Chapitre 1

Introduction Au Calcul Formel

**1.1 Définitions**

* Le terme calcul formel (en anglais computer algebra) désigne l’étude de la *représentation* et de la *manipulation effective* d’*objets mathématiques* dans un *ordinateur*.
* En d’autres termes, le calcul formel consiste à étudier dans quelle mesure et à quelle vitesse un ordinateur peut faire des mathématiques.
* Pour un problème donné, nous serons donc amené à *produire un algorithme* (i.e., une suite de règles opératoires permettant, en un nombre finie d’étapes, d’arriver avec certitude au résultat) et en *évaluer les performances*.
* Les systèmes de calcul formel manipulent principalement trois types d’objets à savoir des *nombres*, e.g., des entiers relatifs (arithmétique), des *polynômes* et des *matrices* (algèbre linéaire).
* Un premier aspect du calcul formel consiste donc à proposer des *algorithmes de calcul efficaces pour la manipulation de ces structures de bases*.
* Un deuxième aspect se propose de ramener d’autres problèmes à ces derniers comme par exemple montrer que la plupart des opérations d’algèbre linéaire peuvent se réduire à calculer des produits de matrices.
* Les procédés que nous construirons reposeront toujours sur de l’algèbre donc avant de construire un algorithme il faudra toujours se placer dans un certain *modèle/cadre algébrique*.

**1.2. Structures Algébriques Effectives**

Étant donnée une structure algébrique (e.g., groupe, anneau, corps, espace vectoriel) la première question qui se pose avant de construire des algorithmes manipulant des objects de cette structure est le problème de la calculabilité :

suis-je capable de faire des calculs dans cette structure et d’apprendre à un ordinateur à les faire ?

Le principal problème est le *test d’égalité* c’est-à-dire le problème de décider si une expression donnée dans cette structure vaut 0 ou non.

En effet pour effectuer certains calculs, comme par exemple une division, il est souvent crucial de déterminer si une expression représente 0 ou non. Notons que ce *test à zéro* n’est pas décidable dans l’ensemble des nombres réels R.

En calcul formel, on se ramène donc toujours à faire des calculs dans une *structure effective* où le test à zéro est décidable.

**Définition** : Une structure algébrique est dite effective si l’on dispose :

1. d’une structure de données pour en représenter les éléments ;
2. d’algorithmes pour en effectuer les opérations et pour y tester l’égalité à zéro.

Le lemme suivant regroupe les résultats d’effectivité sur lesquels nous nous appuierons dans ce cours.

**Lemme** : On a les résultats suivants :

1. L’anneau Z des entiers relatifs est effectif ;

2. Si A est un anneau effectif, alors son corps des fractions est effectif, A[X] est effectif et pour N $ϵ$ N

 ,A[X]/(XN+1) est effectif ;

3. Si K est un corps effectif, alors sa clôture algébrique $\overbar{K}$, l’espace vectoriel Kn et l’anneau Mn(K) sont

 effectifs.

**1.3. Calcul Formel Versus Calcul Numérique**

Les méthodes du calcul numérique classique sont basées sur deux types de données :

* les entiers relatifs, représentés par un nombre fixe de positions binaires (32 ou 64 bits) ;
* les nombres représentés par le système de virgule Flottante Normaliée (VFN).

Un nombre en VFN binaire représenté par un mot mémoire de taille fixe structuré selon le schéma suivant :

(0 ou 1, m positions binaires, 0 ou 1, c positions binaires) qui représente (le signe du nombre, la mantisse ou partie fractionnaire, signe de l’exposant de 2, la caractéristique de longueur c positions binaires) ; par exemple (1,101110,0,101) représente -0,101110 x 2+101. La première position binaire de la mantisse étant toujours égale 1. Ce système de représentation entraîne des limitations évidentes qui seront illustrées sur quelques exemples dans ce qui suit.

**Quelques lacunes du système de virgule flottante**

**Exemple 1**: Dans un système VFN décimal avec une mantisse de longueur 2, les nombres positifs accessibles sont :

 ........ 0,10 x 10-1 0,10 x 100 0,10 x 10+1 0,10 x 10+2 ................

 ........ 0,11 x 10-1 0,11 x 100 0,11 x 10+1 0,11 x 10+2 ................

 ... ... ... ... ...

