CHAPITRE 1

Calcul symbolique *versus* calcul numérique

On présente dans ce chapitre les deux facettes autour desquelles s'articule l'ouvrage, à savoir le calcul symbolique d'une part et le calcul scientifique (ou numérique) d'autre part. Dans le cadre du calcul symbolique, les objets sont des entités formelles, à savoir des symboles codant des variables (`A,B,...,X,Y,t,...`), des nombres entiers ou rationnels ($\mathbb{Z}$ ou $\mathbb{Q}$), des polynômes à coefficients entiers ou rationnels, des listes ou suites de tels objets, des ensembles, voire des êtres relevant de l'algèbre formelle, par exemple les anneaux $\mathbb{Z}[T]$, $\mathbb{Z}[T_1, ..., T_p]$, $\mathbb{Q}[T]$, $\mathbb{Q}[T_1, ..., T_p]$, les corps $\mathbb{Q}(T)$, $\mathbb{Q}(T_1, ..., T_p)$, des groupes finis, etc. Dans le cadre du calcul scientifique au contraire, les objets de base seront des nombres (réels ou complexes) codés (donc nécessairement approchés) suivant le principe de la virgule flottante ou des tableaux de tels nombres (dits matrices lorsqu'ils sont bi-dimensionnels). Le calcul scientifique (ou numérique) se trouve être un calcul « avec pertes » (entaché d'un calcul d'erreur qu'il conviendra de savoir contrôler), au contraire du calcul symbolique qui, lui, de par le fait qu'il est fondé sur la manipulation de symboles formels, réalise un calcul « sans pertes ». Cette ubiquité sera omniprésente dans cet ouvrage, au vu de notre souci de conduire en parallèle les approches à ces deux types de calculs, en même temps que la prise en main des logiciels qui les accompagnent. Nous la retrouverons constamment au fil de l'ouvrage ainsi que dans les travaux dirigés guidés qui l'illustrent et l'enrichissent.

Nous tenterons donc dès ce premier chapitre d'envisager une présentation « croisée », en même temps que conduite en parallèle, de ces deux types d'outils, les uns relevant du calcul formel, les autres du calcul scientifique. Les usages de chacun de ces deux types de logiciels pour le type de calcul (formel ou scientifique) auquel il est en priorité dédié se complètent en effet l'un avec l'autre, ce qui explique notre approche.

1.1. Une présentation des outils logiciels (sous licence ou libres)

Cette monographie a donc un double objectif :

— d'une part, proposer une première initiation aux bases du calcul *symbolique* (ou encore *formel*) et à celles du calcul *scientifique* (ou encore *numérique*), offrant la possibilité de mettre en situation certains acquis (relevant tant de l'algèbre que de l'analyse) du programme des deux premières années de licence ;

— d'autre part, permettre une familiarisation accélérée à la prise en main de plusieurs logiciels, les uns dévolus au calcul symbolique (`Maple`, `Mathematica`, ou le logiciel libre `Sage` que nous exploiterons beaucoup dans ce cours), les autres au calcul scientifique (en l'occurrence `MATLAB` ou le logiciel libre `Scilab`) et à la programmation sous ces environnements (en particulier à la programmation sous `Python` sous laquelle se trouve régie la syntaxe de `Sage`).

Nous déclinerons dans cette section ces deux types d'outils logiciels, classés ici suivant l'une ou l'autre de leur finalité (calcul symbolique ou calcul scientifique).

1.1.1. Les outils logiciels du calcul symbolique ou formel. Les logiciels `Maple` ou `Mathematica` comptent parmi les logiciels les plus utilisés dans le monde académique pour ce qui concerne le calcul symbolique. Le logiciel `Maple` [1] est avant tout un logiciel de calcul formel. Il propose des outils de calcul symbolique et fournit en parallèle de puissants outils de graphisme et de calcul. Il n'est donc pas à proprement conçu pour le calcul scientifique, quand bien même il peut être détourné à cet effet. L'ossature de son noyau s'appuie essentiellement sur l'algèbre polynomiale et la théorie algébrique de l'élimination (qui seront au cœur du chapitre 2 de cet ouvrage), ainsi que sur des outils tels que la dérivation et l'intégration formelle des fonctions appartenant à ce que l'on appelle communément la *classe de Liouville* : fractions rationnelles, logarithmes et fonctions afférentes telles que arctan, exponentielle, fonctions trigonométriques ou hyperboliques, solutions d'équations différentielles de nature algébrique, c'est-à-dire à coefficients polynomiaux. Ce logiciel académique (payant) `Maple` est, avec `Mathematica` [2] (lui aussi payant), l'un des logiciels de calcul formel les plus utilisés dans les milieux universitaires. D'autres logiciels (libres) le complètent pour des tâches plus spécifiques : `PARI/GP` [3] pour ce qui concerne plus spécifiquement la théorie des nombres (avec ses applications à la cryptographie et au codage), `Macaulay2` [4], `CoCoA` [5], `Reduce` [6] , pour l'algèbre commutative et l'algèbre polynomiale en plusieurs variables ainsi que leurs applications en géométrie algébrique, en ingénierie et en robotique.

