentation est de fait très fortement conseillée dans les autres langages, mais de nombreux programmeurs débutants la négligent).

Une fonction essentielle, et inclassable, est la fonction d'aide, que vous pouvez appliquer à la plupart des noms d'objets, de modules, de fonctions :

```
help()  # ouverture de la page d'aide de l'objet spécifié
```

Par exemple `help(print)` renvoie la page d'aide sur la fonction `print()` réalisant l'affichage d'une chaine de caractères.

II Les variables

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).

II.1 Affectation

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 :

```
Affectations avec opération:
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 (division réelle)
a //= 2  # Équivaut à a = a // 2 (division entière)
a %= 3   # Équivaut à a = a % 3 (reste modulo 3)
a **= 4  # Équivaut à a = a ** 4 (puissance)
etc.
```

II.2 Affichage

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 chaine de caractère. 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 (voir section « chaînes de caractères »).

II.3 Type et identifiant

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, en norme IEEE 754 sur 64 bits, voir chapitre 1)
complex  # Nombres complexes
bool     # Booléens (True / False)
list     # Listes
set      # Ensembles
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({1,2})
<class 'set'>
>>> type('abc')
<class 'str'>
>>> type(lambda x:x*x)
<class 'function'>
```

D'autres types plus complexes peuvent être rattachés à un type donné, comme les itérateurs de liste (objet permettant d'énumérer les uns après les autres les éléments d'une liste, ce qui permet d'appliquer à la liste une boucle `for`.

Nous rencontrerons d'autres types dans des modules spécifiques, en particulier les type `array` et `matrix` du module `numpy`, pour les tableaux et les matrices.

III Objets et méthodes

Python utilise certains concepts de la programmation objet, notamment au travers la possibilité de modifier un objet en lui appliquant une méthode. Pour lister l'ensemble des méthodes associées à une classe d'objets nommée `cl` :

```
help(cl)    # aide associée à la classe 'cl', en particulier, description des
            # méthodes définies sur les objets de cette classe
```

III.1 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 de réels (retourne un objet de type float même si
               # le résultat théorique est un entier; peut être appliqué à
               # des entiers, qui sont alors convertis en flottants
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)
               # s'applique à des entiers ou réels.
divmod(x)      # renvoie le couple (x // y, x % y)
               # permet de faire les deux calculs en utilisant une seule
               # fois l'algorithme 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
```

III.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.__and__(y)   ou   x & y     ou   x and y    # et
x.__or__(y)    ou   x | y     ou   x or y     # ou
x.__xor__(y)   ou   x ^ y                     # ou exculsif
                                 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)
```

Attention, suivant la classe de `y`, le test de l'appartenance est plus ou moins long. Ainsi, si y est une liste, il se fait en temps linéaire, alors qu'il est en temps constant pour un ensemble.

III.3 Les listes

Une liste est une suite ordonnée d'objets, pouvant être de type différent. Ces objets peuvent éventuellement être eux-même des listes (listes imbriquées). Ils peuvent même être égaux à la liste globale (définition récursive).

```
Notation d'une liste
[1,2,3,4]      # énumération des objets entre []
[[1,2],[1],1] # liste dont les deux premiers termes sont des listes
[]             # liste vide
```

L'accès à un élément d'une liste est direct (contrairement à de nombreux langages, ou l'accès se fait par chaînage, en suivant les pointeurs à partir du premier élément). L'indexation des éléments commence à 0 :

```
Accès aux éléments d'une liste par indexation positive:
>>> li = [1,2,3]
>>> li[0]
1
>>> li[2]
3
>>> li[3]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IndexError: list index out of range
```

On peut aussi accéder aux éléments par indexation négative. Le dernier élément de la liste est alors numéroté -1. Avec la liste de l'exemple précédent, on obtient par exemple :

```
>>> li[-1]
3
>>> li[-3]
1
>>> li[-4]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IndexError: list index out of range
```

Cette possibilité est surtout intéressante pour accéder aux derniers éléments d'une liste.

Voici une liste des opérations et méthodes les plus utilisées sur les listes. Pour une liste plus exhaustive, utilisez `help(list)`. On indique par un (m) le fait que la méthode modifie directement la liste sur laquelle elle agit. Les autres retourne le résultat en sortie de fonction.

```
Opérations et méthodes applicables à une liste:
len(L)              # longueur (nombre d'éléments) de L
L1 + L2             # concaténation des listes
n * L               # pour n entier: concaténation répétée de L avec elle-même.
L.append(a)         # ajout de l'objet a en fin de liste (m)
L.insert(i,a)       # insertion de l'objet a en position i (m)
L.remove(a)         # retrait de la première occurrence de a (m)
L.pop(i)            # retrait et renvoi de l'élément d'indice i (m)
                    # par défaut, si i non précisé: dernier élément
L.index(a)          # position de la première occurrence de a
                    # ValueError si a n'est par dans la liste
L.count(a)          # nombre d'occurrences de a dans la liste
a in L              # teste l'appartenance de a à L
L.copy()            # copie simple de L
                    # attention aux problèmes de dépendance des attributs
L.reverse()         # retourne la liste (inversion des indexations) (m)
L.sort()            # trie la liste dans l'ordre croissant (m)
```

```
                    # (si composée d'objets comparables)
                    # voir ci-dessous pour des paramètres
```

Le tri d'une liste possède deux paramètres : un qui permet de préciser si le tri est croissant ou décroissant, l'autre qui permet de définir une clé de tri (une fonction : on trie alors la liste suivant les valeurs que la fonction prend sur les éléments de la liste)

```
Paramètres de tri:
li.sort()               # tri simple
li.sort(reverse=True)   # tri inversé
li.sort(key = f)        # tri suivant la clé donnée par la fonction f
```

La fonction donnant la clé de tri peut être une fonction prédéfinie dans un module (par exemple la fonction `sin` du module `math`), une fonction définie précédemment par le programmeur (par `def`), ou une fonction définie sur place (par `lambda`). Voir plus loin pour plus d'explications à ce propos.
Lors de la copie d'une liste (par la méthode `copy`, ou par affectation directe), les attributs gardent même adresse. Donc modifier la copie modifie aussi l'original, et réciproquement.
Pour copier avec modification d'adresse des attributs, utiliser un slicing (voir ci-dessous) `M = L[:]`. Cela ne rend pas la copie complètement indépendante de l'original, si les objets de la liste sont eux-même des structures composées (listes, ensembles...). Il faut dans ce cas faire une copie profonde, à l'aide d'une fonction définie dans le module `copy` :

```
Copie profonde (complètement indépendante à tous les niveaux):
>>> import copy
>>> M = copy.deepcopy(L)
```

Aux méthodes précédentes, s'ajoute pour les listes une technique particulièrement utile, appelée saucissonnage, ou slicing. Il s'agit simplement de l'extraction d'une tranche de la liste, en précisant un indice initial et un indice final. On peut définir un pas p, ce qui permet de n'extraire qu'un terme sur p (par exemple les termes d'indice pair ou impair en prenant $p = 2$)

```
Technique de slicing:
L[i:j]        # Extraction de la tranche [L[i], ... , L[j-1]]
L[i:j:p]      # De même de p en p à partir de L[i], tant que i+k*p < j
```

À noter que :
- Si le premier indice est omis, il est pris égal à 0 par défaut.
- Si le deuxième indice est omis, il est pris égal à la longueur de la liste par défaut (on extrait la tranche finale)
- Si le troisième indice est omis, il est pris égal à 1 par défaut (cas de la première instruction ci-dessus)
- Un pas négatif permet d'inverser l'ordre des termes
- Le slicing est possible aussi avec des indexations négatives.

Par exemple :

```
>>> M = [0,1,2,3,4,5,6,7,8,9,10]
>>> M[3:6]
[3, 4, 5]
>>> M[2:8:2]
[2, 4, 6]
>>> M[:3]
[0, 1, 2]
>>> M[3::3]
[3, 6, 9]
>>> M[::5]
[0, 5, 10]
```

```
>>> M[:2:4]
[0]
>>> M[:5:4]
[0, 4]
>>> M[-3:-1]
[8, 9]
>>> M[2:6:-3]
[]
>>> M[6:2:-3]
[6, 3]
```

III.4 Les ensembles

La structure d'ensemble (`set`) ressemble à celle de liste, mais sans ordre défini sur les ensembles, et sans répétition possible des éléments.

```
Notation d'un ensemble
{1,2,3,4}     # énumération des objets entre {}
{1,1,2,3,4,4} # même ensemble que le premier
set()         # ensemble vide
```

Attention, `{}` désigne non pas l'ensemble vide, mais le dictionnaire vide (nous n'évoquerons pas la structure de dictionnaire cette année ; si vous êtes intéressés, découvrez-la par vous même, c'est une structure assez importante).

```
Principales opérations et méthodes applicables à un ensemble
len(S)                    # cardinal
S.add(a)                  # ajoute a à l'ensemble S, si pas déjà  présent (m)
S.discard(a)              # enlève a de S si c'en est un élément (m)
S.remove(a)               # comme discard, mais retourne une erreur
                          # si a n'est pas élément de S (m)
S.pop()                   # retourne et enlève un élément au hasard de S (m)
S.intersection(T)         # retourne l'intersection de S et T
S.union(T)                # retourne l'union de S et T
S.difference(T)           # retourne les éléments de S qui ne sont pas dans T
S.symmetric_difference(T) # retourne la différence symétrique
S.isdisjoint(T)           # retourne True ssi S et T sont disjoints
S.issubset(T)             # retourne True ssi S est inclus dans T
a in S                    # teste l'appartenance de a à S
S.copy()                  # copie simple de S
                          # Attention aux problèmes de dépendance des attributs
```

Ici encore, on peut faire une copie complètement indépendante à l'aide de `deepcopy` du module `copy`

Du fait même de la nature d'un ensemble, on ne peut pas extraire un élément par indexation, ni appliquer la méthode de slicing à un ensemble.

III.5 Les tuples

Les `tuples` sont des n-uplets. Ils sont notés entre parenthèses :

```
Notation d'un tuple:
(1,2,3,4)     # énumération des objets entre ()
```

Dans certains cas, l'omission des parenthèses est supportée.

Contrairement à une liste, ils ont une taille fixée (aucune méthode ne peut ajouter une coordonnée en place). On peut accéder à une coordonnée par indexation, mais pas la modifier :

```
>>> u = (2,3)
>>> u[1]
3
>>> u[1]=2
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'tuple' object does not support item assignment
```

```
Principales opérations et méthodes applicables à un tuple
len(U)                    # nombre de coordonnées
U+V                       # retourne la concaténation
n * U   ou   U * n        # concaténation répétée n fois
U.count(a)                # retourne le nombre d'occurrences de a
U.index(a)                # retourne le premier indice de a, ou une
                          # erreur si a n'est pas un attribut
a in U                    # teste l'appartenance de a à U
```

III.6 Les chaînes de caractères

Il s'agit d'un assemblage non interprété de caractères.

```
Notation d'une chaîne de caractères:
'abcd'       # énumération des caractères successifs entre ''
"abcd"       # énumération des caractères successifs entre ""
"""abcd"""   # énumération des caractères successifs entre """ """ (raw string)
```

L'intérêt d'avoir 2 délimiteurs possibles est de pouvoir utiliser sans problème une apostrophe ou un guillemet dans une chaîne de caractère. Si les deux apparaissent dans la chaîne, on peut utiliser une raw string. Celle-ci permet aussi d'utiliser des retours-chariot explicites. Sans raw string, les retours à la ligne se notent avec le caractère spécial `'\n'`.
Les raw string sont aussi utilisés pour commenter les fonctions (voir plus loin)

L'accès à un caractère d'une chaîne se fait par indexation de la même manière que pour les listes. Le premier caractère est indexé par 0. Les indexations négatives sont possibles dans la même mesure que pour les listes.

```
Accès aux éléments d'une chaîne par indexation positive:
>>> ch = 'abcd'
>>> ch[2]
'c'
>>> ch[-1]
'd'
```

Les slicings (saucissonnages) peuvent se faire également sur les chaînes de caractères, de la même manière que pour les listes :

```
>>> ch = 'azertyuiopqsdfghjklmwxcvbn'
>>> ch[3:12]
'rtyuiopqs'
>>> ch[20:10:-1]
```

```
'wmlkjhgfds'
>>> ch[::2]
'aetuoqdgjlwcb'
```

Une liste (non exhaustive) des opérations et méthodes les plus utiles. Voir `help(str)` pour (beaucoup) plus.

```
Opérations et méthodes sur les chaînes de caractères:
len(c)              # longueur (nombre de caractères)
print(c)            # affichage à l'écran de la chaîne c
c + d               # concaténation des chaînes
n * c               # concaténation répétée n fois
c.join(li)          # concaténation des chaînes de la liste li
                    # valable avec d'autres itérables
c.split(sep)        # scinde c en une liste de chaînes, en
                    # recherchant les séparateurs sep
c.center(n)         # centre n dans une chaîne de longueur n
c.format(x,y,...)   # insertion des nombres x, y,... aux endroits
                    # délimités par des {} dans la chaîne
d in c              # teste si d est sous-chaîne de c
c.count(d)          # nombre d'occurrences ne se recouvrant pas de la chaîne d
c.find(d)           # indice de la première occurrence de d
                    # renvoie -1 si pas d'occurrence
c.index(d)          # indice de la première occurrence de d
                    # erreur si pas d'occurrence
c.rfind(d)          # de même que find pour la dernière occurrence
c.rindex(d)         # de même que index pour la dernière occurrence
c.replace(d,e)      # remplace toutes les sous-chaînes d par e
```

Découvrez les nombreuses autres méthodes avec `help`. En particulier, plusieurs méthodes pour gérer majuscules et minuscules, ou pour des tests sur la nature des caractères.

Nous précisons la méthode `format`, particulièrement utile pour paramétrer l'aspect des insertions de nombres dans le texte. Pour commencer, nous donnons des exemples simples :

```
>>> 'Il y a {} arbres'.format(10)
'Il y a 10 arbres'
>>> 'Il y a {} tulipes et {} roses'.format(3,4)
'Il y a 3 tulipes et 4 roses'
```

Il existe différents paramètres sur le format des valeurs numériques. Ces paramètres sont à indiquer entre les accolades :

```
:g      # choisit le format le plus adapté
:.4f    # Écriture en virgule flottante, fixe le nombre de décimales, ici 4.
:.5e    # Écriture en notation scientifique, fixe le nombre de décimales, ici 5.
:<15.2e # Fixe la longueur de la chaîne (elle est remplie par des espaces),
        # et justifie à gauche. Le 2e a la même signification que plus haut.
:>15.2e # Fixe la longueur de la chaîne, et justifie à droite.
:^15.2e # Fixe la longueur de la chaîne, et centre.
```

Par exemple, après import du module `math` :

```
>>> 'pi␣vaut␣environ␣:␣{:^12.6f}!'.format(math.pi)
'pi␣vaut␣environ␣:␣␣␣3.141593␣␣!'
```

III.7 Les itérateurs

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` (voir section sur les boucles)
L'itérateur le plus utile pour nous sera `range`, énumérant des entiers dans un intervalle donné.

```
L'itérateur range:
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 $b-1$.
Certaines classes composées possèdent une méthode `objet.__iter__()`, permettant de le transformer en itérateur. Cette conversion est rarement effectuée explicitement, mais intervient de façon implicite. Elle permet de faire des boucles dont l'indice porte sur les éléments de ces structures (par exemple les éléments d'une liste, d'un ensemble ou d'une chaîne de caractère).

```
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 chaine.
```

On dit que les types `list`, `set` et `str` sont itérables.
Cela permet également des définitions de structures composées par construction puis par compréhension, en rajoutant une condition :

```
Définitions de listes et ensembles par compréhension:
[ f(i) for i in range(n) if g(i)] # liste des f(i) si la condition
                                  # g(i) est satisfaite
De même avec les ensembles. La clause if est facultative.
range peut être remplacé par un itérateur ou un objet itérable.
```

Par exemple :

```
>>> [i*i for i in [2,5,7]]
[4, 25, 49]
>>> {i for i in range(100) if i%7 == 2}
{65, 2, 37, 72, 9, 44, 79, 16, 51, 86, 23, 58, 93, 30}
```

Remarquez le désordre !

III.8 Conversions de types

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

```
Conversions de type.
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 itérable x (set, tuple, str, range)
                  # en objet de type liste.
set(x)            # Convertit un itérable x en set
```

```
tuple(x)          # Convertit un itérable x en tuple
eval(x)           # Convertit la chaîne x en un objet adéquat, si possible
```

Par exemple :

```
>>> float(2)
2.0
>>> complex(2)
(2+0j)
>>> str({2,3,1})
'{1, 2, 3}'
>>> str(77+2)
'79'
>>> list({1,2,3})
[1, 2, 3]
>>> list('ens')
['e', 'n', 's']
>>> set([1,2,3])
{1, 2, 3}
>>> set(range(10))
{0, 1, 2, 3, 4, 5, 6, 7, 8, 9}
>>> tuple('abc')
('a', 'b', 'c')
>>> eval('2')
2
>>> eval('[2,3]')
[2, 3]
```

IV Structuration d'un programme

IV.1 Notion de programme

Un programme est constitué d'une succession d'instructions, écrites dans un fichier, structurée de façon rigoureuse à l'aide de la syntaxe appropriée. Cette succession d'instruction peut être précédée par la définition d'un certains nombres de fonctions (dans le même fichier, ou dans un fichier annexe qu'il faut alors importer) qui pourront alors être utilisées dans le programme

Les instructions peuvent être :

- une affectation
- l'utilisation d'une fonction, ou d'une méthode sur un objet, ou un calcul
- une structure composée (conditionnelle, itérative, gestion d'exception...)

Les structures composées peuvent porter sur des blocs de plusieurs instructions. Il est important de pouvoir délimiter de façon très précise ces blocs. Contrairement à beaucoup de langages utilisant des délimiteurs de type `begin` et `end`, la délimitation des blocs en Python se fait uniquement par l'indentation. Par exemple, comparons le deux petits programmes suivants :

```
# Importance des indentations
x = y = 1
for i in range(5):
    x += 1
    y += 1
print(x,y)
```

et :

```
# Importance des indentations
x = y = 1
for i in range(5):
    x += 1
y += 1
print(x,y)
```

Le premier renvoie `6 6` tandis que le second renvoie 6 2. En effet, l'incrémentation de y n'est pas dans la boucle, et n'a donc lieu qu'une fois dans le second programme.

La taille de l'indentation n'a pas d'importance, mais dans un bloc donné, il faut être cohérent (la même indentation sur chaque ligne du bloc). Cependant, dans un soucis d'unification, on adopte généralement la règle suivante (respectée par les environnements adaptés *via* la tabulation) :

> **Un nouveau bloc sera déterminé par une indentation de 4 caractères de chacune de ses lignes.**

Évidemment, des blocs peuvent être imbriqués les uns dans les autres. Dans ce cas, on additionne les indentations (un bloc à l'intérieur d'un autre bloc principal sera donc indenté de 4 espaces supplémentaires, soit 8 caractères).

Un programme peut interagir avec l'utilisateur, soit en lui demandant d'entrer des données (entrée), soit en agissant sur les périphériques (sortie).

- La demande d'entrée de donnée par l'utilisateur se fait avec :

  ```
  input('texte')
  ```

  Cette instruction affiche 'texte' à l'écran, puis attend que l'utilisateur entre une donnée, validée par l'Entrée. On associe souvent cette instruction à une affectation de sorte à conserver la donnée fournie :

  ```
  >>> x = input('Que voulez-vous me dire? ')
  Que voulez-vous me dire? ZUT!
  >>> x
  'ZUT!'
  ```

  Ici, le texte `'Que voulez-vous me dire? '` a été affiché par l'ordinateur, et l'utilisateur a tapé 'ZUT !', valeur qui a été stockée dans la variable x. Évidemment, cette fonction `input` est plus intéressante en programme qu'en ligne de commande, afin de pouvoir donner la main à l'utilisateur.

  Attention, toute donnée entrée par l'utilisateur l'est au format `str`, y compris les valeurs numériques. Si on veut les utiliser dans un calcul, il faut donc d'abord les convertir, à l'aide de la fonction `eval` :

  ```
  >>> x = input('Entrez une valeur : ')
  Entrez une valeur : 2
  >>> x
  '2'
  >>> x = eval(input('Entrez une valeur : '))
  Entrez une valeur : 2
  >>> x
  2
  ```

- Nous ne ferons cette année que des sorties sur écran, à l'aide de la fonction `print`, ou des sorties sur des fichiers. Nous reparlerons plus loin de la lecture et l'écriture de fichiers en Python.

IV.2 Les 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 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'instruction `lambda` :

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

Cela permet par exemple de donner une fonction en paramètre (clé de tri par exemple), sans avoir à la définir globalement par `def`

IV.3 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 positif
- Les instructions `instructions2` sont effectuées uniquement si le `test1` est négatif et le `test2` positif
- Les instructions `instructions3` sont effectuées uniquement si le texttttest1 et le `test2` sont négalifs et le `test3` est positif
- ...
- Les instructions `instructions4` sont effectuées dans tous les autres cas.

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
```

Avertissement 2.4.1

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

Remarque 2.4.2

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`).

IV.4 Les 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 valeur 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
```

Si les instructions ne dépendent par de i, on peut remplacer `range(1,n+1)` par `range(n)`.
Que fait par exemple le code suivant ?

```
x = 0
for i in range(1,1001):
    x += 1 / i
```

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.

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

En revanche, ce n'est pas le cas de la boucle `for` sur d'autres objets itérables.

IV.5 La gestion des exceptions

L'exécution de certaines instructions peut fournir des erreurs. Il existe différents types d'erreur en Python, par exemple : `ZeroDivisionError`, `ValueError`, `NameError`, `TypeError`, `IOError`, etc.

Dans certaines conditions, on veut pouvoir continuer l'exécution du programme tout de même, éventuellement en adaptant légèrement l'algorithme au cas problématique détecté. On dispose pour cela de la structure `try`, qui dans sa version la plus simple, s'écrit :

```
try:
    instructions1
except:
    instructions2
else:
    instructions3
```

L'ordinateur essaie d'effectuer la série d'`instructions1`.
- S'il survient une erreur d'exécution, il effectue les `instructions2` et saute les `instructions3`
- Sinon, il saute les `instructions2` et exécute les `instructions3`.

On peut aussi rajouter une clause `finally:` qui sera exécutée dans les deux situations.

La clause `except` est obligatoire, la clause `else` est facultative. Si on n'a rien a faire dans la clause `except`, on peut utiliser l'instruction vide `pass`.

La gestion des exceptions peut différer suivant le type d'erreur qui intervient. Dans certains conditions, il faut donc pouvoir distinguer plusieurs démarches à suivre suivant l'erreur obtenue. En effet, considérons l'exemple suivant :

```
y = input("Entrez une valeur à inverser : ")
try:
    x = 1 / eval(y)
except:
    print("0 n'a pas d'inverse, Banane!")
else:
    print(x)
```

Lançons l'exécution du programme. Si on entre la valeur 2, on obtient en sortie 0.5. Si on entre la valeur 0, on se fait injurier. Mais si on entre une lettre (disons a), c'est maintenant l'évaluation de a qui pose problème (car il ne s'agit pas d'une valeur numérique). Ainsi, l'ordinateur va détecter une erreur et donc exécuter la clause `except`. On va donc se retouver face à un « 0 n'a pas d'inverse, Banane : » assez inapproprié. Pour cela, il faut pouvoir ne considérer que le cas où l'erreur obtenue est une division par 0. En effectuant `1/0` dans la console, on récupère le nom de cette erreur, et on peut indiquer que la clause `except` ne doit porter que sur ce type d'erreur :

```
y = input("Entrez une valeur à inverser : ")
try:
    x = 1 / eval(y)
except ZeroDivisionError:
    print("0 n'a pas d'inverse, Banane!")
else:
    print(x)
```

Le comportement n'est pas changé si on entre les valeurs 2 ou 0. En revanche, avec la lettre a, on obtient :

```
Entrez une valeur à inverser : a
Traceback (most recent call last):
  File "try.py", line 3, in <module>
    x = 1 / eval(y)
  File "<string>", line 1, in <module>
NameError: name 'a' is not defined
```

Si on veut pouvoir gérer différemment plusieurs types d'exceptions, on peut mettre plusieurs clauses `except` :

```
y = input("Entrez une valeur à inverser : ")
try:
    x = 1 / eval(y)
except ZeroDivisionError:
    print("0 n'a pas d'inverse, Banane!")
except NameError:
    print("C'est ça que t'appelles une valeur, Banane?")
else:
```

```
    print(x)
```

Cette fois, entrer la lettre a provoque le second message d'insultes.

On peut aussi provoquer volontairement des erreurs (on dit « soulever des erreurs ou des exceptions »). Cela se fait avec l'instruction `raise` :

```
raise NomErreur('texte')
```

Le `NomErreur` doit être un des types d'erreur existant (on peut éventuellement en définir d'autres, mais nous n'étudierons pas cette année cette possibilité), le texte est la petite explication s'affichant lors du message d'erreur :

```
>>> raise TypeError('Quel est le rapport avec la choucroute?')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: Quel est le rapport avec la choucroute?
```

On peut par exemple utiliser cela pour définir une fonction sur un intervalle donné. Supposons que nous voulions définir $f : x \mapsto x^2 - 2$ sur l'intervalle $[-1, 3]$ (allez savoir pourquoi !). On peut vouloir soulever une erreur si on cherche à évaluer f en un réel x hors de cet intervalle. Voici comment faire :

```
def f(x):
    if (x<-1) or (x>3):
        raise ValueError('valeur hors du domaine de définition de f')
    return x ** 2 - 2

print('f({})={}'.format(2,f(2)))
print('f({})={}'.format(4,f(4)))
```

L'exécution de ce programme retourne le résultat suivant :

```
f(2)=2
Traceback (most recent call last):
  File "raise.py", line 8, in <module>
    print('f({})={}'.format(4,f(4)))
  File "raise.py", line 4, in f
    raise ValueError('valeur hors du domaine de définition de f')
ValueError: valeur hors du domaine de définition de f
```

À noter que soulever une erreur arrête *de facto* le programme, à moins d'inclure cette erreur dans une clause `try` de gestion des exceptions.

Enfin, on peut utiliser ce qu'on vient de voir dans le simple but de redéfinir les messages d'erreurs. En reprenant le premier exemple, et en ne gérant que l'exception relative à la division par 0, on peut par exemple ecrire :

```
y = input("Entrez une valeur à inverser : ")
try:
    x = 1 / eval(y)
except ZeroDivisionError:
    raise ZeroDivisionError("0 n'a pas d'inverse, Banane!")
else:
    print(x)
```

Pour la valeur 0, on obtient alors :

```
Traceback (most recent call last):
  File "try.py", line 3, in <module>
    x = 1 / eval(y)
ZeroDivisionError: division by zero

During handling of the above exception, another exception occurred:

Traceback (most recent call last):
  File "try.py", line 5, in <module>
    raise ZeroDivisionError("0 n'a pas d'inverse, Banane!")
ZeroDivisionError: 0 n'a pas d'inverse, Banane!
```

À noter que le comportement est totalement différent du premier exemple, puisqu'ici, le programme est interrompu s'il y a une division par 0.

V Modules complémentaires

Nous terminons ce chapitre par une description rapide d'un certain nombre de modules qui nous seront utiles en cours d'année.

V.1 Utilisation d'un module

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 \texttt{mod}. On peut alors utilier les fonctions
  #de ce module en les faisant précéder du préfixe \texttt{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()
```

  Un alias d'usage très courant est :

```
import numpy as np
```

- **Import d'une fonction d'un module**

```
from mod import fonct
  # Cette instruction permet d'importer la définition de la fonction
```

```
  # \texttt{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 précédemment), 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 plus d'une gestion plus lourde, si le module est gros.

Nous passons en revue les modules qui nous seront les plus utiles. Il existe une foule d'autres modules (se renseigner sur internet en cas de besoin), dans des domaines très spécifiques.

V.2 Le module `math`

Ce module contient les fonctions et constantes mathématiques usuelles, dont nous citons les plus utiles (utiliser `help()` pour avoir la liste de toutes les fonctions du module)

```
### Exponentielle et logarithme ###
e             # constante e, base de l'exponentielle
exp(x)        # exponentielle de x
log(x)        # logarithme népérien
log10(x)      # logarithme en base 10

### Fonctions trigonométriques ###
pi            # le nombre pi
cos(x)        # cosinus
sin(x)        # sinus
tan(x)        # tangente
acos(x)       # arccos
asin(x)       # arcsin
atan(x)       # arctan

### Fonctions hyperboliques ###
cosh(x)       # cosinus hyperbolique
sinh(x)       # sinus hyperbolique
tanh(x)       # tangente hyperbolique

### parties entières ###
floor(x)      # partie entière au sens mathématique
ceil(x)       # partie entière par excès

### Autres fonctions ###
sqrt(x)       # racine carrée
factorial(x)  # factorielle (pour x entier)
```

V.3 Le module `numpy` (calcul numérique)

Il est impossible de faire le tour de façon rapide de ce module définissant un certain nombre d'objets indispensables en calcul numérique, en particulier l'objet matriciel, et toutes les règles associées. Nous ne faisons qu'un survol rapide de cet aspect, en vous laissant découvrir le reste à l'aide de l'aide ou des guides d'utilisation.
Le module `numpy` contient lui-même des sous-modules, en particulier le sous module `linalg`. Nous ne précisons pas dans ce qui suit le préfixe relatif au module principal `numpy` (souvent abrégé en `np`), mais nous précisons les préfixes correspondant aux sous-modules

```
### La structure array (np.array): tableau multidimentionnel ###
array(liste) # convertisseur: transforme une liste en tableau
             # En imbriquant des listes dans des listes, on peut
             # obtenir des tableaux multidimentionnels

### Exemples ###
array([1,2,3])      # tableau à 3 entrées (matrice ligne)
array([[1,2],[3,4]]) # matrice 2x2 de première ligne (1,2) seconde ligne (3,4)
array([[1],[2],[3]]) # matrice colonne

### fonctions de structure sur les array ###
shape(A)     # format du tableau (nombre de lignes, de colonnes...,
             # sous forme d'un tuple
rank(A)      # profondeur d'imbrication (dimension spatiale)
             # ATTENTION, ce n'est pas le rang au sens mathématique!
```

Quelques différences entre les listes et les `np.array` :

- Homogénéité : toutes les entrées doivent être de même type
- Le format est immuable. La taille est définie à partie de la première affectation. Pour initialiser, il faut donc souvent créer un tableau préalablement rempli de 0 ou de 1.

```
### Création de tableaux particuliers ###
ones(n)         # Crée une matrice ligne de taille n constituée de 1.
ones((n1,n2))   # matrice de taille n1 x n2, constituée de 1
                # se généralise à des tuples
zeros(n)        # Crée une matrice ligne de taille n constituée de 0.
zeros((n1,n2))  # matrice de taille n1 x n2, constituée de 0
                # se généralise à des tuples
eye(n)          # matrice identité d'ordre n (des 1 sur la diagonale)
diag(d0,...,dn) # matrice diagonale de coefficients diagonaux d1,...,d_n
linspace(a,b,n) # matrice ligne constituée de n valeurs régulièrement réparties
                # entre $a$ et $b$
arange(a,b,p)   # matrice ligne des valeurs de a à b en progressant de pas p
fromfunction(f,(n,))          # matrice ligne remplie des valeurs f(k) pour
                              # k de 0 à n-1
fromfunction(f,(n1,dots,nd))  # De même pour plus de dimension
                              # f dépend ici de d variables
```

Les `np.array` permettent le calcul matriciel. Les opérations matricielles usuelles sont définies :

```
### Opérations matricielles ###
A + B             # Somme de deux matrices
a * M             # Multiplication par un scalaire
A * B             # Produit coefficient par coefficient (produit de Schur)
```

```
                  # ATTENTION, ce n'est pas le produit matriciel
dot(A,B)          # Produit matriciel
linalg.det(A)     # déterminant
trace(A)          # trace
transpose(A)      # transposée
power(A,n)        # A puissance n
linalg.inv(A)     # inverse de A
linalg.eigvals(A) # valeurs propres de A
linalg.eig(A)     # valeurs propres et base de vecteurs propres
```

La résolution de systèmes de Cramer est programmée aussi :

```
### Résolution de systèmes de Cramer ###
linalg.solve(A,b) # Donne l'unique solution du système de Cramer Ax=b
```

Enfin, 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 np.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
```

V.4 Le module `scipy` (calcul scientifique)

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

```
### Calcul numérique ###
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.optimize.newton(f,a)     # de même par la méthode de la sécante
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
```

V.5 Le module `matplotlib` (tracé de courbes)

Ce module donne un certain nombre d'outils graphiques, notamment pour le tracé de courbes. Nous n'utiliserons que 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 abscisse 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.matshow(T)               # Affiche une image constituée de points dont
                             # les couleurs sont définies par les valeurs du
                             # tableau T de dimension 2 (à  voir comme un
                             # tableau de pixels)
```

V.6 Autres modules (`random`, `time`, `textttsqlite3`,...)

Parmi les autres modules que nous utiliserons, citons `random`, qui proposent des fonctions de génération aléatoire, qui peuvent rentrer en conflit avec celles définies dans `numpy.random` :

```
### Module random ###
random.randint(a,b)   # entier aléatoire entre a et b inclus.
                      # Notez la différence avec la fonction de numpy
sample(l,k)           # Liste aléatoire d'éléments distincts de la liste l
                      # Deux occurrences d'une même valeur dans l sont
                      # considérées comme distinctes.
```

A part pour cette dernière fonction qui peut s'avérer utile, ce module est plutôt moins fourni que celui de `numpy`

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

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

Cette fonction sera utilisée comme suit :

```
import time
debut = perf_counter()
```

```
instructions
fin = perf_counter()
temps_execution = fin - debut
```

Enfin, le module `sqlite3` nous sera utile pour traiter des bases de données en fin d'année. Supposons ce module importé sous le nom `sql`

```
### Module sqlite3 ###
connection = sql.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("instruction SQL")
    # Exécute l'instruction donnée, dans la syntaxe SQL,
    # dans la base pointée par le curseur
```

VI Lecture et écriture de fichiers

Nous voyons enfin comment il est possible d'accéder aux fichiers en Python (lecture, écriture). Sont définies dans le noyau initial les fonctions suivantes :

```
### Lecture, écriture dans un fichier ###
f = open('nom_fichier','r' ou 'w' ou 'a')
     # ouvre le fichier nom_fichier (accessible ensuite dans la variable f)
     # nom_fichier peut contenir un chemin d'accès, avec la syntaxe standard
     # pouvant différer suivant le système d'exploitation.
     # 'r' : ouverture en lecture seule
     # 'w' : ouverture en écriture (efface le contenu précédent)
     # 'a' : ouverture en ajout (écrit à la suite du contenu précédent)
f.readline()
     # lit la ligne suivante (en commançant à la première) (renvoie un str)
f.readlines()
     # renvoie une liste de str, chaque str étant une ligne du fichier
     # en commençant à la ligne en cours s'il y a déjà  eu des lectures
f.write('texte')
     # Écrit le texte à la suite de ce qui a été écrit depuis l'ouverture
f.close()
     # ferme le fichier f.
```

Dans le module `os`, et plus précisément le sous-module `os.path` on trouve des fonctions permettant de gérer la recherche de fichiers dans une arborescence, et la manipulation des chemins et noms de fichiers.

```
### Fonctions de manipulation de chemins dans os.path ###
abspath('chemin')   # transforme un chemin (relatif) en chemin absolu
basename('chemin')  # extrait la dernière composante du chemin
dirname('chemin')   # extrait le répertoire
isfile('chemin')    # teste si le chemin correspond à un fichier
isdir('chemin')     # teste si le chemin correspond à un répertoire
splitext('chemin')  # revoie (partie initiale, extension de fichier)
etc.
```

3 Variables informatiques

Dans ce chapitre, nous nous intéressons à la nature d'une variable informatique, donc à la structure de la mémorisation de données de calcul. Nous étudions les structures de données composées, et nous évoquons certains problèmes liés à la « mutabilité » (anglicisme largement utilisé en français) des variables.

I Notion de variable informatique

La variable est le concept fondamental de la programmation (plus précisément de la programmation impérative).

Définition 3.1.1 (Variable informatique)
Une variable est la donnée de :
- une localisation en mémoire, représentée par son adresse (identifiant de la variable)
- un nom donné à la variable pour la commodité d'utilisation (appels, affectations)

Ainsi, une variable est à voir comme une correspondance entre un nom et un emplacement en mémoire. Une variable `x` va pointer vers un certain emplacement en mémoire, en lequel il pourra être stocké des choses.

Définition 3.1.2 (Contenu d'une variable)
Le contenu d'une variable est la valeur (ou l'objet) stockée à l'emplacement mémoire réservé à la variable. Le déchiffrage du code binaire de ce contenu dépendra du `type` de la variable.

Ainsi, dire qu'une variable x est égale (à un moment donné) à 2, signifie qu'on a une variable, dont le nom est x, et qu'à l'emplacement mémoire réservé est stockée la valeur 2.

Définition 3.1.3 (Affectation)
L'affectation est l'action de définir le contenu d'une variable, c'est-à-dire, explicitement, de stocker une valeur donnée à l'emplacement mémoire associé.

L'affectation se fait toujours en associant (suivant une certaine syntaxe qui dépend du langage) la valeur à stocker et le nom de la variable. Ainsi, en Python la syntaxe est :

```
x = 3
```

Attention, l'égalité d'affectation n'est pas commutative : on place le nom de la variable à gauche, et la valeur à droite. Par ailleurs, il ne faut pas confondre l'égalité d'affectation, du test d'égalité entre deux valeurs, souvent noté différemment (par exemple, en Python, le test d'égalité est noté `a == b`).

Définition 3.1.4 (Lecture, ou appel)

La lecture, ou l'appel d'une variable est le fait d'aller récuperer en mémoire le contenu d'une variable, afin de l'utiliser dans une instruction.

La lecture se fait le plus souvent en donnant le nom de la variable :

```
>>> a = 2
>>> a
2
>>> b = a + 3
>>> b
5
```

Définition 3.1.5 (État d'une variable)

L'état momentanée d'une variable est la donnée de cette variable (donc son nom et son adresse) et de son contenu

Définition 3.1.6 (Type d'une variable)

Les variables peuvent avoir différents types, prédéfinis, ou qu'on peut définir. Le type représente la nature du contenu (un réel, un entier, un complexe...).

Le type détermine la taille à réserver en mémoire pour la variable, ainsi que l'algorithme de traduction de l'objet en code binaire et inversement.