 ... ... ... ... ...

 ........ 0,99 x 10-1 0,99 x 100 0,99 x 10+1 0,99 x 10+2 ................

Il est facile de remarquer que cette représentation de R+ est de plus en plus dense près de zéro est de moins en moins dense lorsqu’on s’en éloigne. Ensuite, on peut aussi vérifier que la somme des deux éléments 0,10 x100 et 0,10 x102 est 0,10 x102; autrement dit, l’addition est inopérante sur ces deux nombres.

En fait, lorsqu’on opère sur des nombres d’ordres de grandeur différents ce phénomène se produit ; il est dû à l’*erreur d’arrondi* ou *de troncature*.

**Exemple 2** : L’évaluation du polynôme 1 + x + x2 + x3 + . . . + x7 + x8 + x9 en x = 0,1 est égale à l’évaluation du polynôme 1 + x + x2 + x3 + . . . + x6 en cette même valeur de x, avec un langage évolué comme FORTRAN, en précision ordinaire. Ceci est vrai car avec une mantisse de longueur 5, qui correspond à ce qu’on appelle la simple précision, l’addition de 1 à 10-7 donne 1.

**Exemple 3** : Suivant le même principe, on voit que l’opération d’addition :

 1 + 0,009 + 0,009 + 0,009 + 0,009 + 0,009

dans un système VFN à trois chiffres significatifs (mantisse de longueur 3) donne comme résultat 1 si les additions sont effectuées de gauche à droite et 1,04 si les additions sont effectuées de droite à gauche. Donc l’opération d’addition n’est plus commutative dans un système VFN.

**Exemple 4** : Soit l’expression suivante : y := d x (a – b + c)

Avec un système VFN à 3 chiffres significatifs où : d = 100, a = 321, b = 321, c = 0,88. En procédant comme dans l’exemple 1, a – b = 0, a – b + c = 0,88 d’où d x (a + b – c) = 88. L’ordre des opérations peut être effectué différemment de l’exploration gauche-droite, par exemple si on utilise un *optimiseur*. Si le programme comporte l’expression h := a + c puis l’expression y := d x (a – b + c), l’optimiseur évalue d’abord h : = 321 + 0,88, c.à.d. h = 321, puis y := d x (h – b) c.à.d. d x (321 – 321) et le résultat est 0. C’est le phénomène d’amplification d’une erreur troncature par un grand coefficient.

**Exemple 5** : On peut vérifier que y(x) = 1/e. e-x est solution de l’équation différentielle très élémentaire :

 y" – y' = 0 avec les conditions initiales ; y(0) = 1/e et y'(0) = -1/e

Si on veut vérifier l’efficacité d’une méthode numérique par *pas*, par exemple les méthodes d4Euler, de Range-Kutta, etc., il faut commencer par donner une valeur numérique à la valeur initiale 1/e, par exemple 0,3679 ( valeur approchés par excès) puis une valeur numérique à -1/e, par exemple -0,3677 (valeur approché par défaut), puis on effectue le calcul *numérique*. On obtient le résultat représenté graphiquement ci-dessous, où la courbe obtenue numériquement est dessinée par points et la courbe exacte en trait plein. On peut clairement constaté le phénomène de bifurcation qui se produit ( même si 1/e et -1/e sont approchés avec plus de chiffres significatifs). Pour expliquer ce phénomène, nous pouvons remarquer que la solution générale de l’équation homogène est :

 y(x) = ae-x + be+x

les valeurs des paramètres a et b, qui correspondent à la solution particulière avec les conditions initiales imposées, sont solutions du système linéaire :

 y(0) = a + b = 0,3679

 y'(0) = -a + b = -0,3677

c.à.d. a = 0,3678 et b = 0,0001. Donc la solution est : y(x) = 0,3678e-x + 0,0001e+x .

Nous constatons que la croissance de la solution numérique vers +∞ est due uniquement à l’*erreur de représentation* sur 1/e, qui entraîne dans la solution la présence d’un coefficient b non nul (même très petit) de la composante fondamentale *dominante* e+x quand x tends vers +∞.