Parmi tous ces divers logiciels, nous privilégierons beaucoup dans cet ouvrage le logiciel libre `Sage` (de fait construit sur la base de diverses sources de logiciels libres, dont `Maxima` et `PARI/GP` mentionné plus haut), que le lecteur de cet ouvrage, afin d'en profiter pleinement, se devra impérativement de télécharger et d'installer [7] depuis le site `http://www.sagemath.org/fr`, offrant toutes les potentialités du calcul symbolique implémenté sous `Maple`, ce sous le langage informatique `Python`. Comme les précédents logiciels auxquels nous avons fait référence, `Sage` est un logiciel (ou plutôt une plate-forme sur laquelle s'articulent diverses sources logicielles) conçu sur

1. Aujourd'hui dans sa version `Maple18`, voir le site dédié `http://www.maplesoft.com` .
2. `Mathematica9` actuellement, voir le site dédié `http://www.wolfram.com/mathematica` .
3. Développé et maintenu à Bordeaux, voir le site dédié `http://pari.math.u-bordeaux.fr`, où le logiciel peut être téléchargé.
4. Développé par les algébristes et géomètres Daniel Grayson, Michael Stillman, David Eisenbud, et que l'on peut télécharger sur le site `http://www.math.uiuc.edu/Macaulay2` .
5. Développé initialement à Gênes sous l'impulsion de Lorenzo Robbiano, et que l'on peut télécharger sur le site dédié `http://cocoa.dima.unige.it` .
6. Voir le site `http://reduce-algebra.com` où ce logiciel peut être téléchargé.
7. Le logiciel `Sage` est conçu pour l'environnement `Linux` ; il peut aussi fonctionner sous `Windows`, mais sous une machine virtuelle (`VirtualBox`) fonctionnant, elle, sous `Linux`. Tous les codes accompagnant cet ouvrage ayant été rédigés comme des fichiers `.sws` sous le `Notebook` de `Sage`, on conseille ici vivement de travailler avec `Sage` directement sous `Linux` plutôt que sous une machine virtuelle (`VirtualBox`) sous `Windows`. Si l'aide est loin d'être parfaite (c'est pour l'instant l'un des points faibles de cette plate-forme), le recours au `web` s'avère un précieux auxiliaire du fait surtout de la diversité et de la convivialité des divers forums de discussion dédiés à cet outil.

la base d'un langage *interprété* et non *compilé*, ce qui explique fait que les instructions initiant les blocs structurés tels les boucles `do`

```
for i in range(N):
    instruction_0
    instruction_1
```

ou les boucles `while` sous une restriction

```
while [restriction_0]:
    instruction_0
    instruction_1
```

ou plusieurs

```
while [restriction_0] and [restriction_1]:
    instruction_0
    instruction_1
```

correspondant à des boucles de calculs appelées à être mises en série soient exécutées lentement. La syntaxe présidant à l'écriture des boucles (englobant des blocs structurés imbriqués les uns dans les autres) sous `Sage` est celle de l'environnement `Python` : un bloc structuré est identifié grâce au respect de l'*indentation*, nous y reviendrons constamment lors de la rédaction des codes afférents aux exemples, exercices ou séances de travaux pratiques guidées, proposés dans cet ouvrage. Il convient de bien noter la nécessité du symbole « : » précédant la liste (indentée) des instructions à opérer dans une boucle donnée. Les boucles décisionnelles (ayant vocation aussi à être emboîtées dans les précédentes) ont pour syntaxe :

```
if [condition_0]:
   instruction_0_0
   instruction_0_1
   ...
# le bloc suivant ne figure
# que dans les procedures a plus de 2 choix
elif [condition_1]:
   instruction_1_0
   instruction_1_1
   ...
# l'ajout de k-1 blocs elif
# est possible
...
else:
   instruction k_0
   instruction k_1
   ...
```

Le lecteur est invité à travailler dans cet ouvrage sous l'environnement `Sage` et par conséquent à se familiariser au préalable, ce qui n'est pas chose difficile, à la syntaxe de l'environnement `Python` aujourd'hui introduit dans l'enseignement secondaire.

