Calcul algébrique et formel

J. Davenport, T. Gautier, N. Revol, J.-L. Roch et G. Villard

ENSIMAG 1995

Table des matières

Chapitre 1

Introduction à MAPLE Thierry GAUTIER

1.1 MAPLE : un système de calcul formel

Les différents livres sur MAPLE définissent celui-ci comme un *environnement de calcul mathématique*. Qu'est-ce qu'un environnement de calcul mathématique sur un ordinateur ? Dans les exemples de ce chapitre nous allons en montrer deux aspects. Premièrement, MAPLE permet de faire du calcul numérique comme tous les langages classiques (Fortran, Pascal) et possède aussi des fonctions beaucoup plus évoluées. Ensuite, un tel logiciel permet de manipuler des expressions très générales comme des polynômes, des séries formelles et aussi des systèmes d'équations différentielles ou des programmes. Le regroupement, dans un même environnement, de ces deux outils de calcul numérique et symbolique, associés à des outils graphiques évolués, fait l'originalité et la puissance de ce type de logiciel.

Nous allons, dans ce premier chapitre, présenter MAPLE à travers des exemples. Ces exemples, bien que parfois naïfs, doivent permettre au débutant de se familiariser avec les principales notions et de maîtriser un certain nombre de fonctionnalités de base. Ils permettront aussi de montrer la différence entre les possibilités de MAPLE et celles de Pascal. Si la manipulation symbolique des expressions est quelque chose de très puissant, encore faut-il pouvoir simplifier ces expressions et les comparer entre elles. Nous terminerons ces exemples sur un paradigme de l'utilisation de tels environnements de calcul mathématique : ils ne se substitueront jamais à l'étude d'un problème avant sa « résolution ».

Calcul formel et calcul numérique

Soit l'équation $x^2 + 12x - 5$, quelles sont ses racines ? Avec un programme écrit en Pascal il est possible de les calculer *numériquement*, avec une erreur de l'ordre de 10 chiffres significatifs on obtiendrait le résultat suivant :

$$x_0 \approx -12.40312424, \;\; x_1 \approx 0.4031242374.$$

Avec MAPLE, si on demande la résolution de cette équation (le symbole `>` est le « prompt » indiquant que l'on travaille sous MAPLE) par la fonction `solve` qui retourne une liste des solutions [1] :

```
> solve(x^2 + 12*x - 5) ;
```

On obtient le résultat (`sqrt` = *square root* = racine carrée) :

```
   -6 + sqrt(41),  -6 - sqrt(41)
```

1. dans ce cas simple

La résolution de cette équation est *exacte*, le résultat obtenu n'est pas une approximation numérique, mais une représentation formelle de la solution sous forme de radicaux.

Un environnement de calcul mathématique manipule des objets (entiers, rationnels, polynômes, séries...) de manière formelle ; on appelle aussi ces logiciels des *systèmes de calcul formel*. La fraction rationnelle 1002/13 ne sera pas approchée, comme en Pascal, soit par le quotient de la division euclidienne des entiers, soit par un développement décimal de la forme 77.07692... avec un certain nombre de chiffres :

```
> 1002/13;
                                   1002
                                   ----
                                    13
```

Néanmoins, on peut forcer une évaluation numérique par la fonction **`evalf`** (*eval floating point*) qui retourne une approximation numérique de son argument :

```
> evalf(1002/13);

                                 77.07692308
```

En rajoutant un second argument (qui est optionnel), on peut aussi préciser le nombre de chiffres de l'approximation numérique :

```
> evalf(1002/13, 50) ; # avec 50 chiffres !!!

         77.076923076923076923076923076923076923076923076923
```

Sur la ligne MAPLE ci-dessus, le symbole **#** signale un début de commentaire, ce commentaire se finissant avec la ligne.

Cette représentation formelle des objets a un avantage indéniable par rapport au calcul numérique : aucune approximation n'étant faite au cours des calculs, les résultats ne sont pas entachés d'erreurs numériques.

Considérons la suite (cet exemple est dû à Jean-Michel Muller[40]) :

$$\left\{ \begin{array}{lcl} a_{n+1} & = & 111 - \frac{1130}{a_n} + \frac{3000}{a_n a_{n-1}} \\ a_0 & = & \frac{11}{2} \\ a_1 & = & \frac{61}{11} \end{array} \right.$$

La limite de cette suite est 6. Écrivons un programme Pascal qui affiche les approximations numériques des itérés successifs. Les treize premières valeurs sont reproduites dans la table 1.1.1.

n	valeur calculée
0	5.500000000000000
1	5.545454545454546
2	5.590163934426229
3	5.633431085043981
4	5.674648620510027
5	5.713329052378342
6	5.749120919664605
7	5.781810919824309
8	5.811314226859892
9	5.837656352257866
10	5.860948153832368
11	5.881319751541142
12	5.898177025615013

TABLE 1.1.1 Convergence de la suite vers 6.

n	valeur calculée
13	5.897965247556456
14	5.647011084038567
15	0.968339944529745
16	−507.321605162467449

TABLE 1.1.2 Changement du comportement de la suite.

n	valeur calculée
17	107.120635232806222
19	100.023518606060165
21	100.000083852795825
23	100.000000299887063
25	100.000000001074625
27	100.000000000003865
29	100.000000000000014
30	100.000000000000000

TABLE 1.1.3 Convergence vers une autre limite.

Initialement, la suite semble converger vers 6, mais à partir de a_{13} la suite décroît jusqu'à des valeurs (complètement erronées[2]) (table 1.1.2). Ensuite la suite semble converger vers une autre

2. Les nombres manipulés ici ne sont que des approximations numériques, aussi sont-ils forcément faux. Mais dans ce cas aucun chiffre n'est correct, la perte de précision est complète.

limite qui est 100 (table 1.1.3). Que s'est-il passé ?

n	valeur calculée	approximation numérique
0	$\frac{11}{2}$	5.500000000
1	$\frac{61}{11}$	5.545454545
2	$\frac{341}{61}$	5.590163934
3	$\frac{1921}{341}$	5.633431085
4	$\frac{10901}{1921}$	5.674648621
$\vdots$	$\vdots$	$\vdots$
12	$\frac{14281397141}{2420922961}$	5.899153906
13	$\frac{84467679721}{14281397141}$	5.914524951
14	$\frac{500702562701}{84467679721}$	5.927741408
15	$\frac{2973697798081}{500702562701}$	5.939050485
16	$\frac{17689598897861}{2973697798081}$	5.948687492
17	$\frac{105374653934041}{17689598897861}$	5.956870732
$\vdots$	$\vdots$	$\vdots$
29	$\frac{2220052422953488836415401}{370319178017119886864 21}$	5.994970163
30	$\frac{13311001311974775399767 81}{2220052422953488836415401}$	5.995804952
$\vdots$	$\vdots$	$\vdots$

TABLE 1.1.4 Valeurs rationnelles calculées sous MAPLE et leurs approximations numériques.

Si, de la même façon que le programme Pascal, on écrit un programme MAPLE qui calcule les itérés successifs en arithmétique rationnelle, on obtient les valeurs de la table 1.1.4. La valeur de a_{100} est :

$$\frac{3919911780443470697630731873534775187362173746498050984709582036730929382067381}{653318631388679958306808321275343473365006007205019222633302014072286448118001}$$

dont une approximation numérique est 5.999999988... Sous MAPLE et en utilisant des rationnels, la suite semble converger vers 6.

Bien entendu, si en MAPLE on utilise aussi l'arithmétique réelle (avec un nombre fixé de chiffres significatifs, par exemple 100), alors le phénomène observé avec le programme Pascal se reproduira toujours. Le problème vient de la représentation des nombres réels (un nombre fixé de chiffres) qui provoque des erreurs d'approximation, et de la nature très instable du point fixe de la suite. En utilisant les rationnels de MAPLE, le calcul est exact, mais la taille de chacun des itérés devient de plus en plus grande et les opérations arithmétiques de plus en plus coûteuses en temps et en occupation mémoire.

Calculons les points fixes de la suite. Ceux-ci sont solutions de l'équation $x = 111 - \frac{130}{x} + \frac{3000}{x^2}$, soit :

```
> solve(x = 111-1130/x+3000/(x*x)) ;
          100,5,6
```

les points fixes sont $a' = 100$, $a'' = 5$ et $a''' = 6$ (on pouvait se douter que 100 et 6 étaient solutions). MAPLE étant un logiciel qui contient des algorithmes de résolution d'équations récurrentes, utilisons-les :

```
>  rsolve( { a(n+1) = 111 - 1130/a(n) + 3000/a(n)/a(n-1), a(0)=11/2, a(1)=61/11 } , a(n) );
```

```
                      61                    1130        3000
rsolve({a(1) = ----, a(n + 1) = 111 - ---- + ------------, a(0) = 11/2}, a(n))
                      11                    a(n)   a(n) a(n - 1)
```

La fonction `rsolve` admet deux paramètres : le premier est un ensemble (entre les accolades `{...}`), le second est le terme général de la suite dont on cherche une forme explicite. Pour cet exemple, MAPLE ne peut pas déterminer une forme explicite, et la valeur retournée est une expression non évaluée de l'appel à la fonction[3]. Certaines fonctions MAPLE acceptent ce type d'expressions non évaluées comme, par exemple, la fonction `asympt` qui permet de calculer un développement asymptotique d'une fonction à une variable. Le lecteur pourra vérifier que MAPLE (au moins cette version V.3) n'arrive pas à calculer un tel développement.

Nous arrêtons ici la comparaison avec le calcul numérique à travers Pascal. Les moyens mis en œuvre en calcul numérique et en calcul formel sont de natures différentes. Le numéricien va chercher à discrétiser le problème avant de le résoudre (en utilisant une représentation discrète des nombres réels). En calcul formel, on va calculer une solution par un algorithme direct « plus mathématique ». Cela se révèle au travers des algorithmes utilisés : les méthodes de résolution numérique sont généralement des méthodes itératives, souvent plus stables numériquement, et qui utilisent des propriétés sur les structures de données obtenues après discrétisation (cf. gradient conjugué qui n'utilise que l'opération « produit matrice-vecteur »). Ces méthodes sont rapides mais parfois incapables de résoudre certains problèmes à cause de leurs mauvais conditionnements numériques. Les méthodes du calcul formel sont itératives (méthode de Newton sur des séries) ou directes (pas de problème de stabilité numérique). En revers de médaille, si elles sont exactes et ne sont pas sujettes à des erreurs numériques, elles sont plus lentes.

Il ne faut pas s'attendre au « miracle » avec un logiciel de calcul formel tel que MAPLE. Ce type de logiciel ne remplacera pas l'étude mathématique en amont de la résolution d'un problème. Par contre, une fois l'étude faite, ce type de logiciel permet de mener à bien une résolution.

Une approche directe et naïve comme nous l'avons entrepris sur l'exemple de la suite montre les limitations des approches purement numériques et purement symboliques (exemple de la fonction `rsolve`). MAPLE, MATHEMATICA ou tout autre système de calcul formel sera toujours limité par la puissance de ses algorithmes.

Calcul symbolique

Un logiciel de calcul formel comme MAPLE est avant tout un formidable outil de manipulation d'*expression*. L'une des applications industrielles d'un tel outil est la génération de code de calcul numérique pour la résolution d'un problème physique : par une manipulation symbolique des équations générales qui régissent ce système physique, et avec la description d'un cas particulier, un tel système est capable de générer un code numérique adapté et plus performant qu'un code numérique pour des problèmes généraux.

Montrons sur quelques exemples les capacités de calcul symbolique de MAPLE. Par exemple, cherchons à simplifier l'expression $\sum_{i=1}^{10} cos(\frac{i\pi}{10})$. Si l'on rentre sous MAPLE cette expression :

```
> sum(cos(i*Pi/10),i=1..10);
```

3. La forme générale est $a_n = \frac{\alpha 100^{n+1} + \beta 6^{n+1} + \gamma 5^{n+1}}{\alpha 100^n + \beta 6^n + \gamma 5^n}$. Les conditions initiales choisies donnent par exemple (α, β et γ étant définies à une constante multiplicative près) $\alpha = 0$, $\beta = \gamma = 1$. De plus, 100 est point fixe stable alors que 6 et 5 sont instables.

on obtient la valeur :

```
                        -1
```

Nous verrons dans le paragraphe 1.4.3 que MAPLE simplifie toutes les expressions qui sont entrées. Il fait même plus que cela : toutes les expressions simplifiées peuvent être mémorisées et réutilisées de manière très efficace. Cela permet d'utiliser moins de mémoire (partage des sous-expressions communes) et, parfois, moins de temps de calcul.

Mais la simplification d'expressions est un problème difficile. Qu'est-ce qu'une forme simplifiée d'une expression ? Cela peut être une forme ayant un nombre minimum de termes. Ou bien, dans le cas d'une fraction rationnelle, sa forme réduite, ou sa décomposition en éléments simples.

Reprenons l'équation qui donne les points fixes de la suite définie auparavant et introduisons un paramètre e : $111 - \frac{1130}{x} + \frac{3000}{x^2} = (x - e)$. Nous allons construire un exemple simple, qui consiste à calculer les solutions de cette équation en fonction du paramètre e, puis ensuite nous fixerons sa valeur à 0. Par continuité des racines de cette nouvelle équation en fonction de e, nous devrions retrouver les solutions précédentes $(100, 5, 6)$.

Comme précédemment, cherchons les racines en fonction de e à l'aide de la fonction `solve`. Le deuxième paramètre de la fonction `solve` permet de préciser par rapport à quelle inconnue (ou variable) on cherche les solutions. Le résultat est stocké dans la variable `r` par l'affectation (symbole `:=`) du résultat de la fonction :

```
> r := solve( 111 -1130/x + 3000/x^2 = (x-e),   x);
```

on obtient alors le résultat :

```
              1/3
       r := %1    - %2 + 37 + 1/3 e,

                   1/3                                  1/2    1/3
           - 1/2 %1    + 1/2 %2 + 37 + 1/3 e + 1/2 I 3    (%1    + %2),

                   1/3                                  1/2    1/3
           - 1/2 %1    + 1/2 %2 + 37 + 1/3 e - 1/2 I 3    (%1    + %2)

                                     2         3
%1 := 31248 + 3542/3 e + 37/3 e  + 1/27 e

                                                 2         3 1/2
     + 5/9 (- 2392347 + 2971926 e + 81573 e  + 360 e )

                                                       2
                         - 2977/3 - 74/3 e - 1/9 e
%2 :=                    -------------------------
                                      1/3
                                    %1
```

Le résultat n'est pas des plus concis : les trois premières lignes décrivent les solutions (séparées par une virgule) de l'équation. Les symboles `%1` et `%2` renvoient à des sous-expressions communes qui sont définies par les affectations `%1 := 31248 ...` (sur deux lignes !) et `%2 := ...`.

Récupérons la troisième solution (on accède, comme dans un tableau, au troisième élément de `r` - on verra plus tard que `r` est une « séquence d'expressions » séparées par des virgules) :

```
> r3 := r[3] ;
```

```
                                                            2
                    1/3          - 2977/3 - 74/3 e - 1/9 e
      r3 := - 1/2 %1     + 1/2 -------------------------- + 37 + 1/3 e
                                             1/3
                                           %1

                          /                                 2\
                     1/2 |  1/3    - 2977/3 - 74/3 e - 1/9 e |
           - 1/2 I 3     |%1     + --------------------------|
                          |                    1/3            |
                          \                  %1               /

                              2          3
%1 := 31248 + 3542/3 e + 37/3 e   + 1/27 e

                                                 2         3 1/2
     + 5/9 (- 2392347 + 2971926 e + 81573 e   + 360 e )
```

et utilisons la fonction **simplify** de MAPLE :

```
> simplify( r3 );
             2/3                    2          1/3          1/3       1/2   2/3
    1/6 (- %1    - 8931 - 222 e - e   + 222 %1    + 2 e %1    - I 3    %1

                  1/2          1/2         1/2  2    /  1/3
         + 8931 I 3    + 222 I 3    e + I 3    e )  /  %1
                                                   /

                             2    3
%1 := 843696 + 31878 e + 333 e   + e

                                                2         3 1/2
     + 15 (- 2392347 + 2971926 e + 81573 e   + 360 e )
```

Il n'est pas évident que le résultat soit plus simple. Nous verrons dans le paragraphe 1.4.3 différentes directives pour aider MAPLE à simplifier dans le sens que nous souhaiterons.

Continuons avec la racine **r3** et fixons la valeur du paramètre e à 0 :

```
> e := 0 ;
                                       e := 0

> print( r3 );

                  1/3      2977                 1/2 /  1/3      2977 \
          - 1/2 %1     - ------- + 37 - 1/2 I 3    |%1     - -------|
                             1/3                    |             1/3|
                         6 %1                       \         3 %1   /

                                                            1/2
%1 :=                        31248 + 5/9 (-2392347)
```

Le résultat devrait être plus simple : une évaluation numérique donne (le caractère " est associé à la valeur de retour de la dernière fonction évaluée par MAPLE) :

```
> evalf(") ;
                                                    -8
                          6.000000000 - .87*10   I
```

Sur ce critère numérique, on peut supposer que `r3` a pour valeur 6, reste à le vérifier formellement [4].