Remarque 3.1.7 (Déclaration de variables)

- Certains langages imposent de *déclarer* les variables avant de les utiliser. C'est lors de cette déclaration que l'ordinateur attribue une localisation en mémoire à la variable. Pour le faire, il doit connaître le type de la variable. Ainsi, on déclare toujours une variable en précisant son type.
- Certains langages (dont Python) dispense l'utilisateur de cette déclaration préalable des variables. Dans ce cas, un emplacement en mémoire est alloué à une variable lors de l'affectation. Le type (nécessaire pour savoir quelle place attribuée) est déterminé automatiquement, suivant la nature de la valeur stockée lors de cette affectation.

```
>>> x=3
>>> type(x)
<class 'int'>
>>> x=3.
>>> type(x)
<class 'float'>
>>> x=(3,4)
>>> type(x)
<class 'tuple'>
```

Définition 3.1.8 (Typage dynamique, typage statique)
On dit qu'un langage de programmation a un typage statique s'il impose la déclaration préalable. Dans le cas contraire, on parle de typage dynamique.

L'attribution automatique du type peut parfois être ambiguë. Par exemple, si l'utilisateur veut stocker la valeur 3, comme indiqué plus haut, Python va le stocker sous un type entier. Mais l'utilisateur pouvait avoir envie d'en faire une utilisation réelle. Cela est possible du fait que le type d'une variable peut changer :

```
>>> x = 3
>>> type(x)
<class 'int'>
>>> x /= 2
>>> x
1.5
>>> type(x)
<class 'float'>
```

La variable `x` est passée du type entier au type réel.

Définition 3.1.9 (Typage faible)
On dit qu'un langage a un typage faible (ou est faiblement typé) s'il autorise une variable à changer de type au cours de son existence.

Ainsi, notre essai ci-dessus montre que Python est faiblement typé.
Mais physiquement, comment gérer le fait qu'une variable réelle prend plus de place en mémoire qu'une variable entière ? La réponse est simple :

```
>>> x =3
>>> id(x)
211273705504
>>> x /= 2
>>> id(x)
140635669676560
```

L'emplacement de `x` a changé. En fait, toute affectation modifie l'emplacement mémoire (donc l'identifiant) d'une variable, même si elle conserve le même type.

Certaines méthodes modifient certaines variables en place (donc sans changement d'adresse) : il ne s'agit dans ce cas pas d'affectation.

II Structures de données

Il est fréquent de regrouper plusieurs valeurs dans une variable ayant une structure plus complexe capable de mémoriser plusieurs valeurs selon une hiérarchie déterminée, par exemple sous forme d'une liste ou d'un ensemble.

Définition 3.2.1 (Structure de données, attribut)
Une *structure de données* est une organisation de la mémoire en vue de stocker un ensemble de données (n'étant pas nécessairement de même type). Les données qui constituent la structure sont appelées *attributs*

Une structure de données est déterminée par la façon dont sont rangés les attributs les uns par rapport aux autres, et la façon à laquelle on accède à ces données. Ainsi, il existe plusieurs types de structures de données.
Une structure de donnée est définie par une *classe*, possédant un type (le nom de la classe, déterminant la structure en mémoire à adopter), et un ensemble de méthodes associées à la classe (des fonctions permettant de modifier d'une façon ou d'une autre un objet)

Définition 3.2.2 (Instantiation)

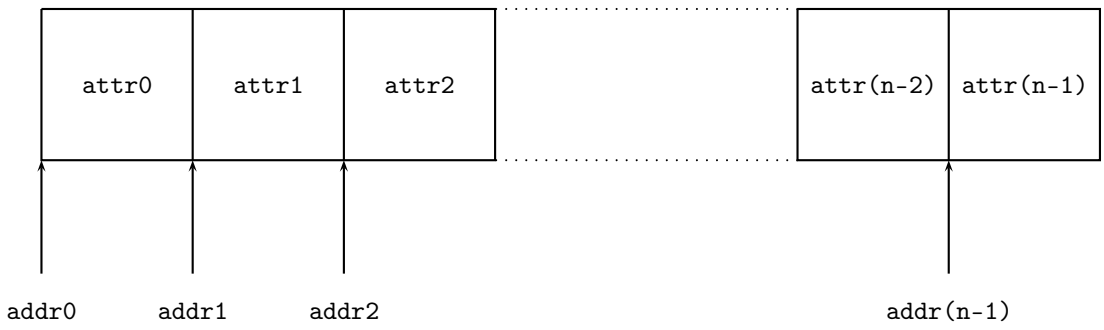
Une instantiation d'une classe est la création d'une variable dont le type est celui de la classe. Il s'agit donc de la création :
- d'un nom de variable, couplé à une adresse mémoire et un type
- d'un contenu, à l'adresse mémoire en question, dont la structure de stockage dépend de la classe.

Une des structures les plus utilisées est la structure de tableau (les listes de Python par exemple). Nous l'évoquons ci-dessous un peu plus en détail.

Définition 3.2.3 (Structure de tableau)

Un tableau (statique) informatique est une structure de données séquentielle (un rangement linéaire des attributs), dont tous les attributs sont de même type. Un tableau a une taille fixe, déterminée au moment de sa création.

En mémoire, les différents attributs d'un tableau sont stockés de façon contiguë. L'homogénéité du type des attributs permet alors de stocker chaque attribut sur un nombre déterminé d'octets. Ainsi, la place prise en mémoire ne dépend pas du type d'objet stocké, et ne risque pas de varier au cours du temps :



Les données étant rangées de façon contiguë, la connaissance de l'adresse initiale `addr0` détermine complètement l'adresse de début de l'attribut i. Plus explicitement, si les données sont d'un type nécessitant une place N de stockage, on aura une relation du type : `addr(i)= addr(0) + Ni`
Ainsi :

Proposition 3.2.4 (Complexité des opérations élémentaires sur un tableau)

(i) L'accès à un élément d'un tableau se fait en temps constant (ne dépendant ni de la taille du tableau, ni de la valeur de l'indice)

(ii) La recherche d'un élément se fait en $O(n)$, c'est-à-dire au plus en un temps proportionnel à la taille du tableau (il faut parcourir tout le tableau dans l'ordre)

(iii) L'insertion d'un élément se fait en $O(n)$ (il faut décaler tous les suivants, par réécritures)

(iv) La suppression d'un élément se fait en $O(n)$ (il faut décaler aussi)

(v) La recherche de la longueur se fait en temps constant (c'est une donnée initiale)

On peut améliorer cette structure, en autorisant des tableaux de taille variable. On parle alors de tableaux dynamiques

Définition 3.2.5 (Tableaux dynamiques)

Un tableau dynamique est une structure de données séquentielle de taille variable, pour des données ayant tous le même type.

Définition 3.2.6 (Capacité, taille)

- La capacité d'un tableau est le nombre de cases momentanément disponibles en mémoire pour le tableau
- La taille d'un tableau (ou la longueur) est le nombre de cases réellement utilisées (les autres étant considérés comme vides).

La capacité d'un tableau dynamique peut évoluer suivant les besoins ; en particulier, insérer des éléments peut faire dépasser la capacité. Dans ce cas, la capacité est doublée : il est donc créé un nouveau tableau de taille double. On ne peut pas se contenter de créer les cases manquantes, car comme pour un tableau, les cases doivent être contiguës en mémoire, et il n'est pas certain que les cases suivant les cases déjà existantes soient disponibles.
Ainsi, une augmentation de capacité nécessite en principe un changement d'adresse donc une réécriture complète (on recopie les données). C'est pour cette raison que lorsqu'on augmente la capacité, on le fait franchement (on double), quitte à avoir des cases inutilisée, afin de ne pas avoir à faire ces réécritures à chaque opération. C'est la raison de la distinction entre capacité et taille.
La capacité et la taille du tableau sont stockés en des emplacements directement accessibles. Ainsi, n désignant la taille d'un tableau, on peut étendre les résultats obtenus pour les tableaux standards :

Proposition 3.2.7 (Complexité des opérations élémentaires sur un tableau dynamique)

(i) L'accès à un élément d'un tableau dynamique se fait en temps constant (ne dépendant ni de la taille du tableau, ni de la valeur de l'indice)

(ii) La recherche d'un élément se fait en $O(n)$,

(iii) L'insertion d'un élément se fait en $O(n)$ (sauf lorsque cela induit un dépassement de capacité)

(iv) La suppression d'un élément se fait en $O(n)$

(v) La recherche de la longueur se fait en temps constant (il faut juste récupérer la donnée en mémoire).

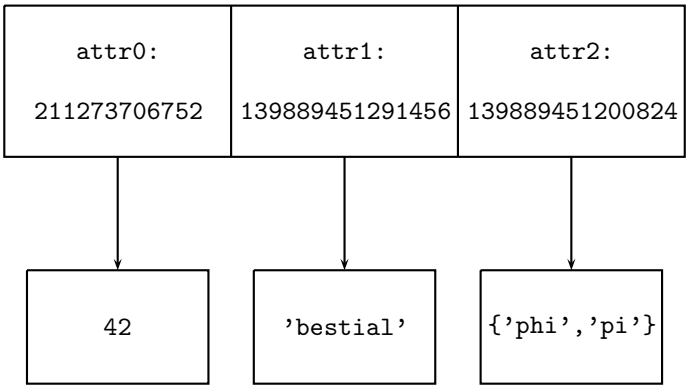
Définition 3.2.8 (Listes Python)

Une liste en Python est assimilable à une structure de tableau dynamique, dont les attributs sont des pointeurs (adresses) vers des emplacements mémoires où sont stockées les données de la liste

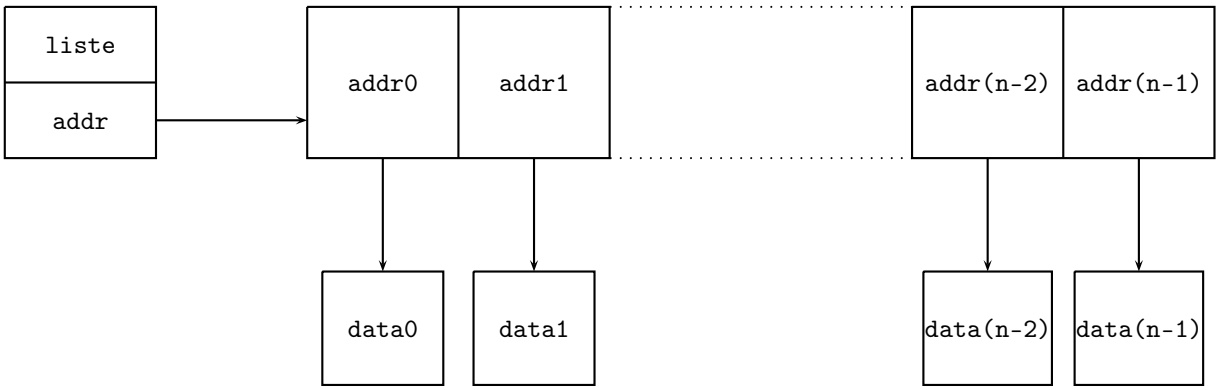
Ainsi les attributs réels d'une liste sont tous de même type (des adresses), comme nécessaire pour un tableau. Mais les objets cibles (qu'on appellera aussi attribut par abus, ou donnée) peuvent être de type quelconque, et différents les uns des autres.
Par exemple, créons la liste suivante :

```
>>> liste = [42,'bestial',{'phi','pi'}]
>>> id(liste)
139889451112064
>>> id(liste[0])
211273706752
>>> id(liste[1])
139889451291456
>>> id(liste[2])
139889451200824
>>> type(liste[0])
<class 'int'>
>>> type(liste[1])
<class 'str'>
>>> type(liste[2])
<class 'set'>
```

La structure de cette liste en mémoire sera donc :



Plus généralement, on aura une structure en mémoire du type suivant :



Remarque 3.2.9

Cela ne correspond pas à la terminologie adoptée dans beaucoup d'autres langages, ou une liste possède une structure différente (liste chaînée, ou doublement chaînée) : dans une structure de ce type, les données ne sont pas nécessairement stockées de façon contiguë. À l'emplacement de la donnée k, on trouve la donnée à mémoriser, ainsi que l'adresse de la donnée de rang $k+1$ (c'est le chaînage), et éventuellement de rang $k-1$ (en cas de double-chaînage). La donnée intiale de la liste est l'adresse

du premier terme. Ainsi, pour accéder au k-ième élément, on est obligé de parcourir tous les premiers éléments de la liste, en suivant les liens successifs : l'accès à un élément n'est plus en temps constant ; ni le calcul de la longueur qui nécessite de parcourir toute la liste. En revanche, l'insertion et la suppression se font en temps constant (une fois la position repérée).

Définition 3.2.10 (Structure d'ensemble, set)

Un ensemble informatique est une structure de données, possédant un ensemble F d'emplacements en mémoire disponible, et dans laquelle les éléments sont stockés à une certains place en fonction de leur valeur. Plus précisément, les données sont passées à la moulinette : on leur applique une fonction de hachage, dont l'image est F : la fonction renvoie le lieu de stockage.

Comme il existe une variété d'objets à représenter bien plus grande que l'ensemble F, plusieurs objets pourront être stockés à la même place (dans une structure composée) : on n'a fait que partitionner l'ensemble initial en sous-ensembles (beaucoup) plus petits
Les éléments ne sont pas numérotés donc il n'y a pas d'accès à un élément donné. La recherche d'un élément se fait cette fois en temps constant : on applique la fonction de hachage, et on vérifie qu'on trouve bien notre élément à la place obtenue (en fait presque constant, au cas où il y a plusieurs éléments) L'ajout et la suppression se font aussi en temps constant
En Python, les ensembles sont du type `set`, et représentés entre accolades : `{1,2,3}` par exemple.

III Mutabilité ; cas des listes

Définition 3.3.1 (Mutabilité)

Un type d'objets est dit *mutable* si les objets de ce type sont modifiables (sans changement d'adresse, donc pas sous forme d'une réaffectation). Il est dit *non mutable* ou *immuable* sinon.

```
>>> liste = [1,2]
>>> id(liste)
140098803414224
>>> liste[1]+=1
>>> liste
[1, 3]
>>> id(liste)
140098803414224
>>> couple = (1,2)
>>> couple[1]+=1
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'tuple' object does not support item assignment
```

L'exemple ci-dessus montre que les listes sont mutables : on peut changer leurs attributs, sans changer l'adresse de la liste elle-même. En revanche, les tuples ne semblent pas mutables.

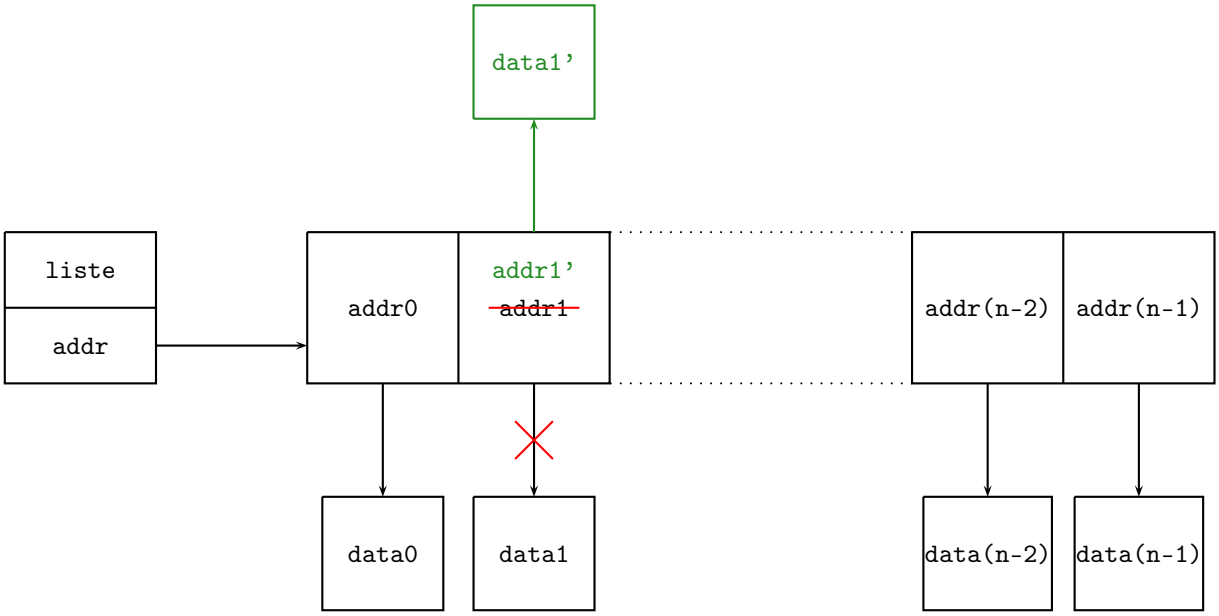
Exemple 3.3.2 (Mutabilité des principales classes en Python)

- Les `list` et les `set` sont mutables.
- Les types numériques (`int`, `float`, `boolean`, `complex`...), les `tuples`, les `str` (chaînes de caractères) ne sont pas mutables.

Voyons quel est l'effet de la modification sur les attributs réels de la liste, donc sur l'adresse des données :

```
>>> liste = [1,2]
>>> id(liste[0])
211273705440
>>> id(liste[1])
211273705472
>>> liste[1]+=1
>>> id(liste[1])
211273705504
```

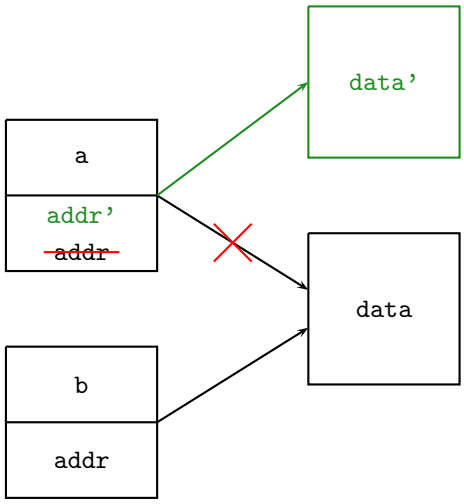
Ainsi, comme toute réaffectation sur une variable, l'opération effectuée sur la donnée modifie son adresse (en fait, les types numériques sont non mutables). Cette nouvelle adresse est stockée dans la liste.
Une affectation sur un des attributs peut donc être représenté de la façon suivante :



Intéressons-nous maintenant à la copie de listes. Pour commencer, étudions ce qu'il se passe quand on copie un entier, et qu'on modifie une des deux copies :

```
>>> a = 1
>>> b = a
>>> id(a)
211273705440
>>> id(b)
211273705440
>>> a += 1
>>> a
2
>>> b
1
>>> id(a)
211273705472
>>> id(b)
211273705440
```

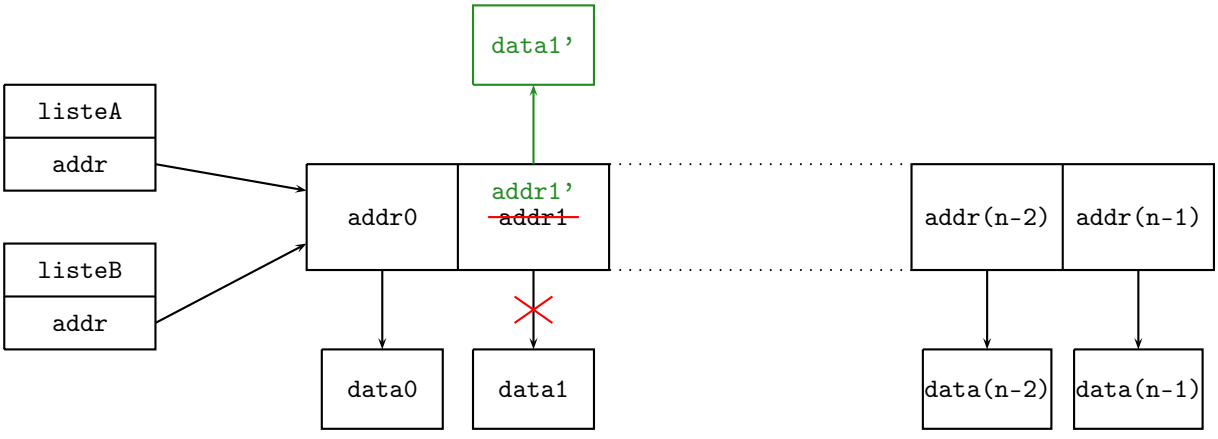
Au moment de la copie, la donnée n'a pas été recopiée : il a été créé une variable `b` pointant vers la même adresse que `a`. Ainsi, la donnée n'est stockée qu'une fois en mémoire. La modification de `a` par réaffectation se fait avec modification d'adresse, donc avec copie ailleurs du résultat. Comme `b` pointe toujours vers l'ancienne adresse, le contenu de `b` n'est pas modifié. Cela peut se représenter ainsi :



De façon générale, on a toujours indépendance entre une variable et sa copie lorsqu'on travaille avec un objet non mutable (car toute modification se fait avec changement d'adresse).
Voyons maintenant ce qu'il en est des listes. En particulier, que se passe-t-il sur une copie d'une liste lorsqu'on modifie un attribut de la liste initiale ?

```
>>> listeA=[1,2,3]
>>> listeB = listeA
>>> id(listeA)
140293216095552
>>> id(listeB)
140293216095552
>>> listeA[1]+=2
>>> listeA
[1, 4, 3]
>>> listeB
[1, 4, 3]
```

Les modifications faites sur la liste initiale sont aussi visibles sur la copie ! Ainsi, la copie n'est pas du tout indépendante de l'original. Ceci s'explique bien par le diagramme suivant :



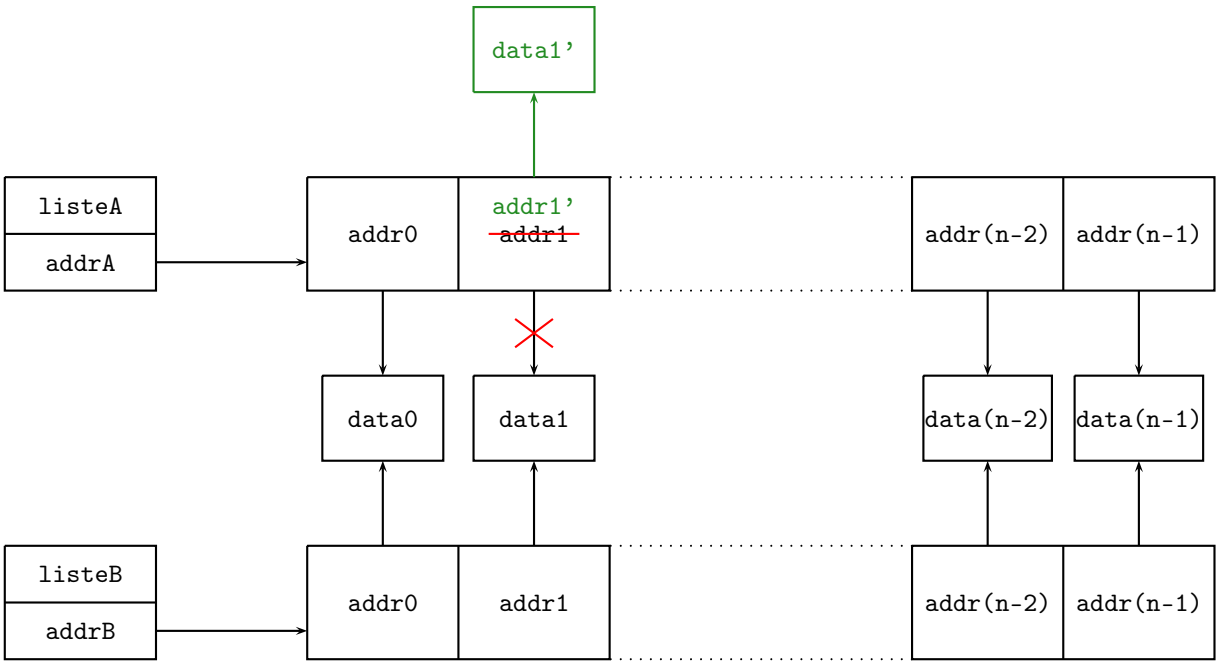
La modification n'affecte par directement l'adresse de `listeA`, mais une adresse d'un des attributs, adresse stockée en un espace mémoire vers lequel pointe également `listeB`.
Pour rendre les copies « plus » indépendantes, il faut recopier les adresses des données ailleurs. Cela peut se faire par exemple par un slicing (couper une tranche d'une liste), qui se fait par copie des adresses en un autre emplacement mémoire. En revanche, les données elles-mêmes conservent la même adresse (donc les identifiants des attributs de `listeA` et `listeB` sont les mêmes) :

```
>>> listeA = [1,2,3]
>>> listeB = listeA[:]
>>> id(listeA)
140379366079808
>>> id(listeB)
140379366057168
>>> id(listeA[0])
211273705440
>>> id(listeB[0])
211273705440
```

Modifions maintenant un attribut de `listeA`, et voyons l'effet sur `listeB` :

```
>>> listeA[0]+=3
>>> listeA
[4, 2, 3]
>>> listeB
[1, 2, 3]
>>> id(listeA[0])
211273705536
>>> id(listeB[0])
211273705440
```

Cette fois, la modification n'est pas faite sur `listeB`. Ceci s'explique aisément par le diagramme suivant :

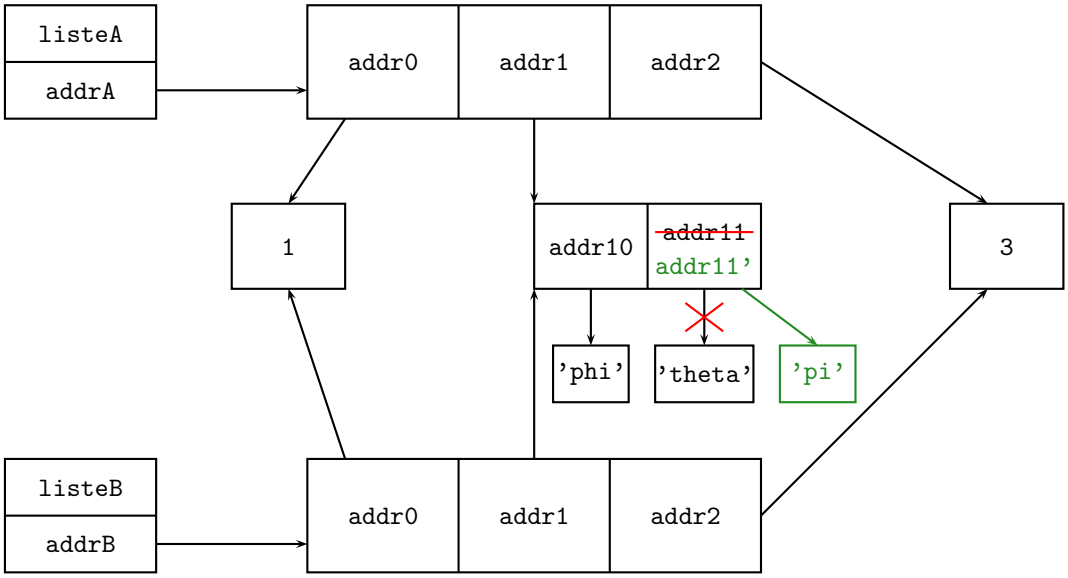


Mais cela ne règle pas entièrement le problème. En effet, si les données elles-mêmes sont mutables, elles peuvent être modifiées sans changement d'adresse. Dans ce cas, la copie pointe encore vers la même

adresse que l'objet modifié : la modification est à nouveau visible sur la copie. Nous illustrons cela dans le cas où un des attributs est une liste :

```
>>> listeA=[1,['phi','theta'],3]
>>> listeB = listeA[:]
>>> listeA[1][1] = 'pi'
>>> listeA
[1, ['phi', 'pi'], 3]
>>> listeB
[1, ['phi', 'pi'], 3]
```

Cela s'illustre par le diagramme suivant :



Pour régler complètement le problème, on peut faire une copie récursive, qui va explorer la liste en profondeur, afin de recopier à des adresses différentes les sous-listes, sous-sous-listes etc. Une copie récursive se fait avec la fonction `deepcopy` disponible dans le module `copy` :

```
>>> listeA = [1, ['phi','theta'],3]
>>> import copy
>>> listeB = copy.deepcopy(listeA)
>>> listeA[1][1] = 'pi'
>>> listeA
[1, ['phi', 'pi'], 3]
>>> listeB
[1, ['phi', 'theta'], 3]
```

Ainsi, l'original et la copie sont complètement indépendants.

4 Algorithmique élémentaire

Dans ce chapitre, nous introduisons les concepts élémentaires de l'algorithmique. Nous revoyons tout d'abord les structures élémentaires à partir desquelles sont construits tous les algorithmes écrits dans un langage impératif. Ces structures ont déjà été introduites au cours du chapitre 2 dans le cadre de Python. Nous étudions ensuite des méthodes d'étude de ces algorithmes, notamment l'étude de la terminaison (le fait que l'algorithme s'arrête, autrement dit qu'il finisse par renvoyer un résultat), de la correction (le fait que le résultat renvoyé est bien le résultat attendu), et enfin de la complexité. À la fin du chapitre, nous étudions un petit nombre d'algorithmes élémentaires qui interviennent couramment dans les programmes.

I Algorithmes

I.1 Définition

Le mot *Algorithme* vient du nom du mathématicien arabe *Al Khwarizmi*, auteur au IX-^{e} siècle d'un ouvrage faisant la synthèse de la résolution des équations du second degré, suivant le signe des coefficients (afin d'éviter l'usage du signe moins). L'ouvrage en question, proposant des méthodes de résolution par manipulations algébriques (réduction à des formes connues) a donné sont nom à l'*algèbre*. Les méthodes exposées peuvent s'apparenter à des algorithmes : on y expose, par disjonction de cas (structure conditionnelle) des façons systématiques de résoudre un certain problème, ne laissant ainsi rien au hasard. Il s'agit bien d'un algorithme de résolution.

Définition 4.1.1 (Algorithme)
Un algorithme est une succession d'instructions élémentaires, faciles à faire et non ambiguës, déterminées de façon unique par les données initiales, et fournissant la réponse à un problème posé.

Le développement de l'informatique a marqué l'essor l'algorithmique, mais cette discipline n'est pas l'apanage de l'informatique. La notion d'algorithme est liée mathématiquement à la possibilité de résolution systématique d'un problème, donc à la notion de méthode de calcul. On trouve dans le domaine purement mathématique de nombreux algorithmes :

- tous les algorithmes de calcul des opérations élémentaires (addition posée, multiplication posée...)
- l'algorithme de la division euclidenne par différences successives
- l'algorithme d'Euclide du calcul du pgcd
- l'algorithme de résolution des équations de degré 2
- l'algorithme du pivot de Gauss pour résoudre les systèmes d'équations linéaires, et répondre à d'autres questions d'algèbre linéaire.
- l'algorithme de Hörner pour l'évaluation d'un polynôme
- etc.

Les questions qu'on peut se poser sont alors les suivantes :

1. Quelles sont les structures simples élémentaires à partir desquelles sont construits les algorithmes.
2. L'algorithme s'arrête-t-il ? (problème de la terminaison)
3. L'algorithme renvoie-t-il le résultat attendu ? (problème de la correction)
4. Combien de temps dure l'exécution de l'algorithme, notamment lorsqu'on le lance sur de grandes données ? (problème de la complexité).

I.2 Le langage

L'étude algorithmique formelle nécessite de se dégager de toute contrainte idiomatique relevant des spécificités de tel ou tel langage. Pour cela, nous utiliserons un pseudo-code, indépendant de toute implémentation. Les traductions ultérieures dans un langage de programmation spécifique se font alors sans difficulté, sachant que certains langages offrent parfois certaines possibilités supplémentaires (qui sont essentiellement du confort, mais n'ajoutent rien à la description formelle des algorithmes, comme par exemple le fait de pouvoir construire une boucle `for` sur un objet itérable queconque en Python).

Nous décrirons systématiquement un algorithme en :

1. lui donnant un nom ;
2. définissant les données initiales (variables d'entrée, préciser leur type)
3. définissant la sortie (variables de résultat, préciser leur type)
4. donnant le bloc d'instructions définissant l'algorithme.

La structure générale est donc la suivante :

Algorithme 4.1 : Nom ou description de l'algorithme

Entrée : a,b,... : type
Sortie : c,d,... : type

instructions ;
renvoyer (c, d,...)

La dernière ligne, permet de définir le ou les résultats.

I.3 Les structures élémentaires

Un algorithme repose sur certains types d'instructions élémentaires. Une instruction est une phrase du langage de programmation indiquant à l'ordinateur une ou plusieurs actions à effectuer, induisant un changement d'état de l'ordinateur (c'est-à-dire une modification de la mémoire).
Les différentes structures élémentaires à partir desquelles sont contruits les algorithmes en programmation impérative sont, en plus de l'évalutation d'expression, de l'utilisation de fonctions antérieures, et de l'affectation (notée $x \leftarrow$ valeur) :

1. **La séquence :**
   Il s'agit d'un regroupement d'instructions élémentaires, qui seront exécutées successivement. Il s'agit de la notion de *bloc* en informatique, délimité suivant les langages par des balises de début et fin (`begin`, `end`), ou tout simplement par l'indentation, comme en Python.
   L'intérêt de la séquence est de pouvoir considérer plusieurs instructions comme une seule, et de pouvoir inclure cette succession dans des structures plus complexes (structures conditionnelles, répétitives...)
   Nous écrirons un bloc de la façon suivante :
   La plupart du temps, un bloc permettra de délimiter la portée d'une structure composée. Dans ce cas, le mot `bloc` sera remplacé par le mot clé de la structure correspondante, par exemple `début pour` et `fin pour`

Algorithme 4.2 : Bloc

début bloc
| instructions
fin bloc

2. **La structure conditionnelle simple :**

   C'est la structure permettant d'effectuer des disjonctions de cas. Elle permet de distinguer plusieurs cas si ces différents cas nécessitent des modes opératoires différents. Cela permet également de traiter les cas particuliers.

   La structure basique est un branchement à deux issues :

   `si` *condition* `alors` *instructions* `sinon` *instructions*

   La condition peut être n'importe quel booléen, par exemple le résultat d'un test, ou le contenu d'une variable de type booléen. De plus, la clause alternative (`else`) est le plus souvent facultative (algorithmes 3 et 4)

Algorithme 4.3 : Structure conditionnelle simple sans clause alternative

Entrée : classe : entier
Sortie : ∅

si *classe = 4* **alors**
| `Afficher('Bestial!')`
fin si

Algorithme 4.4 : Structure conditionnelle simple avec clause alternative

Entrée : classe : entier
Sortie : ∅

si *classe = 4* **alors**
| `Afficher('Bestial!')`
sinon
| `Afficher('Khrass')`
fin si

Certains langages autorisent le branchement multiple, soit *via* une autre instruction (par exemple `case of` en Pascal), soit comme surcouche de l'instruction basique. Nous utiliserons cette possibilité en pseudo-code (algorithme 5).

Algorithme 4.5 : Structure conditionnelle multiple

Entrée : classe : entier
Sortie : ∅

si *classe = 4* **alors**
| `Afficher('Bestial!')`
sinon si *classe = 3* **alors**
| `Afficher('Skiii!')`
sinon
| `Afficher('Khrass')`
fin si

Évidemment, d'un point de vue de la complétion du langage, cette possibilité n'apporte rien de neuf, puisqu'elle est équivalente à un emboîtement de structures conditionnelles (algorithme 6)

Algorithme 4.6 : Version équivalente

Entrée : classe : entier
Sortie : $\varnothing$

si *classe = 4* **alors**
| `Afficher('Bestial!')`
sinon
| **si** *classe = 3* **alors**
| | `Afficher('Skiii!')`
| **sinon**
| | `Afficher('Khrass')`
| **fin si**
fin si

3. **Les boucles**

Une boucle est une succession d'instructions, répétée un certain nombre de fois. Le nombre de passages dans la boucle peut être déterminé à l'avance, ou peut dépendre d'une condition vérifiée en cours d'exécution. Cela permet de distinguer plusieurs types de boucles :

(a) **Boucles conditionnelles, avec condition de continuation.**

On passe dans la boucle tant qu'une certain condition est réalisée. Le test de la condition est réalisé avant le passage dans la boucle. On utilise pour cela une boucle `while` ou `tant que` en français. Voici un exemple :

Algorithme 4.7 : Que fait cet algorithme ?

Entrée : ε (marge d'erreur) : réel
Sortie : $\varnothing$

$u \leftarrow 1$;
tant que $u > \varepsilon$ **faire**
| $u \leftarrow \sin(u)$
fin tant que

Ici, on calcule les termes d'une suite définie par la récurrence $u_{n+1} = \sin(u_n)$. On peut montrer facilement que cette suite tend vers 0. On répète l'itération de la suite (donc le calcul des termes successifs) tant que les valeurs restent supérieures à ε.

Comme dans l'exemple ci-dessus, une boucle `while` s'utilise le plus souvent lorsqu'on ne connaît pas à l'avance le nombre de passages dans la boucle. Souvent d'ailleurs, c'est le nombre de passages dans la boucle qui nous intéresse (afin, dans l'exemple ci-dessus, d'estimer la vitesse de convergence de la suite). Dans ce cas, il faut rajouter un compteur de passages dans la boucle, c'est-à-dire une variable qui s'incrémente à chaque passage dans la boucle :

La valeur finale de i nous dit maintenant jusqu'à quel rang de la suite il faut aller pour obtenir la première valeur inférieure à ε (et par décroissance, facile à montrer, toutes les suivantes vérifieront la même inégalité).

(b) **Boucles conditionnelles, avec condition d'arrêt**

Il s'agit essentiellement de la même chose, mais exprimé de façon légèrement différente. Ici, on répète la série d'instructions jusqu'à la réalisation d'une certaine condition. Il s'agit de la boucle `repeat... until...`, ou `repéter... jusqu'à ce que...`, en français. L'algorithme précédent peut se réécrire de la façon suivante, de façon quasi-équivalente :

La différence essentielle avec une boucle `while` est que, contrairement à une boucle `while`, on passe nécessairement au moins une fois dans la boucle. À part ce détail, on a équivalence entre

Algorithme 4.8 : Calcul de i tel que $u_i \leqslant \varepsilon$

Entrée : ε (marge d'erreur) : réel
Sortie : i : entier

$u \leftarrow 1$;
$i \leftarrow 0$;
tant que $u > \varepsilon$ **faire**
  $u \leftarrow \sin(u)$;
  $i \leftarrow i + 1$
fin tant que
renvoyer i

Algorithme 4.9 : Vitesse de convergence de u_n

Entrée : ε (marge d'erreur) : réel
Sortie : i (rang tel que $u_i \leqslant \varepsilon$) : entier

$u \leftarrow 0$;
$i \leftarrow 0$;
repéter
  $u \leftarrow \sin(u)$;
  $i \leftarrow i + 1$
jusqu'à ce que $u \leqslant \varepsilon$;
renvoyer i;

les deux structures, en remplaçant la condition de continuation par une condition d'arrêt (par négation). Ainsi, une boucle `repeat instructions until condition` est équivalente à :

Algorithme 4.10 : Structure équivalente à `repeat... until...` : version 1

instructions ;
tant que $\neg$ *condition* **faire**
  instructions
fin tant que

Remarquez ici le passage forcé une première fois dans la succession d'instructions (bloc isolé avant la structure).