Les procédures sous `Sage` se rédigent suivant l'architecture suivante (disons que la procédure est baptisée par exemple ici `TOTOSage`) :

```
sage:
def TOTOSage(entree_0,entree_1,...):
      chaine d'intructions (avec boucles emboitees)
      ...
      return sortie_0, sortie_1, ...
```

Il faut constamment veiller (voir ci-dessus) à bien respecter les indentations dans les suites d'instructions enchaînées[1]. Une fois cette procédure exécutée (donc validée), l'instruction `TOTOSage(entree0specifiee, 1specifiee,...)` affiche la réponse suivante :

```
sage:
TOTOSage(entree_0specifiee,entree_1specifiee,...)
ans:
(sortie_0specifiee,sortie_1specifiee,...)
```

L'autre manière de procéder, si l'on ne souhaite pas l'affichage des résultats, est le suivant :

```
sage:
S = TOTOSage(entree_0specifiee,entree_1specifiee,...)
sage:
S[0]
ans:
sortie_0specifiee
sage:
S[1]
ans:
sortie_1specifiee,...
sage:
S[0]; S[1]
# sur la meme ligne de commandes
ans:
sortie_0precisee
sortie_1precisee
sage:
S[0]
S[1]
# sur deux lignes distinctes
ans:
sortie_1precisee
# la premiere sortie a ete ecrasee
sage:
S[0], S[1]
ans: (sortie_0precisee,sortie_1precisee)
```

1. Lors de la rédaction des procédures sous l'environnement `Notebook`, le curseur se position sur une position indentée après tout retour chariot suivant un symbole « : » déclarant le début d'une boucle d'instructions.

Sous `Maple`, l'architecture de la réalisation de procédures est tout aussi simple, mais toutefois différente. Prenons par exemple une procédure `TOTOMaple` que l'on rédigera dans une fenêtre (préalablement ouverte, une fois le curseur positionné derrière le prompteur `>`, en cliquant juste sur l'onglet `Insert`, puis sur `Code Edit Region`). La procédure `TOTOMaple` se rédige ainsi :

```
TOTOMaple := proc(entree_1,entree_2,...)
for k from 1 to N do
instructions_1
end do;
...

while [restriction_1] do
instructions_2
end do;
...

if [condition_1] then
instructions_3
elseif
instructions_4
else
instructions_5
end if;
```

Sous `Maple`, la déclaration (ou la redéfinition d'une variable `x`) doit se faire avec la syntaxe `x := ...` (ne pas oublier ici le « `:` ». Si cette déclaration `x:= ...` sous le prompteur est suivie de « `:` », le résultat `x` n'est pas affiché ; la même déclaration (toujours sous le prompteur), cette fois suivie de « `;` », provoque cette fois l'affichage de `x`. Voici un exemple, celui de la déclaration d'un polynôme en une variable `P`, puis de son évaluation en un point (routine `subs(X=a,P)`) :

```
> P := X^7 + X + 1: subs(X=8,P):
> P := X^7 + X +1; subs(X=8,P);
ans:
P := X^7+X+1
2097161
> P:= X^7+X+1; subs(X=8,P):
ans:
P := X^7+X+1
>> P:= X^7+X+1: subs(X=8,P);
ans:
2097161
```

Attention toutefois au fait que cette règle se trouve renversée, on le verra, sous les logiciels de calcul scientifique : faire suivre l'instruction sous le prompteur `x = ...` de « `;` » empêche l'affichage (ce qui est fort utile si `x` est un tableau de grandes, voire très grandes dimensions). Nous y reviendrons plus loin.

Les objets de `Maple` s'organisent en trois catégories :

— les *suites ordonnées* (générées par `seq`) d'objets `o1,o2,...` ; ce sont les applications d'un ensemble fini dans un ensemble d'objets (contenant toutes les entrées de la liste) ;
— les *listes* : ce sont les suites ordonnées d'objets d'un certain ensemble d'objets O, mais considérées comme prises entre crochets `[o1,o2,...]` ;
— les *ensembles* : ce sont les ensembles-images des suites ordonnées de longueur fixée ℓ considérées cette fois comme applications à valeurs dans un ensemble d'objets O ; on les délimite par une prise d'accolades $\{o1, o2, ...\}$.