Il semble que la valeur de l'expression de la variable `r3` soit [5] 6. Essayons de simplifier cette expression :

```
> simplify( r3 );
                 2/3                  1/3      1/2   2/3           1/2
               %1    + 8931 - 222 %1    + I 3    %1    - 8931 I 3
         - 1/6 ----------------------------------------------------
                                          1/3
                                        %1

                                            1/2
%1 :=                     843696 + 13395 I 3
```

Le résultat est loin d'être simple. Essayons de comparer cette expression à la valeur 6 (la commande `evalb` permet de forcer l'évaluation d'expression booléenne, ici représentée par le test d'égalité `=`) :

```
> evalb( r3 = 6 );
                     false
```

Que penser de ceci ? MAPLE semble répondre (un peu trop) rapidement, puisqu'en utilisant la fonction `radnormal` de la bibliothèque MAPLE (pour calculer une forme « normale » d'une expression contenant des « radicaux »), on obtient la valeur désirée :

```
> readlib( radnormal ) ; # afin de charger en memoire cette fonction
> radnormal( r3 );
                                      6
```

Enfin, si l'on suppose que MAPLE calcule juste (ce qui n'est pas toujours évident, au regard de la valeur `false` retournée précédemment), alors l'expression de `r3` est bien 6. Comment décider ? La simplification et l'égalité d'expressions sont des problèmes très difficiles (insistons là-dessus !). MAPLE, comme les autres logiciels de calcul formel, est un outil très puissant et en perpétuelle évolution. Il est tout à fait normal qu'il reste quelques erreurs [6]. Moralité : penser à vérifier que toutes les simplifications possibles ont été effectuées avant de se convaincre d'un résultat.

1.2 Présentation de MAPLE

MAPLE a été conçu vers 1980 par des chercheurs de l'université de Waterloo (Canada) : K. Geddes, G. Gonnet, B. Char. L'architecture du système est modulaire, avec un noyau de base et une librairie écrite dans le langage défini par MAPLE. Dans cette section, nous allons présenter le langage offert par MAPLE.

MAPLE est un système interactif. Chaque instruction tapée par l'utilisateur est immédiatement exécutée. Ce fonctionnement est celui d'un « interpréteur ».

4. Des méthodes numériques permettent de décider si la valeur que l'on calcule est bien 6, pour peu que l'on sache calculer une borne sur le nombre de chiffres significatifs qu'il faut utiliser au cours de l'évaluation. En utilisant une « arithmétique d'intervalle », il serait possible d'évaluer cette expression en encadrant le résultat final par des bornes « au pire » : cela ne permet pas de décider mais permet de valider le résultat obtenu.

5. Rappelons que l'exemple a été construit pour qu'il en soit ainsi.

6. Comme dans tout développement logiciel important, il y a toujours des erreurs. L'important est de les cerner, puis de les corriger.

1.2.1 L'interpréteur Maple

Une fois Maple lancé, le logiciel est en attente d'une commande (attente signalée par le caractère >) entrée par l'utilisateur. Une commande peut être une expression arithmétique, un appel de fonction ou une déclaration de fonction. Une fois la commande analysée et les erreurs syntaxiques détectées, Maple l'exécute immédiatement (cf. figure 1.1), affiche le résultat et se remet en attente d'une nouvelle commande.

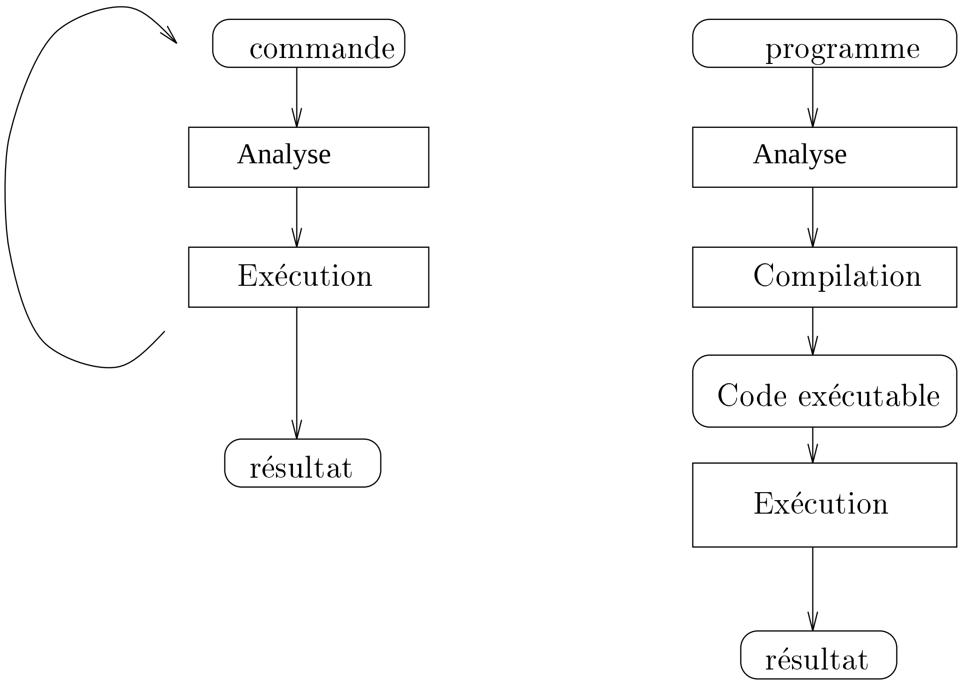


Fig. 1.1 – *Fonctionnement d'un interpréteur par rapport à un compilateur.*

Le langage de Maple est la définition d'une syntaxe (proche de celle de Pascal), de mots-clés (comme les structures « classiques » de contrôle) et d'une sémantique d'évaluation des expressions. Mais c'est aussi un ensemble de structures de données pour la représentation et la manipulation d'objets complexes (cf. paragraphe 1.2.7).

Nous n'approfondirons pas la sémantique de toutes les constructions syntaxiques (voir les manuels de Maple). Généralement il s'agit de la « sémantique mathématique »[7]. Mais nous détaillerons la sémantique d'évaluation de certaines structures de données et surtout la sémantique du passage des paramètres d'une fonction. La définition et l'appel d'une fonction seront étudiés au paragraphe 1.3.

1.2.2 Calculer avec des symboles

En Maple, la déclaration des variables est automatique (sauf pour les tableaux) lors de leur utilisation dans des expressions. Le langage n'est pas typé comme peut l'être Pascal, mais chaque expression a un type interne associé à sa structure de données. Nous étudierons les conversions entre

7. ou « sémantique du bon sens »

structures de données au paragraphe 1.4, nous verrons aussi les différentes fonctions de simplification et de manipulation associées. Leur représentation interne sera abordée dans le paragraphe 1.4.4.

L'évaluation d'une variable non associée à une valeur est la chaîne de caractères qui la représente. On parle alors d'un *symbole*. Dès que l'on associe une expression à une variable (en l'affectant), l'évaluation de celle-ci est l'évaluation de l'expression qui lui est affectée (le symbole est alors attaché à une valeur). Une variable peut être dé-assignée, son évaluation est alors la chaîne de caractères qui la définit.

```
> a := x + y ;
                                  a := x + y

>  x := 2 ; # association a x d'une valeur

                                    x := 2

> a ;  # evaluation de a
                                    2 + y
> readlib ( unassign ) ; # comme pour la fonction radnormal de l'exemple au-dessus
> unassign( 'x' );
> a;
                                    x + y
```

La fonction `unassign` est une fonction qui n'est pas automatiquement chargée (cf. fonctions de la bibliothèque). Pour l'utiliser il faut entrer `readlib( unassign )` . Voir le paragraphe 1.5 concernant la bibliothèque Maple.

On peut aussi dé-assigner une variable en l'affectant à son symbole (son nom) ! En effet, l'affectation `a := <expr>` provoque l'évaluation de l'expression `expr` puis son affectation à la variable `a`. Si l'on entre `a := a`, alors que `a` est déjà affectée, cela ne marche pas puisque l'évaluation du `a` à droite de l'affectation retourne sa valeur qui est ré-affectée à la variable `a`. En Maple il existe un opérateur qui permet de bloquer une (et une seule) évaluation d'expression : il suffit d'entrer une expression entre les symboles « `'` » (*quote*, ou apostrophe).

```
> x:=2 ;
                                    x := 2
> y:=3;
                                    y := 3
> a := x+y ;

                                    a := 5

> a := 'x+y'; # on differe l'evaluation de x+y qui devrait etre 5

                                  a := x + y

> a;  # l'evaluation de a est l'evaluation de l'expression qui lui est associee

                                      5
> x := 'x' : # de-affectation de x
> a ;        # evaluation de a

                                    x + 3
```

Ce mécanisme, qui consiste à manipuler des symboles et éventuellement leur attacher des expressions, couplé à une sémantique d'évaluation, est à la base du calcul symbolique. Ce mécanisme est aussi à la base du langage Lisp.

1.2.3 Précisions sur la syntaxe

Constantes et chaînes de caractères

Les constantes MAPLE sont les constantes numériques et les chaînes de caractères. Une constante numérique est un entier, un rationnel ou un nombre réel.

Exemple :

```
> 12345678910111213141516171819202122232425262728 ;  # c'est un entier

      12345678910111213141516171819202122232425262728

> - " / 345 ;  # formation d'un rationnel a partir du resultat precedent (symbole ")

       12345678910111213141516171819202122232425262728
     - -----------------------------------------------
                              345
> b := `ceci est une chaine de caracteres`;

                     b := ceci est une chaine de caracteres
```

Les chaînes de caractères sont séparées par le caractère « ` » (attention ce n'est pas le même symbole que le quote « ' » du paragraphe 1.2.2). On peut concaténer deux chaînes de caractères avec la fonction `cat` : `cat(`qwerty`, `azerty`);` retourne la chaîne de caractères `` `qwertyazerty` ``.

Le nombre de chiffres de ces nombres est arbitraire. Par défaut MAPLE calcule avec des nombres réels sur 10 chiffres significatifs. Pour changer ce nombre, il suffit d'affecter une nouvelle valeur à la variable (globale) `Digits`.

Exemple :

```
> Digits ;   # Evaluation de la variable `Digits`

                              10

> evalf(Pi); # Pi : constante predefinie en Maple

                         3.141592654

> Digits := 30;
                         Digits := 30

> evalf(Pi);

               3.14159265358979323846264338328
```

La table suivante résume les principales constantes mathématiques pré-définies sous MAPLE :

constantes	nom sous MAPLE
vrai, faux	`true, false`
π	`Pi`
e	`E`
$i = \sqrt{-1}$	`I, (-1)^(1/2)`
∞	`infinity`
γ (constante d'Euler)	`gamma`

TABLE 1.2.5 Principales constantes mathématiques pré-définies.

Les variables

Les variables sont des suites de caractères commençant par une lettre ou le caractère _ (souligné), suivi de lettres, du caractère _ ou de chiffres.

Exemple :

```
> une_variable ;
                              une_variable
> _a12;
                                  _a12
> _ := 4;
                                 _ := 4
```

Remarque : les noms des variables propres à MAPLE commençant par un caractère souligné (_), il est conseillé de ne pas déclarer ses variables par des noms qui commencent par ce caractère.

Toutes les instructions MAPLE retournent une expression qui est leur « évaluation » (cf. paragraphe 1.4.1). Comme le montre l'exemple ci-dessus, l'évaluation d'une variable non affectée est la variable elle-même. Par exemple :

```
i> une_chaine := `ceci est une chaine de caracteres` ;   # une affectation

                   une_chaine := ceci est une chaine de caracteres

> a := 123*x +2 ; # une autre affectation

                             a := 123 x + 2

> a + une_chaine ;   # une addition

                   123 x + 2 + ceci est une chaine de caracteres
```

Peu importe le sens des expressions manipulées, la construction de la dernière expression est syntaxique. Ce problème est caractéristique de deux conceptions d'un langage : pouvoir d'expressivité (comme MAPLE) et fiabilité des constructions syntaxiques (par exemple, Pascal). C'est ce qui fait la « facilité d'utilisation » de MAPLE, mais aussi sa faiblesse dans la conception de fonctions de bibliothèques.

Séparateur d'instructions, caractère ", commentaire

Néanmoins, il est possible de contrôler la valeur retournée par une expression. Suivant le caractère de terminaison d'une instruction, le comportement est différent :

« `inst1 ;` » : l'expression associée à l'instruction `inst1` est affichée à l'écran, comme ci-dessus.

« `inst2 :` » : l'expression n'est pas affichée à l'écran :

```
> b:=12222112:
> # pas d'affichage du resultat
```

MAPLE mémorise dans une variable spéciale la dernière expression évaluée. On y accède par le caractère « " » (guillemet) :

```
> 10!;   # '!' caractere de la factorielle
```

```
                                           3628800
> ifactor( " );  # factorisation de l'entier 3628800
                                       8    4    2
                                    (2)  (3)  (5)  (7)
> deux := 2:
> " ;         # re-evaluation du dernier resultat
                                             2
```

De même, les chaînes de caractères « `""` » (deux guillemets) et « `"""` » (trois guillemets) correspondent aux pénultième et antépénultième résultats.

Le caractère `#` (dièse) marque le début d'un commentaire MAPLE qui se termine à la fin de la ligne.

1.2.4 Les opérateurs

L'opérateur le plus important est l'opérateur d'affectation : `var := expression` . Nous l'avons déjà rencontré et il n'y a rien à ajouter si ce n'est que l'opérande de droite (`expression`) est évalué avant affectation à la variable `var`.

MAPLE définit un ensemble d'opérateurs arithmétiques standards (voir table 1.2.6). Ces opérateurs acceptent des arguments numériques (entiers, réels,...), mais aussi des expressions plus générales. La précédence des ces opérateurs correspond à la précédence des « standards » : « `a + b * c` » est compris comme `a + (b * c)`. En cas d'ambiguïté il est nécessaire de forcer l'associativité des opérateurs par l'utilisation de parenthèses. Cet opérateur `*` de multiplication est commutatif. Nous verrons plus tard le symbole de l'opérateur de la multiplication matricielle.

opérateur	symbole	exemple
addition	`+`	`2 + 3`
soustraction	`-`	`-2 - 3`
multiplication	`*`	`x * y * 5`
division	`/`	`x / 123`
exponentiation	`^` ou `**`	`10^(-5)`
factorielle	`!`	`123!`
valeur absolue	`abs(...)`	`abs( -521.9 )`
arrondi à l'entier le plus proche	`round(...)`	`round( -521.2 )`
plus petit entier plus grand que `a`	`ceil( a )`	`ceil( -521.2 )`
plus grand entier plus petit que `a`	`floor( a )`	`floor( -521.2 )`
arrondi vers 0	`trunc(...)`	`trunc( -521.2 )`

TABLE 1.2.6 Sommaire des opérateurs arithmétiques.

Dans le cas des arguments non numériques, l'évaluation de ces opérateurs est, généralement, la fonction en entrée (comme dans l'exemple de l'affectation de l'`expr` ci-dessous).

Exemple :

Outre les éventuelles simplifications, on remarquera que MAPLE affiche différemment les expressions entrées : le symbole de multiplication est supprimé.

```
> x+y * z ;   # construction d'une expression polynomiale
                        x + y z
```

```
>  expr:= ceil( " - x^2 ) ;

                                             2
                expr := ceil(x + y z - x )

> x:= Pi ; y := exp(1) ; z := 0.1 ;

                         x := Pi

                       y := exp(1)

                         z := .1

>  print(expr) ;
                            -6
```

En plus de ces opérateurs « arithmétiques », MAPLE définit (table 1.2.7) des opérateurs relationnels et booléens.

opérateur	symbole	exemple
égalité	`=`	`2 = 3`
différent de	`<>`	`x <> y+2`
inférieur à	`<`	`x^2  < 5`
supérieur à	`>`	`x > 123`
inférieur ou égal à	`<=`	`-10 <= 5`
supérieur ou égal à	`>=`	`-10 >= 5`
« et » logique	`and`	`(a < b) and (c = 0)`
« ou » inclusif logique	`or`	`(a < b) or (c = 0)`
négation logique	`not`	`not ( (a < b) or (c = 0) )`

TABLE 1.2.7 Sommaire des opérateurs relationnels et booléens.

Les valeurs booléennes `false` (faux) et `true` (vrai) sont retournées par ces opérateurs logiques. La précédence des opérateurs logiques est plus faible que la précédence des opérateurs relationnels (*i.e.* le test `a < b and c = 0` sera compris comme `(a < b) and (c = 0)`), mais afin d'éviter toute ambiguïté de lecture, il est conseillé de mettre entre parenthèses ce type d'expressions.