On peut éviter la répétition de la succession d'instructions en forçant le premier passage à l'aide d'une variable booléenne :

Algorithme 4.11 : Structure équivalente à `repeat... until...` : version 2

$b \leftarrow$ True ;
tant que *(*$\neg$ *condition)* $\vee$ b **faire**
  instructions ;
  $b \leftarrow$ False
fin tant que

Cela a cependant l'inconvénient d'augmenter le nombre d'affectations.

Réciproquement, une boucle `while condition do instructions` est équivalente à la structure de l'algorithme 4.12.

Au vu de ces équivalences, même si en Python, les boucles `repeat` n'existent pas, nous nous autoriseront à décrire les algorithmes en utilisant cette structure, plus naturelle dans certaines

Algorithme 4.12 : Structure équivalente à `while... do...`

si *condition* **alors**
  repéter
    instructions
  jusqu'à ce que ¬ *condition*;
fin si

situations. Il faut cependant garder dans l'esprit que dans le cadre d'une définition formelle d'un algorithme, cela crée une redondance avec la structure `while`.

(c) **Boucles inconditionnelles**

Il s'agit de boucles dont l'arrêt ne va pas dépendre d'une condition d'arrêt testée à chaque itération. Dans cette structure, on connaît par avance le nombre de passages dans la boucle. Ainsi, on compte le nombre de passages, et on s'arrête au bout du nombre souhaité de passages. Le compteur est donné par une variable incrémentée automatiquement :

Algorithme 4.13 : Cri de guerre

Entrée : ∅
Sortie : cri : chaîne de caractères

cri ← 'besti' ;
pour $i \leftarrow 1$ **à** 42 **faire**
  cri ← cri + 'â'
fin pour
cri ← cri + 'l' ;
renvoyer *cri*

I.4 Procédures, fonctions et récursivité

Un algorithme peut faire appel à des sous-algorithmes, ou procédures, qui sont des morceaux isolés de programme. L'intérêt est multiple :

- Éviter d'avoir à écrire plusieurs fois la même séquence d'instructions, si elle est utilisée à différents endroits d'un algorithme. On peut même utiliser une même procédure dans différents algorithmes principaux.
- Une procédure peut dépendre de paramètres. Cela permet d'adapter une séquence donnée à des situations similaires sans être totalement semblables, les différences entre les situations étant traduites par des valeurs différentes de certains paramètres.
- Écrire des procédures permet de sortir la partie purement technique de l'algorithme principal, de sorte à dégager la structure algorithmique de ce dernier de tout encombrement. Cela augmente la lisibilité de l'algorithme.
- Une procédure peut s'appeler elle-même avec des valeurs différentes des paramètres (récursivité). Cela permet de traduire au plus près certaines définitions mathématiques par récurrence. Cela a un côté pratique, mais assez dangereux du point de vue de la complexité si on n'est pas conscient précisément de ce qu'implique ce qu'on écrit.

La procédure **??** est un exemple typique de procédure : il s'agit de l'affichage d'un polynôme entré sous forme d'un tableau. Créer une procédure pour cela permet de pouvoir facilement afficher des polynômes à plusieurs reprises lors d'un algorithme portant sur les polynômes, ceci sans avoir à se préoccuper, ni s'encombrer de la technique se cachant derrière. Dans cette procédure `str` est une fonction convertissant une valeur numérique en chaîne de caractères.

Une fonction est similaire à une procédure, mais renvoie en plus une valeur de sortie. En général, il est conseillé de faire en sorte que la seule action d'une fonction soit ce retour d'une valeur. En particulier, on

Procédure 4.14 : affichepolynome(T)

Entrée : T : tableau représentant un polynôme
Sortie : ∅

```
ch ← '' ;
pour i ← Taille (T)-1 descendant à 0 faire
    si T[i] ≠ 0 alors
        si (T[i] > 0) alors
            si ch ≠ '' alors
                ch ← ch + ' + '
            fin si
        sinon
            ch ← ch + ' - '
        fin si
        si (|T[i]| ≠ 1) ∨ (i = 0) alors
            ch ← ch + str(|T[i]|);
            si i ≠ 0 alors
                ch ← ch + ' * '
            fin si
        fin si
        si i ≠ 0 alors
            ch ← ch + 'X'
        fin si
        si i > 1 alors
            ch ← ch + '^' + str(i)
        fin si
    fin si
fin pour
si ch = '' alors
    ch ← '0'
fin si
Afficher (ch)
```

ne fait aucune interface avec l'utilisateur : pas d'affichage, ni de lecture de valeur (les valeurs à utiliser pour la fonction étant alors passées en paramètres).
La fonction 15 est une fonction simple déterminant le nombre de lettres a dans une chaîne de caractères :
La fonction 16 est une fonction récursive pour le calcul la suite définie par une récurrence simple, en l'occurrence $u_{n+1} = \sin(u_n)$, initialisée par 1.
La fonction 17 est une très mauvaise façon de calculer le n-ième terme de la suite de Fibonacci par récursivité. En pratique, vous calculerez plus vite à la main F_{100} que l'ordinateur par la fonction ci-dessus. Pourquoi cet algorithme récursif est-il si mauvais ?

L'étude de la récursivité est au programme de Spé (ou de Sup en cours d'option). Nous nous contenterons donc cette année d'une utilisation naïve de la récursivité, tout en étant conscient des dangers d'explosion de complexité que cela peut amener.

II Validité d'un algorithme

Deux questions peuvent se poser pour juger de la validité d'un algorithme :
- L'algorithme s'arrête-t-il ? (problème de terminaison)
- L'algorithme renvoie-t-il le résultat attendu ? (problème de correction).

Fonction 4.15 : comptea(ch)

Entrée : ch : chaîne de caractères
Sortie : n : entier

$n \leftarrow 0$;
pour $i \leftarrow 0$ **à** `Taille(`ch`)` **faire**
  si ch[i] = *'a'* **alors**
    $n \leftarrow n + 1$
  fin si
fin pour
renvoyer n

Fonction 4.16 : un(n)

Entrée : n : entier positif
Sortie : un(n) : réel

si $n = 0$ **alors**
  renvoyer 1
sinon
  renvoyer $\sin(\text{un}(n-1))$
fin si

Nous étudions ces deux points dans ce paragraphe. Nous donnons notamment des moyens de justifier la terminaison et la correction d'un algorithme.

II.1 Terminaison d'un algorithme

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` et `repeat`. Nous n'évoquerons pas dans ce cours la terminaison et la correction d'un algorithme récursif (étude se ramenant le plus souvent à une récurrence).
Pour l'étude de la terminaison, partons de l'exemple de l'algorithme d'Euclide (algorithme 18).
La preuve mathématique classique de la terminaison de l'algorithme d'Euclide passe par le principe de descente infinie : les restes successifs (c'est-à-dire les valeurs successives de b) forment une suite strictement décroissante d'entiers positifs (sauf éventuellement à la première étape si la valeur initiale de b est négative ; mais dans ce cas, le premier reste est positif). Le principe de descente infinie assure alors que la suite est finie, donc que l'algorithme finit par s'arrêter.

Cet exemple est l'archétype d'une preuve de terminaison. On prouve la terminaison d'une boucle en exhibant un variant de boucle :

Définition 4.2.1 (Variant de boucle)

On appelle variant de boucle une quantité v définie en fonction des variables $(x_1, \ldots, x_k)$ 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 :

Fonction 4.17 : fibo(n)

Entrée : n : entier positif
Sortie : fibo(n) : réel

si $n = 0$ **alors**
| **renvoyer** 0
sinon si $n = 1$ **alors**
| **renvoyer** 1
sinon
| **renvoyer** $\text{fibo}(n-1) + \text{fibo}(n-2)$
fin si

Algorithme 4.18 : Euclide

Entrée : a, b : entiers positifs
Sortie : d : entier

tant que $b > 0$ **faire**
| $(a, b) \leftarrow (b, a \mod b)$
fin tant que
renvoyer a

Théorème 4.2.2 (Terminaison d'une boucle, d'un algorithme)

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.*

Avertissement 4.2.3

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

Exemple 4.2.4

Dans le cas de l'algorithme d'Euclide, un variant de boucle simple est donné simplement par la variable b (ou $|b|$ si on veut assurer la positivité et la décroissance dès le rang initial). Constatez que la validité de ce variant nécessite une discussion sur le signe de b, nécessaire pour considérer toutes les valeurs d'entrée possibles.

Remarque 4.2.5

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 : c'est le cas par exemple du calcul récursif de la suite de Fibonacci donné ci-dessus, pour des valeurs de n restant raisonnables). Évidemment dans ce cas, même si l'algorithme s'arrête en théorie, c'est d'assez peu d'intérêt.
Nous étudirons ces problèmes liés à la complexité d'un algorithme dans le chapitre prochain.

Remarque 4.2.6

Dans le cas d'une boucle `for`, on peut toujours construire un variant simple. Si la boucle est donnée

par la structure :
`Pour` $i \leftarrow a$ `à` b,
un variant simple est $b - i$

Évidemment, ceci ne fait que traduire un fait intuitivement évident : le nombre de passages dans une boucle `for` est évidemment fini, égal au nombre de valeurs que le compteur de boucle peut prendre. Dans la pratique, cela nous dispense d'un argument de terminaison pour ce type de boucles.

En guise d'exemple développé, nous traitons l'étude de la terminaison de l'algorithme suivant :

Algorithme 4.19 : Mystère

Entrée : a,b : entiers
Sortie : q, r : entiers

$r \leftarrow a$;
$q \leftarrow 0$;
si $b = 0$ **alors**
  Erreur
sinon
  tant que $r \geqslant b$ **faire**
    $r \leftarrow r - b$;
    $q \leftarrow q + 1$;
  fin tant que
  renvoyer q,r
fin si

Le variant de boucle est à contruire à partir du test de continuation de la boucle. Puisqu'on s'arrête dès que $r - b < 0$, que $r - b$ pred des valeurs entières, on peut considérer la quantité $v = r - b$.

- Toutes les opérations étant entières, v est bien un entier
- Par définition même d'une boucle `while`, à l'entrée dans une boucle, la valeur de v est positive
- Il reste à étudier la décroissance stricte. Si on note v_n la valeur de v à l'entrée dans la n-ième boucle, et si on passe effectivement dans les boucles d'indice n et $n + 1$, alors $v_{n+1} = v_n - b$.

À ce stade, on se rend compte qu'il y a un problème : v est bien un variant de boucle pour une valeur initiale de b strictement positive, mais pas pour une valeur négative. Un petit examen de la situation nous fait comprendre qu'effectivement, si b est négatif, les valeurs de r seront de plus en plus grandes, donc le test $r \geqslant b$ sera « de plus en plus vrai ». Il y a peu de chance que l'algorithme s'arrête.

Il est donc nécessaire de rectifier notre algorithme, soit en excluant les valeurs négatives en même temps que la valeur nulle, soit en adaptant l'algorithme. À ce stade, vous aurez bien sûr compris ce que calcule l'algorithme `Mystère` : il ne s'agit de rien de plus que le quotient et le reste de la division euclidienne. Or, ces quantités sont aussi définies pour b strictement négatif, avec la petite adaptation suivant : le quotient q et le reste r de la division euclidenne de a par b sont les uniques entiers tels que

$$a = bq + r \qquad \text{où} \qquad r \in [\![0, |b| - 1]\!].$$

On a fait cette adaptation dans l'algorithme 20.

Revenons à notre preuve de terminaison. Maintenant la quantité $v = r - |b|$ est bien entière, positive en entrée de boucle (si on entre dans une boucle, dans le cas contraire, la terminaison est assurée !), et strictement décroissante d'un passage à l'autre dans la boucle (puisqu'on lui retire la quantité strictement positive $|b|$ à chaque étape, le cas $b = 0$ ayant été écarté initialement).

Ceci prouve la terminaison de l'algorithme 20

Algorithme 4.20 : Division euclidienne ?

Entrée : a,b : entiers
Sortie : q, r : entiers

$r \leftarrow a$;
$q \leftarrow 0$;
si $b = 0$ **alors**
  Erreur
sinon
  tant que $r \geqslant |b|$ **faire**
    $r \leftarrow r - |b|$;
    $q \leftarrow q + 1$;
  fin tant que
  si $b < 0$ **alors**
    $q \leftarrow -q$
  fin si
  renvoyer q,r
fin si

II.2 Correction d'un algorithme

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 18. 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 :

Définition 4.2.7 (Invariant de boucle)

On appelle 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.

Exemple 4.2.8

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.

Étudions encore une fois le cas de l'algorithme de la division euclidenne 20. On sait que le quotient et le

reste doivent satisfaire à la relation $a = bq+r$. Il semble donc naturel de considérer $w = \varepsilon bq+r = |b|q+r$ comme invariant de boucle, où ε est le signe de b (cela permet de gérer le changement final de signe effectué sur q à la fin de l'algorithme). On note w_k la valeur de w à l'entrée dans l'itération d'indice k, la boucle initiale étant d'indice 0. On note de même q_k et r_k les valeurs de q et r à l'entrée dans l'itération k.

- D'après les initialisations de q et r, on a $w_0 = a$
- Si le passage dans l'itération k est effectué, on ajoute 1 à q et on retranche $|b|$ à r. Ainsi :

$$w_{k+1} = |b|q_{k+1} + r_{k+1} = |b|(q_k + 1) + r_k - |b| = |b|q_k + r_k = w_k.$$

  Ainsi, (w) est bien constant.
- À la sortie de structure, en appelant n l'indice de la première boucle non exécutée, on a $r_n < |b|$ et donc :

$$w_n = q_n|b| + r_n = qb + r, \qquad \text{avec } r < |b|$$

  où q et r sont les valeurs retournées (avec changement de signe éventuel). On n'a pas encore tout à fait répondu au problème, puisqu'on doit aussi s'assurer que $r \geqslant 0$. Cette dernière propriété peut être obtenue facilement si on peut justifier que $r_{n-1} \geqslant |b|$. Dans ce cas, $r_n = r_{n-1} - |b|$. On peut aussi l'obtenir facilement lorsque $a \in [\![0, |b| - 1]\!]$ (dans ce cas, il n'y a pas de passage dans la boucle, et $n = 0$). En revanche, si r peut se retrouver à avoir des valeurs négatives, on reste coincé. Or, c'est ce qu'il se passe si la valeur initiale de r est négative, donc si $a \leqslant 0$.

La recherche de l'invariant de boucle nous a donc permis de détecter l'erreur faite dans l'algorithme 20 : celui n'est valable que pour les valeurs positives de a. Nous pouvons donc maintenant donner une version corrigée de l'algorithme de la division euclidienne (algorithme 21)

On pourrait tout de même regrouper les deux cas en combinant les deux tests (en se rendant compte que dans les deux cas, on tombera forcément dans le « trou »), et en gérant le changement de signe par un ε égal à 1 et à -1, mais, si cela racourcit le code, cela augmente en revanche le nombre d'opérations et de tests. Nous en resterons donc à la version de l'algorithme 21.

Nous reprenons donc avec ce nouvel algorithme l'étude de la terminaison et de la correction correction. Nous avons maintenant deux boucles à étudier.

1. Étude de la boucle du cas $a \geqslant 0$:
   - L'étude de la terminaison de cette boucle a déjà été faite.
   - Nous terminons l'étude de correction précédente en constatant que si $a \in [\![0, |b| - 1]\!]$, il n'y a pas de passage dans la boucle, et la valeur retournée est $q = 0$ et $r = a \in [\![0, |b| - 1]\!]$ qui répond bien au problème, et que si $a \geqslant |b|$, il y a au moins un passage dans la boucle. Ainsi, avec les notations précédentes, $n \geqslant 1$, et $r_{n-1} \geqslant |b|$ (sinon on ne serait pas passé dans la boule $n-1$, d'où $r_n = r_{n-1} - |b| \geqslant 0$. Cela fournit donc le point qui manquait précédemment pour conclure que (w) est un invariant de boucle
2. Étude de la boucle du cas $a < 0$.
   - Cette fois, $-r$ est clairement un variant de boucle assurant la terminaison (il est entier, positif tant qu'on entre dans la boucle, et décroissant puisqu'on ajoute à r une valeur strictement positive à chaque étape)
   - Nous prouvons de même que cei-dessus que $w = |b|q + r$ est un invariant pour la boucle dans le cas $a < 0$ (chaque fois qu'on ajoute $|b|$ à r, on retranche 1 à q par compensation). À la sortie multiplie q par le signe de b, donc on a la relation

$$a = |b|q_0 + r_0 = |b|q_n + r_n = bq + r,$$

     et de plus, vu le test initial, on passe au moins une fois dans la boucle, et $r_{n-1} < 0$ (sinon on ne serait pas passé dans la boucle $n - 1$, donc $r_n < |b|$. Comme on ne passe pas dans la boucle n, on a de plus $r_n \geqslant 0$. Ainsi, les valeurs finales de q et r, vérifient bien les propriétés attendues caractérisant la division, ce qui assure la correction de l'algorithme. euclidienne.

Algorithme 4.21 : Division euclidienne, version corrigée

Entrée : a,b : entiers
Sortie : q, r : entiers

$r \leftarrow a$;
$q \leftarrow 0$;
si $b = 0$ **alors**
  Erreur
sinon
  si $a \geqslant 0$ **alors**
    tant que $r \geqslant |b|$ **faire**
      $r \leftarrow r - |b|$;
      $q \leftarrow q + 1$;
    fin tant que
    si $b < 0$ **alors**
      $q \leftarrow -q$
    fin si
    renvoyer q,r
  sinon
    tant que $r < 0$ **faire**
      $r \leftarrow r + |b|$;
      $q \leftarrow q - 1$;
    fin tant que
    si $b < 0$ **alors**
      $q \leftarrow -q$
    fin si
    renvoyer q,r
  fin si
fin si

Retenons de cette étude que non seulement, l'étude de la terminaison et de la correction d'un algorithme permet de prouver sa validité, mais que par ailleurs, cette étude permet de détecter des erreurs dans l'algorithme le cas échéant.

Remarque 4.2.9

Un invariant de boucle peut très bien être un booléen, c'est-à-dire la valeur de vérité d'une propriété. Ainsi, on peut définir comme invariant de boucle, le fait qu'une propriété $\mathcal{P}(n)$ dépendant du rang n de passage dans la boucle soit toujours vraie à l'entrée de boucle.

Nous illustrons cette possibilité sur l'exemple suivant (algorithme 22).

Algorithme 4.22 : Selection du minimum

Entrée : T : tableau
Sortie : T :tableau

pour $i \leftarrow 1$ **à** `Taille` *(T) - 1* **faire**
  si $T[i] < T[0]$ **alors**
    $T[0], T[i] \leftarrow T[i], T[0]$
  fin si
fin pour

La terminaison de cet algorithme ne pose pas de problème (il n'y a pas de boucle `repeat` ou `while`).

Pour la correction, on se convainc sans peine que l'algorithme va rechercher la valeur minimale (ou une des valeurs minimales) du tableau T et la placer en tête de tableau, en modifiant éventuellement l'ordre des autres éléments. On le prouve en considérant l'invariant de boucle $\mathcal{P}(n)$: « à l'entrée dans la boucle de rang $n \geqslant 1$, $T[0]$ est le minimum des $T[i]$ pour $i \in [\![0, n-1]\!]$ ».
L'invariant $\mathcal{P}(n)$ est vrai pour $n = 1$ (initialement, $T[0]$ est le minimum du sous-tableau constitué de l'unique case $T[0]$), et si $\mathcal{P}(n)$ est vraie, alors $\mathcal{P}(n)$ aussi, car à l'entrée dans la boucle n, soit $T[0]$ est le minimum de $T[i]$, $i \in [\![0, n]\!]$, et il le reste alors à la sortie (pas de modification faite), donc à l'entrée dans la boucle suivante ; soit $T[0]$ n'est pas le minimum des $T[i]$, $i \in [\![0, n]\!]$, mais comme il est le minimum des $T[i]$, $i \in [\![0, n-1]\!]$, on a alors $T[n] < T[0]$, et c'est $T[n]$ le minimum reherché. On fait alors l'échange de $T[0]$ et $T[n]$, ce qui place à l'issue de la boucle le minimum en place 0 du tableau, assurant que $\mathcal{P}(n+1)$ est vérifié.
Ainsi, $\mathcal{P}(n)$ est toujours vrai, par principe de récurrence, et pour la dernière valeur de n (taille t du tableau, c'est la valeur d'entrée dans l'itération suivante, celle qui n'a pas lieu), on récupère que $T[0]$ est le minimum des $T[i]$, pour $i \in [\![0, t-1]\!]$, ce qui prouve la correction de l'algorithme.

Même si le fait d'utiliser un « invariant » qui ait l'air plus compliqué que précédemment (une proposition), il ne s'agit que d'un cas particulier de la définition, l'invariant étant ici tout simplement le booléen $\mathcal{P}(n)$.
Dans cette situation, montrer la correction d'un algorithme se fait en 3 étapes :

- initialisation : montrer que l'invariant est vrai avant la première itération.
- conservation : montrer que si l'invariant est vrai avant une itération, il reste vrai avant l'itération suivante.
- terminaison : déduire de l'invariant final la propriété voulue.

III Exercices

Exercice 1 (recherche linéaire)

Écrire un algorithme en pseudo-code prenant en entrée une liste L, et une valeur v, et donnant en sortie un indice i tel que $L[i] = v$, s'il en existe, et la valeur spéciale `NIL` sinon. Étudier la correction et la terminaison de cet algorithme.

Exercice 2 (Tri à bulle)

Étudier la correction et la terminaison de l'algorithme 23 : Proposer une amélioration repérant lors du passage dans la boucle interne, le plus grand indice pour lequel une inversion a été faite. Jusqu'où suffit-il d'aller pour la boucle suivante ? Étudier la correction et la terminaison du nouvel algorithme obtenu.

Algorithme 4.23 : Tri-bulle

```
Entrée : T :tableau de réels, n :taille du tableau
Sortie : T : tableau trié
pour i ← 1 à n − 1 faire
    pour j ← 1 à n − i faire
        si T[j] < T[j − 1] alors
            T[j − 1], T[j] ← T[j], T[j − 1]
        fin si
    fin pour
fin pour
```

Exercice 3 (Tri par sélection)

Donner le pseudo-code de l'algorithme de tri par sélection consistant à rechercher le minimum et à le mettre à sa place, puis à rechercher le deuxième élément et à le mettre à sa place etc. Étudier la terminaison et la correction de cet algorithme.

Exercice 4 (Exponentiation rapide)

Comprendre ce que fait l'algorithme suivant. Étudier sa terminaison et sa correction. Écrire un algorithme récursif d'exponentiation rapide.

Algorithme 4.24 : Exponentiation rapide

Entrée : x : réel, n :entier
$u \leftarrow 1$;
$v \leftarrow 1$;
tant que $n > 1$ **faire**
  $(n, b) \leftarrow \texttt{divmod}(n, 2)$;
  si $b = 1$ **alors**
    $v \leftarrow v \times u$
  fin si
  $u \leftarrow u^2$
fin tant que
renvoyer $u \times v$

Exercice 5 (Décomposition)

Que fait l'algorithme 25 ? Justifier sa terminaison et sa correction.

Algorithme 4.25 : Décomposition

Entrée : n : entier positif
Sortie : L : liste de couples d'entier ; à quoi correspond-elle ?
$p \leftarrow 2$;
$L \leftarrow []$;
tant que $p \leqslant \sqrt{n}$ **faire**
  $i \leftarrow 0$;
  tant que $n \mod p = 0$ **faire**
    $i \leftarrow i + 1$;
    $n \leftarrow n/p$
  fin tant que
  si $i > 0$ **alors**
    Ajouter (p, i) à la liste L
  fin si
  si $p = 2$ **alors**
    $p \leftarrow p + 1$
  sinon
    $p \leftarrow p + 2$
  fin si
fin tant que
si $n > 1$ **alors**
  Ajouter $(n, 1)$ à la liste L
fin si

5 Complexité

I Introduction

La terminaison et la correction d'un algorithme ne suffisent pas à juger de l'utilisabilité d'un algorithme. En effet, un algorithme qui se finit en théorie, mais qui en pratique a un temps de réponse de plusieurs années, voire plusieurs milliards d'années, est assez inutilisable en pratique. Il est donc important de pouvoir estimer le temps de réponse d'un algorithme, afin de juger de sa qualité. La recherche d'algorithmes les plus rapides possibles dans un contexte donné est un des enjeux majeurs de l'informatique.

En général, un algorithme donné est d'autant plus lent que les données initiales qui lui sont fournies sont grandes : un tableau de quelques milliards d'entrée est plus long à trier qu'un tableau de 10 entrées ! Nous estimerons par conséquent la complexité d'une algorithme comme fonction de la taille des données initiales.

En général, les qualités d'algorithmes répondant à une même question sont comparées pour des grandes tailles de données (il s'agit en fait d'une comparaison asymptotique, pour des données de taille infiniment grande). Ainsi, on dira que certains algorithmes de tris parmi les plus rapides (le tri rapide, le tri fusion...) sont quasi-linéaires (c'est-à-dire ont un comportement moyen de l'ordre de $n\ln(n)$ pour un tableau de taille n), alors que d'autres tris, comme le tri par insertion, ont une complexité quadratique (c'est-à-dire de l'ordre de n^2. Les premiers sont donc bien meilleurs que les seconds, lorsque n devient grand.

Cependant, un algorithme moins bon asymptotiquement peut être meilleur pour des petites valeurs de n, ce qui, suivant l'utilisation qu'on veut en faire, peut se révéler intéressant. Ainsi, le tri par insertion est meilleur que le tri rapide pour des tableaux de petite taille.

Par conséquent, l'étude asymptotique de la complexité n'est pas toujours suffisante ni pertinente. Tout dépend de l'utilisation qu'on veut faire des algorithmes. La première chose à faire est donc de bien cibler les besoins et les conditions d'utilisation. Suivant la taille des données à traiter, on n'a pas toujours intérêt à considérer l'algorithme réputé le meilleur : il est peut-être le meilleur dans des situations différentes de celle qu'on a à traiter.

L'étude comparative de la rapidité pour des données de petite taille permet d'ailleurs aussi parfois d'augmenter considérablement la complexité asymptotique de certains algorithmes. En effet, le principe des algorithmes récursifs est de se ramener à des objets de plus petite taille (par exemple, un algorithme de type « diviser pour régner » ramène à un problème de taille n/α, pour une constante α fixée, par exemple $\alpha = 2$). Si on sait qu'en dessous d'un certain seuil, l'algorithme récursif est moins bon qu'un algorithme classique, on a tout intérêt à terminer, pour les petites valeurs, par l'algorithme classique, donc à changer d'algorithme pour les données de plus petite taille. Le principe même de la récursivité étant de fractionner

les objets en objet plus petit, ceci peut avoir un effet très important même sur les objets de grande taille, et on parfois gagner un facteur 10 ou même plus sur le temps de calcul.

Ainsi, l'étude de la complexité asymptotique est importante sur fait des données de plus en plus grandes qu'est amené à manipuler un ordinateur, mais l'étude du temps de réponse sur des données plus petites n'est pas à négliger. Cette dernière se fait souvent de façon empirique. Nous nous contenterons ici de l'étude asymptotique.

Nous estimerons la complexité asymptotique à l'aide des comparaisons usuelles de l'analyse fournies par les notions d'équivalence, négligeabilité et dominance. Nous rappelons (pour des suites ne s'annulant pas) :

Définition 5.1.1 (Comparaisons asymtotiques)

- Équilalence : $u_n \sim v_n$ si et seulement si $\frac{u_n}{v_n} \longrightarrow 1$. Autrement dit, (u_n) et (v_n) ont même ordre de grandeur asymptotique.
- Négligeabilité : $u_n = o(v_n)$ si et seulement si $\frac{u_n}{v_n} \longrightarrow 0$. Ainsi, (u_n) est asymptotiquement infiniment petit devant (v_n)
- Dominance : $u_n = O(v_n)$ si et seulement si $\frac{u_n}{v_n}$ est borné. Ainsi, (u_n) ne grossit pas démesurément par rapport à (v_n).

À ces comparaisons classiquement utilisée en analyse standard, nous ajoutons deux notations dues à Knuth :

- $u_n = \Omega(v_n)$ s'il existe $m > 0$ tel que $\frac{u_n}{v_n} \geqslant m$ à partir d'un certain rang (ainsi, u_n ne devient pas démesurément petit devant v_n)
- $u_n = \Theta(v_n)$ s'il existe $m, M > 0$ tels que $m \leqslant \frac{u_n}{v_n} \leqslant M$ à partir d'un certain rang. Ainsi, (u_n) et (v_n) restent dans un rapport comparable.

Les notations les plus utilisées pour mesurer la complexité asymptotique sont O et Θ (qui est la combinaison de O et Ω). Il convient de remarquer que :

- O mesure le performances d'un algorithme : dire que la complexité est en $O(n^2)$ permet de borner (à constante près) le temps d'exécution. En revanche, l'algorithme peut tout à fait avoir un temps d'exécution bien meilleur : un algorithme en $O(n\ln(n))$ est aussi en $O(n^2)$!
- Ω mesure les limitations d'un algorithme : dire qu'un algorithme est en $\Omega(n)$ affirme qu'il est au moins en temps linéaire. Mais il peut être pire.
- Θ permet donc de classer de façon précise la complexité sur une échelle : un algorithme dont la complexité est en $\Theta(n^2)$ aura de façon effective un temps de réponse de l'ordre de n^2 (à constante près), ni plus lent, ni plus rapide.

II Complexité en temps (modèle à coûts fixes)

II.1 Première approche

Chaque opération effectuée nécessite un temps de traitement. Les opérations élémentaires effectuées au cours d'un algorithme sont :

- l'affectation
- les comparaisons
- les opérations sur des données numériques.

Évaluer la complexité en temps revient alors à sommer tous les temps d'exécution des différentes opérations effectuées lors de l'exécution d'un algorithme.

La durée d'exécution des différentes opérations dépend de la nature de l'opération : une multiplication sur un flottant demande par exemple plus de temps qu'une addition sur un entier. Ainsi, si on veut une expression précise du temps de calcul connaissant la durée d'exécution de chaque opération, il nous faut toute une série de constantes donnant ces différents temps.

Définition 5.2.1 (Modèle à coût fixe)

Le modèle à coût fixe consiste à considérer que la durée des opérations ne dépend pas de la taille des objets (notamment des entiers), et que les opérations sur les flottants sont de durée similaire aux opérations semblables sur les entiers.

Évidemment, suivant la situation, le modèle à coût fixe peut être acceptable ou non : si on est amené à manipuler de très longs entiers, il est évident que le modèle à coût fixe n'est pas bon : multiplier deux entiers à 1.000.000.000 chiffres est plus long que multiplier deux entiers à 1 chiffre !

Nous supposons ici que nous sommes dans une situation ou le modèle à coût fixe est acceptable. Nous nous donnons alors des constantes C_+, C_*, $C_/$ etc correspondant au temps des différentes opérations, ainsi que $C_\leftarrow$ pour l'affectation et $C_{==}$, $C_{<=}$ etc. pour les tests. Ces différentes valeurs peuvent être distinctes, mais sont toutes strictement positives.

Définition 5.2.2 (Complexité en temps)

La complexité en temps d'un algorithme est une fonction C dépendant de la taille n des données (éventuellement de plusieurs variables s'il y a en entrée des objets pouvant être de tailles différentes) et estimant le temps de réponse en fonction de n et des constantes $C_\bullet$ définies ci-dessus.

Étudions par exemple l'algorithme suivant :

Algorithme 5.1 : Évaluation naïve d'un polynôme

Entrée : T : tableau, coefficients d'un polynôme P, a : réel
Sortie : $P(a)$: réel.

$S \leftarrow 0$;
pour $i \leftarrow 0$ **à** n **faire**
  $b \leftarrow 1$;
  pour $j \leftarrow 1$ **à** i **faire**
    $b \leftarrow b \times a$
  fin pour
  $S \leftarrow S + T[i] \times b$
fin pour
renvoyer S

Exercice 6

1. Justifier la correction de l'algorithme ci-dessus.
2. En considérant que l'incrémentation de l'indice des boucles nécessite une addition et une affectation, montrer que pour un polynôme de degré n (donc un tableau indexé de 0 à n), la complexité en temps est :

$$C(n) = n^2\left(\frac{C_+}{2} + \frac{C_*}{2} + C_\leftarrow\right) + n\left(\frac{3}{2}C_+ + \frac{C_*}{2} + 2C_\leftarrow\right) + (4C_\leftarrow + 2C_+ + C_*)\,.$$

Une telle expression aussi précise peut être importante si on a à estimer de façon très précise le temps d'exécution pour des tableaux de taille connue. Cependant, dans l'étude du comportement asymptotique, on n'est souvent intéressé que par l'ordre de grandeur de la complexité. Ici, du fait que le coefficient précédant le terme en n^2 est strictement positif, on obtient :

$$C(n) = \Theta(n^2).$$

On dit que l'algorithme a un coût, ou une complexité, quadratique. Les différents comportements de référence sont :

Définition 5.2.3 (Complexités de référence)
- Coût constant : $C(n) = \Theta(1)$
- Coût logarithmique : $C(n) = \Theta(\ln(n))$
- Coût linéaire : $C(n) = \Theta(n)$
- Coût quasi-linéaire : $C(n) = \Theta(n \ln(n))$
- Coût quadratique : $C(n) = \Theta(n^2)$
- Coût cubique : $C(n) = \Theta(n^3)$
- Coût polynomial : $C(n) = \Theta(n^\alpha)$, $\alpha > 0$
- Coût exponentiel : $C(n) = \Theta(x^n)$, $x > 1$.

On utilisera la même terminologie (en précisant éventuellement « au plus ») si on ne dispose que d'un O au lieu du Θ.

Pour information, voici les temps approximatifs de calcul pour un processeur effectuant 1 milliard d'opérations par seconde, et pour une donnée de taille $n = 10^6$, suivant les coûts :
- $O(1)$: 1 ns
- $O(\ln(n))$: 15 ns
- $O(n)$: 1 ms
- $O(n \ln n)$: 15 ms
- $O(n^2)$: 15 min
- $O(n^3)$: 30 ans
- $O(2^n)$: 10^{300000} milliards d'années !

Ainsi, si on est amené à traiter des grandes données, l'amélioration des complexités est un enjeu important !

Remarque 5.2.4

En pratique, se rendre compte de l'évolution de la compexité n'est par très dur : pour des données de taille suffisamment grande, en doublant la taille des données :
- on n'augmente par le temps de calcul pour un algorithme en temps constant
- on augmente le temps de calcul d'une constante (indépendante de n) pour un coût logarithmique
- on double le temps de calcul pour un algorithme linéaire
- on quadruple le temps de calcul pour un algorithme quadratique
- on multiplie le temps de calcul par 8 pour un algorithme cubique
- ça ne répond plus pour un algorithme exponentiel (sauf si on a pris une valeur initiale n trop ridiculement petite)

II.2 Simplification du calcul de la complexité

Le calcul de la complexité effectué ci-dessus nécessite de distinguer les différentes opérations. De plus, les valeurs des différentes constantes ne sont pas des données absolues : elles dépendent de façon cruciale du processeur utilisé ! Nous voyons dans ce paragraphe comment s'affranchir de la connaissance de ces constantes.

Dans la recherche de la complexité asymptotique sous forme d'un Θ, la connaissance précise des $C_\bullet$ n'est pas indispensable : ces constantes interviennent, comme dans l'exemple traité, dans les coefficients de l'expression obtenu, coefficients qu'il n'est pas utile de connaître explicitement pour obtenie une expression en Θ (seule leur stricte positivité est indispensable)

Proposition 5.2.5 (Effet de la modification d'une des constantes)

Considérons un algorithme dont la complexité $C_1(n)$ vérifie $C_1(n) = \Theta(u_n)$ pour une certaine suite u_n.

Alors la complexité C_2 calculée en modifiant l'une quelconque des constantes, disons $C_\bullet$ en une valeur $C'_\bullet$ également strictement positive, vérifie également $C_2(n) = \Theta(u_n)$.

Corollaire 5.2.6

À condition de garder des valeurs strictement positives, on peut modifier la valeur des constantes $C_\bullet$ comme on veut, sans changer le comportement en $\Theta(u_n)$ de la complexité.

Corollaire 5.2.7 (Calcul simplifié de la complexité asymptotique)

On peut calculer le comportement asymptotique en Θ de la complexité en faisant la supposition que toutes les constantes de temps $C_\bullet$ sont égales à 1.

Autrement dit, le calcul de complexité asymptotique peut se faire sous la supposition que toutes les opérations ont le même temps d'exécution et on peut prendre ce temps d'exécution comme unité temporelle. On peut même considérer, pour simplifier, l'incrémentation d'un compteur de boucle comme nécessitant une seule unité de temps.
Évidemment, ce principe est aussi valable pour obtenir juste un O ou un Ω.

Exemple 5.2.8

Reprendre le calcul de la complexité asymptotique de l'évaluation naïve d'un polynôme sous cet angle.

II.3 Amélioration de la complexité de l'exemple donné

Un même problème peut souvent être résolu par différents algorithmes pouvant avoir des complexités différentes. Cette recherche de la performance est un enjeu capital de l'informatique. Nous l'illustrons sur notre exemple de l'évaluation polynomiale, en donnant deux améliorations possibles, et en recherchant leur complexité.

La première amélioration que nous proposons est le calcul rapide des puissances de a, basé sur la remarque que $a^{16} = (((a^2)^2)^2)^2$, permettant de la sorte de passer de 15 opérations à 4. De façon plus générale, il faut tenir compte de l'imparité possible de l'exposant.

Fonction 5.2 : exp-rapide(a,n)

Entrée : a entier ou réel, n entier
Sortie : a^n
si $n = 0$ **alors**
  renvoyer *1* et sortir
fin si
$y \leftarrow 1$;
$z \leftarrow 1$;
tant que $n > 0$ **faire**
  si n *est impair* **alors**
    $n \leftarrow n - 1$;
    $z \leftarrow z * y$
  sinon
    $n \leftarrow \frac{n}{2}$;
    $y \leftarrow y * y$
  fin si
fin tant que
renvoyer $y * z$

Algorithme 5.3 : Évaluation polynomiale par exponentiation rapide

Entrée : T : tableau des coefficients d'un polynôme P, a réel
Sortie : $P(a)$
$S \leftarrow 0$;
pour $i \leftarrow 0$ **à** n **faire**
  $S \leftarrow S + T[i] \times$ exp-rapide(a, i)
fin pour

Exercice 7

1. Justifier la terminaison et la correction de l'élgorithme d'exponentiation rapide.
2. Soit $C(n)$ le coût le l'exponentiation rapide pour un exposant n. Montrer que pour tout $k \in \mathbb{N}$ et tout $n \in [\![2^k, 2^{k+1}]\!]$,
$$C(2^k) \leqslant C(n) < C(2^{k+1}) + 2k.$$
3. En déduire que $C(n) = \Omega(\ln(n))$.
4. En déduire que l'algorithme d'évaluation polynomial obtenu ainsi a un coût en $\Omega(n \ln(n))$. Pour la minoration, on pourra se restreindre dans un premier temps à des indices entre $\frac{n}{2}$ et n.

