Informatique pour tous
première année

Julien Reichert

2020/2021

Table des matières

Première partie

Cours

Chapitre 1

Représentation des nombres

1.1 Introduction aux bases numériques

Définition

Soit b un entier naturel ≥ 2. On considère un ensemble C de b caractères, usuellement des chiffres à partir de 0, en complétant avec des lettres dans l'ordre alphabétique, donnés dans l'ordre croissant.
On représente un entier naturel en base b en écrivant des caractères de l'ensemble C en séquence, en précisant la valeur de b pour lever toute ambiguïté.
La notation proposée dans ce cours, qui n'est pas uniformisée, est $\overline{a_{n-1}a_{n-2}\dots a_1a_0}^b$, pour représenter le nombre $\sum_{i=0}^{n} a_i b^i$. Dans le nombre en question, le caractère dit de poids fort est le plus à gauche et le caractère dit de poids faible est le plus à droite.

Les bases usuelles sont 10 (base décimale, utilisée de façon naturelle actuellement), 2 (le binaire, fondement de l'informatique) et 16 (les nombres hexadécimaux, utilisés en informatique pour éviter d'avoir trop de chiffres). On trouve également les bases 8 et 12 de manière cependant moins prononcée.

Toutes les opérations arithmétiques apprises au primaire se font de manière similaire dans n'importe quelle base. Par exemple, en base 8, $\overline{35}^8 + \overline{56}^8 = \overline{113}^8$ car $5 + 6$, qui correspond au nombre 11, s'écrit $\overline{13}^8$. On note donc 3 et on retient 1, de même qu'on a une retenue pour les « huitaines », qui devient la « soixante-quatraine ».

Exercice

Calculer en base 8 les sommes $\overline{146}^8 + \overline{334}^8$ et $\overline{357}^8 + \overline{464}^8$, ainsi que le produit

$\overline{54}^8 \times \overline{465}^8$.

La représentation en base b s'étend aux entiers relatifs en précisant le signe.

Proposition

Soit b un entier naturel ≥ 2. On peut représenter un nombre rationnel en base b de manière exacte, c'est-à-dire avec un nombre fini de caractères, si et seulement si le nombre en question est le quotient d'un entier relatif par une puissance de b. Une autre formulation : le nombre en question a pour fraction la plus simplifiée $\frac{p}{q}$, où tous les diviseurs premiers de q sont des diviseurs de b. La représentation comporte alors éventuellement une virgule, et on écrit $\overline{a_{n-1}a_{n-2}\dots a_1a_0,a_{-1}a_{-2}\dots a_{-m}}^b = \sum_{i=-m}^{n-1} a_i b^i$.

Remarque : En base 10, on retrouve la notion de nombres décimaux. Bien entendu, quelle que soit la valeur de b, un nombre irrationnel aura toujours un nombre infini de caractères après la virgule dans sa représentation en base b.

Exemple : Le nombre $\frac{13}{3}$ s'écrit de manière exacte dans toute base multiple de 3.

Proposition

[Pour la culture] Quelle que soit la valeur de b, tout nombre rationnel r a une écriture en base b ultimement périodique, c'est-à-dire qu'à partir d'un certain rang fini, un même motif se répète.

Cette proposition se prouve simplement à l'aide du petit théorème de Fermat quand b est premier (sinon, c'est de l'arithmétique avancée). On peut l'illustrer en considérant les fractions $\frac{1}{p}$ en base b. Si p divise b, alors l'écriture de $\frac{1}{p}$ est exacte : $\frac{1}{p} = \overline{0,a}^b$, où a est le caractère correspondant à l'entier $\frac{b}{p}$, la période étant alors de 1 et le motif étant 0. Sinon, le petit théorème de Fermat dit que b^{p-1} est congru à 1 modulo p, donc diviser 1 par p laisse entrevoir une répétition après $p-1$ étapes. La période est donc $p-1$ ou un de ses diviseurs.

1.2 Représentation des entiers

1.2.1 Entiers naturels

Les informations dans les ordinateurs peuvent être vues comme des signaux électriques. On peut résumer la pratique à cela : chaque unité de mémoire peut être dans deux états, et donc aucun nombre n'est stocké en tant que tel, mais représenté à l'aide d'unités de mémoire dont l'interprétation dépend du contexte.

Ainsi, un ordinateur travaille avec des nombres constitués de 0 et de 1, donc en binaire, et on appelle « *bits* » (pour "binary digits", soit chiffres binaires) les caractères de base du binaire quand ils sont utilisés en informatique. On regroupe éventuellement par paquets de 4 (pour former un caractère hexadécimal) ou de 8 (des *octets*) les bits dans un souci de concision.

Exercice

Au passage, jusqu'à combien peut-on compter sur les doigts ? Et en considèrant qu'on peut les plier selon différents degrés ? Quels problèmes pratiques aurait-on si on voulait utiliser autre chose que du binaire ? (Il s'avère que la logique ternaire a été envisagée et n'est pas totalement tombée dans l'oubli.)

Par exemple, le protocole WEP [1] utilise des clés de 64 ou 128 bits, dont un vecteur d'initialisation et la clé proprement dite, formée de respectivement 10 ou 26 caractères hexadécimaux, qu'on peut aussi écrire comme respectivement 5 ou 13 caractères alphanumériques à l'aide d'une table de 256 caractères.

Exercice

D'ailleurs, combien y a-t-il de bits dans le vecteur d'initialisation ? C'est en fait la méthode de cryptage utilisée avec ce vecteur qui engendre la faille du protocole.

En pratique, la limite de la mémoire d'un ordinateur empêche évidemment d'écrire n'importe quel entier naturel, et on parlera d'entiers sur n bits (souvent 32 ou 64).

1. très peu fiable

Par exemple, les dates d'évènements [2] sont usuellement stockées à l'aide de ce qu'on appelle "timestamp". Cette horloge compte les secondes écoulées depuis le premier janvier 1970 à minuit UTC (pour les systèmes Unix), ce qui causera des problèmes le 19 janvier 2038 pour les systèmes sur 32 bits (qui ont de toute façon déjà presque disparu actuellement). Ceci nous amène à parler de *dépassement arithmétique*.

Les opérations arithmétiques, en binaire comme en décimal et dans toutes les bases, font occasionnellement apparaître des retenues. Que se passe-t-il quand la retenue apparaît ou se propage au-delà du dernier caractère disponible ? Elle est tout simplement perdue. Ainsi, dans la représentation des entiers naturels sur 32 bits, $2^{31} + 2^{31}$ donne 0. C'est ce qu'on appelle le dépassement arithmétique.

1.2.2 Entiers relatifs

Naïvement, on peut penser à réserver un bit dans la représentation d'un entier naturel pour préciser le signe. Cette méthode a l'avantage de la simplicité, mais plusieurs inconvénients, dont l'existence de deux versions de zéro, font qu'en pratique elle est laissée de côté.

Une autre façon de faire pourrait être de considérer que puisqu'on peut écrire les nombres de 0 à $2^n - 1$ sur n bits, on soustrait 2^{n-1} à tous les nombres représentés. L'inconvénient majeur réside cette fois dans les opérations, qui n'ont rien d'évident.

On emploie donc pour représenter un entier relatif la notation dite en *complément à deux*. Il s'agit de distinguer les cas suivant que l'entier soit positif (auquel cas le premier bit est à zéro) ou strictement négatif (auquel cas le premier bit est à 1). Dans le premier cas, on écrit simplement l'entier que l'on veut représenter sur les $n - 1$ derniers bits, et dans le deuxième cas on écrit en fait l'entier plus 2^n sur les n bits.

La notation en complément à deux permet alors d'écrire sur n bits les entiers entre -2^{n-1} et $2^{n-1} - 1$.

Exercice

Il est utile de connaître la représentation des nombres les plus classiques. Elle est évidente pour les nombres positifs, retenir celles de -2^{n-1}, de -1 et d'autres opposés de puissances de 2 est incontournable.

2. notamment pour les logs, il ne s'agit pas d'histoire ici

Propriété : Dans la notation en complément à deux, pour calculer l'opposé d'un entier x, on remplace tous les 0 par des 1 et vice-versa, puis on ajoute 1 à la représentation. Ceci ne marche évidemment pas pour -2^{n-1} (qui s'écrit avec un 1 au début et des 0 partout ailleurs), car son opposé n'est pas représentable sur n bits avec cette notation [3]

Exercice

Puisque l'opposé est une opération involutive, c'est-à-dire que la répéter fait revenir au nombre de départ, on se rend compte que retirer 1 puis remplacer tous les 0 par des 1 et vice-versa donne aussi l'opposé. Exercice pour le lecteur : prouver ceci, ainsi que la propriété ci-dessus.

Comme pour les entiers naturels, le problème du dépassement arithmétique demeure. Sur n bits, ajouter 1 à $2^{n-1}-1$ donne donc -2^{n-1} et non 2^{n-1}. Cela donne en particulier des calculs modulo 2^n.

Pour pallier ce problème, lorsque des grands nombres sont requis, et pour éviter de faire des opérations sur des nombres comprenant trop de chiffres, on coupe les nombres en paquets de $\lfloor \frac{n}{2} \rfloor - 1$ bits [4]. Les langages de programmations disposent généralement d'un type appelé "long" [5], qui reprend cette idée. Pour $n = 32$, les nombres sont donc découpés en paquets de 15, dont le premier paquet donne deux informations : le signe du nombre (premier bit) et le nombre de paquets de 15 bits (sur 15 bits). Les paquets de 15 sont stockés dans des tableaux de taille 16 (en répercutant les retenues aux bons endroits pour que le premier bit de chaque tableau soit à 0 à la fin de chaque calcul).

Le nombre le plus grand qui peut ainsi être représenté pour $n = 32$ est alors $2^{15\times(2^{15}-1)} - 1$, et son écriture tient sur environ 1 Mb. Conseil : ne pas tester à

3. Par conséquent, l'opération proposée redonnerait nécessairement le même nombre. C'est heureusement aussi le cas pour 0.

4. Ceci permet de ne pas avoir de dépassement arithmétique pour les multiplications.

5. Python 3 a le bon goût de fusionner ce type avec le type "int", et nous échappons donc généralement aux dépassements.

faire afficher le nombre le plus grand qui peut être représenté lorsque $n = 64$ [6]. En pratique, pour $n = 64$, le dépassement arithmétique interviendrait alors après un dépassement de mémoire.

1.3 Représentation des réels

Le nom donné en informatique à la représentation des nombres non entiers est la *virgule flottante* (le type correspondant est "float").

Rappelons qu'aucun nombre irrationnel ne peut être représenté de manière exacte avec des caractères de n'importe quelle base. D'ailleurs, tous les nombres représentables de manière exacte en base 2 sont en particulier décimaux, tandis que dans l'autre sens, le nombre $0,2$ par exemple, aussi innocent soit-il, a un développement infini en base 2, à savoir $\overline{0,001100110011\ldots}^2$.

La représentation des nombres réels, qui est habituellement approximative, se fonde sur la base 2 et sur l'écriture dite scientifique, normalement connue pour les nombres décimaux [7].

Grâce à la représentation à virgule flottante, les nombres très grands ou très petits peuvent s'écrire sans une surcharge de caractères peu représentatifs au niveau de la virgule. De toute façon, les limites de la mémoire imposent de procéder rapidement à un arrondi.

Selon la norme IEEE754, avec 64 bits, un nombre en virgule flottante est alors un produit $(-1)^s \times m \times 2^n$, où s est bit de signe, m est la *mantisse* $\overline{1,b_1b_2\ldots b_{52}}^2$ (puisque le 1 est systématique, il n'entre pas dans la représentation qui est donc sur 52 bits) et n est un entier relatif entre -1022 et 1023 écrit sur 11 bits et représenté comme $n + 1023$. Avec 32 bits, la taille de la mantisse passe à 23 bits et l'exposant est sur 8 bits. La disposition des bits est la suivante : d'abord s, puis n, puis m.

On notera que pour comparer deux nombres écrits en virgule flottante, on regarde d'abord le signe, puis à même signe on regarde l'exposant, puis à même exposant on regarde la mantisse.

---

6. L'ordinateur pourrait remplacer avantageusement un radiateur en panne...

7. Rappel : l'écriture scientifique revient à écrire un nombre non nul, éventuellement en tant qu'arrondi, sous la forme $x \times 10^k$, où $k \in \mathbb{Z}$ et $1 \leq x < 10$, et l'écriture est unique.

Les valeurs non utilisées pour les exposants correspondent aux mots de 11 bits 00000000000 et 11111111111. On les réserve pour des valeurs exceptionnelles, qui ne sont pas à connaître. On ne citera pour la culture que le zéro, qui existe en version positive et en version négative (suivant le premier bit), dont les bits de l'exposant et de la mantisse sont tous nuls, et les infinis, qui s'obtiennent quand tous les bits de l'exposant sont à 1 et ceux de la mantisse à 0.

1.4 Conséquences

Il est évident que sur 64 bits, même en ne considérant aucune valeur exceptionnelle, on ne pourrait encoder que 2^{64} valeurs, et les nombres supérieurs à 2^{1024} en valeur absolue (une valeur qui reste certes déraisonnable en sciences) ainsi que ceux qui sont plus proches de zéro que 2^{-1022} n'auraient pas de représentation possibles.

En pratique, on traite les dépassements arithmétiques (et les « soupassements arithmétiques », le deuxième cas) de différentes façons, précisément à l'aide des valeurs exceptionnelles (d'où l'intérêt d'avoir un zéro positif et un zéro négatif, au passage). Le déclenchement d'erreurs à l'aide de valeurs exceptionnelles non évoquées est une façon de faire, peut-être plus efficace que l'utilisation systématique de zéros ou d'infinis seulement.

Dans l'intervalle où les réels sont représentables, la question de l'arrondi se traite de la même façon que pour les nombres usuels : si le premier bit dépassant la taille de la mantisse devrait être un 0, on arrondit le dernier bit par défaut, sinon par excès avec propagation éventuelle de retenue, à l'exception notable du cas où seul un bit à 1 dépasse la taille de la mantisse, ce qui est analogue à un arrondi d'un demi-entier à l'entier le plus proche.

Le fait d'arrondir des valeurs, quand bien même est-il nécessaire, favorise une perte d'informations lorsque des opérations sont effectuées entre des réels de valeurs très éloignées. Par conséquent, en tenant compte des priorités opératoires (et quand la priorité est la même, en effectuant les opérations de gauche à droite), deux expressions donnant dans la réalité le même résultat donneront parfois sur l'ordinateur des résultats différents.

Exemple : Si on calcule $1 + 10^{-100}$, la mantisse restera de 1 car 10^{-100} vaut environ 2^{-332}, ce qui est négligeable devant 1. Dans ce cas, $1 + 10^{-100} - 1 = 0$, alors que $-1 + 1 + 10^{-100}$ donne bien entendu 10^{-100}, ou tout du moins une valeur proche

exprimée en virgule flottante.

Ce qu'il faut alors garder à l'esprit, c'est que les tests d'égalité entre réels sont très peu pertinents et qu'on leur préfèrera une comparaison de type « la différence est suffisamment petite en valeur absolue ».

Annexe : technique de conversion de réels

La technique présentée ici fonctionne pour passer de n'importe quelle base à n'importe quelle autre. Pour autant, le passage préalable par la base 10 est un détour rassurant que la majorité choisira.

Sachant que la conversion de la partie entière et de la partie fractionnaire peut se faire séparément, on sépare dans un souci de simplification le travail en deux étapes analogues, dont le résumé est de procéder à des multiplications par la base à un certain exposant jusqu'à avoir un nombre entre 1 (inclus) et la base (exclue), puis de tronquer la partie entière qui est un nouveau chiffre dans l'écriture en base b.

Ainsi donc, soit l'entier x qu'on veut écrire en base b. Le chiffre des unités en base b de x est x modulo b, c'est-à-dire le reste dans la division euclidienne. Une fois ce chiffre écrit, on considère à présent le quotient de x par b et on recommence jusqu'à ce que le nombre considéré devienne 0. On remarquera que la base dans laquelle x est écrit n'est jamais mentionnée, il convient simplement de faire les divisions par b dans la base dans laquelle x est écrit si on ne veut pas le convertir d'abord en décimal.

Quant à la conversion de la partie fractionnaire d'un nombre, soit y entre 0 (inclus) et 1 (exclu). Ce nombre s'écrira $0,\ldots$ en base b. Le premier chiffre après la virgule, et tous ceux qui suivent jusqu'à épuisement (ou jusqu'à tomber sur zéro), s'obtient ainsi : on multiplie y par b, on reporte la partie entière puis on recommence en considérant la partie fractionnaire.

Exemple : On veut écrire 10π en binaire sur 16 bits, en écriture scientifique tant qu'à faire ; le passage à l'écriture en virgule flottante est une étape supplémentaire mais indépendante de cette technique.

On anticipe le nombre de chiffres à utiliser dans l'approximation de π en base $10\ldots$ au hasard 11. Partons donc de $31,415926535$. La partie entière est 31, ce qui est congru à 1 modulo 2, donnant 1 pour bit des unités. On continue avec le quotient,

qui est 15. En pratique, on va avoir cinq fois de suite un reste de 1 dans les divisions euclidiennes effectuées, jusqu'à tomber sur 0, ce qui donne $\overline{11111}^2$ pour partie entière.

Passons à la partie fractionnaire, à savoir $0,415926535$. On va détailler les étapes :

- $0,415926535 \times 2 \simeq 0,83185307$ (on retire un chiffre en raison de la perte de précision), on écrit le 0 ;
- $0,83185307 \times 2 \simeq 1,6637061$, on écrit le 1 et on le retire ;
- $0,6637061 \times 2 \simeq 1,327412$, on écrit le 1 et on le retire ;
- $0,327412 \times 2 \simeq 0,65482$, on écrit le 0 ;
- $0,65482 \times 2 \simeq 1,3096$, on écrit le 1 et on le retire ;
- $0,3096 \times 2 \simeq 0,619$, on écrit le 0 ;
- $0,619 \times 2 \simeq 1,23$, on écrit le 1 ;
- $0,23$ étant proche d'un quart, on en déduit qu'il y aura ensuite deux 0 et au moins trois 1, donc on arrête le travail ici et on écrit notre nombre final : $\overline{11111,011010100111\ldots}^2$ (sans décider de l'arrondi).

1.5 L'essentiel

Les deux enseignements de ce chapitre sont les suivants : quand on regarde dans les détails, on tombe toujours sur le binaire, et la représentation de l'information doit être aussi efficace que possible, y compris dans la clarté de cette représentation.

Il ne faut alors pas se tromper au niveau de l'interprétation d'une chaîne de bits, ce qui veut aussi dire qu'il ne faut pas confondre complément à deux et virgule flottante.

Une bonne méthode pour ne pas se tromper est simplement de tenter de faire une opération de base sur les représentations : addition de deux entiers, multiplication de deux réels. Si cela semble infernal, c'est peut-être que la représentation est erronée. Il reste le souci de l'encodage de l'exposant selon la norme IEEE754, qui n'est ni intuitif ni particulièrement optimal par rapport à d'autres choix possibles. Dans ce cas, la mémorisation peut s'imposer.

En tout état de cause, si les confusions demeurent, autant ne retenir que la virgule flottante, car ce sont avec les flottants que la plupart des calculs effectifs sont faits.

Et bien entendu, la moralité qui sera rappelée à chaque occasion : les flottants portent en eux une imprécision qui peut être une cause d'erreurs à tout moment, car on ne pourra jamais encoder un ensemble infini dans un espace fini.

TD 1 : Calculs avec des bases numériques

Opérations arithmétiques

Les réponses doivent être données dans la base commune aux termes ou facteurs.

(1) $\overline{45}^{7} + \overline{654}^{7}$

(2) $\overline{21}^{3} \times \overline{1021}^{3}$

(3) $\overline{CAFE}^{16} + \overline{BABE}^{16}$

(4) $\overline{665}^{8} \times \overline{404}^{8}$

Conversions

Convertir...

(5) $\overline{607}^{12}$ en base 6

(6) $\overline{DADA}^{16}$ en base 9

(7) $\overline{101010}^{2}$ en base 3

(8) $\overline{3012}^{4}$ en base 7

Réponses

(1) : $\overline{1032}^{7}$; (2) $\overline{22211}^{3}$; (3) $\overline{185BC}^{16}$; (4) $\overline{335724}^{8}$; (5) $\overline{4011}^{6}$; (6) $\overline{84761}^{9}$; (7) $\overline{1120}^{3}$; (8) $\overline{402}^{7}$.

TD 2 : Représentation des nombres

Avant toute chose, il est recommandé de faire les exercices du cours.

Représentation des entiers

Exercice 1 : Représenter en complément à deux sur 16 bits les nombres 36000 et -42.

Exercice 2 : Quel nombre décimal s'écrit sur 8 bits en complément à deux 11010110 ? Et 11000011 ?

Exercice 3 : On considère un nombre dont la représentation en complément à deux commence par 0, comporte un nombre pair de bits et est un palindrome (c'est-à-dire qu'il se lit de la même façon dans les deux sens). Déterminer deux facteurs premiers du nombre en question. A-t-on une équivalence ?

Représentation des réels

Exercice 4 : Convertir en binaire 0,7 en s'arrêtant une fois la période trouvée[8] et exprimer ce nombre en virgule flottante sur 16 bits (pour ne pas écrire trop d'informations), avec 5 bits d'exposant et 10 bits de mantisse.

Exercice 5 : Même exercice avec $\frac{3}{7}$.

Au passage, utiliser la représentation en virgule flottante pour des rationnels introduit une perte d'information en raison des approximations. C'est pour cela qu'en Python, il est possible de représenter aussi les rationnels comme des couples d'entiers (numérateur et dénominateur), ce qui est géré par un module nommé `fractions`.

Exercice 6 : Caractériser les entiers qui ont une représentation exacte en virgule flottante sur 64 bits.

8. Pour rappel, d'après le cours, elle est de taille 1, 2 ou 4. Exercice supplémentaire : retrouver pourquoi.

Chapitre 2

Algorithmique et programmation I

2.1 Bases de la programmation

2.1.1 Les données et leurs types

La notion de *type* est essentielle en programmation : elle indique quel genre de données on manipule. On peut voir un type comme une catégorie d'objets mathématiques sur lesquels des opérations spécifiques peuvent être effectuées. Un exemple issu de la géométrie : un point dans le plan est la donnée de deux réels qui sont ses coordonnées cartésiennes [1], une droite est la donnée de deux points par lesquels elle passe ; ce qui est possible avec des points (déterminer le milieu du segment entre les points, calculer une distance, ...) ne l'est pas forcément avec des droites (déterminer l'intersection, calculer l'angle, ...) et vice-versa. Les types simples de base en informatique sont les entiers, les flottants [2], les *booléens* (vrai et faux, qui permettent de faire du calcul et de la logique [3]) et les *caractères* [4]. Les séquences de base sont les *n-uplets*, les *listes* et les *chaînes de caractères.* [5].

1. un autre système de coordonnées est bien entendu possible
2. que peu de langages rassemblent avec les entiers dans un type « nombre »
3. même si la plupart des langages se permettent de faire de la logique avec des entiers
4. qui n'ont pas d'existence propre en Python, mais on va faire comme si
5. Il est tout à fait vrai de rétorquer que les chaînes de caractères forment un type simple, mais leur gestion en Python est proche de celle des n-uplets (qui ne sont d'ailleurs étonnamment pas explicitement au programme).

Commençons par les opérations sur les entiers en Python[6]. L'addition se fait avec `+`, la soustraction avec `-`[7], la multiplication avec `*` **qui ne peut pas être implicité**, le quotient de la division euclidienne avec `//`, le reste de la division euclidienne avec `%` et la puissance avec `**`.

Attention : ne pas utiliser pour la puissance l'opérateur `^` (voir ci-après).

Pour écrire un flottant, on utilise des chiffres et un point en guise de séparateur décimal.[8] On peut omettre la partie à gauche ou à droite de la virgule si elle vaut zéro, mais zéro ne peut pas s'écrire simplement `.`, ce sera `0.` ou `.0`. Les opérations sur les flottants sont sensiblement les mêmes, on retrouve `+`, `-`, `*` et `**`, et comme la notion de division euclidienne n'a pas d'intérêt[9], la division se fait par `/` ; elle retourne un flottant même si elle concerne deux entiers (en Python3, du moins).

Les conversions se font à l'aide des fonctions `int()` (troncature), `float()` (conversion d'un entier en lui-même virgule zéro), `math.floor()` (arrondi par défaut, donc la partie entière) et `math.ceil()` (arrondi par excès), les deux dernières nécessitant de charger le module `math` par l'instruction `import math`. L'arrondi (transformant un flottant en flottant) se fait par la fonction `round`, qui prend deux arguments : le nombre à arrondir et le nombre de chiffres après la virgule[10].

Les priorités opératoires sont les mêmes que celles appliquées en mathématiques. Les parenthèses peuvent être utilisées et imbriquées, mais on ne peut pas les remplacer par des crochets.

Comme vu dans le chapitre 1, les flottants étant représentés de manière approchée, l'addition n'est pas associative, mais reste commutative.

---

6. Il va sans dire qu'on se réfèrera au chapitre précédent pour la façon dont l'ordinateur gère les nombres.

7. qui est aussi utilisé pour obtenir l'opposé, contrairement à la calculatrice qui dispose de deux touches différentes

8. Là où nous utiliserions une virgule, Python applique bien entendu la convention anglo-saxonne.

9. bien que les opérateurs `//` et `%` fonctionnent, avec un comportement intuitif

10. Ce deuxième argument est optionnel et l'omettre donnera un arrondi à l'entier le plus proche, par ailleurs la fonction ne retournera pas un flottant mais un entier. Si le deuxième argument est négatif, on obtient un arrondi à la dizaine la plus proche, ou la centaine, etc.

Les booléens sont `True` et `False`[11]. On les utilise notamment pour des tests d'arrêts de boucles `while` et pour des instructions contenant `if`. Les opérations associées sont la *conjonction* `and`, la *disjonction* `or`[12], le *test de différence* `^` (aussi appelé « ou exclusif » et raccourci "xor" en anglais) et la *négation* `not`. La négation est prioritaire sur la conjonction qui est prioritaire sur la disjonction. Ainsi, `True or not True and False` se parenthèse automatiquement `True or ((not True) and False)` et donne `True`. En outre, il est important de souligner que Python[13] *évalue paresseusement* les expressions booléennes, ce qui signifie que quand l'expression `a and b` est évaluée (avec `a` et `b` deux expressions), si `a` se révèle fausse, `b` n'est même pas évaluée ; la conséquence est que `False and 1/0 == 42` ne provoquera pas d'erreur de division par zéro. Il en va de même pour `a or b`, et c'est notamment problématique de l'oublier si l'expression `b` concerne une fonction avec effet de bord.

Les booléens peuvent s'obtenir comme résultat de comparaisons[14] : `<`, `>`, `<=`, `>=`, `==` (test d'égalité, à ne pas utiliser avec un seul égal), `!=` (test de différence). On peut enchaîner les comparaisons[15], le résultat sera vrai si toutes les comparaisons isolées sont vraies. Les comparaisons sont prioritaires sur les opérations booléennes.

Quelques résultats à connaître au cas où :

- `42 == 42.0` est vrai, de même pour tous les entiers.
- D'ailleurs, `True == 1` est vrai, de même que `False == 0`, `True == 1.0` et `False == 0.0`.
- Du coup, `True + True` vaut 2 et `False < True` est vrai.[16]
- On peut (mais on ne doit pas) utiliser une chaîne de caractères dans les tests. Bien que `False == ""` soit faux, la chaîne vide est la seule qui s'assimile à `False` si on l'utilise dans un test.[17]

11. attention aux capitales, qui sont nécessaires
12. Les opérateurs `&` et `|` sont leurs équivalents bit-à-bit pour des entiers, ils ne sont pas à connaître et peuvent aussi être utilisés pour les booléens. Leur utilisation est dangereuse, car d'une part ils ne sont pas paresseux, et d'autre part ils sont prioritaires sur les comparaisons.
13. comme beaucoup d'autres langages
14. de deux entiers, ou deux flottants, voire deux objets de type quelconque
15. ce serait parfois une aberration en mathématiques, mais on le fera là aussi presque exclusivement pour des tests d'intervalles
16. En fait, le type booléen de Python hérite du type entier.
17. Quoi qu'il en soit, il faut éviter de confier à Python le soin de comprendre quelque chose qui n'est pas un booléen comme un booléen.

2.1.2 Variables

Les *variables* en programmation sont assimilables à leur pendant mathématique : « Soit x un réel... ». Ici, il faut donner une valeur, de n'importe quel type, à une variable dès sa création (c'est-à-dire sa première mention, qui est donc nécessairement une affectation) afin de l'utiliser. L'*affectation* d'une variable se fait par le symbole `=`, avec nécessairement le nom de la variable à gauche[18]. Le principe d'une affectation est que la partie à droite du symbole `=` est évaluée, puis le résultat est stocké dans la variable à affecter. Rien n'interdit que le type d'une variable change lors d'une réaffectation, qui consiste simplement à écraser l'ancienne valeur de la variable et ne plus en tenir compte. Ainsi, une réaffectation peut impliquer l'ancienne valeur d'une variable. Voir le code ci-dessous, dont la partie après chaque croisillon[19] est appelée *commentaire* et ignorée par Python :

- `a = 4`
- `a = a + 3 # a + 3 est évalué à 7, donc a vaut maintenant 7`
- `a *= 4 # raccourci pour a = a * 4`

Remarque : Pour aller plus loin, les instructions `a *= 4` et `a = a * 4` ne sont pas tout à fait équivalentes, mais la nuance n'est pas apparente et il n'est pas pénalisable de l'oublier. En fait, `a *= 4` n'est pas une réaffectation comme les autres, mais plutôt une mise à jour de l'ancienne valeur par une opération. En outre le raccourci `++` ou `-` pour ajouter ou retirer un n'existe pas en Python.