1.2.5 Les instructions

L'itération

L'itération en MAPLE (l'équivalent des structures de contrôle `repeat until`, `while` et `for` de Pascal) se présente sous la forme suivante :

$$\textbf{for}\ \text{variable} \left\{ \begin{array}{l} \textbf{in}\ \text{expression} \\ \textbf{from}\ \text{valeur1}\ [\ \textbf{by}\ \text{pas}\ ]\ \textbf{to}\ \text{valeur2} \end{array} \right. \textbf{while}\ \text{condition}\ \textbf{do}\ \text{instructions}\ \textbf{od}$$

À part le bloc d'instructions **do** *instructions* **od** tout le reste est facultatif ! La table 1.2.8 montre les constructions équivalentes entre MAPLE et Pascal.

MAPLE	Pascal
for i **from** 1 **to** n **do** ... **od**	**for** i :=1 **to** n **do begin** ... **end**
while condition **do** ... **od**	**while** condition **do begin** ... **end**
do ... **if** cond **then break** (ou **exit**) **fi od**	**repeat** ... **until** cond

TABLE 1.2.8 Équivalence entre MAPLE et Pascal des instructions d'itérations.

Montrons sur des exemples les utilisations les plus courantes.

Exemple :

Calcul de la somme des carrés des termes de la somme `p` suivante.

```
> p := x+y+z;

                          p := x + y + z

> s:=0 :  for i in p do s := s + i^2 od : s ;

                            2    2    2
                           x  + y  + z
```

Exemple :

Remplissons une matrice d'ordre 4 par des valeurs aléatoires.

```
> A := array(1..4,1..4) : # declaration d'un tableau de dimension 4x4
> for i from 1 to 4  do
>   for j from 1 to 4 do
>     A[i,j] := rand();       # fonction qui retourne aleatoirement un entier
>   od;
> od ;
> print (A);

           [  32062222085  722974121768  604305613921  745580037409 ]
           [                                                         ]
           [ 259811952655  310075487163  797179490457   39169594160 ]
           [                                                         ]
           [  88430571674  960498834085  812920457916  453747019461 ]
           [                                                         ]
           [ 644031395307  920624947349  951053530086  146486307198 ]
```

1.2.6 Instruction conditionnelle

Comme Pascal, MAPLE possède des instructions conditionnelles. Leur syntaxe est donnée par (les crochets indiquent une construction facultative) :

if cond1 **then** inst1 [**elif** cond2 **then** inst2] [**else** int3] **fi**

La construction syntaxique **elif** permet d'emboîter des instructions conditionnelles. On peut en emboîter autant qu'on le désire (ou plutôt autant que nécessaire). Par exemple :

```
> if var = 2 then print(`var vaut 2`) else print(`var ne vaut pas 2`); fi;

                                    var ne vaut pas 2
> var := 2 ;

                                        var := 2

> if var = 2 then print(`var vaut 2`) else print(`var ne vaut pas 2`); fi;

                                       var vaut 2

> # Calcul de la somme des inverses des nombres premiers inferieurs a 10
> s := 0 :
> for i from 1 to 10 do  if isprime(i) then s := s + 1/i  fi od : s ;

                                     247
                                     ---
                                     210
```

1.2.7 Les structures de données

Séquence d'expressions

Une *séquence* d'expressions en MAPLE est une liste ordonnée d'objets séparés par des virgules (`,`), comme par exemple `a1, a2, a3`. La séquence vide est représentée par le symbole `NULL`.

On peut concaténer des séquences par l'opérateur « `,` », par exemple si `seq1` est définie par :

```
> seq1 := a,b,c,d ;
                     seq1 := a, b, c, d
```

On peut construire `seq2` comme suit :

```
> seq2 := seq1, x, y, d ;

                seq2 := a, b, c, d, x, y, d
```

Lors de l'évaluation de l'opérande de droite de l'affectation, la variable `seq1` est évaluée et concaténée à la séquence `x, y, d`. On remarquera qu'une séquence peut contenir des doublons (`d` dans l'exemple ci-dessus).

La fonction `seq` de MAPLE permet de construire des séquences qui dépendent d'un indice.

Exemple :

Construisons la suite des i^3 pour i allant de 1 à 5, et sommons les termes en utilisant l'instruction `for` vue précédemment :

```
> seq3 := seq( i^3 , i=1..10 ) ;

              seq3 := 1, 8, 27, 64, 125, 216, 343, 512, 729, 1000

>  s := 0 : for j in seq3 do  s := s + j : od :  s ;

                                     3025
```

Une séquence d'expressions peut contenir n'importe quelle expression MAPLE, par exemple, construisons les neuf monômes $x^i y^j$ pour $i, j \in \{0, 1, 2\}$.

Exemple :

```
> seq4 := seq ( seq ( x^i * y^j, i=0..2), j=0..2);

                             2            2      2      2   2  2
                seq4 := 1, x, x , y, x y, x  y, y , x y , x  y
```

On peut accéder au i-ème élément d'une séquence par l'opérateur d'indexation `[i]`, comme par exemple `seq4[4]` qui est `y`. Cet opérateur n'est à utiliser qu'en lecture, pour modifier un élément d'une séquence il faut utiliser la fonction `subsop` (cf. paragraphe 1.4).

Les ensembles

Un *ensemble* MAPLE est défini par une séquence d'expressions entourée d'accolades : `{a, b, c, d}`. On peut définir des ensembles d'ensembles, alors qu'une séquence de séquences est une séquence « à plat » : c'est la séquence des éléments de chacune des séquences.

Un ensemble est une séquence d'objets sans doublon (à la différence des séquences). Un ordre, donné par le système, est défini sur toutes les listes. Mais celui-ci peut, pour un même ensemble, varier d'une session de travail à une autre. Cela est dû à la représentation interne d'un ensemble sous MAPLE, aussi ne faut-il pas s'y fier. Par exemple, lors d'une session un ensemble `ens` pourra être représenté par `{x,y,z}` et lors d'une autre session par `{x,z,y}`.

Exemple :

```
> ens1 := {a, b, c, d} ;
                          ens1 := {a, b, c, d}

> ens2 := {a, b, c, d, d};
                          ens2 := {a, b, c, d}
```

Les opérations ensemblistes sont :

`a union b` : union des deux ensembles `a` et `b`, *i.e.* $a \cup b$,

`a intersect b` : intersection des deux ensembles `a` et `b`, *i.e.* $a \cap b$,

`a minus b` : différence des deux ensembles `a` et `b`, *i.e.* $a \setminus b$.

Remarque : les opérateurs `union` et `intersect` peuvent être n-aires. Il faut alors les écrire sous la forme : `'union' (s1, s2, s3, ..., sn)`

Exemple :

Si nous reprenons les deux variables `ens1` et `ens2` précédemment définies :

```
> ens3 :=  ens1 union {x, y, z} ;

                       ens3 := {a, x, y, z, b, c, d}
> ens3 union ens1 ;
                           {a, x, y, z, b, c, d}
> ens3 intersect ens1 ;
                               {a, b, c, d}
> ens3 minus ens1;
                                {x, y, z}
```

De même que pour les séquences, on peut accéder, en lecture seulement, à un i-ème élément d'un ensemble, par exemple `ens3[4]` retourne le symbole `z`. Voir le paragraphe 1.4 pour modifier un élément d'un ensemble par la fonction `subsop`.

Les listes

Une *liste* en MAPLE est une suite ordonnée d'objets entourée de crochets, par exemple `[a,b,c,d]`. À la différence des ensembles l'ordre des éléments est respecté.

Les listes sont proches à la fois des séquences et des ensembles. Elles permettent de représenter des listes d'objets avec des doublons et de manière récursive (liste de listes). En effet, on peut définir des listes de listes, comme pour les ensembles, alors qu'on ne le peut pas avec des séquences.

Exemple :

Définissons des listes de la même manière que le premier exemple des séquences.

```
> liste1 := [a,b,c,d] ;

             liste1 := [a, b, c, d]

> liste2 := [liste1, x,y,d] ;

            liste2 := [[a, b, c, d], x, y, d]

> liste2[1] ;

                       [a, b, c, d]
> liste2[2] ;
                            x
```

Comme pour les séquences et les ensembles, on ne peut pas changer un élément d'une liste en utilisant l'opérateur d'indexation (`liste[3] := ...`). Nous verrons au paragraphe 1.4 comment modifier des listes.

Les tables

Une table sous MAPLE est une collection d'informations similaire au type structuré (`record`) de Pascal. On appelle aussi ce type de structure de données des « *tableaux associatifs* ». Chaque information est accessible grâce à une clé de recherche (l'indice du tableau). Par exemple, on peut définir une table qui associe un nombre à chaque couleur :

```
>  code_couleur := table( [rouge=1, blanc= -2, noir = 0, vert = 3] ) ;

                             code_couleur := table([
                                                   rouge = 1
                                                   blanc = -2
                                                   noir = 0
                                                   vert = 3
                                               ])

> code_couleur[blanc] ;

                                             -2
```

L'opérateur d'indexation `table [ index ]` permet de lire une entrée de la table, mais aussi de la modifier. On peut ajouter des éléments dans la table par une affectation d'une nouvelle entrée :

```
> code_couleur[ beige ] := 1/2 * Pi ;

                              code_couleur[beige] := 1/2 Pi

>  print( code_couleur );

                                   table([
                                       rouge = 1
                                       blanc = -2
                                       noir = 0
                                       vert = 3
                                       beige = 1/2 Pi
                                   ])
```

Les éléments et les indices peuvent être de types différents (nombre, variable, expression).

```
> code_couleur[x+y] := x->sin(x) ;

                          code_couleur[x + y] := sin

> print(code_couleur);
                              table([
                                  rouge = 1
                                  blanc = -2
                                  noir = 0
                                  vert = 3
                                  beige = 1/2 Pi
                                  x + y = sin
                              ])
> code_couleur[x+y];
                                     sin
```

Pour supprimer un élément d'une table il faut supprimer l'expression associée à l'entrée correspondante. Pour cela on peut utiliser la fonction `unassign` vue précédemment (ne pas oublier les apostrophes afin que l'argument ne soit pas évalué, cf. paragraphe 1.3) :

```
> readlib ( unassign ) :
> unassign( 'code_couleur[rouge]' );
> print( code_couleur ) ;

                                   table([
                                       blanc = -2
                                       noir = 0
                                       vert = 3
                                       beige = 1/2 Pi
                                   ])
```

Les tableaux

Les tableaux sont similaires aux tables, excepté que leurs indices sont des entiers compris dans un intervalle. La déclaration des tableaux se fait en précisant cet intervalle.

```
> vect := array ( -1..1 ); # tableau ayant les indices -1, 0 et 1
```

```
                    vect := array(-1 .. 1, [])

> vect[-1] := 0 : vect[1] := 1 : print(vect) ;

                         array(-1 .. 1, [
                             -1 = 0
                             0 = vect[0]
                             1 = 1
                         ])
```

On peut définir des tableaux avec un nombre quelconque de dimensions, chacune d'entre elles étant associée à un intervalle d'indices. L'accès à un élément d'un tableau se fait par l'opérateur d'indexation `tableau [i1, i2, ..., in]`. L'exemple ci-dessus montre une modification des éléments d'indices -1 et 1 de la variable `vect`. L'élément d'indice 0 n'est pas modifié et est affiché sous sa forme « littérale ».

Remarque importante. La sémantique d'affectation de deux tableaux « `A := B;` » est propre à MAPLE. Si on modifie le tableau `A`, la modification est reportée dans `B`. En fait, les deux tableaux partagent le même espace mémoire. L'affectation ne crée pas une copie du tableau `B`. Pour recopier le tableau `B` et stocker la copie dans la variable `A` il faut écrire : `A := copy(B)`. Cela est aussi vrai pour les matrices et les vecteurs.

Les matrices et les vecteurs

Les tableaux sont la structure de données privilégiée pour toutes les fonctions d'algèbre linéaire de la bibliothèque `linalg`. On peut préciser la structure du tableau (creux, symétrique, anti-symétrique). Les matrices et les vecteurs sont des tableaux dont les indices commencent à 1. Pour les déclarer, il existe des fonctions prédéfinies :

```
> with(linalg) :  # utilisation de la bibliotheque des fonctions d'algebre lineaire
Warning: new definition for   norm
Warning: new definition for   trace

> A := matrix(3,3) ;  # creation d'une matrice de taille 3x3

                     A := array(1 .. 3, 1 .. 3, [])

> V := randvector(3); # creation d'un vecteur aleatoire de taille 3

                         V := [ -85, -55, -37 ]
```

Les opérations arithmétiques entre matrices et vecteurs se présentent sous deux formes : fonctionnelle (par exemple `add (A, B)`, `multiply (A, V)`), ou opérationnelle (`A + B` pour l'addition, `A &* B` pour la multiplication). Le symbole « `&*` » utilisé pour ce dernier opérateur est assez particulier et n'a pas d'explication autre qu'historique, il rappelle qu'il s'agit d'une multiplication non commutative.

Toutes les opérations arithmétiques entre matrices ne sont pas évaluées. Si l'on veut le résultat d'une expression arithmétique contenant des matrices, il faut forcer l'évaluation par la fonction `evalm` (pour « évaluation sur des matrices »).

```
> A := randmatrix(2,2) ;B := randmatrix(2,2);
```

```
                                   [ -62    1  ]
                              A := [           ]
                                   [ -47   -91 ]

                                   [ -47   -61 ]
                              B := [           ]
                                   [  41   -58 ]
>  A &* B ;

                                      A &* B
>  evalm( " );

                                [  2955  3724 ]
                                [             ]
                                [ -1522  8145 ]
```

La bibliothèque d'algèbre linéaire contient de nombreuses fonctions qui seront décrites au paragraphe 1.5.3.

1.3 Les fonctions

1.3.1 Déclaration

La déclaration d'une fonction a la forme suivante :

fonc := **proc** (paramètres)

local variables locales ;

option liste d'options ;

instructions (corps de la fonction)

end

Les « variables locales » sont des noms de variables séparés par des virgules. Un seul mot clé `local` est autorisé. Les paramètres de la fonction permettent le passage d'expressions en arguments et en retour. Toutes les fonctions retournent une expression (éventuellement vide). Les options ne seront pas abordées ici et nous laissons au lecteur le soin de consulter le manuel de référence du langage MAPLE.

Par exemple, la fonction $x \mapsto x^2$ peut être affectée à la variable `sqr` (pour *square*, carré) et réutilisée sous ce nom :

```
> sqr := proc(x) x^2  end ;
                                                  2
                                      sqr := x -> x
> x := 2 :  sqr(x+y+z);

                                                 2
                                       (2 + y + z)
```

Dans la définition de la fonction, la variable `x` joue le rôle d'un paramètre formel. Lors de l'appel d'une fonction de la forme `f ( args )`, le nom `f` est évalué, ensuite les arguments le sont à leur tour. Ils sont alors associés aux paramètres formels de la fonction. Enfin les instructions du corps de la fonction sont évaluées et la dernière évaluation est associée à la valeur de retour de la fonction. La fonction renvoyant comme résultat la valeur de cette dernière instruction, il importe de bien placer à la fin de la fonction le résultat désiré.

Dans l'exemple précédent, la définition de la fonction `sqr` est rappelée (évaluée), puis l'argument `x+y+z` de `sqr` est évalué, ce qui donne `2+y+z`, qui est ensuite associé au paramètre formel `x` de `sqr` (ce paramètre formel `x` n'est pas confondu par MAPLE avec la variable globale, `x`, qui est affectée à 2). Le corps de la fonction `sqr` ne comporte qu'une instruction qui élève au carré le paramètre formel `x`. C'est cette dernière expression qui constitue la valeur de retour de la fonction.

Exemple :

La fonction suivante `fibo` retourne le n-ième nombre de Fibonacci.

```
fibo := proc (n)  # definition de(s) parametre(s) formel(s)
  if n = 0 then  1 ;
  elif n = 1 then 1 ;
  else fibo(n-1) + fibo(n-2) ;
  fi ;
end;                # la valeur de retour sera la derniere expression
                    # evaluee (0 ou 1)
```

Exercice. L'écriture récursive de la fonction `fibo` engendre un nombre exponentiel d'appels récursifs. Modifiez l'exemple afin d'écrire une fonction utilisant une instruction d'itération.

Exercice. Produit de deux matrices par la méthode de Strassen-Winograd. Soient en entrée deux matrices A et B, écrivons une fonction qui retourne le produit $A.B$ par la méthode rapide de Strassen-Winograd (1973). La complexité de cet algorithme[8] est $n^{\log_2 7}$ multiplications et $5n^{\log_2 7} - 5n^2$ additions, avec $\log_2 7 \approx 2.807$. En comparaison la méthode classique requiert n^3 multiplications et $n^3 - n^2$ additions. La méthode consiste en la formulation suivante :

$$C = AB = \begin{bmatrix} M_0 + M_1 & M_0 + M_4 + M_5 - M_6 \\ M_0 - M_2 + M_3 - M_6 & M_0 + M_3 + M_4 - M_6 \end{bmatrix}$$

$$\begin{array}{rcrl} M_0 & = & A_{1,1} & B_{1,1} \\ M_1 & = & A_{1,2} & B_{2,1} \\ M_2 & = & A_{2,2} & (B_{1,1} - B_{1,2} - B_{2,1} + B_{2,2}) \\ M_3 & = & (A_{1,1} - A_{2,1}) & (B_{2,2} - B_{1,2}) \\ M_4 & = & (A_{2,1} + A_{2,2}) & (B_{1,2} - B_{1,1}) \\ M_5 & = & (A_{1.1} + A_{1,2} - A_{2,1} - A_{2,2}) & B_{2,2} \\ M_6 & = & (A_{1,1} - A_{2,1} - A_{2,2}) & (B_{1,1} + B_{2,2} - B_{1,2}) \end{array}$$

L'exercice consiste en l'écriture d'une fonction récursive qui réalise cette méthode. On considérera que les matrices en entrées sont carrées et de taille une puissance de 2. Évaluez et discutez la performance de votre algorithme par rapport à l'algorithme de la fonction `multiply` de la bibliothèque d'algèbre linéaire. (On pourra se servir des fonctions `concat` et `stack` de la bibliothèque d'algèbre linéaire).