Un système de calcul formel donne comme solution de l’équation y" – y' = 0, la liste de 2 solutions fondamentales sous forme de fonctions élémentaires (e-x,e+x). Chercher la solution particulière vérifiant les conditions initiales : y(0) = 1/e et y'(0) = -1/e consiste à chercher les coefficients a et b de la solution générale : y(x) = ae-x + be+x en dérivant d’abord formellement y(x) : y'(x) = -ae-x + be+x puis en résolvant le système linéaire : y(0) = a + b = 1/e et y'(0) = -a + b =-1/e. D’où : a = 1/e et b = 0.

Ainsi le calcul formel donne le résultat exacte : y(x) = (-1/e).e-x + 0e+x, mais renseigne aussi sur l’origine de l’instabilité mathématique due à la confrontation d’une solution *dominante* e+x (pour x>= 0) et d’une solution minimale e-x.

Il nous indique également le remède de stabilisation des calculs par changement de variable : x = c – ξ, où c est une constante de R+. la solution générale devient y(ξ) = ae-c+ξ + be+c-ξ et la solution particulière (1/e)e-x devient y(ξ) = (1/e)e-c+ξ. Ce qui inverse la situation précédente : la solution vraie (1/e)e-c+ξ *dominante* cette-fois, sera perturbée éventuellement par une composante *minimale* εece-ξ qui s’effacera devant la vraie solution quand ξ sera croissante.

*Des différents exemples précédents il découle la nécessité d’avoir des outils qui permettent non seulement de travailler de manière exacte, mais encore dans de vraies structures de groupes, d’anneaux, de corps,* qui permettent l’utilisation de paramètres ou de nombres algébriques, et qui permettent d’utiliser des fonctions et applications connues ou d’en fabriquer de nouvelles.

 La difficulté est de faire correspondre à ces objets mathématiques abstraits des objets qui ont une *forme* de *représentation* qui puisse ensuite être réduite en structures de données sur machine.

**1.4. Représentation des entiers relatifs (Integers)**

 Les entiers fournis par les processeurs des ordinateurs sont des entiers modulo une puissance de 2 (typiquement 32 ou 64) ; et sont appelés des entiers machine. Les opérations rendues possibles par les processeurs sont l’addition, la soustraction, la multiplication et parfois la division. La norme ANSI du compilateur C fournit processeur ne le fait pas.

 Pour manipuler des entiers dont la taille dépasse celle d’un mot machine, et donc pouvoir manipuler des entiers avec des tailles quelconques, il est commode de les considérer comme écrits dans une base B assez grande :

 N = a0 + a1B + a2B2 . . . + akBk où 0 <= ai <= B, le signe étant stocké séparément.

Ces nombres peuvent être stockés dans des tableaux d’entiers machine et seront de taille arbitraire.

L’addition et la multiplication sur ces nombres peuvent être réduites à des opérations sur des entiers inférieurs à B2, au prix de quelques propagations de retenues. Le choix de B dépend un peu du processeur ; il peut, selon les cas être aussi grand que le plus grand entier tenant sur un mot machine, ou être égal à sa racine carrée.

**Les opérations de base sur les entiers longs**

**Entiers longs**. Soit $w\geq 1$, on encode l’entier $N$ en utilisant des nombres de $w$ bits (nombres entiers entre 0 et $2w - 1$). On écrit :

$$N=(-1)^{s}\sum\_{i=0}^{l}a\_{i}2^{wi} avec s \in \{0, 1\} et a\_{i} \in \{0, . . . , 2w - 1\} . $$

On représente $N$ par $(s · 2^{w-1} + l + 1, a\_{0}, . . . , a\_{l})$.

Cette représentation est unique si $a\_{l}\ne 0$ et de taille $O\_{w}(log(N))$.

**Exemple**. $-1$ est represent´e par $(2^{w-1} + 1, 1).$

**Restriction**. On doit avoir $-2^{w.2^{w-1}} + 1\leq N\leq -2^{w.2^{w-1}}- 1$

Taille maximale pour 32 bits : 68 719 476 735 bits $≈$ 8 GB

Taille maximale pour 64 bits : $5,9 .10^{20}$ bits $≈$ $6,8 .10^{20}$ GB

**Addition de deux entiers de même signe (et de même longueur).**

$$N=(-1)^{s}\sum\_{i=0}^{l}a\_{i}2^{wi} et M=(-1)^{s}\sum\_{i=0}^{l}b\_{i}2^{wi}$$