Un peu de théorie... Dans l'ordinateur, une variable est une zone mémoire accessible en lecture et en écriture. L'état du système est alors l'ensemble des variables définies à un instant donné de l'exécution d'un programme. Toute expression contenant une variable s'évalue en remplaçant les noms de variables par leur valeur dans l'état courant[20]. Déclarer[21] ou réaffecter une variable modifie donc l'état.

2.1.3 Séquences

Les séquences sont des objets en Python qui possèdent une *taille*, sont *itérables* (c'est-à-dire qu'on peut les parcourir), *indexables* (c'est-à-dire qu'on peut accéder à chacun

18. Le nom d'une variable est composé de lettres, qu'on évitera d'accentuer, de chiffres (mais pas le premier caractère) et de caractères de soulignement.
19. le symbole #, qui ne se lit pas « dièse » et encore moins « hashtag »
20. déclenchant au passage une erreur lorsqu'une des variables n'est pas encore affectée, c'est-à-dire qu'elle n'apparaît pas dans l'état
21. c'est-à-dire créer et affecter

de leurs éléments au moyen d'un indice) et *tranchables* (tentative personnelle de traduction de "sliceable" ; la notion de slice sera vue ci-après).[22] Pour plus d'informations sur ce vocabulaire, voir `http://sametmax.com/les-trucmuchables-en-python/`.

Un n-uplet ("tuple", en anglais) est une collection d'éléments séparés par des virgules (on utiliserait des points-virgules en français), éventuellement entourée de parenthèses.[23] La fonction `len()` donne la taille du n-uplet, et fonctionne en fait pour toutes les séquences. On accède au `i`-ième élément du n-uplet `t` (très souvent stocké dans une variable, mais la syntaxe suivante marche dans tous les cas) par `t[i]`, où les indices commencent à 0, avec une erreur si `i` est supérieur ou égal à la taille de `t`. On peut aussi accéder à un élément à partir de la fin : `t[-1]` est le dernier élément, et ainsi de suite jusqu'à `t[-len(t)]` qui est le premier élément (et là aussi, un indice strictement inférieur provoque une erreur). Enfin, on peut accéder à une tranche (le fameux *slice*) d'un n-uplet en écrivant `t[i:j]`, ce qui donne un k-uplet formé de `t[i]`, `t[i+1]`, ..., `t[j-1]`, en remplaçant tout nombre négatif par l'indice positif correspondant, à la manière décrite à la phrase précédente. Ce k-uplet est vide si `t[i]` est à droite de `t[j-1]` et tronqué aux extrémités de `t` sans provoquer d'erreur si i ou j débordent de la taille.

On ne peut accéder aux éléments d'un n-uplet qu'en lecture, il est juste possible de réaffecter un n-uplet dans son ensemble si on veut le modifier. Les n-uplets peuvent être déconstruits : on écrira `(a, b, c) = (3, 2, 1)`[24] pour affecter d'un coup trois variables (attention à la lisibilité, tout de même). Le fait que la partie à droite du signe `=` soit évalué avant que les affectations soient effectuées permet de trouver une méthode élégante pour échanger le contenu de deux variables. Le test d'appartenance se fait à l'aide de l'opérateur infixe[25] `in` : `2 in (1, 4, 2, 5)` donne `True`. Cet opérateur caractérise en fait des objets dits *conteneurs* (en anglais "container").

Les n-uplets peuvent fusionner entre eux avec l'opérateur `+`. Ils peuvent être reproduits avec l'opérateur `*` utilisé avec un entier à gauche ou à droite.

Attention cependant aux parenthèses : `1, 2 + 3, 4` donne `(1, 5, 4)` tandis que

22. Certains objets n'ont pas toutes ces propriétés, par exemple les ensembles ne sont pas indexables, et il existe des itérables infinis donc sans taille.

23. Le seul 0-uplet existant est `()`, déjà vu en tant qu'argument vide. Un 1-uplet s'écrit de manière spéciale en ajoutant une virgule après l'élément unique. Son utilité est de pouvoir faire des opérations de n-uplet avec un seul élément.

24. dans cet exemple, les parenthèses sont optionnelles des deux côtés

25. c'est-à-dire qu'on l'insère entre ses deux arguments

`(1, 2) + (3, 4)` donne `(1, 2, 3, 4)` ; de même, `3 * 1, 2` donne `(3, 2)`, tandis que `3 * (1, 2)` donne `(1, 2, 1, 2, 1, 2)`.

Une chaîne de caractères se manipule de la même manière qu'un n-uplet. En particulier, il est impossible de modifier un caractère isolé d'une chaîne. Une chaîne de caractères est entourée d'apostrophes ou de guillemets, simples ou triples, l'utilisation de délimiteurs triples permettant d'aller à la ligne en écrivant la chaîne. En outre, si on veut utiliser une apostrophe dans une chaîne, il suffit d'utiliser un autre délimiteur que l'apostrophe pour ne pas avoir de souci, etc (ou échapper l'apostrophe à l'intérieur de la chaîne).

Le test d'appartenance avec `in` peut détecter des facteurs d'une chaîne, c'est-à-dire des chaînes formés de caractères consécutifs d'une chaîne, à la manière des parties de n-uplets. Comparer les résultats des tests `"23" in "123456"`, `"24" in "123456"`, `(2, 3) in (1, 2, 3, 4, 5, 6)` et `(2, 3) in (1, (2, 3), 4, (5, 6))` [26].

Une liste [27] est une collection d'éléments, similaire à un n-uplet, à la différence notable près que cette fois-ci les éléments peuvent être modifiés individuellement. On crée une liste en donnant ses éléments séparés par des virgules et en entourant le tout de crochets au lieu des parenthèses du n-uplet.

Contrairement à d'autres langages, Python ne permet pas de créer un élément dans une liste à un indice trop élevé. Par exemple, si on crée la liste `a = [1, 2, 3]`, elle est de taille 3 et on ne peut pas créer un quatrième élément en écrivant `a[3] = 4`. Il reste cependant possible d'écrire `a += [4]` [28]. Tout ce qu'on a présenté sur les n-uplets fonctionne encore sur les listes.

Passer d'une séquence à une autre peut se faire à l'aide d'un algorithme parcourant les éléments, mais il existe aussi des fonctions pour cela : `list(a)` retourne la liste des éléments de la séquence `a`, donc soit la liste des caractères d'une chaîne soit la liste des éléments d'un n-uplet, et `tuple(a)` retourne le n-uplet des éléments de la

26. constater au passage que certaines parenthèses sont importantes en retirant celles à gauche de `in`

27. Il faut éviter de dire « tableau », car en Python, contrairement à d'autres langages, on appelle « tableau » une structure, absente de la bibliothèque standard, ressemblant à une liste mais contenant préférablement uniquement des entiers ou uniquement des flottants, comme nous le verrons au chapitre suivant.

28. auquel on préfèrera si possible `a.append(4)` pour un élément et `a.extend([4, 5, 6])` pour un itérable regroupant un nombre arbitraire d'éléments

séquence `a`. En revanche, si `a` est une liste ou un n-uplet `str(a)` entoure simplement `a` de guillemets [29], un algorithme (ou la méthode `join`) est nécessaire pour convertir une liste ou un n-uplet de caractères en la chaîne correspondante.

2.1.4 Instructions composées

Séquence d'instructions

Enchaîner deux instructions se fait naturellement, en allant à la ligne entre les deux. Il serait valide d'enchaîner deux instructions sur une même ligne, en les séparant d'un point-virgule, mais on évitera pour la propreté du code de recourir à cela.

Disjonction de cas (if) À NE SURTOUT PAS QUALIFIER DE BOUCLE

Une *disjonction de cas*, ou test, correspond à l'instruction de type **Si ... Alors ... Sinon ...**. Elle s'introduit en Python par le mot-clé `if`, suivi de la condition, qui est interprétée comme un booléen, puis d'un **double-point en fin de ligne**.

La partie correspondant au **Alors**, qui est exécutée lorsque la condition est remplie, est délimitée par ce qu'on appelle **l'indentation** : toutes les lignes dans le corps du test doivent commencer par le même nombre, strictement positif[30], d'espaces par rapport à la ligne du `if`.[31] Sans cela, Python déclenche une erreur de syntaxe[32].

La partie correspondant au **Sinon** peut s'introduire par le mot-clé `else`, suivi d'un **double-point**, avec là aussi une indentation spécifique pour le corps de cette partie, ou par le mot-clé `elif`[33], suivi d'une autre condition et d'un **double-point**, toujours avec une indentation spécifique.

Un bloc avec `elif` est exécuté si aucune des conditions précédentes n'est remplie alors que la condition actuelle l'est ; un bloc avec `else`, nécessairement en toute fin s'il existe, est exécuté si aucune des conditions n'est remplie.

Par exemple, si on veut affecter à la variable `couleur` la valeur `"rouge"` si une autre

29. si `a` est composé de chaînes de caractères, Python gère les conflits de délimiteurs
30. recommandation officielle : quatre
31. On dit que Python est un langage à indentation signifiante.
32. et quelqu'un qui lirait un programme dans un autre langage serait fâché ; en pratique, la première ligne de code pourrait être directement après le double-point, mais ceci aussi est à éviter
33. qui n'est pas au programme, mais très (trop...) utilisé

variable `valeur` est strictement inférieure à 5, `"jaune"` si `valeur` vaut 5 exactement et `"vert"` sinon, on écrira :

```
if valeur < 5:
    couleur = "rouge"
elif valeur == 5:
    couleur = "jaune"
else:
    couleur = "vert"
```

Boucle inconditionnelle (for)

La *boucle inconditionnelle*, correspondant à **Pour ... de ... à ...**, s'écrit en Python comme un parcours de séquence, c'est-à-dire qu'on aura toujours un objet itérable (que ce soit une chaîne de caractères, un n-uplet, une liste, etc.) associé, dont une variable créée pour l'occasion[34] vaudra successivement chacun des éléments dans le corps de boucle.

La syntaxe est `for i in a:`, où `i` est un exemple de nom de variable et `a` est la séquence parcourue. Bien entendu, le double-point est indispensable, ainsi que l'indentation du corps de boucle.

L'exemple le plus typique est celui d'une boucle où `i` prend les valeurs entières de 0 inclus à n exclu, ce qui s'écrit en utilisant la fonction `range`, qui génère une séquence[35] de nombres en fonction de son (ou ses) argument(s).

On écrira alors `for i in range(n):`.

Au passage, si on veut parcourir une liste `l`, il est tout aussi acceptable d'introduire la boucle par `for x in l:` que `for i in range(len(l)):` en commençant le bloc par `x = l[i]`.

Le premier code, de par sa concision, est à privilégier si la connaissance des indices n'est pas essentielle. Il est par ailleurs possible de créer à la fois l'indice et la valeur en écrivant `for i, x in enumerate(l)`. Pour tout dire, il n'est pas exclu que l'utilisation systématique de la fonction `enumerate` limite les confusions entre les indices

34. et qu'on n'a pas besoin d'initialiser, dans ce cas précis
35. qui n'est pas une liste, en fait

et les valeurs.

Remarque : Python ne permet pas de perturber une boucle inconditionnelle par modification de la variable contenant la séquence à parcourir ni par une modification de la variable de boucle, mais elle peut être perturbée si on la fait muter. Constater par exemple que le code suivant imprime effectivement les nombres de 1 à 3 (alors même qu'à la fin la liste `a` est bien vide) :

```
a = [1,2,3]
for x in a:
    a = []
    print(x)
```

Tester également les codes suivants :

```
a = [1,2,3]
for x in a:
    a[2] = 42 # Qui sera imprimé
    print(x)

a = [1,2,3]
for x in a:
    del a[:] # On retire la tranche contenant tout
    print(x)

a = [1,2,3]
for x in a:
    del a # Destruction totale de la variable, plantage.
    print(x)

a = [1,2,3]
for x in a:
    a.append(x) # Oh, une boucle infinie !
    print(x)

for i in range(10):
    i = 10 # Imprimera certes uniquement des 10, mais il y en aura dix.
    print(i)
```

Boucle conditionnelle (while)

La *boucle conditionnelle*, correspondant à **Tant que ... faire ...**, utilise là aussi une condition qui est interprétée comme un booléen.

On écrit `while cond:`, où `cond` est la condition, on n'oublie toujours pas le **double-point** ni l'indentation dans le corps de la boucle, qui est exécuté tant que la condition est remplie, cette condition étant testée entre la fin de chaque passage dans la boucle et le début de l'éventuel passage suivant (mais pas pendant).

Bien souvent, la condition est une inégalité faisant intervenir une variable. L'écueil classique consiste à oublier de mettre à jour la variable dans le corps de la boucle, ce qui fait que le programme y est piégé. C'est l'une des premières choses à vérifier en cas de problème.

Une boucle conditionnelle peut aisément remplacer une boucle inconditionnelle, à savoir :

```
for x in a:
    (blablabla)
```

est équivalent (aux mutations, réaffectations et autres perturbations près) à :

```
i = 0
while i < len(a):
    x = a[i]
    i += 1
    (blablabla)
```

Bien entendu, il est possible d'imbriquer des boucles et tests l'un(e) dans l'autre à un niveau de profondeur arbitraire.

2.1.5 Fonctions

Il est possible de voir les *fonctions* en programmation comme le pendant des fonctions mathématiques : si on dispose d'une fonction telle que $f(x) = x^4 - x^3 + 2x^2 - 3x + 7$, on ne va pas écrire $\pi^4 - \pi^3 + 2\pi^2 - 3\pi + 7$ puis $e^4 - e^3 + 2e^2 - 3e + 7$ pour les images respectives de π et de e par f, mais bien $f(\pi)$ et $f(e)$. En programmation, définir

une fonction permet entre autres de ne pas avoir à recopier des morceaux de code plusieurs fois, et surtout c'est l'essence même de la programmation que de définir des procédures qui vont nécessiter le moins possible l'intervention de l'utilisateur.

Une fonction a un certain nombre d'*arguments*[36] qui sont des valeurs d'entrée sur lesquelles elle est appelée, comme π et e dans notre exemple. En Python, on définit une fonction ainsi : `def f (arg1, arg2, ...):`, où `f` est le nom de la fonction, et les arguments sont les noms arbitraires de variables dans la parenthèse. Comme pour les instructions composées, on n'oublie pas le **double-point** ni l'indentation ensuite. À l'intérieur du bloc où l'on définit la fonction, les arguments définis dans la parenthèse peuvent être utilisés tels quels.

Une fonction a aussi une *valeur de retour*, qui peut être de n'importe quel type. Cette valeur est renvoyée par la fonction quand on utilise l'instruction `return`, qui interrompt toutes les exécutions de la fonction. C'est normal : une fonction ne peut retourner quelque chose qu'une fois.

Une fois la fonction définie, elle s'invoque en écrivant `f(a, b, ...)` où tous les arguments doivent s'évaluer en une constante, c'est-à-dire que s'il s'agit d'expressions, les variables qu'elles contiennent doivent toutes être définies. Une fonction sans argument exécute simplement du code, en modifiant éventuellement des variables dites globales et en ayant éventuellement une valeur de retour (constante, aléatoire voire obtenue par des interactions). On l'appelle avec des parenthèses vides.

La fonction définie précédemment s'écrit et s'appelle alors ainsi sur l'entier 3 :

```
def f(x):
    return x**4-x**3+2*x**2-3*x+7

f(3)
```

2.1.6 Entrées et sorties

L'interaction fait partie des fonctionnalités nécessaires pour qu'un langage de programmation puisse faire tout ce qu'on attendait d'un ordinateur à l'origine. [37]

36. on dit aussi « paramètres »

37. On ne parle pas ici de la possibilité de lire du contenu média, entre autres, ce qui est d'ailleurs aussi prévu en Python. La notion ici évoquée est celle de Turing-complétude (voir annexe).

L'instruction `input(message)` affiche `message` et renvoie ce que l'utilisateur a écrit en réponse. En Python 2, le résultat était du type le plus pertinent en fonction de ce que l'utilisateur saisissait, ce qui n'est plus le cas en Python 3 : on a forcément une chaîne de caractères qu'il faut convertir soi-même en cas de besoin (notamment par la fonction `eval`).

Attention : pour les fonctions présentées ci-avant, il ne faut pas oublier les guillemets autour du message, s'il s'agit d'une chaîne de caractères ; bien entendu, si `message` est une variable, sa valeur sera imprimée par "pretty-print" (si la valeur n'est pas une chaîne de caractères).

En ce qui concerne la lecture d'un fichier, nous allons omettre pour le moment la notion de chemin (voir TP 2) et considérer que tous les fichiers sont dans le même répertoire, chargé par l'interpréteur Python.

Lorsqu'on ouvre un fichier, que ce soit comme ici en mode lecture ou en mode écriture (ou ajout), on utilise la fonction `open` et on définit une variable (c'est mieux) : `fichier = open(nom_du_fichier, "r")`, où `nom_du_fichier` est une chaîne de caractères qui contient bien entendu l'extension également. L'argument `"r"` indique que le fichier est ouvert en lecture seule ("read"), donc il n'est pas modifié.

Il n'est jamais trop tôt pour signaler qu'une fois qu'on a fini de traiter un fichier on le ferme par habitude et pour éviter les problèmes. Cela se fait simplement par `fichier.close()`, où `fichier` est la variable définie précédemment. Faute de fermeture du fichier, certains programmes refuseront temporairement d'afficher le nouveau contenu du fichier, et des conflits en écriture sont possibles.

La syntaxe est peut-être troublante, elle provient de la programmation orientée objet [38], il n'y a pas lieu de se poser trop de questions pour le moment.

Pour récupérer le contenu d'un fichier, on utilise `fichier.read()`, qui retourne la totalité du fichier [39], en tant que chaîne de caractères.

38. C'est le même principe qu'avec les séquences : on écrit `a.append(x)` pour ajouter `x` à la fin de `a`.

39. Ajouter un argument définit le nombre maximal de caractères à lire. Utiliser `readline` au lieu de `read` permet de s'arrêter en fin de la ligne actuelle, en incluant le caractère de fin de ligne, et non en fin de fichier, et `readlines` retourne la liste des lignes restantes.

La gestion de ce contenu est également détaillée dans le TP 2. On signalera simplement l'existence de `fichier.seek(position, mode)` qui place le pointeur de lecture (déplacé par `fichier.write(n)` de `n` caractères et initialement au début du fichier) au `position`-ième caractère (si `mode` vaut `0`), `position` caractères plus loin (si `mode` vaut `1`) ou au `position`-ième caractère en partant de la fin (si `mode` vaut `2`).

L'instruction `print(x)` imprime `x`, quel que soit son type. Après l'impression de `x`, un retour à la ligne est produit, ce qu'on peut modifier en ajoutant un argument `end=y` après ce qui est imprimé, de sorte que le retour à la ligne est remplacé par `y`.

Il est possible d'imprimer plusieurs choses à la fois, avec des types totalement arbitraires, qui seront autant d'arguments de la fonction `print`. L'impression se fait alors successivement en séparant deux arguments par une espace, ce qu'on peut également modifier en ajoutant, également à la fin, un argument `sep=z`.

Par exemple, tester le code suivant :

```
print("A", "B", "C", sep="-", end="!")
```

L'impression dans un fichier nécessite que celui-ci soit ouvert en écriture (auquel cas son contenu original est définitivement supprimé) ou en ajout (auquel cas ce qu'on imprime est mis à la suite).

On utilise encore la fonction `open` : `fichier = open(nom_du_fichier, "w")` ou `fichier = open(nom_du_fichier, "a")`, respectivement, ce qui crée le fichier s'il n'existait pas encore.

Une fois le fichier ouvert, on y écrit ainsi : `fichier.write(contenu)`, la fonction ayant par ailleurs la particularité d'avoir une valeur de retour, à savoir le nombre de caractères écrits. Cela peut toujours servir...

2.2 Preuves de programmes et d'algorithmes

Écrire un programme, c'est bien. Écrire un programme qui marche, c'est mieux ! Le problème qui se pose, passée l'étape des erreurs de syntaxe, est de savoir d'une part si un programme qu'on a écrit termine effectivement sur toute entrée, ou du moins sur toute entrée pour laquelle on veut qu'il termine, et lorsqu'il termine s'il fait effectivement ce que l'on attend de lui.

Les preuves de terminaison et de correction sont souvent proches de raisonnements par récurrence, et la plupart du temps elles ne se compliquent que lorsque le programme fait intervenir des boucles. [40]

2.2.1 Preuves de terminaison

Commençons par une bonne nouvelle : une boucle inconditionnelle (`for`) termine toujours en Python, sauf en cas de mutation de l'itérable à parcourir ou si celui-ci est de taille infinie, ce qui constitue généralement une faute de goût.

Par ailleurs, écrire par exemple `for i in range(2, 5, -1)` donnera une boucle qui n'est jamais exécutée (l'itérable est en fait vide) et `for i in range(42, 45, 0)` donnera une erreur car Python n'accepte pas zéro en troisième argument de `range`.

Remarque : Ce n'est pas le cas de tous les langages, le code PHP suivant produit une boucle infinie : `$n = 1; for ($i = 0 ; $i < $n ; $i++) $n++;`

En pratique, nous allons nous concentrer sur les preuves de terminaison de programmes utilisant une boucle conditionnelle (`while`).

La théorie mathématique que l'on peut utiliser dans pour ainsi dire tous les cas est la suivante : l'ensemble $\mathbb{N}$ est bien fondé, c'est-à-dire qu'il n'existe pas de suite infinie strictement décroissante d'entiers naturels. Pour prouver qu'une boucle conditionnelle termine, il suffit de trouver une expression dépendant des variables du programme et qui s'évalue en un entier décroissant strictement à chaque tour dans la boucle.

Une telle expression, appelée *variant*, est souvent assez rapide à trouver.

Prenons pour exemple un morceau du programme, écrit plus tôt dans ce chapitre, qui simule une boucle inconditionnelle (ça tombe bien...) :

```
i = 0
while i < len(a):
    x = a[i]
    i += 1
```

40. Oui, parce qu'en fait une suite d'instructions simples n'est pas vraiment difficile à analyser !

On prend pour variant `len(a)-i`. Puisque la boucle est quittée dès que `i` ne sera plus strictement inférieur à `len(a)`, on peut considérer que cette expression est toujours un entier naturel. Ensuite, puisque `a` n'est jamais modifié, sa taille non plus à plus forte raison, alors que `i` est augmenté d'un à chaque tour dans la boucle. Par conséquent, le variant décroît strictement à chaque tour, ce qui garantit la terminaison du programme.

Un deuxième exemple, venu du futur (en l'occurrence, la section suivante), est la fonction qui calcule la somme des chiffres d'un entier naturel représenté en base 10 :

```
def somme_chiffres(n):
    n = abs(n) # je n'aime pas les négatifs !
    s = 0
    while n > 0:
        s = s + (n % 10)
        n = n // 10
    return s
```

Cette fois-ci, on prend simplement pour variant `n`, qui est positif d'après la première ligne et dont la décroissance est garantie puisqu'on procède à une division euclidienne par 10 d'un nombre strictement positif (par hypothèse) en fin de boucle.

Nous verrons plus tard que les variants peuvent dépendre de variables qui ne sont pas des entiers, comme des réels (recherche de zéro d'une fonction) ou des séquences (le variant concerne alors souvent leur taille).

2.2.2 Preuves de correction

Une fois qu'on a prouvé que le programme terminait [41], il faut encore établir qu'il fait ce pour quoi on l'a écrit. [42]

Tester sur plusieurs entrées, notamment des cas limites, est une solution assez efficace en pratique, mais très peu rigoureuse du point de vue scientifique. On préfèrera écrire une preuve formelle du programme, ce qui sera relativement facile dans les cas étudiés dans ce cours, mais qui peut être très contraignant dans la vie réelle. [43]

41. et même dans les cas où il ne termine pas, en fait
42. On parle de *spécification* d'un programme : dire à quoi il devrait servir.
43. Une troisième approche existe, c'est la construction de modèles, donnant lieu à une branche

Comme pour la terminaison, un programme sans boucle est relativement aisé à prouver. Cette fois-ci, en revanche, les boucles inconditionnelles devront être étudiées, et on les traitera comme les boucles conditionnelles.

L'outil présenté ici est l'*invariant de boucle*. Il s'agit d'une propriété dont on veut prouver qu'elle est vraie en entrant dans une boucle et qui le demeure à chaque tour jusqu'à la sortie de la boucle. L'invariant devra être pertinent, de sorte qu'en le prouvant on aura prouvé que le programme est correct. [44]

Prenons encore l'exemple de la fonction `somme_chiffres`. L'invariant de boucle que nous allons utiliser est :

La variable `s` contient la somme des `i` derniers chiffres de l'argument de la fonction à la fin du `i`-ième tour.

Cet invariant sera difficile à prouver tel quel, donc nous allons l'étendre en précisant :

... et de plus la variable `n` contient $[\frac{|x|}{10^i}]$ (où `x` est l'argument de la fonction) à la fin du `i`-ième tour [45].

Comme ce sera souvent le cas, il s'agit de faire une récurrence.

On suppose l'argument de la fonction non nul, pour que la boucle soit parcourue au moins une fois.

- À la fin du 0^e tour, c'est-à-dire avant d'entrée dans la boucle, la variable `s` contient bien `0` et la variable `n` contient bien le nombre en entrée.
- Supposons l'invariant vrai à la fin du `k`-ième tour dans la boucle. Alors à la fin du tour suivant on divise `n` par `10`, ce qui fait que la deuxième partie de l'invariant est héréditaire, et de plus on a entre temps ajouté à `s` le chiffre des unités de `n`, qui est le `(k+1)`-ième chiffre de l'argument en partant de la fin, d'après la deuxième partie de l'invariant et la définition du `i`-ième chiffre d'un nombre écrit dans n'importe quelle base. Par conséquent, la variable `s` contient bien la somme des `k+1` derniers chiffres de l'argument de la fonction

---

de l'informatique appelée *model-checking* en anglais.

44. Pour aller plus loin, les preuves de programmes utilisent en général la logique de Hoare, bien plus forte que les invariants de boucle.

45. ... existant si `n` ne valait pas encore `0` au tour précédent, pour être totalement rigoureux

à la fin du `(k+1)`-ième tour.
— En sortie de la boucle, `n` vaut `0` et donc le nombre de tours est le nombre de chiffres de l'argument de la fonction, qui est un de plus que la partie entière du logarithme décimal de ce nombre. Ainsi, `s`, qui est aussi la valeur retournée, vaut bien la somme des chiffres de l'argument, une fois la boucle terminée, ce qui prouve la correction du programme.

2.3 Complexité

Il apparaît très vite que, étant donné un problème, on peut difficilement compter le nombre de programmes qui le résolvent.[46] En fait, même sans tenir compte de programmes différents implémentant un algorithme commun, cesdits algorithmes peuvent être très variés, et on les discriminera en fonction de leur efficacité, c'est-à-dire leur coût en opérations élémentaires et l'espace mémoire qu'ils utilisent.

Ceci est exactement la notion de *complexité*, respectivement en temps et en espace. En fait, dans la mesure où les ordinateurs ont une capacité de plus en plus grande, la complexité en espace est un critère moins prépondérant que la complexité en temps.[47]

Prenons un exemple pratique avec la méthode de Horner pour déterminer l'image d'un réel par une fonction polynomiale.

En écrivant un programme qui calcule $a_n x^n + a_{n-1} x^{n-1} + \cdots + a_0$, le nombre d'additions est de n et le nombre de multiplications est de $n + (n-1) + \cdots + 0$, soit $\frac{n(n+1)}{2}$.

La complexité en espace est la taille de la réponse finale, ou plus précisément la plus grande taille d'une valeur intermédiaire.

On peut améliorer un tel programme en se rendant compte que la valeur de x^i est recalculée pour tous les monômes de degré supérieur. En stockant les valeurs de tous les x^i, de sorte de ne les calculer qu'une fois par $n-1$ multiplications, on fait alors passer le nombre de multiplications à $2n-1$. Cette amélioration nette de la complexité en temps nécessite malheureusement un coût en espace supplémentaire pour le stockage des valeurs des x^i.

46. Et pourtant ils sont en nombre dénombrable !
47. Les problèmes viennent surtout du temps exponentiel, l'espace restant souvent raisonnable.

La méthode de Horner consiste à se rendre compte que

$$a_n x^n + a_{n-1}x^{n-1} + \cdots + a_1 x + a_0 = \underbrace{(\ldots((}_{n-1\text{ '('}} a_n x + a_{n-1})x + a_{n-2})\ldots)x + a_1)x + a_0,$$

et donc à réaliser seulement n multiplications et n additions, sans utiliser d'espace mémoire supplémentaire.

L'exemple ci-dessus permet de voir que la complexité se fait en fonction des paramètres du problème (ici n, mais aussi les coefficients du polynôme qui détermineront la taille des nombres stockés, qui influe sur le temps demandé pour faire les opérations arithmétiques), et elle soulève la question de l'unité exacte employée : on ne peut pas parler de temps alors même que les performances de l'ordinateur peuvent causer des écarts de temps bien plus importants entre deux programmes. La première définition mentionnait les opérations élémentaires, or une multiplication est toujours plus coûteuse qu'une addition pour un processeur. Ainsi, il s'agit de retenir un critère pertinent et adapté au problème.

Pour de nombreux algorithmes, notamment ceux qui sont vus en option et en seconde année, on distingue la complexité dans le pire des cas et la complexité dans le meilleur des cas (la complexité en moyenne ou le coût amorti ne figurant pas dans le programme).

Cette nuance s'illustre par un programme naïf déterminant si un entier n est premier en testant la divisibilité par tous les entiers entre 2 et $\sqrt{|n|}$, avec une réponse immédiate pour des nombres proches de 0 (discutable).

Dans le meilleur des cas, notre nombre est pair (et $\neq \pm 2$). Alors la réponse est `False` dès le premier test de divisibilité.

Dans le pire des cas, notre nombre est premier, ce qui nécessite $\sqrt{|n|} - 1$ tests de divisibilité (critère retenu).