8. Le premier algorithme rapide est dû à Strassen (1969). Il est de complexité $n^{\log_2 7}$ multiplications et $6n^{\log_2 7} - 6n^2$ additions.

1.3.2 Complément

Les fonctions MAPLE peuvent avoir un nombre quelconque d'arguments. Les variables suivantes, qui sont accessibles dans le corps de toute fonction, permettent de les manipuler.

procname : la valeur de cette variable est le nom de la fonction.

nargs : nombre d'arguments lors de l'appel à la fonction.

args : séquence des `nargs` arguments d'appel.

Notons qu'avec ces variables, on pourrait définir des fonctions avec aucun paramètre formel (ce n'est pas conseillé). On peut aussi retourner l'appel (fonction + arguments) si une opération de la fonction n'est pas possible suivant les arguments présents (comme la fonction `rsolve` dans l'exemple introductif, page 11). Une telle fonction est de la forme suivante :

```
rsolve := proc ( .... ) # les parametres eventuels
local  pas_possible,    # variable locale booleenne qui vaut true si la
                        # resolution n'est pas possible
       ... ;            # autres variables locales
...
...                     # algorithme de resolution
...
  if pas_possible
  then
    RETURN ( 'procname(args)' ) ;
  fi ;
...                     # suite de l'algorithme
end:
```

L'instruction `RETURN` permet de sortir prématurément de la fonction. L'expression en argument de `RETURN` est entre apostrophes pour qu'elle ne soit pas évaluée, sinon son évaluation engendre un appel récursif à `rsolve` qu'il n'est plus possible d'arrêter (en fait le programme MAPLE provoque une erreur telle que le système d'exploitation de la machine l'arrête).

Exemple :

Écrivons une fonction, à nombre quelconque d'arguments, qui calcule le maximum des éléments passés en argument.

```
maxi := proc()        # aucun parametre n'est declare
  local result ,      # contient le resultat
        i ;           # variable temporaire
  if nargs =0 then ERROR (`et les arguments ? `) ;
  else
    result := args[1] ;
    for i in nargs
    do
       # on pourrait rajouter un test sur le type des objets
       # afin de verifier que la comparaison est possible
       if result < args[i] then result := args[i] fi ;
    od ;
    result ;  # evaluation de la variable locale qui sera
              # la valeur de retour de la fonction.
  fi ;
end:
```

Le comportement de cette fonction est :

```
> maxi(1,2,3,3,1,-2,9);

                              9
> maxi();
Error, (in maxi) et les arguments ?
> maxi(a,43,2);
Error, (in maxi) cannot evaluate boolean
```

Exercice. Rajoutez l'idée précisée dans le commentaire de la fonction précédente afin de vérifier le type des arguments. On pourra se servir de la fonction `type` et ne considérer que les types numériques (regarder la description sous l'interpréteur par la commande `? type[numeric]`).

1.4 Manipulation des expressions

Toutes les constructions syntaxiques de MAPLE sont des expressions. Une expression est un graphe sans cycle. Chaque nœud du graphe est étiqueté. Les étiquettes sont de deux natures : les constantes ou les opérateurs (les opérateurs arithmétiques, ou l'opérateur « appel à une fonction »...). À chacun des nœuds est associé un type. Mais MAPLE n'est pas un langage typé dans le sens où aucune déclaration (sauf les tableaux) ou vérification de type n'est faite par l'interpréteur. Si l'on effectue une opération ensembliste sur des listes une erreur sera générée, mais c'est le code de la fonction associée à l'opérateur, et non l'interpréteur MAPLE, qui la génère.

Nous verrons plus en détail la représentation des expressions en MAPLE au paragraphe 1.4.4. Dans cette partie nous présenterons sommairement les principales fonctions de manipulation des expressions.

1.4.1 Évaluation

Il existe plusieurs fonctions qui permettent d'évaluer explicitement des expressions dans des contextes différents. Cette description sommaire ne remplace en rien le manuel de référence de MAPLE.

– `eval ( e, niv )` : évaluation explicite. En utilisant cette fonction, toutes les variables globales de l'expression `e` sont évaluées. Les variables locales d'une procédure sont évaluées à 1 niveau seulement. Par exemple, sous l'interpréteur (*i.e.* pas lors de l'exécution d'une fonction), considérons la suite d'affectations suivante :

```
> a := b ; b := c ; c := d ; d := 1 ;

                           a := b

                           b := c

                           c := d

                           d := 1
> a;
                             1
> eval(a);
                             1
> eval(a,1);
                             b
```

```
> eval(a,2);
                                  c
```

Sous l'interpréteur, l'instruction `eval( a );` est similaire à `a;` . Dans une fonction où `a` est une variable locale, l'instruction `a ;` est synonyme de `eval(a, 1)`. Il est parfois nécessaire de rajouter dans une fonction l'instruction `eval( a );`. Une certaine expérience de MAPLE est nécessaire pour assimiler ces différences.

- `evalb ( e ) ;` : force l'évaluation d'une expression booléenne. Par exemple si l'on entre `x = x;` sous l'interpréteur, aucune simplification n'est faite pour retourner `true`. Si l'on veut une telle réponse, il est alors nécessaire forcer l'évaluation par la fonction `evalb`.
- `evalc ( e ) ;` : essaie de décomposer les nombres complexes apparaissant dans une expression en leurs parties réelles et imaginaires. Cet évaluateur contient une « base de données » sur des décompositions de fonctions classiques. On peut rajouter des fonctions dans cette « base ». Par exemple `evalc( exp( I ) )` retourne `cos(1) + I sin(1)`.
- `evalf ( e ) ;` : retourne une approximation numérique de l'expression `e` en tenant compte de la valeur de la variable `Digits` pour le nombre de chiffres significatifs.
- `evalhf ( e ) ;` ; retourne une approximation numérique en utilisant les fonctions numériques du processeur sur lequel fonctionne MAPLE. Cette fonction est dépendante de la machine. Généralement la précision est d'environ 15 chiffres significatifs. À noter que les calculs numériques par cette fonction sont, pour la même précision, beaucoup plus rapides que par la fonction `evalf`.
- `evalm ( e ) ;` : évalue l'expression `e` contenant des opérateurs entre matrices (comme `&*`). Voir le paragraphe sur les structures de données des matrices et vecteurs (cf. page 26).

Nous ne parlerons pas des autres évaluateurs comme `evaln`, `evala` et renvoyons le lecteur aux aides en ligne correspondantes (`?evaln`, et `?evala` sous l'interpréteur).

1.4.2 Conversion des structures de données

Certaines fonctions de la bibliothèque de MAPLE retournent des structures de données qui peuvent être incompatibles avec leurs utilisations ultérieures. Il s'agit alors de convertir une structure de données en une autre.

La fonction réalisant ceci sous MAPLE est la fonction `convert ( exp, type, opts) ;` : elle essaie de convertir l'expression `expr` dans la structure de données `type`, des arguments optionnels `opts` permettent de spécifier plus finement la conversion. Par exemple :

```
> convert( 1.23456, fraction );
                                    3858
                                    ----
                                    3125
> s := series(exp(x),x,4);
                                     2         3       4
                      s := 1 + x + 1/2 x  + 1/6 x  + O(x )

> convert(s,polynom); # supprime le terme O(..)
                                     2         3
                           1 + x + 1/2 x  + 1/6 x
```

Le second argument de la fonction est obligatoirement l'un des types suivants :

`+`	`*`	D	base	binary	binomial
confrac	decimal	degrees	diff	double	equality
exp	expln	expsincos	factorial	float	fraction
GAMMA	hex	horner	hostfile	hypergeom	lessthan
lessequal	list	listlist	ln	matrix	metric
mod2	multiset	name	octal	parfrac	polar
polynom	radians	radical	rational	ratpoly	RootOf
set	sincos	sqrfree	tan	trig	vector

La richesse de cette fonction est telle que nous laissons au lecteur le soin de regarder les descriptions correspondantes. Par exemple, pour avoir la description de la conversion vers le type `rational`, il suffit de taper :

```
> ? convert[ rational ]
```

1.4.3 Simplification

La fonction simplify

La simplification de base d'une expression ne suffit pas toujours. Dans les exemples suivants, Maple retourne des expressions qui valent zéro :

```
> zero1 := 4^(1/2) - 2;
                                            1/2
                                  zero1 := 4    - 2

> zero2 := sin(x)^2 + cos(x)^2 - 1;

                                           2         2
                            zero2 := sin(x)  + cos(x)  - 1

> zero3 := (f(x)^2-1)/(f(x)-1) - f(x) -1 ;

                                         2
                                     f(x)  - 1
                            zero3 := --------- - f(x) - 1
                                      f(x) - 1
```

La fonction Maple `simplify` effectue des simplifications plus évoluées, avec au besoin un argument supplémentaire précisant le domaine de simplification (`trig,sqrt,exp,powr,...`). Sur les exemples ci-dessus :

```
> simplify( zero1 );

                           0
> simplify( zero2, trig );

                           0
> simplify( zero3 );

                           0
```

Sans le second argument, Maple effectue une recherche du type de domaine à considérer. Lors d'une simplification, on peut préciser des contraintes sur les variables. Par exemple simplifions :

```
> (x^2)^(1/2) ;
```

```
                                 2 1/2
                               (x )
> simplify( " );
                              csgn(x) x

> simplify( ", assume = positive );

                                  x
```

À noter que dans les premières versions de MAPLE, la première simplification de l'exemple retournait `x`.

Mettre sous une forme normale une expression : la fonction normal

La fonction `normal` de MAPLE permet de mettre des expressions faisant intervenir des opérateurs arithmétiques sous la forme normale « numérateur-dénominateur ». Avec cette fonction il est possible de tester l'égalité à zéro d'expressions rationnelles. Appliquée sur des expressions plus générales, cette fonction parcourt récursivement la structure et normalise (dans le sens précédent) toutes les sous-expressions rationnelles.

```
> normal( (x^2-y^2)/(x-y)^3 );
                                 y + x
                              ----------
                                       2
                              (- x + y)

> normal( (f(x)^2-1)/(f(x)-1) );
                               f(x) + 1

> normal( sin(x*(x+1)-x) );
                                     2
                               sin(x )
```

Regrouper les termes d'un polynôme par rapport à une variable : la fonction collect

La fonction `collect` permet de regrouper des termes d'un polynôme à une ou plusieurs variables par rapport à une variable. On peut préciser une liste de variables et la fonction effectue alors le regroupement de manière récursive par rapport à chacune des variables, en commençant par la première de la liste.

```
>  p := (x+1)*y + x*(z-1);

                       p := (x + 1) y + x (z - 1)
> collect( p, x);

                           (y + z - 1) x + y
> collect( p, z);

                          x z + (x + 1) y - x
```

Développer une expression polynomiale : la fonction expand

Lorsqu'on entre un polynôme contenant des produits de polynômes, MAPLE ne développe pas les produits, il les garde sous une forme factorisée. Cela peut poser certains problèmes à des fonctions pour lesquelles la forme développée est nécessaire. C'est le cas de la fonction `coeff` qui retourne le coefficient d'un monôme passé en argument. MAPLE offre donc la fonction `expand` qui développe tous les produits présents dans une expression polynomiale.

```
> p := 1+x+x^2 ;
                                        2
                         p := 1 + x + x

> q := 1+y+y^2 ;
                                        2
                         q := 1 + y + y

> r := p * q ;
                                 2            2
                    r := (1 + x + x ) (1 + y + y )

> coeff( r, x, 2) ;   # pour avoir le terme en x^2 du polynome r
Error, unable to compute coeff
> expand( r );
                       2                2    2    2      2  2
              1 + y + y  + x + x y + x y  + x  + x  y + x  y

> coeff( ", x, 2);
                                      2
                             1 + y + y
```

1.4.4 Représentation des expressions : les fonctions op, nops et whattype

Les expressions MAPLE sont représentées par des arbres n-aires : chaque nœud de l'arbre a un nombre fini de fils, la nature du nœud étant soit un opérateur (`+,-,*,/,&*`, appel de fonction, indexation dans un tableau,...), soit une constante (numérique ou chaîne de caractères).

De plus chaque nœud possède un type (qui est le type de sa structure de données) : un nœud opérateur est de type ...l'opérateur ! Par exemple l'expression `a * b - c` est de type « `+` ». La structure de l'arbre dépend de la priorité des opérateurs, ici `*` est plus prioritaire que `-`.

On accède à ces informations par les fonctions suivantes :

`whattype( e ) ;` : retourne le type de l'expression `e`. Cette fonction est moins utilisée que la fonction `type` qui permet de tester le type d'une expression.

`nops ( e ) ;` : retourne le nombre de fils de l'expression (ou nombre d'opérandes). Par exemple « `nops( a * b - c )` » vaut 2, « `nops ( a*b*c )` » vaut 3 (les termes sont regroupés au sein d'un même nœud, `*` dans cet exemple).

`op( i, e ) ;` : retourne le `i`-ème opérande du nœud `e`. Par exemple, la valeur de « `op( 2, a * b - c )` » est « `b` », celle de « `op( 3, a*b*c )` » est « `c` ». Notons que `op (0, e)` est défini et dépend de la structure de données.

Par exemple considérons l'expression :

```
>  p := a+x*2 ;
                          p := a + 2 x

> whattype( p ) ;
                               +

> op(p);
                             a, 2 x

> op(0,p);
```

```
                                          +
> op(1,p);
                                          a
> op(2,p);
                                         2 x
> whattype ( x -> x^2 );
                                      procedure
> whattype( a,b,c ) ;
                                       exprseq
> whattype( [a,b,c,d] );
                                         list
```

Avec ces fonctions on peut parcourir toutes les structures de données de MAPLE. Voir à ce sujet le chapitre qui concerne les structures de données internes de MAPLE dans le manuel de référence du langage.

1.4.5 Modification des expressions : les fonctions subsop et subs

MAPLE offre une fonction spéciale qui permet de modifier le `i`-ème opérande d'une expression : `subsop ( eq1, eq2,...,eqn, expr )`. Chacune des expressions `eqI` est de la forme `indI = exprI`. La valeur de retour de cet appel est une nouvelle expression correspondant à `expr` dans laquelle chaque opérande d'indice `indI` est remplacé par `exprI`. Par exemple, pour remplacer le 3-ème opérande de `liste1` définie par :

```
> liste1 := [a,b,c,d] ;
                              liste1 := [a, b, c, d]

> subsop(3=`indice 3`, liste1);

                               [a, b, indice 3, d]
> liste1;
                                  [a, b, c, d]
```

Cette fonction permet de modifier des éléments particuliers pour les structures de données *séquence, ensemble, liste*. Pour une variable `a` d'une de ces structures, nous avons vu que l'opérateur d'indexation, « `a [i]` », retournait son `i`-ème élément. En fait, pour ces structures, « `a[i]` » est équivalent à « `op(i, a)` ». La fonction `subsop` permet alors de construire une copie de la structure après *substitution* de `op (i,a)` par l'expression précisée.

MAPLE offre une autre fonction, `subs`, qui permet des substitutions plus évoluées. Le principe est le même que pour la fonction `subsop`, excepté que ce n'est pas un opérande repéré par son indice qui est substitué, mais une sous-expression repérée par sa construction syntaxique (on appelle ceci du *pattern-matching*, ce qui pourrait se traduire littéralement par « mise en correspondance de motifs »).

La forme d'appel de cette fonction est : `subs( motif = remp, expr ) ;` . La valeur retournée est l'expression correspondant à `expr` dans laquelle la sous-expression `motif` est remplacée par l'expression `remp`. Si aucune sous-expression n'est trouvée alors la fonction retourne `expr`.

```
>  ens1 := { a,b,c,d,e,f };
                             ens1 := {a, b, d, c, e, f}

> subs( {e=q, b=q}, ens1 );
                                 {a, q, d, c, f}
```

L'exemple ci-dessous montre bien que la recherche de la sous-expression à modifier ne se fait que sur des critères syntaxiques :

```
>  expr := cos(x)^2 + y^2 - 1 ;

                                          2    2
                            expr := cos(x)  + y  - 1

> subs( cos(x)^2 = z^2, expr );

                                    2    2
                                   z  + y  - 1
> subs( cos(x)^2-1 = z, expr );

                                       2    2
                                 cos(x)  + y  - 1
```

1.4.6 Appliquer une fonction sur tous les éléments d'une structure de données : la fonction map

La fonction `map` permet d'appliquer une même fonction sur tous les éléments d'une structure de données. Par exemple, étant donnée une matrice `A`, l'instruction `map( x-> x^2, A) ;` retourne une nouvelle matrice dont les coefficients sont ceux de `A` au carré.