**Algorithme.**

(1) Faire $t\_{0}\leftarrow 0$

(2) Pour $i = 0$ à $l$, faire

 $c\_{i}\leftarrow a\_{i}+b\_{i}+t\_{i},t\_{i+1}\leftarrow 0$

 Si $c\_{i}\geq 2^{w}$, alors faire $c\_{i}\leftarrow c\_{i}-2^{w},t\_{i+1}\leftarrow 1$

(3) Si $t\_{i+1}=1$ alors poser $c\_{i+1}\leftarrow 1$ et retourner $\left(s+l+2, c\_{0}, . . . , c\_{l+1}\right),$

 sinon retourner $\left(s+l+1, c\_{0}, . . . , c\_{l}\right).$

**Complexité.** $O\left(l\right)= O(logN)$ opérations sur des mots de w bits. De ce point de vue, l’algorithme est optimal.

**Addition de deux entiers de signes différents (et de même longueur).**

$$N=(-1)^{s}\sum\_{i=0}^{l}a\_{i}2^{wi} et M=(-1)^{s}\sum\_{i=0}^{l}b\_{i}2^{wi} avec N+M \geq 0$$

**Algorithme.**

(1) Faire $t\_{0}\leftarrow 0$

(2) Pour $i = 0$ à $l$, faire

 Si $a\_{i}\leftarrow b\_{i}+t\_{i}$, alors faire

 $c\_{i}\leftarrow a\_{i}-b\_{i}-t\_{i},t\_{i+1}\leftarrow 0,$

 sinon faire

 $c\_{i}\leftarrow 2^{w}+a\_{i}-b\_{i}-t\_{i},et t\_{i+1}\leftarrow 1$

(3) Retourner $\left(l+1, c\_{0}, . . . , c\_{l}\right).$ [non réduit]

**Remarques**. Complexité $O(l)$ optimale aussi.

Tester si $N +M \geq 0$ se fait en temps $O(l)$ aussi.

Possibilité de faire un algorithme sans l’hypothèse $N +M \geq 0$.

**Multiplication de deux entiers (de même longueur).**

$$N=(-1)^{s}\sum\_{i=0}^{l}a\_{i}2^{wi} et M=(-1)^{s}\sum\_{i=0}^{l}b\_{i}2^{wi} $$

**Algorithme.**

(1) Pour $i = 0$ à $l$, faire $T\_{i} \leftarrow (a\_{i} · |M|) · 2^{wi},$

(2) Retourner $(-1)^{s+t}\sum\_{i=0}^{l}T\_{i}.$

**Remarques**. La multiplication de l’entier long $|M|$ par un entier de $w$ bits se fait en $O(l)$ opérations.

La multiplication par $2^{wi}$ est juste un décalage.

La somme finale se fait avec l’algorithme précédent.

Complexité. $O(l^{2})$ opérations sur des mots de $w$ bits. Complexité non optimale !

**1.5. Approximation des nombres réels par des nombres rationnels**

**Les Fractions Continues**

Quel que soit le nombre de décimales qu’on choisisse de mettre pour représenter un nombre réel, π par exemple, il ne s’agira que d’une approximation, qui n’est jamais exactement égale au véritable , par exemple



**Les fractions continues, c’est une autre manière de représenter et d’approximer des nombres réels**, **une alternative à l’écriture décimale.**

Pour commencer, si on veut approximer  on pourrait écrire :



On peut décider de prendre l’inverse de ce petit quelque chose



et donc d’écrire



Si maintenant on laisse tomber ce qu’il y a après la virgule derrière le nombre 7, on obtient



On peut à nouveau inverser ce petit quelque chose dans le nombre 7,062513 et voir que



Si on injecte ça dans notre approximation, on obtient



Evidemment vous me voyez venir, **on peut continuer à jouer à ce petit jeu plusieurs fois de suite** et obtenir que



On peut noterez les points de suspension tout en bas : en fait on peut empiler les étages à l’infini sans que ça ne s’arrête jamais.

**Une représentation des nombres réels**

De manière générale, on appelle fraction continue une expression de la forme :



Ce qu’il y a d’intéressant, c’est que **tout nombre réel peut s’écrire sous cette forme**. Et réciproquement spécifier la suite des nombres entiers $a\_{0};a\_{1;}a\_{2;}a\_{3}…$ suffit pour reconstruire le nombre de départ. Il s’agit donc d’une manière nouvelle de représenter ou d’identifier un nombre réel.