Mais il est évidemment possible de faire mieux : ayant déjà calculé a^{i-1} au passage précédent dans une boucle, il est inutile de reprendre le calcul de a^i depuis le début : il suffit de multiplier la puissance obtenue précédemment par a, ce qui ne nécessite qu'une opération supplémentaire.
Cette remarque peut se traduire par l'égalité suivante, dans laquelle on a mis dès que possible les termes x en facteur :

$$a_0 + a_1 x + a_2 x^2 + \cdots + a_{n-1} x^{n-1} + a_n x^n = a_0 + x(a_1 + x(a_n + \cdots + x(a_{n-1} + x a_n))).$$

Cette factorisation est à la base de l'algorithme de Hörner :

Algorithme 5.4 : Hörner

Entrée : T, coefficients d'un polynôme P, a : réel
Sortie : $P(a)$
$S \leftarrow T[n]$;
pour $i \leqslant n-1$ **descendant à** 0 **faire**
  $S \leftarrow S * a + T[i]$
fin pour
renvoyer (S)

Exercice 8

1. Montrer la correction de l'algorithme de Hörner.
2. Montrer que l'algorithme de Hörner a un coût linéaire ($\Theta(n)$).

III Complexité dans le meilleur ou le pire des cas, en moyenne

III.1 Complexité dans le meilleur et le pire des cas

Il n'est parfois pas possible d'exprimer la complexité (que ce soit par une expression exacte, ou par une estimation asymptotique) pour une entrée quelconque. En effet, le nombre d'opérations effectuées, est parfois très dépendante des valeurs fournies en entrée : des valeurs de même taille peuvent parfois nécessiter des temps de calcul très différents. Un exemple très simple est fourni par le test le plus élémentaire de primalité :

Algorithme 5.5 : Test de primalité

Entrée : p : entier > 1
Sortie : b booléen, résultat du test de primalité
$i \leftarrow 2$;
$m = \sqrt{p}$;
tant que $i \leqslant m$ **faire**
  si $p \mod i = 0$ **alors**
    renvoyer *False* et sortir
  fin si
fin tant que
renvoyer *True*

Exercice 9

1. Justifier la terminaison et la correction de cet algorithme
2. Montrer que si p est pair (ce qui peut arriver pour p grand !) le temps de réponse est en $\Theta(1)$,
3. Montrer que si p est premier (Euclide m'a affirmé un jour qu'on peut aussi trouver des p aussi grand qu'on veut vérifiant cela), le temps de réponse est en $\Theta(\sqrt{p})$ (dans le modèle à coût constant, qui devient discutable ici pour des grandes valeurs de p).

Définition 5.3.1 (Complexité dans le pire et le meilleur des cas)

- Pour des objets de taille n, la complexité dans le meilleur des cas est le nombre minimal $C_-(n)$ d'opérations à effectuer pour des objets dont la taille est n.
- Pour des objets de taille n, la complexité dans le pire des cas est le nombre maximal $C_+(n)$ d'opérations à effectuer pour des objets dont la taille est n.

Pour un objet T donnée de taille n, on a donc toujours :

$$C_-(n) \leqslant C(T) \leqslant C_+(n),$$

où $C(T)$ désigne le nombre d'opérations à effectuer pour l'objet T.

Exemple 5.3.2

En considérant que la taille des entiers est donnée par le nombre de leur chiffres (donc en gros par $\log(p)$), et en remarquant que pour tout n, il existe des nombres pairs à n chiffres (ça, c'est trivial), et également des nombres premiers à n chiffres (ça l'est un peu moins, ça peut être vu comme une conséquence du postulat de Bertrand affirmant qu'il y a toujours un nombre premier entre m et $2m$), on obtient, pour l'algorithme de primalité :

$$C_-(n) = \Theta(1) \qquad \text{et} \qquad C_+(n) = \Theta(10^{\frac{n}{2}}).$$

Dans ce cas, même si le nombre d'opérations $C(n)$ pour des objets de taille n est impossible à définir explicitement (car dépendant de l'objet donné en entrée), on dira que la complexité vérifie :

$$C(n) = O(C_+(n)) \qquad \text{et} \qquad C(n) = \Omega(C_-(n)).$$

Ainsi, pour l'algorithme de primalité, $C(n) = \Omega(1)$ et $C(n) = O(10^{\frac{n}{2}})$.

Exercice 10

On propose une amélioration au tri à bulle vu dans un exercice du chapitre précédent, consistant à compter le nombre d'échanges effectués lors d'un passage du début à la fin du tableau, et a s'arrêter

lorsqu'aucun échange n'est effectué (ce qui signifie que les termes sont dans l'ordre). Déterminer la complexité dans le meilleur et dans le pire des cas, en donnant à chaque fois un exemple de configuration initiale associée.

Remarque 5.3.3

De nombreux tris (mais pas tous) sont plus rapides sur des tableaux presque triés (obtenus par exemple en ajoutant un petit nombre d'éléments à un tableau initialement trié, c'est une situation très fréquente, correspondant par exemple à l'ajout de quelques éléments dans une base qu'on maintient triée). Certains algorithmes, réputés plus lents sur des tableaux généraux, peuvent être plus rapides dans cette situation que des algorithmes récursifs réputés plus rapides dans des situations générales. Ainsi, avant de se précipiter vers un algorithme réputé rapide, il est important d'analyser le contexte dans lequel l'algorithme doit être utilisé.

III.2 Complexité en moyenne

La complexité dans le pire et dans le meilleur des cas n'est pas forcément une donnée très représentative du comportement général d'un algorithme : il s'agit souvent de situations bien particulières, qui, certes, dans certains situations, peuvent se produire souvent (voir les tris), mais dans d'autres, peuvent rester exceptionnelles. Dans ces situations, la notion de complexité en moyenne est plus intéressante.

Définition 5.3.4

La complexité en moyenne $C_m(n)$ est la moyenne du nombre d'opérations à effectuer pour chacun des objets de taille n. Si certains objets sont plus probables que d'autres, il faut en tenir compte en pondérant convenablement. Ainsi, de façon plus formelle, cette moyenne correspond à l'espérance de la variable aléatoire $C(n)$ donnant le nombre d'opérations à effectuer, définie sur l'espace probabilité Ω_n des objets de taille n

Très souvent, on considère la mesure de probabilité uniforme sur l'ensemble des objets.

Exercice 11

On considère un alphabet $\mathcal{A}$ à k lettres. On effectue, sur des tableaux constitués de caractères de l'alphabet $\mathcal{A}$, la recherche du premier a. Déterminer la complexité moyenne de cet algorithme, sachant que les tableaux considérés sont remplis aléatoirement.

III.3 Algorithmes randomisés

Il est fréquent d'introduire un aléa articiel de sorte à éviter les cas extrêmes. En effet, les cas les meilleurs, mais aussi les pires, correspondent dans certains situations, à des situations fréquentes. Ainsi est-on parfois amené à trier des données en sens inverse, à les remettre dans le bon sens, à trier un tableau presque trié, dans un sens ou dans l'autre. Ces différentes actions nous amènent souvent à fleurter avec le meilleur des cas (ce qui n'est pas trop gênant), mais aussi avec le pire des cas (ce l'est plus).
Introduire un aléa permet parfois de se ramener (avec une probabilité forte) à un cas général, dont la complexité sera plutôt de l'ordre moyen.

Définition 5.3.5 (Algorithme randomisé)

On parle d'algorithme randomisé lorsqu'on a introduit dans l'algorithme un aléa dont le but est d'éviter les cas extrêmes

Nous introduisons rapidement le tri rapide dont le principe général (et très vague) est le suivant :

Fonction 5.6 : trirapidenonrandomisé(T)

Entrée : T : tableau à trier
Sortie : T : tableau trié
Comparer tous les éléments autres que $T[0]$ à $T[0]$ et séparer en 2 tableaux ;
U : tableau des plus petits ;
V : tableau des plus grands ;
$U \leftarrow$ trirapidenonrandomisé(U) ;
$V \leftarrow$ trirapidenonrandomisé(V) ;
renvoyer concaténation de U, $[T[0]]$ et V.

On peut montrer, mais ce n'est pas complètement évident, que la complexité en moyenne du tri rapide est en $O(n \ln(n))$. C'est aussi la complexité dans le meilleur des cas, correspondant au cas où on divise à chaque fois le tableau en deux sous-tableaux de même taille (donc que la valeur pivot choisie se trouve au milieu).

Exercice 12

Montrer que le temps de calcul du tri rapide sur un tableau trié (dans un sens ou dans l'autre) est en $O(n^2)$.

Il s'agit d'ailleurs là de la complexité dans le pire des cas. Ainsi, le tri rapide est particulièrement mauvais sur les listes triées et presque triées, dans un sen ou dans l'autre. Comme dit plus haut, il s'agit de situations fréquentes dans la vie courante. Une façon d'y remédier est d'introduire un aléa dans le choix du pivot :

Fonction 5.7 : trirapiderandomisé(T)

Entrée : T : tableau à trier
Sortie : T : tableau trié
Choisir i aléatoirement dans $[\![0, n-1]\!]$ (indices du tableau) ;
Comparer tous les éléments autres que $T[i]$ à $T[i]$ et séparer en 2 tableaux ;
U : tableau des plus petits ;
V : tableau des plus grands ;
$U \leftarrow$ trirapiderandomisé(U) ;
$V \leftarrow$ trirapiderandomisé(V) ;
renvoyer concaténation de U, $[T[i]]$ et V.

De cette façon, on subit moins les effets de bord, et on peut bénéficier (sauf très grande malchance) d'un algorithme efficace dans toutes circonstances.

IV Limitations du modèle à coûts fixes

Si on est amené à manipuler des grands nombres, il est évident qu'on ne peut plus considérer le coût des opérations comme indépendant des entrées. On exprime alors le coût des opérations en fonction de $\lg(N)$, le nombre de chiffres de N en base 2 (ou choi d'une autre base à convenance). Les différentes opérations ont alors un coût qu'il convient de ne pas négliger, et qui diffère suivant les opérations. Par exemple, par l'algorithme standard, le coût de l'addition $m + n$ va être de l'ordre de $\max(\lg(m), \lg(n))$, alors que le coût du produit mn va être de l'ordre de $\lg(m) \times \lg(n)$. Le coût du produit peut être amélioré en utilisant des algorithmes plus sophistiqués, comme l'algorithme de Karatsuba, ou encore un algorithme basé sur la transformée de Fourier rapide.

Dans certaines situations, il est possible d'éviter les grands nombres. C'est le cas par exemple si on travaille modulo K, où K est un entier fixé. Attention à la maladresse consistant dans cette situation à

réduire modulo K à la fin du calcul. Comme exemple, reprenons l'algorithme de Hörner, adapté pour une évaluation modulo K. Que pensez-vous des deux algorithmes ci-dessous ? Lequel est meilleur ?

Algorithme 5.8 : Hörner 2

Entrée : T, coefficients d'un polynôme P, a : réel, K base du modulo
. **Sortie** : $P(a)$
$S \leftarrow T[n]$;
pour $i \leqslant n-1$ **descendant à** 0 **faire**
| $S \leftarrow S * a + T[i]$
fin pour
renvoyer $(S \mod K)$

Algorithme 5.9 : Hörner 3

Entrée : T, coefficients d'un polynôme P, a : réel, K base du modulo
. **Sortie** : $P(a)$
$S \leftarrow T[n] \mod K$;
pour $i \leftarrow n-1$ **descendant à** 0 **faire**
| $S \leftarrow ((S * a + T[i]) \mod K$
fin pour
renvoyer (S)

V Complexité en mémoire, ou en espace

Le but de l'étude de la complexité en mémoire, que nous aborderons très peu ici, est d'étudier l'encombrement en mémoire du fait de l'execution d'un algorithme. Cette encombrement peut provenir de mises en mémoire explicites, ou d'empilements implicites de données et d'instructions, par exemple dans le cadre d'un algorithme récursif.

Une mauvaise complexité en mémoire ralentit le programme (car la mise en mémoire doit se faire loin du processeur), et peut aller jusqu'à la saturation de la mémoire, cas dans lequel l'algorithme ne peut terminer son exécution.

VI Étude de quelques algorithmes de recherche

Ces quelques algorithmes interviennent fréquemment dans des algorithmes plus élaborés. On se contente souvent d'ailleurs d'utiliser des fonctions toutes faites, sans vraiment se préoccuper de ce qui se cache derrière. Il est cependant important de connaître leur fonctionnement, ainsi que leur complexité (afin de pouvoir estimer la complexité des algorithmes qui les utilisent).

VI.1 Recherche du maximum d'une liste

Problème : Étant donné une liste de réels, trouver le maximum de cette liste (ou de façon similaire, son minimum).

Idée de résolution : Progresser dans le tableau en mémorisant le plus grand élément rencontré jusque-là. On peut aussi mémoriser sa place.

On utilise la convention usuelle de Python pour les indexations : un tableau de taille n est supposé indexé de 0 à $n-1$.

Algorithme 5.10 : Recherche maximum

Entrée : T, tableau de réels de taille n
Sortie : $\max(T)$
$M \leftarrow T[1]$;
pour $i \leftarrow 1$ **à** $n-1$ **faire**
  si $T[i] > M$ **alors**
    $M \leftarrow T[i]$
  fin si
fin pour
renvoyer (M)

Exercice 13

Montrer que l'algorithme `Recherche maximum` est correct, et que son coût est en $\Theta(n)$.

VI.2 Recherche d'un élément dans une liste

Problème : Étant donné une liste, trouver un terme particulier dans cette liste, s'il y est.

Idée de résolution : Parcourir la liste jusqu'à ce qu'on trouver l'élément souhaité. Variantes : si on veut toutes les occurrences, on parcourt la liste en entier ; si on veut la dernière occurrence, on parcourt le tableau à partir de la fin.

Algorithme 5.11 : Recherche première occurrence

Entrée : T, tableau de réels de taille n ; a élément à chercher dans T
Sortie : i : indice de la première occurrence de T, ou `None` si pas d'occurrence
$i \leftarrow 0$;
tant que $i < n$ *et* $T[i] \neq a$ **faire**
  $i \leqslant 1$
fin tant que
si $i < n$ **alors**
  renvoyer (i)
fin si

Remarquez qu'on utilise ici les tests booléens dans le mode « paresseux », ce qui signifie que pour obtenir la valeur du booléen « b_1 et b_2 », dès lors que b_1 est faux, on ne va pas voir b_2 et on renvoie « faux ». Ainsi, dans cette boucle, si $i = n$, le test $T[i] \neq a$ n'est pas effectué. Sinon, on aurait un problème de validité

des indices (on sort du tableau). Beaucoup de langages (dont Python) effectue les opérations booléennes en mode paresseux.

Exercice 14

1. Montrer la terminaison et la correction de l'algorithme `Recherche première occurrence`
2. Montrer que sa complexité dans le pire des cas est en $\Theta(n)$, et dans le meilleur des cas en $\Theta(1)$.
3. Montrer que si T est un tableau constitué de N objets supposés équiprobables, et que a est un de ces objets, la complexité en moyenne tend vers N lorsque n tend vers ∞. Ainsi, la complexité moyenne est en $\Theta(1)$, si on considère N comme une constante. Si on est amené à faire varier à la fois la taille n du tableau et le nombre N d'objets différents, on peut écrire $C_m(n, N) = \Theta(N)$.
4. Comment modifier l'algorithme ci-dessus pour qu'il renvoie la dernière occurrence ? Toutes les occurrences. Quelle est la complexité dans ce cas ?

Remarque 5.6.1

Si le tableau T est utilisé essentiellement pour des tests d'appartenance, il faut se demander si la structure de tableau est vraiment adaptée : il est peut-être plus intéressant de considérer une structure d'ensemble, le test d'appartenance s'y faisant plus rapidement.

VI.3 Recherche dans un liste triée

Si la liste est triée, rechercher un élément, ou rechercher la position d'un élément à insérer, se fait beaucoup plus rapidement.

Problème : Étant donné un tableau T triée et un élément a comparable aux éléments du tableau, trouver la première occurrence de a (donc pouvoir dire si a est dans le tableau ou non), ou, de façon équivalente, trouver le plus petit indice i tel que $a \leqslant T[i]$ (dans ce cas, a est dans le tableau, de plus petit occurrence i, si et seulement si $a = T[i]$). Ce dernier problème étant un peu plus général, c'est lui qu'on étudie.

Idée 1 : parcourir le tableau dans l'ordre pour repérer i. Cela ne change pas grand chose à la méthode de recherche dans un tableau non trié. En particulier, en moyenne, le coût sera linéaire.

Idée 2 : faire une dichotomie, en coupant à chaque étape le tableau en 2, afin de n'en garder que la moitié pertinente. Intuitivement, cela va beaucoup plus vite, puisqu'à chaque étape, on réduit le nombre de possibilités de moitié.

Algorithme 5.12 : Recherche par dichotomie dans tableau trié

```
Entrée : T, tableau de réels de taille n, trié ; x élément à positionner dans T
Sortie : i : indice minimal tel que T[i] ⩾ x, éventuellement i = n
a ← 0 ;
b ← n − 1 ;
si T[b] < x alors
    renvoyer (n)
sinon si T[a] ⩾ x alors
    renvoyer (0)
sinon
    tant que b − a > 1 faire
        c = ⌊(a+b)/2⌋ ;
        si T(c) ⩾ x alors
            b ← c
        sinon
            a ← c
        fin si
    fin tant que
fin si
renvoyer (b)
```

Exercice 15

1. Prouver la terminaison et la correction de cet algorithme
2. Justifier que sauf dans le cas où les tests initiaux sont positifs, son coût est en $\Theta(\ln(n))$.
3. Expliquer comment améliorer l'algorithme si on ne recherche par nécessairement la première occurrence.
4. Expliquer comment récupérer la dernière occurrence.
5. Comment récupérer l'ensemble de toutes les occurrences ?

VI.4 Autres algorithmes

Les recherches ci-dessus peuvent bien sûr s'adapter à une chaîne de caractères. Un problème fréquent est celui de la recherche d'un mot dans un texte : on peut positionner le mot en toute place du texte, et faire une comparaison lettre à lettre : en cas d'échec, on déplace le mot d'un cran, tant que c'est possible. On se rend bien compte que la complexité est en $O(nk)$, où n est la longueur du texte et k la longueur du mot.

Moyennant un prétraitement du texte (confection d'une table triée des suffixes, pouvant se faire en $\Theta(n\ln(n))$), on peut améliorer les performances de la recherche en descendant à $O(\ln(n))$ (en considérant k comme constante), sans compter le prétraitement. Avec le prétraitement, on y perd. Ainsi, cela s'avère intéressant si le prétraitement est fait une fois pour toutes (dans un texte finalisé qui n'est pas trop amené à être modifié ; des mises à jours restent possibles), et est antérieur à la demande de recherche. C'est ce type d'algorithmes qui est à la base des recherches dans les pages internet.

On verra dans le chapitre suivant quelques algorithmes de tri de tableau. Ces algorithmes sont utiles dans la vie pratique tous les jours (par exemple lorsqu'on ordonne des données dans un tableur ou une base de données). Ils sont aussi à la base de nombreux algorithmes plus sophistiqués. Il est donc important de pouvoir trouver des tris de complexité minimale. Nous verrons plusieurs algorithmes en $\Theta(n^2)$ (tri à bulles, tri par sélection, tri par insertion), et quelques algorithmes en $\Theta(n\ln(n))$ (en moyenne) (tri rapide, tri fusion). On peut montrer que dans une situation générale, c'est le plus rapide qu'on puisse faire. Sous certaines hypothèses supplémentaires (par exemple travailler sur un ensemble fini de données), on peut descendre encore.

6 Calculs d'intégrales

Le but de ce chapitre est d'étudier quelques algorithmes permettant un calcul approché efficace d'intégrales. Ce chapitre est le premier chapitre d'une série de chapitres d'analyse numérique, dont le but est d'obtenir des méthodes approchées pour la résolution de certains problèmes mathématiques. L'intérêt en est que d'une part, les solutions exactes ne sont pas toujours faciles à obtenir mathématiquement, voire impossibles, et que d'autre part, dans de nombreuses disciplines scientifiques (physique, ingéniérie...), une bonne valeur approchée de la solution est souvent largement suffisante. Évidemment, pour des études de ce type, il est important de savoir évaluer le temps de calcul d'une part, combiné d'autre part à la précision du résultat. Ce n'est que cette maîtrise de la précision du résultat qui rend la méthode valide.

Le principe général du calcul des intégrales est la quadrature : l'interprétation géométrique des intégrales est connue de tous : il s'agit de l'aire de la courbe. L'idée est alors d'approcher la surface sous la courbe par une somme de surfaces élémentaires dont l'aire est facile à calculer, par exemple des rectangles. Ainsi :

- On prend une subdivision $a = \sigma_0 \leqslant \sigma_1, \leqslant \cdots \leqslant \sigma_n \leqslant b$ de l'intervalle d'intégration $[a, b]$ (figure 6.1),
- on approche l'intégrale entre σ_i et σ_{i+1} (c'est-à-dire la surface sous la protion de courbe entre ces deux valeurs) par une aire facile à calculer
- On recolle les morceaux par sommation, en utilisant de façon sous-jacente la relation de Chasles.
- Lorsque le pas de la subdivision tend vers 0, l'approximation devient bonne

On rappelle :

Définition 6.0.2 (Subdivision et pas d'une subdivision)

1. Une subdivision de l'intervalle $[a, b]$ est une séquence finie $(\sigma_i)_{i\in[\![0,n]\!]}$ telle que :

$$a = \sigma_0 < \sigma_1 \cdots < \sigma_{n-1} < \sigma_n = b.$$

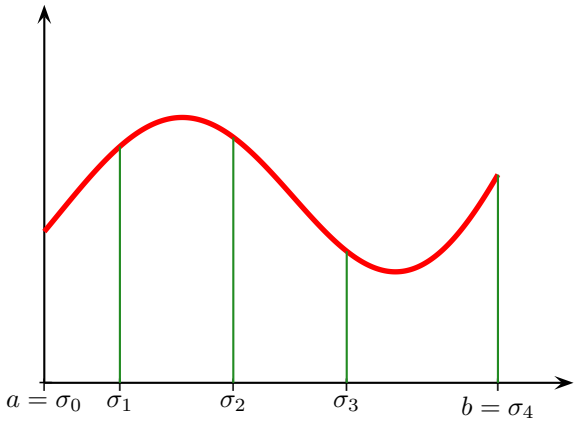


FIGURE 6.1 – Découpage de l'intégrale suivant une subdivision

2. Le pas p d'une subdivision $a = \sigma_0 < \sigma_1 < \cdots \sigma_n = b$ est la distance maximale entre deux points successifs de la subdivision :
$$p = \max_{i \in [\![1,n]\!]} \sigma_i - \sigma_{i-1}.$$

Les trois premières méthodes que nous étudions suivent ce schéma, en considérant des approximations de plus en plus fines sur chaque sous-intervalle de la subdivision.

I La méthode des rectangles

On commence par exposer la méthode des rectangles. Il s'agit en fait d'approcher l'intégrale par des « sommes de Riemann », donc nous donnons la construction ci-dessous (voir figure 6.2)

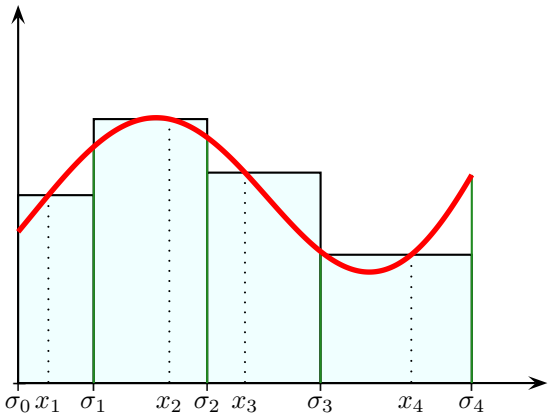


FIGURE 6.2 – Somme de Riemann

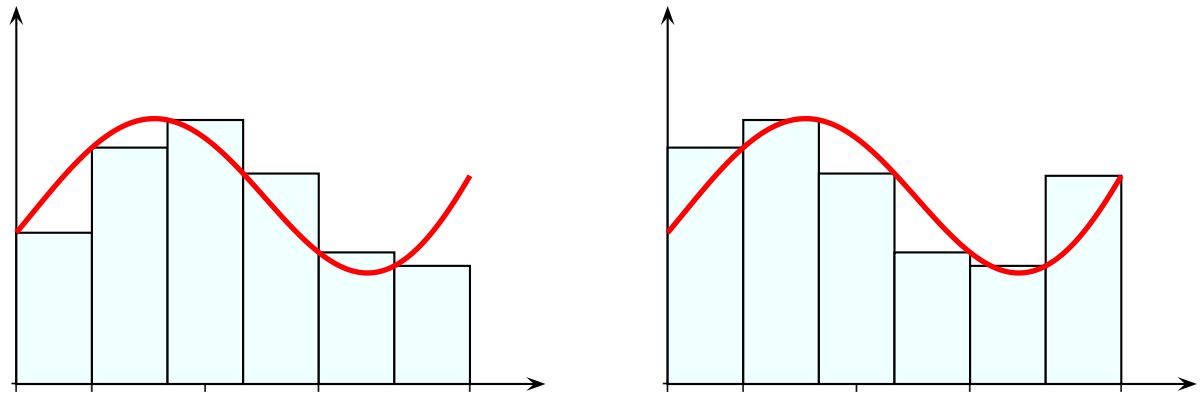


FIGURE 6.3 – Méthode des rectangles

Soit $\Sigma = \{a = \sigma_0 < \cdots < \sigma_n = b\}$ une subdivision de $[a, b]$, et $X_\sigma = \{x_1 \leqslant x_2 \leqslant \cdots \leqslant x_n\}$ des points associés à cette subdivision, c'est-à-dire :
$$\forall i \in [\![1, n]\!], \quad x_i \in [\sigma_{i-1}, \sigma_i].$$

On définit alors $I(\Sigma, X_\Sigma)$ l'approximation de l'intégrale obtenue en approchant la courbe sur $[\sigma_{i-1}, \sigma_i]$ par la fonction constante de valeur $f(x_i)$. Ainsi, le morceau d'aire sous la courbe entre les coordonnées σ_{i-1} et σ_i est approchée par l'aire d'un rectangle de base $[\sigma_{i-1}, \sigma_i]$ et de hauteur $f(x_i)$.
Ainsi :
$$I(\Sigma, X_\Sigma) = \sum_{i=1}^{n} f(x_i)(\sigma_{i-1} - \sigma_i).$$

Théorème 6.1.1 (Convergence des sommes de Riemann)
Soit f une fonction de classe $\mathcal{C}^0$ sur $[a,b]$. Alors la somme de Riemann $I(\Sigma, X_\Sigma)$ converge vers $\int_a^b f(t)\ \mathrm{d}t$ lorsque le pas p_Σ tend vers 0.

Ce résultat reste vrai dans un cadre plus général (voir cours de mathématiques sur les sommes de Riemann, dans le chapitre sur les intégrales).
On peut majorer la vitesse de convergence avec quelques hypothèses complémentaires :

Théorème 6.1.2 (Vitesse de convergence des sommes de Riemann)
Soit f une fonction de classe $\mathcal{C}^1$, et M_1 un majorant de $|f'|$ sur $[a,b]$. On a alors :

$$\left| \int_a^b f(t)\ \mathrm{d}t - I(\Sigma, X_\Sigma) \right| \leqslant M_1 p_\Sigma.$$

En informatique, il est assez naturel d'utiliser une subdivision régulière $\Sigma_n = \{a + k \cdot \frac{b-a}{n}.\}$. Dans ce cas, le pas est $p_n = \frac{1}{n}$, et on parle de « méthode des rectangles ».

Théorème 6.1.3 (Vitesse de convergence de la méthode des rectangles)
Soit Σ_n la subdivision régulière à n pas de l'intervalle $[a,b]$, et X_{Σ_n} une séquence de points associés. Alors

$$\left| I(\Sigma_n, X_{\Sigma_n}) - \int_a^f f(t)\ \mathrm{d}t \right| \leqslant \frac{M_1(b-a)}{n}.$$

On a donc une convergence de la méthode des rectangles en $O\left(\frac{1}{n}\right)$.

Le plus souvent, on prend pour la séquence X_{Σ_n} des points particuliers vis-à-vis des intervalles de la subdivision, par exemple la borne inférieure, la borne supérieure, ou le milieu.

Définition 6.1.4 (Méthode du point milieu)
Le choix des milieux des intervalles de la subdivision régulière fournit la méthode appelée *méthode du point milieu.*

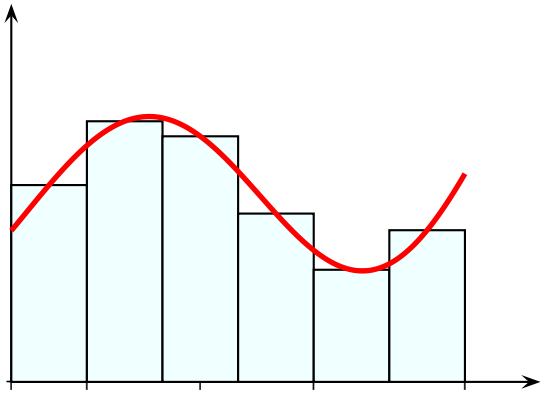


Figure 6.4 – Méthode du point milieu

Notons de façon plus générale, pour $\alpha \in [0,1]$, $X_{n,\alpha}$ la séquence de points $(x_1, \ldots, x_n)$ définie par :

$$x_i = a + (i - 1 + \alpha)\frac{b-a}{n}.$$

Le choix de la borne inférieur de chaque intervalle de la subdivision correspond au choix de la valeur $\alpha = 0$; le choix de la borne supérieure au choix de $\alpha = 1$, et le choix du milieu à $\alpha = \frac{1}{2}$. On peut donner explicitement les formules d'approximation dans ce cas :

- Borne inférieure : $I(\Sigma_n, X_{n,0}) = \dfrac{b-a}{n} \displaystyle\sum_{i=0}^{n-1} f\left(a + i\frac{b-a}{n}\right)$
- Borne supérieur : $I(\Sigma_n, X_{n,1}) = \dfrac{b-a}{n} \displaystyle\sum_{i=1}^{n} f\left(a + i\frac{b-a}{n}\right)$
- Point milieu : $I(\Sigma_n, X_{n,\frac{1}{2}}) = \dfrac{b-a}{n} \displaystyle\sum_{i=0}^{n-1} f\left(a + \left(i + \frac{1}{2}\right)\frac{b-a}{n}\right)$.

Définition 6.1.5 (Méthode exacte sur f)

On dit qu'une méthode d'approximation séquentielle (c'est-à-dire une méthode fournissant une suite $I_n(f)$ convergeant vers la valeur $I(f)$ recherchée) est exacte pour une fonction f si pour tout $n \in \mathbb{N}$, $I_n(f) = I(f)$.

Intuitivement, plus le famille des fonctions pour laquelle une méthode est exacte est grosse, plus la méthode a de chances de fournir de bonnes approximations.

Proposition 6.1.6

La méthode des rectangles avec un choix $X_{n,\alpha}$, $\alpha \neq \frac{1}{2}$, est exacte pour les fonctions constantes, mais pas pour les fonctions affines non constantes.

Proposition 6.1.7

La méthode du point milieu est exacte pour toutes les fonctions affines.

Ceci peut laisser présager une meilleure convergence de la méthode du point milieu : à approximation à l'ordre 1 (donc par la tangente) sur chaque intervalle de la subdivision, on va avoir une bonne compensation des erreurs su la première et la deuxième moitié de l'intervalle. Ces compensations vont permettre, moyennant une hypothèse de régularité plus forte, d'obtenir une convergence en $O\left(\frac{1}{n^2}\right)$.

Théorème 6.1.8 (Vitesse de convergence de la méthode du point milieu)

Soit f une fonction de classe $\mathcal{C}^2$ sur $[a, b]$ et M_2 un majorant de f'' sur cet intervalle. Alors

$$\left| \int_0^1 f(t)\ \mathrm{d}t - I(\Sigma_n, X_{n,\frac{1}{2}}) \right| \leqslant \frac{M_2}{24n^2}.$$

Ainsi, la méthode du point milieu converge en $O\left(\frac{1}{n^2}\right)$

Exemple 6.1.9

Trouver un exemple pour lequel la convergence de la méthode des rectangles avec la borne inférieure n'est pas en $O\left(\frac{1}{n^2}\right)$. Ainsi, la méthode du point milieu donne une réelle amélioration.

II La méthode des trapèzes

La méthode des trapèzes consiste à faire une approximation de la courbe non plus par des fonctions en escalier, mais par une fonction affine par morceaux, chaque morceau étant une corde de la courbe. Ainsi, étant donnée une subdivision $\Sigma = \{\sigma_0, \ldots, \sigma_n\}$, on définit la fonction affine rejoignant les points de

coordonnée $(\sigma_i, f(\sigma_i))$. L'intégrale de la fonction sur un intervalle de la subdivision est alors approchée par l'aire d'un trapèze. Cette aire est donnée, pour l'intervalle $[\sigma_{i-1}, \sigma_i]$, par

$$(\sigma_i - \sigma_{i-1})\frac{f(\sigma_i) + f(\sigma_{i-1})}{2}.$$

Pour éviter une trop grande technicité, on se place directement dans le cas d'une subdivision régulière Σ_n.

Proposition 6.2.1 (Méthode des trapèzes)
Soit f une fonction continue sur $[a,b]$. L'approximation de $\int_a^b f(t)\,\mathrm{d}t$ par la méthode des trapèzes est donnée par la suite :

$$I_n(f) = \frac{b-a}{n}\sum_{k=1}^{n}\frac{1}{2}\left(f(\sigma_k) + f(\sigma_{k-1})\right),$$

où pour tout $k \in [\![0,n]\!]$, $\sigma_k = a + k\dfrac{b-a}{n}$. Cette somme peut se réécrire plus simplement :

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

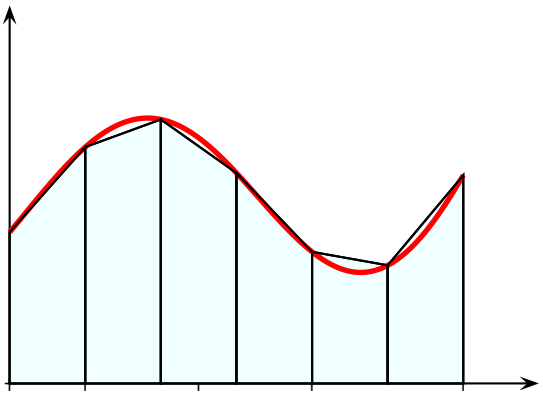


Figure 6.5 – Méthode des trapèzes

Proposition 6.2.2
La méthode des trapèzes est exacte pour les fonctions affines.

On peut donc espérer une convergence similaire à la méthode du point milieu. C'est ce qu'on établit ci-dessous :

Théorème 6.2.3 (Vitesse de convergence de la méthode des trapèzes)
Soit f une fonction de classe $\mathcal{C}^2$ sur $[a,b]$, et $I_n(f)$ la suite des approximations par la méthode des trapèzes. Soit M_2 un majorant de $|f''|$. Alors

$$\left|\int_a^b f(t)\,\mathrm{d}t - I_n(f)\right| \leqslant \frac{M_2(b-a)^3}{6n^2}.$$

La convergence est donc en $O\left(\frac{1}{n^2}\right)$.

Corollaire 6.2.4 (Retour sur la vitesse de CV de la méthode des rectangles)
Si $f(0) \neq f(1)$, on a l'équivalent suivant pour la vitesse de convergence de la méthode des rectangles avec choix des points au bord gauche des intervalles :

$$\int_a^b f(t)\ \mathrm{d}t - I(\Sigma_n, X_{n,0}) \underset{+\infty}{\sim} \frac{(b-a)(f(b)-f(a))}{2n}.$$

Confrontez ce résultat à une situation simple dans laquelle cette différence se calcule de façon exacte, par exemple le cas d'une fonction affine. Dans ce cas, l'équivalent est une égalité.

Remarque 6.2.5
La méthode des trapèzes n'est pas fondamentalement meilleure que la méthode du point milieu ; la constante obtenue dans la majoration est même un peu moins bonne.

III La méthode de Simpson

Il s'agit ici d'approcher localement la courbe par des courbes polynomiales de degré 2. Un polynôme de degré au plus 2 est entièrement déterminé par l'image de 3 points (il s'agit d'une interpolation de Lagrange) : étant donné $x_1 < x_2 < x_3$ trois réels distincts, et y_1, y_2, y_3 trois réels, l'unique polynôme de degré au plus 2 tel que pour tout $i \in [\![1,3]\!]$, $f(x_i) = y_i$ est donné par :

$$P(X) = y_1 \frac{(X-x_2)(X-x_3)}{(x_1-x_2)(x_1-x_3)} + y_2 \frac{(X-x_1)(X-x_3)}{(x_2-x_1)(x_2-x_3)} + y_3 \frac{(X-x_1)(X-x_2)}{(x_3-x_1)(x_3-x_2)}.$$

La méthode de Simpson consiste à approcher la courbe sur chaque intervalle $[\alpha, \beta]$ d'une subdivision de l'intervalle $[a,b]$ par l'unique parabole coïncidant avec la courbe aux points α, β et $\frac{\alpha+\beta}{2}$.
D'après ce qui précède, ce polynôme est donné par :

$$\begin{aligned} P(X) &= f(\alpha)\frac{(X-\beta)\left(X-\frac{\alpha+\beta}{2}\right)}{(\alpha-\beta)\left(\alpha-\frac{\alpha+\beta}{2}\right)} + f(\beta)\frac{(X-\alpha)\left(X-\frac{\alpha+\beta}{2}\right)}{(\beta-\alpha)\left(\beta-\frac{\alpha+\beta}{2}\right)} + f\left(\frac{\alpha+\beta}{2}\right)\frac{(X-\alpha)(X-\beta)}{\left(\frac{\alpha+\beta}{2}-\alpha\right)\left(\frac{\alpha-\beta}{2}-\beta\right)}. \\ &= \frac{1}{(\alpha-\beta)^2}\left(f(\alpha)(X-\beta)(2X-(\alpha+\beta)) + f(\beta)(X-\alpha)(2X-(\alpha+\beta) + 4f\left(\frac{\alpha+\beta}{2}(X-\alpha)(X-\beta)\right)\right) \end{aligned}$$

Pour exprimer notre approximation, on calcule donc l'intégrale de la fonction polynomiale associée entre α et β, en procédant par étapes :

$$\int_\alpha^\beta (t-\beta)(2t-(\alpha+\beta))\ \mathrm{d}t = \int_\alpha^\beta (t-\alpha)(2t-(\alpha+\beta))\ \mathrm{d}t = \frac{(\beta-\alpha)^3}{6} \qquad \text{et} \qquad \int_\alpha^\beta (t-\alpha)(t-\beta)\ \mathrm{d}t = -\frac{(b-a)^3}{6}.$$

On obtient donc :

$$\int_\alpha^\beta P(t)\ \mathrm{d}t = \frac{(\beta-\alpha)}{6}\sum_{k=0}^{n-1}\left(f(\alpha) + 4f\left(\frac{\alpha+\beta}{2}\right) + f(\beta)\right).$$

Définition 6.3.1 (Méthode de Simpson)
On obtient la méthode d'approximation de Simpson en approchant, sur tout intervalle de la subdivision régulière de $[a,b]$ en n pas, la courbe par le polynôme d'interpolation de degré au plus 2, coïncidant avec la courbe aux bords et au milieu de l'intervalle. Ainsi, l'approximation au rang n est donnée par :

$$I_n(f) = \frac{b-a}{6n}\sum_{k=0}^{n-1}\left(f(\sigma_k) + 4f\left(\frac{\sigma_k+\sigma_{k+1}}{2}\right) + f(\sigma_{k+1})\right),$$

où $\sigma_k = a + k\frac{b-a}{2}$.