1.5 Les fonctions de la bibliothèque de Maple

La bibliothèque de Maple est immense : 95% des fonctions Maple sont écrites dans son langage. Le nombre de fonctions de la bibliothèque est de plusieurs centaines. Chaque fonction offrant en général plusieurs options, le nombre total de combinaisons est très important (voir la taille du manuel de référence de la bibliothèque Maple. . .).

Maple classe ses fonctions dans :

library : ensemble de fonctions qui ne sont pas automatiquement chargées lors du lancement de Maple. Ces fonctions n'ont pas forcément de liens les unes avec les autres. Pour avoir un aperçu de ces fonctions, entrer la commande `? library`. On charge une fonction de la *library* par la commande « `readlib ( le nom de la fonction);` ».

package : ou module. Maple offre plusieurs modules (algèbre linéaire, calcul de bases de Gröbner, affichage graphique) qui sont des ensembles de fonctions d'un même domaine d'application.

1.5.1 Aide en ligne

Pour rechercher efficacement une documentation sur une fonction, ou un groupe de fonctions (par exemple toutes les fonctions d'algèbre linéaire), Maple offre une aide en ligne, accessible depuis l'interpréteur. Pour recherche une aide sur une fonction de nom `fonction`, il suffit d'entrer l'instruction :

```
> ? fonction
```

Maple recherche dans sa base de données une description de cette fonction. Si jamais celle-ci n'existe pas, il retourne (parfois) une liste de fonctions lexicographiquement proches.

Cette commande `?` permet aussi d'avoir une description de l'index (`?index`), de l'index des fonctions de la bibliothèque (`?library`), ou bien la liste des modules de MAPLE (`?packages`).

```
> ? index
HELP FOR: Index of help descriptions

CALLING SEQUENCE:
   ?index[<category>]   or   help(index, <category>);

SYNOPSIS:
- The following categories of topics are available in the help subsystem.

     library       Index of descriptions for standard library functions
     packages      Index of descriptions for library packages of functions
     libmisc       Index of descriptions for miscellaneous lib functions

     statements    Index of descriptions for Maple statements
     expressions   Index of descriptions for Maple expressions
     datatypes     Index of descriptions for Maple datatypes
     tables        Index of descriptions for tables and arrays in Maple
     procedures    Index of descriptions for Maple procedures
     misc          Index of descriptions for miscellaneous facilities
```

1.5.2 Organisation de la bibliothèque MAPLE

Cette commande (`?`) est certainement la plus utilisée de MAPLE. Aussi pour rechercher une information est-il bon de connaître les différents modules de MAPLE.

Afin de ne pas trop utiliser de mémoire, les fonctions de ces modules ne sont pas accessibles dès que l'on lance MAPLE : il faut les charger en mémoire par la commande `with( module ):` où `module` est le nom du module en question (`linalg`, `plots`,...). Sur les machines UNIX, un fichier `.mapleinit` permet de charger automatiquement, lors du lancement de MAPLE, les packages dont on se sert le plus souvent. Il est conseillé de regarder comment faire ceci suivant les systèmes d'exploitation de la machine.

MAPLE possède aussi une *library*, pour charger une fonction de nom `fonction` de la *library* il faut faire `readlib( fonction )`. Voici la liste des packages de MAPLE V.3 :

```
> ?packages

HELP FOR: Index of descriptions for packages of library functions

SYNOPSIS:
- The following packages are available:

             numapprox:   numerical approximation
             combinat:    combinatorial functions
             DEtools:     differential equation tools
             difforms:    differential forms
             Gauss:       create domains of computation
             GaussInt:    Gaussian integers
             geom3d:      three-dimensional Euclidean geometry
             geometry:    two-dimensional Euclidean geometry
             grobner:     Grobner bases
             group:       permutation and finitely-presented groups
             liesymm:     Lie symmetries
             linalg:      linear algebra
             logic:       Boolean logic
```

```
            networks:    graph networks
            np:          Newman-Penrose formalism
            numtheory:   number theory
            orthopoly:   orthogonal polynomials
            padic:       p-adic numbers
            plots:       graphics package
            powseries:   formal power series
            projgeom:    projective geometry
            simplex:     linear optimization
            stats:       statistics
            student:     student calculus
            totorder:    total orders on names

- For information see ?<package> where <package> is from the above list.  This
  will give a list of the functions available in the package.   To cause all
  functions in a package to be defined in the session, do: with(<package>);
```

1.5.3 Algèbre linéaire

La bibliothèque des fonctions d'algèbre linéaire de MAPLE est bien fournie. Ces fonctions sont soit des fonctions de calcul (déterminant, rang, produit de matrices, calcul de formes normales...) soit des fonctions de déclaration et manipulation des matrices (matrice diagonale, matrice identité...).

```
> with(linalg):
Warning: new definition for   norm
Warning: new definition for   trace
> A := diag( 1, 2, 3, 4);

                                    [ 1  0  0  0 ]
                                    [            ]
                                    [ 0  2  0  0 ]
                               A := [            ]
                                    [ 0  0  3  0 ]
                                    [            ]
                                    [ 0  0  0  4 ]

> B := x - A ;

                                    B := x - A

> det(B);

                    (- 1 + x) (- 2 + x) (- 3 + x) (- 4 + x)

> charpoly(A,X);

                        (X - 1) (X - 2) (X - 3) (X - 4)
```

1.5.4 Sortie graphique

Le package `plots` permet d'afficher des graphiques (courbes, ensemble de courbes, courbes de niveau) en deux dimensions ou bien en trois dimensions. Des fonctions d'animation permettent d'enchaîner les graphiques. Nous verrons ces fonctions lors des TPs.

1.6 Comment faire pour...

1.6.1 ... mesurer les performances d'un algorithme

Un algorithme de calcul formel consomme deux ressources importantes : du temps et de la mémoire. MAPLE offre une fonction `time()` (sans arguments) qui retourne le temps passé dans les calculs depuis le lancement de MAPLE. Mesurer le temps d'exécution de l'appel à une fonction `f` avec des arguments `arg` peut se faire comme suit :

```
> # t est une variable temporaire qui contient la date de debut d'execution
> t := time() :  R := f ( args ): t := time() - t ;
```

La variable globale `status` (une séquence d'expression) permet d'avoir des informations sur l'utilisation des ressources utilisées par MAPLE. Les principales ressources sont :

`status[1]` : total du nombre de demandes d'allocation mémoire en « mots ». La taille d'un mot est de 4 octets (1 octet = 8 bits). Cette information compte le nombre de requêtes à l'allocateur mémoire de MAPLE, c'est une information sur la taille mémoire physique qui serait nécessaire s'il n'y avait aucune réutilisation de celle-ci.

`status[2]` : total du nombre de « mots » physiquement alloués par MAPLE. Cette information est la « consommation » réelle en mémoire par MAPLE.

`status[3]` : temps passé dans les calculs.

Les autres éléments (de `status[4]` à `status[8]`) sont des informations qui concernent l'utilisation du gestionnaire de mémoire de MAPLE.

Le lecteur intéressé pourra consulter l'aide en ligne à propos de la fonction `showtime`.

1.6.2 ... écrire une page d'aide sous MAPLE

Un code informatique quel qu'il soit est toujours d'autant plus clair qu'il est bien documenté. On a déjà pu voir comment utiliser la commande `help` (aussi notée `?`) pour accéder à la documentation disponible par défaut. Il est aussi possible à l'utilisateur d'écrire ses propres fichiers de documentation et d'y accéder par la même commande `help`. C'est ce que nous décrivons maintenant.

Soit à documenter la fonction `manuel` qui à partir d'un fichier de texte retourne un fichier au format de la documentation MAPLE. On écrit un texte descriptif dans un fichier « fichdoc » :

```
  FUNCTION: manuel - transforme un fichier texte au format maple

  CALLING SEQUENCES:
manuel(nom,fichier);

  PARAMETERS:
        - "nom" : le nom de la fonction a documenter
        - "fichier" : le fichier texte descriptif

  SYNOPSIS:
        manuel lit un fichier texte documentaire d'une fonction
donnee et retourne un fichier de documentation au format maple.


  EXAMPLES:
```

```
> makehelp(makehelp,fichdoc);
```

On transforme maintenant le fichier « fichdoc » en un fichier au format MAPLE :

```
> readlib(makehelp);
> makehelp(manuel,fichdoc);
```

et on sauvegarde ce fichier dans un nouveau fichier « fichdocmaple » :

```
> save `help/text/manuel`, fichdocmaple;
```

On peut alors accéder à la documentation de la fonction `manuel` en tapant :

```
> ? manuel
```

Bien entendu, cette documentation n'est pas rajoutée aux fichiers du système. Elle reste chez l'utilisateur et doit être rechargée à chaque nouvelle session. Le mieux est donc d'inclure le fichier « fichdocmaple » dans le fichier contenant le code de la fonction elle-même : à chacune de ses utilisations la fonction est alors documentée.

1.6.3 ... étudier un algorithme d'une fonction de la bibliothèque

Il est toujours intéressant de savoir quel est l'algorithme utilisé par Maple pour calculer une fonction. Étant donné que très peu d'informations existent à propos des algorithmes utilisés, il est parfois nécessaire de se plonger dans le « listing » des fonctions. Heureusement, Maple offre des fonctions permettant d'afficher ce listing. La commande `interface` permet de positionner la variable `verboseproc` afin de pouvoir lire une fonction `f` par l'instruction `print ( eval(f) ) ;` (l'instruction `print( f )`, pour une fonction ne suffit pas, puisque l'évaluation simple de `f` est `f` !).

Il est instructif de faire l'essai sur une fonction quelconque : le moindre listing tient sur plusieurs pages ! Voici l'exemple d'une petite fonction de la bibliothèque `linalg` :

```
> interface( verboseproc=2 );
> print(eval(JordanBlock));

proc(t,size)
local i,j,Jb;
options `Copyright 1993 by Waterloo Maple Software`;
    if not type(size,'integer') then ERROR(`matrix size must be integer`) fi;
    if nargs <> 2 then ERROR(`must have 2 arguments`) fi;
    Jb := array(1 .. size,1 .. size);
    for i to size-1 do
        for j to i do  Jb[i,j] := 0 od;
        Jb[i,i] := t;
        Jb[i,i+1] := 1;
        for j from i+2 to size do  Jb[i,j] := 0 od
    od;
    for j to size-1 do  Jb[size,j] := 0 od;
    Jb[size,size] := t;
    RETURN(op(Jb))
end
```

Exercice. Trouvez ce que fait la fonction `JordanBlock` ci-dessus.

Exercice. Étudiez le listing de la fonction `multiply` de la bibliothèque `linalg` (produit de matrices). Comparez l'algorithme utilisé par rapport à celui de l'exercice sur la méthode de Strassen-Winograd au paragraphe 1.3.

Chapitre 2

Algèbre linéaire

Gilles VILLARD

MAPLE est l'un des systèmes de calcul formel les plus performants en algèbre linéaire. De la résolution de systèmes linéaires au calcul de valeurs propres, jusqu'au calcul de fonctions de matrices ou la mise sous forme canonique, large est la gamme des fonctions disponibles pour la manipulation et le calcul matriciels. Vu les différents types de données qui peuvent être utilisés, on saura mettre à profit ces possibilités pour des matrices dont les coefficients appartiennent à des domaines variés. On peut pratiquer le calcul numérique sur des matrices à coefficients flottants. On peut aussi surtout utiliser des algorithmes exacts par exemple pour des matrices booléennes, sur des corps finis, sur les rationnels, sur des extensions algébriques ou encore sur des polynômes.

Dans ce chapitre, nous allons faire un tour d'horizon rapide des notions de base importantes pour utiliser ces possibilités en algèbre linéaire. Loin de nous de vouloir être exhaustifs, nous voulons surtout faciliter la tâche du débutant. Les fonctions que nous allons utiliser sont accessibles d'une part par l'intermédiaire du `package linalg`. Donc en particulier il est toujours plus que recommandé d'utiliser le manuel en ligne : `?linalg` pour un survol du `package` et la liste de toutes les fonctions disponibles, ou par exemple `?linalg[determinant]` pour des renseignements sur une fonction particulière, ici le calcul du déterminant. D'autre part, nous ferons appel à des fonctions trouvées dans la bibliothèque utilisateur, `share library`. Rappelons que `with(share)` permet d'accéder à cette dernière, et que par exemple on peut charger un `package` par `readshare (‘linalg/normform‘)` pour le calcul de formes canoniques de matrices.

Voilà en guise de brève introduction. Outre certaines fonctions de MAPLE qui seront présentées, nous étudierons quelques algorithmes anciens ou très récents propres au domaine du calcul formel. Ces exemples ont pour but d'introduire certaines méthodes utilisées fondamentalement pour limiter les temps pratiques de résolution des problèmes. Plus précisément, nous commencerons au paragraphe 2.1 par apprendre à créer des matrices et à les manipuler. Au paragraphe 2.2 nous aborderons la résolution de systèmes linéaires. Nous traiterons ensuite au paragraphe 2.3 du calcul de valeurs propres. Les paragraphes 2.4 et 2.5 aborderont alors un thème cher au calcul algébrique qui est celui du calcul de formes canoniques de matrices. Nous le verrons dans deux cas : quand la relation d'équivalence utilisée est la similitude puis avec l'équivalence sur un anneau principal. Au paragraphe 2.6 nous montrerons comment « reprogrammer » certaines fonctions pour qu'elles puissent s'exécuter quand les matrices ont des coefficients dans des domaines tels que les extensions algébriques ; ceci pour outre-passer les limitations actuelles du système. Pour conclure, et à partir de ces différents exemples, nous proposerons un TP.

Avertissement. *Nous ne donnerons que de brefs aperçus des preuves des résultats que nous utiliserons. Le lecteur pourra se reporter aux références que nous proposons tout au long du texte pour* *approfondir telle ou telle démonstration ou notion.*

2.1 Fonctionnalités de base

Pour une session utilisant le `package` d'algèbre linéaire, le plus facile pour disposer de toutes les fonctions est de taper

```
with(linalg);
```

2.1.1 Initialisation et évaluation de matrices

Une fois que la bibliothèque est chargée, des matrices peuvent alors être initialisées par :

```
a:=matrix(3,3,[1,2,3,4,5,6,7,8,9]);
b:=matrix([[1,0,0],[0,1,1],[1,1,1]]);

       [ 1  2  3 ]
       [         ]
  a := [ 4  5  6 ]
       [         ]
       [ 7  8  9 ]

       [ 1  0  0 ]
       [         ]
  b := [ 0  1  1 ]
       [         ]
       [ 1  1  1 ]
```

Attention, une des particularités des matrices est que les symboles les représentant ne sont *a priori* pas évalués par l'interpréteur. Leur évaluation doit donc être appelée explicitement par l'utilisateur. Taper le seul symbole de la matrice ne produit « rien » :

```
a;
                a
```

Il faut utiliser la fonction `eval` pour « voir » la matrice :

```
eval(a)
       [ 1  2  3 ]
       [         ]
       [ 4  5  6 ]
       [         ]
       [ 7  8  9 ]
```

De même, dans une fonction utilisateur, un `RETURN(a)` ne produirait « rien » à part le symbole `a`. En revanche, un `RETURN(eval(a))` renverrait effectivement la valeur de la matrice `a`.

Le mot `matrix` est un type connu du système :

```
type(a,matrix);
           true
type(b,'matrix'('integer'));
           true
```

et peut donc être utilisé dans le corps d'une fonction pour prévenir des erreurs.

Ce que nous avons dit pour l'évaluation des matrices prévaut aussi au moment de leur recopie. L'instruction `c:=b` (puisqu'il n'y a pas évaluation de `b`) ne recopie pas physiquement la matrice `b` mais définit simplement une identité de symboles. Donc attention : une modification ultérieure de `b` modifierait `c` par la même occasion. Pour une recopie physique :

```
c:=copy(b);
```

Nous pouvons maintenant produire des expressions arithmétiques de matrices. Les opérateurs usuels sont disponibles. On peut utiliser les opérations matricielles et scalaires, ainsi élever au carré puis ajouter 2 fois l'identité s'exprime par $a^2 + 2$. Le mécanisme d'évaluation doit toujours être appelé à la main, `evalm` est la fonction correspondante :

```
evalm(a^2+2);
  [  40   40   48 ]
  [               ]
  [  74   83   96 ]
  [               ]
  [ 116  126  152 ]
```

Le symbole particulier `&*` doit être utilisé pour la multiplication non commutative :

```
evalm(3*a&*b);
```

calcule l'expression où la première multiplication est scalaire (commutative) et la deuxième est matricielle. En d'autres termes une constante est entendue comme un multiple constant de l'identité, à la dimension déduite du contexte.