Les *opérandes* d'une liste ou d'un ensemble constituent dans le premier cas la suite des objets de la liste, dans le second cas l'ensemble des objets de l'ensemble (mais libéré cette fois de la prise entre accolades, ce qui permet de le réorganiser avec un autre ensemble). La longueur d'une liste est son nombre d'opérandes `nops` ; ceci vaut aussi pour le cardinal d'un ensemble. Voici quelques petites manipulations élémentaires pour s'entraîner :

```
>L1:=[1,X,X,3,4,X^2];op(L1);
ans:
L1 := [1, X, X, 3, 4, X^2]
1, X, X, 3, 4, X^2
> E1:={op(L1)};op(E1);
ans:
E1 := {1, 3, 4, X, X^2}
1, 3, 4, X, X^2
>S1:=seq((-1)^k,k=1..8);L2:=[S1];op(L2);
ans:
S1 := -1, 1, -1, 1, -1, 1, -1, 1
L2 := [-1, 1, -1, 1, -1, 1, -1, 1]
-1, 1, -1, 1, -1, 1, -1, 1
>E2:={op(L2)};op(E2);
ans:
E2 := {-1, 1}
-1, 1
>S2:=seq((-1)^k/(k+3),k=1..5);
ans:
S2 := -1/4, 1/5, -1/6, 1/7, -1/8
>L3:=[S2];E3:={op(L2),op(L3)};
ans:
L3 := [-1/4, 1/5, -1/6, 1/7, -1/8]
E3 := {-1, 1, -1/4, -1/6, -1/8, 1/5, 1/7}
>E3[2]; E3[6];
ans:
1
1/5
>op(0,L3);op(0,E3);
ans:
list set
```

```
>L4:= [seq((-1)^k/(k+1),k=0..9)];nops(L4);L4[5];L4[8];
ans:
L4 := [1,-1/2,1/3,-1/4,1/5,-1/6,1/7,-1/8,1/9,-1/10]
10
1/5
-1/8
```

Thème de travaux pratiques 1.1 (première prise en main de `Sage` : une mise en forme d'un test de la conjecture de Syracuse grâce à une boucle `while`). On sera fréquemment appelé au fil de ce cours à rédiger des codes impliquant une ou plusieurs boucles `while`, ainsi que des boucles de décision ; on retrouve pareille architecture dans la rédaction de codes couplés avec des test d'arrêt de procédure. La célèbre *conjecture de Syracuse* a été soulevée par le mathématicien allemand Lothar Collatz autour de 1930 et demeure toujours aujourd'hui (sans doute pour longtemps, car aucun angle d'attaque connu ne semble envisageable) à l'état de conjecture. Elle se formule très simplement. Étant donné un nombre entier N, on génère la suite d'entiers $(u_k(N))_{k\geq 0}$ de la manière inductive suivante :

$$u_0(N) = N, \qquad u_{k+1}(N) = \begin{cases} u_k(N)/2 \text{ si } u_k(N) \text{ est pair} \\ 3N+1 \text{ si } u_k(N) \text{ est impair} \end{cases} \tag{1.1}$$

La conjecture stipule que, quelque soit la valeur de l'entier N initiant la suite, la suite $(u_k(N))_{k\geq 0}$ finit par atteindre au bout d'un temps fini la valeur $u_{k_0}(N) = 1$.

(1) À supposer que la suite $(u_k(N))_{k\geq 1}$ atteigne pour la première fois la valeur 1 au cran k_0 ($u_{k_0}(N) = 1$), quel est le comportement ultérieur de cette suite ?

(2) Rédiger sous `Sage` une procédure qui, si on lance

```
sage:
S = SYRACUSE(N,Niter)
```

avec une variable d'entrée prenant une valeur spécifiée `N` dans $\mathbb{N}^*$ et un nombre maximal `Niter` d'itérations lui aussi spécifié, retourne :
- le commentaire `conjecture non validee` si la suite $(u_k(\mathtt{N}))_{k\geq 0}$ s'est révélée incapable d'atteindre 1 au bout d'un nombre d'itérations égal à `Niter` de la procédure de décision décrite en (1.1), suivi des valeurs de $u_{\mathtt{Niter}}(\mathtt{N})$ et de `N` ;
- retourne sinon la liste de tous les termes de la suite depuis $u_0(\mathtt{N}) = \mathtt{N}$ jusqu'au premier terme $u_{k_0}(\mathtt{N})$ valant 1 (inclus) comme sortie `S[0]` ainsi que le nombre nécessaire d'itérations pour y parvenir, c'est-à-dire la longueur de la suite `S[0]` moins 1, comme sortie `S[1]`.