Cette dernière expression peut être arrangée un peu en regroupant les termes $f(\sigma_k)$, mais l'expression obtenue est un peu plus obscure :

$$I_n(f) = \frac{(b-a)}{3n}\left(\sum_{k=1}^{n-1} f(\sigma_k) + 2\sum_{k=0}^{n-1} f\left(\frac{\sigma_k + \sigma_{k+1}}{2}\right)\right) + \frac{b-a}{6n}\cdot(f(b)+f(a)).$$

Théorème 6.3.2 (Vitesse de convergence de la méthode de Simpson)

Soit f une fonction de classe $\mathcal{C}^4$ sur $[a,b]$, et M_4 un majorant de $|f^{(4)}|$ sur cet intervalle. Soit $I_n(f)$ l'approximation de Simpson de rang n. Alors :

$$\left|\int_a^b f(t)\ \mathrm{d}t - I_n(f)\right| \leqslant \frac{M_4}{2880}\cdot\frac{1}{n^4}.$$

Ainsi, la convergence de la méthode de Simpson est en $O\left(\frac{1}{n^4}\right)$.

On rencontre parfois aussi la méthode de Romberg, qui est en fait basée sur la méthode des trapèzes, associé à un procédé d'accélération de convergence.

IV La méthode de Monte-Carlo

La dernière méthode que nous étudions est d'un type très différent des précédents, puisqu'il s'agit d'une méthode probabiliste.
La méthode de Monte-Carlo exploite le fait que l'intégrale peut être interprétée la longueur de l'intervalle d'intégration multipliée par la « valeur moyenne » de f sur cet intervalle. On obtient alors la méthode suivante :

Définition 6.4.1 (Méthode de Monte-Carlo)

Soit f une fonction continue sur $[a,b]$, et soit $X_1, \cdots, X_n$ des variables aléatoires mutuellement indépendantes suivant toutes la loi uniforme sur $[a,b]$ (on parle d'échatillon indépendant identiquement distribué (i.i.d.) de loi parente la loi uniforme sur $[a,b]$). On définit alors la variable aléatoire I_n par :

$$I_n = \frac{(b-a)}{n}\sum_{i=1}^{n} f(X_i).$$

La méthode de Monte-Carlo consiste à approcher f par une réalisation de la variable aléatoire I_n.

La justification de cette méthode provient du fait que l'espérance de I_n est égale à l'intégrale qu'on veut calculer, et que la variance tend vers 0 lorsque n tend vers $+\infty$, ce qui assure que pour n grand, la probabilité d'obtenir une valeur proche de l'intégrale voulue est grande.
On rappelle (ou on admet) le théorème de transfert, permettant le calcul des espérances de variables aléatoires $f(X)$: étant donné une variable aléatoire X de fonction de densité φ définie sur $\mathbb{R}$, et f une fonction continue, l'espérance de $f(X)$ existe si et seulement si l'intégrale ci-dessous converge absolument, et dans ce cas :

$$E(f(X)) = \int_{\mathbb{R}} \varphi(t) f(t)\ \mathrm{d}t.$$

Appliquons cela à une variable aléatoire suivant la loi uniforme sur $[a,b]$: sa densité est $\mathtt{un}_{[a,b]}$.

Lemme 6.4.2

Soit X une variable aléatoire suivant une loi uniforme sur $[a,b]$, et f une fonction continue sur $[a,b]$. Alors

$$E(f(X)) = \int_a^b f(t)\ \mathrm{d}t \qquad et \qquad V(f(X)) = \int_a^b f(t)^2\ \mathrm{d}t - \left(\int_a^b f(t)\ \mathrm{d}t\right)^2 = J(f).$$

Proposition 6.4.3 (Convergence de la méthode de Monte-Carlo)

Pour tout $n \in \mathbb{N}$,

$$E(I_n) = \int_a^b f(t)\ \mathrm{d}t \qquad et \qquad V(I_n) = \frac{J(f)}{n},$$

où la quantité $J(f)$ est définie dans le lemme précédent.

La méthode de Monte-Carlo est beaucoup plus lente et moins fiable que les méthodes étudiées précédemment. En revanche, elle est immédiate à adapter au cas des fonctions de plusieurs variables, et est donc assez utilisée dans ce cadre.

On peut par exemple obtenir ainsi une approximation de π en considérant la fonction f définie sur $[0,2]^2$ par

$$f(x,y) = \begin{cases} 1 & \text{si } x^2 + y^2 \leqslant 4 \\ 0 & \text{sinon.} \end{cases}$$

L'intégrale de cette fonction sur $[0,2]^2$ nous donne l'aire d'un quart de disque de rayon 2 (ou le volume d'un cylindre de hauteur 1 et de base un quart de disque de rayon 2).

7

Résolution numérique d'équations

Le but de ce chapitre est d'exposer quelques méthodes de résolutions d'équations $f(x) = 0$, où $f : I \to \mathbb{R}$ est une fonction définie sur un intervalle I. La fonction f sera supposée continue (et pour certaines méthodes un peu plus que cela même).
L'étude de ces méthodes passera par la description algorithmique de la méthode, permettant l'implémentation dans un langage de programmation, puis par les justifications de convergence et de rapidité. Pour assurer la convergence, il est parfois nécessaire d'ajouter des hypothèses (assurant l'existence d'une solution, ce qui est souvent obtenu par le TVI, ou même parfois, assurant l'existence d'une solution suffisamment proche de la valeur d'initialisation, ce qui nécessite alors une localisation grossière préalable).
Par ailleurs, en cas de non unicité des solutions, même en cas de convergence, il n'est pas toujours possible de savoir vers quel zéro on aura convergence, si on n'effectue pas un prétraitement permettant de séparer les racines.

Les méthodes que nous étudions sont tout d'abord la dichotomie (cette méthode est d'ailleurs une preuve possible du TVI assurant l'existence d'une solution), puis la méthode de la fausse position, qui en est une variante, la méthode de la sécante, et la version « limite » de cette méthode, qui est la méthode de Newton. La méthode de Newton est en quelque sorte l'aboutissement de la méthode de la sécante et assure une convergence plus rapide, mais elle nécessite la connaissance de la dérivée. Cette dérivée peut être donnée explicitement par l'utilisateur, ou peut être obtenue numériquement (mais cela nécessite que la courbe soit localement suffisamment lisse, car les microvariations ne pourront pas être prises en considération par une méthode numérique). Nous abordons le problème de la dérivation numérique dans la dernière partie de ce chapitre.

La dichotomie et la méthode de Newton sont au programme de l'informatique commune de MPSI/PCSI. Les méthodes de fausse position, et de la sécante, ainsi que la dérivation numérique, sont hors-programme.

Le lecteur surpris par l'absence de dessins alors qu'ils semblent s'imposer peut se rassurer : ces dessins seront rajoutés dans la prochaine version de ce texte (l'année prochaine...). En attendant, ils seront faits au tableau.

I Dichotomie

La dichotomie est une des preuves possibles de la démonstration du théorème des valeurs intermédiaires. Le principe 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. Ainsi :

Méthode 7.1.1 (Description de la méthode de dichotomie)

- 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 $f(a_0)f(b_0) \leqslant 0$.
- On coupe l'intervalle en 2 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.

Cela se traduit par l'algorithme suivant, en supposant initialement que $f(a)f(b) < 0$ (il faudrait faire un test et retourner une erreur si ce n'est pas le cas) :

Algorithme 7.1 : Dichotomie

Entrée : f : fonction ;
$a < b$: intervalle initial de recherche, tel que $f(a)f(b) < 0$;
ε : marge d'erreur
Sortie : x : valeur approchée à ε près d'un zéro de f sur $[a, b]$

```
tant que b − a > ε faire
    c ← (a+b)/2 ;
    si f(a)f(c) ⩽ 0 alors
        b ← c
    sinon
        a ← c
    fin si
fin tant que
renvoyer (a+b)/2
```

Remarquez qu'on obtient de la sorte une valeur approchée à $\frac{\varepsilon}{2}$ en fait : on pourrait se contenter de comparer $b - a$ à 2ε.

Proposition 7.1.2 (Validité et rapidité de l'algorithme de dichotomie)

La fonction f étant supposée continue, et en notant x la valeur retournée :

(i) L'algorithme s'arrête (la terminaison est assurée)

(ii) il existe un zéro de f dans l'intervalle $]x - \varepsilon, x + \varepsilon[$

(iii) Le temps de calcul est en $O(-\ln(\varepsilon))$. Plus précisément, le nombre d'étapes est $\left\lceil \log_2\left(\frac{b-a}{\varepsilon}\right)\right\rceil$

(iv) On peut aussi dire que la convergence est linéaire (en $\Theta(n)$) en le nombre de décimales obtenues (donc en exprimant $\varepsilon = 10^{-n}$)

Remarque 7.1.3

On gagne un facteur 2 en précision à chaque itération. Ainsi, il faut un peu plus de 3 itérations pour gagner une décimale supplémentaire.
Par exemple, en partant d'un intervalle initial de longueur 1, on obtient une précision 10^{-10} au bout de 33 itérations.

Cela reste un bon algorithme, mais si f elle-même est longue à calculer, tout gain de complexité peut être important. Nous verrons un peu plus loin la méthode de Newton, qui donne une convergence nettement plus rapide.

Remarque 7.1.4 (Comparatif de la méthode de dichotomie)

- Points forts :
  - simplicité,
  - convergence assurée,
  - vitesse raisonnable.
- Points faibles :
  - Nécessite un encadrement préalable du zéro ;
  - s'il y a plusieurs zéros dans l'intervalle, on n'en obtient qu'un
  - Moins rapide que la méthode de Newton, ou la méthode de la sécante.

II Méthode de la fausse position (HP)

La méthode de la fausse position (regula falsi) est aussi appelée méthode de Lagrange ou méthode d'interpolation linéaire.

Méthode 7.2.1 (Méthode de la fausse position, regula falsi)

Initialisation par un intervalle $[a, b]$, tel que $f(a)f(b) \leqslant 0$

- On approche f par la corde en a et b (interpolation de lagrange d'ordre 1, c'est-à-dire interpolation linéaire)
- On détermine le point d'intersection de cette corde et de l'axe des abscisses. Soit c_1 son abscisse.
- Des deux intervalles $[a, c_1]$, $[a, c_2]$, l'un au moins assure un changement de signe de f, on garde celui-ci
- On recommence sur l'intervalle conservé, jusqu'à obtenir la précision souhaitée.

Pour pouvoir mettre en forme l'algorithme, il faut savoir calculer la nouvelle borne de l'intervalle :

Proposition 7.2.2 (Expression de la solution par interpolation linéaire)

Soit f définie sur $[a, b]$, telle que $f(a)f(b) < 0$. L'abscisse du point d'intersection de la corde en a et b et de l'axe des abscisses est

$$c = a - \frac{(b-a)f(a)}{f(b) - f(a)}.$$

Il faut également régler le problème de la condition d'arrêt. On ne peut en effet par prendre comme condition d'arrêt le fait que $b - a < \varepsilon$, car les deux bornes ne convergent en général pas vers une limite commune, comme le montre l'exemple d'une fonction convexe (c'est toujours la même borne qui est modifiée)

On peut alors décider de prendre comme condition d'arrêt l'existence d'un zéro dans l'intervalle $[c-\varepsilon, c+\varepsilon]$, où c est la nouvelle borne calculée.

Ainsi, cela nous donne l'algorithme 7.2.

Théorème 7.2.3 (Convergence de la méthode de la fausse position)

Soit f une fonction continue, et $[a_n, b_n]$ l'intervalle obtenu après la n-ième itération. Alors les suites (a_n) et (b_n) sont convergentes, et l'une au moins des deux limites est un zéro de f, l'autre étant stationnaire.

Pour cela nous utilisons le lemme suivant

Lemme 7.2.4

Sous les hypothèses du théorème précédent, si les limites ℓ et ℓ' de (a_n) et (b_n) sont distinctes et si $f(\ell) \neq f(\ell')$, l'une des deux suites (a_n) et (b_n) est stationnaire.

Ainsi, la méthode converge toujours, mais cela n'assure pas pour autant la terminaison de l'algorithme donné. En effet, dans le cas d'une convergence vers une valeur autour de laquel il existe une concentration de 0 (disons plusieurs zéros à ε près), le fonction peut changer de signe plusieurs fois localement, ce qui peut fausser le test d'arrêt.

Avertissement 7.2.5

La vitesse de convergence peut être assez mauvaise, si la courbe est plate autour de son zéro.

Exemple 7.2.6

Cas de la fonction $x \mapsto x^3$, avec $[a_0, b_0] = [-\frac{1}{2}, 1]$. On a alors pour tout n, $b_n = 1$, et

$$a_{n+1} = a_n - \frac{1 - a_n}{1 - a_n^3} a_n^3.$$

On peut montrer que $a_n \underset{+\infty}{\sim} \dfrac{1}{\sqrt{2n}}$.
Ainsi, pour obtenir $|a_n| < \varepsilon$, il faut à peut près $\frac{1}{2}\varepsilon^{-2}$ itérations. La convergence est nettement moins bonne que par dichotomie. Exprimée en nombre de décimales obtenues, la complexité est exponentielle (en $O(10^{2n})$ pour une approximation à 10^{-n} près).

On estime la vitesse de convergence avec des conditions supplémentaires assurant une convergence raisonnable.

Proposition 7.2.7 (Vitesse de convergence de la méthode de fausse position)

Soit f une fonction convexe sur $[a, b]$ et de classe $\mathcal{C}^2$ telle que $f(a) < 0$ et $f(b) > 0$. On suppose que $f' > 0$ sur $[a, b]$. Alors la convergence est en $O(n)$, où n est le nombre de décimales obtenues. Il s'agit donc d'une convergence linéaire en le nombre de décimales.

La preuve de ce résultat montre que la convergence est d'autant plus rapide asymptotiquement que $f'(x)$ est grand. En revanche, lorsque $f'(x)$ est proche de 0, on s'approche du cas critique d'une courbe plate au voisinage du zéro (voir l'exemple de $x \mapsto x^3$).

Remarque 7.2.8

La proposition précédente reste valable si $f(a) > 0$ et $f(b) < 0$ (échanger le rôle de a et b), ou pour une fonction concave (considérer $-f$ qui est convexe, ce qui revient à faire une symétrie par rapport à l'axe des abscisses).

Remarque 7.2.9 (Comparatif de la méthode de fausse position)

- Points positifs
  - Simplicité de l'interpolation, approche assez intuitive.
  - Convergence théorique assurée
- Points négatifs
  - Condition d'arrêt pas nette
  - Vitesse de convergence mal maîtrisée, pouvant être mauvaise dans certaines situations

III Méthode de la sécante (HP)

La méthode de la sécante est une amélioration de la méthode de la fausse position. La seule différence réside dans le choix de l'intervalle conservé à chaque étape : on garde systématiquement l'intervalle formé par les deux dernières bornes calculées, même si la présence d'un zéro dans cet intervalle n'est pas assurée par un changement de signe. En particulier, cela complique l'étude de la convergence, car si $f(a_n)f(b_n) > 0$, le point suivant ne sera pas dans l'intervalle $[a_n, b_n]$.

Méthode 7.3.1 (Méthode de la sécante)

- On part d'un intervalle $[a, b] = [c_0, c_1]$ sur lequel f est continue. On n'impose pas de changement de signe sur cet intervalle.
- On calcule c_2 le point d'intersection de la corde avec l'axe des abscisses.
- On refait pareil sur l'intervalle $[c_1, c_2]$, puis $[c_2, c_3]$ etc, jusqu'à obtenir une approximation suffisante.
- Les intervalles $[c_n, c_{n+1}]$ sont à comprendre par $[c_{n+1}, c_n]$ si $c_{n+1} < c_n$.

Ainsi, on n'a cette fois qu'une suite à calculer, déterminée par la relation de récurrence exprimant l'intersection de la corde et de l'axe des abscisses :

$$c_{n+2} = c_{n+1} - f(c_{n+1})\frac{c_{n+1} - c_n}{f(c_{n+1}) - f(c_n)}.$$

La condition peut être gérée comme dans la méthode de la fausse position, ou alors en s'arrêtant lorsque deux valeurs consécutives sont proches l'une de l'autre (mais le contrôle de l'erreur est alors moins bon). On obtient l'algorithme 7.3.

Avertissement 7.3.2 (La méthode n'est pas toujours bien définie)

- Comme on n'impose plus de changement de signe de f sur $[c_n, c_{n+1}]$, la valeur de c_{n+2} peut sortir de cet intervalle, et même de l'intervalle $[a, b]$ initial : si les valeurs de $f(c_n)$ et $f(c_{n+1})$ sont proches, la corde est presque parallèle à l'axe des abscisses et c_{n+2} est éloigné de c_n et c_{n+1}. Dans ce cas, c_{n+2} peut sortir du domaine de définition
- Si les valeurs de $f(c_n)$ et $f(c_{n+1})$, la valeur de c_{n+2} ne peut pas être calculée. Dans ce cas également, la valeur méthode ne peut pas être itérée

Lemme 7.3.3

Supposons f de classe $\mathcal{C}^2$ sur $[a, b]$, tel que $f(a)f(b) < 0$. Supposons que f' ne s'annule pas sur $[a, b]$ et soit $m_1 = \min_{[a,b]}(|f'|) > 0$, et $M_2 = \max_{[a,b]}(|f''|)$. Si de plus, $\frac{M_2}{m_1}|b - a| \leqslant 1$, alors la fonction f admet un unique zéro x dans $[a, b]$, la suite $(c_n)_{n\in\mathbb{N}}$ est bien définie, et pour tout $n \in \mathbb{N}$,

$$|c_{n+2} - x| \leqslant 2|c_{n+1} - x| \cdot |c_n - x|.$$

Remarque 7.3.4

La condition $\frac{M_2}{m_1}|b - a| \leqslant 1$ peut être obtenue par restriction de l'intervalle $[a, b]$, si m_1 et M_2 sont connus. Cela nécessite une localisation grossière du zéro de f (par exemple par dichotomie).

Pour résoudre cette inégalité, on utilise les deux lemmes suivants :

Lemme 7.3.5

Soit $(y_n)_{n\in\mathbb{N}}$ une suite positive telle que pour tout $n \in \mathbb{N}$, $y_{n+2} \leqslant y_{n+1}y_n$. Alors

$$\forall n \in \mathbb{N}, \quad y_n \leqslant a^{\varphi^n},$$

où $\varphi = \frac{1+\sqrt{5}}{2}$ est le nombre d'or, solution positive de l'équation $\varphi^2 = \varphi + 1$, et $a = \max(y_0, y_1^{\varphi})$.

Lemme 7.3.6

Soit (y_n) une suite positive telle qu'il existe $a \in]0,1[$, $M > 0$ et $\rho > 0$ tels que pour tout $n \in \mathbb{N}$, $y_n \leqslant M a^{\rho^n}$. Alors il existe $N > 0$ et $b \in]0,1[$ tels que pour tout $n \geqslant N$, $y_n \leqslant b^{\rho^n}$.

En particulier, en posant $y_n = 2|c_n - x|$, on obtient :

Théorème 7.3.7 (Convergence locale de la méthode de la sécante)

Supposons f de classe $\mathcal{C}^2$ sur $[a,b]$, tel que $f(a)f(b) < 0$. Supposons que f' ne s'annule pas sur $[a,b]$ et soit $m_1 = \min_{[a,b]}(|f'|) > 0$, et $M_2 = \max_{[a,b]}(|f''|)$. Alors la méthode de la sécante converge, et avec les notations ci-dessus, pour tout n à partir d'un certain rang :

$$|c_n - x| \leqslant a^{\varphi^n}.$$

On dit que la méthode est *d'ordre φ*.

Remarque 7.3.8

Plus généralement, une méthode est d'ordre β si, en notant (x_n) l'approximation de rang n et x la valeur recherchée, il existe a tel que pour tout n à partir d'un certain rang, on ait :

$$|c_n - x| \leqslant a^{\beta^n}.$$

Dire qu'une méthode est d'ordre β signifie grossièrement que le nombre de décimales correctes est multiplié par β à chaque étape.

En effet,

$$-\log_{10}(a^{\beta^n}) = -\beta^n \log_{10}(a).$$

Or, cette expression donne à peu près le rang du premier chiffre non nul après la virgule de α^{β^n}, donc une minoration du rang du premier chiffre de $|c_n - x|$, donc une minoration du nombre des premières décimales communes à c_n et x.
Ainsi, le nombre de décimales correctes est multiplié par environ 1.618 par la méthode de la sécante. En supposant initialement $b - a = 1$, et vérifiant les hypothèses du théorème, en 5 itérations, on a déjà 11 décimales correctes, et en 10 itérations, on a plus de 100 décimales ! La convergence est donc très rapide, beaucoup plus que la dichotomie.

Remarque 7.3.9

Si $f'(x) \neq 0$, le résultat précédente assure la convergence de la méthode de la sécante dès lors que l'initialisation se fait suffisamment proche du zéro x. En effet, on peut conrôler localement f' et f et restreindre l'intervalle initiale suffisamment.

Remarque 7.3.10 (Comparatif de la méthode de la sécante)

- Points forts :
  - Convergence très rapide (méthode d'ordre φ)
  - Facilité d'expression (ne nécessite pas la dérivée contrairement à la méthode de Newton)
- Points faibles :

* Instabilité : on n'est pas assuré de la convergence ; il faut se placer suffisamment près de la racine pour avoir la convergence. la distance initiale dépend d'un minorant de $|f'|$ et d'un majorant de $|f''|$.
* Condition d'arrêt mal assurée.

IV Méthode de Newton

La méthode de Newton est l'aboutissement. Lorsque la méthode de la sécante converge, les valeurs des c_i sont de plus en plus proches, et la corde est alors une bonne approximation de la tangente. La méthode de Newton consiste à remplacer dans l'algorithme de la sécante la corde par la tangente. On n'a alors plus besoin des deux bornes de l'intervalle (la seconde ne servait qu'à calculer la corde).

Méthode 7.4.1 (Méthode de Newton)

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.

Proposition 7.4.2 (Récurrence associée à la méthode de Newton)

On a alors, pour tout $n \in \mathbb{N}$, si la suite (x_n) est définie :

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

On obtient donc l'algorithme 7.4.

Exemple 7.4.3 (Méthode de Heron)

La méthode de Heron pour le calcul de $\sqrt{a}$ n'est rien d'autre que la méthode de Newton appliquée à la fonction $x \mapsto x^2 - a$. La relation de récurrence est alors :

$$x_{n+1} = \frac{1}{2}\left(x_n + \frac{a}{x_n}\right).$$

Avertissement 7.4.4

Comme pour la méthode de la sécante, la méthode de Newton peut être mal définie (impossible à itérer).

Pour assurer la convergence, il faut donc se placer, comme pour la méthode de la sécante, suffisamment près du zéro recherché.

Théorème 7.4.5 (Convergence locale de la méthode de Newton)

Soit f de classe $\mathcal{C}^2$ s'annulant en x. Supposons $|f'| \geqslant m_1$ et $|f''| \leqslant M_2$ sur $B(x,\delta)$, et $x_0 \in B(x,\delta)$ tel que $|x - x_0|\frac{M_2}{m_1} < 1$. Alors :
- *pour tout $n \in \mathbb{N}$, $|x_{n+1} - x| \leqslant |x_n - x|^2 M_2 m_1$*
- *il existe $a \in]0,1[$ tel que pour tout n assez grand, $|x_n - x| \leqslant a^{2^n}$.*

En particulier, la méthode de Newton est d'ordre 2 : on double le nombre de décimales correctes à chaque itération.

La méthode de Newton assure donc une convergence très rapide, meilleure que la méthode de la sécante.

Remarque 7.4.6 (Comparatif de la méthode de Newton)

- Points forts :
  - Convergence extrêmement rapide
- Points faibles :
  - Mêmes problèmes d'instabilité que la méthode de la sécante : il faut initialiser près du zéro pour être assuré de la convergence.
  - Utilisation de la dérivée de f, nécessitant d'être fournie par l'utilisateur, ou calculée numériquement (mais cela n'est satisfaisant que pour une fonction n'ayant pas trop de microvariations).

V Le problème de la dérivation numérique (HP)

La méthode de Newton nécessitant l'utilisation d'une dérivation numérique, on étudie maintenant une méthode de dérivation numérique, en essayant d'optimiser l'erreur.

Méthode 7.5.1 (Dérivation numérique)

On approche $f'(x)$ par $\frac{f(x+h)-f(x-h)}{2h}$, où h est suffisamment petit.

Proposition 7.5.2

L'erreur d'approximation théorique est de l'ordre de h^2, et l'erreur d'arrondi de l'ordre de $\frac{|f(x)|}{|h|}r$, où r est l'erreur élémentaire.

Corollaire 7.5.3 (Valeur optimale de h)

Sous l'hypothèse que f et des premières dérivées sont de l'ordre de grandeur de 1 (ni trop petit, ni trop gros) la valeur optimale de h minimisant l'erreur entre $f'(x)$ et la valeur calculée $\frac{f(x+h)-f(x-h)}{2h}$ est de l'ordre de $\sqrt[3]{r}$, où r est l'erreur élémentaire.

En effet, on peut montrer à l'aide de la formule de Taylor-Young, appliquée entre x et $x+h$, puis entre x et $x-h$, que $\frac{f(x+h)-f(x-h)}{2h}$ diffère de $f'(x)$ d'un terme de l'ordre de grandeur de αh^2. De plus, les grandeurs qui interviennent dans ces calculs sont de l'ordre de grandeur de $\frac{f(x)}{h}$, donc de $\frac{1}{h}$. L'erreur d'arrondi sur ces termes sera donc de l'ordre de $\frac{r}{h}$. Ainsi, en sommant l'erreur d'arrondi et l'erreur théorique, on a une erreur de l'ordre de $\alpha h^2 + \frac{r}{h^2}$. En minimisant cette fonction en h (par une étude de fonction), on se rend compte qu'elle admet un minimum en une quantité $\beta\sqrt[3]{r}$.

Les ordres de grandeur de f et ses dérivées intervenant dans les hypothèses (qui restent très vagues, comme le résultat lui-même) étant pris à la racine cubique pour obtenir l'ordre de grandeur du h optimal, cela laisse tout de même une bonne marge possible : un facteur 1000 sur l'ordre de grandeur de f induit un facteur 10 sur sur l'ordre de grandeur de h.

Par exemple, en norme IEEE 754, avec une erreur élémentaire r comprise entre 10^{-15} et 10^{-18}, on obtient une valeur optimale de h autour de 10^{-5} à 10^{-6}.
Dans la pratique, on pourra prendre $h = 10^{-5}$ ou 10^{-6}.

Algorithme 7.2 : Regula falsi

Entrée : f : fonction ;
$a < b$: initialisation telle que $f(a)f(b) < 0$;
ε : marge d'erreur
Sortie : x : zéro de f dans $[a, b]$

repéter
  $c \leftarrow a - \frac{(b-a)f(a)}{f(b)-f(a)}$;
  si $f(a)f(c) < 0$ **alors**
    $b \leftarrow c$
  sinon
    $a \leftarrow c$
  fin si
jusqu'à ce que *f change de signe sur* $[c - \varepsilon, c + \varepsilon]$;
renvoyer (c)

Algorithme 7.3 : Méthode de la sécante

Entrée : f : fonction ;
$a < b$: intervalle initial ;
ε : marge d'erreur
Sortie : x : zéro de f

repéter
  $c \leftarrow a - \frac{(b-a)f(a)}{f(b)-f(a)}$;
  $a, b \leftarrow b, c$
jusqu'à ce que *f change de signe sur* $[c - \varepsilon, c + \varepsilon]$;
renvoyer (c)

Algorithme 7.4 : Méthode de Newton

Entrée : f : fonction ; f' : dérivée de f ;
a : valeur initiale ;
ε : marge d'erreur
Sortie : x : zéro de f

repéter
  $a \leftarrow a - \frac{f(a)}{f'(a)}$
jusqu'à ce que *f change de signe sur* $[a - \varepsilon, a + \varepsilon]$;
renvoyer (c)

8

Résolution numérique d'équations différentielles

Le but de ce chapitre est de répondre au problème suivant : étant donnée une équation différentielle écrite sous la forme

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

et une initialisation $u(x_0) = y_0$, calculer une approximation de la fonction u sur l'intervalle I.
On admet le théorème de Cauchy-Lipschitz donnant l'unicité de la solution à ce problème sous certaines conditions, qu'on supposera réunies.

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 pourra 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 x. Dans ce cas, on fait de même 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 $x_k = a + \frac{k}{n}(b - a)$, pour $k \in [\![0, n]\!]$. On cherche ensuite des approximations des $u(x_k)$.
Évidemment, tout ce qu'on va faire pourrait être fait avec des subdivisions non régulières, mais pour simplifier l'approche, on considérera toujours la subdivision régulière.

Notre but est de donner plusieurs méthodes d'approximation des valeurs $u(x_k)$, et d'étudier, pour la plus simple des méthodes, la convergence des valeurs approchées vers les valeurs exactes lorsque n tend vers l'infini. Le problème de la convergence 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 des algorithmes donnant une erreur acceptable, en un temps raisonnable
- Contrôler précisément l'erreur
- Trouver un compris entre temps de calcul et précision.

Nous ne prétendons pas répondre exhaustivement à ces questions dans ce cours, dont le point de vue se veut essentiellement descriptif et intuitif. Par exemple, le contrôle précis de l'erreur est un problème dépassant les techniques dont nous disposons.

Enfin, remarquons que même si on expose les méthodes pour $a < b$ (donc pour des valeurs de x supérieures à la valeur d'initialisation), tout ce qu'on dit est valable aussi lorsque $b < a$, en considérant l'intervalle $[b, a]$. Dans ce cas, le pas est négatif, et on progresse vers l'arrière.

I Méthode d'Euler

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 8.1. 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. Vous illustrerez ces problèmes de divergence en TP.

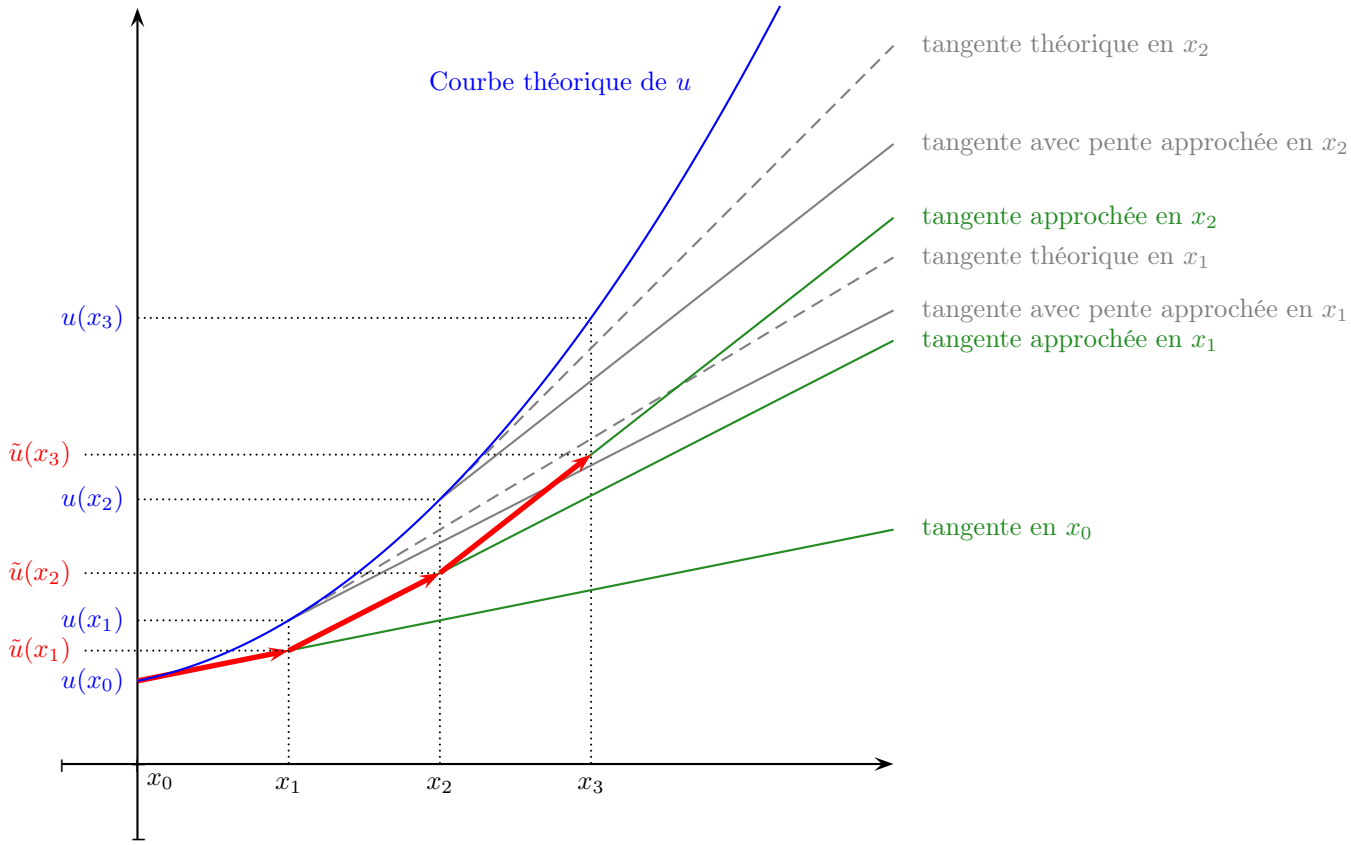


FIGURE 8.1 – Méthode d'Euler

Méthode 8.1.1 (Méthode d'Euler)

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

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

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

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

Proposition 8.1.2 (Majoration de l'erreur pour le calcul de u_1)

Soit M_2 un majorant de u''. Alors

$$|u_1 - u(x_1)| \leqslant \frac{M}{2} h^2.$$

Proposition 8.1.3 (Inégalité de récurrence pour l'erreur)

Supposons que f est L-lipschitzienne par rapport à sa seconde variable, et notons h le pas de la subdivision. En notant, pour tout $k \in [\![0, n]\!]$, $e_k = |u(x_k) - u_k|$, on a :

$$\forall k \in [\![0, n]\!], \quad e_{k+1} \leqslant (hL + 1)e_k + \frac{M}{2} h^2.$$

Corollaire 8.1.4 (Majoration de l'erreur)

Supposons que f est L-lipschitzienne par rapport à sa seconde variable. En négligeant les erreurs d'arrondi dans les calculs, l'erreur faite sur le calcul de $u(x_k)$ vérifie :

$$\forall k \in \mathbb{N}, \quad e_k \leqslant \frac{Mh}{2L} \left((1 + hL)^k - 1 \right) \leqslant \mathrm{e}^{khL} \frac{Mh}{2L}.$$

Cette majoration reste valable même pour des valeurs de k dépassant n, si on poursuit la méthode d'Euler au-delà de la valeur b (mais dans ce cas, on a un contrôle exponentiel divergent, donc très mauvais)
Si on reste dans l'intervalle compact $[a, b]$, on obtient une majoration uniforme de l'erreur :

$$\forall k \in [\![0, n]\!], \quad e_k \leqslant \frac{Mh}{2L} \mathrm{e}^{(b-a)L}.$$

Nous étudions maintenant l'exemple particulier de l'équation différentielle $u' = au$, avec $u(0) = 1$. Évidemment, la solution théorique est bien connue, il s'agit de l'exponentielle : $x \mapsto e^{ax}$. La connaissance de la solution théorique va nous permettre de déterminer de façon plus précise l'erreur faite par la résolution par la méthode d'Euler, dans ce cas particulier. Notre but par l'étude de cet exemple est de montrer qu'en général, on ne peut pas avoir un meilleur contrôle en $+\infty$ (lorsqu'on continue la méthode au-delà du point b) que ce contrôle exponentiel obtenu dans le corollaire qui précède.
On considère ici le pas $h = \frac{1}{n}$. Implicitement, cela revient à considérer l'intervalle initial d'étude $[0, 1]$, puis à prolonger le calcul à l'infini.
On montre qu'alors, on a, pour tout $k \in \mathbb{N}$,

$$u_k = \left(1 + \frac{a}{n}\right)^k$$

On vérifie d'ailleurs la cohérence de ce résultat au point 1 correspondant à $k = n$, l'expression trouvée étant de limite e^a (calcul classique et facile).

L'erreur s'exprime alors de façon exacte :

$$e_k = \left| f\left(\frac{k}{n}\right) - u_k \right| = \left| \mathrm{e}^{a\frac{k}{n}} - \left(1 + \frac{a}{n}\right)^k \right| \underset{k\to+\infty}{\sim} \mathrm{e}^{a\frac{k}{n}}.$$

Ainsi, pour tout pas $\frac{1}{n}$ fixé, la méthode d'Euler prolongée à l'infini donne une divergence exponentielle sur cet exemple.

Remarque 8.1.5 (Méthode d'Euler pour f indépendante de u)

Si f ne dépend que de x, l'équation à résoudre est $y' = f(x)$. Il s'agit d'une primitivation, donc d'un calcul d'intégrale. La relation de récurrence sur les termes u_k est alors simplement

$$y_{k+1} = y_k + f(x_k)h.$$

On se rend compte que la méthode d'Euler appliquée à cette situation correspond très précisément à la méthode des rectangles pour le calcul des intégrales, dont on connait bien la lenteur et le manque d'efficacité...

II Notion d'ordre d'une méthode

De façon générale, lorsqu'on dispose d'une méthode de résolution approchée d'équations différentielles, on veut pouvoir estimer son efficacité, et en particulier l'erreur d'approximation faite par cette méthode. Les calculs exacts sont parfois durs à mener : majorer de façon exacte l'erreur dans le cas de la méthode particulièrement simple d'Euler a déjà demandé beaucoup de travail. Pour des méthodes plus sophistiquées, cela relève quasiment de l'exploit !

On opère alors une hypothèse simplificatrice pour le calcul de l'erreur, en négligeant l'erreur faite sur la pente lors du calcul de la tangente approchée. Ainsi, on calcule l'erreur comme si la pente pour la k-ième étape était la pente théorique. Notant e'_k l'erreur ainsi calculée au rang k, et f'_{k+1} l'erreur supplémentaire issue du calcul de u_{k+1} à partir de u_k, on a alors $e'_{k+1} = e'_k + f_{k+1}$. Ainsi,

$$e'_k = \sum_{i=1}^{k} f_i.$$

Cette formule dit simplement que pour obtenir l'erreur au rang k, on somme toutes les erreurs induites par chaque étape.