2.1.2 Construction de matrices, sous-matrices

Bien souvent une matrice est obtenue comme construite à partir d'autres matrices ou comme sous-matrice. Plusieurs fonctions fournissent donc des raccourcis bien agréables quand il s'agit de construire une matrice donnée. Cela peut être `augment` qui « colle » plusieurs matrices entre elles ou `stack` qui les « empile » :

```
augment(stack(a,diag(1,1,1)),matrix(6,3,-4));
     [ 1  2  3  -4  -4  -4 ]
     [ 4  5  6  -4  -4  -4 ]
     [ 7  8  9  -4  -4  -4 ]
     [ 1  0  0  -4  -4  -4 ]
     [ 0  1  0  -4  -4  -4 ]
     [ 0  0  1  -4  -4  -4 ]
```

ou encore `extend(a,3,4,-1)` qui rajouterait 3 lignes et 4 colonnes à a pour les remplir de -1. Plusieurs structures de matrices sont connues du système. Une matrice diagonale est créée par la fonction `diag`. On peut créer un bloc compagnon à partir d'un polynôme (coefficient de tête égal à 1 et sous forme développée) ou créer un bloc de Jordan :

```
diag(companion(x^3-2*x+1/4,x),JordanBlock(3**{1/2},2));
     [ 0  0  -1/4     0        0   ]
     [ 1  0    2      0        0   ]
     [ 0  1    0      0        0   ]
     [               {1/2}         ]
     [ 0  0    0    3          1   ]
     [                       {1/2} ]
     [ 0  0    0      0     3      ]
```

Nous laissons au lecteur la découverte des matrices de Toeplitz, Sylvester, Vandermonde ou autres. Enfin, on dispose de la fonction `randmatrix` pour générer des matrices aléatoirement. Pour une matrice polynomiale, on se donne comme outil une fonction qui génère des polynômes de degré 4 aux coefficients compris entre 0 et 10 :

```
mes_poly := proc() Randpoly(4, x) mod 10 end;
A:=randmatrix(3,3,entries=mes_poly,sparse);
      [    4      2              4    3      2              4    3         ]
      [ 6 x  + 9 x  + x + 9   5 x  + x  + 9 x  + 3 x + 8   6 x  + x  + 6 x + 1 ]
      [                                                                    ]
 A := [         0                       0                       0          ]
      [                                                                    ]
      [                        4     3     2                               ]
      [         0           9 x  + 5 x  + 5 x  + 7 x + 5            0          ]
```

2.1.3 Opérations élémentaires

Voilà, forts de toutes les matrices que nous pouvons construire, nous pouvons effectuer des calculs. Les paragraphes qui suivent traiteront d'algorithmes « élaborés ». Mais toute opération complexe est plus facilement mise au point si elle est spécifiée en terme d'opérations plus simples ou élémentaires. Voyons donc ces dernières.

On peut commencer par accéder aux éléments de la matrice. Ceci se fait naturellement en spécifiant l'indice de ligne et de colonne du coefficient visé, par exemple `A[1,2]:=x^3-1/2` affecte le deuxième coefficient de la première ligne. Les opérations élémentaires sont ensuite écrites à l'aide des fonctions `rowdim` et `coldim` qui retournent les nombres de lignes et de colonnes :

```
rowdim(A);
          3
```

et à l'aide de `row` et `col` qui permettent d'accéder aux lignes et aux colonnes séparément :

```
col(A,2);
            4    3      2                    4      3      2
     [ 5 x  + x  + 9 x  + 3 x + 8, 0, 9 x  + 5 x  + 5 x  + 7 x + 5 ]
```

On peut aussi multiplier une ligne ou une colonne d'une matrice par un scalaire avec `mulrow` et `mulcol`. On peut multiplier une ligne (ou une colonne) par un scalaire et l'ajouter à une autre. Initialisons une matrice et ajoutons k fois la troisième ligne à la deuxième :

```
B:=matrix([[u,v,w],[0,k*x,k*y],[0,-x,-y]]); B:=addrow(B,3,2,k);
                [ u   v    w ]      [ u   v    w ]
                [            ]      [            ]
           B := [ 0  k x  k y ]      [ 0   0    0 ]
                [            ]      [            ]
                [ 0  - x  - y ]      [ 0  - x  - y ]
```

On peut aussi transposer une matrice par `transpose` et appliquer une même fonction à tous ses coefficients, c'est ce qu'il faut utiliser pour simplifier une matrice : appliquer la fonction `simplify` à tous ses coefficients. Pour élever au carré les coefficients de la précédente matrice, on peut faire par exemple :

```
map(z-> z^2,B);
        [  2    2    2 ]
        [ u    v    w  ]
        [              ]
        [  0    0    0 ]
        [              ]
        [       2    2 ]
        [  0   x    y  ]
```

2.1.4 Limitations

Pour terminer ce paragraphe « débuts avec l'algèbre linéaire », on peut mettre en évidence certaines limitations actuelles de ce qui est fourni par le système (et non les limitations du système en lui-même). Nous voulons noter ici que les fonctions fournies par MAPLE ne s'exécutent pas sur n'importe quel domaine de calcul. Si l'on se donne par exemple le nombre α comme étant une racine du polynôme (irréductible) $x^2 + x + 1$ et que l'on construit une matrice à l'aide de α :

```
alpha:=RootOf(x^2+x+1);
U:=matrix([[alpha,alpha+1],[alpha+2,alpha]]);
            [             2                        2              ]
            [   RootOf(_Z   + _Z + 1)    RootOf(_Z   + _Z + 1) + 1 ]
      U := [                                                      ]
            [           2                            2            ]
            [ RootOf(_Z   + _Z + 1) + 2    RootOf(_Z   + _Z + 1)   ]
```

l'appel à la fonction de calcul du rank de MAPLE (`rank`) produit une erreur :

```
rank(U);
Error, (in linalg[gausselim]) matrix entries must be rational polynomials
```

Heureusement ce problème sera aisément contourné au paragraphe 2.6.

2.2 Systèmes linéaires et inversion de matrices

Dans le cadre de ce paragraphe rentreraient toutes les considérations concernant la résolution d'un système linéaire, le calcul du rang, l'inversion d'une matrice ou le calcul du polynôme caractéristique. Toutes ces fonctions existent en MAPLE: `linsolve`, `rank`, `inverse` et `charpoly`. Le lecteur intéressé pourra se reporter au livre remarquable de Bini et Pan [15] qui dresse un survol (presque exhaustif) des différentes méthodes employées en calcul formel pour résoudre ces problèmes.

Comme exemples représentatifs des techniques employées, nous allons voir ici comment trianguler une matrice exactement tout en limitant le grossissement des coefficients qui apparaissent au cours des calculs. Nous montrerons ensuite comment résoudre un système linéaire à un coût asymptotiquement à peine plus élevé qu'en calcul numérique.

2.2.1 Méthode de Bareiss

On considère une matrice A à coefficients dans un anneau (l'anneau des entiers par la suite). Pour simplifier, la matrice est supposée carrée de dimension n et inversible (sans perte de généralité). Le problème consiste à calculer une matrice équivalente à A qui soit triangulaire supérieure et obtenue par des opérations sur les lignes. Une contrainte supplémentaire est que tous les calculs doivent rester dans l'anneau des coefficients de A. Une méthode naïve consiste, en guise de pivotage, à multiplier allègrement les coefficients de certaines lignes par des entiers bien choisis, pour en annuler d'autres. Sur un exemple :

```
A:=matrix([[3,4],[5,2]]); A:=mulrow(A,2,3);A:=addrow(A,1,2,-5);
        [ 3  4 ]       [ 3   4 ]
   A := [      ]       [       ]
        [ 15  6 ]      [ 0  -14 ]
```

Malheureusement, on peut facilement montrer qu'un tel algorithme est exponentiel, non du point de vue du nombre d'opérations effectuées sur les entiers (c'est une élimination, cela reste polynomial), mais du point de vue binaire. En effet la longueur du codage des entiers (logarithme de leurs valeurs) qui interviennent est *grosso modo* doublée à chaque étape (zéros dans la colonne k) et augmente donc exponentiellement. Si les entiers sont initialement bornés en valeur absolue par β et que l'on note $B^{(k)}$ une borne sur les entiers à l'étape k, $1 \leq k \leq n-1$, on a une relation du type

$$B^{(k)} = \mathcal{O}\left([B^{(k-1)}]^2\right) = \mathcal{O}(\beta^{2^k}).$$

Un moyen très utilisé pour remédier à ce phénomène est d'utiliser une triangularisation « à la Bareiss » [16]. Cela consiste à introduire des simplifications (divisions exactes : n'oublions pas que nous voulons garder des entiers) pour limiter la taille des nombres. L'algorithme se présente alors (quitte à permuter des lignes, on suppose que les coefficients diagonaux rencontrés sont non nuls) :

```
simp:=1;
for k from 1 to n-1 do
       for i from k+1 to n do
              for j from k to n do
                  A[i,j]:=(A[k,k]*A[i,j]-A[i,k]*A[k,j])/simp;
              od;
       od;
       simp:=A[k,k];
od;
```

Si l'on note $A_{i,j}^{(k)}$ le coefficient (i,j) de la matrice après l'annulation de tous les coefficients en colonne k sous la diagonale, on a :

$$A_{i,j}^{(k)} = \left(1/A_{k-1,k-1}^{(k-2)}\right) \begin{bmatrix} A_{k,k}^{(k-1)} & A_{k,j}^{(k-1)} \\ A_{i,k}^{(k-1)} & A_{i,j}^{(k-1)} \end{bmatrix}.$$

Qu'advient-il maintenant des entiers au cours de l'élimination ? Regardons sur un exemple 3×3. La matrice après la première étape est :

$$A^{(2)} = \begin{bmatrix} A_{1,1} & A_{1,2} & A_{1,3} \\ 0 & A_{1,1}A_{2,2} - A_{1,2}A_{2,1} & A_{1,1}A_{2,3} - A_{1,3}A_{2,1} \\ 0 & A_{1,1}A_{3,2} - A_{1,2}A_{3,1} & A_{1,1}A_{3,3} - A_{1,3}A_{3,1} \end{bmatrix}.$$

Après la deuxième étape, le coefficient $(3,3)$ est seul modifié une deuxième fois :

$$(A_{1,1}A_{2,2} - A_{1,2}A_{2,1}) * (A_{1,1}A_{3,3} - A_{1,3}A_{3,1}) - (A_{1,1}A_{2,3} - A_{1,3}A_{2,1}) * (A_{1,1}A_{3,2} - A_{1,2}A_{3,1})$$
$$= A_{1,1}\Delta_{3,3}$$

où $\Delta_{3,3}$ est le déterminant de A. La division par $A_{1,1}$ demandée par l'algorithme est donc exacte et l'on obtient :

$$A_{3,3}^{(2)} = \Delta_{3,3}.$$

D'après Bareiss [16] on sait que toutes les divisions effectuées au cours de l'algorithme sont exactes (les coefficients restent entiers). De plus, notons $\Delta_{i,j}$ le mineur d'ordre i de la matrice A construit sur les i premières lignes et avec les $i-1$ premières colonnes complétées de la j-ième. On sait qu'à la fin de l'algorithme, tous les coefficients de la matrice triangulaire sont des mineurs de la matrice A, plus précisément :

$$A_{i,j}^{(n-1)} = \Delta_{i,j}.$$

Nous avons supposé qu'il n'y avait pas de permutation à effectuer, tout se généralise sans difficulté au cas d'une matrice quelconque.

Si les coefficients de la matrice A en entrée sont bornés en valeur absolue par un entier β, on peut utiliser la borne d'Hadamard pour borner tous les mineurs qui interviennent. En effet un déterminant construit sur k vecteurs donnés est borné par le volume qu'indiqueraient ces vecteurs s'ils étaient orthogonaux, donc tous les mineurs sont bornés par $\mathcal{O}(\beta^n n^{n/2})$. Autrement dit, la longueur du codage des entiers qui interviennent (logarithme des valeurs) est bornée par $\mathcal{O}(n \log(n\beta))$ et reste polynomiale. On montre que le coût total de l'élimination est en $\mathcal{O}(n^5 \log^2(n\beta))$ (arithmétique standard) [17].

2.2.2 Méthode p-adique

Le détail de ce paragraphe peut être trouvé dans l'article de Dixon [20] ou le livre de Gregory et Krishnamurthy [21]. Pour résoudre un système linéaire, on se placera maintenant sur un corps, celui des rationnels. On pourrait utiliser la méthode précédente, en effet, une fois la matrice triangularisée, il est facile de résoudre un système triangulaire. Cependant on peut donner d'autres méthodes de coûts moindres. Une première alternative serait d'utiliser une arithmétique multi-modulaire [17]. Mais on peut encore faire mieux.

• **Préliminaire.** Soit un nombre rationnel inconnu x. Pour un nombre premier p et un entier positif m donnés, on sait seulement que $x = ab^{-1} \equiv \chi \bmod p^m$. On se propose d'appliquer l'algorithme d'Euclide généralisé pour calculer la fraction x à partir de son résidu χ modulo p^m. Soient deux couples d'entiers (a_{-1}, b_{-1}) et (a_0, b_0), on considère la suite (a_i, b_i), pour $i \geq 1$, donnée par :

$$\begin{cases} a_i &= a_{i-2} - q_i a_{i-1} \\ b_i &= b_{i-2} - q_i b_{i-1}, \end{cases}$$

où q_i est le quotient de la division entière de a_{i-2} par a_{i-1}. On sait que pour tout $i \geq 1$, $(a_0 b_i - b_0 a_i) \equiv (a_{-1} b_0 - b_{-1} a_0) \bmod p^m$.

On construit la suite précédente avec $(a_{-1}, b_{-1}) = (p^m, 0)$ et $(a_0, b_0) = (\chi, 1)$. On sait donc que pour tout $i \geq 1$ on a

$$a_i b_i^{-1} \equiv \chi \bmod p^m.$$

On admettra que si les deux entiers recherchés a et b vérifient

$$|a|, |b| \leq \lambda p^{m/2} \tag{2.1}$$

où λ est une constante majorée par 0.619 ($\lambda^2 + \lambda - 1 = 0$), alors $(a, b) = (a_I, b_I)$ pour I le premier indice tel que $|a_I| < p^{m/2}$.

Appliquons le raisonnement pour calculer a et b tels que $x = a/b$ sachant que $x \equiv 49 \bmod 13^2$ et que $|a|, |b| \leq 13\lambda$. On trouve $a = 5$ et $b = 7$.

• **Application à la résolution de systèmes linéaires.** Soit A une matrice $n \times n$ et b un vecteur de dimension n tous deux à coefficients entiers majorés en valeur absolue par un entier β. On cherche à résoudre le système $Ax = b$. Soit p un nombre premier tel que l'inverse de A modulo p existe, on la note C : $AC = CA = Id \bmod p$.

On construit pour cette résolution deux suites de vecteurs $\{\chi_k\}$ et $\{b_k\}$ pour $k \geq 0$ par :

$$\begin{cases} b_0 & = & b \\ \chi_k & = & Cb_k \bmod p \\ b_{k+1} & = & (b_k - A\chi_k)/p. \end{cases}$$

Remarquons que les b_k, $k \geq 0$, ont des composantes entières :

$$b_k - A\chi_k = A(Cb_k - \chi_k) \equiv 0 \bmod p.$$

Si pour un entier m on pose

$$\chi = \sum_{k=0}^{m-1} \chi_k \; p^k,$$

on voit que $A\chi \equiv b \bmod p^m$. En effet,

$$\begin{aligned} A\chi & = \textstyle\sum_{k=0}^{m-1} p^k A\chi_k \\ & = \textstyle\sum_{k=0}^{m-1} p^k (b_k - pb_{k+1}) = b_0 - p^m b_m = b - p^m b_m. \end{aligned}$$

Cette relation permet donc de calculer le résidu χ du vecteur inconnu x modulo p^m. Suivons maintenant le préliminaire pour trouver les valeurs rationnelles des composantes de x. Il faut que p^m, donc le nombre d'itérations, soit assez grand en fonction d'une borne connue sur les composantes. En utilisant la borne d'Hadamard $\mathcal{O}(\beta^n n^{n/2})$ pour borner les numérateurs et les dénominateurs, l'inégalité 2.1 du préliminaire dit qu'il faut choisir le nombre d'itérations m tel que :

$$\beta^n n^{n/2} \leq \lambda p^{m/2}$$

soit m de l'ordre de $\mathcal{O}(n \log(n\beta))$. Évaluons pour terminer le coût de cette résolution. Par récurrence, on obtient facilement que que les composantes des b_k restent bornées en valeur absolue par $2n\beta$. En utilisant le fait que le coût du produit modulo p de deux entiers bornés en valeur absolue par M est $\log(M)$, le coût d'une itération (un produit matrice-vecteur) est en $\mathcal{O}(n^2 \log(n\beta))$, donc le coût de toutes les itérations est en :

$$\mathcal{O}(n^3 \log^2(n\beta)).$$

C'est en fait le coût total de l'algorithme (on montre que les remontées des composantes de x en appliquant le préliminaire mènent à la même complexité).