On dira alors que la complexité du programme [48] est dans le meilleur des cas un test de divisibilité et dans le pire des cas $\sqrt{|n|} - 1$ tests de divisibilité, ce qui correspond conventionnellement à un programme fonctionnant en temps exponentiel, car le cal-

48. ce n'est pas la complexité du problème, qui correspond à la complexité du meilleur programme imaginable, exprimée asymptotiquement en pratique

cul de complexité considère la taille de n, donc le nombre de bits utilisés dans sa représentation [49], qui est $\lceil \log_2(n+1) \rceil$.

Le calcul de la complexité en temps d'un algorithme se fait par une analyse de sa structure, ainsi :

- Une instruction de base, par exemple un calcul arithmétique ou logique, une affectation, etc. aura un coût négligé ou retenu comme une unité (comme dit précédemment).
- Le coût de plusieurs instructions qui se suivent, notamment au sein d'un bloc, est la somme des coûts des instructions. Cela vaut bien entendu si les instructions qui se suivent sont eux-mêmes des blocs.
- Le coût d'un test conditionnel est au plus le maximum des coûts des blocs qui le composent, auquel on ajoute au plus le coût d'évaluation de tous les booléens dans la portée du `if` et des `elif` éventuels.
- Le coût d'une boucle inconditionnelle est la somme des coûts des instructions du bloc pour chaque tour dans la boucle. Dans les nombreux cas où les coûts sont tous identiques, on peut procéder à une multiplication du coût commun par le nombre de tours.
- Le coût d'une boucle conditionnelle est également la somme des coûts des instructions du bloc pour chaque tour dans la boucle. Malheureusement, cette fois-ci, le nombre de tours dans la boucle n'est pas forcément connu. En règle générale, on cherche juste une bonne majoration, ceci étant.

Lorsque des boucles sont imbriquées, il s'agit de faire une somme double, qui se limite dans les meilleurs cas à une double multiplication.

Par exemple, le programme suivant a un coût de $mn+1$ affectations, mn impressions et $2mn$ multiplications :

```
s = 1
for i in range(n):
    for j in range(m):
        print(42*s)
        s *= -1
```

Dans le programme précédent, la rigueur imposait d'ajouter 1 au coût en raison de

49. certains informaticiens considèrent que les nombres sont représentés en « unaire », donc par autant de symboles `0` que leur valeur, ce qui donne la complexité attendue en racine de n

la première affectation, mais il est évident que ceci est négligeable quand m et n sont assez grands. En pratique, la complexité ne se donne presque jamais exactement mais toujours de manière asymptotique, c'est-à-dire en négligeant tout ce qui peut être négligé afin de donner un ordre de grandeur concis.

Pour ce faire, on utilise les notations de Landau : soient f, g des fonctions [50]. On dit que :

- $f = o_{+\infty}(g)$ si la limite en $+\infty$ de $\frac{f}{g}$ est 0.
- $f = \mathcal{O}_{+\infty}(g)$ si $\frac{f}{g}$ est bornée au voisinage de $+\infty$.
- $f = \Omega_{+\infty}(g)$ si $g = \mathcal{O}_{+\infty}(f)$ (et donc cette notation est utilisée plus rarement).
- $f = \Theta_{+\infty}(g)$ si $f = \mathcal{O}_{+\infty}(g)$ et $g = \mathcal{O}_{+\infty}(f)$.
- $f \sim_{+\infty} g$ si la limite en $+\infty$ de $\frac{f}{g}$ est 1, donc si $g - f = o_{+\infty}(g)$.

On s'autorise à écrire $f = \mathcal{O}(g)$ pour aller plus vite. [51]. Ainsi, on dira que la complexité en temps de notre programme est un $\mathcal{O}(mn)$. L'inconvénient de notre notation est que pour nous, $10^9 n = \mathcal{O}(n)$, ce qui n'est pas pertinent au niveau d'un ordinateur. Toutefois, de telles extrémités n'interviennent pas si fréquemment que cela.

Les algorithmes étudiés ici auront généralement une complexité en temps qui est un $\Theta(1)$ (temps constant), un $\Theta(\log n)$ (temps logarithmique [52]), un $\Theta(n)$ (temps linéaire), un $\Theta(n^2)$ (temps quadratique), un $\Theta(n^3)$ (temps cubique) ou un $\Theta(a^n)$ (temps exponentiel, pour un $a > 1$). En deuxième année, les algorithmes de tri seront présentés, et leur complexité en temps est dans les bons cas un $\Theta(n \log n)$.

Signalons pour finir qu'en théorie de la complexité, l'imprécision va encore plus loin : on se contente de dire par exemple qu'un algorithme est en temps polynomial, sans déterminer a priori pour quel $k \in \mathbb{N} \setminus \{0\}$ la complexité en temps est un $\mathcal{O}(n^k)$ sans être un $\mathcal{O}(n^{k-1})$ (on cherche parfois même des k non entiers).

Quelques rapides informations sur la complexité en espace : elle se calcule à peu près comme la complexité en temps, mais il faut tenir compte de la réutilisation ou non de la mémoire au fur et à mesure de l'exécution. En quelque sorte, la complexité en espace est la taille maximale de la mémoire utilisée à tout moment de l'exécution. Les

50. de $\mathbb{N}$ dans $\mathbb{N}$ pour nous

51. On omet en informatique de préciser que la comparaison se fait en l'infini, ce qui n'est pas le cas en mathématiques En fait, il devrait même plutôt y avoir des $\in$ au lieu des $=$.

52. Peu importe la base du logarithme, vu que le rapport entre deux logarithmes se retrouvera dans la constante multiplicative.

deux programmes suivants, tous deux de complexité linéaire en temps [53] et retournant la même chose, auront ainsi des complexités en espace respectives de $\mathcal{O}(\log n)$ et $\mathcal{O}(n \log n)$ bits, en raison de la taille nécessaire pour écrire les entiers en binaire :

```
def somme1(n):
    s = 0
    for i in range(n):
        s += i
    return s

def somme2(n):
    l = list(range(n)) # !?
    s = 0
    for x in l:
        s += x
    return s
```

2.4 Algorithmes de base

Commençons cette section par une réflexion sur la conception d'algorithmes et de programmes. Imaginons un(e) étudiant(e) face à un exercice lors d'une séance de TP, comme par exemple écrire une fonction calculant la somme des chiffres d'un entier écrit en base 10.

Jusqu'à ce que l'on atteigne un certain niveau en programmation, la première étape est de prendre des valeurs arbitraires et de déterminer le résultat de la fonction que l'on cherche, en prenant conscience de la méthode employée. Ici, pour le nombre 267, on isole les chiffres et on additionne : $2 + 6 + 7$ (ou dans l'autre sens).

Bien entendu, isoler les chiffres est une opération immédiate pour un humain, mais il faut expliquer à l'ordinateur comment faire. Ici, en se souvenant du premier chapitre, on sait que 7 est simplement le reste de la division euclidienne de 267 par 10, idem pour 6 et 26 (en tant que `267 // 10`) par 10, et 2 est le dernier reste.

La deuxième étape est de jouer le rôle de la fonction et de donner à un interlocuteur

53. en nombre d'instructions élémentaires, vu que des additions d'entiers interviennent et qu'on ne souhaite pas prendre pour unité une opération sur un bit

le rôle de l'ordinateur : « Tu prends ton nombre, s'il est non nul tu calcules son reste dans la division par 10 et tu le mets de côté, puis tu prends le quotient dans la division par 10 et tu répètes. À la fin, tu additionnes ce que tu as mis de côté. »[54]

La troisième étape est alors de penser l'algorithme en pseudo-code, qui est une traduction du français vers un langage plus adapté aux calculs dans la mesure où les mots de liaison ne vont pas polluer les instructions : « Soit n le nombre, soit $s = 0$, tant que $n \neq 0$ (soit $s = s + (n \bmod 10)$, soit $n = [n/10]$), retourner s. »

La dernière étape est la transcription de l'algorithme en un programme :

```
def somme_chiffres(n):
    n = abs(n) # je n'aime pas les négatifs !
    s = 0
    while n > 0:
        s = s + (n % 10)
        n = n // 10
    return s
```

Une autre façon de présenter les choses est sous la forme d'un jeu de rôle faisant intervenir un aveugle et un idiot[55], où l'aveugle représente le programmeur, ne connaissant pas les données que le programme doit traiter puisque celui-ci est écrit par avance et pour n'importe quelle entrée, et l'idiot représente le programme, qui n'est pas censé avoir d'autre intelligence propre que celle de celui qui le dirige.

2.4.1 Algorithmes sur les listes

La plupart des algorithmes de cette section seront écrits en pseudo-code. Leur écriture en Python se fera en TP.

Déterminer la présence d'une valeur dans une liste, sans disposer de précision, revient à la parcourir en faisant des comparaisons à chaque étape, et il n'y a pas de raison que l'ordre de parcours soit autre que de gauche à droite. On écrit donc :

```
Fonction recherche_liste(element, liste):
```

54. Pour éviter de saturer la mémoire de l'interlocuteur, mieux vaut additionner au fur et à mesure, ce qui nécessite de stocker les résultats intermédiaires dans une variable.

55. je prends la responsabilité de cette invention...

```
    Pour i de 0 à taille(liste)-1: {
        Si liste[i] == element: { Retourner Vrai }
    }
    Retourner Faux
```

Dans ce premier algorithme, on considère que les indices dans la liste commencent à zéro et qu'une instruction `Retourner` interrompt l'exécution de la fonction.

Une deuxième version compte le nombre d'occurrences de l'élément dans la liste, suivie d'un algorithme déterminant le premier indice où l'élément apparaît, avec un message d'échec éventuel (en Python, la valeur de retour sera plutôt `None`) :

```
Fonction occurrences_liste(element, liste):
    compteur = 0
    Pour i de 0 à taille(liste)-1: {
        Si liste[i] == element: { compteur = compteur + 1 }
    }
    Retourner compteur

Fonction position_liste(element, liste):
    Pour i de 0 à taille(liste)-1: {
        Si liste[i] == element: { Retourner i }
    }
    Retourner "Introuvable"
```

Pour finir, on peut déterminer la liste des indices où l'élément apparaît, en notant bien qu'il est recommandé de ne pas se servir d'une telle fonction pour compter le nombre d'occurrences en raison du coût de construction de la liste des indices.

Remarque : La syntaxe `positions[] = i` de la fonction suivante est récupérée des langages de type C, Java et PHP, elle n'existe pas en Python. Il s'agit d'un ajout à la fin de la liste, qu'on écrirait en Python à l'aide de la fonction `append`.

```
Fonction positions_liste(element, liste):
    positions = []
    Pour i de 0 à taille(liste)-1: {
        Si liste[i] == element: { positions[] = i }
    }
```

```
    Retourner positions
```

La recherche du maximum d'une liste de nombres se fait là encore par un parcours, en mémorisant le maximum partiel à tout moment.

Si un élément de la liste n'est pas numérique, il est normal d'obtenir un message d'erreur, ce qui ne nous concerne pas au moment d'écrire l'algorithme. Le pseudo-code suivant retourne le premier indice où le maximum apparaît et le maximum lui-même :

```
Fonction cherche_max(liste):
    indice = 0
    max = liste[0]
    Pour i de 1 à taille(liste)-1: {
        Si liste[i] > max: {
            max = liste[i]
            indice = i
        }
    }
    Retourner (indice, max)
```

Remarque : Mémoriser le maximum évite de devoir accéder à chaque étape à `liste[indice]`.

Le calcul de la moyenne d'une liste de nombres n'est pas plus compliqué, et on fait appel à cette fonction pour calculer la variance. [56]

```
Fonction moyenne(liste):
    somme = 0
    Pour i de 0 à taille(liste)-1: { somme = somme + liste[i] }
    Retourner (somme / taille(liste))

Fonction variance(liste):
    m = moyenne(liste)
    somme = 0
    Pour i de 0 à taille(liste)-1: { somme = somme + (liste[i]-m)**2 }
```

56. **Attention : cet appel ne doit surtout pas se faire à l'intérieur de la boucle « Pour », sinon le nombre de parcours n'est plus de deux !**

```
    Retourner (somme / taille(liste))
```

2.4.2 Dichotomie

La dichotomie[57] (du grec : couper en deux) est un principe intuitif consistant à séparer un ensemble en deux parties égales dont on sait qu'une est ignorée.

L'illustration la plus classique est le problème de deviner un nombre entre 1 et $2^{n-1}-1$ en au plus n essais, avec l'information à chaque essai si le nombre à deviner est supérieur, inférieur ou égal au nombre testé.

Il s'agit de commencer par tester 2^{n-2}, et à chaque étape d'ajouter ou de retrancher (suivant le résultat de la comparaison) 2^{n-3}, puis 2^{n-4}, etc. jusqu'à trouver le résultat.

Déterminer la présence d'une valeur dans une liste triée suit exactement le même principe (on compare avec la valeur les éléments dont les indices sont calculés selon la méthode présentée au paragraphe précédent).

L'algorithme écrit ci-dessous va en pratique restreindre l'intervalle d'indices où l'élément peut être, en le divisant toujours en deux parties dont la taille est égale (à un près), de sorte qu'on n'ait pas besoin d'avoir une taille totale égale à une puissance de deux moins un.

La fonction retourne un indice où la valeur apparaît, avec éventuellement un message d'échec.

```
Fonction recherche_dicho(element, liste):
    n = taille(liste)
    Si n == 0: { Retourner "Introuvable" }
    ind_min, ind_max = 0, n-1
    Tant que ind_max >= ind_min: # vu comme on écrit la suite,
# il est possible que ind vaille ind_min et donc qu'à l'étape suivante
# ind_max vaille ind_min-1, par exemple si on cherche -1 dans [0,1]
    {
        ind = (ind_min + ind_max)//2
        Si liste[ind] == element: { Retourner ind }
        Sinon si (liste[ind] < element) == (liste[0] < element):
```

57. C'est pas faux !

```
        { ind_min = ind+1 }
        # on cherche entre ind+1 et ind_max si :
        # - la liste est croissante et l'élément est supérieur
        # - ou la liste est décroissante et l'élément est inférieur
        Sinon: { ind_max = ind-1 }
        # on cherche entre ind_min et ind-1 sinon
    }
    Retourner "Introuvable"
```

La complexité s'analyse ainsi : on se doute qu'il s'agit d'une fonction croissante en la taille de la liste, et en posant c_n le nombre d'instructions élémentaires pour une recherche dichotomique dans une liste de taille n, on a $c_{2n+1} \leq c_n + \mathcal{O}(1)$ et $c_{2n} \leq \max(c_n, c_{n-1}) + \mathcal{O}(1)$ car au pire on cherche dans une sous-liste après avoir fait les comparaisons et affectations du tour de boucle en cours, ce qui permet d'écrire pour une valeur $n = 2^k$ l'inégalité $c_{2^k} \leq c_{2^{k-1}} + \mathcal{O}(1) \leq \cdots \leq k\mathcal{O}(1) = \mathcal{O}(k)$, d'où $c_n = \mathcal{O}(\log_2(n))$.

Pour obtenir la liste de tous les indices où la valeur apparaît, qui est une liste d'entiers consécutifs puisqu'on fait une recherche dans une liste triée, il suffit de regarder à gauche et à droite d'un des indices favorables jusqu'où on a encore la valeur… le faire par une sorte de dichotomie permet de garder un coût raisonnable.

Exercice

Écrire un algorithme effectuant une double dichotomie pour trouver entre quel indice et quel autre indice une valeur apparaît dans une liste.

On utilise également la dichotomie sur des réels, notamment en appliquant le théorème des valeurs intermédiaires pour chercher un antécédent d'un réel par une fonction continue si on dispose d'un encadrement. En théorie, cela s'appliquerait aussi pour une fonction monotone, sans garantie d'existence d'antécédent, mais les problèmes de précision que l'on a évoqués au chapitre précédent peuvent engendrer des résultats erronés. [58]

Puisque nous travaillons sur des réels, la recherche d'antécédent se fera à un ε donné

58. Cela dit, pour une fonction continue, des erreurs d'approximation existent aussi, mais c'est moins grave que d'annoncer qu'il n'existe pas d'antécédent alors qu'il y en a un, ou vice-versa.

près, c'est-à-dire qu'au lieu d'obtenir un réel x tel que $f(x) = c$, on s'arrête dès que $b - a \leq \varepsilon$ (ceci pour avoir une complexité logarithmique en le rapport $\frac{b-a}{\varepsilon}$ indépendamment de la fonction).

Exercice

Prouver la complexité ci-avant.

Concernant les fonctions monotones non continues, on ajoute un paramètre h tel que la recherche est considérée comme réussie si l'image du réel testé est éloignée de moins de h de c et échouée si l'intervalle de recherche est devenu trop petit. Nécessairement, ε devra être très petit afin de garantir que, quand il y a effectivement un antécédent, il soit rare de voir la recherche échouer parce que $|f(x) - c| > h$ alors que l'intervalle est déjà trop petit. Les fonctions s'écriront donc ainsi :

```
Fonction recherche_antecedent_continue(f, c, eps, a, b):
    Tant que b-a > eps: {
        x = (a + b) / 2
        image = f(x)
        Si (image-c) * (f(a)-c) < 0: { b = x }
        # au moins un antécédent est entre a et (a + b) / 2
        Sinon: { a = x } # entre (a + b) / 2 et b
    }
    Retourner (a + b) / 2

Fonction recherche_antecedent_monotone(f, c, h, eps, a, b):
    Tant que abs(f((a+b)/2)-c) > h et b-a > eps: {
        x = (a + b) / 2
        image = f(x)
        Si (image-c) * (f(a)-c) < 0 : { b = x }
        # l'antécédent putatif est entre a et (a + b) / 2
        Sinon: { a = x } # il est entre (a + b) / 2 et b
    }
    Si abs(f((a+b)/2)-c) <= h: { Retourner (a+b)/2 }
    Sinon: { Retourner "Introuvable" }
```

Exercice

Que retournent exactement ces fonctions ?

2.4.3 Calcul approché d'une intégrale

Ce paragraphe est consacré à deux méthodes d'approximation de la valeur d'une intégrale par des intégrales simples : sur des rectangles et des trapèzes. Nous allons voir comment implémenter en Python ces méthodes avec diverses variantes, ainsi que leur efficacité.

Commençons par la méthode des rectangles. Elle consiste à approcher une fonction continue par une fonction constante par morceaux, que l'on intègre sur les intervalles où elle est constante.

Un humain peut d'ailleurs aisément borner la valeur de l'intégrale en encadrant la fonction par deux fonctions constantes par morceaux.

En revanche, sans propriété sur la fonction (la monotonie aiderait, par exemple, ou éventuellement le caractère lipschitzien), déterminer minimum et maximum locaux n'est pas évident.

La première version de notre programme implémentant la méthode des rectangles consiste à découper notre intervalle en un nombre de segments donné en argument, chacun étant de même taille.

Ensuite, la valeur prise pour la fonction constante par morceaux sera au choix soit la valeur en la borne inférieure, soit la valeur en la borne supérieure, soit la valeur en le milieu du segment.

Ceci est regroupé dans une fonction annexe appelée `valeur_mode`.

```
def valeur_mode(f, deb, fin, m):
    if m == "inf":
        return f(deb)
    if m == "sup":
        return f(fin)
    return f((deb+fin)/2) # m == "mil"
```

```
def integrale_rectangle(f, a, b, n, m):
    assert n > 0 and b > a and m in ["inf", "sup", "mil"], "Erreur d'arguments"
    t = (b-a)/n # taille du segment
    somme = 0
    for i in range(n):
        deb, fin = a + i*t, a + (i+1)*t
        somme += t * valeur_mode(f, deb, fin, m)
    return somme
```

Une variante consiste à choisir des segments de tailles arbitraires, qui sont alors donnés comme la liste de leurs extrémités. On choisit alors souvent sur chaque segment la valeur de la fonction en son milieu.

```
def integrale_rectangles(f, bornes, m):
    assert m in ["inf", "sup", "mil"], "Le mode est inconnu"
    somme = 0
    for i in range (len(bornes)-1):
        deb, fin = bornes[i], bornes[i+1]
        somme += (fin-deb) * valeur_mode(f, deb, fin, m)
    return somme
```

La méthode des trapèzes, quant à elle, consiste à approcher une fonction continue par une fonction affine par morceaux. Elle est en fait équivalente à une méthode des rectangles dans laquelle sur chaque segment la constante retenue serait la moyenne des valeurs aux bornes du segment.

Comme pour la méthode des rectangles, nous commençons par le cas où les segments sont tous de même taille.

```
def integrale_trapeze(f, a, b, n):
    t = (b-a)/n # taille du segment
    somme = 0
    for i in range(n):
        deb, fin = a + i*t, a + (i+1)*t
        somme += t * (f(deb)+f(fin))/2
    return somme
```

```
def integrale_trapezes(f, bornes):
    somme = 0
    for i in range (len(bornes)-1):
        deb, fin = bornes[i], bornes[i+1]
        somme += (fin-deb) * (f(deb)+f(fin))/2
    return somme
```

L'erreur, soit la différence avec la valeur exacte de l'intégrale, est de l'ordre de $\frac{1}{n}$ dans le cas de la méthode des rectangles, sauf si on choisit le milieu de l'intervalle, auquel cas elle est de l'ordre de $\frac{1}{n^2}$, tout comme la méthode des trapèzes. N'oublions pas qu'à cela s'ajoutent de toute façon les erreurs dues à la représentation des réels.

2.4.4 Recherche d'un motif dans une chaîne de caractères

Le dernier algorithme présenté ici, la recherche d'un *motif* dans une chaîne de caractères, existe bien entendu en Python (voir la fin du TP 2 à ce sujet). Nous allons présenter ici une version naïve de l'algorithme.

Le problème est le suivant : on dispose d'une chaîne de caractères `s`, en principe assez longue, et on veut déterminer si une autre chaîne de caractères `m`, appelée le motif, en est une sous-chaîne, c'est-à-dire qu'en extrayant une suite ininterrompue de caractères de `s`, on obtient exactement `m`.

Intuitivement [59], il suffit d'extraire toutes les sous-chaînes de taille `len(m)` de `s` possibles, donc les `s[i:i+len(m)]` pour `i` entre `0` et `len(s)-len(m)` inclus.

En pratique, l'extraction et la vérification nécessitant deux parcours de la sous-chaîne extraite, la vérification se fera à la volée pour diviser par deux la complexité en temps dans le pire des cas, et surtout pour avoir une complexité en espace minimale : il suffit de mémoriser les indices des positions où la recherche en est.

Le programme ci-après détermine le premier indice de début d'une sous-chaîne de `s` qui soit exactement `m`, en retournant `-1` sinon. [60] On peut bien entendu adapter ce code pour obtenir des programmes résolvant des variantes du problème de recherche de motif présenté. [61]

59. et avec la syntaxe de Python
60. Il est donc équivalent dans à la méthode `find` sans argument supplémentaire.
61. J'espère que vous comprenez que ça sent l'énoncé de DS.

```
def recherche_motif(s, m):
    for i in range(len(s)-len(m)+1):
        ind = 0
        while ind < len(m) and s[i+ind] == m[ind]:
            ind += 1
        if ind == len(m):
            return i
    return -1
```

En ce qui concerne la complexité en temps, le pire cas serait d'avoir un motif de type `"aa...ab"` à rechercher dans une chaîne ne contenant que des `a`. Il faudrait parcourir le motif entier pour se rendre compte que la recherche échoue, et ce à toutes les positions, d'où une complexité en $\mathcal{O}(|s|.|m|)$.

L'algorithme de Knuth-Morris-Pratt apporte une amélioration consistant à faire un pré-traitement du motif, afin que le nombre de parcours de `s` ne dépende pas de `m`, le principe étant qu'en cas d'échec on recule à une position pertinente du motif (pas nécessairement au début) et donc sans se forcer à reprendre la recherche du début. La complexité est alors baissée et devient un $\mathcal{O}(|s| + |m|)$. En outre, en deuxième année de l'option informatique, nous verrons la recherche de motifs comme une application des automates, et on présentera un modèle qui « devine » à quel endroit le motif se trouvera pour ne faire là aussi qu'un parcours.

On remarquera que, telle qu'on a écrit la recherche de motifs, `s` et `m` peuvent être n'importe quel type de séquence, et non nécessairement le même type.

2.5 Compléments

Cette partie du polycopié présente des notions qui ne sont pas traitées dans le cours, mais éventuellement utiles pour les TP (auquel cas elles sont rappelées), et surtout utiles dans la perspective d'un approfondissement important de la programmation dans la suite de la scolarité ou en-dehors des études.

2.5.1 Variables globales et variables locales, etc.

Dans un programme en Python, toutes les variables qui sont créées en-dehors de la portée d'une fonction sont appelées *variables globales*. Ces variables sont considérées comme existantes et donc utilisables à partir du moment où l'instruction où elles sont déclarées est exécutée. Comme une fonction peut être vue comme une variable, cela permet de revenir sur le fait qu'il faille exécuter le contenu de l'éditeur pour pouvoir appeler dans la console les fonctions qui y sont définies. [62]

Ainsi, dans le code suivant, la fonction `f` est d'abord définie, puis la variable `a`, de sorte que lors de l'appel ultérieur à `f(4)`, `n+a` peut effectivement être calculé et vaut `7`, qui est imprimé.

```
def f(n):
    print(n+a)

a = 3
f(4)
```

On peut ainsi écrire une fonction qui en fait intervenir une autre, définie plus tard [63], à condition que la fonction ne soit pas appelée avant que l'autre ne soit définie.

À l'intérieur du code définissant une fonction, toutes les variables qui sont définies, de même que les arguments d'une fonction, sont appelées *variables locales* : elles n'ont pas d'existence en-dehors de la fonction, sauf si elles partagent le nom d'une variable globale, auquel cas cette dernière est localement oubliée, c'est-à-dire qu'on ne peut pas accéder à sa valeur. Ainsi, le code suivant imprimera la valeur `42`, puis `0`, ce qui témoigne du fait que la valeur de `a` n'est pas affectée en-dehors du code de la fonction `f` :

62. Cela vaut aussi et surtout pour la prise en compte des corrections d'erreurs.
63. ce qui ne serait par exemple pas autorisé en Caml

```
def f(a):
    a = 42
    print(a)

a = 0
f(4)
print(a)
```

Les TP réalisés jusqu'ici ont sans doute fait remarquer que certaines fonctions avaient un effet de bord modifiant leur argument, lorsqu'il s'agissait de listes par exemple. En pratique, lorsqu'une liste est un argument d'une fonction, la variable locale correspondante est un alias pour cette liste, et toute modification de l'un entraîne une modification de l'autre, ce qui est exactement le même comportement que lorsqu'on écrit `b = a`.

Ces effets de bord peuvent s'obtenir pour d'autres variables globales à condition de les déclarer comme telles dans la fonction. Pour autant, mieux vaut éviter d'avoir recours à cela.

Ainsi donc, le code suivant imprimera deux fois la valeur `42`, ce qui signifie que `a` a été modifiée.

```
def f():
    global a
    a = 42
    print(a)

a = 0
f()
print(a)
```

En pratique, si `a` avait été un argument de la fonction, Python aurait provoqué une erreur car un argument et une variable globale, censés coexister, ne peuvent pas avoir le même nom.

La question qui demeure est celle de variables ni locales ni globales pour une fonction qui est elle-même définie dans une autre (voir la sous-section suivante).

Dans ce cas, le mot-clé est `nonlocal`, et le code suivant imprimera successivement `42` puis `0`, car la variable `a`, locale à `f` mais pas à `g`, est modifiée par cette dernière, sans conséquence cependant pour la variable globale également nommée `a`.

```
def f():
    a = 73
    def g():
        nonlocal a
        a = 42
    g()
    print(a)

a = 0
f()
print(a)
```

On retiendra tout de même que ces pratiques sont esthétiquement douteuses.

Au passage, un tel souci permet d'expliquer pourquoi en exécutant les deux codes, pourtant semblables, Python déclenche une erreur dans le deuxième cas seulement :

```
def f1(a):
    print(b)
b = "hello world"
f1(b)

def f2(a):
    b = b + "coucou"
b = "bonjour"
f2(b)
```

Dans le même ordre d'idée, lorsque Python définit une fonction `f` qui dépend d'une autre, disons `g`, puisque chaque appel à `f` exécute son code, on pourrait se demander si c'est au moment de la définition de `f` que le code de `g` est copié ou si Python appelle `g` au moment de l'appel de `f`.

La réponse peut être anticipée de ce qui précède : puisque `g` peut ne pas encore avoir été définie sans que des problèmes se posent, la deuxième réponse est retenue.

En fait, Python fait ce que l'on appelle des *liaisons dynamiques*, de sorte que redéfinir `g` après coup modifie le comportement de `f` sans qu'on n'ait besoin de relancer la définition de celle-ci.

Ainsi, le code suivant imprimera successivement `1` puis `2`, signifiant que le code de la fonction `f` aura effectivement changé par la redéfinition de `g` :

```
def g():
    print(1)
def f():
    g()
f()
def g():
    print(2)
f()
```

Sans entrer dans le détail des mécanismes qui régissent Python, on peut donner un principe qui donnera l'idée de son comportement : une variable d'un type *mutable* sera susceptible de subir un effet de bord dans une fonction, une variable *immutable*[64] non. Jusque là, les listes sont les seuls objets mutables que nous avons vus, ce qui rejoint la discussion précédente.

En Python, lors d'un appel d'une fonction, on passe les variables en argument *par référence*, c'est-à-dire que ce qui est manipulé est une sorte de pointeur vers une adresse mémoire où se trouve la valeur en question, qui sera éventuellement modifiée, ce qui a sur les variables pointant sur l'adresse mémoire les conséquences qu'on a vues.

Une autre possibilité en programmation (dans d'autres langages, en pratique) est le *passage par valeur*, ce qui signifie qu'on procède à une évaluation et c'est la valeur qui est transmise à la fonction, en omettant totalement la provenance de ladite valeur[65].

64. Oui, ça se dit !

65. Pour les objets non mutables, le passage se fait aussi par référence, mais par définition aucune modification n'est possible.