On fait la deuxième hypothèse simplificatrice, consistant à dire que l'erreur f_k induite par l'étape k est comparable à l'erreur qu'on ferait en reprenant comme point de départ $(x_k, u(x_k))$ au lieu de $(x_k, \tilde{u}(x_k))$. Cette hypothèse se comprend facilement dans le cadre de la méthode d'Euler (figure 8.2), à partir du moment où on a l'hypothèse sur les tangentes. Dans cette figure, les u'_k représentent les points qui seraient calculés successivement par la méthode d'Euler si on disposaient des valeurs exactes des pentes aux points x_i.

On définit alors l'erreur de consistance par :

Définition 8.2.1 (Erreur de consistance)

L'erreur de consistance d'une méthode de résolution d'ED est la quantité :

$$e(h) = \sum_{k=1}^{n} |u'_k - u(x_k)|,$$

où $h = \frac{b-a}{n}$, et u'_k est la valeur approchée qu'on obtiendrait par la méthode utilisée, en prenant comme point de départ $(x_k, y(x_k))$ (donc la valeur théorique).

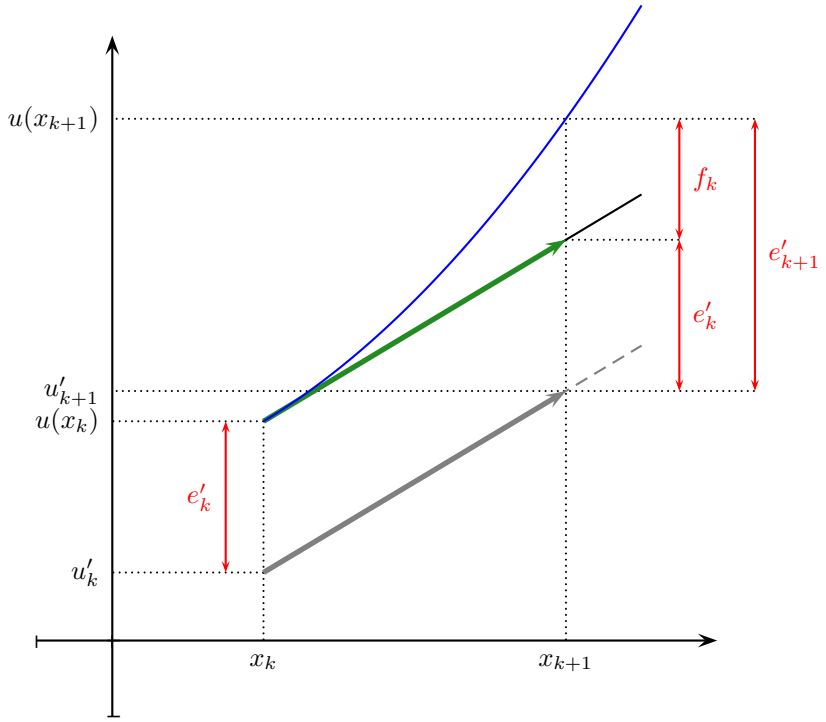


Figure 8.2 – Erreurs pour la méthode d'Euler

Définition 8.2.2 (Ordre d'une méthode)

L'ordre d'une méthode de résolution numérique d'ED est l'ordre de la puissance de h contrôlant l'erreur de consistance.

Exemple 8.2.3 (Ordre de la méthode d'Euler)

Le calcul de $|u_1 - u(t_1)|$ effectué plus haut pour la méthode d'Euler est valable plus généralement pour le calcul de $|u'_k - u(x_k)|$. On obtient alors l'erreur de consistance :

$$e(h) \leqslant \sum_{k=1}^{n} \frac{M_2}{2} h^2 = \frac{M_2(b-a)}{2} h.$$

Ainsi, la méthode d'Euler est d'ordre 1.

Remarques 8.2.4

1. La majoration de l'exemple précédent ne donne *stricto sensu* pas l'ordre de la méthode, mais seulement une minoration de l'ordre. Pour justifier que l'ordre est exactement 1, il faut montrer qu'on ne peut pas obtenir une meilleure convergence en exhibant un cas particulier dans lequel on sait calculer l'erreur de consistance de façon exacte. On pourrait reprendre l'exemple de l'exponentielle et calculer l'erreur de consistance dans ce cas particulier afin de conclure.
2. Très souvent, on peut obtenir une majoration uniforme $|u'_k - u(x_k)| \leqslant Mh^a$, valable sur tout l'intervalle $[a, b]$ (c'est-à-dire pour tout $k \in [\![1, n]\!]$). L'ordre de la méthode est dans ce cas (au moins) $a - 1$.

III Méthode de Runge-Kutta, HP

La méthode de Runge-Kutta a été mise au point au tout début du 20e siècle par Carl Runge et Martin Wilhelm Kutta, deux mathématiciens allemands.

L'idée est grossièrement la même que pour la méthode d'Euler : on subdivise l'intervalle d'étude, et on approche à chaque étape la courbe par une certaine droite, dont la pente est choisie judicieusement. Dans la méthode d'Euler, la pente choisie est la pente de la tangente au point initial de l'intervalle. La méthode de Runge-Kutta consiste à prendre comme pente une meilleure valeur approchée de la pente moyenne entre x_k et x_{k+1} : cette pente moyenne nous donnerait alors la valeur exacte de $u(x_{k+1})$ (si on part de la valeur exacte de $u(x_k)$). Plus on s'approche de cette pente, plus l'erreur va être petite
L'idée est alors de faire une moyenne des pentes en différents points répartis de façon déterminée sur l'intervalle $[x_k, x_{k+1}]$ (figure 8.3). Ces pentes en des points intermédiaires seront elles même calculées de façon approchée par une méthode de type Euler. Les différentes méthodes de Runge-Kutta correspondent alors au choix du nombre de points intermédiaires pour le calcul de la pente moyenne approchée, et au choix de leur répartition dans l'intervalle $[x_k, x_{k+1}]$.

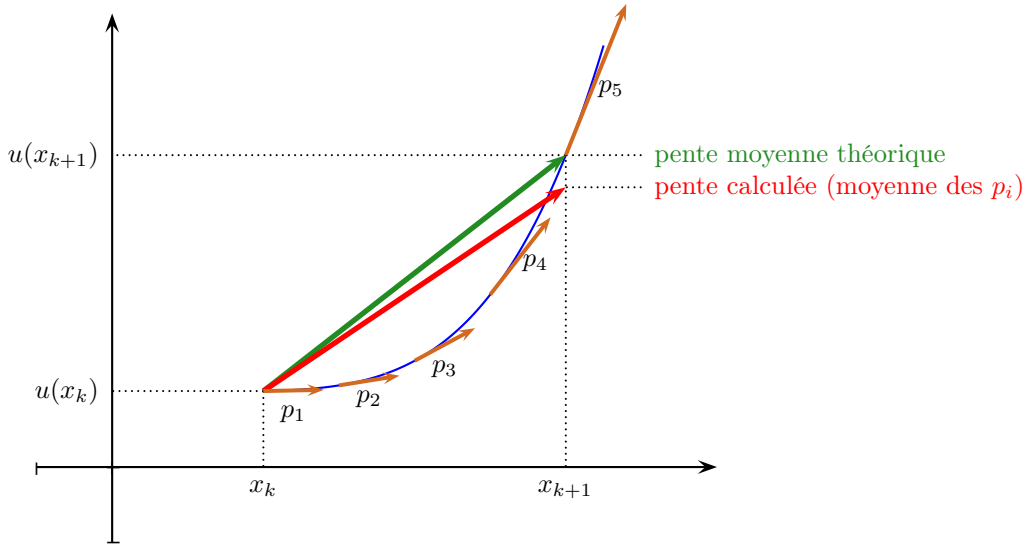


FIGURE 8.3 – Pente moyenne sur un intervalle $[x_k, x_{k+1}]$

Soit s un entier (correspondant au nombre de points intermédiaires pour le calcul de la moyenne des pentes). Soit $\beta_1 \leqslant \cdots \leqslant \beta_s$ des réels de $[0, 1]$ donnant la répartition des points $q_1, \ldots, q_s$ dans $[x_k, x_{k+1}]$, (les β_i correspondent à une paramétrisation linéaire par $[0, 1]$ de l'intervalle $[x_k, x_{k+1}]$). Ainsi :

$$q_i = x_k + \beta_i h,$$

où $h = x_{k+1} - x_k$ est le pas de la subdivision, supposée régulière. Soit ensuite $(\alpha_1, \ldots, \alpha_s)$ un s-uplet de réels de $[0, 1]$ tels que

$$\alpha_1 + \cdots + \alpha_s = 1.$$

On approche la pente moyenne $p = \dfrac{u(x_{k+1}) - u(x_k)}{h}$ sur l'intervalle $[x_{k+1}, x_k]$ par la moyenne pondérée par les α_i des pentes théoriques aux points q_i, à savoir :

$$p \simeq \sum_{i=1}^{s} \alpha_i f(q_i, u(q_i)).$$

Comme on ne connaît pas précisément $u(q_i)$, on utilise une approximation des $u(q_i)$, calculée suivant le même procédé que dans la méthode d'Euler, par approximations successives.

Méthode 8.3.1 (calcul d'une valeur approchée de la pente moyenne sur $[x_0, x_1]$)

Pour expliquer la construction, plaçons-nous sur l'intervalle $[x_0, x_1]$. Ainsi, $u(x_0) = y_0$ est connu. En notant p_i la pente calculée au point q_i, on procède de la sorte :

- On calcule p_1 par la méthode d'Euler en partant de x_k. On commence par déterminer une valeur approchée v_1 de $u(q_1)$:
$$v_1 = u(x_0) + (q_1 - x_0)f(x_0, u(x_0)).$$
On définit alors $p_1 = f(q_1, v_1)$
- On calcule p_2 en repartant du début de l'intervalle (en x_0), mais en approchant cette fois la courbe par la droite de pente p_1 calculé dans l'étape précédente, et issue du point $(x_0, u(x_0))$. Une approximation de $u(q_2)$ est alors :
$$v_2 = u(x_0) + (q_2 - x_0)p_1 \qquad \text{et} \qquad p_2 = f(q_2, v_2).$$
- De façon plus générale, p_k étant déterminé, p_{k+1} est calculé en repartant de x_0 et en approchant la courbe par une droite de pente p_k issue de $(x_0, u(x_0))$:
$$v_{k+1} = u(x_0) + (q_{k+1} - x_0)p_k \qquad \text{et} \qquad p_{k+1} = f(q_{k+1}, v_{k+1}).$$
- Les p_i étant tous déterminés ainsi, on approche p la pente moyenne sur $[x_0, x_1]$ (c'est-à-dire le taux d'accroissement) par
$$p \simeq \sum_{i=1}^{s} \alpha_i p_i.$$

Méthode 8.3.2 (Méthode de Runge-Kutta)

Les β_i (donc la répartition des q_i dans un intervalle $[x_k, x_{k+1}]$) et les poids α_i étant donnés, la méthode de Runge Kutta s'opère de la façon suivante : on calcule successivement des valeurs approchées u_k de $u(x_k)$. Supposant u_k déterminé, on détermine u_{k+1} ainsi :

- on calcule une approximation p de la pente sur $[x_k, x_{k+1}]$ comme expliqué ci-dessus, en considérant qu'on peut remplacer dans les calculs la valeur exacte $u(x_k)$ par la valeur approchée u_k.
- On calcule u_{k+1} en approchant f sur $[x_k, x_{k+1}]$ par la droite issue de (x_k, u_k), de pente p :
$$u_{k+1} = u_k + hp.$$

Nous explicitons ci-dessous les trois cas les plus fréquents (méthodes Runge-Kutta d'ordre 1, 2 et 4), obtenues pour certains choix de s, des répartitions β_i et des pondérations α_i.

Définition 8.3.3 (Méthode de Runge-Kutta d'ordre 1, RK1)

La méthode de Runge-Kutta d'ordre 1 consiste en le choix de $s = 1$ (un seul point pour obtenir l'approximation), $\alpha_1 = 1$ et $\beta_1 = 0$ (pour faire la moyenne, on considère la pente au bord gauche de l'intervalle)

On reconnait évidemment une méthode déjà étudiée !

Proposition 8.3.4 (RK1 = Euler)

La méthode de Runge-Kutta d'ordre 1 n'est autre que la méthode d'Euler. Cela justifie son ordre 1.

Définition 8.3.5 (Méthode de Runge-Kutta d'ordre 2, RK2)

La méthode de Runge-Kutta d'ordre 2 consiste en le choix de $s = 1$, $\alpha_1 = 1$ et $\beta_1 = \frac{1}{2}$.

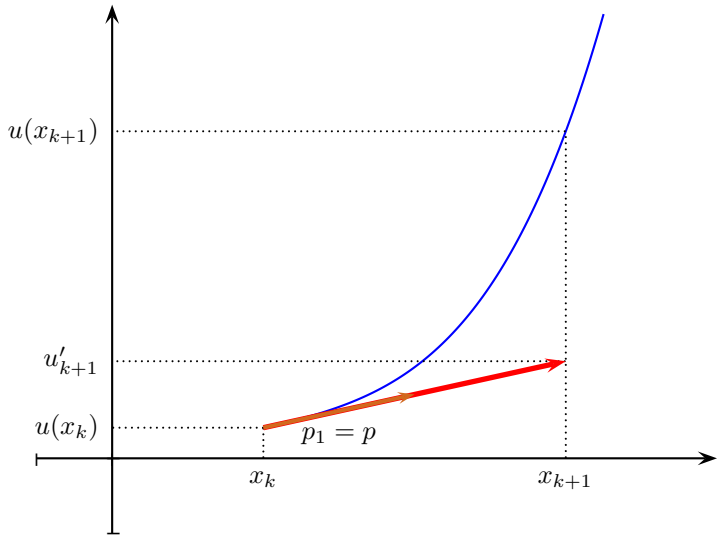


FIGURE 8.4 – Runge-Kutta d'ordre 1

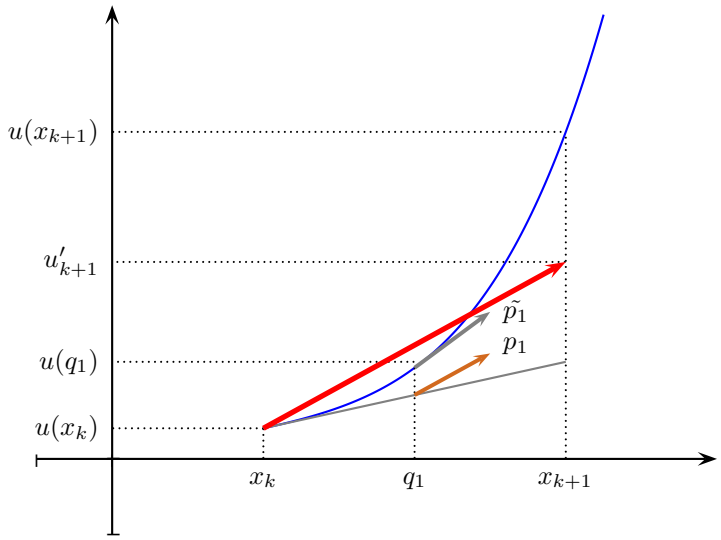


FIGURE 8.5 – Runge-Kutta d'ordre 2

Ainsi, dans cette méthode, la pente moyenne est approchée par la pente au point médian. Graphiquement, ce choix semble bien meilleur (figure 8.5). D'ailleurs, on a déjà rencontré une méthode d'analyse numérique dans laquelle le choix du point médian était judicieux. Les deux situations ne sont d'ailleurs pas sans rapport l'une avec l'autre :

Remarque 8.3.6 (RK2 et méthode du point milieu)

La méthode RK2 pour la résolution de $y' = f(x)$ n'est autre que le calcul intégral par la méthode du point milieu.

Méthode 8.3.7 (Calcul des approximations successives par RK2)

Suivant la description générale, u_k étant connu, on obtient u_{k+1} par la méthode RK2 de la façon suivante :

$$v = u_k + \frac{h}{2} f(x_k, u_k) \qquad p = f(x_k + \frac{h}{2}, v) \qquad y_{k+1} = y_k + hp.$$

Théorème 8.3.8 (ordre de RK2)

Comme son nom l'indique, la méthode RK2 est une méthode d'ordre 2.

On présente maintenant l'algorithme de Runge-Kutta classique, appelé algorithme de Runge-Kutta d'ordre 4, pour une raison que je vous laisse deviner. C'est l'algorithme de résolution numérique d'ED le plus fréquemment employé, du fait qu'il allie simplicité de programmation (une fois qu'on a compris la méthode) et rapidité de convergence. Cette rapidité de convergence est aussi ce qui assure la fiabilité du calcul, puisqu'elle permet d'éviter une trop grande accumulation d'erreurs d'arrondi.

Définition 8.3.9 (Méthode de Runge-Kutta d'ordre 4, RK4)

La méthode de Runge-Kutta d'ordre 4 consiste en le choix de $s = 4$, de la répartition $(\beta_1, \beta_2, \beta_3, \beta_4) = (0, \frac{1}{2}, \frac{1}{2}, 1)$, et des pondérations $(\alpha_1, \alpha_2, \alpha_3, \alpha_4) = \left(\frac{1}{6}, \frac{2}{6}, \frac{2}{6}, \frac{1}{6}\right)$.

Le fait de choisir deux fois le même point n'est pas redondant : en notant q ce point milieu, on commence par trouver une première approximation de la valeur de $u(q)$, donc une première approximation de la pente p en q, puis, on en recalcule une nouvelle approximation (qu'on espère meilleure, ou au moins compensant l'erreur précédente) en utilisant cette fois pour le calcul la nouvelle pente choisie (voir figure 8.6).

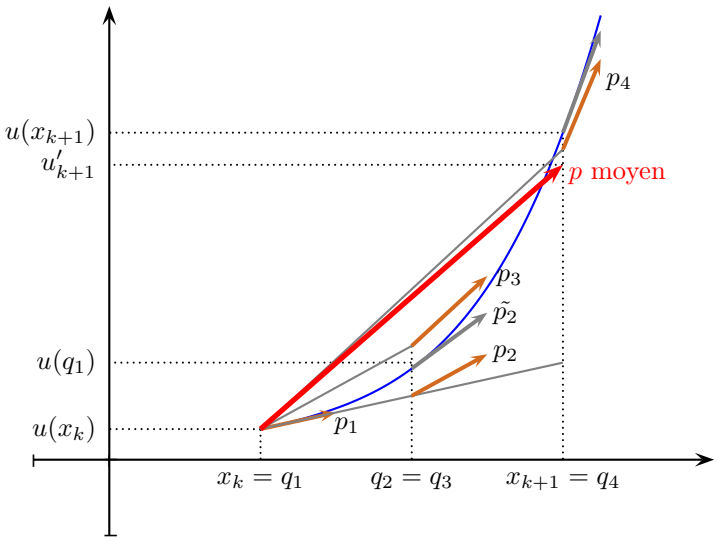


FIGURE 8.6 – Runge-Kutta d'ordre 4

Méthode 8.3.10 (Calcul des approximations successives par RK4)

Suivant la description générale, u_k étant connu, on obtient u_{k+1} par la méthode RK4 par l'enchaînement suivant de calculs :

- $v_1 = u_k$
- $p_1 = f(x_k, v_k)$
- $v_2 = u_k + \frac{h}{2}p_1$
- $p_2 = f(x_k + \frac{h}{2}, v_2)$
- $v_3 = u_k + \frac{h}{2}p_2$
- $p_3 = f(x_k + \frac{h}{2}, v_3)$
- $v_4 = u_k + hp_3$
- $p_4 = f(x_k + h, v_4)$

- $p = \frac{1}{6}(p_1 + 2p_2 + 2p_3 + p_4)$
- $u_{k+1} = u_k + ph$.

Le choix des pondérations $1, 2, 2, 1$, les deux points du milieu étant confondus, nous rappelle la pondération $1, 4, 1$ de la méthode de Simpson. Ici aussi, la méthode de Simpson peut être vue comme un cas particulier de la méthode RK4 :

Remarque 8.3.11 (RK4 et méthode de Simpson)

La méthode RK4 appliquée à la résolution de $y' = f(x)$ n'est autre que la méthode de Simpson.

Cette constatation peut nous faire espérer une très bonne convergence et efficacité de la méthode RK4 ! En effet, on obtient, sans suprise vu le nom donné à la méthode :

Théorème 8.3.12 (Ordre de la méthode RK4, admis)

La méthode RK4 est d'ordre 4.

9 Résolution numérique de systèmes linéaires

Le but de ce chapitre est l'étude de méthodes algorithmiques de résolutions de systèmes $AX = B$, où $A \in \mathcal{M}_{n,p}(\mathbb{R})$ et $B \in \mathcal{M}_{n,1}(\mathbb{R})$, l'inconnue X étant un élément de $\mathcal{M}_{p,1}(\mathbb{R})$. On peut bien sûr aussi travailler avec des coefficients complexes (en utilisant un langage manipulant des complexes, ce qui est le cas de Python), ou dans tout autre corps (si le langage connait les opérations dans ce corps, qu'elles soient implémentées initialement, ou qu'elles aient été programmées par la suite).
Savoir résoudre de tels systèmes est d'une importance capitale dans de nombreux domaines, par exemple en physique, en ingéniérie, ou encore dans toute l'imagerie vectorielle (l'image pixélisée est représentée par une matrice ; de nombreuses opérations sur les images nécessitent alors des résolutions de systèmes de taille importante). Dans ces domaines, les systèmes rencontrés sont souvent de très grande taille. Il est donc essentiel de savoir les résoudre de la façon la plus efficace possible. Nous nous contentons dans ce chapitre de rappeler la méthode du pivot, de tester l'aspect numérique, et d'étudier rapidement sa complexité. Nous introduisons également la décomposition $A = LU$ d'une matrice (Lower-Upper), donc en produit d'une matrice triangulaire inférieure et triangulaire supérieure : cela permet de ramener la résolution d'un système à la résolution de 2 systèmes triangulaires. Enfin, nous traitons de problèmes d'instabilité numériques, qu'on peut rencontrer dans le cas où le résultat d'un système est très sensible à une petite variation du second membre. Nous introduisons la notion de conditionnement, permettant de mesurer cette sensibilité, mais sans pour autant entrer dans une étude approfondie de cette notion.

I Structures de données adaptées en Python

Pour tout travail sur des données vectorielles et matricielles, nous utiliserons le module `numpy` de Python. Nous le supposerons importé sous l'alias `np`.
Le module `numpy` définit en particulier un type `array`, sur lequel sont définies certaines opérations matricielles usuelles. Nous renvoyons au chapitre 2 pour une description plus complète de ce module et des possibilités offertes. Nous renvoyons à l'aide de Python pour une description exhaustive.

L'instruction permettant de construire un tableau est `np.array`. Elle transforme une liste simple d'objets de même type numérique en tableau à une ligne.

```
>>> A = np.array([1,3,7,9])
>>> A
array([1, 3, 7, 9])
>>> type(A)
<class 'numpy.ndarray'>
>>> np.array([1,3,7,'a'])
```

```
array(['1', '3', '7', 'a'],
      dtype='<U1')
```

Le deuxième exemple ci-dessus montre la nécessité de l'homogénéité des éléments du tableau, qui doivent tous être de même type : les données numériques sont converties en chaînes de caractères (l'inverse n'étant pas possible). Le `dtype` indique un code pour le type des données (ici un texte en unicode)

Une liste de listes sera transformée en tableau bi-dimensionnel, chaque liste interne étant une des lignes de ce tableau. Il est évidemment nécessaire que toutes les listes internes aient la même taille. Dans le cas contraire, l'objet créé sera un tableau unidimensionnel, dont les attributs sont de type `list`. L'instruction `rank` donne la dimension spatiale du tableau (donc le degré d'imbrication) : 1 pour un tableau ligne, 2 pour une matrice, etc.

```
>>> A = np.array([[1,2,3],[3,6,7]])
>>> A
array([[1, 2, 3],
       [3, 6, 7]])
>>> type(A)
<class 'numpy.ndarray'>
>>> np.rank(A)
2
>>> B = np.array([[1,2,3],[3,6]])
>>> B
array([[1, 2, 3], [3, 6]], dtype=object)
>>> np.rank(B)
1
```

Le deuxième exemple illuste le fait que du point de vue de Python, B n'est pas bidimentionnel, mais est un tableau ligne, dont chaque donnée est une liste.

On peut bien sûr créer des objets de dimension plus grande :

```
>>> C = np.array([[[[1,2],[3,4]],[[5,6],[7,8]]],[[[9,10],[11,12]],[[13,14],[15,16]]]])
>>> C
array([[[[ 1, 2],
         [ 3, 4]],

        [[ 5, 6],
         [ 7, 8]]],


       [[[ 9, 10],
         [11, 12]],

        [[13, 14],
         [15, 16]]]])
>>> np.rank(C)
4
```

La dimension spatiale est suggérée par les sauts de ligne.
On peut récupérer le format (nombre de lignes, de colonnes, etc) grâce à la fonction `shape` :

```
>>> np.shape(A)
(2, 3)
>>> np.shape(B)
```

```
(2,)
>>> np.shape(C)
(2, 2, 2, 2)
```

Remarque 9.1.1 (tuple de taille 1)

Notez la syntaxe particulière pour le format de B. Cette syntaxe permet de différencier l'entier 2 du tuple $(2,)$ constitué d'une unique coordonnée. La virgule est nécessaire, car les règles de parenthésage ne permettent pas de différencier 2 et (2). Nous retrouverons cette syntaxe par la suite.

Comme les listes, les tableaux sont des objets mutables. Il faut donc faire attention à la dépendance possible entre plusieurs copies d'un tableau :

```
>>> D = A
>>> D
array([[1, 2, 3],
       [3, 6, 7]])
>>> A[1,1]= 8
>>> A
array([[1, 2, 3],
       [3, 8, 7]])
>>> D
array([[1, 2, 3],
       [3, 8, 7]])
```

On peut éviter ce désagrément en utilisant la fonction `np.copy`, équivalent de `deepcopy` pour les tableaux :

```
>>> D = np.copy(A)
>>> A[1,1]=10
>>> A
array([[ 1, 2, 3],
       [ 3, 10, 7]])
>>> D
array([[1, 2, 3],
       [3, 8, 7]])
```

Remarque 9.1.2 (Différences entre un tableau `numpy` et une liste)

- Toutes les entrées doivent être de même type
- Le format est immuable, et défini à la création du tableau : on ne peut pas modifier ce format en place (suppression ou ajout de ligne...), à moins de définir une nouvelle variable. En particulier, cela nécessite de définir dès le début un tableau de la bonne taille, par exemple rempli de 0 ou de 1. Il existe des fonctions permettant de le faire sans effort

```
# Création d'un tableau de 0
>>> np.zeros(5)
array([ 0., 0., 0., 0., 0.])
>>> np.zeros((5,3))
array([[ 0., 0., 0.],
       [ 0., 0., 0.],
       [ 0., 0., 0.],
       [ 0., 0., 0.],
```

```
      [ 0.,  0.,  0.]])

# Création d'un tableau de 1
>>> np.ones(3)
array([ 1.,  1.,  1.])
>>> np.ones((3,8))
array([[ 1.,  1.,  1.,  1.,  1.,  1.,  1.,  1.],
      [ 1.,  1.,  1.,  1.,  1.,  1.,  1.,  1.],
      [ 1.,  1.,  1.,  1.,  1.,  1.,  1.,  1.]])

# Création de la matrice identité I (prononcez à l'anglaise!)
>>> np.eye(3)
array([[ 1.,  0.,  0.],
      [ 0.,  1.,  0.],
      [ 0.,  0.,  1.]])

# Création d'une matrice diagonale:
>>> np.diag([1,3,2,7])
array([[1, 0, 0, 0],
      [0, 3, 0, 0],
      [0, 0, 2, 0],
      [0, 0, 0, 7]])

# Création d'une matrice (f(i,j,k,...))
>>> def f(i,j):
...     return i+j
...
>>> np.fromfunction(f,(3,4))
array([[ 0.,  1.,  2.,  3.],
      [ 1.,  2.,  3.,  4.],
      [ 2.,  3.,  4.,  5.]])
>>> np.fromfunction(lambda x: x**2 ,(6,))
array([ 0.,   1.,   4.,   9.,  16.,  25.])
```

Remarquez dans la dernière fonction l'utilisation du tuple $(6,)$ pour désigner le format d'un tableau ligne. On peut aussi créer des tableaux de valeurs uniformément espacées :

```
# Création d'un tableau de valeurs régulièrement espacées entre a et b
# en imposant le nombre de valeurs
# Attention au fait que a et b sont comptabilisés dans les n valeurs.
>>> np.linspace(5,9,10)
array([ 5.        ,  5.44444444, 5.88888889, 6.33333333, 6.77777778,
       7.22222222, 7.66666667, 8.11111111, 8.55555556, 9.        ])
>>> np.linspace(5,9,11)
array([ 5. , 5.4, 5.8, 6.2, 6.6, 7. , 7.4, 7.8, 8.2, 8.6, 9. ])

# Création d'un tableau de valeurs régulièrement espacées entre a et b
# en imposant le pas
# Attention au fait que la borne b est prise au sens strict.
>>> np.arange(5,10,0.5)
array([ 5. , 5.5, 6. , 6.5, 7. , 7.5, 8. , 8.5, 9. , 9.5])
>>> np.arange(5,10,0.7)
```

```
array([ 5. , 5.7, 6.4, 7.1, 7.8, 8.5, 9.2, 9.9])
```

De nombreuses opérations matricielles sont définies. Attention aux faux amis cependant. Comme on l'a vu, `np.rank` ne correspond pas au rang de la matrice, mais à sa dimension spatiale. Autre faux ami : le produit $A * B$ n'est pas le produit matriciel usuel, mais le produit de Schur (produit coordonnée par coordonnée). Le produit matriciel est donné par la fonction `np.dot`

```
>>> A = np.array([[1,2],[3,4]])
>>> B = np.array([[2,2],[0,3]])
# Produit de Schur
>>> A*B
array([[ 2, 4],
       [ 0, 12]])
# Produit matriciel
>>> np.dot(A,B)
array([[ 2, 8],
       [ 6, 18]])
```

Les fonctions les plus utiles sont décrites dans le chapitre 2. Retenons en particulier (puisque c'est ce qui nous intéresse dans ce chapitre) la fonction `np.linalg.solve` pour la résolution d'un système $AX = B$ (le vecteur B doit être donné sous forme d'un tableau ligne, ou d'une liste)

```
>>> np.linalg.solve(np.array([[1,3],[2,-3]]),[1,4])
array([ 1.66666667, -0.22222222])
```

Cette fonction ne résout que les systèmes de Cramer :

```
>>> np.linalg.solve(np.array([[1,3],[2,6]]),[1,2])
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "/usr/lib64/python3.3/site-packages/numpy/linalg/linalg.py", line 328, in solve
    raise LinAlgError('Singular matrix')
numpy.linalg.linalg.LinAlgError: Singular matrix
```

II Rappels sur la méthode du pivot de Gauss

La méthode du pivot est une méthode d'échelonnement d'une matrice donnée, par opérations admissibles (c'est-à-dire reversibles) sur les lignes. Ces opérations préservent le noyau, donc les solutions du système homogène. Si on effectue les mêmes opérations sur la matrice colonne B du second membre, on obtient le même espace affine de solutions.

On rappelle que les opérations admissibles sont :
- $L_i \leftrightarrow L_j$: l'échange de deux lignes (permutation)
- $L_i \leftarrow \lambda L_i$: la multiplication d'une ligne par un scalaire **non nul** (dilatation)
- $L_i \leftarrow L_i + \lambda L_j$: l'ajout à une ligne d'une autre, multipliée par un scalaire (transvection).

On rappelle que ces opérations correspondent matriciellement à la multiplication à gauche par certaines matrices de codage (voir le cours de mathématiques).

On rappelle les grandes lignes de l'algorithme d'échelonnement de Gauss :

Méthode 9.2.1 (Échelonnement par la méthode du pivot)
- On cherche un élément non nul dans la première colonne ;

- **Le choix du pivot est important**. Pour des raisons de précision numérique, il est judicieux de choisir le pivot de **valeur absolue maximale.** On fera aussi attention au test à 0, à faire sous forme d'une inégalité, et non d'une égalité stricte, si on ne veut pas avoir de surprises désagréables.
  - Si on n'en trouve pas, on passe à la colonne suivante, et on recommence de même
  - Si on en trouve, on le positionne en haut par un échange. C'est notre premier pivot. On annule tous les autres coefficients de la première colonne à l'aide de celui-ci, par des opérations de transvection, puis on passe à la colonne suivante, mais en ne touchant plus à la première ligne (contenant le premier pivot). En particulier, la recherche d'un pivot suivant ne se fera pas sur la première ligne, et ce pivot ne sera remonté que sur la deuxième ligne.
- On continue de la sorte jusqu'à épuisement des colonnes de A. Les différents pivots se positionnent sur les lignes successives.
- Quitte à faire des opérations de dilatation, on peut s'arranger pour qu'à l'issue du calcul, tous les pivots soit égaux à 1.

Méthode 9.2.2 (Pivot remontant)

- Quitte à conserver en mémoire la liste des indices de colonne des pivots, on retrouve facilement la position et la valeur des pivots. On peut alors, par des opérations de tranvection, annuler tous les coefficients situés au dessus des pivots, de sorte à ce que les pivots soient les seuls composantes non nuls de leurs colonnes.
- On fera attention a être économe en calcul pour cette étape : commencer par le dernier pivot évite de manipuler trop de termes non nuls (et donc d'accumuler des erreurs sur des termes qui nous intéressent). Par ailleurs, il est inutile de faire les opérations sur les colonnes précédant le pivot (opérations triviales, mais qui prennent du temps informatiquement parlant).

Dans le cas de la résolution d'un système $AX = B$, en faisant en parallèle les mêmes opérations sur B, on obtient donc un système équivalent $A'X = B'$, où A' est échelonnée, de pivots tous égaux à 1, les éléments situés en-dessous, et au-dessus des pivots étant nuls. Notons $j_1 < j_2 < \cdots < j_r$ les colonnes des r pivots (on remarquera que r est le rang de A). On obtient alors facilement une solution particulière, ainsi qu'une base de l'espace des solutions de l'équation homogène. On note $B' = \begin{pmatrix} b_1 \\ \vdots \\ b_n \end{pmatrix}$.

Méthode 9.2.3

- Si le système est compatible (c'est le cas ssi $b_i = 0$ pour tout $i > r$), une solution particulière est le vecteur $X_0 = (x_i)_{1 \leqslant i \leqslant p}$, tel que

$$x_i = \begin{cases} b_k & \text{si } i = j_k,\ k \in [\![1, r]\!] \\ 0 & \text{sinon} \end{cases}$$

- Une base est obtenue en énumérant les vecteurs X_j obtenus, pour $j \in [\![1, p]\!] \setminus \{j_1, \ldots, j_r\}$ par la construction suivante :
  - les coordonnées de X_j distinctes de j et des j_k sont nulles
  - la j-ième coordonnée de X_j est égale à 1
  - la colonne extraite de X_j en ne conservant que les lignes $j_1, \ldots, j_r$ est égale à $-C_j$, où C_j est la j-ième colonne de A', tronquée au-delà de la r-ième coordonnée.

Théorème 9.2.4 (Complexité de la méthode du pivot)

La méthode du pivot sur une matrice de $\mathcal{M}(n, p)$ est en $O(np^2)$. En particulier, elle est cubique sur des

matrices carrées.

Comme on l'a vu en cours de mathématiques, la méthode du pivot peut être adaptée de différentes manières :

Méthode 9.2.5 (Adaptations de la méthode du pivot)

- **Calcul du rang :** on se contente d'un premier passage (échelonnement simple), en comptant le nombre de pivots utilisés, ce qui donnera le rang.
- **Calcul de l'inverse :** on effectue un échelonnement double, en remarquant que dans la première étape (descente), si on ne trouve pas de pivot admissible sur une colonne, la matrice n'est pas inversible (on obtiendra au bout une matrice triangulaire dont certains coefficients diagonaux sont nuls) : on peut donc interrompre l'algorithme dans ce cas. L'échelonnement double de la matrice, avec normalisation des pivots, nous donne alors, en cas d'inversibilité, la matrice I_n. On effectue en parallèle les mêmes opérations sur la matrice initiale I_n, on obtient en fin d'algorithme la matrice A^{-1}.
- **Calcul de la matrice de passage** de l'échelonnement, c'est-à-dire de P telle que $PA = A'$: c'est une généralisation de la méthode de calcul de l'inverse, en suivant exactement le même principe : P est le produit des matrices de codage des opérations, donc obtenu en effectuant sur I_n les opérations correspondantes. On trouve donc P en effectuant en parallèle sur I_n les mêmes opérations que sur A pour obtenir A'.

III Décomposition LU

Définition 9.3.1 (Décomposition LU)

Une décomposition LU d'une matrice carrée A est une décomposition de A en un produit $A = LU$ d'une matrice triangulaire inférieure L (Lower) et d'une matrice triangulaire supérieure (Upper).

On peut obtenir, sous certaines conditions, une décomposition LU en adaptant la dernière variante de la méthode du pivot exposée ci-dessus. Au lieu de rechercher P tel que $PA = A'$, on recherche $Q \in \mathrm{GL}_n(\mathbb{K})$ tel que $A = QA'$. On pourrait se dire qu'il suffit d'inverser P, mais cela oblige à faire 2 pivots (un premier pour réduire A, l'autre pour inverser P), ce n'est pas très optimal. On adapte alors la méthode précédente, en remarquant que si $P = P_k \cdots P_1$, où les P_i codent les opérations successives, alors

$$Q = P_1^{-1} \cdots P_k^{-1} = I_n P_1^{-1} \cdots P_k^{-1}.$$

Méthode 9.3.2 (Trouver Q telle que $A = QA'$)

On trouve Q en effectuant sur I_n les opérations sur les colonnes correspondant aux matrices inverses des matrices de codage des opérations sur les lignes effectuées sur A pour obtenir A'. Plus explicitement, à chaque opération sur les lignes de A, on effetue une opération correspondante sur les colonnes de I_n :

- $L_i \leftarrow \lambda L_i$ correspond à $C_i \leftarrow \frac{1}{\lambda} C_i$
- $L_i \leftrightarrow L_j$ correspond à $C_i \leftrightarrow C_j$
- $L_i \leftarrow L_i + \lambda L_j$ correspond à $C_j \leftarrow C_j - \lambda C_i$.