Il faut bien voir que ce coût est assez remarquable : la résolution d'un système linéaire par une méthode numérique quelconque en arithmétique standard est en $\mathcal{O}(n^3)$. C'est dire que l'on vient de résoudre le problème de manière exacte dans un facteur asymptotiquement seulement logarithmique comparé aux méthodes approchées !

2.3 Valeurs propres

On sait combien le problème du calcul des valeurs propres d'une matrice est fondamental en analyse numérique. Nous allons voir comment MAPLE se comporte dans ce domaine, et quels sont les apports du calcul formel à la compréhension des phénomènes.

Soit la matrice proposée en exemple par C. Moler [22] :

```
        [ -149  -50  -154 ]
   A := [  537  180   546 ]
        [  -27   -9   -25 ]
```

En utilisant la fonction `eigenvals`, on voit que A possède trois valeurs propres distinctes réelles :

```
eigenvals(A);
        1,2,3
```

Observons ce qui se passe sous l'action d'une petite perturbation de A, ceci nous conduit à :

```
 P:=epsilon -> matrix([[130*epsilon-149,-50-390*epsilon,-154],
    [537+43*epsilon,180-129*epsilon,546],[133*epsilon-27,-9-399*epsilon,-25]]);
```

Ici nous avons défini une nouvelle fonction sous MAPLE qui nous retourne pour tout ϵ donné, une matrice $P(\epsilon)$ telle que $P(0) = A$. Par exemple :

```
print(P(-0.0005),P(0.0001));
     [ -149.0650  -49.8050  -154 ]  [ -148.9870  -50.0390  -154 ]
     [  536.9785  180.0645   546 ], [  537.0043  179.9871   546 ]
     [  -27.0665   -8.8005   -25 ]  [  -26.9867   -9.0399   -25 ]
```

La figure 2.1 montre alors comment se comportent la partie réelle et la partie imaginaire de la première valeur propre de $P(\epsilon)$ en fonction de ϵ (la valeur propre est 1 en zéro). En « zoomant », on se rendrait compte qu'à partir d'une valeur positive de ϵ proche de zéro, appelons-la ϵ_0, la partie imaginaire devient strictement positive. Que s'est-il passé ?

Pour comprendre, on peut calculer le polynôme caractéristique $\chi_\epsilon(x)$ de $P(\epsilon)$ à l'aide de la fonction `charpoly` :

```
chi:=charpoly(P(epsilon),x);
          3      2                    2
  chi := x  - 6 x  + 11 x - epsilon x  + 492512 epsilon x - 1221271 epsilon - 6
```

son discriminant est lui-même calculé à l'aide de la fonction `discrim` :

```
delta:=discrim(chi,x);
                                                       2
   delta := 4 - 5910096 epsilon + 1403772863224 epsilon
                                             3                      4
        - 477857003880091920 epsilon  + 242563185060 epsilon
```

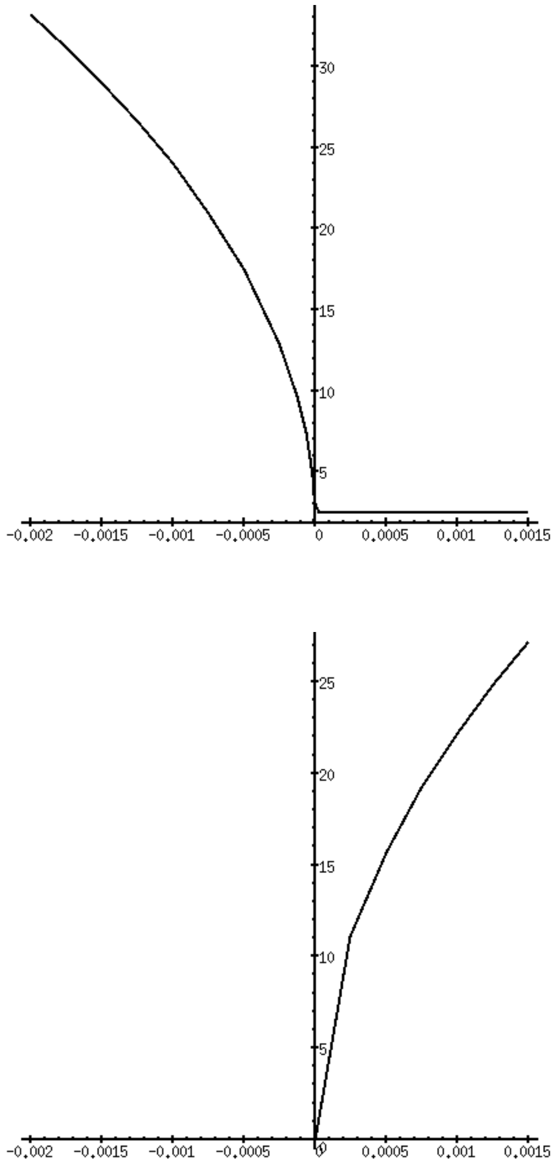


FIG. 2.1 – Partie réelle et partie imaginaire de la première valeur propre de $P(\epsilon)$.

Et les racines (calculées par `fsolve`) de ce discriminant sont :

```
fsolve(delta);
                                 -6                  7
               .7837924906*10   , .1970031041*10
```

Le premier de ces nombres correspond au ϵ critique de tout à l'heure, $\epsilon_0 \approx 0.0000007837924$. En ϵ_0 le discriminant s'annule : deux valeurs propres réelles se joignent en ϵ_0 pour n'en former qu'une seule, soit λ_0, double et complexe. Si l'on veut étudier maintenant la matrice $P(\epsilon_0)$, on peut calculer ses valeurs propres :

```
eigenvals(P(0.0000007837924906));
      1.546279509, 1.548938630, 2.904782575
```

On a beau savoir (grâce au calcul exact) qu'il y a une valeur propre double, quelle que soit la précision de ce dernier calcul, tant qu'il est approché (et de plus effectué en arithmétique flottante) la valeur propre double est toujours contaminée par les erreurs. Elle apparaît seulement par les valeurs très proches de deux des valeurs propres, soient λ_0' et λ_0'', qui ont été calculées : il faut faire très attention pour donner un sens à l'égalité. Dans la situation présente on peut s'en accommoder sans conséquence, un calcul de rang par décomposition en valeurs singulières fournirait un très bon test d'égalité. Cependant, si maintenant la question est de savoir si l'on trouve un ou deux vecteurs propres associés à la valeur propre double λ_0 ainsi caractérisée, le calcul numérique devient plus qu'ambigü. Essayons de faire les calculs exactement. Le premier besoin est de savoir manipuler ϵ_0 exactement. Qu'à cela ne tienne, on definit ϵ_0 (`e0` ci-dessous en MAPLE) comme étant racine du discriminant (il est irréductible), et on construit la matrice $P(\epsilon_0)$ correspondante :

```
e0:=RootOf(delta):
Pe0:=P(e0):
print(Pe0);
             [ 130 %1 - 149   - 50 - 390 %1   -154 ]
             [  537 + 43 %1    180 - 129 %1    546 ]
             [  133 %1 - 27    - 9 - 399 %1    -25 ]

%1 := RootOf(
                                            2                      3              4
    1 - 1477524 _Z + 350943215806 _Z  - 119464250970022980 _Z  + 60640796265 _Z)
```

Un moyen radical de connaître la structure de $P(\epsilon_0)$ est de calculer sa forme de Jordan. La fonction correspondante de `linalg` ne permet pas de le faire. On peut en revanche faire appel au `package normform` de la bibliothèque utilisateur :

```
with(share); readshare(`linalg/normform`);
J:=jordan(Pe0,{RootOf(delta)});

    110319292597811224534 95   3   283005565738990253   179242416783149808 9735
 [- ----------------------- %1  + ------------------ - ---------------------- %1
       96703623305979008           96703623305979008       8791238482361728
        217332430796812777761111 27375   2
      + ---------------------------- %1  , 0, 0]
             96703623305979008
          297216174096883795   217332430796812777761111 27375   2
       [0, ------------------ - ----------------------------- %1
          193407246611958016         193407246611958016
             110319292597811224534 95   3   179243295906998045 1463
```

```
       + ---------------------- %1  + --------------------- %1, 1]
             193407246611958016            17582476964723456
            297216174096883795   217332430796812777611127375   2
    [0, 0, ------------------ - ----------------------------- %1
            193407246611958016         193407246611958016
            1103192925978112245349 5   3   179243295906998045146 3
         + ---------------------- %1  + --------------------- %1]
              193407246611958016            17582476964723456
%1 := RootOf(
                                    2                        3               4
    1 - 1477524 _Z + 350943215806 _Z  - 119464250970022980 _Z  + 60640796265 _Z)
```

L'expression ci-dessus est bien entendu très compliquée. On peut l'interpréter facilement en l'évaluant (maintenant *a posteriori*) en une valeur approchée de ϵ_0 :

```
JJ:=map(x->subs(%1=0.000000784,x),J);
              [ 2.904815336       0             0      ]
        JJ := [      0       1.547592724        1      ]
              [      0            0        1.547592724 ]
```

pour se rendre compte qu'elle possède un bloc de dimension deux associé à la valeur propre double. La réponse à la question que l'on s'était posée est que la valeur propre λ_0 est associée à un unique vecteur propre.

On aurait pu, pour fournir un peu moins d'efforts, se contenter de la forme de Frobenius :

```
F:=frobenius(Pe0,{RootOf(delta)});
                   [ 0  0   1221271 %1 + 6  ]
              F := [ 1  0  - 11 - 492512 %1 ]
                   [ 0  1       6 + %1      ]
%1 := RootOf(
                                    2                        3               4
    1 - 1477524 _Z + 350943215806 _Z  - 119464250970022980 _Z  + 60640796265 _Z)
```

cette dernière possède un seul bloc compagnon, la valeur propre double ne peut donc pas être associée à deux vecteurs propres distincts.

Nous venons de traiter par le détail un exemple qui montre la portée du calcul formel en algèbre linéaire. Quand les limites du calcul approché sont atteintes, l'outil apparaît irremplaçable. Mais ne soyons pas irréalistes, ne nous trompons surtout pas de question : on ne résoudra pas exactement un problème de valeurs propres en vraie grandeur (plusieurs milliers d'équations). Ce qu'il faut entendre (dans ce cas précis), c'est que le calcul formel peut et doit aider à la compréhension des problèmes auxquels se trouve confronté l'ingénieur, éventuellement de manière plus qualitative que quantitative, puisqu'il peut et doit aider le même ingénieur à concevoir les méthodes numériques qu'il utilisera.

2.4 Formes canoniques : similitude

Le calcul des *formes canoniques* ou *formes normales de matrices* est un problème central de l'algèbre linéaire. Il est difficile de donner une vraie définition, on peut tout de même dire qu'une forme est dite normale modulo une certaine relation d'équivalence quand elle représente les éléments de sa classe et qu'elle en révèle une « certaine propriété » commune. En corollaire, en révélant cette propriété commune, la forme canonique revêt elle-même, en général, une structure « simple » et permet de simplifier d'autant les calculs.

Ainsi, si la relation d'équivalence visée est la similitude de matrices sur un corps, les formes auxquelles on se trouve confronté sont la *forme polycyclique* (*forme d'Hessenberg à décalage*), la *forme canonique rationnelle* (*forme de Frobenius, forme de Jordan rationnelle*) et la *forme de Jordan*. La structure révélée des classes d'équivalence est ici la structure de l'espace décomposé en différents types de sous-espaces invariants par les applications linéaires associées.

Tout ceci est largement connu d'un point de vue théorique (le problème de l'épreuve du concours aux ENSI en Maths Appli. en traitait en 94). En revanche, d'un point de vue algorithmique, en particulier en calcul formel, de nombreuses difficultés se posent toujours. Nous n'entrerons pas ici dans les détails. L'algorithmique actuelle repose en grande partie sur les livres classiques tels que celui de Mac Duffee [27], de Newman [26], de Gantmacher [23] et ceux de Gohberg, Lancaster et Rodman concernant les matrices polynomiales [25, 24]. Rappelons ici encore le livre de Bini et Pan [15]. Pour aborder un thème applicatif, celui de la théorie du contrôle, nous ferons appel à l'ouvrage de Kailath [28].

Quant à Maple, nous avons eu l'occasion, au paragraphe précédent, d'utiliser les formes de Frobenius et de Jordan, nous n'y reviendrons pas. Les premiers algorithmes polynomiaux exacts pour les calculer ont été donnés assez récemment [29, 30]. Le thème est en plein développement depuis.

Maple permet aussi d'aller plus loin avec par exemple le calcul de l'exponentielle d'une matrice. Les formes normales sont en effet une clé possible pour calculer des fonctions de matrices. Les applications sont nombreuses notamment pour ce qui est de la résolution d'équations différentielles.

Pour terminer ce bref survol, on peut dire que les développements les plus récents de la recherche visent à mettre au point l'algorithmique des faisceaux de matrices [23, 31]. Il s'agit de calculer des formes normales de n-uplets de matrices, par l'obtention d'une transformation et son application simultanée à toutes les matrices du n-uplet.

2.5 Formes canoniques : équivalence

À la manière de Gantmacher [23], la théorie du paragraphe précédent peut aussi être appréhendée par le biais de l'équivalence de matrices polynomiales. À nouveau nous nous restreindrons ici aux matrices carrées inversibles, mais sans perte de généralité. Nous allons considérer des matrices dont les coefficients sont des polynômes sur un corps K, par exemple le corps des rationnels.

2.5.1 Premières notions

Définition 1 *Une matrice unimodulaire sur $K[x]$ est une matrice inversible. Elle est donc carrée et son déterminant est un élément non nul de K.*

Définition 2 *Deux matrices $A(x)$ et $B(x)$ sur $K[x]$ sont dites associées ou équivalentes à gauche s'il existe une matrice unimodulaire $U(x)$ telle que $A(x) = U(x)B(x)$. Les deux matrices sont dites équivalentes s'il existe deux matrices unimodulaires $U(x)$ et $V(x)$ telles que $A(x) = U(x)B(x)V(x)$.*

Ces notions d'équivalence nous permettent d'introduire les formes normales d'Hermite et de Smith des matrices sur $K[x]$.

Théorème 1 *Une matrice $A(x)$ de $K[x]$ est dite sous forme normale d'Hermite si elle est triangulaire supérieure, ses polynômes diagonaux sont unitaires et les polynômes hors diagonaux sont de degrés inférieurs à celui du polynôme diagonal de la même colonne. Toute matrice est associée à gauche à une unique matrice $H(x)$ sous forme d'Hermite, $H(x) = U(x)A(x)$.*

```
      [  x - 1    x    x + 1 ]
      [  2                   ]
 A := [ x   - 1    x      x   ]
      [  x - 1  x + 2  x - 2 ]
hermite(A,x);
     [ x - 1  0       5/2 x + 1     ]
     [   0    1         -3/2        ]
     [                    2         ]
     [   0    0   2/5 x + x   + 2/5 ]
```

Théorème 2 *Une matrice $A(x)$ de $K[x]$ est dite sous forme normale de Smith si elle est diagonale, ses polynômes diagonaux sont unitaires et chacun divise le suivant. Toute matrice est équivalente à une unique matrice $S(x)$ sous forme de Smith, $S(x) = U(x)A(x)V(x)$.*

```
smith(A,x);
 [ 1  0                0              ]
 [ 0  1                0              ]
 [                             2      ]
 [ 0  0  1/5 (x - 1) (2 x + 5 x   + 2) ]
```

Les polynômes diagonaux de la forme de Smith sont appelés polynômes invariants de la matrice $A(x)$. Ils caractérisent complètement une classe d'équivalence : deux matrices équivalentes ont les mêmes polynômes invariants. Comme dit plus haut, le lien avec les formes canoniques sous la relation de similitude est classique :

Théorème 3 *Soit B une matrice carrée sur K et F sa forme de Frobenius. Alors les polynômes invariants de la matrice caractéristique $B - xI$ sont les polynômes associés aux blocs compagnons de F.*

```
          [   193/58     53/116    255/116    213/58     9/58   ]
          [   -95/29     25/58    -231/58    -79/29    55/58   ]
     B := [   -48/29     12/29     -55/29    -68/29    -9/29   ]
          [    53/29     41/58     100/29     63/29    47/116  ]
          [    72/29    -40/29      68/29     44/29   -30/29   ]
```

```
frobenius(B); smith(B-x,x);
```

```
     [ 0  -1  0  0   0 ]     [ 1  0  0     0          0       ]
     [                 ]     [ 0  1  0     0          0       ]
     [ 1   0  0  0   0 ]     [ 0  0  1     0          0       ]
     [ 0   0  0  0   3 ]     [            2                   ]
     [                 ]     [ 0  0  0  x  + 1        0       ]
     [ 0   0  1  0  -1 ]     [                                ]
     [                 ]     [                    3      2    ]
     [ 0   0  0  1   3 ]     [ 0  0  0     0     x  - 3 x  + x - 3 ]
```

Voilà pour quelques aspects théoriques, intéressons-nous maintenant aux aspects algorithmiques. Certains résultats relèvent du domaine de la recherche puisque les complexités des problèmes ne sont pas encore pleinement maîtrisées.