2.5.2 Fonctions locales, fonctions anonymes

De même qu'on utilise des variables locales dans le code de fonctions, puisqu'on a dit que les fonctions étaient des variables, il est possible de définir ce qu'on appelle des *fonctions locales*, c'est-à-dire des fonctions qui n'existent que dans le code d'une fonction. Sans entrer dans les considérations sur l'efficacité en termes d'utilisation de mémoire (ni du fait qu'à chaque appel les fonctions locales soient redéfinies) dans certains langages, voici un exemple :

```
def coeff_bin(n, m):
    def fact(x):
        s = 1
        for i in range(1, x+1):
            s *= x
        return s
    return fact(n)//(fact(m)*fact(n-m))
```

Bien entendu, les variables locales à la fonction locale n'ont pas d'existence dans la fonction principale. En fait, les fonctions locales définissent un niveau d'imbrication supérieur par rapport aux fonctions, et rien n'exclut d'aller encore plus profondément en créant une fonction locale à une fonction locale.

Pour les fonctions locales que l'on n'utiliserait qu'une fois et dont le code serait court, il n'est pas nécessaire de faire une définition sur plusieurs lignes, et on peut avoir recours à des *fonctions anonymes* à l'aide du constructeur `lambda` provenant du lambda-calcul.

La syntaxe est la suivante : `lambda x_1, x_2, ..., x_n : valeur_de_retour`, et la sémantique est équivalente à :

```
def f(x_1, x_2, ..., x_n):
    return valeur_de_retour # en fonction des arguments
f
```

Le `f` final est l'objet de type fonction obtenu par l'instruction avec `lambda`. Ainsi, pour appeler directement une fonction anonyme, par exemple une fonction qui élève au carré, de même qu'on écrirait `carre(10)`, on écrira

```
(lambda x : x * x)(10).
```

Par conséquent, pour obtenir la somme des carrés des entiers de `1` à `n`, on donne les quatre versions équivalentes (la quatrième étant réservée aux casse-cous) :

```
def somme_carres1(n):
    s = 0
    for i in range(n+1):
        s += i * i
    return s

def somme_carres2(n):
    def carre(x):
        return x * x
    return sum(map(carre, range(n+1)))

def somme_carres3(n):
    return sum(map(lambda x: x * x, range(n+1)))

somme_carres4 = lambda n: sum(map(lambda x: x * x, range(n+1)))
```

L'exemple précédent permet d'introduire la fonction `map`, qui applique son premier argument, nécessairement une fonction, à chacun des éléments de son deuxième argument, nécessairement un itérable mais de type quelconque, formant un objet itérable qui est renvoyé et qu'on convertit généralement en liste (ou en n-uplet, voire en chaîne de caractères à l'aide de la méthode `join`).

2.5.3 Exceptions

Cette sous-section est à lire en parallèle du TP 4, où de nombreux messages d'erreur sont présentés.

Déclencher une *erreur* peut se faire facilement[66]. Outre les messages renvoyés par Python quand on lui demande par exemple de diviser par zéro, ce qu'on appelle *exception* peut être de nature très variée.

La première que l'on utilisera[67] est l'erreur d'*assertion*. Elle permet d'éviter une erreur ultérieure dans un programme... en déclenchant une erreur tout de suite[68], lorsqu'un test que l'on effectue n'est pas vérifié. Ceci permet d'éviter une syntaxe lourde à l'aide d'un test conditionnel, forçant à entrer tout de suite dans un gros bloc.

Les assertions se font à l'aide de la syntaxe ci-après :
`assert verification, message_si_expression_fausse`

L'expression `verification` doit s'évaluer en un booléen[69] et le message (optionnel) doit s'évaluer en une chaîne de caractères... quand une erreur est déclenchée (sinon, vu qu'il est ignoré, Python ne nous embêtera pas).

En fait, les exceptions ont un type, qui est précisé en début de message d'erreur, et la valeur des exceptions est donnée entre guillemets ensuite. Ainsi, quand on demande la racine d'un nombre strictement négatif, la syntaxe est correcte, le type aussi, l'erreur réside dans la valeur : Python renvoie `ValueError: math domain error`. De même, si on veut accéder à un élément inexistant dans une liste (ou dans un itérable en général), c'est une erreur d'indice : `IndexError: list index out of range`.

Déclencher une exception manuellement se fait par le mot-clé `raise`. On écrira donc `raise ValueError("Je voulais 42")` pour déclencher une erreur de valeur « maison ».

Tout cela serait bien inutile si rencontrer une erreur faisait irrémédiablement cesser l'exécution d'un programme. Par chance, toute exception peut être rattrapée.

66. voire involontairement...
67. ... qu'on a déjà utilisée
68. bonjour l'efficacité
69. avec la même remarque que pour les tests conditionnels : `assert 1` est possible, mais moche

La syntaxe est la suivante :

```
try: # on se met dans un contexte où chaque exception est rattrapée
    code_avec_exceptions_possibles
# il s'agit bien entendu d'un bloc, donc indentation et double-point
except (un_type_d_exception, un_autre):
    autre_code_sans_que_les_nouvelles_exceptions_soient_rattrapees
# le code ci-dessous est exécuté si une exception d'un des types est soulevée
except un_autre_type:
    autre_code_idem
except: # toutes les autres exceptions sont rattrapées ici
    encore_un_autre_code_idem
else: # si aucune exception n'est déclenchée, ce code est exécuté
    code_pas_idem
finally: # ce code est exécuté à la fin dans tous les cas
    code_final
# étonnamment, un return dans le finally remplacerait tout autre return
```

En imbriquant les `try`, on peut rattraper une exception soulevée au moment où une exception était rattrapée, mais trouver un programme faisant naturellement ceci semble très artificiel. Ce qui est en revanche envisageable est de rattraper une erreur afin de transmettre un message à l'utilisateur avant de redéclencher l'erreur en écrivant simplement `raise`. La fonction `exc_info` du module `sys`, qui n'a pas d'argument, renvoie un triplet dont le premier élément est le type de l'exception déclenchée et le deuxième élément est sa valeur. Ceci s'illustre par le code suivant :

```
import sys
def kamikaze(a):
    try:
        i = 0
        while True:
            1 / a[i]
            i += 1
    except:
        print("Une erreur a été déclenchée pour i =",i,": son type est",
sys.exc_info()[0],"et le détail est \"",sys.exc_info()[1],"\"")
        raise
```

```
kamikaze([1, 2, 3])
kamikaze([1, 2, 3, 0])
kamikaze([1, 2, 3, "0"])
```

Il est possible de mettre la valeur des exceptions dans une variable, pour s'en servir au moment de rattraper les erreurs :

```
def kamikaze2(a):
    try:
        kamikaze(a)
    except IndexError as valeur:
        if str(valeur) == "list index out of range":
        # car valeur est du type de l'exception
            print("a est une liste de nombres tous non nuls")
        else:
            print("""a n'est pas une liste
mais tous ses éléments sont des nombres non nuls""")
    except:
         print("le parcours de a n'a pas pu se finir")

kamikaze2([1, 2, 3])
kamikaze2([1, 2, 3, 0])
kamikaze2([1, 2, 3, "0"])
```

Annexe : Turing-complétude

Un langage de programmation n'a besoin que de sept instructions pour être qualifié de Turing-complet, c'est-à-dire en mesure d'effectuer les opérations que la machine théorique inventée par Alan Turing peut faire, en omettant la limite de mémoire.

On suppose qu'un interpréteur du langage en question travaille sur une liste, de taille supposée non bornée, d'entiers modulo 256[70], munie d'un pointeur sur une case et en ayant deux flux d'octets à disposition, l'un pour être lu et l'autre sur lequel on écrit (voir aussi la section correspondante).

Les sept instructions sont :

— ajouter 1 au contenu de la case pointée[71] ;
— décaler le pointeur à gauche ou le décaler à droite ;
— remplacer le contenu de la case pointée par l'octet suivant du flux en lecture ;
— écrire le contenu de la case pointée dans le flux ouvert en écriture ;
— ouvrir une boucle « tant que le contenu de la case pointée est non nul » ;
— fermer cette boucle.

Un célèbre langage appelé brainfuck se limite effectivement à ces instructions, et il en existe de nombreuses variantes, dont le non moins célèbre ook.[72].

70. ou une autre valeur dépendante de la table de caractères considérée
71. avec éventuellement une instruction qui retranche 1 pour éviter de faire 255 ajouts
72. En pratique, sous réserve de maîtriser quelques notions d'analyse lexicale, d'analyse syntaxique et de compilation... ou simplement de s'inspirer de programmes le faisant pour d'autres langages, tout le monde peut créer sa propre version du brainfuck... vraiment !

2.6 L'essentiel

Difficile de résumer un chapitre de programmation alors que toutes les notions abordées sont essentielles.

Le mieux est de faire un parallèle avec l'apprentissage d'une nouvelle langue : il faut y aller progressivement, s'efforcer de maîtriser la base au plus vite et surtout pratiquer intensément et régulièrement.

Normalement, tous les éléments nécessaires pour faire n'importe quel programme exigible en classes préparatoires, et même n'importe quel programme de base, sont quelque part dans le chapitre, qui peut être consulté en cas de besoin en cas de blocage devant un exercice, sans bien entendu oublier l'aide qu'internet peut fournir.

Les preuves de programme, qui ont la réputation de rebuter à peu près tout le monde, constituent pourtant aussi un outil puissant pour comprendre les éventuelles erreurs d'algorithmique qu'on a faites. Il ne faut donc pas les négliger. D'ailleurs, elles peuvent aussi discriminer de façon importante des copies aux concours à contenu égal par ailleurs.

Personnellement, je ne suis pas un adepte de l'apprentissage par cœur, préconisant plutôt de savoir tout refaire plutôt que de le restituer sans flexibilité, donc je recommande de refaire les algorithmes de base au programme, et éventuellement d'en écrire plusieurs versions différentes, sans consulter le cours (ni les versions déjà écrites, du coup).

TD 3 : Penser la programmation en Python

Ce TD reprend des exercices de l'excellent livre de Wack et al.

Exercice 1 : Anticiper le résultat de `1 - 1./3 - 1./3 - 1./3` et de `1 - 0.2 - 0.2 - 0.2 - 0.2 - 0.2`[73]. Qu'en serait-il de `1 - 0.25 - 0.25 - 0.25 - 0.25` ?

Exercice 2 : Expliquer les résultats de `round(1.05, 1)`, `round(10.05, 1)`, `round(100.05, 1)` et `round(1000.05, 1)`.

Exercice 3 : Donner deux exemples pour lequel `int(a/b)` et `a // b` donnent des résultats différents pour des raisons différentes.

Exercice 4 : Écrire des expressions booléennes traduisant les conditions suivantes sur des flottants.

- Le point de coordonnées (x, y) appartient au disque de centre c et de rayon r.
- Les points de coordonnées (x, y) et (z, t) sont situés sur une même droite parallèle à l'un des axes du repère.
- Les points de coordonnées (x, y) et (z, t) sont deux sommets opposés d'un carré dont les côtés sont parallèles aux axes du repère.
- Il existe un triangle dont les côtés mesurent respectivement a, b et c.

Exercice 5 : Écrire des expressions booléennes traduisant les conditions suivantes sur des entiers.

- n est divisible par 5.
- m et n sont tels qu'un des deux est multiple de l'autre.
- m et n sont de même signe.
- m, n et p sont de même signe.
- n est le plus petit multiple de 7 supérieur à 10^{100}.
- m, n et p sont distincts deux à deux[74].

Exercice 6 : Écrire une suite d'instructions pour échanger les valeurs de x et de y.

Exercice 7 : Partant de l'état initial vide, donner les six états successifs lors de l'exécution des instructions suivantes, évaluées du haut vers le bas :

---

73. Cela ne marcherait pas avec `1 - 5 * 0.2` !
74. Il existe une solution élégante.

```
x = 3
y = 2
x = 1 + y * x
y = y
y = 1.2 - x
x = y - 1.2
```

TD 4 : Terminaison, correction et complexité

Pour les fonctions successives, prouver la terminaison et calculer la complexité en fonction des arguments, déterminer la spécification précise et prouver la correction.

```
from random import randint
from copy import deepcopy

def FisherYates(l):
    n = len(l)
    ll = deepcopy(l)
    for i in range(n-1, 0, -1):
        j = randint(0, i)
        ll[j], ll[i] = ll[i], ll[j]
    return ll

# ----------------------------------------

def r(s):
    rep = ""
    for i in range(len(s)):
        car = s[i]
        rep = car + rep
    return rep

# ----------------------------------------

def c(s, m):
    rep = 0
    for i in range(len(s)):
        if s[i:i+len(m)] == m:
            rep += 1
    return rep

# ----------------------------------------
```

```
def verif1(l):
    i = 0
    while i < len(l)-1 and l[i] <= l[i+1]:
        i += 1
    return i == len(l)-1

# ----------------------------------------

def verif2(l):
    n = len(l)
    if n <= 2:
        return True
    i = 2
    flag = True
    while i < len(l):
        if l[i] != l[0] + i*(l[1]-l[0]):
            flag = False
            i = len(l)
        i += 1
    return flag

# ----------------------------------------

def verif3(l, ll):
    for x in l:
        if x not in ll:
            return False
    for x in ll:
        if x not in l:
            return False
    return True
```

Chapitre 3

Ingéniérie numérique et simulation

3.1 Compléments de programmation : Python pour les scientifiques

Dans cette section, nous allons voir comment utiliser Python pour résoudre des problèmes concrets en analyse, en se servant de bibliothèques conçues spécialement pour cet usage. Le module `math`, dont les fonctions ont déjà pu dépanner pour l'un ou l'autre problème qui s'est posé jusque là, permet de constater l'utilité de ne pas avoir à réécrire des routines (et aussi les optimiser en termes d'efficacité et de complexité) servant de briques de base pour les programmes auxquels on aura fréquemment recours.

Ainsi, les bibliothèques introduites pour ce chapitre sont `numpy` et `scipy`, l'une permettant d'utiliser une nouvelle structure de données, les tableaux (`array`) multidimensionnels, relativement proches des listes mais prévus pour travailler sur des nombres (notamment en algèbre linéaire), l'autre étant une surcouche de `numpy` contenant des fonctions diverses et variées, dont une partie permettent de court-circuiter les algorithmes sur lesquels nous allons travailler dans la suite du chapitre.

3.1.1 Petit détour par les bibliothèques

Un *module* Python est en quelque sorte un ensemble d'instructions qu'on va pouvoir importer. En un sens, tout fichier écrit depuis le début de l'année au cours des TP peut être vu comme un module et son code récupéré, pour un effet similaire au fait de

l'exécuter (en ne tenant pas compte des réinitialisations de la console). Importer un module sert usuellement à définir des variables et fonctions [1] pour gagner du temps.

Une *bibliothèque* peut à première vue se confondre avec un module, mais elle doit être vue comme une boîte à outils, chargée dans un but spécifique, toujours en gardant l'idée de ne pas avoir à recréer ses outils *ex nihilo*. En pratique, les fonctions usuelles qui ont été vues au chapitre précédent font eux-mêmes partie d'une bibliothèque, appelée bibliothèque standard (*core library*) et qui est importée systématiquement par Python.

Pour information ou rappel, un module (ou une bibliothèque, donc) peut être importé en entier ou un sous-ensemble de ses fonctions peut être importé. La syntaxe de l'importation change, de même que celle des appels aux fonctions (et aux variables en général) importées. Les morceaux de code suivants produisent la même chose mais on ne peut pas échanger une ligne d'un morceau avec une ligne de l'autre sous peine de déclencher une erreur de nom :

```
import mon_module # tout le module est disponible
mon_module.ma_fonction(mon_module.ma_variable)

import mon_module as momo
# momo sera un alias, et mon_module sera inconnu
momo.ma_fonction(momo.ma_variable)

from mon_module import ma_fonction, ma_variable
# le reste du module est inaccessible
ma_fonction(ma_variable)

from mon_module import * # tout est importé
ma_fonction(ma_variable)
```

En utilisant Edupython, on est amené à voir apparaître des rectangles jaunes expliquant le principe d'une fonction dont on commence à écrire le nom (vive l'autocomplétion). C'est de la documentation de code, et ces « *docstrings* » peuvent bien entendu être écrits par n'importe qui : par exemple, au moment d'écrire une fonction, juste en-dessous de la ligne de définition, écrire une chaîne de caractères (généralement sur plusieurs lignes, et donc avec des triples guillemets comme délimiteurs)

1. et des classes en programmation objet

fera comprendre à Python qu'on a précisé un message pour l'utilisateur, et ceci sera imprimé dès lors qu'on demandera des informations à l'aide de la fonction `help`, qui prend en argument un nom de fonction (pas seulement, mais passons), que ce soit de la bibliothèque standard, d'un module ou d'une bibliothèque importée (en préfixant par le nom de celui-ci ou celle-ci, le cas échéant) ou définie par l'utilisateur.

Bien entendu, documenter une fonction fait partie des critères rendant un code agréable à lire, de même que des noms de variable et de fonction pertinents, des commentaires sur des passages délicats et une indentation rigoureuse (de toute façon Python ne laisse pas grandement le choix sur ce dernier point).

Pour aller encore plus loin, il est possible de faire des annotations au niveau des fonctions, qui contribuent encore plus à leur lisibilité. Il est possible d'en faire un réflexe pour les utilisateurs de Caml, et pour tout le monde en fait.[2]

Des bibliothèques et modules peuvent contenir eux-mêmes des sous-modules, qu'il faut parfois importer même si la bibliothèque l'a été (nous verrons des exemples dans la sous-section sur `scipy`).

3.1.2 La bibliothèque `numpy`

Remarque importante : ce cours se lit en parallèle du TP 7 ; il faudra le consulter (voire le connaître) au moment de réaliser ledit TP.

Autre remarque importante : Par souci d'esthétique, le préfixe `numpy.` devant les noms de fonctions est omis, mais il faut se souvenir que la façon d'importer la bibliothèque conditionne sa présence ou non dans les programmes.

La bibliothèque `numpy` fournit la structure, adaptée aux calculs mathématiques, de *tableau* (`array`) de dimension quelconque. Un tableau est censé contenir des nombres (mais comme d'habitude Python ne se plaindra pas tout de suite sinon) et si possible les tailles (au sens des tailles de liste) au sein d'une même dimension sont égales, par exemple un tableau de dimension 2 devrait correspondre à une liste de listes dont toutes les longueurs sont identiques.

2. Les annotations ont été présentées dans le sujet de Centrale en 2017 : `https://www.concours-centrale-supelec.fr/CentraleSupelec/2017/Multi/sujets/2014-028.pdf`. Un autre lien utile à ce sujet : `http://sametmax.com/les-annotations-en-python-3/`.

On crée un tableau par la fonction `array`, l'argument étant une séquence (pouvant donc contenir d'autres séquences, qui elles-mêmes peuvent etc.). Les modifications de tableaux se font avec les fonctions `append`, `insert` et `delete`. Attention, il ne s'agit pas de méthodes, et si `a` est un tableau, `a.append(0)` retournera une erreur d'attribut. Par ailleurs, ces fonctions ne font pas d'effet de bord mais créent un nouveau tableau contenant le résultat de la modification.

On notera que si on ne précise pas un axe en troisième argument, le résultat sera présenté sous forme de tableau de dimension 1 contenant tous les éléments de manière « aplatie » (“*flattened*”), et si ce qu'on ajoute à un tableau en précisant un axe ne respecte pas sa structure, Python provoque une erreur.

L'axe fait référence à la dimension, de sorte que l'axe `0` concerne le niveau de profondeur minimal, soit les éléments du tableau eux-mêmes, et les axes vont de 0 à la dimension moins un[3]. En fait, tout se passe comme si `append` avec précision d'axe fusionnait deux tableaux compatibles.

La fonction `arange` permet de générer des tableaux similaires aux objets de type `range`, à ceci près qu'ici, les arguments sont des flottants. En cherchant bien, on peut donc malheureusement trouver des cas où les problèmes d'arrondis conduiront à une erreur au niveau de la dernière valeur d'un tel tableau. On préfèrera donc la fonction `linspace`, qui demande comme arguments le début, la fin et le nombre d'éléments, permettant de déduire le pas, les éléments formant une suite arithmétique.

Les *matrices* sont à part parmi les tableaux, ayant leur type propre et des fonctions qui ne s'appliquent (correctement) qu'à elles. On les crée à l'aide de la fonction `matrix` dont l'argument est (par exemple) un tableau à deux dimensions ou une chaîne de caractères formatée où espaces ou virgules séparent des lignes et des points-virgules séparent les colonnes.

Une matrice supporte diverses méthodes : `m.getI()` retourne l'inverse de `m`, `m.getT()` retourne sa transposée, `m.diagonal()` retourne ses éléments diagonaux (de type matrice à une ligne) et `m.trace()` retourne sa trace (de type matrice, étonnamment). En outre, on dispose de fonctions de conversions `m.tolist()`, `m.getA()` et `m.getA1()` qui renvoient respectivement la liste de listes, le tableau et le tableau aplati lui correspondant.

3. Ou de l'opposé de la dimension à -1, en fait il s'agit de parcourir le n-uplet `shape(t)` correspondant à la forme du tableau `t`.

Attention : l'accès à un élément d'une matrice se fait par `M[i, j]` et non `M[i][j]`. Cette syntaxe est également utilisable pour les tableaux, mais pas pour les indexables contenant des indexables de la bibliothèque standard. L'avantage de cette façon d'accéder à des éléments est que cela facilite grandement le slicing.

Le produit matriciel se fait à l'aide de la fonction `dot` (qui retourne le produit scalaire lorsque ses deux arguments sont des tableaux de dimension 1). En cas de besoin, Python tente de convertir les arguments en matrices s'il ne s'agit que de tableaux voire de listes (par exemple). La fonction `identity` retourne un tableau correspondant à la matrice identité de taille son argument [4] ; une version ayant plus de fonctionnalités est `eye`, dont les arguments sont le nombre de lignes (seul obligatoire), le nombre de colonnes (autant que de lignes par défaut), le décalage à droite de la diagonale et le type des données [5]. La forme d'une matrice (plus généralement d'un tableau) s'obtient par la fonction `shape`, retournant un n-uplet (si l'argument ne correspond pas parfairement à une matrice, le n-uplet ne sera pas complet).

Matrices et tableaux peuvent être triés à l'aide de la fonction `sort`, ayant quelques paramètres optionnels, dont le plus important est l'axe pour une matrice, déterminant suivant quel axe les données sont triées (en choisissant `None`, le tableau est aplati, par exemple).

Pour les deux structures aussi, le test d'égalité se fait par la fonction `array_equal`, puisque l'opérateur `==` retourne un tableau de la forme commune de ses deux membres donnant le résultat des tests d'égalité composante par composante, et le booléen `False` si cette comparaison composante par composante ne peut pas s'effectuer.

Au passage, la fonction `iterable` retourne `1` si son argument est itérable et `0` sinon. L'intérêt est cependant plus ou moins limité (on peut imaginer une vérification préalable avant de faire un parcours afin d'éviter un message d'erreur).

Terminons par les *polynômes*, qui s'obtiennent eux aussi par une fonction spéciale : `poly1d`. Les polynômes bénéficient comme les tableaux d'une impression travaillée, et on peut choisir le nom de la variable imprimée (n'importe quelle chaîne de caractères,

4. On peut donner un deuxième argument, choisissant le type des éléments du tableau, `float` par défaut, `int` ou `bool` sinon.

5. En pratique, on peut préciser les arguments optionnels dans le désordre, en précisant le nom de l'argument (mais tous les arguments nommés doivent être les plus à droite), voir par exemple `eye(3, dtype=int, k=1)`.

j'ai bien dit n'importe laquelle) en précisant un argument `variable=...` dans la fonction `poly1d`, dont le premier argument est la liste des coefficients (numériques) en commençant par le plus haut degré OU le premier argument est la liste des racines (écrites autant de fois que leur multiplicité) et un deuxième argument est alors mis à `True`.

On évalue un polynôme comme s'il s'agissait d'une fonction, l'argument pouvant être un polynôme, ce qui est équivalent à l'utilisation de la fonction `polyval`.

On accède aux coefficients (resp. racines) de `p` en écrivant `p.c` (resp. `p.r`), en prenant garde aux problèmes d'arrondis (ceci sera répété assez souvent dans tout le chapitre, de toute manière). Une autre façon de procéder est d'utiliser la fonction `roots` ou la fonction `poly` sur le polynôme, respectivement.

Sommes, produits et puissances de polynômes [6] se font naturellement avec les mêmes opérateurs que pour les nombres ou avec des fonctions spécifiques (`polyadd`, `polysub`, `polymul`), et on dispose même de la division euclidienne, avec l'opérateur `/` ou la fonction `polydiv`, qui retourne le couple de polynômes (quotient,reste).

Enfin, les fonctions `polyder` et `polyint` permettent respectivement de dériver et d'intégrer (éventuellement plusieurs fois) un polynôme, en précisant pour la deuxième fonction des constantes d'intégration (soit aucune, soit une, soit au moins autant que le nombre de fois que l'on intègre, qui est le deuxième argument de la fonction).

3.1.3 La bibliothèque `scipy`

Remarque préliminaire : la bibliothèque `scipy` contient toutes les fonctions de la bibliothèque `numpy`, ce qui signifie qu'il ne sert à rien d'importer cette dernière en même temps. En revanche, le préfixage est évidemment à modifier [7].

La bibliothèque `scipy` est essentiellement une collection de fonctions utiles en analyse numérique, et pour servir d'appui aux scientifiques en général (c'est un peu l'idée suggérée par le nom de la bibliothèque...).

Sans entrer dans le détail ni cataloguer les fonctions principales comme on l'a fait

6. Attention, il existe une fonction `square` qui s'applique à des polynômes, mais elle élève les coefficients au carré individuellement.

7. `scipy.array()`, etc.

pour `numpy`, signalons l'existence de fonctions fournissant une alternative intéressante pour les situations étudiées dans le reste du chapitre :

- Problèmes matriciels : le sous-module `linalg` de `scipy` (de `numpy` en fait) fournit de nombreuses fonctions, dont `solve` (donc `scipy.linalg.solve`, le sous-module n'ayant pas à être chargé par une instruction particulière), telle que `solve(a, b)` retourne le tableau `x` (de mêmes dimensions que `b`) tel que le produit de `a` par `x` donne `b`, si `a` est inversible.
- Résolution numérique d'équations sur les réels : le sous-module `optimize` de `scipy` (pas de `numpy`, à charger par l'instruction `import scipy.optimize`, qui est obligatoire, que `scipy` ait été chargé ou non) est particulièrement adapté, avec notamment une fonction `minimize` cherchant le minimum d'une fonction (il faut aussi un deuxième argument initialisant la recherche) et une fonction `newton` appliquant la méthode éponyme (là aussi, un deuxième argument est nécessaire et déterminera parmi les zéros d'une fonction lequel sera trouvé). On notera que les méthodes employées pour ces fonctions sont également implémentées dans les calculatrices scientifiques. [8]
- Approximation de solutions d'équations différentielles : c'est dans le sous-module `integrate` de `scipy` (également absent de `numpy` et à charger à part) qu'on trouve la fonction `odeint`, dont le fonctionnement n'est pas aussi compliqué qu'il peut en avoir l'air : `scipy.integrate.odeint(f, y0, tab)` retourne la liste des valeurs de `y(t)`, où `y` est solution de `y'(t) = f(y(t), t)` (donc `f` prend deux arguments), pour `t` parcourant `tab` (un tableau non vide), sachant que pour le premier élément de `tab`, la valeur est fixée à `y0`.

Bien entendu, les fonctions de la bibliothèque ont été optimisées et seront plus efficaces (disons au moins autant) que celles qu'on écrira à la main. Ceci se prouve aisément : si on pouvait faire strictement mieux à la main en peu de lignes pour le cas général d'un problème, alors à quoi bon inclure une version moins bonne dans un module ? [9]

3.2 Résolution de systèmes linéaires (Gauss)

Remarque : La théorie mathématique de base autour du pivot de Gauss est supposée connue. En cas de besoin, une version légère (programme du BTS Services Informa-

8. Tant d'élèves de Terminale ont demandé pourquoi leur calculatrice demandait à l'utilisateur de donner en plus de l'intervalle de recherche une abscisse particulière...

9. Plus précisément, chercher à faire strictement mieux en se creusant la tête est un peu le principe de la recherche.

tiques aux Organisations) se trouve actuellement à l'adresse `http://jdreichert.fr/Enseignement/Archives/SIO/matrices.pdf` (section 2.3).

Il convient de distinguer l'algorithme du *pivot de Gauss* tel que présenté dans la théorie et sa mise en œuvre (quel que soit le langage utilisé) sous la forme d'un programme. En effet, il est désormais acquis que le travail sur des flottants n'offre pas la précision absolue que le calcul formel pur peut attendre, et l'exactitude même des résultats peut être mise en doute dans certains cas (heureusement minoritaires, et en pratique il faut souvent se forcer pour tomber sur des contre-exemples).

Cette mise en garde vaut bien entendu pour tout le reste du chapitre (à plus forte raison !), et peut motiver le programmeur soucieux à choisir une représentation des nombres (qui ne se limiteront presque jamais aux entiers) plus adéquate : si seuls des rationnels figurent dans la matrice d'un système linéaire, on représentera effectivement ceux-ci comme des couples d'entiers (et tant qu'à faire comme le couple (numérateur,dénominateur) canonique), ce qui fait l'objet du TP 8.

L'algorithme du pivot de Gauss proprement dit est relativement simple : on se ramène par des opérations élémentaires sur les lignes [10] à un système triangulaire qu'on résout par une phase de remontée comme on le ferait en lycée.

Il est également possible de se ramener directement à un système diagonal, voire à la matrice identité (certaines variantes sont mieux adaptés à un problème particulier que d'autres, et c'est cette dernière qu'on retiendra).

Ceci suppose que la matrice du système soit inversible (on dit que le système est de Cramer), et dans le cas contraire l'algorithme peut détecter et signaler un problème ou simplement provoquer une erreur (on préfère éviter de recevoir une réponse ne laissant pas remarquer que le système n'est pas de Cramer). Un raffinement ultime serait de paramétrer l'ensemble des solutions pour des systèmes qui ne sont pas de Cramer, mais ce n'est pas un attendu.