On obtient alors une décomposition LU en remarquant que beaucoup de matrices d'opérations sont triangulaires : il suffit donc de se limiter aux opérations pour lesquelles la matrice associée (donc son inverse) est triangulaire inférieure : c'est le cas des opérations de dilatation et de transvection $L_i \leftarrow L_i + \lambda L_j$ lorsque $i > j$. Or, s'il n'y a pas besoin de faire d'échanges de lignes, le pivot de Gauss peut s'effectuer avec ces seules opérations. De plus, les opérations de dilatation ne sont pas strictement nécessaires, et les

matrices triangulaires en jeu ont alors toutes une diagonale de 1. On obtient alors, sous une hypothèse nous assurant qu'on n'aura pas d'échange de ligne à faire :

Théorème 9.3.3 (Décomposition LU)

Soit $A \in \mathrm{GL}_n(\mathbb{R})$, telle que pour tout $k \in [\![1, n]\!]$, la sous-matrice carrées obtenue par extraction des k premières lignes et des k premières colonnes soit inversible. Alors A admet une décomposition LU. On peut de plus imposer que L soit à diagonale égale à 1.

Cette décomposition se trouve explicitement par la méthode suivante :

Méthode 9.3.4 (Trouver une décomposition LU)

Réduire A en une matrice échelonnée U, en n'utilisant ni échange de ligne, ni dilatation (si on veut L de diagonale 1). Cette matrice échelonnée est triangulaire supérieure. On effectue sur I_n les opérations sur les colonnes correspondant aux opérations sur les lignes dans le sens décrit plus haut, cela fournit la matrice L.

Remarque 9.3.5 (Intérêt de la décomposition LU)

- Cette décomposition est intéressante notamment dans le cas où on a un grand nombre de systèmes $AX = B_i$ à résoudre, associés à la même matrice A. Plutôt que de refaire tout le calcul depuis le début pour chaque vecteur B_i, on fait le calcul de la décomposition une fois pour toutes : on est ramené à la résolution de 2 systèmes triangulaires $LUX = B$. On obtient d'abord UX par résolution du système de matrice L, puis X par résolution du système de matrice U.
- Il pourra être avantageux de programmer explicitement un algorithme spécifique de résolution des systèmes triangulaires inférieurs et supérieurs, afin d'optimiser au maximum les temps de calcul, en omettant les opérations avec les coefficients qu'on sait être nuls.
- Cette méthode est plus avantageuse que de calculer A^{-1} jusqu'au bout puis de faire les calculs de $A^{-1}B_i$. En effet, si les résolutions des systèmes linéaires triangulaires sont bien programmés, elles ne sont pas plus coûteuses que le calcul de $A^{-1}B_i$, cela permet donc d'éviter la phase finale du calcul de A^{-1}.

Cette méthode possède tout de même des inconvénient notables, comme pas exemple le fait que l'on ne peut pas choisir le pivot. Cela peut éventuellement diminuer la précision de calcul.

IV Problèmes de conditionnement

On part de l'exemple suivant :

$$H_n = \left(\frac{1}{i+j-1}\right)_{1 \leqslant i,j \leqslant n}.$$

Cette matrice est appelée matrice de Hilbert. On observe sur des exemples que la résolution de $H_n X = B$ est problématique :

```
>>> np.linalg.solve(H,[1,1,1,1])
array([ -4.,  60., -180., 140.])
>>> np.linalg.solve(H,[1.01,0.99,1.01,0.99])
array([ 1.16,  3.  , -43.8 , 51.8 ])
```

Cet exemple illustre la très grande variabilité du résultat en fonction du second membre B : une toute petite variation sur B fournit de très grosses variations sur la solution du système. Cela peut engendrer de grosses imprécisions sur les calculs, du fait que toute erreur d'arrondi sur B sera beaucoup amplifiée lors de la résolution du système.

Il n'y a pas grand chose à faire, sinon savoir détecter quelles sont les matrices qui vont nous donner des situations similaires, de sorte à pouvoir valider la précision des calculs dans les autres cas. Pour cela, on utilise une quantité permettant de mesurer l'amplification d'une variation sur B lors de la résolution du système.

Définition 9.4.1 (Norme matricielle)

La norme (euclidienne canonique) de $X \in \mathcal{M}_{n,1}(\mathbb{R})$ est définie par

$$\|X\| = \sqrt{x_1^2 + \cdots + x_n^2}, \text{ où } X = \begin{pmatrix} x_1 \\ \vdots \\ x_n \end{pmatrix}.$$

On définit alors la norme d'une matrice $M \in \mathcal{M}_n(\mathbb{R})$ par :

$$\|A\| = \sup_{X \in \mathbb{R}^n \setminus \{0\}} \frac{\|AX\|}{\|X\|}.$$

Cette quantité est bien définie du fait que

$$\sup_{X \in \mathbb{R}^n \setminus \{0\}} \frac{\|AX\|}{\|X\|} = \sup_{X \in B(0,1)} \|AX\|,$$

(provenant de l'invariance par dilatation de X), en utilisant la compacité de $B(0,1)$ et la continuité de la norme d'un vecteur de $\mathbb{R}^n$.

Proposition 9.4.2

$A \mapsto \|A\|$ est une norme sur $\mathcal{M}_n(\mathbb{R})$:

(i) $\|A\| = 0$ si et seulement si $A = 0$

(ii) pour tout $\lambda \in \mathbb{R}$, $\|\lambda A\| = |\lambda| \cdot \|A\|$,

(iii) pour tout A, B, $\|A + B\| \leqslant \|A\| + \|B\|$ (inégalité triangulaire)

De plus, il s'agit d'une norme matricielle*, c'est-à-dire :*

(iv) pour tout $A, B \in \mathcal{M}_n(\mathbb{R})$, $\|AB\| \leqslant \|A\| \cdot \|B\|$.

Pour le dernier point, on pourra utiliser le lemme suivant :

Lemme 9.4.3

Soit $A \in \mathcal{M}_n(\mathbb{R})$ et $X \in \mathcal{M}_{n,1}(\mathbb{R})$. On a :

$$\|AX\| \leqslant \|A\| \cdot \|X\|.$$

On définit alors le conditionnement par :

Définition 9.4.4 (Conditionnement d'une matrice)

Soit $A \in \mathrm{GL}_n(\mathbb{R})$. Le conditionnement de A est :

$$\mathrm{cond}(A) = \|A\| \cdot \|A^{-1}\|.$$

Proposition 9.4.5

On a toujours $\text{cond}(A) \geqslant 1$.

Nous allons voir que le conditionnement permet de contrôler les variations de la solution X du système en fonction des variations de B. Plus $\text{cond}(A)$ est proche de 1, plus ce contrôle va être bon. Si $\text{cond}(A)$ est grand, ce contrôle va être mauvais ; C'est ce qu'il se passe pour les matrices de Hilbert (on peut montrer que $\text{cond}(H_4) = 15514$). Ainsi, un conditionnement proche de 1 est une garantie de fiabilité tdu résultat.

Théorème 9.4.6

Soit B et $B + \delta B$ deux colonnes de $\mathbb{R}^n$, et X et $X + \delta X$ les solutions obtenues pour le système de matrice A, associé à ces deux seconds membres. On a alors :

$$\frac{\|\delta X\|}{\|X\|} \leqslant \text{cond}(A)\frac{\|\delta B\|}{\|B\|}.$$

Ce théorème affirme que les variations relatives sur B sont amplifiées sur la solution du système par un facteur au plus égal à $\text{cond}(A)$. Assez logiquement, on ne peut pas espérer un meilleur contrôle de l'erreur sur X que sur B (car $\text{cond}(A) \geqslant 1$), et une grande valeur de $\text{cond}(A)$ nous donne un très mauvais contrôle.

Définition 9.4.7 (Bon et mauvais conditionnement)

Si $\text{cond}(A)$ n'est pas trop grand par rapport à 1, on dit que A est bien conditionné. Si A est grand par rapport à 1, on dit que A est mal conditionné.

C'est une notion qui reste très subjective, on ne quantifie pas la valeur seuil passant de l'une à l'autre des deux situations.

10 Bases de données relationnelles

Un des grands défi du XXI^{e} siècle, rendu possible par l'émergence d'internet à la fin du XX^{e} siècle, est de regrouper, d'ordonner, et de mettre à disposition un grand nombre de données, en vue d'une utilisation par des experts (études statistiques...) ou des particuliers (renseignement...). Cette immense mise en commun de données éparpillées nécessite des structures spéciales : outre la mémoire nécessaire pour le stockage des données, il faut aussi pouvoir structurer ces données de façon à pouvoir en extraire facilement ce qui nous intéresse : extraire par exemple d'une base de donnée musicale les oeuvres de compositeurs russes, ou les sonates pour flûte écrites entre 1700 et 1750. Il faut donc une structure permettant de dégager rapidement les principales caractéristiques d'une oeuvre (compositeur, date, pays, instruments, type d'oeuvre, etc).

La structure de base de donnée (BDD) relationnelle répond à cette question. Nous nous proposons dans ce chapitre d'étudier le façon intuitive cette structure de BDD relationnelle, avant d'en proposerons une formalisation algébrique. Nous verrons dans le chapitre suivant un langage adapté à la manipulation de ces bases de données (SQL), ainsi que les différentes opérations algébriques possibles en vue de faire des requêtes (extraire les informations précises qui nous intéresse).

Nous illustrons ce chapitre et le suivant par la création d'une base de donnée musicale.

I Environnement client / serveur

Le principe-même d'une base de données est que de nombreuses personnes peuvent y avoir accès. Cependant, il est peu souhaitable que tout le monde ait un accès direct à cette base, afin d'éviter les accidents de manipulation qui pourraient faire perdre des données de la base. Pour cette raison, on opère en général une séparation stricte entre la machine (ou machine virtuelle) sur laquelle est stockée la base, et les machines des utilisateurs. Les utilisateurs n'ont en fait accès à la base (et la machine qui la contient) que *via* des « requêtes » dont la syntaxe et l'inoffensivité sont bien contrôlées. Par ailleurs, les utilisateurs ont des droits bien établis : la plupart d'entre eux n'ont qu'un droit de consultation ; les utilisateurs ayant un droit d'ajout ou de modification sont en petit nombre et souvent supervisés par un superutilisateur unique (root).
Cette séparation nette entre la base elle-même et les utilisateurs conduit à la notion d'environnement client/serveur, ou à l'architecture 3-tiers qui en est une variante améliorée. Dans une telle structure, les logiciels sont scindés en 2 : une partie légère du coté du client, et une partie lourde (le traitement) du côté du serveur.

I.1 Le serveur informatique

Définition 10.1.1 (Serveur informatique)
Le serveur informatique est un dispositif informatique offrant des services à des clients.

Ces services peuvent être de différents types suivant la nature du serveur :
- partage de fichiers ;
- partages de données (BDD) ;
- hébergeurs web : accès au Web, stockage de pages webs
- serveurs de courrier électronique : stockage et envoi de messages
- réseaux locaux : mise en commun au sein d'une entreprise des ressources informatiques, que ce soit matérielles (imprimantes...) ou logicielles (cela évite d'avoir à intaller les logiciels lourds sur toutes les machines)
- serveur local à un ordinateur : gère les différents utilisateurs de l'ordinateur
- cloud computing : offre aux clients une grande puissance de calcul. Cela permet de mutualiser la puissance : plutôt que d'augmenter individuellement leur capacité de calcul par l'achat de matériel neuf, les clients payent le droit de se servir des ressources calculatoires mises à disposition par un fournisseur

Un serveur est en communication avec plusieurs clients généralement un grand nombre (figure 10.1). La communication entre le serveur et les clients se fait suivant un certain protocole de communication strict et sécurisé, dans la mesure où la sécurité et l'intégrité du serveur sont bien assurées par ce protocole.

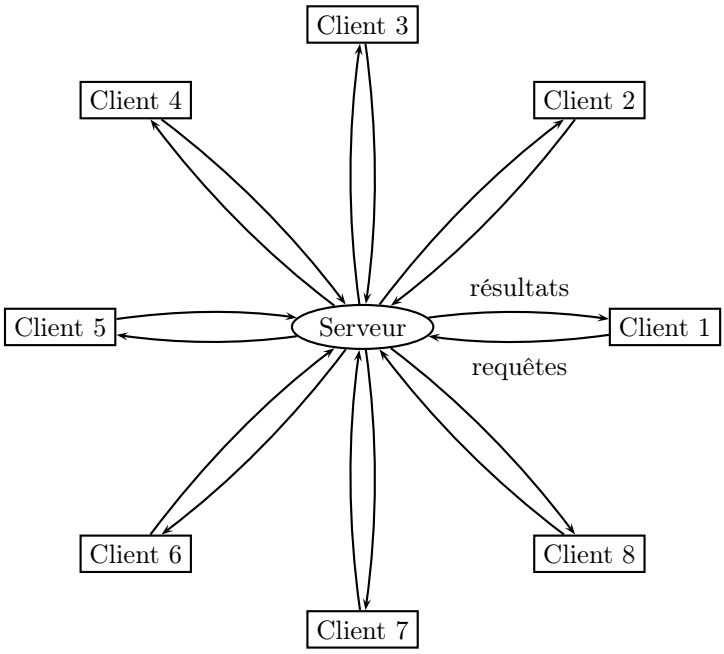


FIGURE 10.1 – Architecture client-serveur

Les caractéristiques démandées à un serveur sont :
- la rapidité de communication (débit)
- la rapidité de traitement
- la capacité d'adaptation à la demande (augmentation de la taille en mémoire, du nombre de requêtes simultanées...)
- la disponibilité 24h sur 24.

Les serveurs sont donc très souvent des machines beaucoup plus puissantes que la moyenne, et très fiables (afin d'éviter les pannes matérielles).

I.2 Le client

Définition 10.1.2 (Client)
Le client est un matériel logiciel ou informatique, permettant l'envoi de requêtes à un serveur donné, et la réception des réponses.

Il s'agit donc d'une interface légère gérant de façon conviviale la communication avec un serveur, traduisant les requêtes de l'utilisateur de façon adéquate.

Les logiciels associés à une architecture client-serveur sont organisés en trois couches :
- la gestion du stockage de l'information
- le traitement des requêtes, donc la partie calculatoire
- le client : l'interface avec l'utilisateur (entrée des requêtes, affichage des résultats

La première est essentiellement destinée au maintien et à la mise à jour du serveur. La seconde est souvent la partie lourde du logiciel ; quant à la troisième, il ne s'agit que d'une interface légère de communication. Les deux premières couches sont installées sur le serveur lui-même, alors que la troisième couche (le client) est installée chez tous les utilisateurs. Cela présente une économie logicielle (seule une interface légère est installée en plusieurs exemplaires), ainsi qu'une sécurité, puisque l'utilisateur n'a pas un accès physique direct au serveur, mais ne peut communiquer avec lui que par l'intermédiaire de l'interface client, qui ne transmet que des requêtes inoffensives.

I.3 Architecture 3-tiers

Définition 10.1.3 (Architecture 3-tiers)
Il s'agit d'une évolution de l'architecture client-serveur, dans laquelle les trois couches logicielles exposées ci-dessus sont physiquement séparées. Ainsi, le serveur lui-même est scindé en deux serveurs :
- serveur 1 : stockage de l'information
- serveur 2 : serveur métier, ie serveur effectuant la gestion logicielle

La terminologie provient de l'anglais « tier » signifiant « couche ».

Le client communique avec le serveur métier qui lui-même communique avec le serveur de stockage (figure 10.2) : le client est davantage séparé du serveur de stockage, ce qui minimise les risques de pertes de données suite à des erreurs (involontaires ou malveillantes) de manipulation.

Les bases de données actuelles sont contruites sur ce schéma (figure 10.3). Le serveur métier s'appelle le système de gestion de base de donnée (SGBD).

On peut voir plusieurs intérêts à une telle dissociation :
- dissociation BDD-SGBD : il n'y a pas de communication directe entre le client et la base : la passage obligatoire par le serveur de gestion, contrôlé rigoureusement par un superutilisateur, est une garantie suplémentaire de sécurité pour la base. Ce superutilisateur (administrateur de la base de donnée) est en théorie l'unique personne pouvant modifier la base. C'est elle qui attribue des droits aux clients (droit de consultation, de modification, d'ajout...)
- dissociation client-BDD : outre la sécurité, cette dissociation permet un enrichissement de la base indépendante des utilisateurs.
- dissociation client-SGBD : de même, les logiciels de gestion peuvent être actualisés indépendamment du client : ainsi, une amélioration logicielle peut se faire uniquement au niveau du serveur, et ne nécessite pas une actualisation au niveau de chaque client. Par ailleurs, seule une interface légère est nécessaire du côté client, ce qui ne nécessite pas une machine particulièrement performante.

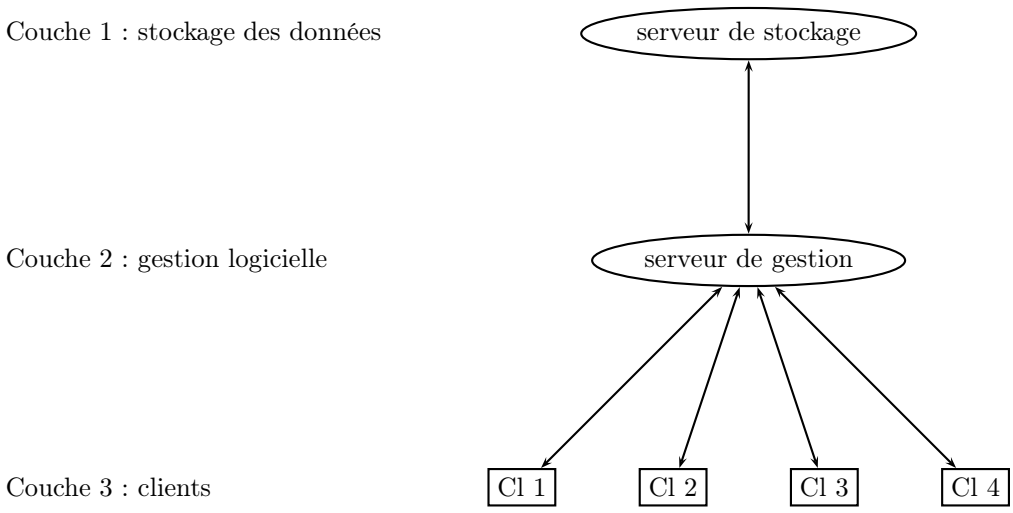


FIGURE 10.2 – Architecture 3-tiers

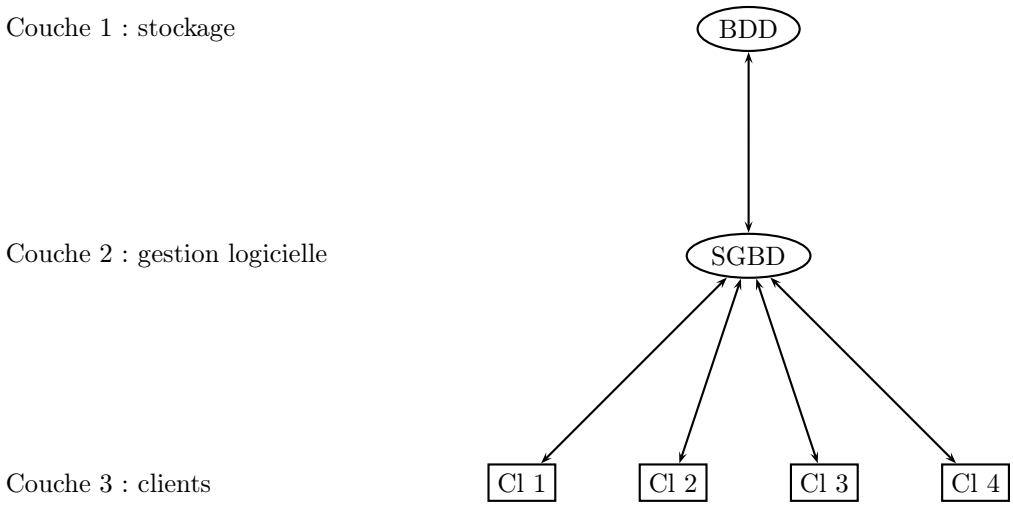


FIGURE 10.3 – Architecture 3-tiers pour une BDD

I.4 Une autre architecture

Il existe d'autres architectures de réseaux. Citons par exemple l'architecture pair-à-pair. Dans cette architecture, le serveur ne sert par au stockage : les différentes données sont stockées au niveau des clients. Le serveur gère alors la possibilité pour chaque client d'accéder au contenu partagé des autres clients.

II Bases de données

II.1 Présentation intuitive

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.

Imaginons par exemple que nous souhaitions faire une base de donnée musicale avec les types de données suivantes :

- Compositeur
- Dates de naissance et de mort
- Pays
- Oeuvre
- Date de composition
- Type d'oeuvre (symphonie, concerto...)
- instrumentation (orchestre, choeur, soliste, ensemble de chambre...)
- Instruments solistes
- Famille des instruments solistes (vents, cordes...)

La première idée pouvant venir à l'esprit est une présentation linéaire sous forme d'un tableau (figure 10.4).
On se rend compte assez rapidement que cette présentation assez simpliste présente un certain nombre d'inconvénients, comme par exemple les redondances (on indique les dates et pays des compositeurs autant de fois qu'il y a d'oeuvre d'eux dans la base !), ainsi que les entrées multiples (dans l'instrumentation ou les solistes par exemple), qui compliquent les recherches.

On peut alors penser à séparer les données en plusieurs tableaux. Par exemple un tableau des compositeurs, un tableau des oeuvres et un tableau des instruments (figure 10.5)
Les redondances ne sont pas complètement supprimées : par exemple indiquer dans l'instrumentation qu'il y a des solistes est redandant avec la colonne soliste, qui n'est remplie que dans ce cas. On peut supprimer cette information. Par ailleurs, il reste le problème des entrées multiples. Ce problème-là n'est pas très dur à régler : il suffit de dupliquer la ligne autant de fois qu'il y a d'entrée de l'attribut multiple. Mais cela au risque de créer de nouvelles redondances qu'il faudra supprimer (tableau **??**, on n'y indique que le tableau OEUVRES, les autres étant inchangées)
Pour supprimer les redondances ainsi apparues, on scinde à nouveau le tableau :

- un premier tableau identifiera l'oeuvre (on peut y mettre aussi la date, déterminée entièrement et de façon unique par l'oeuvre, ainsi que le type). On créera un identifiant de l'oeuvre, construit par exemple sur les premières lettres du compositeur puis une numérotation, de façon à ce que la donnée de cet identifiant caractérise de façon unique l'oeuvre.
- Un deuxième tableau donnera les ensembles instrumentaux utilisés (orchestre, choeur, quatuor...) pour chacune des oeuvres, identifiées par l'identifiant du premier tableau.
- Un troisième donne les instruments solistes.

Cela nous donne les tableaux de la figure 10.7
Évidemment, les lignes des deux derniers tableaux comportant une entrée NIL sont inutiles. La non-présence de ces éléments dans le tableau suffit à retrouver cette information. On peut donc supprimer toutes ces lignes. Par ailleurs, si l'on veut par la suite pouvoir faire des références précises à ces tableaux, il faut pouvoir identifier chacune des lignes de ces tableaux de façon unique et non équivoque. Pour cette raison, on a souvent recours à un identifiant (une numérotation, qui peut être purement utilitaire et ne pas avoir de sémantique particulière).

Enfin, comme il est plus facile de créer la strcture de la base de donnée initialement que de faire des modifications par la suite (surtout si de nombreuses données sont rentrées initialement), on réfléchit attentivement à toutes les contraintes et possibilités liées aux attributs, de façon à limiter le nombre de dépendances pouvant apparaître par la suite, ou la rupture d'un identifiant. Nous illustrons ces propos par deux exemples :

- Supposons que nous voulions ajouter Haendel à notre liste de compositeurs. Quel pays mettre ? L'Allemagne ou l'Angleterre ? Il serait préférable de pouvoir mettre les deux. Ainsi, le compositeur ne détermine pas le pays, et on a une entrée multiple. Comme dans le cas des instrumentations, cela peut inciter à scinder le tableau des compositeurs, en séparant les nationalités. Ce cas n'est pas unique, on peut aussi songer à Scarlatti (Italie ou Espagne ?), Rossini (Italie ou France ?), Stravinski (Russie ou France ?), Rachmaninov (Russie ou USA ?) ou encore Johann-Christian Bach (Allemagne, Italie ou Angleterre ?). Cela dépend aussi de savoir si on veut indiquer dans cette colonne la nationalité admin-

Nom	naissance	mort	pays	oeuvre	date	type	instrumentation	solistes	famille
Couperin	1668	1733	France	Messe des couvents	1690	messe	soliste	orgue	claviers
Bach	1685	1750	Allemagne	Variations Goldberg	1740	variations	soliste	clavecin	claviers
Bach	1685	1750	Allemagne	L'art de la fugue	1750	recueil	soliste	clavecin	claviers
Bach	1685	1750	Allemagne	Le clavier bien tempéré I	1722	recueil	soliste	clavecin	claviers
Bach	1685	1750	Allemagne	Messe en si mineur	1748	messe	orchestre, choeur, chanteurs		
Mozart	1756	1791	Autriche	Don Giovanni	1787	opéra	orchestre, choeur, chanteurs		
Mozart	1756	1791	Autriche	Die Zauberflöte	1791	opéra	orchestre, choeur, chanteurs		
Mozart	1756	1791	Autriche	Concerto pour flûte et harpe	1778	concerto	orchestre, solistes	flûte, harpe	bois, cordes
Mozart	1756	1791	Autriche	Messe du couronnement	1779	messe	orchestre, choeur, chanteurs		
Mozart	1756	1791	Autriche	Requiem	1791	messe	orchestre, choeur, solistes		voix
Beethoven	1770	1827	Allemagne	Symphonie n° 5	1808	symphonie	orchestre		
Beethoven	1770	1827	Allemagne	Symphonie n° 9	1824	symphonie	orchestre, choeur, chanteurs		
Beethoven	1770	1827	Allemagne	Quatuor n° 13	1825	quatuor	quatuor à cordes		
Beethoven	1770	1827	Allemagne	Grande Fugue	1825	quatuor	quatuor à cordes		
Beethoven	1770	1827	Allemagne	Sonate n° 29	1818	sonate	soliste	piano	claviers
Beethoven	1770	1827	Allemagne	Sonate pour violon et piano n° 5	1801	sonate	solistes	violon, piano	cordes, claviers
etc									

FIGURE 10.4 – Tableau des données brutes

COMPOSITEURS			
Nom	naissance	mort	pays
Couperin	1668	1733	France
Bach	1685	1750	Allemagne
Mozart	1756	1791	Autriche
Beethoven	1770	1827	Allemagne

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

OEUVRES					
Nom	oeuvre	date	type	instrumentation	solistes
Couperin	Messe des couvents	1690	messe	soliste	orgue
Bach	Variations Goldberg	1740	variations	soliste	clavecin
Bach	L'art de la fugue	1750	recueil	soliste	clavecin
Bach	Le clavier bien tempéré I	1722	recueil	soliste	clavecin
Bach	Messe en si mineur	1748	messe	orchestre, choeur, chanteurs	
Mozart	Don Giovanni	1787	opéra	orchestre, choeur, chanteurs	
Mozart	Die Zauberflöte	1791	opéra	orchestre, choeur, chanteurs	
Mozart	Concerto pour flûte et harpe	1778	concerto	orchestre, solistes	flûte, harpe
Mozart	Messe du couronnement	1779	messe	orchestre, choeur, chanteurs	
Mozart	Requiem	1791	messe	orchestre, choeur, chanteurs	
Beethoven	Symphonie n° 5	1808	symphonie	orchestre	
Beethoven	Symphonie n° 9	1824	symphonie	orchestre, choeur, chanteurs	
Beethoven	Quatuor n° 13	1825	quatuor	quatuor à cordes	
Beethoven	Grande Fugue	1825	quatuor	quatuor à cordes	
Beethoven	Sonate n° 29	1818	sonate	soliste	piano
Beethoven	Sonate pour violon et piano n° 5	1801	sonate	solistes	violon, piano

Figure 10.5 – Première repartition des données

istrative, ou le pays d'accueil. Si c'est cette dernière interprétation qu'on choisit, il convient également de reconsidérer le cas de Beethoven.

- Le dernier exemple soulève un autre problème : le patronyme ne caractérise pas le compositeur, et ne peut pas servir d'identifiant. Ainsi, nombreuses sont les familles musiciennes dans l'histoire de la musique : la prolifique famille Bach, bien entendu, mais également les Couperin, les Scarlatti, les Mozart, ou plus récemment les Strauss ou les Alain ; sans compter les homonymes non familiaux (comme Johann et Richard Strauss). Le prénom même ne suffit pas à régler le problème (comme l'indique le problème des Johann Strauss père et fils). Ainsi, là aussi, il faut créer un identifiant. Prendre comme identifiant les 3 premières lettres du nom n'est pas suffisant (problème d'homonymie, ou plus généralement, de noms commençant de la même façon, comme Chopin et Chostakovitch, ou Schubert et Schumann). On peut par exemple prendre les 3 premières lettres, suivies d'un numéro pour distinguer les homonymes.

On arrive au final au schéma de la figure 10.8, comportant 6 tableaux.

OEUVRES					
Nom	oeuvre	date	type	instrumentation	solistes
Couperin	Messe des couvents	1690	messe	soliste	orgue
Bach	Variations Goldberg	1740	variations	soliste	clavecin
Bach	L'art de la fugue	1750	recueil	soliste	clavecin
Bach	Le clavier bien tempéré I	1722	recueil	soliste	clavecin
Bach	Messe en si mineur	1748	messe	orchestre	
Bach	Messe en si mineur	1748	messe	choeur	
Bach	Messe en si mineur	1748	messe	chanteurs	
Mozart	Don Giovanni	1787	opéra	orchestre	
Mozart	Don Giovanni	1787	opéra	choeur	
Mozart	Don Giovanni	1787	opéra	chanteurs	
Mozart	Die Zauberflöte	1791	opéra	orchestre	
Mozart	Die Zauberflöte	1791	opéra	choeur	
Mozart	Die Zauberflöte	1791	opéra	chanteurs	
Mozart	Concerto pour flûte et harpe	1778	concerto	orchestre	flûte
Mozart	Concerto pour flûte et harpe	1778	concerto	orchestre	harpe
Mozart	Messe du couronnement	1779	messe	orchestre	
Mozart	Messe du couronnement	1779	messe	choeur	
Mozart	Messe du couronnement	1779	messe	chanteurs	
Mozart	Requiem	1791	messe	orchestre	
Mozart	Requiem	1791	messe	choeur	
Mozart	Requiem	1791	messe	chanteurs	
Beethoven	Symphonie n° 5	1808	symphonie	orchestre	
Beethoven	Symphonie n° 9	1824	symphonie	choeur	
Beethoven	Symphonie n° 9	1824	symphonie	chanteurs	
Beethoven	Symphonie n° 9	1824	symphonie	orchestre, choeur, chanteurs	
Beethoven	Quatuor n° 13	1825	quatuor	quatuor à cordes	
Beethoven	Grande Fugue	1825	quatuor	quatuor à cordes	
Beethoven	Sonate n° 29	1818	sonate	soliste	piano
Beethoven	Sonate pour violon et piano n° 5	1801	sonate		violon
Beethoven	Sonate pour violon et piano n° 5	1801	sonate		violon

FIGURE 10.6 – Tableau sans entrée multiple

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

INSTRUMENTATION	
IdOeuvre	formation intrumentale
COU 1-1	NIL
BAC 1-1	NIL
BAC 1-2	NIL
BAC 1-3	NIL
BAC 1-4	orchestre
BAC 1-4	choeur
BAC 1-4	chanteurs
MOZ 1-1	orchestre
MOZ 1-1	choeur
MOZ 1-1	chanteurs
MOZ 1-2	orchestre
MOZ 1-2	choeur
MOZ 1-2	chanteurs
MOZ 1-3	orchestre
MOZ 1-4	orchestre
MOZ 1-4	choeur
MOZ 1-4	chanteurs
MOZ 1-5	orchestre
MOZ 1-5	choeur
MOZ 1-5	chanteurs
BEE 1-1	orchestre
BEE 1-2	orchestre
BEE 1-2	choeur
BEE 1-2	chanteurs
BEE 1-3	quatuor à cordes
BEE 1-4	quatuor à cordes
BEE 1-5	NIL
BEE 1-6	NIL

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

Figure 10.7 – Tableau sans les redondances pour les oeuvres

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

FIGURE 10.8 – Structure finale de la base de donnée

II.2 Dépendances et redondances

Pour supprimer les redondances, on peut commencer par étudier les dépendances entre les différentes entrées. Par exemple, la date de naissance d'un compositeur ne dépend que du compositeur, et non de l'oeuvre. De même, la famille d'instrument ne dépend que de l'instrument, et non de l'oeuvre.

Définition 10.2.1 (Attribut)

Un attribut est l'une des caractéristiques des données à saisir. Il s'agit donc du titre des colonnes. Ainsi, dans notre exemple, on dispose des attributs COMPOSITEUR, NAISSANCE, MORT, PAYS, OEUVRE...

Définition 10.2.2 (Dépendance fonctionnelle)

On dit qu'il existe une dépendance fonctionnelle de l'attribut A vers l'attribut B si la valeur de l'attribut B ne dépend de rien d'autre que de la valeur de l'attribut A. Autrement dit, toutes les entrées possèdant la même valeur de l'attribut A possèdent également une même valeur de l'attribut A. Une telle dépendance sera notée $A \longrightarrow B$.
On peut étendre la définition aux sous-ensembles d'attribut : il existe une dépendance fonctionnelle d'un sous-ensemble $\mathcal{A}$ d'attributs vers un sous-ensemble d'attributs $\mathcal{B}$ si les valeurs données aux différents attributs de $\mathcal{A}$ déterminent entièrement les valeurs des attributs de $\mathcal{B}$.

Cette définition dépend des entrées de la base : plus la base est fournie, plus la diversité des cas particuliers est susceptible de briser une dépendance fonctionnelle. Pour l'étude des dépendances fonctionnelles, on supposera la base remplie de toutes les entrées imaginables en rapport avec le thème de la base, afin de prendre en considération l'ensemble des cas possibles. Ainsi, dans notre exemple, nous avons les dépendances fonctionnelles suivantes :

- OEUVRE $\longrightarrow$ NOM,PRÉNOM, MORT, NAISSANCE, PAYS, DATE, TYPE
- IDCOMP $\longrightarrow$ MORT, NAISSANCE, NOM, PRÉNOM
- INSTRUMENT $\longrightarrow$ FAMILLE
- l'oeuvre détermine un sous-ensemble de formations instrumentales, mais pas une formation instrumentale. Identifier séparément tous les couples (oeuvre/formation instrumentale) permet de traduire cela par la dépendance :
  IDINSTR $\longrightarrow$ OEUVRE, FORMATION
- IDSOL $\longrightarrow$ OEUVRE, INSTRUMENT de la même manière.
- IDNAT $\longrightarrow$ PAYS, COMPOSITEUR
- En revanche, on n'a pas de dépendance fonctionnelle de NOM vers PRÉNOM, comme le montre l'exemple de la famille Bach, ni entre NOM, PRÉNOM et PAYS, comme le montre l'exemple de Haendel.
- On peut aussi s'interroger sur la dépendance NOM, PRÉNOM $\longrightarrow$ IDCOMP, qui n'est valable que s'il n'y a pas d'homonyme. L'exemple de Johann Strauss montre que ce point est discutable.

Remarquez que si une famille $\mathcal{A}$ détermine un attribut B, mais aucun sous-ensemble strict de $\mathcal{A}$, on peut toujours se ramener au cas où B est déterminé par un unique attribut, en créant un nouvel attribut A, identifiant de l'ensemble des données fournies par $\mathcal{A}$. C'est ce qu'on a fait en introduisant l'indentifiant IDCOMP, permettant d'exprimer de façon plus élémentaire les dépendances induites par le groupe NOM, PRÉNOM.
On a de façon évidente :

Proposition 10.2.3 (Dépendances composées)

Si $Acal \to \mathcal{B}$, $\mathcal{B}' \subset \mathcal{B}$ *et* $\mathcal{B}' \to \mathcal{C}$, *alors* $\mathcal{A} \to \mathcal{C}$.

Ainsi, la dépendance fonctionnelle OEUVRE $\longrightarrow$ NOM peut se retrouver en composant les dépendances OEUVRE $\longrightarrow$ IDCOMP et IDCOMP $\longrightarrow$ NOM.

Définition 10.2.4 (Dépendances élémentaires)

Une dépendance $A \longrightarrow B$ est élémentaire, si elle ne s'écrit pas comme composition non triviale de dépendances.

Ainsi, dans l'exemple ci-dessus, OEUVRE $\longrightarrow$ IDCOMP est une dépendance élémentaire, mais par OEUVRE $\longrightarrow$ NOM.

Définition 10.2.5 (Graphe ADF)

Le graphe ADF (graphe des attributs et des dépendances fonctionnelles) est le graphe dont les sommets sont les attributs et les arêtes orientées indiquent les dépendances élémentaires.

Ainsi, dans notre exemple, on obtient le graphe de la figure 10.9.

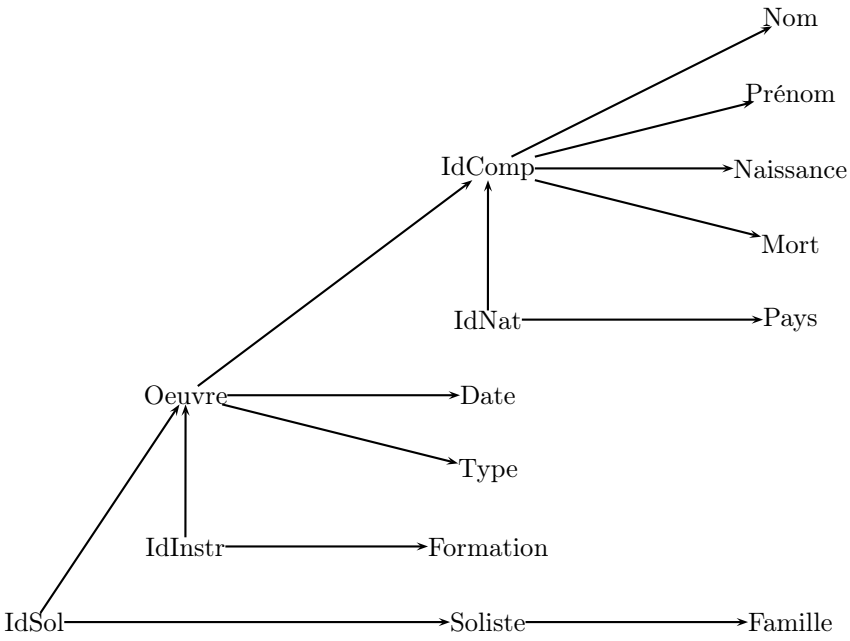


FIGURE 10.9 – Graphe ADF

Remarque 10.2.6

- Un graphe ADF linéaire $A_1 \longrightarrow A_2 \longrightarrow \cdots \longrightarrow A_n$ est telle que la donnée de A_1 détermine successivement les données de A_2, cdots A_n. Cela ne signifie pas que les entrées ne sont pas redondantes. En effet, suivant les valeurs de A_1, A_2 peut prendre plusieurs fois la même valeur, et les attributs suivants sont alors tous répétés.
- Un graphe ADF arborescent orienté de hauteur 1 (donc constitué uniquement de la racine A et de fils directs $B_1, \ldots, B_n$, ces derniers n'ayant pas de fils) est tel que A_1 détermine $B_1, \ldots, B_n$ indépendamment. Le tableau dont les entrées sont les attributs $B_1, \ldots, B_n$ est donc sans redondance.