Pour éviter de remplir des pages de fractions qui s’empilent, on utilise une notation plus compacte, on écrit simplement

$$[a\_{0};a\_{1;}a\_{2;}a\_{3}…]$$

Vous pouvez très bien voir cette écriture comme une alternative au développement décimal : **plutôt que de spécifier un nombre en écrivant la suite de ses décimales, vous pouvez donner la suite de son développement en fraction continue**.

Cette écriture a comme avantage qu’elle est indépendante du choix d’une base, alors que l’écriture décimale est bien sûr liée à la base 10 : en base 6 ou 23, les « décimales » de  seraient bien sûr différentes ! Mais le développement en fraction continue a plein d’autres avantages pratiques et esthétiques : commençons par la pratique.

**Faire des approximations**

Une chose que l’on fait tout le temps avec le développement décimal, c’est de le tronquer pour faire des approximations. Par exemple si j’écris $π≈3,1415$, j’approxime en tronquant son développement décimal.

Avec le développement en fraction continue, on peut faire pareil. Si on tronque le développement de après le deuxième nombre, on obtient

![\pi \approx [3;7] = 3 + \cfrac{1}{7}]()

Si je réduis le membre de droite au même dénominateur, cela donne



qui est **une fameuse approximation** de . Mais si je tronque plus loin le développement en fraction continue de , et après réduction au même dénominateur, **j’obtiens d’autres approximations rationnelles** (c’est-à-dire sous forme de fractions)

![\pi \approx [3;7;15] = \frac{333}{106}=3.14150943396226]()

![\pi \approx [3;7;15;1] = \frac{355}{113}=3.14159292035398]()

![\pi \approx [3;7;15;1;292] = \frac{103993}{33102}=3.14159265301190]()

Il est vrai que l’écriture décimale fournit elle aussi des approximations rationnelles de , par exemple quand on écrit que est égal à 3,1415, on écrit en fait que



Mais vous voyez qu’il faut aller chercher des dénominateurs et numérateurs très élevés, là où les approximations issues du développement en fraction continue utilisent des nombres bien plus petits.

En fait on peut même démontrer que **les approximations rationnelles issues des fractions continues sont les meilleures possibles de toutes** (« meilleures » dans le sens : tout approximation rationnelle meilleure que ça devra faire appel à des numérateurs et dénominateurs plus élevés).

L’approximation 355/113 de est particulièrement intéressante, car **elle est vraiment excellente (6 chiffres après la virgule) tout en utilisant un dénominateur pas trop grand**



Une des raisons pour lesquelles cette approximation est particulièrement bonne, c’est qu’elle est issue d’une troncation du développement en fraction continue effectuée juste avant un grand nombre (292), puisque

![\pi = [3;7;15;1;292;1;1;2;1;\cdots]]().

Or si nous reprenons les constructions du début, nous constaterons qu’un grand nombre à un endroit donné du développement signifie un petit « pouillième » après la virgule. Bref **si on utilise des fractions continues pour créer des approximations rationnelles de nombres réels, c’est en général une bonne idée de les couper juste avant un grand nombre**.

**Quelques développements en fractions continues**

On sait que le développement décimal d’un nombre rationnel est périodique (ou tout du moins il fini toujours par être périodique), par exemple



En revanche, il n’est que rarement fini (ça ne se produit que quand le dénominateur n’admet que 2 et/ou 5 comme facteur premier).

Au contraire **le développement en fraction continue d’un rationnel est lui toujours fini**. Et quand il est infini mais périodique, c’est que l’on a affaire à des nombres dits **irrationnels quadratiques** : ceux qui sont de la forme . Un exemple simple c’est dont le développement en fraction continue est enfantin

![\sqrt{2} = [1;2;2;2;2;2;2;\cdots]]()

Mais **le plus beau des développements en fraction continue est celui du célèbre nombre d’or**

![\phi = [1;1;1;1;1;1;1;1;1;\cdots]]()

Remarque : on a dit que de grands nombres dans le développement signifiaient que le nombre s’approximait efficacement. Le fait que celui du nombre d’or ne contienne que des 1 implique que **le nombre d’or est le nombre réel le plus difficile à approximer par des rationnels** !