La représentation utilisée pour les données privilégie `numpy` ; de toute façon, ce n'est que lorsqu'on appelle la fonction qu'on peut savoir d'après son argument si la structure utilisée est un tableau. Pour autant, il est envisageable d'utiliser des listes de

10. c'est plus adapté aux systèmes linéaires, car cela revient à multiplier à gauche par des matrices d'opérations élémentaires, et on ne peut pas multiplier un vecteur colonne à droite par une matrice carrée en dimension supérieure à 1

listes de la bibliothèque standard. [11]

Afin de pouvoir résoudre plusieurs problèmes parallèles (notamment résoudre un système linéaire, inverser une matrice quand c'est possible, déterminer son rang [12]...), on peut envisager que la matrice en entrée soit une fusion d'une matrice de système avec des vecteurs compatibles.

En effet, un système linéaire de n équations pour autant d'inconnues sera représenté par une matrice à n lignes et $n+1$ colonnes, la dernière étant le vecteur rassemblant les constantes du second membre. Ceci peut se généraliser à plusieurs membres de droite d'un coup, puisque le pivot de Gauss ne dépend jamais du second membre. En particulier, si on fusionne à droite de la matrice d'un système la matrice identité, on obtient l'inverse de ladite matrice en faisant le pivot.

Passons sans plus attendre à l'algorithme. On procède précisément ainsi :

Pour chacune des lignes, dont on note `i` le numéro (entre `0` et `n-1` où `n` est la taille de la matrice), de la matrice étendue (avec toutes les colonnes supplémentaires accolées), on regarde le coefficient à la colonne `i`, sachant qu'on pose comme invariant de boucle que **les coefficients des colonnes strictement inférieures à la colonne sur laquelle on travaille sont nuls sauf les coefficients diagonaux qui valent** `1`.

Soit le coefficient à la colonne `i` est non nul, ce qui en fait notre pivot, soit il est nul, ce qui fait qu'on échange la ligne `i` avec une ligne d'indice supérieur dont le coefficient à la colonne `i` est non nul, et si aucune ligne ne correspond, c'est que le système n'est pas de Cramer, auquel cas on fait un traitement détaillé plus bas.

Une fois qu'on dispose d'un pivot, on divise la ligne `i` par la valeur de l'élément qui y figure à la colonne `i`, puis on retire à chaque ligne d'indice `j` différent de `i` le produit de la ligne `i` par la valeur de l'élément qui figure à la colonne `i` de la ligne `j` (si c'est `0`, on ne fait rien, mais ce n'est pas un problème).

En particulier, ceci prouve l'hérédité de l'invariant de boucle, car les éléments aux colonnes d'indice strictement inférieur à `i+1` de la ligne `i+1` sont maintenant nuls (les `i` premiers par hypothèse de récurrence, le numéro `i` par l'opération effectuée).

11. Le lecteur intéressé pourra comparer les temps d'exécution suivant la structure de données.
12. à condition d'avoir une bonne gestion des cas où elle n'est pas inversible dans l'algorithme de base

Si le système n'est pas de Cramer, on ignore la ligne et la colonne `i` et on continue (à condition de vouloir déterminer le rang, car beaucoup d'autres problèmes perdent leur sens ou deviennent triviaux, comme par exemple le calcul du déterminant), le rang final étant le nombre de `1` sur la diagonale, les autres éléments diagonaux étant tous des `0` (attention, des éléments non diagonaux peuvent être quelconques s'il y a un `0` sur la diagonale).

En Python, cela donne :

```
def cherche_pivot(M, i):
    if M[i][i] == 0.: # comparaison de flottants, nécessaire mais dangereuse
        j = i+1 # on cherche le premier élément non nul plus bas dans la colonne
        while (j < len(M) and M[j][i] == 0.):
            j += 1
        return j
    else:
        return i

def echange(M, i, j): # Pour un tableau, on ne peut pas faire M[i], M[j] = M[j], M[i]
    for ind in range(len(M[i])):
        M[i][ind], M[j][ind] = M[j][ind], M[i][ind]

def division(M, i, rapport):
    for k in range(len(M[i])):
        M[i][k] = M[i][k] / rapport

def transvection(M, i, j, rapport):
    for k in range(len(M[j])):
        M[j][k] = M[j][k] - rapport*M[i][k]

def pivot(M): # on suppose le système de Cramer et on déclenche une erreur sinon
    n = len(M)
    for i in range(n):
        j = cherche_pivot(M, i)
        if j == n: # tous les M[j][i] sont nuls, la matrice n'est pas inversible
            raise ValueError("Le système n'est pas de Cramer")
### Ici, on peut adapter le code pour déterminer le rang de la matrice
### en sortant de ce tour de boucle (passer au i suivant)
### et en baissant d'un le rang qui vaut n au début
        if j > i:
```

```
            echange(M, i, j)
        rapport = M[i][i] # vu que cet élément sera 1 ensuite, mieux vaut le retenir !
# autre possibilité : pas besoin de rapport et k décroissant
        division(M, i, rapport)
        for j in range(n):
            if j != i:
                rapport = M[j][i] # même remarque
                transvection(M, i, j, rapport)
# Pas besoin de return, la fonction agit par effet de bord.
```

ATTENTION : Si d'aventure ce code est testé pour des tableaux, il faut prendre garde à les initialiser avec des flottants. En effet, comparer les codes suivants :

```
numpy.array([1/2]) # array([0.5]), tout va bien
truc = numpy.array([1])
truc/2 # array([0.5]), tout va bien
truc[0] /= 2
truc # array([0]), tout va mal
```

Ceci s'explique par le fait que `truc/2` crée un nouveau tableau dont les éléments sont des flottants, alors que la modification d'un élément de `truc` conserve sa structure.

On déduit du programme écrit précédemment les solutions d'un système et l'inverse d'une matrice (on suppose `numpy` importé) :

```
def solution_systeme(A, B):
# A a n lignes et n colonnes, B est un tableau de dimension 1 à n éléments
    N = numpy.insert(A, len(A), B, axis=1)
    pivot(N)
    return N[:,-1]

def inverse(M): # M a n lignes et n colonnes
    N = numpy.append(M, numpy.identity(len(M)), axis=1)
    pivot(N)
    return N[:,len(M):]
```

Un problème peut se poser dans l'implémentation de l'algorithme : puisqu'on divise une ligne par le pivot retenu, quand cette valeur est proche de zéro, cela fait apparaître des valeurs très grandes dans la matrice, et ces valeurs se propagent partout.

Puisque le choix du pivot n'est pas censé changer le résultat (en théorie, car en pratique un choix peut conduire à une erreur d'approximation qu'un autre choix aurait permis d'éviter), la méthode du pivot partiel[13] choisit à chaque étape le plus grand élément en valeur absolue de la colonne étudiée parmi les lignes d'indice supérieur ou égal. Si le maximum en question est nul, on conclut comme précédemment que le système n'est pas de Cramer.

On remarquera que cette méthode coûte certes presque systématiquement des échanges (l'étude de complexité ci-après permettra de voir que ce n'est pas si grave), mais qu'elle est particulièrement utile, surtout quand on fait tourner le pivot « à la main », où d'autres critères interviennent, mais toujours dans l'esprit de chercher le meilleur pivot possible.

La complexité de l'algorithme du pivot de Gauss, en opérations élémentaires (opérations arithmétiques ou affectations) est cubique : en remplaçant successivement chaque appel de fonctions par le code de celle-ci, on voit apparaître une imbrication de trois boucles inconditionnelles sur des ensembles dont la taille est de l'ordre de la taille de la matrice.

On retiendra que choisir un pivot, même si ceci se fait en parcourant la matrice entière, ne provoque pas de surcoût asymptotique et contribue parfois à rendre les calculs plus rapides.

Pour autant, les contextes scientifiques dans lesquels nous pouvons être amenés à résoudre un système linéaire comprennent des cas où la taille de la matrice du système peut être dissuasive, et on pourrait s'intéresser au calcul de l'inverse avec une meilleure complexité.

Bien que le produit matriciel naïf soit de complexité cubique, Volker Strassen a établi un algorithme « diviser-pour-régner »[14] en temps $\mathcal{O}(n^{\log_2(7)})$ (soit un exposant d'environ $2,8$), et la course à l'optimisation pour approcher le temps quadratique (borne inférieure absolue, puisqu'il faut tout de même au moins lire les matrices en entrée) a amené à trouver des complexités approchant $\mathcal{O}(n^{2,3})$. Il s'avère que l'inversion peut se faire à l'aide de multiplications pour les matrices symétriques, ce qui se généralise aussi pour les matrices quelconques, mais c'est une autre histoire (c'est-à-dire que c'est un TP de deuxième année).

13. par opposition au pivot total, qui cherche le plus grand pivot sur les lignes **et** sur les colonnes
14. cf. option info

Quelques variantes des fonctions écrites ci-avant :

```
def echange(M, i, j):
    """Opération élémentaire M[i] <-> M[j]"""
    tampon = M[i].copy() # copy() : précaution nécessaire à cause de numpy
    M[i] = M[j]
    M[j] = tampon
    # Rappel : pour un tableau, on ne peut pas faire M[i],M[j] = M[j],M[i]

def divise(M, i, l=None):
    """Opération élémentaire M[i] <- M[i] / l"""
    if l == None:
        l = M[i, i]
    for k in range(M.shape[1]): # len(M[i]) donne 1 pour des matrices !
        M[i, k] /= l
    # Ou simplement ici M[i] /= l

def transvection(M, i, j, l=None):
    """Opération élémentaire M[j] <- M[j] - l * M[i]"""
    if l == None:
        l = M[j, i]
    for k in range(M.shape[1]):
        M[j, k] -= l * M[i, k]

def cherche_pivot(M, i):
    """Première ligne >= i dont l'élément de colonne i est non nul."""
    ind = i
    while (ind < len(M) and M[ind, i] == 0.):
        ind += 1
    assert ind < len(M), "Matrice non inversible"
    return ind

def cherche_pivot_partiel(M, i):
    """Ligne >= i dont l'élément de colonne i est maximal en valeur absolue."""
    ind_max = i
    for ind in range(i, len(M)):
        if abs(M[ind, i]) > abs(M[ind_max, i]):
            ind_max = ind
```

```
    assert M[ind_max, i] != 0., "Matrice non inversible"
    return ind

def pivot(M, partiel=False):
    for i in range(len(M)):
        if partiel:
            echange(M, i, cherche_pivot_partiel(M,i))
        elif M[i,i] == 0:
            echange(M, i, cherche_pivot(M,i))
        divise(M, i)
        for j in range(len(M)):
            if j != i:
                transvection(M, i, j)
```

La structure utilisée ici est nécessairement le tableau ou la matrice, contrairement à l'autre version qui utilisait des listes de listes ou des tableaux.

3.3 Résolution d'équations $f(x) = 0$ (Newton)

Nous avons déjà vu au chapitre précédent la méthode par dichotomie pour résoudre les équations $f(x) = 0$. La *méthode de Newton* a l'avantage de converger bien plus rapidement vers une solution, dans la mesure où la dichotomie réduit de moitié l'intervalle de recherche à chaque itération, alors que la méthode de Newton est dite de convergence quadratique, c'est-à-dire qu'à chaque itération l'écart à la solution est élevé au carré... une fois qu'il est suffisamment petit, bien entendu.

La méthode de Newton présente cependant quelques inconvénients, en plus de ceux qui viennent naturellement quand on utilise des flottants. D'une part, la terminaison n'est pas garantie, c'est-à-dire que dans certains cas le programme échoue à arriver aussi près d'une solution que l'on veut, ce que la dichotomie peut toujours faire quand la précision des flottants ne cause pas de trouble, avec même de possibles déclenchements d'erreurs en raison de divisions par zéro.

D'autre part, on a besoin de connaître la dérivée de la fonction en théorie, c'est pourquoi un premier aménagement consiste à étudier une valeur approchée de la dérivée, à savoir le rapport entre la différence de deux images de la fonction et celle de leurs antécédents, ce qui ramène à la méthode de la sécante.

En pratique, certaines conditions, qui sont usuellement satisfaites, garantissent la terminaison (en ne considérant pas les erreurs d'approximation) : f de classe $\mathcal{C}^1$, convexe sur un intervalle où elle s'annule et dans lequel on prend un point de départ d'image positive par f. Bien entendu, la condition duale est également suffisante (concave et point de départ d'image négative).

Les principes mathématiques généraux de la méthode de Newton sont les suivants :

- Tout comme on approche une intégrale par des sommes d'aires de rectangles ou de trapèzes par les méthodes éponymes, on approche la solution d'une équation $f(x) = 0$ par l'intersection de la tangente à la courbe en le point de départ et l'axe des abscisses, et en répétant le processus à chaque nouveau point trouvé. Cela demande qu'on ne tombe jamais sur une valeur non nulle en un point où la dérivée s'annule, en particulier, car ceci déclencherait une erreur.
- La formule de Taylor-Young pour une fonction f au moins deux fois dérivable : $f(\alpha) = f(u_n) + (\alpha - u_n)f'(u_n) + \frac{(\alpha-u_n)^2}{2}f''(u_n) + o((\alpha - u_n)^2)$.
- En posant (u_n) la suite des valeurs successives calculées par la méthode de

Newton, on a $u_{n+1} = u_n - \frac{f(u_n)}{f'(u_n)}$ (d'où l'intérêt que $f'(u_n)$ ne s'annule jamais), et si on pose α la solution qu'on approche et δ_n l'écart relatif $u_n - \alpha$, la formule précédente se réécrit :

$$f(u_n) = \delta_n f'(u_n) - \frac{\delta_n^2}{2} f''(u_n) + o(\delta_n^2).$$

On a alors $\delta_{n+1} = u_{n+1} - \alpha = u_n - \frac{f(u_n)}{f'(u_n)} - \alpha = \delta_n - \frac{\delta_n f'(u_n) - \frac{\delta_n^2}{2} f''(u_n) + o(\delta_n^2)}{f'(u_n)}$, soit $\delta_{n+1} = \frac{\frac{\delta_n^2}{2} f''(u_n) + o(\delta_n^2)}{f'(u_n)}$. Comme f' et f'' sont bornées (et f' supposée strictement positive) sur un intervalle fermé contenant α et u_n, on déduit que $\delta_{n+1} = \mathcal{O}(\delta_n^2)$.

Si on ne fournit pas la dérivée, en choisissant une valeur pour h (la plus petite est la meilleure), on approche la dérivée de f en a par $\frac{f(a+h)-f(a)}{h}$. Une possibilité encore plus efficace est de prendre $\frac{f(a+h)-f(a-h)}{2h}$ afin d'avoir un développement de Taylor impair, car alors l'écart avec la dérivée est de l'ordre de h^2 (la constante multiplicative fait intervenir la dérivée tierce de f, dont on suppose l'existence).

La *méthode de la sécante*, quant à elle, utilise en guise de dérivée le rapport entre les images et les valeurs sur les deux dernières itérations, c'est-à-dire $\frac{f(u_n)-f(u_{n-1})}{u_n-u_{n-1}}$. La convergence ne sera plus quadratique (elle reste bien meilleure que pour une simple dichotomie), et il est nécessaire que la racine que l'on approche soit simple.

Ainsi, en pratique, on choisira la méthode la mieux adaptée à la fonction dont on dispose (même `scipy` ne semble pas disposer d'une dérivation formelle pour des fonctions autres que polynomiales, tout au plus peut-on approcher le nombre dérivé en un point par `scipy.misc.derivative`), à la vitesse de convergence souhaitée, au besoin d'une réponse correcte, etc.

La plupart du temps, par ailleurs, les programmes effectivement utilisés mélangent Newton et la dichotomie quand la première méthode ne semble pas converger (ou pour détecter de prime abord la présence de zéros et les localiser grossièrement). On pourra à ce titre consulter la documentation de fonctions telles `newton`, `brentq`, etc. dans le sous-module `optimize` de `scipy`. Notamment, `newton` dispose d'un argument optionnel qui est la dérivée, et si l'utilisateur ne fournit rien, la méthode de la sécante est utilisée par défaut.

La mise en œuvre de la méthode de Newton simple à proprement parler est la sui-

vante :

```
def newton(fonction, derivee, depart, epsilon):
    xn = depart
    xnplusun = xn - fonction(xn) / derivee(xn)
    # les divisions par zéro ne sont pas rattrapées
    while abs(xnplusun - xn) > epsilon:
        xn = xnplusun
        xnplusun = xnplusun - fonction(xnplusun) / derivee(xnplusun)
    return xnplusun
```

3.4 Autour des équations différentielles (Euler)

Pareillement à la méthode de Newton, la *méthode d'Euler* passe par une approximation locale de fonctions, ici les solutions d'une équation différentielle, par des fonctions affines, pour lesquelles on peut évidemment résoudre simplement les problèmes en question.

Le fondement mathématique de la méthode d'Euler est le théorème de Cauchy sur l'existence et l'unicité d'une solution à une équation différentielle quand on impose une valeur initiale. C'est cette solution qu'on cherche à approcher, en en calculant en fait non pas une expression formelle mais un certain nombre d'images, par exemple sur un intervalle donné à raison d'un point toutes les h unités d'abscisse. [15]

Or donc, si on calcule $f(x+h)$ en disposant de $f(x)$ (ou d'une valeur approchée), sachant que f vérifie en tout point $f'(x) = G(f(x), x)$, où G fait donc intervenir la fonction f et sa variable x (éventuellement constante suivant l'un ou l'autre), on sait que

$$f(x+h) - f(x) = \int_x^{x+h} f'(x)\mathrm{d}x = \int_x^{x+h} G(f(t), t)\mathrm{d}t$$

et l'approximation se fait ici : on remplace $\int_x^{x+h} G(f(t), t)\mathrm{d}t$ par $hG(f(x), x)$ (méthode d'Euler explicite).

L'implémentation en Python est donc la suivante : on fournit G qui décrit l'équation différentielle, ainsi que l'ensemble des valeurs où le calcul doit être fait et la valeur initiale, d'où le programme ci-dessous.

15. Et là, on a `arange` et `linspace` qui resurgissent !

```
def euler(G, y0, les_x):
    les_y = [y0]
    xplush, y = les_x[0], y0
    # pour information, xplush se lit x + h...
    for i in range(1, len(les_x)):
        x, xplush = xplush, les_x[i]
        y = y + (xplush-x)*G(y, x)
        les_y.append(y)
    return les_y # les_y peut être une liste, pour nous
```

La méthode d'Euler est dite d'ordre 1, c'est-à-dire que l'erreur en valeur absolue entre les valeurs calculées et les vraies valeurs de la solution est dominée et domine le pas à la puissance 1 quand ce pas tend vers 0.

Une autre façon de voir les choses est que cette méthode est exacte quand la solution est une fonction polynomiale de degré au plus 1. D'autres méthodes sont d'ordre supérieur : la méthode de Heun est d'ordre 2 et celle de Runge-Kutta est d'ordre 4.

Une analogie possible est de considérer les approximations d'intégrales par les diverses méthodes : celle des rectangles est d'ordre 1 (et correspond à la méthode d'Euler pour laquelle $G(f(x), x)$ ne dépend pas de $f(x)$), celle des trapèzes est d'ordre 2, et des méthodes d'ordre supérieur consistent à approcher la fonction non pas par des fonctions constantes ou affines mais polynomiales de degré raisonnable, par exemple la méthode de Simpson prend trois points (début, milieu et fin de chaque petit intervalle) et considère la parabole qui passe par ces points. [16]

Contrairement à la méthode de Newton, la méthode d'Euler ne présente pas de souci en termes de terminaison, et la complexité se lit directement : plus le pas est petit, plus il y a de valeurs à calculer (nombre inversement proportionnel en considérant un pas constant), et la complexité est précisément de l'ordre d'une constante multipliée par le nombre de valeurs à calculer.

En fait, c'est justement le compromis entre la complexité et la précision qui va motiver dans le choix du pas, ainsi que dans le choix de la méthode (bien qu'on recommande souvent de sacrifier la simplicité pour l'humain, car après tout ce n'est pas lui qui

16. Cette méthode est d'ordre 4 et donc elle donne la valeur exacte pour les intégrales de fonctions polynomiales d'ordre 3 également. Elle est habituellement utilisée par les calculatrices. Elle a fait l'objet du concours blanc de 2018.

fait les calculs).

Pour des équations d'ordre supérieur, disons n, le programme ci-avant est réutilisable, en considérant que `y_0`, comme les autres futurs éléments de `les_y`, doit contenir n valeurs qui sont l'image par les dérivées 0-ièmes (donc la fonction elle-même) à $n-1$-ième de la fonction construite ici, avec une structure de tableau nécessaire pour que les opérations `+` et `*` ne provoquent pas de souci. [17]

Considérons par exemple l'équation $y''(x) = 2y'(x)+3y(x)-\cos x$. C'est une équation différentielle linéaire du second ordre, qui fait intervenir une fonction G ayant cette fois pour paramètres x, $y(x)$ et aussi $y'(x)$. On peut en fait voir cette équation différentielle comme une équation différentielle linéaire d'ordre 1 sur des vecteurs de dimension 2, à savoir

$$(y'', y') = (2y' + 3y - \cos x, y') = G_2((y', y), x),$$

où $G_2(Y, x) = (2Y[0] + 3Y[1] - \cos(x), Y[0])$ en adaptant la notation.

Construire pas à pas des fonctions à l'aide d'images en un nombre de plus en plus grand au fur et à mesure que l'intervalle se restreint prend tout son intérêt lorsqu'on a l'appui d'une représentation graphique, et c'est précisément ce qui reste à découvrir en Python. Le TP 9 présente le sous-module `matplotlib.pyplot` qui sera utilisé dans les TP 10 et 11 pour travailler avec les méthodes de Newton et d'Euler, entre autres.

Pour aller plus loin, la méthode d'Euler implicite consiste à approcher en tout point $G(y(t), t)$ non pas par $G(y(x), x)$ (point précédent) mais $G(y(x+h), x+h)$, créant ainsi une équation d'inconnue $y(x+h)$ qu'on résoudra par exemple à l'aide de la méthode de Newton. [18]

17. Qui se rappelle ce que donnent `(1, 2)+(3, 4)*5` et `numpy.array([1, 2]) + numpy.array([3, 4])*5` ?

18. Cette méthode a fait l'objet du concours blanc de 2017.

3.5 L'essentiel

Le principe évoqué au chapitre précédent s'applique tout autant ici : mémoriser les algorithmes par cœur n'est pas ce qui est attendu, mais plutôt de les comprendre. C'est notamment pour cela que le pivot a été présenté sous sa version modularisée, afin de voir qu'il n'y a que des courts fragments de code qui s'assemblent harmonieusement.

La structure de tableau, et plus généralement tout ce que `numpy` apporte d'utile, est un incontournable pour les concours, dès lors qu'il s'agit d'appliquer ce que l'on étudie en informatique pour des problèmes concrets d'autres disciplines. Il faudra prendre garde à ne pas confondre les tableaux et les listes, notamment au niveau des fonctions pour lesquelles il n'y a pas de compatibilité, ainsi qu'au niveau des différences de comportement (l'exemple fondamental étant `append`).

La méthode d'Euler est tombée à tous les concours en 2015, première session où toutes les filières avaient une épreuve d'informatique telle qu'on la connaît actuellement. En outre, elle se cache pour ainsi dire derrière tous les TIPE comprenant de la modélisation physique, quand il ne faut pas utiliser de méthode plus puissante encore. Ainsi, on accordera un soin particulier à la réalisation du TP associé dans sa totalité.

Chapitre 4

Bases de données

4.1 Introduction

Un des fondements et motivations de l'étude des bases de données relationnelles et, du point de vue théorique, de l'algèbre relationnelle est l'existence de problèmes algorithmiques plus ou moins concrets pour lesquels les structures de données étudiées auparavant ne sont pas aussi efficaces que l'on peut l'espérer.

Avant tout, une question en lien avec notre sujet : on souhaite trier un ensemble de personnes. Selon quels critères ?

Alors que les listes ou tableaux d'entiers se triaient dans presque tous les cas suivant l'ordre canonique (croissant ou décroissant), il n'y a pas d'ordre de référence pour les humains. C'est bien évidemment parce que chacun a ses particularités, qu'on représentera pour des objets quelconques par des attributs.

De tels attributs permettent un regroupement (on peut par exemple mettre ensemble toutes les personnes ayant des lunettes / des lentilles / subi une chirurgie au laser / etc.) voire un classement (tri par la taille, l'âge, etc.).

En pratique, le classement est toujours imaginable dans la mesure où on peut associer à une valeur d'un attribut une chaîne de caractères qui la décrit et considérer par exemple l'ordre lexicographique.

Le problème des structures de données que l'on connaît, c'est que si on tente de présenter des données regroupées selon un certain attribut, faire un regroupement selon un autre sera très complexe.

Imaginons qu'on mette les étudiants de SPE d'un lycée dans un tableau, ce tableau contenant un sous-tableau par classe et dans ce sous-tableau les élèves sont regroupés par classe l'année précédente.

Récupérer la liste des anciens PCSI 2 nécessite un parcours de chaque sous-tableau pour accéder au sous-sous-tableau de la PCSI 2. Bien entendu, à ce moment-là, récupérer la liste des PSI est rapide car elle est déjà fournie (on peut avoir besoin d'aplatir les sous-sous-tableaux suivant l'énoncé). L'idée est cependant de garder la même simplicité pour tous les problèmes de ce genre.

Ainsi, nous allons étudier dans ce chapitre les moyens d'exécuter des opérations (ou requêtes, sur un modèle concret de bases de données) sur des structures d'objets pourvus d'attributs.

4.2 Algèbre relationnelle

L'*algèbre relationnelle* est un modèle théorique développé dans les années 1970, en grand lien avec la théorie des ensembles. Nous allons nous appuyer sur cette dernière pour définir notre vocabulaire, et dans la section suivante un parallèle sera fait entre chaque opération présentée ici et chaque requête introduite dans le modèle concret des bases de données.

Soit un objet caractérisé par un certain nombre d'*attributs*. On va considérer cet objet comme le n-uplet formé par les valeurs de ces attributs dans un ordre fixé pour chaque objet.[1] Il est possible que des doublons soient alors créés dans un ensemble d'objets, ce qu'on autorise dès lors qu'on ne met pas en place une sorte d'identifiant sur lequel nous reviendrons.

Une *relation* est un ensemble fini d'objets (appelés ici *valeurs* ou *enregistrements*, dont le nombre est le *cardinal* de la relation, noté par un croisillon) ainsi définis.

1. Ceci peut faire penser à la programmation orientée objet, bien que cette dernière ait d'autres buts et utilités. Pour autant, on peut tout à fait s'en servir pour traiter des entrées de bases de données.

On précisera habituellement la structure de cette relation (voir deux définitions plus loin), tout comme on précise en théorie des ensembles sur quels ensembles une relation binaire est construite. L'homonymie n'est par ailleurs pas due au hasard.

Le *domaine* d'un attribut est l'ensemble de ses valeurs possibles (et non pas nécessairement l'ensemble des valeurs prises, tout comme une fonction réelle peut ne pas être surjective quand bien même son ensemble d'arrivée est défini comme étant $\mathbb{R}$ conventionnellement).

Un *schéma relationnel* est la précision de la structure d'une relation, sous la forme de n-uplet formé des couples (attribut, domaine) dans l'ordre (on pourra omettre le domaine dans certains cas).

Afin de permettre une représentation efficace d'une relation, on utilisera simplement un tableau (au sens habituel du terme), d'où le nom de *table* que l'on verra dans la section sur les bases de données.

Les opérations sur les relations, ou *opérateurs relationnels*, sont essentiellement des recherches d'enregistrements ou se fondent dessus en effectuant un traitement sur les résultats. Produire un même résultat peut se faire de plusieurs façons différentes, mais pas nécessairement avec la même complexité, que ce soit du point de vue algorithmique ou de celui de la taille de la formule mathématique correspondante (les deux sont souvent en relation).

Soient deux relations R_1 et R_2 de même schéma. L'*union* (ou réunion) de R_1 et R_2, notée $R_1 \cup R_2$, est l'ensemble des enregistrements de R_1 et de ceux de R_2. Attention, un élément apparaissant au moins une fois dans R_1 et au moins une fois dans R_2 apparaîtra toujours exactement une fois dans $R_1 \cup R_2$. Par analogie, on définit l'*intersection* de R_1 et R_2, notée $R_1 \cap R_2$, comme l'ensemble des enregistrements communs à R_1 et R_2. Enfin, la *différence* de R_1 et R_2, notée $R_1 - R_2$, est l'ensemble des enregistrements de R_1 qui ne sont pas dans R_2. Dans les trois cas, le schéma de la relation « composée » est le schéma commun aux deux relations. Ceci étant, si on n'impose pas que les schémas soient communs mais seulement *compatibles* (c'est-à-dire les noms d'attributs peuvent ne pas être les mêmes, mais les domaines doivent être identiques, et à plus forte raison le nombre d'attributs doit être identique pour les deux relations).

On note que certaines propriétés de la théorie des ensembles ne sont plus forcément

vraies en algèbre relationnelle : en raison de la présence possible de doublons, il n'est pas exclu que $\#(R_1 \cup R_2) < \max(\#R_1, \#R_2)$ et que $\#(R_1 \cap R_2) > \min(\#R_1, \#R_2)$. De même, la formule $\#(R_1 \cup R_2) = \#R_1 + \#R_2 - \#(R_1 \cap R_2)$ est potentiellement fausse.

Soient une relation R de schéma S et S' un sous-ensemble de S. La *projection* de R selon S', notée $\pi_{S'}(R)$, est la relation de schéma S' dont les enregistrements ne sont alors définis que par les valeurs pour les attributs dans S'. Ici, des doublons peuvent être créés, et le cardinal de la projection est identique au cardinal de R.