On utilisera pour cela le fait que `mod (a,2)` désigne sous `Sage` le reste de la division euclidienne d'un entier `a` $\in \mathbb{Z}$ par 2 (0 si `a` est pair, 1 si `a` est impair).

(3) Rédiger une procédure analogue

```
S = SYRACUSEMaple(N,Niter);
```

réalisant la même chose que `SYRACUSE` sous `Maple`. La condition [{`a` pair}] (respectivement [{`a` impair}]) s'énonce sous la forme `type(a,even)=True` (respectivement `type(a,odd)=True`). D'autre part, pour ajouter un nouvel objet

`a` à une liste d'objets `Liste`, on redéfinira donc la variable `Liste` en posant `Liste := [op(Liste),a]` ; en effet `op(Liste)` désigne la liste de *opérandes* ou objets de la liste `Liste` (organisée comme une suite ordonnée).

1.1.2. Les outils logiciels du calcul scientifique ou numérique. Aux antipodes de `Maple` et du calcul symbolique (calcul « sans pertes »), le logiciel (payant) `MATLAB`, qui utilise aussi un langage de programmation ressemblant plutôt à un langage de *script* qu'à un langage compilé tel que le `C` ou `FORTRAN 90`, est un logiciel de calcul scientifique (donc cette fois de calcul « avec pertes »). L'ossature de son noyau repose sur le calcul matriciel [1]. Si ce type de langage *script* exécute les instructions bien plus lentement qu'un langage compilé (même remarque qu'à propos de `Maple`, mis à part que les choses s'avèrent plus cruciales ici car les tableaux en jeu dans les calculs peuvent être de taille énorme), ce type de langage dit « interprété » se justifie par un temps de conception et de « débogage » excessivement réduits par rapport à ceux que nécessite un langage compilé. Dans le milieu industriel où il est communément répandu, `MATLAB` (souvent couplé avec `SIMULINK`, ces deux logiciels étant gérés par la société `Mathworks`) est un logiciel professionnel (payant) servant essentiellement à concevoir des maquettes de programmes et à les tester, ce avant une phase ultérieure de programmation, cette fois sous un logiciel compilé tel `C++` ou `FORTRAN 90`. Comme `Maple`, `MATLAB` offre une riche bibliothèque de fonctions et de documentation. Les `toolboxes` suivants : `Image Processing`, `Signal Processing`, `Statistics`, `Partial Differential Equations`, `Wavelets` pourront être mis à contribution dans cet ouvrage, mais le noyau du logiciel suffira en général à nos besoins.

Il faut mentionner ici que le logiciel libre `Scilab` réalise un clone de `MATLAB` ; il est conçu et géré par le Consortium `Scilab` (INRIA, ENPC). Il est couramment utilisé, mais essentiellement dans le milieu universitaire [2]. D'autre part, `Sage` (que nous avons déjà mentionné au titre de logiciel libre offrant les potentialités de `Maple` ou `Mathematica` sous l'environnement `Python`) intègre l'interface avec `Scilab`. À l'heure actuelle, certes seul `MATLAB` est réellement connu et exploité dans le milieu industriel, mais l'essor de `Scilab` et de `Sage` est indéniable. Les programmations sous `MATLAB` ou `Scilab` sont très proches. Un code sous `MATLAB` est rédigé dans un fichier `.m` sous la forme

```
function [Sortie1,...,SortieN] = Code(Entree1,...,EntreeM)
% commentaires et description
Blocs d'instructions
end
```

Un code sous `Scilab` est rédigé dans un fichier `.sci` sous la forme quasiment identique :

```
function [Sortie1,...,SortieN] = Code(Entree1,...,EntreeM)
// commentaires et description
```

1. L'acronyme `MATLAB` vient d'ailleurs de là : `MATLAB` pour `Matrix Laboratory`.

2. On peut télécharger le logiciel libre `Scilab` depuis le site `http://www.scilab.org`. L'introduction de `Scilab` dans l'enseignement secondaire est aujourd'hui en phase bien avancée et il existe un lien où l'on peut télécharger une version conçue pour l'apprentissage de ce logiciel dans les lycées : `http://www.scilab.org/fr/community/education/maths` .