Méthode 10.2.7 (Trouver la structure de la BDD à partir du graphe ADF)

Nous supposons ici que le graphe ADF est arborescent, non réduit à un noeud. On dira qu'il est terminal sinon. On appelle noeud interne un noeud duquel part au moins une flèche, et fils d'un noeud A tout noeud (interne ou terminal) accessible de A en une étape, en suivant une arête orientée convenablement. Tout noeud terminal est fils d'au moins un noeud interne.

- La remarque précédente nous permet de retrouver assez rapidement la structure finale de notre base de donnée : à chaque noeud interne A va correspondre un tableau T_A, dont les entrées seront ce

noeud et ses fils (mais pas les descendants suivants). Ainsi chaque sous-graphe ADF de ces tableaux sera un graphe de hauteur 1, donc sera non redondant.
- Chacun de ces tableaux T_A aura un identifiant, c'est à dire une colonne dont l'entrée déterminera toutes les autres. Cet identifiant est la colonne correspondant à la racine du sous-graphe ADF (donc au noeud interne considéré)

Remarque 10.2.8

- Il y a des contraintes imposées sur la compatibilité des tableaux. Ainsi, si le fils B de A est lui-même un noeud interne, l'attribut B apparaît à la fois dans T_A et dans T_B, et sert d'identifiant au tableau T_B, mais par au tableau T_A : une entrée possible pour B peut apparaître plusieurs fois ou pas du tout dans T_A ; mais si elle apparaît au moins une fois, elle doit être présente aussi dans T_B (le tableau T_B est à voir comme une définition et précision de toutes les valeurs de l'attribut B pouvant se voir par ailleurs). Pour un attribut C d'un tableau T, notons $T(C)$ l'ensemble des valeurs prises par l'attribut C dans le tableau T. La contrainte ci-dessus s'exprime de la façon suivante : $T_A(B) \subset T_B(B)$ (contrainte d'inclusion). De plus, on a une contrainte d'unicité pour la colonne B de T_B, formalisant la notion d'identifiant : chaque valeur possible de B dans ce tableau n'apparaît qu'une fois.
- Ce sont ces contraintes qui définissent la notion de clé : l'attribut B de T_B constitue une clé primaire (un identifiant auquel on peut référer dans un autre tableau, avec contrainte d'unicité). L'attribut B de T_A constitue une clé étrangère (il est accompagné d'une référence à une clé primaire, donc à un attribut d'un autre tableau, avec contrainte d'inclusion).

Remarque 10.2.9

La structure ADF n'est pas nécessairement arborescente. Pouvez-vous imaginer un exemple de structure contenant une boucle ?

III Algèbre relationnelle

III.1 Schéma relationnel et relation

Nous formalisons maintenant le concept de base de donnée.

Définition 10.3.1 (Attribut)

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 colonnes des différentes tables ci-dessus. On note $\mathcal{A}$ l'ensemble des attributs

Exemple 10.3.2

Dans l'exemple ci-dessus, les attributs sont NOM, PRÉNOM, NAISSANCE, MORT, OEUVRE etc.

Définition 10.3.3 (Domaine d'un attribut)

Le domaine d'une attribut est l'ensemble des valeurs possibles pour cet attribut. Le domaine de l'attribut A sera noté $\mathrm{dom}(A)$.

Exemple 10.3.4

- N'ayant aucune restriction autre que l'utilisation de l'alphabet pour désigner les noms des compositeurs, en notant L l'alphabet latin (avec les lettres accentuées, l'apostrophe et l'espace) donc l'ensemble de toutes les lettres, et L^* le monoïde libre engendré par L (donc tous les mots qu'on peut former avec cet alphabet), on a dom(NOM) $= L^*$
- On a dom(NAISSANCE) $= \mathbb{Z}$ (même s'il est assez peu probable que notre base contienne des compositeurs nés avant l'an 1).
- On peut restreindre certains domaines : par exemple dom($TYPE$) peut être restreint à un certain nombre de types, par exemple :
  dom(TYPE) = { opéra, symphonie, sonate, trio, quatuor, messe, etc. },
  en catégorisant les types d'oeuvres tels qu'on le souhaite.
- De même, dom(INSTRUMENT) ou dom(FAMILLE) peuvent être définis comme des ensembles finis déterminés (à condition de lister tous les instruments, et toutes les familles d'instrument).

Définition 10.3.5 (Schéma relationnel)

Un schéma relationnel (ou schéma de relation) est une application $f : \mathcal{A} \longrightarrow \mathcal{D}$, où $\mathcal{A}$ est un ensemble d'attributs, $\mathcal{D}$ est un ensemble de domaines (donc un ensemble d'ensembles), et $f : A \in \mathcal{A} \mapsto \text{dom}(A) \in \mathcal{D}$.

Par exemple, en notant M l'ensemble de tous les caractères, on peut prendre pour $\mathcal{D}$ l'ensemble $\mathcal{P}(M^*)$, mais cela ne donne aucune contrainte sur le format des entrées. On peut imposer que les entrées soient des chaînes de caractères alphabétiques quelconques, ou des chaînes de caractères de longueur bornée, ou imposée, ou des mots d'une certaine langue, ou que ce soient des entiers, ou des dates sous un certain format (JJ/MM/AAAA), ou des éléments d'un ensemble fini etc.

Si $\mathcal{A} = \{A_1, \ldots, A_n\}$, on parlera simplement du schéma relationnel $\mathcal{S} = (A_1, \ldots, A_n)$, la donnée des domaines dom(A_i) étant implicite. Avec cette convention, les attributs du schéma relationnel sont ordonnés.

Définition 10.3.6 (Relation, valeur, enregistrement)

1. Une relation $\mathcal{R}$ (ou $\mathcal{R}(\mathcal{S})$) associée à un schéma relationnel $\mathcal{S} = (A_1, \ldots, A_n)$ est un ensemble fini de n-uplets de dom(A_1) $\times \cdots \times$ dom(A_n).
2. Les éléments de $\mathcal{R}$ sont appelés valeurs (ou enregistrements) de la relation.
3. $|\mathcal{R}|$ est appelé cardinal de la relation, et est généralement noté $\sharp R$.

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.

III.2 Clés

Comme nous l'avons constaté sur des exemples, en général, les données stockées dans une base de donnée vont être réparties dans plusieurs tables. Ainsi, une base de donnée sera définie par plusieurs schémas relationnels et plusieurs tables. Certaines colonnes d'une table renvoient à des colonnes d'autres tables. ce sont ces références qui permettront ensuite de faire des recherches croisées sur les différentes tables simultanément. La formalisation de ces références est faite par la notion de clé, intimement liée à celle d'identifiant.

Définition 10.3.7 (Identifiant)

Un identifiant d'une table est un ensemble d'attibuts (pouvant être réduit à un unique attribut) tel que les valeurs prises par ces attributs déterminent toutes les autres valeurs de l'enregistrement (donc déterminent toute la ligne).

Ainsi, la donnée des valeurs prises sur les attributs constituant l'identifiant détermine sans ambiguïté (donc *identifient*) la ligne à laquelle on fait référence : 2 lignes différentes ont des valeurs différentes de ces attributs.

Remarque 10.3.8

1. Un identifiant peut être constitué de plusieurs colonnes. Par exemple, dans la table OEUVRE, l'ensemble { IDOEUVRE, COMPOSITEUR, OEUVRE } constitue un identifiant.
2. Par définition, { IDOEUVRE } est encore une identifiant. Si le singleton $\{A\}$ est un identifiant, on dira simplement que A est un identifiant de la table.
3. Par définition, une relation est un ensemble d'enregistrement : il ne peut donc pas y avoir d'enregistrement multiple (plusieurs fois la même ligne). Ainsi, toute table possède au moins un identifiant qui est l'ensemble de tous ses attributs.
4. Tout ensemble d'attribut $\mathcal{I}$ tel qu'il existe $\mathcal{I}' \subset \mathcal{I}$ tel que $\mathcal{I}'$ soit un identifiant est lui aussi un identifiant. Une table possède donc en général plusieurs identifiants.

Définition 10.3.9 (Identifiant minimal)

Un identifiant est minimal dès lors qu'ôter un attribut de cet identifiant supprime le caractère identifiant.

Exemple 10.3.10

- { IDOEUVRE, COMPOSITEUR, OEUVRE } n'est pas un identifiant minimal
- IDOEUVRE st un identifiant minimal
- { COMPOSITEUR, OEUVRE } est aussi un identifiant minimal. En effet COMPOSITEUR n'est pas un identifiant (la plupart des compositeurs ont écrit plus d'une oeuvre, même ceux dont la notoriété est basée sur une unique oeuvre). OEUVRE n'est pas non plus un identifiant, puisque plusieurs compositeurs peuvent avoir composé une symphonie n° 5.
- Ainsi, il peut y avoir plusieurs identifiants minimaux. Certains identifiants minimaux peuvent être contitués de plusieurs attributs. Il n'existe pas toujours d'identifiant minimal constitué d'un unique attribut (on n'aurait pas été obligé d'introduire l'attribut IDOEUVRE).

Remarque 10.3.11

La notion d'identifiant prend tout son sens lorsque la table est suffisamment remplie et contient tous les cas de figure possibles. En effet, si la table OEUVRE n'est pas plus remplie que dans l'exemple donnée, l'attribut OEUVRE constitue également un identifiant, mais évidemment, cela n'a pas beaucoup de pertinence puisqu'on voit facilement que le caractère identifiant a de fortes chances d'être brisé en grossissant la base de donnée (ce qui est la vocation d'une BDD !) Ainsi, la recherche d'un identifiant acceptable ne se fait pas simplement de façon automatique en regardant les données déjà entrées, mais en considérant tous les cas de figures envisageables, même s'ils ne se sont pas encore présentés dans la base.

Remarque 10.3.12

On peut toujours se ramener à la situation où la table possède au moins un identifiant constitué d'un unique attribut. Il suffit pour cela de partir d'un identifiant (constitué de plusieurs attributs), d'ajouter un nouvel attribut ID, et de numéroter de façon artificielle les différentes valeurs possibles prises par les attributs identifiants. C'est ce que nous avons fait en introduisant IDOEUVRE ou IDINSTR ou IDSOL. Pour les deux premiers, nous avons essayé de garder une numérotation significative, pour le dernier, c'est une numérotation qui ne possède en soi pas tellement de sens, et dont le seul intérêt est le caractère identifiant.

Par commodité, on supposera (sans perte de généralité) que toutes les tables que nous considérerons ont au moins un identifiant réduit à un attribut.

Définition 10.3.13 (Clé primaire – Primary Key)

Une clé primaire d'une relation $\mathcal{R}$ est le choix d'un attribut A_{PK} (primary key) identifiant la relation.

On pourrait définir une clé primaire comme un ensemble d'attributs identifiant, mais la remarque précédente nous permet de faire cette simplification sans perte de généralité.

Exemple 10.3.14

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

Certaines colonnes de ces tables renvoient alors à des identifiant : ainsi, IDCOMP de la table NATIONALITÉ renvoie à l'identifiant IDCOMP de la table compositeur.

- Cette référence peut se faire avec des noms de colonnes distincts, ce qui impose de définir explicitement la référence : par exemple la colonne COMPOSITEUR de la table OEUVRE renvoie à la colonne IDCOMP de la table COMPOSITEUR. Ceci nous amène à la notion de clé secondaire.

Définition 10.3.15 (Clé étrangère – Foreign Key)

Soit $\mathcal{R}(\mathcal{S})$ une relation. Une clé étrangère est la donnée d'un coupe $(A_{FK}, Rcal'(Scal'))$ constitué d'un attribut A_{FK} de $\mathcal{S}$ et d'un schéma relationnel $\mathcal{S}'$ muni d'une clé primaire A'_{PK}, et telle que $\mathcal{R}(A_{FK}) \subset \mathcal{R}'(A'_{PK})$ (contrainte d'inclusion, ou contrainte référentielle), où $\mathcal{R}(A_{FK})$ désigne l'ensemble des valeurs prises par A_{FK} dans la table.

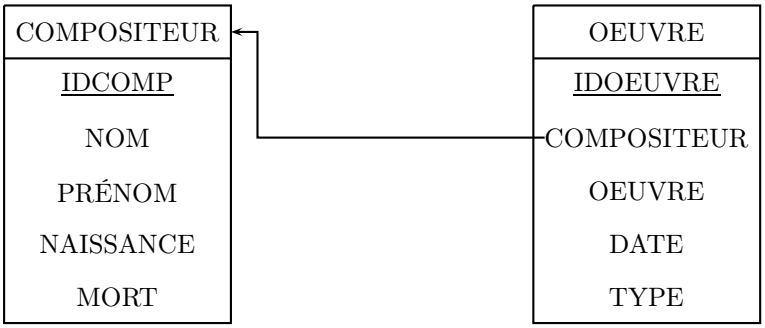
Ainsi, les entrées de la table $\mathcal{R}$ correspondant à l'attribut A_{FK} renvoie aux entrées de la table $\mathcal{R}'$ correspondant à l'attribut A'_{PK}, et toute valeur prise par A_{FK} doit aussi être prise par A'_{PK} (donc avoir été définie dans le tableau référent).

Ces notions peuvent être définies uniquement sur le schéma relationnel, en imposant alors des contraintes sur les entrées à venir de la base (contrainte d'unicité pour la clé primaire, contraite référentielle pour la clé étrangère) : le SGBD peut refuser une entrée dans la BDD si elle ne respecte par ces contraintes, afin de préserver l'intégrité des clés.

On pourra donc se contenter de représenter ces notions sur les schémas relationnels. Un schéma relationnel sera représenté par un tableau listant ses attributs, par exemple :

COMPOSITEUR
IDCOMP
NOM
PRÉNOM
NAISSANCE
MORT

Dans ce tableau, la clé primaire sera indiquée en la soulignant, comme on l'a fait ci-dessus. Une clé étangère entre un attribut d'une table et une autre table sera représentée en reliant par une flèche l'attribut correspondant et la table référente :



Dans cette représentation, il n'y a pas d'ambiguïté pour savoir à quelle colonne du premier tableau réfère la flèche (la référence se fait vers l'identifiant déclaré, donc la clé primaire).

III.3 Schéma de base de donnée

Définition 10.3.16 (Schéma de BDD)

Un schéma de BDD est un ensemble

$$E = \{\mathcal{T} = (\mathcal{S}, A_{PK}, \{(A_{FK,i}; \mathcal{S}_i), i \in I\})\}.$$

dont les éléments sont des triplets formés de :
- un schéma relationnel $\mathcal{S}$
- une clé primaire A_{PK} associée au schéma relationnel $\mathcal{S}$
- un ensemble de clés étrangères $(A_{FK,i}, \mathcal{S}_i)$ d'attributs de $\mathcal{S}$ vers des schémas relationnels $\mathcal{S}_i$.

Il s'agit donc de la donnée de la structure même d'une base de donnée. Chaque triplet élément de E est la donnée d'une table de la base de donnée, de son identifiant (donc on impose que chaque table ait un identifiant), et des références éventuelles vers une ou plusieurs autres tables de la base (par la donnée des clés étrangères).

On peut facilement étendre cette définition au cas où les clés sont données par des ensembles d'attribut plutôt qu'un attribut unique.

Définition 10.3.17 (Base de donnée)

Soit E un schéma de BDD. Une BDD de schéma E est un ensemble $\{\mathcal{R}_\mathcal{T}, \mathcal{T} \in E\}$ de relations (tables), vérifiant les contraintes d'unicité et les contraintes référentielles liées aux clés primaires et étrangères définies par E.

On verra dans le prochain chapitre comment effectuer des opérations sur une telle structure. Toutes les requêtes (recherches) peuvent se faire à partir de combinaisons de contructions algébriques simples (algèbre relationnelle). Nous étudierons ces requêtes au travers du langage SQL (Structured Query Language),

actuellement prédominant dans ce domaine. La force de cette approche des bases de donnée, et de ce langage, vient du fait que les opérations de l'algèbre relationnelle peuvent alors toutes s'écrire avec une seule instruction : SELECT.

IV Exercices

Exercice 16

Décrire sous forme d'un diagramme avec des flèches la structure complète du schéma de BDD défini par l'exemple développé tout au long de ce chapitre.

Exercice 17

Imaginer comment enrichir la structure de cette base de donnée en y incluant la possibilité de répertorier les enregistrements que l'on possède (il faut donc inclure la diversité des interprétations, et des supports musicaux, et la possibilité d'avoir plusieurs oeuvres sur un même support)

Exercice 18

Définir un schéma de BDD dont le but est de stocker les informations suivantes relatives aux classes préparatoires scientifiques au lycée : élèves et renseignements divers sur les élèves, professeurs et renseignements divers, matières, types de classe (PCSI, MPSI...), classes.

Exercice 19

Enrichir l'exemple de l'exercice précédent en y incluant l'emploi du temps et les salles (et leur capacité, ou d'autres caractéristiques)...

Exercice 20

Comment gérer un hôtel composé de 3 bâtiments, et de chambres de capacités diverses et prix divers ?

11

SQL : Création d'une BDD et requêtes

Dans ce chapitre, nous voyons comment créer et interroger une base de données. Le langage qui s'est imposé pour cela est le langage SQL (Structured Query Language). La plupart des SGBD actuels utilisent ce langage, avec éventuellement quelques variantes. Ceux ne l'utilisant pas (ACCESS par exemple) ont une interface de traduction permettant à l'utilisateur d'entrer ses requêtes en SQL.

Le langage SQL est utilisé tout aussi bien pour la création, le rmplissage et l'interrogation de la base. En ce sens, les 3 instructions principales sont :

- `CREATE` pour créer une table
- `INSERT` pour ajouter des données
- `SELECT ... FROM ... WHERE ...` pour interroger la base (requête)

Nous utiliserons Python pour créer et manipuler les bases de données. Pour cela, nous utilisons la bibliothèque `sqlite3`. Le principe est ensuite de créer une connection sur une BDD avec la fonction `connect`, puis de créer un « curseur » sur cette connection. Grâce à ce curseur, on peut ensuite exécuter sur la base des instructions SQL, grâce à la méthode `execute()` applicable au curseur créé. On passe en paramètre de cette méthode l'instruction SQL sous forme d'une chaîne de caractère (par exemple raw string pour éviter tout problème d'utilisation des guillemets, apostrophes et retours à la ligne).
La connection doit être fermée en fin de programme.
Par exemple, pour créer une table `compositeur` dans une base de données, l'instruction SQL est :

```
CREATE TABLE IF NOT EXISTS compositeur (
    idcomp varchar(6) NOT NULL,
    nom varchar(30) NOT NULL,
    prenom varchar(30) NOT NULL,
    naissance int(4),
    mort int(4),
    PRIMARY KEY (idcomp)
    );
```

Si on veut créer cette base dans un SGBD muni d'une interface graphique conviviale, il est possible de rentrer directement cette instruction telle quelle. Pour créer cette table via Python, on écrit :

```
import sqlite3

connection = sqlite3.connect('musique.db')

cur = connection.cursor()
```

```
cur.execute("""CREATE TABLE IF NOT EXISTS compositeur (
    idcomp varchar(6) NOT NULL,
    nom varchar(30) NOT NULL,
    prenom varchar(30) NOT NULL,
    naissance int(4),
    mort int(4),
    PRIMARY KEY (idcomp)
    );""")

connection.commit()
connection.close()
```

De même, chaque requête doit ensuite être envoyée via l'instruction `cur.execute("""requête""")`. Le résultat parvient sous forme d'un objet itérable, les objets (sous format `tuple`) pouvant être énuméré à l'aide de `for`. Par exemple, la fonction suivante créer l'affichage des objets de l'itérable `cur`, à employer à l'issue d'une requête.

```
def affiche(curseur):
    for L in curseur:
        print(L)
```

Chaque ligne `L` du résultat de la requête est affiché sous forme d'un tuple.

Un des intérêts de l'utilisation d'un langage de programmation (par exemple Python) pour l'interrogation des bases de données est de pouvoir facilement utiliser, et donc traiter informatiquement, le résultat de la requête.

I Création d'une base de données

I.1 Création des tables

Nous ne nous étendrons pas trop sur la création d'une base de données. La création d'une table se fait avec l'instruction `CREATE`
Si on veut recréer une base déjà existante (par exemple obtenue lors d'une tentative précédente qui ne nous satisfait pas), on peut commencer par effacer les tables existantes, à l'aide de `DROP TABLE`. Pour ne pas faire d'erreur, on peut ajouter une option `IF EXISTS` :

```
DROP TABLE IF EXISTS compositeur;
```

On créer une nouvelle table avec `CREATE TABLE`. L'option `IF NOT EXISTS` permet de ne pas retourner d'erreur au cas où la table existe déjà (dans ce cas, il n'y a pas de nouvelle table créée)
Lorsqu'on crée une table, il faut préciser :

1. son nom
2. pour chacun des attributs : son nom, son format et d'éventuelles contraintes sur lesquelles nous ne nous étendons pas. La seule que nous mentionnons (en plus des contraintes liées aux clés) est la contrainte `NOT NULL`, imposant qu'une valeur soit nécessairement donnée à cet attribut lors d'un enregitrement d'une donnée.
3. la donnée de la clé primaire, et eventuellement des clés étangères.

Par exemple, pour créer une table `oeuvre` (la table `compositeur` étant définie comme plus haut) :

```
CREATE TABLE IF NOT EXISTS oeuvre (
    idoeuvre varchar(10) NOT NULL,
    compositeur varchar(6) NOT NULL,
```

```
    oeuvre varchar(128) NOT NULL,
    date int(4) NOT NULL,
    type varchar(20) NOT NULL,
    FOREIGN KEY (compositeur) REFERENCES compositeur,
    PRIMARY KEY (idoeuvre));
```

La clé étangère renvoie à la table `compositeur`. L'identification se fait sur la clé primaire de cette table. Une autre syntaxe possible pour la déclaration des clés est :

```
CREATE TABLE IF NOT EXISTS oeuvre (
    idoeuvre varchar(10) NOT NULL PRIMARY KEY,
    compositeur varchar(6) NOT NULL REFERENCES compositeur,
    oeuvre varchar(128) NOT NULL,
    date int(4) NOT NULL,
    type varchar(20) NOT NULL);
```

Les formats classiques des attributs sont `varchar(n)`, `int(n)`, `decimal(n,m)`, pour des chaînes de caractère, des entiers ou des décimaux. Le paramètre n indique le nombre maximal de caractères, et m indique le nombre de chiffres après la virgule. D'autres types existent (par exemple `date`). À découvrir soi-même.

On peut définir des contraintes beaucoup plus précises sur les différents attributs (notamment des contraintes d'unicité). Ceci dépasse les objectifs de ce chapitre d'introduction.

Il existe également des instructions permettant de modifier des tables existantes, en particulier `ALTER TABLE`. On peut à l'aide de cette instruction modifier le nom d'un attribut, les contraintes, ajouter ou enlever des clés... Là encore, cela va au-delà des objectifs de ce chapitre.

I.2 Entrée des données

Entrer une ligne dans une table se fait avec l'instruction `INSERT INTO ... VALUES ...` , suivant la syntaxe suivante :

```
INSERT INTO compositeur (idcomp, nom, prenom, naissance, mort) VALUES
    ('BAC 1', 'Bach', 'Johann Sebastian', 1685, 1750),
    ('MOZ 1', 'Mozart', 'Wolfgang Amadeus', 1756, 1791),
    ('BEE 1', 'Beethoven', 'Ludwig (van)', 1770, 1827),
    ('BOU 1', 'Boulez', 'Pierre', 1925, NULL);
```

On indique donc le nom de la table, les attributs qu'on souhaite remplir, dans un ordre déterminé, et les n-uplets que l'on souhaite définir, séparés par une virgule. On termine la liste par un point-virgule. Comme le montre la dernière ligne entrée, certains attributs peuvent rester non définis, ce qui revient à leur attribuer la valeur `NULL`. Ceci est possible si l'attribut n'a pas été défini avec la contrainte `NOT NULL` lors de la création de la table.

En n'indiquant qu'un sous-ensemble des attributs de la table, on peut ne remplir que quelques colonnes. Les autres seront alors complétées par la valeur `NULL` :

```
INSERT INTO compositeur (idcomp, nom, prenom) VALUES
    ('RAC 1', 'Rachmaninov', 'Serge'),
    ('DVO 1', 'Dvorak', 'Antonin');
```

On peut aussi modifier des données déjà rentrées avec `UPDATE` :

```
UPDATE compositeur SET naissance = 1873, mort = 1943 WHERE idcomp = 'RAC 1'
```

II Interrogation d'une BDD (Requêtes)

Nous abordons ici 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 et étrangères). 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.

L'ensemble des requêtes données en exemple ici porte sur une base de donnée musicale respectant la structure de BDD exposée dans le chapitre précédent, à l'exception des formations musicales, laissées de côté. Ainsi, en plus des deux tables `compositeur` et `oeuvre` déjà créées dans les exemples précédents, nous avons aussi les tables suivantes à créer :

```
CREATE TABLE IF NOT EXISTS nationalité (
    idnat int(3) NOT NULL,
    compositeur varchar(6) NOT NULL,
    pays varchar(30) NOT NULL,
    PRIMARY KEY (idnat),
    FOREIGN KEY (compositeur) REFERENCES compositeur );

CREATE TABLE IF NOT EXISTS soliste (
    idsol int(5) NOT NULL,
    oeuvre varchar(10) NOT NULL,
    instrument varchar(30) NOT NULL,
    PRIMARY KEY (idsol),
    FOREIGN KEY (instrument) REFERENCES instrument,
    FOREIGN KEY (oeuvre) REFERENCES oeuvre );

CREATE TABLE IF NOT EXISTS instrument (
    instrument varchar(30) NOT NULL,
    famille varchar(30) NOT NULL,
     PRIMARY KEY (instrument) );
```

Il est important de noter que la syntaxe utilisée ici est celle définie dans la bibliothèque SQLITE de Python, qui s'écarte parfois de la synthaxe standard. De manière générale, d'une version à l'autre de SQL, il existe souvent des petites variantes dans la syntaxe.

II.1 Requêtes simples

Définition 11.2.1 (Requête simple)
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
```

Remarque 11.2.2

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 projection sur 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',)
```

Formellement, il s'agit d'une projection sur l'ensemble des coordonnées selectionnées. Si le tableau s'appelle `TAB`, et les attributs selectionnés $A_{i_1}, \ldots, A_{i_k}$, on notera formellement $\mathrm{TAB}[A_{i_1}, \ldots, A_{i_k}]$.
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` :

```
#Requête:
SELECT DISTINCT famille
       FROM instrument""")

#Réponse:
('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 :

```
#Requête:
SELECT *
   FROM compositeur
   WHERE naissance = 1685

#Réponse:
('BAC 1', 'Bach', 'Johann Sebastian', 1685, 1750)
('HAE 1', 'Haendel', 'Georg Friedrich', 1685, 1759)
('SCA 1', 'Scarlatti', 'Domenico', 1685, 1757)
```

Voici une liste non exhaustive de tests possibles pour exprimer les conditions :

```
# TESTS POSSIBLES POUR LA CONDITION WHERE...

= > < <> <= >= # comparaisons sur des nombres ou chaînes de caractère.
               # Attention, 'Z'<'a'
IS NULL        # teste si la valeur n'a pas été attribuée
IN (A1,A2,...) # teste l'appartenance à une liste
BETWEEN a AND b # appartenance à un intervalle (nombre ou chaîne)
LIKE '_1'      # compare à une chaîne de caractère où _ représente
```

```
                # un caractère quelconque
LIKE '%1'       # compare à une chaîne de caractère où % représente
                # une chaîne (éventuellement vide) quelconque
AND OR NOT      # opérations booléennes usuelles, à parenthéser bien
```

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 SUR LES ATTRIBUTS

+ * - /            # Opérations arithmétiques usuelles
CHAR_LENGTH(t)     # longueur de la chaîne de caractères t
ch1 || ch2         # concaténation
REPLACE(attribut,ch1,ch2) # remplace dans l'attribut les sous-chaînes
                      # ch1 par ch2
LOWER(ch)          # met en minuscule
UPPER(ch)          # met en majuscule
SUBSTR(ch,a,b)     # extrait la sous-chaîne des indices a à b
                   # SUBSTRING dans de nombreuses versions
```

Voici quelques exemples :

```
# Requête: compositeurs dont le nom a 6 lettres

    SELECT nom
        FROM compositeur
        WHERE LENGTH(nom) = 6""")

#Réponse:
('Mozart',)
('Rameau',)
('Marais',)
('Chopin',)
('Brahms',)
('Wagner',)
('Boulez',)
('Dvorak',)
```

```
# Requête: retourner Prénom Nom sous forme d'une unique chaîne, pour
#         les compositeurs dont le nom commence par B

    SELECT prenom || ' ' || nom
        FROM compositeur
        WHERE nom LIKE 'B%' """)

# Réponse:
('Johann Sebastian Bach',)
('Ludwig (van) Beethoven',)
('Johann Christian Bach',)
('Johannes Brahms',)
('Pierre Boulez',)
```

```
# Requête: NOM en majuscules, prénom en minuscules, pour les
#         compositeurs morts entre 1820 et 1830

    SELECT UPPER(nom), LOWER(prenom)
        FROM compositeur
        WHERE mort BETWEEN 1820 AND 1830""")

# Réponse:
('BEETHOVEN', 'ludwig (van)')
('SCHUBERT', 'franz')
```

```
# Requête: les 3 premières lettres du nom des compositeurs nés avant
#         1600

    SELECT UPPER(SUBSTR(nom,1,3))
        FROM compositeur
        WHERE naissance < 1600""")

# Réponse:
('DOW',)
('HAL',)
```

Certaines opérations mathématiques portant sur l'ensemble des valeurs d'un attribut sont possibles (fonctions agrégatives) :

```
# 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 agréatives servent à définir de nouvelles colonnes, mais ne peuvent pas être utilisées dans une condition. Un exemple :

```
# Requête: 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.

```
# 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')

# Résultat:
('BAC 1', 'Sonate pour violon 1 BWV 1001', 1720)
('BEE 1', 'Sonate 1', 1795)
('BRA 1', 'Sonate 2', 1852)
('CHO 1', 'Sonate 2', 1839)
('HAE 1', 'Sonate pour fûte à bec HWV 358', 1710)
('MOZ 1', 'Sonate pour piano 1 K279', 1774)
```

Enfin, notons qu'il est possible d'ordonne les résultats par ordre croissant (numérique ou alphabétique) grâce à ORDER BY :

```
# Compositeurs nés enre 1870 et 1890, par ordre de naissance

    SELECT nom, naissance, mort
        FROM compositeur
        WHERE naissance BETWEEN 1870 AND 1890
        ORDER BY naissance""")

# Résultat:
('Rachmaninov', 1873, 1943)
('Ives', 1874, 1954)
('Ravel', 1875, 1937)
('Stravinsky', 1882, 1971)
```

II.2 Sous-requêtes

Définition 11.2.3 (Sous-requête)
Une sous-requête est une requête portant sur l'extraction d'attributs d'une même table, la condition s'exprimant à l'aide d'attributs d'une autre table. Plus précisément, on extrait les attributs $A_1, \cdots A_k$, et on exprime sur A_1 une condition portant sur les attributs de la table T_1, telle que (A_1, T_1) soit une clé étrangère. On peut combiner grâce aux opérations booléennes, plusieurs tests, pour les différents A_i, impliquant plusieurs tables T_i, en suivant les clés étrangères.

La syntaxe à suivre est la suivante :

```
SELECT A1, A2, ..., Ak
    FROM T
    WHERE A1 IN (SELECT B1
                    FROM T1
                    WHERE condition)
```

Ici, B1 est la clé primaire de T_1, donc l'attribut vers lequel réfère la clé étrangère (A_1, T_1). On exprime dons la condition sur B_1, en extrayant de T_1 l'attribut B_1, en vérifiant la condition requise. On teste ensuite l'appartenance de A_1 à la liste obtenue.

Remarque 11.2.4

- Il est important de remarquer que le résultat d'une requête SELECT a le même format qu'une table, et peut donc être réutilisé dans une autre requête SELECT comme toute table. On peut donc imbriquer des instructions SELECT les unes dans les autres.
- On peut aussi remonter les clés étrangères : dans ce cas, la clé étrangère est (B_1, T), et A_1 est la clé primaire de T.

Un exemple :

```
# Les compositeurs de la base ayant écrit au moins une musique de film:
   SELECT nom, prenom
        FROM compositeur
        WHERE idcomp in (SELECT DISTINCT compositeur
                        FROM oeuvre
                        WHERE type = 'film')

# Réponse:
('Chostakovitch', 'Dimitri')
('Prokofiev', 'Serge')

# Les oeuvres avec hautbois:
   SELECT compositeur, oeuvre
        FROM oeuvre
        WHERE idoeuvre in (SELECT DISTINCT oeuvre
                          FROM soliste
                          WHERE instrument = 'hautbois')

# Réponse:
('MIL 1', 'La cheminée du roi René')
('POU 1', 'Sonate pour hautbois et piano')
('VIV 1', '12 concertos pour violon et hautbois op 7')
```

On peut enchaîner de la sorte des sous-requêtes en suivant les clés étrangères, si la condition porte sur une table accessible en plusieurs étapes :

```
# Oeuvres de compositeur français utilisant un instrument soliste
# dans la famille des cordes frottées
    SELECT compositeur, oeuvre
        FROM oeuvre
        WHERE (idoeuvre in (SELECT DISTINCT oeuvre
                        FROM soliste
                        WHERE instrument IN
                             (SELECT instrument
                              FROM instrument
                              WHERE famille = 'cordes frottées')))
          AND
             (compositeur in (SELECT idcomp
                              FROM compositeur
                              WHERE idcomp IN
                                  (SELECT compositeur
                                   FROM nationalité
                                   WHERE pays = 'France')))
```

```
('MAR 1', 'Pièces à une et deux violes, premier livre')
('MAR 1', 'Deuxième livre de pièces de viole')
('MAR 1', 'Pièces de viole, troisième livre')
('MAR 1', 'Pièces à une et à trois violes, quatrième livre')
('MAR 1', 'Pièces de viole, cinquième livre')
```

Le principe de la sous-requête permet aussi d'utiliser le résultats de fonctions agrégatives dans un test :

```
# Compositeurs morts 20 ans avant l'âge moyen des compositeurs de la
# base
   SELECT nom, prenom, naissance, mort
       FROM compositeur
       WHERE mort - naissance +20 < (SELECT AVG(mort-naissance)
                      FROM compositeur)

('Mozart', 'Wolfgang Amadeus', 1756, 1791)
('Schubert', 'Franz', 1797, 1828)
('Chopin', 'Frédéric', 1810, 1849)
('Moussorgski', 'Modeste', 1839, 1881)
('Purcell', 'Henry', 1659, 1685)
```

II.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 des « jointures » permettant des requêtes complexes sur plusieurs tables. Nous isolons cette dernière construction dans le paragraphe suivant.

Définition 11.2.5 (Intersection)

L'intersection de deux extractions de deux table peut se faire à condition que les attributs extraits des deux tables soient de même format. Généralement, elle se fait sur les attributs reliés par une clé (l'un d'eux étant une clé primaire). 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 BETWEEN 1700 AND 1800
   INTERSECT
   SELECT DISTINCT compositeur FROM oeuvre
                     WHERE type = 'sonate'""")
('BEE 1',)
('MOZ 1',)
('SCH 1',)
```

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
```

Définition 11.2.6 (Union)
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.
On peut aussi faire des exceptions $A \setminus B$, sur des tables de même type :

```
# Instruments de la famille bois non utilisés par un compositeur
# allemand dans notre base.

SELECT instrument FROM instrument
                    WHERE famille = 'bois'
EXCEPT
SELECT instrument FROM soliste
    WHERE oeuvre in
       (SELECT idoeuvre FROM oeuvre WHERE compositeur in
            (SELECT idcomp FROM compositeur
             WHERE idcomp in
                      (SELECT compositeur FROM nationalité
                       WHERE pays = 'Allemagne')))
```

La dernière opération ensembliste est le produit cartésien.

Définition 11.2.7 (Produit cartésien de deux tables)
Le produit cartésien de deux tables $\mathcal{R}_1$ et $\mathcal{R}_2$ est l'ensemble des $n+p$-uplets $(x_1, \ldots, x_n, y_1, \ldots, y_p)$, pour toutes les $(x_1, \ldots, x_n) \in \mathcal{R}_1$ et $(y_1, \ldots, y_p) \in \mathcal{R}_2$, sans préoccupation de concordance des attributs de $\mathcal{R}_1$ et $\mathcal{R}_2$ 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.

II.4 Jointure

La technique de la jointure permet de former une table à l'aide d'une selection d'attributs provenant de deux tables différentes :

Définition 11.2.8 (Jointure)
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, typiquement d'un côté une clé étrangère vers l'autre table, de l'autre la clé primaire), 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 `INNER JOIN ... ON ...` (le terme `INNER` étant facultatif) :

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

On peut combiner cette jointure avec une sélection conditionnelle `WHERE` :

```
# 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

('Dowland', 'John', 'A dream')
('Dowland', 'John', 'Dowland s First Gaillard')
('Dowland', 'John', 'Captain Candish s Gaillard')
('Mozart', 'Wolfgang Amadeus', 'Sonate pour piano 1 K279')
('Mozart', 'Wolfgang Amadeus', 'Sonate pour piano 2 K280')
('Mozart', 'Wolfgang Amadeus', 'Sonate pour piano 3 K281')
('Mozart', 'Wolfgang Amadeus', 'Sonate pour piano 4 K282')
('Mozart', 'Wolfgang Amadeus', 'Sonate pour piano 5 K283')
('Mozart', 'Wolfgang Amadeus', 'Sonate pour piano 7 K309')
('Mozart', 'Wolfgang Amadeus', 'Symphonie 6 K43')
('Rachmaninov', 'Serge', 'Concerto pour piano 1')
('Schubert', 'Franz', 'Erlkönig')
```

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

('Bach', 'Johann Sebastian', 'Sonate pour flûte et clavecin BWV 1030', 'flûte')
('Boulez', 'Pierre', 'Dialogue de l ombre double', 'clarinette')
('Haendel', 'Georg Friedrich', 'Sonate pour flûte à bec HWV 358', 'flûte')
('Haendel', 'Georg Friedrich', 'Sonate pour flûte à bec HWV 365', 'flûte')
('Milhaud', 'Darius', 'La cheminée du roi René', 'flûte')
('Milhaud', 'Darius', 'La cheminée du roi René', 'hautbois')
('Milhaud', 'Darius', 'La cheminée du roi René', 'clarinette')
('Milhaud', 'Darius', 'La cheminée du roi René', 'basson')
('Poulenc', 'Francis', 'Sonate pour hautbois et piano', 'hautbois')
('Poulenc', 'Francis', 'Sonate pour clarinette et piano', 'clarinette')
('Vivaldi', 'Antonio', '12 concertos pour violon et hautbois op 7', 'hautbois')
```

Une belle liste musicale pour clore ce cours d'informatique...