Soient R_1 et R_2 deux relations de schémas respectifs S_1 et S_2. Le *produit cartésien* de R_1 et R_2, noté $R_1 \times R_2$, est l'ensemble des couples d'enregistrements (e_1, e_2), où $e_1 \in R_1$ et $e_2 \in R_2$. Le schéma de ce produit cartésien est la réunion avec doublons de S_1 et de S_2, et son cardinal est le produit des cardinaux des relations. Si de plus $S_2 \subseteq S_1$, on peut définir la *division cartésienne* de R_1 par R_2, notée $R_1 \div R_2$, par la relation, de schéma $S_1 \setminus S_2$, dont les enregistrements e sont tels que pour tout enregistrement e_2 de R_2, la fusion de e et de e_2 donne un enregistrement de R_1. On note que $(R_1 \times R_2) \div R_2 = R_2$, mais $(R_1 \div R_2) \times R_2 \subseteq R_1$ sans garantie d'égalité (et il faut aussi que $R_1 \div R_2$ existe, pour commencer). [2]

Soient une relation R de schéma S et A' un attribut de domaine D commun avec un attribut A dans S. Le *renommage* de A en A', noté $\rho_{A \to A'}(R)$, donne une nouvelle relation dont le schéma est S'. On peut imaginer un renommage multiple, par ailleurs. Puisque le nom de l'attribut n'a pas vraiment d'influence sur les enregistrements, on justifie ici le fait de nécessiter des schémas compatibles pour l'union, l'intersection et la différence, plutôt que nécessairement des schémas identiques.

Soient S un schéma relationnel et R une relation de schéma S. Étant donné un *prédicat* P, c'est-à-dire une fonction dépendant d'un certain nombre d'arguments et retournant un booléen, d'argument un enregistrement de R, la *sélection* de R selon P est l'ensemble des enregistrements e dans R tels que $P(e)$ soit vrai, qu'on écrit simplement $\sigma_P(R) = \{e \in R \mid P(e)\}$. Un prédicat de base est la comparaison de la valeur pour un certain attribut avec un certain élément de son domaine.

2. On peut faire un parallèle avec la division euclidienne. La division cartésienne de R_1 par R_2 est la plus grande relation R_q pour laquelle il existe une relation R_r telle que la réunion de R_r et de $R_q \times R_2$ soit R_1.

Remarque : Le rapport étroit entre la logique et la théorie des ensembles[3] apparaît dans les propriétés suivantes.

— $\sigma_{P \text{ ET } Q}(R) = \sigma_P(R) \cap \sigma_Q(R)$ (privé des doublons) ;
— $\sigma_{P \text{ OU } Q}(R) = \sigma_P(R) \cup \sigma_Q(R)$ (idem) ;
— $\sigma_{\text{NON } P}(R) = R - \sigma_P(R)$ (idem) ;

La *jointure symétrique*[4] de R_1 et R_2 selon l'égalité d'attributs $A_1 = A_2$ (où chaque A_i est dans le schéma de R_i), notée $R_1 \bowtie_{A_1=A_2} R_2$, est l'ensemble des enregistrements du produit cartésien $R_1 \times R_2$ dont les valeurs pour les attributs A_1 et A_2 sont égales.

On notera qu'il n'est pas nécessaire que le domaine des attributs soient les mêmes, et que la jointure sera vide si l'intersection des domaines est vide, par exemple. Par ailleurs, puisqu'on introduit ici une redondance, on peut se permettre de ne retenir qu'un attribut entre A_1 et A_2 dans le produit cartésien.

Ainsi, $R_1 \bowtie_{A_1=A_2} R_2 = \sigma_{A_1=A_2}(R_1 \times R_2)$, avec éventuellement l'application d'une projection retirant A_1 ou A_2 au résultat.

Restent les fonctions d'agrégation, que nous verrons plus particulièrement dans la section sur les bases de données.

Toute l'algèbre relationnelle repose sur la composition d'opérations présentées ci-avant.

4.3 Bases de données relationnelles

4.3.1 Introduction

Une base de données est une implémentation d'un ensemble de relations (appelées ici *tables*) telles que décrites précédemment. Le stockage en lui-même de la base de données dépend du système de gestion, l'une des possibilités étant du texte clair, ou de façon plus pratique un tableur.

Les opérations de l'algèbre relationnelle sont bien entendu implémentées elles aussi (du moins les principales), ainsi que des opérations de modification, telles que l'ajout d'un enregistrement (appelé ici *entrée*), sa suppression ou sa modification, de même

3. Au programme de mathématiques dans d'autres filières.
4. les autres jointures n'étant pas au programme

que la modification de la structure d'une table (ce qui modifie au passage les entrées), voire l'ajout ou la suppression de tables, entre autres.

4.3.2 Clés

Dans une base de données, il peut être utile de disposer de contraintes sur les entrées assurant qu'on puisse identifier chacune par un index, notamment en vue de faire des relations entre tables, et surtout afin de garantir l'unicité d'une valeur pour un attribut, ou éventuellement seulement d'un n-uplet.

C'est le concept de *clé*. Une clé (en SQL, cela se repère par un champ `UNIQUE` dans la structure) est un sous-ensemble d'attributs tel que l'insertion ou la modification d'une entrée ne soit possible que si le n-uplet de valeurs pour ces attributs ne se retrouve dans aucun autre enregistrement.

Si la table n'a pas de doublon, son schéma est en pratique une clé, mais bien entendu on préfère que le nombre d'attributs soit aussi restreint que possible.

En pratique, on ajoute souvent à la structure d'une table un attribut spécial, dont le nom fait référence à un identifiant, prévu pour être une clé, on dispose même d'une fonctionnalité dite `AUTO INCREMENT`, grâce à laquelle on n'a pas besoin de renseigner la valeur de l'identifiant, car un compteur associé à la table sait quelle sera la prochaine valeur à affecter.

Une *clé primaire* est une clé pour laquelle la recherche est optimisée ; tout ceci se fait au sein de la base de données.

Le concept de *clé étrangère* est en lien avec celui que nous venons de présenter, à ceci près qu'il n'y a pas d'unicité dans la table où une telle clé figure. En fait, une clé étrangère est un attribut correspondant normalement à une clé (souvent la clé primaire, pour profiter de l'optimisation) dans une autre table, là aussi en vue de faire des associations.

Par exemple, si on dispose d'une table contenant une liste d'élèves et d'une table contenant une liste d'épreuves, on peut créer une troisième table avec trois attributs : une clé étrangère correspondant à l'identifiant d'un élève, une autre correspondant à l'identifiant d'un devoir, et finalement la note obtenue.

C'est le contenu de cette troisième table qui peut être étudié afin de faire des calculs, et les résultats seront traités à l'aide des deux premières tables (une fois la moyenne, le maximum, etc. calculés, à qui cela correspond-il ?).

4.3.3 Le langage SQL

Afin de communiquer avec un serveur de bases de données, un langage spécifique est nécessaire. Ce langage consiste à formuler des requêtes, reprenant les opérations de l'algèbre relationnelle, et à en récupérer les résultats. Aux requêtes classiques de gestion du contenu des tables s'ajoutent des requêtes d'administration, moins souvent utilisées (et qu'on préfèrera largement faire faire par un intermédiaire).

Un langage quasi incontournable pour exécuter ces requêtes est SQL [5]. En pratique, de nombreux systèmes de gestion de bases de données (ou SGBD) utilisent des surcouches de SQL, avec du sucre syntaxique. On citera les systèmes MySQL [6], PostgreSQL, Oracle, etc. Quant à SQLite, dont le nom est tout aussi connu [7], il ne s'agit pas d'un SGBD mais d'un moteur de bases de données (qu'il y a dans les SGBD, donc), nécessitant de mettre un peu plus les mains dans le cambouis.

Les requêtes principales concernant les données se regroupent en quatre catégories : insertion, recherche, mise à jour et suppression [8].

L'insertion se contente d'ajouter une entrée dans la table, en renseignant tous les attributs nécessaires (si des attributs manquent, il faut préciser quelles valeurs correspondent à quels attributs pour lever l'ambiguïté, ce qui permet également de renseigner les attributs dans l'ordre que l'on souhaite, heureusement).

Les autres requêtes nécessitent de préciser à l'aide de conditions sur quel(le)s entrée(s) l'opération doit être effectuée (il s'agit de la projection). Ces conditions portent sur les attributs de la table : comparaisons, tests d'égalité, voire recherche de sous-chaîne, etc. À ce titre, il est possible de faire appel à des fonctions prédéfinies dans le langage dont certaines sont données ci-après (fonctions d'agrégation).

5. pour *Structured Query Language*, qui se traduit par « langage de requête structurée »
6. le seul que j'aie utilisé jusqu'à présent
7. et qui est intégré à Edupython, entre autres
8. Pour ceux qui font l'option informatique, notez le rapprochement avec des structures de données abstraites composées.

La syntaxe est la suivante :

- Insertion : `INSERT INTO <table>(<attributs>) VALUES (<valeurs>)`, la partie renseignant les attributs entre parenthèses étant optionnelle si on les fournit tous ;
- Recherche : `SELECT <attribut>,...,<attribut> FROM <table>`, suivi de `WHERE <condition>` la plupart du temps, avec le joker `*` pour demander tous les attributs, ou la version sans doublon `SELECT DISTINCT`[9] ;
- Mise à jour : `UPDATE <table> SET <attribut>=<valeur>,..., <attribut>=<valeur> WHERE <condition>` ;
- Suppression : `DELETE FROM <table> WHERE <condition>`.

Ordinairement, les attributs sont entourés d'accents graves et les valeurs sont entourées de guillemets ou d'apostrophes quand il s'agit de chaînes de caractères (mais c'est autorisé pour les autres types aussi, le logiciel se charge quoi qu'il arrive de donner aux valeurs de chaque attribut le bon type).

La partie commençant par `WHERE` est toujours optionnelle, et si elle est absente on fait la requête sur toutes les entrées. Un point-virgule, séparant deux requêtes consécutives est fréquemment mis en toute fin, mais il n'y est pas nécessaire.

Exemples :

- `INSERT INTO ‘Etudiants‘(‘Nom‘,‘Prenom‘) VALUES ("Martin","Jean");` insère une entrée dans une table nommée `Etudiants` dont tous les champs sauf éventuellement `Nom` et `Prenom` précisés ici sont optionnels ;
- `INSERT INTO ‘Notes‘ VALUES (1,1,12);` insère cette fois une entrée dans une table à exactement trois champs ;
- `SELECT ‘Etudiant‘ FROM Notes WHERE ‘Epreuve‘ = 1 AND ‘Note‘ >= 10` sélectionne dans la table mentionnée les identifiants des étudiants ayant eu la moyenne à l'épreuve d'identifiant `1` ;
- `SELECT * FROM Etudiants WHERE Prenom REGEXP BINARY "J"` sélectionne tous les attributs des étudiants dont le prénom contient la lettre capitale `J`[10] ;
- `UPDATE ‘Notes‘ SET ‘Note‘ = ‘Note‘ + 2 WHERE ‘Epreuve‘ = 2` ajoute deux points à toutes les entrées correspondant à la deuxième épreuve ;

9. L'équivalent en algèbre relationnelle est une projection composée avec une sélection, en notant que ces deux opérations commutent ; pour supprimer les doublons, une astuce consiste à faire l'union avec la relation vide. Attention, la sélection vient du mot-clé `WHERE`, et `SELECT` induit une projection !

10. Le mot-clé `BINARY` permet que la recherche soit sensible à la casse, c'est-à-dire distingue les lettres capitales (majuscules) et en bas de casse (minuscule).

— `DELETE FROM Etudiants WHERE Nom = "Martin"` supprime tous les élèves dont le patronyme est `Martin`. Attention aux doublons ![11]

Pour aller plus loin, MySQL dispose de sucre syntaxique concernant la façon de présenter les résultats de la requête, et certains des paramètres ci-après, ne relevant pas à proprement parler de l'algèbre relationnelle, sont parfois propres à un système de gestion de bases de données ou à un autre.

Pour trier les résultats selon un attribut spécifique, on ajoutera (obligatoirement après la condition, si elle existe) `ORDER BY <attribut>` (ordre croissant) ou `ORDER BY <attribut> DESC` (ordre décroissant), avec d'autres possibilités dont le notable `ORDER BY RAND()` pour laisser le hasard décider.

Il est également possible de n'afficher qu'un nombre limité de résultats (ce qui se combine bien avec ce qui précède) en écrivant `LIMIT <nombre> OFFSET <premier>`, ce qui donnera les résultats à partir de celui dont l'indice (qu'on comprendra comme un indice en Python, en commençant également à zéro) est précisé en "*offset*" et au nombre précisé en "*limit*" ou en s'arrêtant avant s'il n'y en a pas assez.

Les requêtes concernant la structure des tables, ou des bases de données en général, sont soumises à la même remarque que les requêtes d'administration (qui gèrent les utilisateurs, entre autres) : dans de nombreux cas, on ne les effectuera pas en les saisissant mais par exemple à l'aide d'une interface.

On retiendra simplement `TRUNCATE <table>` pour vider une table (retirer toutes les entrées), `DROP TABLE <table>` pour la supprimer, ainsi que `DROP DATABASE <BDD>` pour supprimer une base de données entière, `RENAME TABLE <table> TO <nom>` pour renommer une table, `ALTER TABLE <table> ORDER BY <attribut>` pour la trier et `ALTER TABLE <table> ADD PRIMARY KEY(<attribut>)` pour y ajouter une clé primaire.

4.3.4 Correspondance avec l'algèbre relationnelle

Comme on l'a vu, les opérateurs booléens dans les conditions introduites par `WHERE` s'écrivent en toutes lettres (et en anglais, bien entendu) : il s'agit de `AND`, `OR` et `NOT` [12],

11. phpMyAdmin signale le nombre de lignes affectées par de telles requêtes, ce qui peut être utile pour arranger une catastrophe immédiatement.

12. Les lettres capitales sont essentiellement esthétiques, pour la lisibilité et la clarté.

auxquelles on ajoute `XOR`, correspondant à soit ... soit ..., mais on regrette l'absence de `NAND` et `NOR`, qui s'obtiennent néanmoins aisément par composition.

Si on souhaite plutôt faire une union, une intersection ou une différence de résultats de requêtes, il s'agit de combiner les requêtes en les séparant par les mots-clés `UNION`, `INTERSECT` ou `EXCEPT`.

Le renommage au sein d'une requête [13] se fait par le mot-clé `AS` après un nom d'attribut. Ceci est notamment utile lorsque la requête fait intervenir des fonctions d'agrégation, où exploiter le résultat nécessite quasiment de lui donner un nom d'attribut pratique. On peut également renommer des tables, notamment pour lever des ambiguïtés dans des jointures. Dans les deux cas, écrire le vrai nom puis l'alias sans `AS` est aussi autorisé mais moins esthétique.

Le produit cartésien se fait par la mention de tables séparées par des virgules ou par le mot-clé `JOIN` (simulant une jointure sans condition). [14] La division cartésienne, quant à elle, n'est pas implémentée par défaut.

La jointure utilise, on l'aura compris, le mot-clé `JOIN`, la condition associée s'exprimant juste après par `ON <égalité entre attributs>`.

Puisque les attributs peuvent provenir de deux tables, les noms d'attributs étant parfois identiques d'une table à l'autre (à éviter), d'éventuels noms d'attributs redondants sont à préfixer par le nom de leur table, il est même envisageable de tout préfixer. Exemple : `SELECT nom FROM classe JOIN edt ON classe.id = edt.classe WHERE edt.salle = 218 AND edt.date = "mercredi" AND edt.heure = 8`, pour la liste des noms des étudiants qui sont dans une classe qui a cours en 218 le mercredi à 8 h. Sans jointure, on écrirait `SELECT nom FROM classe WHERE id_classe IN (SELECT classe FROM edt WHERE id_salle = 218 AND date = "mercredi" AND heure = 8)`.

Terminons par les fonctions d'*agrégation*. Il s'agit ici d'appliquer une fonction à un nombre a priori indéterminé d'arguments, qui sont des valeurs d'attributs d'un certain nombre d'enregistrements ; les enregistrements sur lesquels la fonction est appliquée forment ce qu'on appelle un *agrégat*.

13. ... de recherche, en pratique

14. De ces deux versions, la virgule est la moins prioritaire en cas de requêtes complexes, mais le savoir n'est pas essentiel, un peu comme les opérateurs booléens bit-à-bit en Python.

Le regroupement se fait en écrivant `GROUP BY <attribut>`, après quoi il ne reste qu'une entrée pour chaque valeur ou n-uplet de valeurs du ou des attributs en question (agissant comme une clé), entrée qui est l'agrégat en question dont on peut sélectionner le résultat de l'application d'une fonction présentée dans la suite.

On notera que sans regroupement (ni sélection), la fonction s'applique sur tout le résultat de la sélection. Nous nous contentons de présenter ici les fonctions les plus classiques (celles du programme, en fait).[15] Il s'agit du comptage (`COUNT`), du minimum (`MIN`), du maximum (`MAX`), de la somme (`SUM`) et de la moyenne (`AVG`). Ainsi, on se donne des opérateurs supplémentaires dans l'algèbre relationnelle, opérateurs qui se traduisent en SQL par des fonctions au nom évident.

L'agrégation crée de nouveaux attributs (dont on a dit qu'ils sont renommés pour faciliter leur utilisation). Elle peut alors se composer avec des sélections (ou regroupements) après avoir été appliquée, ce qui n'est pas la même chose que les regroupements ou sélections avant d'appeler la fonction.

Par exemple, si on veut trouver les étudiants qui ont au dessus de 12 en moyenne dans une table dont les attributs sont un identifiant d'étudiant, un identifiant d'épreuve et une note (cf. TD 5), il faut faire un regroupement des entrées avec le même identifiant d'étudiant, un calcul de moyenne puis une sélection, ce qui s'écrit par exemple `SELECT Etudiant FROM (SELECT Etudiant,AVG(Note) AS Moyenne FROM notes GROUP BY Etudiant) AS Id WHERE Moyenne > 12`.[16] Une autre possibilité est de faire un filtrage en aval, dans la mesure où la clause introduite par `WHERE` est un filtrage en amont, donc avant le regroupement, donc sans pouvoir utiliser le résultat de fonctions d'agrégations. Le mot-clé pour le filtrage en aval est `HAVING`, il concerne des champs agrégés en priorité. Évidemment, on met dans ce cas `HAVING` après `GROUP BY`, et la requête précédente se réécrit donc `SELECT Etudiant FROM notes GROUP BY Etudiant HAVING AVG(Note) > 12`.

Bien entendu, les fonctions d'agrégations peuvent être appelées plus simplement. Par exemple, le cardinal d'une table s'obtient par `SELECT COUNT(*) FROM <table>`, et l'enregistrement maximisant un attribut : `SELECT MAX(<attribut>) FROM <table>`, qu'on peut projeter avec des attributs « compatibles ».

15. Une liste plus complète est donnée dans le mémento SQL accessible à la page `http://sql.sh/wp-content/uploads/2013/02/mysql-aide-memoire-sql-950px.png`.

16. La partie `AS Id` est imposée par SQL, car il faut faire un renommage lorsque l'on utilise ce qu'il appelle une « table dérivée ».

4.3.5 Encore un peu d'architecture

Dans le modèle qu'on a présenté, l'interaction avec la base de données n'est pas évoquée. On peut imaginer que l'utilisateur (humain ou programme) et la base de données sont seuls au monde, ce qui peut arriver mais n'est pas le reflet de la réalité.

En pratique, un serveur de base de données est souvent accessible en ligne à de nombreux utilisateurs, appelés *clients*, et d'éventuelles restrictions provenant du propriétaire du serveur peuvent limiter le nombre de connexions simultanées, mais deux clients effectuant en même temps des requêtes peuvent suffire à causer des problèmes, car suivant l'ordre de traitement des requêtes le résultat peut différer.

Ceci se rapproche de la théorie de la concurrence. On pourra également se renseigner sur le principe de l'exclusion mutuelle (*MUTEX*), utilisant des algorithmes pour verrouiller l'accès à des données (ou un programme, dans d'autres cas) lorsqu'il est déjà utilisé. Dans la pratique, ce genre de verrouillage doit être indétectable au sens où tout se passe suffisamment vite pour que les utilisateurs ne soient pas affectés par un tel conflit.

Une architecture simple, appelée *architecture client-serveur*, dans laquelle les clients communiquent directement avec le serveur (accessible via un réseau comme internet). La gestion des connexions simultanées se fait en quelque sorte par l'effectuation des requêtes en blocs appelés des *transactions*. Les requêtes sont ajoutés en file dans les transactions en fonction de leur moment d'arrivée et traitées dans le même ordre.

Ceci n'exclut évidemment pas qu'une même requête effectuée deux fois puisse donner deux résultats différents si une modification s'est glissée entre les deux occurrences.

Cependant, le plus souvent, il existe un intermédiaire entre les clients et le serveur, qui ne communiquent plus qu'indirectement. Cet intermédiaire est un *serveur applicatif*, qui va faire le lien et éventuellement encadrer les requêtes, voire les limiter.

L'architecture ainsi obtenue est appelée *architecture trois-tiers*[17]. Un exemple est la plate-forme phpMyAdmin, grâce à laquelle il est possible de gérer une base de données sans même connaître la syntaxe du langage SQL.

Les requêtes sont limitées par les droits dont l'utilisateur dispose, mais de telles

17. formée du *tiers utilisateur*, du *tiers applicatif* et du *tiers base de données*

restrictions sont en fait également mises en place lors d'une connexion directe à la base de données (et en pratique phpMyAdmin dispose d'une console SQL qui simule une communication directe avec le serveur).

Entre les deux architectures ci-avant, on peut citer l'architecture MVC (Modèle-Vue-Contrôleur), dans laquelle seules les modifications nécessitent de transiter par ce qui fait office de serveur applicatif, alors que consulter la base de données peut être fait directement.

4.4 L'essentiel

Dans ce chapitre, un constat s'impose : l'algèbre relationnelle en elle-même, bien que support théorique et domaine intéressant des mathématiques qui n'est pas enseigné dans cette matière en tant qu'extension de la théorie des ensembles, est délaissée aux concours au profit de son application aux bases de données, c'est-à-dire des questions de SQL.

Ainsi, faire totalement l'impasse sur ce chapitre est possible sans impact sur le reste d'un sujet, mais cela n'en demeure pas moins une mauvaise idée au regard de la facilité des questions qui en traiteraient.

La syntaxe de SQL a le bon goût d'être très proche d'une simple transcription en anglais de l'ordre que l'on souhaiterait donner à la console gérant la base de données (à l'exception notable près que la projection s'écrit en tout début alors qu'on n'obtient qu'à la fin les attributs que l'on voudrait récupérer). Ici encore, de l'entraînement s'impose, un entraînement qui est bien accessible à l'aide des deux bases de données mises à disposition pour le TD et le TP présent sur ces notes de cours.

TD 5 : Une base de données sommaire

Dans ce TD, nous allons travailler sur une base de données présentée en cours en première année, et effectuer des requêtes SQL, en plus de pouvoir procéder pour le travail chez soi à des manipulations à l'aide de la plate-forme PhpMyAdmin. L'adresse de la base de données est `http://phpmyadmin.online.net`, et l'identifiant est db86896.

La base de données peut être récupérée pour une utilisation individuelle sur un serveur personnel [18].

La structure associée est la suivante :

— Table Etudiants, avec les attributs Id (clé primaire avec auto-incrémentation, entier naturel), Classe (chaîne de caractères), Nom (idem) et Prenom (idem). On peut considérer que (Nom, Prénom) est également une clé.
— Table Examens, avec les attributs Id (clé primaire avec auto-incrémentation, entier naturel), Date (chaîne de caractères) et Coeff (entier naturel).
— Table Notes, avec les attributs Etudiant (entier naturel), Examen (entier naturel) et Note (entier naturel). Le couple (Etudiant, Examen) est une clé.

Dans chacun de ces exercices, on se contentera d'écrire une requête correspondante.

Exercice 1 : Déterminer qui a la meilleure note à l'examen numéro 2 parmi les MPSI 2.

Exercice 2 : Déterminer la moyenne du meilleur étudiant.

Exercice 3 : Déterminer combien de MPSI 1 ont sous la moyenne.

Exercice 4 : Déterminer quelle classe a la meilleure moyenne.

Pour les deux exercices suivants, on considère que les étudiants de toutes les classes passent les mêmes examens.

Exercice 5 : Déterminer dans quelle classe est l'étudiant ayant majoré le premier examen.

18. `http://jdreichert.fr/Enseignement/CPGE/IPTSUP/bdd_td_5.sql`

Exercice 6 : Déterminer le nombre de majors par classe.

Exercice 7 : Déterminer la classe avec le plus grand pourcentage d'étudiants ayant la moyenne.

Lexique

Deuxième partie

Travaux pratiques

TP 0 : Introduction

En guise de (re)découverte de l'algorithmique, le TP ici proposé consiste à ressortir de l'oubli [19] un langage informatique, Logo [20], et d'en utiliser les principes pour écrire en pseudo-code des instructions basiques pour faire des dessins en déplaçant une tortue.

Pour les éventuels intéressés, il est possible d'utiliser un interpréteur en ligne du langage Logo à l'adresse `http://www.calormen.com/jslogo/`, une aide à la syntaxe étant fournie.

Les textes en bleu de chaque TP représentent ce que les étudiants découvrent par eux-mêmes. Les textes en vert sont des exercices corrigés à la demande.

Comme signalé au premier paragraphe, dans notre langage, appelé sommairement « Logo-- », nous réalisons des figures en déplaçant une tortue dans un repère orthonormé du plan à l'aide de différentes instructions.

Contrairement au cas de la plupart des librairies graphiques des langages de programmation habituels, on s'interdira ici de forcer une nouvelle position par « téléport ». Tous les déplacements se feront en fonction de l'orientation de la tortue, à l'aide des instructions **avancer** (A), **reculer** (R), **tourner à gauche** (TG) et **tourner à droite** (TD). Tourner se fait sur place et à angle droit. Par commodité, on autorise la syntaxe « A×n », où n est un entier, qui représente le fait d'avancer de n unités. De même pour « R×n ».

On constate une redondance entre les instructions A×n et R×n, de même qu'entre les instructions TG et TD. La plupart des langages disposent de ce qu'on appelle de manière informelle « sucre syntaxique », des instructions qui ne sont pas indispensables mais qui facilitent la vie du programmeur et n'ont pas à être redéfinies par lui-même avant d'écrire un programme.

À l'aide des instructions de déplacement, la tortue peut aller de n'importe quelle configuration (la donnée de l'abscisse, de l'ordonnée et de l'orientation) à n'importe quelle autre. La tortue disposant d'un crayon, son déplacement est alors tracé.

Sans restriction, les seules figures qui peuvent être obtenues sont celles qu'on peut

19. tout est relatif, il existe tout de même un module en python qui lui rend hommage
20. `http://fr.wikipedia.org/wiki/Logo_(langage)`

représenter sans lever le crayon. Comme il sera exclu de téléporter la tortue (ce qui est cependant possible en Logo), il faut trouver une solution. Parler de saut en serait une, mais pour se conformer à Logo on préfèrera ajouter les instructions **lever le crayon** (L) et **baisser le crayon** (B). Au début, le crayon est levé. On considère qu'il n'y a pas de souci si on lève un crayon levé ou si on baisse un crayon baissé.

Afin de disposer d'une puissance de calcul suffisante, on ajoute aux instructions les constructions de base de la programmation :

- Si <condition> alors <instructions> sinon <instructions> Fin Si.
  La partie « sinon » n'est pas obligatoire.
- Pour <nom de variable> de <seuil> à <seuil> faire <instructions> Fin Pour.
  La variable choisie est augmentée ou diminuée de 1 *à chaque passage dans la boucle, pour progresser du premier seuil au second.*
- Tant que <condition> faire <instructions> Fin Tant que.

Le balisage de fin n'est pas nécessaire dans les langages, ou les conventions d'écriture de pseudo-code, pour lesquels il existe un moyen de prévenir les ambigüités.

Exercice 1 : Réaliser un programme qui permet de tracer six lignes horizontales de $(0, i)$ à $(5, i)$ pour $-5 \leq i \leq 0$, avec pour configuration de départ $(0, 0)$ et une orientation vers la droite. Comparer avec les programmes d'autres étudiants.

Exercice 2 : Réaliser un programme qui permet de tracer une spirale partant de $(0, 0)$ construite de la manière suivante : 1 pas à droite, 1 pas en haut, 2 pas à gauche, 2 pas en bas, 3 pas à droite, etc. jusqu'à avoir fait douze étapes. Essayer de trouver une façon de commencer par « Pour i de 1 à 12 ».

Exercice 3 : Réaliser un programme qui envoie la tortue en $(0, 0)$ avec l'orientation vers le haut, quelle que soit sa position de départ. Il peut être pratique de commencer par forcer une orientation.

Pour ce faire, j'offre deux fonctions de repérage de la tortue :

- `pos()` renvoie le couple de coordonnées de la tortue. On peut s'en servir en écrivant `(x,y) <- pos()`, ce qui a pour effet de mettre dans x l'abscisse de la tortue et dans y l'ordonnée de la tortue au moment de l'appel de la fonction ;
- `orientation()` renvoie l'orientation de la tortue, parmi « haut », « bas », « gauche », « droite ».

Exercice 4 : Trouver le moyen d'écrire la fonction `orientation()` à l'aide de `pos()`. On prendra garde à ce que la tortue soit dans la même position et la même orientation au début et à la fin de l'exécution (sinon, il se produit ce qu'on appelle un effet de bord).

TP 1 : Familiarisation avec Python

Au cours de l'année (et de la suivante), les TP réalisés dans le cadre de l'enseignement de l'informatique pour tous nécessiteront l'écriture de programmes en Python. Il est bien entendu possible de réaliser cela seul sur un éditeur de texte, mais de nombreuses solutions sont faites pour faciliter la vie lors des différentes étapes de l'élaboration de programmes, ne serait-ce que la coloration syntaxique qui permet de détecter les coquilles les plus flagrantes et les erreurs de parenthésage.

Dans ce lycée, les ordinateurs disposent d'IDE [21], notamment Edupython, qui est en outre bien documenté [22] et Pyzo, qui est actuellement disponible pour toutes les épreuves sur machine en Python aux concours. L'objectif de ce TP est de manipuler ces logiciels et de découvrir Python par la même occasion.

Pour commencer, Edupython propose de gagner du temps lorsque l'on souhaite utiliser le module `turtle` : dans le menu, en choisissant un nouveau fichier, il est possible de cliquer sur « tortue », ce qui crée un nouveau fichier contenant deux lignes d'en-tête important les fonctions de la bibliothèque lycée (regroupant de nombreux modules utiles, et préchargeant en gros toutes les fonctions mathématiques utilisables en lycée, mais pas seulement) et le module `turtle`, renommé pour le script `tortue`, c'est-à-dire qu'au lieu de préfixer les fonctions du module par « `turtle.` », il faut les préfixer par « `tortue.` » [23]. En outre, la dernière ligne, `tortue.mainloop()`, affiche le résultat du corps du fichier dans une fenêtre séparée où la tortue opère. Pour pouvoir faire tourner de nouveau un programme, il faut fermer la fenêtre graphique produite. Il est possible de le faire par un simple clic de souris dessus à condition d'avoir utilisé l'instruction `tortue.exitonclick()`.

En ce qui concerne Pyzo, l'apparence est différente, mais on retrouve les caractéristiques propres aux IDE, et notamment ce qui nous intéresse le plus : une zone éditeur et une zone console. Les raccourcis sont évidemment différents entre Edupython et Pyzo, et le lecteur curieux pourra les découvrir en profondeur. On signalera le plus utile : exécuter le script entier, ce qui se fait par `Ctrl+F9` avec Edupython et `Ctrl+E` avec Pyzo.

Les commandes de base du module `turtle` sont `forward(x)` et `back(x)`, où `x` est

21. sigle anglais s'explicitant en français « environnement de développement intégré »
22. `http://download.tuxfamily.org/edupython/EduPython1.0.pdf`
23. ce que je ne ferai pas dans la suite

le nombre de pixels du déplacement, `left(a)` et `right(a)`, où a est l'angle de rotation (en degrés par défaut, mais une fonction permet de changer cela), `up()` et `down()` pour lever ou baisser le crayon. Voici aussi quelques commandes pratiques : `hideturtle()` et `showturtle()` pour masquer ou afficher la tortue, `speed(vitesse)` pour régler la vitesse (un entier de 1 à 10 pour les vitesses normales, croissantes, ou 0 pour la vitesse maximale), `pencolor(couleur)` qui prend en argument soit un nom anglais de couleur entre guillemets [24] soit trois réels entre 0 et 1 indiquant respectivement les degrés de rouge, de vert et de bleu [25], `pensize(taille)` pour modifier l'épaisseur du dessin, `bgcolor(couleur)` pour changer la couleur de fond.

D'autres commandes permettent de tracer une figure géométrique directement, notamment les cercles, ce qui serait laborieux autrement.

La syntaxe des tests et boucles en Python est la suivante : "`if ...:`" [26] représente le « si », "`for ...in ...:`" représente le « pour » et nécessite de donner une structure à parcourir et non deux seuils [27] et "`while ...:`" représente le « tant que ».

Dans tous les cas, il faut aller à la ligne et faire ce qu'on appelle **indenter** : décaler toutes les lignes du corps de chaque boucle ou conditionnelle d'au moins une espace, et d'autant d'espaces à chaque ligne (bien entendu il y en a plus lorsque l'on imbrique d'autres boucles ou tests).

Exercice 1 : Réécrire avec la syntaxe de Python les programmes permettant de tracer les figures des deux premiers exercices du TP 0. Exécuter ces programmes sur les deux éditeurs.

Exercice 2 : Recopier les exemples donnés dans la documentation d'Edupython, en constatant le cas échéant à quel point une erreur dans l'indentation change tout. [28]

Exercice 3 : À partir des exemples donnés, modifier les instructions pour obtenir des

24. En Python, on utilise les apostrophes et les guillemets, simples ou triples.
25. Le code RGB est un classique de la manipulation d'images en informatique, il est en général donné par trois entiers entre 0 et 255.
26. Attention : le double point est obligatoire ; en outre, le respect de la casse (minuscules et majuscules, en français courant) est important : par exemple, écrire `If` au lieu de `if` déclencherait une erreur.
27. En pratique, la structure générique est `range(n)` qui contient les entiers de 0 à $n-1$ inclus. On peut donner un autre argument, l'entier de départ, à `range`, voire un troisième qui serait le pas.
28. L'indentation disparaît quand on copie-colle depuis un PDF, malheureusement pour vous !

variantes des dessins précédents. Le but est d'anticiper le résultat, voire d'obtenir un résultat choisi à l'avance. Les seules limites sont celles de l'imagination (ou pas). [29]

Pour créer une variable, il suffit de l'affecter, c'est-à-dire la mettre du côté gauche d'un simple signe égal. La première valeur de la variable, qu'elle conservera jusqu'à son éventuelle réaffectation, est alors ce qu'il y a du côté droit du signe égal. À ce moment-là, toutes les occurrences du nom de la variables seront remplacées par la valeur en question pour Python.

Attention : La variable créée par une boucle `for` n'a d'existence que dans la boucle dans certains langages, mais Python fonctionne différemment. Pour autant, mieux vaut rester attentif.

Pour définir une fonction, on utilise la syntaxe suivante : `def nom_fonc(arguments):` puis, comme pour tout bloc, on indente toutes les lignes correspondant à la définition. Par exemple, on peut écrire la fonction dessinant (en admettant que le crayon soit baissé) un carré de côté `n` pixels ainsi :

```
def carre(n):
  for i in range(4):
    tortue.forward(n)
    tortue.right(90)
```

Le retour à l'indentation normale délimite la définition de la fonction. Appeler `carre` avec une valeur entière donnera le dessin correspondant une fois `tortue.mainloop()` ajouté.

Pour tirer un nombre au hasard, on utilise les fonctions du module `random`, également dans la bibliothèque `lycee`. La fonction `random()` retourne un réel entre 0 inclus et 1 exclu (sachant que de toute façon la probabilité de tomber sur zéro pile est nulle), la fonction `randint(a,b)` retourne un entier entre a et b inclus, et la fonction `uniform(a,b)` retourne un réel entre a et b, dans tous les cas suivant une loi (discrète ou moralement continue suivant le cas) équirépartie. D'autres fonctions dans le module permettent de manipuler les lois normales, etc.

Exercice 4 : Écrire une séquence d'instructions permettant d'obtenir une ligne de 400

29. Le web regorge d'exemples de programmes utilisant le module turtle et donnant des dessins magnifiques...

pixels (si possible assez épaisse) avec un dégradé de couleur, d'abord d'une couleur précise vers le noir (triplet (0,0,0)), puis vers le blanc (triplet (1,1,1)), puis d'une couleur à l'autre (penser aux barycentres[30]).

Exercice 5 : Écrire une séquence d'instructions permettant d'obtenir un cercle dont les 8 secteurs angulaires de 45 degrés sont d'une couleur aléatoire. Pour le plaisir, on peut combiner cet exercice avec le précédent pour obtenir 8 dégradés.

Exercice 6 : Écrire une séquence d'instructions pour simuler une marche aléatoire de la tortue et voir ses déplacements. On pourra envisager pour condition d'arrêt la sortie d'un carré centré en l'origine et de côté entre 20 et 40 unités au choix.

30. On va faire comme si c'était encore au programme en mathématiques...

TP 2 : Entrées et sorties
Travail à faire pendant la séance
1 `print` et ses subtilités

Pour commencer, définir en Python les trois fonctions ci-dessous :

```
def carre(n):
    print(n*n)

def carre2(n):
    return n * n

def carre3(n):
    print(n*n)
    return 42
```

Exercice 1 : Appeler les trois fonctions dans la console pour `n` valant `42`. Faire en suite de même dans l'éditeur. Commenter.

Exercice 2 : Écrire dans les deux endroits `print(carre(42))`, puis de même pour les deux autres fonctions. Commenter.

Exercice 3 : Écrire dans la console `carre(42)+36`, puis de même pour les deux autres fonctions. Commenter.

Exercice 4 : Quelle est la valeur de retour de `carre(n)` pour tout `n` ? Mais alors, pourquoi `print(carre(42))` imprime-t-il `1764` ? Comparer `print` et `return`.

Exercice 5 : Détailler le fonctionnement de `print` d'après son comportement sur :

```
x = """Bonjour
le monde"""
for i in range(10):
    print("Le caractère numéro", i, "de", "x", "est", x[i],
"et le caractère numéro", i+6, "de", x, "est", x[i+6], sep="*", end="!")

r = range(10)
print("range(10) :", r, "et la liste :", list(r), sep="\n")
```

2 Lecture du flux d'entrée

Exercice 6 : Écrire dans la console `input("Je vais doubler ce nombre :") * 2`, commenter le résultat. Attention, en Python 2, il n'y aurait rien de spécial à signaler.

Exercice 7 : Utiliser `type(input("Entrée libre :"))` en saisissant des valeurs de toutes sortes. Commenter le résultat.

Exercice 8 : Refaire les deux questions précédentes en utilisant `eval(input("etc."))`.

Exercice 9 : Écrire une fonction qui renvoie une liste de nombres demandée à l'utilisateur selon un protocole au choix.

Exercices pour réfléchir après la séance

À partir du second semestre, la plupart des TP seront suffisamment fournis pour que la séance de deux heures ne suffise pas à les terminer. La partie ainsi laissée de côté n'est cependant pas superflue.

Ainsi, il est recommandé de revenir sur ce TP en fin de semestre, une fois le cours correspondant traité.

3 Lecture d'un flux et écriture dans un fichier

On rappelle les commandes utiles pour ce TP : `fichier = open(chemin,mode)` met dans la variable `fichier` ce qu'on appelle un objet, dont la partie de la manipulation qui nous intéresse regroupe les fonctions de lecture, d'écriture et de fermeture.

On écrira alors `contenu = fichier.read()` pour mettre dans une variable l'ensemble du fichier[31], `fichier.write(texte)` pour écrire dans le fichier (dans les deux cas, sous réserve que le mode d'ouverture le permette, qui suppose lui-même des droits suffisants sur le fichier et/ou dans le dossier où le fichier, accessible en suivant le chemin précisé, se trouve), et `fichier.close()` une fois la manipulation terminée.

Concernant les chemins, pour ne pas parler du système d'exploitation utilisé, nous

31. Ajouter un argument entre les parenthèses précise le nombre maximal de caractères à lire.

nous limitons à des chemins relatifs, entre le dossier courant de Python (dépendant de l'éditeur utilisé) et le dossier contenant le fichier à lire. Dans tous les cas, les fichiers et dossiers sont organisés dans une arborescence, et naviguer d'un dossier à l'autre se fait d'un dossier à son parent, toujours noté `../` (à la unix) ou `..\` (à la windows), ou d'un dossier à un de ses fils, noté par son nom suivi d'une barre oblique de sens adéquat.

Ainsi, si je suis dans le dossier Jeux de mon bureau et que je veux accéder au fichier `lolcat.jpg` du dossier `Images` de mon bureau, je dois suivre le chemin `../Images/lolcat.jpg`.

Les fonctions servant à la navigation sont dans le module `os`.

La fonction `os.listdir(chemin)` retourne la liste des éléments du dossier accessible via `chemin`, avec différentes erreurs possibles : si le chemin n'existe pas, s'il mène sur un fichier et non un dossier, si on n'a pas de droits de lecture dans le dossier, etc. Pour changer de dossier, on utilise `os.chdir(chemin)`, avec les mêmes erreurs possibles, sachant qu'on peut déterminer le dossier où l'on se situe actuellement par `os.getcwd()`. D'autres fonctions du module sont documentées en écrivant `help(os)`.

Avant tout, un complément sur les chaînes de caractères. La recherche de motifs fait l'objet d'un cours en fin de semestre, mais bien entendu un tel algorithme a été écrit en Python. En plus de l'opérateur infixe `in` déterminant si une valeur est dans un itérable, ou si une chaîne de caractères est incluse dans une autre, Python 3 dispose des méthodes `index` et `find`.

L'avantage de `find` est qu'elle ne retourne pas de message d'erreur si la recherche échoue (mais la valeur `-1`), son inconvénient est qu'elle ne peut s'utiliser que sur une chaîne de caractères.

Les deux ont la même syntaxe ; la voici pour `find` : `chaine.find(x,debut,fin)` retourne le premier indice `i` tel que `i` $\geq$ `debut`, `chaine[i:(i+len(x))] == x` et `i+len(x)` $\leq$ `fin`. On peut omettre l'argument `fin`, et aussi l'argument `debut` (mais à condition de ne pas préciser de fin).

Cette partie du TP s'appuie sur un fichier de faux-texte [32] qu'il faut commencer par récupérer et mettre, pour plus de commodité, dans le dossier courant de votre éditeur

32. accessible ici : `http://jdreichert.fr/Enseignement/CPGE/Divers/lorem_ipsum.txt`

préféré.

Exercice 10 : Ouvrir le fichier de faux-texte et mettre son contenu dans une variable `contenu`. Déterminer le quarante-deuxième caractère de ce texte. Déterminer la taille du fichier.

Exercice 11 : Pour chaque lettre de l'alphabet français (hors lettres accentuées), déterminer si elle apparaît dans le texte. Combien de parcours du texte complet, qu'ils soient implicites ou explicites, sont effectués ? Si la réponse est 26, il s'agit de faire mieux. [33]

Exercice 12 : Écrire à la fin du fichier un message quelconque, et constater que l'ajout s'est bien fait.

Exercice 13 : Écrire une fonction qui prend en argument une chaîne de caractères et qui retourne la liste des occurrences de cette chaîne dans le fichier [34].

33. Si la réponse est 26 débuts de parcours, mais forcément vite interrompus au point que cela paraisse suffisant, il est temps de découvrir la complexité dans le pire des cas, avec un texte qui contiendrait un million de fois le même caractère puis un sous-ensemble des autres caractères.

34. Attention, il est possible qu'une occurrence commence alors que la précédente n'est pas finie... chercher un exemple.

TP 3 : Représentation des nombres - manipulations

Le comportement de l'interpréteur Python dépendra souvent dans ce TP de l'architecture de la machine ainsi que de la version de Python utilisée. Lorsque vous lancez Python, la version est précisée dans la première ligne de l'interpréteur, et elle est rappelée lorsque la fonction `help()` est utilisée sans argument.

Avec Edupython, la version 3 (précisément 3.2.5 à l'heure où ce sujet de TP est écrit) est utilisée, ce qui ne permet pas de constater des dépassements arithmétiques avec les entiers.

Diverses informations, notamment les seuils pour la virgule flottante, sont visibles en demandant `float_info` (dans le module `sys`, donc il faut au préalable écrire `from sys import float_info` par exemple). Quelques-unes de ces informations sont relativement faciles à comprendre.

Il est possible de faire de l'arithmétique dans les bases les plus usuelles en informatique (soit 2 et 16) en préfixant les nombres par `0b` (forçant le binaire) ou `0x` (forçant l'hexadécimal). Cependant, la réponse est donnée en base 10 par défaut. On peut cependant effectuer des conversions à l'aide des fonctions `bin` et `hex`, qui retournent des chaînes de caractères.

Attention : il n'est alors pas possible de faire des opérations sur les chaînes obtenues, précisément parce qu'elles ne sont plus des entiers. En fait, additionner `'0b10'` et `'0b1'` donnera `'0b100b1'`, soit la concaténation des chaînes.

La fonction `int` est interdite dans ce TP, car le programme qu'on demande dans l'exercice 2 revient exactement à réécrire `int(a,0)` [35].

Exercice 1 : Faire quelques calculs en binaire et en hexadécimal, par exemple en s'inspirant du TD 1.

Exercice 2 : Écrire une fonction qui transforme une chaîne de caractères obtenue à l'aide de `bin` ou de `hex` en le nombre qu'elle représente.

35. Dans les autres cas, le deuxième argument optionnel de la fonction `int` est une base entre 2 et 36 à partir de laquelle le premier argument, une chaîne de caractères tous alphanumériques, doit être convertie en décimal.

Pour l'exercice précédent, deux choses fondamentales sur les chaînes sont à utiliser : l'accès à un caractère donné, à l'aide de `s[i]`, où `s` est le nom de la chaîne et `i` l'indice en commençant à 0, et la longueur, qui s'obtient à l'aide de la fonction `len`.

Exercice 3 : Trouver trois valeurs `x`, `y` et `z` telles que le résultat de la commande Python `x + (y + z)` soit différent de celui de `(x + y) + z`.

Exercice 4 : Écrire une fonction qui retourne le discriminant d'un polynôme du second degré à partir des trois coefficients `a`, `b` et `c`. Écrire ensuite une fonction qui retourne l'ensemble des solutions réelles d'une équation polynomiale définie par les trois mêmes coefficients. Tester la fonction pour les fonctions $x^2 + 1.4x + 0.49$, $x^2 + 0.2x + 0.01$ et $x^2 + x + \frac{1}{4} + 10^{-20}$. Commenter.

L'intérêt de ne pas faire de tests d'égalité sur des réels se dessine. Malheureusement, au vu du troisième exemple, on ne peut pas non plus se permettre d'assimiler à zéro une valeur certes proche mais différente. Les logiciels de calcul formel tentent de pallier ce genre de problèmes que l'on rencontre lorsque l'on manipule des réels comme ici sans outils adaptés.

TP 4 : Gagner un peu d'indépendance

Dans ce TP, nous allons présenter des méthodes pour déboguer un programme, c'est-à-dire détecter les erreurs qu'on y a écrites, puis apprendre à se servir de la documentation fournie avec les fonctions et modules.

I - Déboguer un programme

1 Erreurs de syntaxe

Les premières erreurs que nous évoquerons ici sont les erreurs de syntaxe. Elles sont détectables à de multiples titres : elles provoquent des messages explicites renvoyés par Python, un bon éditeur les signale par du surlignage rouge... et un examen minutieux du code suffit de toute façon, avec l'expérience.

La plupart du temps, il s'agit de parenthésage incorrect, de l'oubli (voire de l'ajout indu) d'un double point ou d'une mauvaise indentation.

Exercice 1 : Le code ci-dessous comporte six erreurs. Relever les six messages d'erreur de Python en les corrigeant dans l'ordre[36].

```
def sextuple erreur(n
print "Espace indue dans le nom, double-point manquant,
parenthèse ouverte, indentation oubliée, délimiteur simple
dans la chaine et print sans parenthèses"
```

On notera que les erreurs sont annoncées comme `SyntaxError`, sauf l'erreur d'indentation qui a son « code » à part.

2 Comprendre les messages d'erreur de Python

Dans cette section, des programmes contenant diverses erreurs seront écrits, et il s'agit d'analyser la réaction mécontente du moteur Python.

Exercice 2 : Relever les messages d'erreur de Python en exécutant les programmes ci-dessous et les expliquer.

36. A priori c'est ligne par ligne et de gauche à droite, mais il faut suivre les positions indiquées par la console.

(Attention, le principe n'est pas de corriger les programmes qui n'ont de toute façon pas d'intérêt mais de comprendre les messages d'erreur et de supprimer les morceaux de code ensuite.)

```
for i in range(input()):
    print(i)

"deux" * "quatre"

s = "bonjour"
s[0] = "B"
s[7]
ss[6]

def carre(n):
    return n * n
carre()

for i in 42:
    print ("Il y en a qui ont essayé...")

len(42)

"J'ai".index("perdu")

x = 42
if x = 3:
    print("""En C ou en PHP, ça marcherait,
là c'est une autre erreur de syntaxe""")
    return "Et ceci provoque une autre erreur de syntaxe"
```

3 Baliser les programmes

Afin de comprendre le comportement de certains programmes, et entre autres remarquer pourquoi il ne termine pas ou ne fait pas ce qu'on attend de lui, faire un balisage peut être intéressant. C'est notamment le cas pour des programmes relativement longs, dont aucun exemple n'a à ce jour été écrit en TP.

L'idée est de choisir un endroit douteux dans une boucle et d'y forcer une impression d'un message explicite, permettant de constater le comportement de Python et les possibles déviations par rapport à ce qu'on attendait.

Ainsi, imaginons une simple boucle conditionnelle qui ne terminerait pas :

```
l = [1, 2, 3]
i = 0
while i < len(l):
    print(l[i])
    i =+ 1 # erreur ici : on voulait sans doute écrire += 1
```

Pour s'en rendre compte, il suffit d'ajouter le balisage en question et de voir que `i` vaut partout `1` :

```
l = [1, 2, 3]
i = 0
while i < len(l):
    print(l[i])
    i =+ 1
    print(i)
```

Un autre cas classique est de mettre `print("Coucou numéro XXX")` pour différents `XXX` dans tous les cas d'un test conditionnel afin de savoir si un des blocs n'est jamais visité.

Exercice 3 : S'engager à utiliser ce genre de méthodes à l'avenir afin de pouvoir corriger des bugs par soi-même.

4 Déboguer avec un IDE

Une autre façon de procéder est d'utiliser un débogueur intégré à l'éditeur. Edupython permet, comme les débogueurs standards, de placer des points d'arrêt dans du code.

Il s'agit de cliquer sur la zone à gauche d'une ligne afin de placer un tel point, et de lancer l'exécution non pas avec le bouton de lecture mais avec un bouton un peu à droite, comprenant le symbole de lecture et un trait vertical (la touche F4 est un

raccourci, sinon).

Dans ce cas, l'exécution pausera à chaque fois qu'elle arrivera à une ligne marquée comme point d'arrêt.

C'est là qu'intervient la partie inférieure d'Edupython. Plutôt que d'afficher la console, un onglet permet d'avoir un aperçu des variables, ce qui permettra d'avoir les avantages de la méthode présentée à la section précédente.

Malheureusement, la mise à jour de ces variables n'est apparemment pas visible quand le moteur est en pleine exécution, sinon un œil affuté pourrait bien plus profiter d'un tel aperçu de l'évolution des variables.

Pyzo possède également un débogueur, avec bien entendu des raccourcis propres. L'inspection des variables se fait d'une autre façon.

Exercice 4 : Déboguer des programmes au choix, qu'ils aient déjà été écrits dans des TP antérieurs ou qu'ils soient improvisés pour l'occasion.

II - *RTFM*

RTFM : Read The Fucking Manual. En clair, avant de poser des questions, il s'agit de chercher la réponse par soi-même.

Pour ce faire, Python dispose d'une fonction formidable, qui s'appelle `help`. Elle affiche la documentation de code (voir chapitre 3) associée à la fonction ou la méthode ou la classe ou le module à propos duquel on demande de l'aide.

Tester par exemple (plutôt dans la console, par principe) :

```
help(sum)
help([].append) # pour une méthode, il faut trouver un objet du bon type
help(42)
import random
help(random.randint)
help(random)
```

Exercice 5 : Étudier le manuel du module `random` en détail, et écrire avec cinq fonctions différentes de ce module (`random`, `randint`, `choice`, `sample` et `shuffle`) autant de fonctions qui permettent de faire un tirage sans remise de `n` éléments dans une liste `l`, avec `l` et `n` en argument.

TP 5 : Listes (et boucles)

Dans ce TP, il s'agit de réaliser des manipulations élémentaires de liste, notamment à l'aide de boucles.

À la fin de chaque exercice, discuter des différentes façons possibles de procéder et comparer les solutions de chacun.

Exercice 1 : Générer la liste de taille 7 contenant des zéros uniquement. Générer ensuite la liste de taille 7 contenant les lettres du mot « Bonjour ».

Exercice 2 : Générer la liste décroissante des entiers de 100 à 70.

Exercice 3 : Générer la liste croissante répétant deux fois chaque entier de 0 à 7.

Exercice 4 : Générer la liste croissante répétant $n+1$ fois chaque entier n de 0 à 7.

Exercice 5 : Générer la liste faisant successivement tous les comptes à rebours de n à 0 pour n entre 0 et 10.

Exercice 6 : Écrire un programme qui détermine le maximum d'une liste (on pourra admettre qu'elle ne contient que des nombres).

Exercice 7 : Écrire un programme qui détermine la moyenne des éléments d'une liste (même hypothèse). Une version idéale en Python 3 retourne un entier si d'une part tous les éléments sont entiers et d'autre part la moyenne tombe sur un nombre entier.

Exercice 8 : Générer une liste de six nombres entiers entre 1 et 6 (inclus) pris aléatoirement (fonction `randint` du module `random`). Déterminer si 6 est dans la liste.

Exercice 9 : Répéter l'expérience mille fois en comptant le nombre de fois où 6 était dans la liste.[37]

Et pour rester dans le même thème, un exercice sans listes :

Exercice 10 : (D'après un fait divers entendu autour de l'an 2000) Un homme lance

37. Exercice de mathématiques : donner la probabilité d'apparition d'un 6 dans la liste et comparer avec la fréquence obtenue.

sans arrêt un dé à six faces et s'est engagé à ne s'arrêter que s'il tombe six fois de suite sur un 6. Faire faire l'expérience par un programme et compter le nombre de lancers nécessaires. On ne demandera pas de calculer l'espérance du nombre de lancers.[38]

38. Exercice de mathématiques : si !

TP 6 : Algorithmes de base

Travail à faire pendant la séance

Dans ce TP, les algorithmes de base au programme, du moins ceux qui n'ont pas encore été traités dans les TP précédents, sont à écrire en Python.

Exercice 1 : Écrire une fonction qui prend en entrée une liste et une valeur quelconque et qui retourne le premier indice de la liste où la valeur se trouve.

Exercice 2 : Écrire une fonction qui prend en entrée une liste et une valeur quelconque et qui retourne le nombre de fois où la valeur figure dans la liste (on appelle cela le nombre d'occurrences).

Exercice 3 : Écrire une fonction qui prend en entrée une liste et une valeur quelconque et qui retourne la liste des indices de la liste où la valeur se trouve.

Exercice 4 : Écrire une fonction qui prend en entrée deux listes et qui retourne la liste des indices de la première liste où tous les éléments de la deuxième liste commencent à apparaître dans l'ordre et à la suite. Ceci correspond à la recherche d'un motif dans une chaîne de caractères, qu'on adapte ici.

Exercice 5 : Écrire une fonction retournant la variance d'une liste de nombres.

Exercice 6 : Écrire une fonction de recherche dichotomique d'un élément dans une liste contenant des nombres dans l'ordre croissant.

Exercice 7 : Écrire une fonction de recherche d'un zéro d'une fonction continue, puis une fonction de recherche de zéro d'une fonction monotone.

Exercices pour réfléchir après la séance

Prouver la correction et calculer la complexité de chaque fonction écrite dans ce TP. Si une boucle conditionnelle intervient, il faut d'abord prouver la terminaison.

TP 7 : Introduction à numpy

Dans ce TP, nous allons présenter des fonctions utiles, notamment pour l'algèbre linéaire, issues de la bibliothèque `numpy`.

Travail à faire pendant la séance

1 Le type `array`

Exercice 1 : Créer des tableaux de différentes dimensions et différentes formes. Constater à quel point l'impression dans la console est travaillée.

Exercice 2 : Consulter la documentation des fonctions `append`, `insert` et `delete` (avec `help` ou grâce à l'autocomplétion d'Edupython). Quelle est la différence majeure entre leur effet et leur version connue sur les listes ? [39]

Exercice 3 : Constater le résultat des instructions suivantes. (Attention, c'est normal que certaines lignes déclenchent des erreurs ! À part les deux premières lignes, mieux vaut écrire dans la console...)

```
a = numpy.array([[1, 2, 3],[4, 5, 6]])
aa = numpy.array([[[1, 2], [3, 4], [5, 6]],[[1, 2], [3, 4], [5, 6]]])

numpy.append(a, 7)
numpy.append(a, 7, axis=0)
numpy.append(a, 7, axis=2)
numpy.append(a, [7, 8, 9], axis=0)
numpy.append(a, [[7, 8, 9]], axis=0)
numpy.append(a, [[7, 8, 9]], axis=1)
numpy.append(a, [[7], [8]], axis=1)

numpy.append(aa, [[[7, 8], [7, 8], [7, 8]]], axis=0)
numpy.append(aa, [[[7, 8], [7, 8], [7, 8]]], axis=1)
```

39. La méthode `append`, la méthode `insert` et la méthode `pop` ; il n'existe pas de méthode `delete`, mais `remove` retire la première occurrence d'un élément précisé, avec une erreur s'il n'y figure pas.

Exercice 4 : Deviner le comportement de Python avec une syntaxe reprenant la notion de *slice* sur les instructions ci-après, puis constater le résultat. Essayer aussi les fonctions d'insertion et de suppression sans le troisième argument (en pratique, le tableau est aplati avant l'opération, ce qui rend le comportement intuitif).

Une petite information sur le *slice* : on rappelle que pour un objet supportant cette opération (la majorité des indexables) `l`, l'objet `l[deb:fin:pas]` est de même type que `l` et contient ses éléments aux positions données par `range(deb, fin, pas)`. Avec un tableau `t`, on écrit `t[numpy.s_[deb:fin:pas]]`. Les opérations semblent interchangeables, mais il est préférable d'utiliser celles qui correspondent au type de données.

En fait, l'objet `numpy.s_` contient tous les entiers naturels, on s'en sert normalement uniquement en en considérant des tranches finies.

```
a = numpy.array([[0, 1, 2, 3, 4, 5], [6, 7, 8, 9, 10, 11]])

numpy.insert(a, 1, 42, axis=0)
numpy.insert(a, 1, [42], axis=0)
numpy.insert(a, 1, [42, 42, 42, 42, 42, 42], axis=0)
numpy.insert(a, 1, [42, 42], axis=0)
numpy.insert(a, 1, [42], axis=1)
numpy.insert(a, [1, 2], [42], axis=1)
numpy.insert(a, numpy.s_[2:5], 42, axis=1)

numpy.delete(a, 1, axis=0)
numpy.delete(a, [1, 2], axis=1)
numpy.delete(a, numpy.s_[2:5], axis=1)
```

Exercice 5 : Écrire une fonction simulant `arange` en se servant de `linspace` et vice-versa.

Exercice 6 : Lire la documentation de la fonction `sort` et trier des tableaux de dimension 3 en utilisant les paramètres optionnels.

Exercice 7 : Écrire une fonction qui prend en entrée un tableau et qui renvoie sa version aplatie.

Au passage, constater l'effet des codes suivants (on réutilisera le tableau `a` d'avant) :

```
a[0], a[1] = a[1], a[0] # boum !
# ici réinitialiser a
buff = a[0]
a[0] = a[1]
a[1] = buff # boum !
# réinitialiser de nouveau
a[0], a[1] = a[1].copy(), a[0].copy() # ouf !
```

Exercices pour réfléchir après la séance

2 Un peu d'algèbre linéaire

Exercice 8 : Créer quelques matrices et tester les possibilités de l'utilisation de chaînes de caractères comme premier argument.

Exercice 9 : Utiliser les fonctions du cours opérant sur les matrices sur des exemples de matrices, carrées ou non.

Exercice 10 : Créer une fonction retournant une matrice `M` à `n` lignes et `m` colonnes telle que `M[i, j]` vaille `|j-i|`.

3 Les polynômes avec `numpy`

Exercice 11 : Résoudre des problèmes sur les polynômes (par exemple issus de colles de mathématiques) à l'aide de la structure de polynôme fournie avec `numpy` et les fonctions vues en cours.

Exercice 12 : Écrire deux fonctions retournant le PGCD de deux polynômes obtenu de deux façons différentes.

4 TP étendu pour la deuxième année

Exercice 13 : Consulter la documentation du sous-module `linalg` de `numpy` et utiliser des fonctions issues de la théorie spectrale.

Exercice 14 : Réécrire la fonction calculant le déterminant d'une matrice carrée (avec une certaine tolérance pour le type de l'argument) de manière récursive ou itérative.

TP 7bis : Introduction à Scilab

Attention : ce TP a été amicalement fourni par G. Thouroude, je n'en suis pas l'auteur.

Le logiciel Scilab est un logiciel spécialisé dans le calcul numérique. Conformément au programme, la partie calcul numérique (correspondant au second semestre) peut être mise en œuvre aussi bien avec Python qu'avec Scilab. On donne ici une présentation rapide de Scilab.

Exercice 1 : Entrer les commandes ci-dessous dans la console et observer les résultats.

```
2+%pi
(1+%e)/2
%eps
1+%eps
%i^2
(1-%i)^2
exp(3)
%e^3
cos(%pi)
sin(%pi)
2/3
```

Scilab accepte aussi la notation ** au lieu de `^` pour calculer une puissance. Par contre on verra qu'il existe une notation .^ dans Scilab (mais pas la notation .**).

Exercice 2 : Entrer les commandes ci-dessous et observer les résultats obtenus.

```
a=2
a=a+1
A=2*%pi
a+A
ans+a
a=1.2e-3
a=rand()
```

On peut supprimer la variable a en écrivant `clear a`. On supprime toutes les variables en écrivant simplement `clear`.

Exercice 3 : En réalité Scilab est codé pour travailler avec des vecteurs. Voici comment créer des vecteurs. On remarquera que la virgule ou un espace permet de séparer deux coefficients sur une même ligne, qu'un point virgule permet de changer de ligne, et que la transposée de x est x'. **Attention ici l'indexation démarre à 1.**

```
x=[1 2 3 4]
x=[1,2,3,4]
y=[1;2;3;4]
x'
y'
x(1)
y(1)
x(2)=3
y(2)=3
x(2:3)
z=[1:2:9]
t(1)=1
t(20)=1
t
length(x)
length(t)
```

Exercice 4 : Et maintenant des matrices. Observer les résultats obtenus.

```
eye(4,4)
ones(4,2)
zeros(4,2)
A=[1,2,3;4,5,6;7,8,9]
A(2,3)
A(2,:)
A(:,3)
A(2:3,2:3)
A'
zeros(A)
ones(A)
eye(A)
B=rand(A)
size(A)
```

Exercice 5 : On va maintenant apprendre à faire du calcul matriciel. Comprendre les différentes opérations de multiplication et de puissance à l'aide des lignes de commandes suivantes :

```
x=[1;1]
A=[1,2;3,4]
A*x
A*A
A.*A
A^2
A.^2
eye(2,2)./A
A+1
x+1
x.^2
inv(A)
A^(-1)
```

Exercice 6 : À l'aide de Scilab, résoudre le système :

$$\begin{cases} x-y+z-t=1 \\ x+y-z-t=-1 \\ x+y+z-t=0 \\ x-y-z+t=2 \end{cases}$$

Exercice 7 : Nous allons maintenant apprendre à utiliser SciNotes et créer des fonctions. Ouvrir SciNotes (menu applications) et entrer le programme ci-dessous puis l'exécuter. Normalement un joli graphique apparaitra.

(Attention, le programme consiste en la définition d'une fonction et son appel à la page suivante.)

```
function y=u(n)
    y=1
    for k=1:n
        y=(y+1)^0.5
    end
endfunction
```

```
clf()
plot(0:10,feval(0:10,u),"bo")
title("Suite u")
xlabel("n")
ylabel("u")
```

Pour créer une fonction en Scilab on écrit :

```
function sortie=nom(arguments)
    ...
endfunction
```

Pour faire une boucle for on écrit :

```
for variable debut:fin
    ...
end
```

Pour faire une boucle while on écrit :

```
while condition
    ...
end
```

Pour faire un test conditionnel on écrit :

```
if condition then
    ...
elseif condition then
    ...
else
    ...
end
```

TP 8 : Autour du pivot de Gauss

Travail à faire pendant la séance

Exercice 1 : Écrire des fonctions pour les quatre opérations arithmétiques de base $(+, -, \times, \div)$ sur des rationnels écrits sous forme de couples `(a, b)`, où `a` est un entier relatif et `b` est un entier strictement positif premier avec `a`. Le résultat de chaque opération arithmétique devra respecter les mêmes propriétés et on fixera l'écriture d'un entier `n` comme `(n, 1)`.

Attention : Temps recommandé : 5 minutes pour se faire expliquer le principe, 15 minutes pour écrire l'ensemble, si possible avec l'algorithme d'Euclide dans la foulée.

Exercice 2 : Écrire l'algorithme du pivot de Gauss pour une matrice (type au choix) dont tous les éléments sont des rationnels déjà écrits sous la forme précédente. Une conversion préalable est nécessaire pour pouvoir commencer avec une liste de listes dont les éléments sont des entiers ou des couples d'entiers, et on peut envisager de déclencher un message d'erreur (a priori au niveau de la conversion) si un élément de la matrice n'est pas un entier ou un couple d'entiers (si le deuxième entier du couple est un `0`, l'erreur viendra toute seule ultérieurement).

Attention, pour l'exercice précédent, on peut très bien résoudre une équation de type `AX = B`, où des éléments de `B` sont des flottants, puisqu'on ne fait jamais de comparaison impliquant `B`, ce qui serait précisément la source des problèmes.

Ainsi, les opérations arithmétiques impliquant un couple d'entiers représentant une fraction d'une part et un flottant (ou un entier) d'autre part sont également à écrire, pour que le code soit complet.

Exercice 3 : Écrire aussi l'algorithme du pivot partiel dans les mêmes conditions. Si on a pensé à modulariser, il suffit d'écrire une fonction de plus et de changer l'appel.

Exercice 4 : Recopier les programmes correspondant au pivot de Gauss et au pivot partiel, puis comparer les solutions sur des exemples simples entre le cours, le TP et la fonction `numpy.linalg.solve`.

Exercice 5 : En particulier, tester les systèmes ci-après et voir l'intérêt des fonctions

écrites ici. [40]

$$\begin{cases} x + \frac{1}{4}y + z = 0 \\ x + \frac{1}{3}y + 2z = 0 \\ y + 12z = 1 \end{cases} \qquad \begin{cases} x + (10^{15}+1)y + z = 1 \\ x + (1+10^{-15})y + 2z = 0 \\ -10^{15}y + z = 0 \end{cases}$$

Pour faciliter le travail, voici une représentation en listes de listes de ces systèmes.

```
mat1 = [[1, (1, 4), 1], [1, (1, 3), 2], [0, 1, 12]]
mat1flott = [[1, 1/4, 1], [1, 1/3, 2], [0, 1, 12]]
sol1 = [0, 0, 1]

mat2 = [[1, 10**15+1, 1], [1, (1+10**15, 10**15), 2], [0, -10**15, 1]]
mat2flott = [[1, 10**15+1, 1], [1, 1+10**(-15), 2], [0, -10**15, 1]]
sol2 = [1, 0, 0]
```

Exercices pour réfléchir après la séance

Au niveau de l'exercice 4, on peut envisager de comparer aussi les temps d'exécution. Pour savoir le temps d'exécution d'une fonction dont on connaît le nombre d'arguments, une possibilité est :

```
from time import time
def temps(f, arg1, arg2): # on suppose qu'il y a deux arguments ici
  t = time()
  f(arg1, arg2)
  return time() - t
```

Au passage, quel problème soulèverait le fait d'écrire une fonction `temps` utilisée avec un seul argument `expr` afin d'appeler la fonction en mettant en argument l'expression `f(arg1, arg2)` et de ne pas avoir à fixer un nombre d'arguments pour `f` ?

Exercice 6 : Écrire aussi l'algorithme du pivot total.

40. Les exemples sont issus de l'excellent livre de Wack et al., où la dernière ligne n'a pas de signe moins devant $10^{15}y$, ce qui fait que la matrice est déclarée comme de rang 2 par la fonction `rank` du module `numpy`, mais Python peut tout de même trouver une solution.

Pour rappel, le pivot total consiste à faire des échanges de lignes **et** de colonnes afin d'utiliser le plus grand pivot en valeur absolue sur l'ensemble des lignes et colonnes encore considérées. Les échanges sur les colonnes nécessitent une répercussion sur le second membre final.

Il est possible de simplifier les choses en ne faisant pas d'échange de colonnes, mais alors il s'agit de mémoriser les colonnes déjà étudiées.

Exercice 7 : Écrire aussi un algorithme s'appuyant le principe de n'importe laquelle des trois versions du pivot pour calculer le rang d'une matrice.

On pourra se demander comment faire pour éviter de copier-coller l'algorithme du pivot pour commencer...

TP 9 : Graphismes

Dans ce TP, nous travaillerons avec le module graphique `matplotlib`, en fait avec son sous-module `pyplot`.

Pour éviter d'avoir des noms de fonctions à rallonge, on commencera par l'importer, conventionnellement par `import matplotlib.pyplot as plt`. Les fonctions présentées ici sont toutes issues du sous-module.

Sans entrer dans le détail sur la façon dont des images sont représentées pour un rendu à l'écran (y compris par exemple lorsqu'on lit ce même texte), on signalera que le lecteur curieux pourra se renseigner sur les mots-clés "blit" et "double buffering".

En fait, le but ici n'est pas d'afficher des images en général mais de tracer des graphes de fonctions et des courbes dans un plan pour commencer.

1 La base

Suivant le langage, la gestion d'une fenêtre graphique change parfois grandement. Dans notre cas, le contenu va être préparé (« Que va-t-on représenter ? ») et paramétré (« Comment va-t-on le représenter ? ») avant d'être affiché ou rafraîchi dans une nouvelle fenêtre.

Commençons par la fonction `plot`, dont l'utilisation standard est de préparer un ensemble de lignes ou un nuage de points (suivant un argument de paramétrage ; par défaut on trace des lignes bleues) pour toutes les ordonnées dans l'argument (indexable de taille `N`) avec les abscisses valant par défaut les entiers de `0` à `N-1`.

Si on souhaite modifier les abscisses, il faut préciser un autre argument **avant**, qui sera le tableau des abscisses pour une correspondance élément par élément, donc Python déclenche une erreur si les tailles ne correspondent pas.

Attention : ne pas donner une liste de couples comme seul argument, car cela serait interprété comme deux figures à représenter (bien entendu, si c'est effectivement ce qu'on souhaite, il n'y a pas de souci).

Une fois la figure préparée et stockée dans une variable, la fonction `show`, prenant cette figure en argument (ou sans argument si on veut représenter la dernière figure

sans crainte d'ambigüité), la représente. En cas de modification ultérieure, la fonction `draw` rafraîchit la figure courante (donc sans argument).

Pour fermer une fenêtre graphique, cliquer sur la croix habituelle est une possibilité, mais il est préférable d'utiliser la fonction `close` avec la figure en argument [41].

Une remarque : certaines fonctions peuvent être appelées sur un tableau (parfois sur d'autres séquences) et retournent alors un tableau composé des images de tous les éléments par la fonction ; d'autres ne le permettent pas.

Par exemple, la fonction `math.sin` ne le permet pas, alors que la fonction `numpy.sin` le permet. La fonction `abs` le permet naturellement, mais dans le doute, on peut avoir recours à la fonction `map` (prenant en argument une fonction et un itérable) ou, si on travaille déjà avec `numpy`, utiliser la fonction `vectorize` de cette bibliothèque : `numpy.vectorize(f)` retourne une fonction appelable sur une séquence et retournant le tableau regroupant les images par la fonction des éléments de la séquence (même si un seul argument est donné, ce qui diffère du comportement des fonctions comme `numpy.sin`).

2 Le paramétrage

On notera que la fenêtre graphique dispose de quelques outils pour naviguer dans la figure ou zoomer de différentes façons. Bien évidemment, ceci peut aussi se faire à l'aide de fonctions, qui font gagner en précision ce qu'elles font perdre en simplicité. Les axes sont traités à l'aide de la fonction `axis`, qui peut s'appeler de différentes façons, dont on retiendra notamment `plt.axis('off')` pour faire disparaître le repère et `plt.axis([xmin, xmax, ymin, ymax])` pour redéfinir les bornes en abscisse et en ordonnée de la partie représentée de la figure (naviguer fait réapparaître le reste). La grille en arrière-plan peut aussi être modifiée, par exemple on la fait disparaître en écrivant `plt.grid(None)`.

3 Fonctions avancées

Au lieu (ou en plus) d'afficher une figure, on peut la sauvegarder en tant que fichier sous divers formats. On peut se servir pour cela de la fonction `savefig`, prenant

41. Pour information, en Caml, fermer manuellement une fenêtre graphique déclenche une erreur d'entrée/sortie.

en argument principal le nom du fichier (sous forme d'un chemin si on souhaite l'enregistrer dans un autre dossier que le dossier courant).

Une image est gérée comme un tableau `numpy`. Python est prévu pour pouvoir principalement manipuler le format PNG, ce qui impose (sauf bricolage) que les images ouvertes et converties soit sous cedit format. On se sert pour cela de la fonction `imread` (la sauvegarde se fait à l'aide de la fonction `imsave` qui précise d'abord le nom du fichier). La fonction `imshow` est alors utilisée au lieu de `plot` pour afficher une image sur les axes. Attention à la palette de couleurs sélectionnée, modifiable par autant de fonctions que de palettes, par exemple `autumn`, `gray`, etc. Pour ceux qui ne disposent pas d'une image sous la main, le sous-module `misc` de la bibliothèque `scipy` dispose des fonctions `face` et `ascent`, qui produisent des images sur lesquelles on peut appeler directement `imshow`.

Enfin, si représenter des points est insuffisant, notamment lorsqu'on veut dessiner des surfaces, on dispose de la fonction `fill` qui remplit un polygone dont le premier argument est un indexable qui liste les abscisses des sommets du polygone et le deuxième argument est un indexable qui en liste les ordonnées.

Un exemple d'utilisation des fonctions graphiques :

```
import math, matplotlib.pyplot as plt

def coordonnees(a, k, t):
    return (a*(math.cos(t) + 2*k*math.cos(t/2)), a*math.sin(t))

liste_abscisses = [coordonnees(3, 1, t/10000)[0] for t in range(200000)]
liste_ordonnees = [coordonnees(3, 1, t/10000)[1] for t in range(200000)]
plt.show(plt.plot(liste_abscisses, liste_ordonnees))
```

Pour résumer l'utilisation majoritaire du module : créer une liste d'abscisses, éventuellement en préparant une séquence à parcourir, créer une liste d'ordonnées de même taille, utiliser plot et show.

Exercice 1 : Écrire une fonction prenant en paramètres trois flottants x, y et r et qui trace le cercle de centre (x, y) et de rayon r, en dessinant des points qui le constituent en nombre (par exemple mille par radian).

Exercice 2 : Écrire une fonction prenant en paramètres une fonction f et deux flottants a et b et qui trace la courbe de la fonction f entre les abscisses a et b, là encore à raison d'un certain nombre de points par unité d'abscisse.

Exercice 3 : « S'il vous plaît, dessine-moi un mouton ! »

Exercice 4 : Au passage, écrire une fonction agissant comme `vectorize`.

TP 10 : Autour de la méthode de Newton

Ce TP est une extension de la section 3 du chapitre 3, pour se pencher sur les mises en œuvre alternatives de la méthode de Newton, ainsi qu'une première application du TP 9 pour faire quelques graphismes simples.

Travail à faire pendant la séance

Exercice 1 : Écrire une version de la méthode de Newton qui prend en premier argument un polynôme `numpy`, de sorte que le polynôme dérivé n'ait pas à être fourni car on le calcule soi-même.

Pour ce premier exercice, un guide de survie pour les polynômes (cf. aussi le cours) :

```
P1 = numpy.poly1d([1, -5, 6]) # X**2-5X+6
P2 = numpy.poly1d([1, 1], True) # X**2-2X+1, True -> liste des racines
P1[0] # 6, coefficient constant, donc l'ordre est inversé
P1.r # tableau des racines, attention aux approximations
P1.c # tableau des coefficients, de nouveau dans l'ordre de départ
len(P1) # ATTENTION, c'est le degré, donc P1[len(P1)] existe EXCEPTIONNELLEMENT
```

Exercice 2 : Tester dans la fonction précédente un cas où le polynôme dérivé s'annule pour la valeur de départ, et constater le message plutôt obscur. Observer le même message en créant une forme indéterminée avec des polynômes.

Exercice 3 : Écrire une version de la méthode de Newton qui représente la fonction utilisée et les tangentes successives à chaque itération sur une même figure.

Exercices pour réfléchir après la séance

Exercice 4 : Écrire un programme appliquant la méthode de la sécante.

Exercice 5 : Écrire une version de la méthode de Newton qui calcule une approximation de la dérivée.

TP 11 : Autour de la méthode d'Euler

Ce TP est une extension de la section 4 du chapitre 3, pour se pencher sur les mises en œuvre alternatives de la méthode d'Euler, ainsi qu'une deuxième application du TP 9 pour faire quelques graphismes moins simples.

Travail à faire avant la séance

Exercice 1 : Écrire (si possible sans s'aider du cours) une fonction implémentant la méthode d'Euler, compatible avec les vecteurs.

Exercice 2 : Tester ce programme pour l'équation différentielle $y'' = -y$ avec comme condition initiale $y'(0) = 1$ et $y(0) = 1$. D'où l'intérêt de la compatibilité avec les vecteurs.

À titre indicatif, la solution est $t \mapsto \cos(t) + \sin(t)$.

Travail à faire pendant la séance

Exercice 3 : Écrire une version de la méthode d'Euler (vectorielle, tant qu'à faire) qui trace les lignes entre les différents points calculés, ainsi que la solution effective (quand on la connaît et qu'on la fournit) sur une même figure.

Exercice 4 : Modifier le code précédent pour faire apparaître plusieurs schémas d'Euler avec des pas de plus en plus petits, pour pouvoir constater le résultat.

Exercice 5 : Application à la physique #1 : la chute verticale avec frottements turbulents. Représenter la vitesse en fonction du temps, lorsque celle-ci est solution de l'équation différentielle $v' = g - kv^2$.

Exercice 6 : Application à la physique #2 : le pendule amorti. Représenter l'angle en fonction du temps, lorsque celui-ci est solution de l'équation différentielle

$$\theta'' = -(\frac{g}{l})^2 \sin(\theta) - k\theta'$$

pour des valeurs arbitraires de l et k.

Comme on ne demande pas une solution mathématique, il n'est pas nécessaire d'utiliser l'approximation de $\sin(\theta)$ par θ pour les petits angles. [42]

42. Évidemment, on perd de la précision tout de même, et sans doute plus encore qu'en récupérant la solution de l'approximation quand le pas est trop grand.

Exercice 7 : Application à la chimie #1 : cinétique chimique [43]. Représenter la concentration d'un élément A dans le cas d'une réaction réversible (A donne B et B donne A), lorsque les concentrations de A et B sont solutions du système différentiel

$$\left\{\begin{array}{rcrcl} c_A' &=& k_B c_B &-& k_A c_A \\ c_B' &=& k_A c_A &-& k_B c_B \end{array}\right.$$

Représenter la concentration des éléments A, B et C dans le cas de réactions successives (A donne B et B donne C), lorsque les concentrations de A, B et C sont solutions du système différentiel

$$\left\{\begin{array}{rcrcl} c_A' &=& &-& k_A c_A \\ c_B' &=& k_A c_A &-& k_B c_B \\ c_C' &=& k_B c_B & & \end{array}\right.$$

Représenter la concentration des éléments A, B et C dans le cas de réactions successives (A donne B et B donne C), lorsque les concentrations de A, B et C sont solutions du système différentiel (d'ordre 2, à présent)

$$\left\{\begin{array}{rcrcl} c_A' &=& &-& k_A c_A^2 \\ c_B' &=& k_A c_A^2 &-& k_B c_B^2 \\ c_C' &=& k_B c_B^2 & & \end{array}\right.$$

Représenter la concentration des éléments A, B et C dans le cas d'une réaction inspirée de la réaction oscillante de Belousov-Zhabotinsky, lorsque les concentrations de A, B et C sont solutions du système différentiel (d'ordre 2 aussi, avec comme paramètres suggérés $k_1 = 0.05, k_2 = 0.8, k_5 = 1.58, k_6 = 0.2, k_7 = 0.4, \alpha = 1$ et β entre 0.5 et 2.4 ajustable [44]).

$$\left\{\begin{array}{rcl} c_A' &=& k_1\alpha c_B + k_5\alpha c_A - k_2 c_A c_B - 2k_6 c_A^2 \\ c_B' &=& -k_1\alpha c_B - k_2 c_A c_B + k_7\beta c_C \\ c_C' &=& k_5\alpha c_A - k_7 c_C \end{array}\right.$$

Représenter la concentration des éléments A, B et C dans le cas de réactions parallèles (A donne B et A donne C), lorsque les concentrations de A, B et C sont solutions du système différentiel

$$\left\{\begin{array}{rcl} c_A' &=& -(k_B + k_C)c_A \\ c_B' &=& k_B c_A \\ c_C' &=& k_C c_A \end{array}\right.$$

43. Remerciements spéciaux à mes collègues chimistes !

44. Source : `https://www.lespritsorcier.org/blogs-membres/reaction-oscillante-de-belousov-zhabotinsky/`, merci à S. B. pour le lien.

Les systèmes différentiels peuvent intervenir dans d'autres domaines, par exemple en dynamique des populations (modèles proie-prédateur, pour lesquels on peut voir une analogie avec la cinétique chimique), mais aussi et surtout en mécanique avancée, par exemple les nombreuses fois où j'ai été appelé à l'aide pour un TIPE ces dernières années !

Pour les plus motivés, il existe des méthodes dites à pas adaptatifs (plus la valeur absolue de la fonction décrivant l'équation différentielle est faible au point d'évaluation, plus on se permet de faire un grand pas), ce qui est utilisé dans la fonction `odeint`, et des méthodes dites à pas multiple (on utilise plus de valeurs précédentes que la dernière, ce qui est le cas de la méthode d'Adams par exemple). En particulier, il existe d'autres façons de procéder qu'en faisant des schémas explicites, on a évoqué les schémas implicites dans le cours, d'autres méthodes sont plus puissantes notamment la méthode de Runge-Kutta d'ordre 4. Il alors est intéressant de lancer les deux méthodes en parallèle et comparer les performances en termes de précision (en zoomant sur les figures) et de temps.

Exercices pour réfléchir après la séance

À la suite d'une discussion avec un éminent collègue de physique, il apparaît qu'en TIPE, non seulement certains étudiants font face à des équations différentielles coriaces, mais en plus les paramètres de l'équation en question ne sont pas toujours connus, et il s'agit de tenter de les déterminer sans trop d'erreur à partir de résultats expérimentaux.

Tous ceux qui ambitionnent de faire un TIPE où ils pourraient se retrouver dans cette situation sont invités à s'informer d'ores et déjà sur les deux méthodes (de difficulté croissante) de descente de gradient et de recuit simulé.

TP 12 : Bases de données

Pour voir une mise en œuvre pratique des bases de données, rien de tel qu'un exemple concret et véritablement utilisé : la gestion des tournois sur le site de la fédération française de skat[45]. Les entrées correspondent à la saison 2016/2017 (sauf le dernier tournoi, ayant lieu après l'écriture du sujet...).

Ce TP se fait en totalité sur la plateforme phpMyAdmin mise à disposition sur `http://phpmyadmin.online.net`, l'identifiant est **db94381** et le mot de passe est communiqué au cours de la séance.

La base de données constituée au moment de préparer le TP a été exportée et elle peut être récupérée pour une utilisation individuelle sur un serveur personnel[46]. La structure de la base de données est relativement complexe et ne sera donc pas détaillée dans le présent sujet.

Travail à faire pendant la séance

Faire tous les exercices faciles (* et **).

Exercice 1 (*) : Naviguer sur la plateforme phpMyAdmin afin de la prendre en main.

Exercice 2 (*) : Écrire et exécuter une requête qui permet de récupérer le nombre de joueurs s'étant inscrits au tournoi d'identifiant `4`.

Exercice 3 (**) : Écrire et exécuter une requête qui permet de récupérer le nombre de joueurs s'étant inscrits au moins une fois à un tournoi.

Exercice 4 (*) : Écrire et exécuter une requête qui permet de déterminer le nombre de fois où un joueur était à la même table aux deux séries.

Exercice 5 (**) : Écrire et exécuter une requête qui permet de déterminer quel joueur licencié a le plus grand nombre de points sur la saison.

Exercice 6 (**) : Écrire et exécuter une requête qui permet de déterminer à quel tournoi le nombre d'inscrits était le plus grand et le nombre de ces inscrits.

45. Pour tout dire, les requêtes effectuées sur le site sont souvent moins complexes, dans la mesure où les calculs et agrégations peuvent tout aussi bien être faites à l'aide du langage de scripts qui récupère le contenu de la base de données en exécutant les requêtes SQL.

46. `http://jdreichert.fr/Enseignement/CPGE/IPTSUP/bdd_tp_12.sql`

Note : Des joueurs préinscrits ne se présentant pas occasionnent un trou dans la liste des inscrits. On ne peut donc pas totalement se fier au maximum du champ concerné.

Exercice 7 (**) : Écrire et exécuter une requête qui permet de déterminer à quel tournoi le meilleur score était le plus haut et ce meilleur score.

Exercice 8 (****) : Même exercice mais cette fois-ci on cherche le meilleur score pour des tandems. Le score d'un tandem est la somme des scores des deux joueurs qui le composent.

Exercice 9 (*****) : Écrire et exécuter une requête qui permet de récupérer la liste des joueurs ayant été au moins une fois dans l'équipe de leur club, ainsi que le nombre de fois où ils y ont été, le tout par ordre décroissant de ce nombre.

Note : Ensuite, on pourra décider de l'ordonner par numéro de licence croissant afin d'avoir une présentation club par club.

Exercice 10 (***) : Écrire et exécuter une requête qui permet de déterminer pour chaque tournoi le club ayant le plus d'inscrits.

Note : Pour préparer les placements de secours de la saison suivante, un script a même été réalisé afin de savoir le nombre de participants par club affilié et par tournoi.

Exercice 11 (*) : Écrire et exécuter une requête qui permet de déterminer le nombre de fois où un joueur était à une place de même parité aux deux séries.

Idée : L'algorithme de positionnement a été écrit de sorte qu'un minimum de joueurs soit aux deux séries à une place impaire, où ils doivent noter les scores (double marquage). Les tables incomplètes et les départs après la première série font qu'il n'est pas toujours possible de mettre chaque joueur une et une seule fois à une place impaire.

Exercice 12 (**) : Écrire et exécuter une requête qui permet de récupérer les non licenciés avec nombre de participations et total sur la saison, classés par total décroissant sur la saison.

Idée : Les non-licenciés ne figurent pas dans le classement annuel. On pourrait ici constater que certains seraient très bien placés et qualifiables pour la finale.

Exercice 13 (***) : Écrire et exécuter une requête qui permet de déterminer le nombre de fois que deux joueurs, en les précisant, ont été à la même table, si ce nombre est au moins deux. Adapter cette requête pour déterminer le nombre de paires de joueurs qui ont déjà été au moins deux fois à la même table.

Idée : Des joueurs s'interrogent sur le générateur aléatoire en raison du grand nombre de fois où ils ont joué ensemble.[47] Un raffinement de cette dernière requête est de déterminer le nombre de paires de joueurs qui ont été n fois à la même table pour chaque n (parmi ceux où la réponse n'est pas 0).

Exercice 14 (***) : Écrire et exécuter une requête qui permet de récupérer les noms de tandems apparaissant le plus souvent avec le nombre d'occurrences associées. Il est cependant important que les tandems correspondent aux mêmes joueurs.

Idée : lors de l'inscription d'un tandem, le formulaire contient des tandems classiques qu'on peut inscrire en un clic, et le but est de déterminer ces tandems. On pourra demander en choix alternatif de récupérer les tandems avec les noms associés et les numéros de tournois dans l'ordre alphabétique des noms de tandems.

Exercice 15 (***) : Écrire et exécuter une requête qui permet de récupérer les noms de tandems associés à un utilisateur pour tous les tournois et tous les utilisateurs.

Idée : à la fin d'un tournoi, un mail est envoyé aux utilisateurs - table non reportée sur cette base de données - ayant souscrit à cette option, avec les résultats du tournoi ; le nom du joueur apparaît alors en rouge et le but est de mettre aussi en rouge ses tandems. La requête demandée ici est une généralisation, mais une simple clause WHERE suffit pour obtenir ce qui était souhaité.

Exercices pour réfléchir après la séance

Faire les autres exercices.

47. Trouver un moyen de leur expliquer rapidement le paradoxe des anniversaires est une tâche ardue...