2.5.2 Premières notions de coût

Le calcul des réductions (calcul de la forme normale et des matrices de passage) est polynomial depuis le travail de Kannan [36] pour la forme d'Hermite et depuis une contribution de l'auteur [35] pour la forme de Smith.

• **Forme d'Hermite : calcul direct.**
La première difficulté, concernant la forme d'Hermite, réside dans la croissance des tailles des coefficients des polynômes au cours des calculs (rappelons-nous ce que nous avons vu avec les systèmes linéaires : croissance exponentielle *versus* polynomiale). Le problème est analogue à celui qui se pose avec l'algorithme d'Euclide quand il s'agit de calculer le pgcd de deux polynômes (à une variable). Nous supposons que l'algorithme d'Euclide est bien connu. Le nombre d'opérations nécessaires pour calculer le pgcd de deux polynômes de degré n dans $K[x]$ est en $\mathcal{O}(n^3)$ (algorithme naïf) en terme d'opérations dans K [17], mais en terme de coût binaire, par exemple sur les rationnels, les coefficients des polynômes souffrent en général d'une croissance exponentielle de leur taille.

Le problème se pose de façon équivalente si l'on calcule la forme d'Hermite par triangularisation directe de la matrice. On entend par là un algorithme de triangularisation de la matrice via, par définition, des matrices unimodulaires directement dans $K[x]$ (c'est ainsi que fonctionne la fonction MAPLE `hermite`). Plus précisément pour mettre un zéro en position $(2,1)$ de la matrice

$$A(x) = \begin{bmatrix} p & r \\ q & s \end{bmatrix}$$

on peut considérer l'identité de Bezout : $up + vq = g$ où g est le pgcd de p et q, et construire la matrice :

$$U(x) = \begin{bmatrix} u & v \\ -q/g & p/g \end{bmatrix}$$

qui est bien unimodulaire et qui conduit à

$$H'(x) = U(x)A(x) = \begin{bmatrix} g & ur + vs \\ 0 & (-qr + ps)/g \end{bmatrix}$$

Les coefficients de $H'(x)$ sont tous des polynômes puisque la transformation $U(x)$ est unimodulaire. En généralisant ceci, une élimination de « type Gauss » conduit facilement à une matrice triangulaire. Éventuellement, cette matrice n'est pas encore sous forme d'Hermite : nous l'avons notée $H'(x)$ ci-dessus : les polynômes hors-diagonaux peuvent ne pas être réduits en degrés. On se convaincra que des opérations de lignes supplémentaires peuvent être effectuées pour réduire les degrés mais sans modifier la structure triangulaire déjà atteinte.

Cette méthode paraît satisfaisante, il n'en n'est rien. Il a fallu beaucoup d'astuce à Kannan [36] pour parvenir à démontrer qu'un tel point de vue permettait d'obtenir un algorithme polynomial (« avec des bornes astronomiques » aux dires mêmes de l'auteur).

• **La matrice de Sylvester.**
Une solution beaucoup plus élégante consiste à se ramener à un problème d'algèbre linéaire sur le corps K (le paragraphe 2.2 a essayé de montrer pourquoi alors la résolution peut être menée efficacement). L'idée est assez récente [28], elle a été mise en œuvre pour la forme d'Hermite par Labhalla, Lombardi et Marlin [34]. Nous la préciserons au paragraphe suivant, essayons ici de la comprendre dans le cas d'une matrice 2×1, c'est-à-dire pour deux polynômes. Notons au passage

qu'un calcul de forme d'Hermite peut être vu comme un « gros calcul de pgcd » : c'est d'ailleurs un calcul de pgcd de matrices.

Pour le calcul du pgcd de polynômes donc, la technique utilisée est celle des sous-résultants [18, 19]. Pour une très vague – et imprécise – idée : soit à calculer le pgcd des deux polynômes suivants :

```
print(p,q);
          4       3        2                 4       3        2
        x  - 10 x  + 35 x  - 50 x + 24, x  - 15 x  + 73 x  - 129 x + 70
```

D'après l'identité de Bezout, il existe deux polynômes u et v de degrés au plus 4 (ici) tels que $up+vq$ soit égal au pgcd recherché. Considérons l'espace vectoriel des polynômes de degrés au plus 8 (donnés par les vecteurs de leurs coefficients). C'est donc dire que le pgcd appartient à l'espace vectoriel engendré par p, xp, x^2p, x^3p, x^4 et q, xq, x^2q, x^3q, x^4q. En considérant ces vecteurs par ligne on construit la matrice suivante, dite *matrice de Sylvester* :

```
S:=sylvester(p,q,x);
            [ 1  -10   35   -50   24     0     0    0 ]
            [ 0   1   -10   35    -50   24     0    0 ]
            [ 0   0    1    -10   35    -50   24    0 ]
            [ 0   0    0     1    -10   35    -50  24 ]
            [ 1  -15   73  -129   70     0     0    0 ]
            [ 0   1   -15   73   -129   70     0    0 ]
            [ 0   0    1    -15   73   -129   70    0 ]
            [ 0   0    0     1    -15   73   -129  70 ]
```

Dire que le pgcd est dans l'espace vectoriel des lignes de S indique que par combinaison « judicieuse » de ces lignes, on doit pouvoir retrouver le vecteur donnant le pgcd. Il suffit en fait de triangulariser la matrice :

```
gausselim(S);
             [ 1  -10   35  -50   24   0    0    0 ]
             [ 0   1   -10   35  -50   24   0    0 ]
             [ 0   0    1   -10   35  -50   24   0 ]
             [ 0   0    0    1   -10   35  -50  24 ]
             [ 0   0    0    0    -5   38  -79  46 ]
             [ 0   0    0    0    0    1    -3   2 ]
             [ 0   0    0    0    0    0    0    0 ]
             [ 0   0    0    0    0    0    0    0 ]
```

la dernière ligne nous indique que le pgcd est $g = x^2-3*x+2$. Mais arrêtons-nous là ! Les références sur le sujet se trouvent facilement. Retenons que : le calcul du pgcd se ramène, ou plutôt *se réduit* – en terme de complexité – à la mise sous forme triangulaire d'une matrice : pour deux polynômes $p = p_nx^n + p_{n-1}x^{n-1} + \ldots + p_1x + p_0$ et $q = q_mx^m + q_{m-1}x^{m-1} + \ldots + q_1x + q_0$ la matrice associée est :

$$S(p,q) = \begin{bmatrix} p_n & p_{n-1} & \ldots & \ldots & p_1 & p_0 & 0 & \ldots & \ldots & 0 \\ 0 & p_n & p_{n-1} & \ldots & \ldots & p_1 & p_0 & 0 & \ldots & 0 \\ \vdots & \ddots & \ddots & \ddots & \ddots & \ddots & \ddots & \ddots & \ddots & \vdots \\ 0 & \ldots & 0 & p_n & p_{n-1} & \ldots & \ldots & p_1 & p_0 & 0 \\ 0 & \ldots & \ldots & 0 & p_n & p_{n-1} & \ldots & \ldots & p_1 & p_0 \\ q_m & q_{m-1} & \ldots & \ldots & q_1 & q_0 & 0 & \ldots & \ldots & 0 \\ 0 & q_m & q_{m-1} & \ldots & \ldots & q_1 & q_0 & 0 & \ldots & 0 \\ \vdots & \ddots & \ddots & \ddots & \ddots & \ddots & \ddots & \ddots & \ddots & \vdots \\ 0 & \ldots & 0 & q_m & q_{m-1} & \ldots & \ldots & q_1 & q_0 & 0 \\ 0 & \ldots & \ldots & 0 & q_m & q_{m-1} & \ldots & \ldots & q_1 & q_0 \end{bmatrix} \in K^{m+n,m+n}. \tag{2.2}$$

Problème : quid de la réciproque?

Nous allons voir au paragraphe suivant que cette technique, développée initialement pour le calcul du pgcd de polynômes, peut être généralisée pour calculer la forme d'Hermite.

• **Forme de Smith : davantage de problèmes.**
Pour la forme de Smith la situation est un peu plus compliquée. Les premiers algorithmes (*i.e.* les applications directes des méthodes mathématiques) consistent à utiliser le calcul de la forme d'Hermite comme « boîte noire ». On peut schématiser les choses ainsi : on calcule la forme d'Hermite $H_1(x)$ de $A(x)$ puis la forme d'Hermite $H_2(x)$ de ${}^tH_1(x)$, puis la forme d'Hermite de ${}^tH_2(x)$ ... jusqu'à obtenir une matrice diagonale. On montre que ce processus se termine en $\mathcal{O}(n^3)$ étapes au pire ($A(x)$ de dimension et de degré n).

Une telle approche n'est pas satisfaisante du point de vue algorithmique : le problème de la croissance des coefficients (avec lequel nous commençons à être familiers) se pose ici dramatiquement. Est-il seulement possible de montrer que ces itérations ne conduisent pas à une croissance exponentielle ? La question était posée. Elle a été contournée et ce sont d'autres méthodes qui permettent de calculer la forme de Smith en temps polynomial.

2.5.3 Algorithmes déterministes

Nous allons voir comment la méthode utilisant la matrice de Sylvester peut être généralisée au cas des matrices de polynômes [37, 34]. On se placera toujours dans le cas carré régulier. Auparavant, voyons-en un thème applicatif.

• **Un exemple d'application.**
Un grand domaine applicatif pour la forme d'Hermite est celui de l'automatique. Nous renvoyons ici au livre de Kailath [28]. Par exemple, il peut s'agir en théorie du contrôle d'étudier le comportement d'un système linéaire constant du type :

$$\begin{cases} x'(t) = Ax(t) + Bu(t) \\ v(t) = Cx(t) \end{cases}$$

Le vecteur $u(t) \in K^m$ est dit *vecteur d'entrées*, le vecteur $v(t) \in K^p$ est dit *vecteur de sorties* et $x \in K^n$ est le *vecteur d'états*. La donnée des trois matrices (C, A, B) (avec les dimensions adéquates) est la donnée d'une réalisation de la *matrice de transfert* du système :

$$H(s) = C(sI - A)^{-1}B.$$

Par transformée de Laplace cette matrice de transfert transforme les entrées en les sorties.

Le problème des formes canoniques est ici classique et s'applique aux réalisations : il s'agit de trouver des réalisations canoniques. Cela peut être une réalisation avec les dimensions les plus petites possibles et dont les matrices ont une structure particulière. Nous voyons que le lien avec les formes canoniques de matrices est « immédiat ». Le lecteur se reportera à l'ouvrage de Kailath [28] pour un exposé limpide sur le sujet. Retenons simplement, pour l'illustration, que la matrice de transfert est une matrice de fractions rationnelles qui peut s'écrire comme quotient de deux matrices polynomiales :

$$H(s) = N(s)/D(s)$$

et que, de même que l'on réduit une fraction de polynômes en simplifiant par le pgcd, aller vers une réalisation canonique du système demande ici de simplifier $H(s)$ par un calcul de pgcd matriciel avec $N(s)$ et $D(s)$. Cette simplification, qui a en fait pour effet de « normaliser » les degrés des polynômes, permet de calculer une réalisation avec A de dimension minimale. Le test de primalité de matrices aussi est important pour les tests PBH (Popov-Belevitch-Hautus) de contrôlabilité ou d'observabilité. Intuitivement, le système est dit contrôlable si étant donné un état il existe une entrée qui conduit à lui, il est dit observable si étant données entrées et sorties, il existe un état qui leur soit compatible. Les tests PBH nous disent que le système est contrôlable si $sI - A$ et B sont premières entre elles à gauche, qu'il est observable si C et $sI - A$ sont premières entre elles à droite.

• **Le pgcd de matrices.** Le lecteur se reportera au livre de Mac Duffee [27] pour les définitions que nous reprenons.

Définition 3 *Un pgcd de matrice $A(x)$ et $B(x)$ à droite (noté pgcdd) est une matrice $G(x)$ qui vérifie :*

- $G(x)$ est un diviseur à droite de $A(x)$ et $B(x)$: il existe $\bar{A}(x)$ et $\bar{B}(x)$ telles que

$$A(x) = \bar{A}(x)G(x), \quad B(x) = \bar{B}(x)G(x)$$

- si $G_1(x)$ est un autre diviseur commun à droite alors ($G(x)$ en est un multiple) il existe une matrice $W(x)$ telle que $G(x) = W(x)G_1(x)$.

De même que le pcgd de deux polynômes n'est unique que quand on choisit un représentant (par exemple celui qui est unitaire) particulier dans la classe des pgcd possibles, le pgcd de matrices n'est pas unique, le choix d'un représentant particulier peut être celui de la forme d'Hermite :

Lemme 1 *Pour deux matrices $A(x)$ et $B(x)$ (carrées régulières) calculons la forme d'Hermite de ${}^t[A(x), B(x)]$:*

$$H(x) = \begin{bmatrix} U_{11}(x) & U_{12}(x) \\ U_{21}(x) & U_{22}(x) \end{bmatrix} \begin{bmatrix} A(x) \\ B(x) \end{bmatrix} = \begin{bmatrix} G(x) \\ 0 \end{bmatrix}$$

alors $G(x)$ est un pgcdd de $A(x)$ et $B(x)$.

Preuve. La matrice $U(x)$ est unimodulaire, soit $V(x)$ son inverse, en écrivant ${}^t[A(x), B(x)] = V(x){}^t[G(x), 0]$ on obtient que $G(x)$ est un diviseur à droite. On a aussi $G(x) = U_{11}(x)A(x) + U_{12}(x)B(x)$ (identité de Bezout) donc si $G_1(x)$ est un autre diviseur à droite, avec $A(x) = A_1(x)G_1(x)$ et $B(x) = B_1(x)G_1(x)$ on arrive à $G(x) = [U_{11}(x)A_1(x) + U_{12}(x)B_1(x)]G_1(x)$, ce qui termine la démonstration.

```
print(A,B);
                       [         2     ]
   [ x - 1  x - 1 ]  [ x - 1  x  - 1 ]
   [   x      x   ]  [   x        x  ]
hermite(stack(A,B),x);
              [ 1     1   ]
              [       2   ]
              [ 0  x  - x ]
              [ 0     0   ]
              [ 0     0   ]
G:=submatrix(",1..2,1..2);
              [ 1     1   ]
              [       2   ]
              [ 0  x  - x ]
```

Définition 4 *Deux matrices $A(x)$ et $B(x)$ sont dites premières entre elles à droite si leurs pgcdd sont des matrices unimodulaires.*

```
RightCoPrime(A,B,x);
 false
print(C,F);
   [ x + 1  x - 1 ]  [ x + 1    x   ]
   [   x      x   ]  [   x    x + 2 ]
RightCoPrime(C,F,x);
 true
```

(Cet extrait est en MAPLE grenoblois ... il n'est pas livré avec la version publique ...). Voilà, nous n'irons pas plus loin dans cette partie de la théorie. Sachons seulement que beaucoup des techniques polynomiales s'appliquent aux matrices, c'est ce que nous allons voir en particulier avec l'utilisation de la matrice de Sylvester.

• **Un algorithme pour les formes d'Hermite et de Smith.** Nous n'allons pas rentrer dans les détails, mais nous espérons qu'avec ce qui précède (identité (2.2)) tout sera un tant soit peu intuitif !

Pour deux matrices $A(x)$ et $B(x)$ on construit une matrice de Sylvester généralisée sur le modèle de la matrice de Sylvester pour deux polynômes donnée par l'équation (2.2). On suppose que les deux matrices sont de degré n. Si l'on définit pour un polynôme :

$$S(p)=\begin{bmatrix} p_n & p_{n-1} & \dots & \dots & p_1 & p_0 & 0 & \dots & \dots & 0\\ 0 & p_n & p_{n-1} & \dots & \dots & p_1 & p_0 & 0 & \dots & 0\\ \vdots & \ddots & \ddots & \ddots & \ddots & \ddots & \ddots & \ddots & \ddots & \vdots\\ 0 & \dots & 0 & p_n & p_{n-1} & \dots & \dots & p_1 & p_0 & 0\\ 0 & \dots & \dots & 0 & p_n & p_{n-1} & \dots & \dots & p_1 & p_0 \end{bmatrix}\in K^{n^2,n+n^2} \tag{2.3}$$

la matrice de Sylvester généralisée associée à $A(x)$ et $B(x)$ est :

$$S(A,B)=\begin{bmatrix} S(A_{11}) & S(A_{12}) & \dots & S(A_{1n})\\ \dots & \dots & \dots & \dots\\ S(A_{n1}) & S(A_{n2}) & \dots & S(A_{nn})\\ S(B_{11}) & S(B_{12}) & \dots & S(B_{1n})\\ \dots & \dots & \dots & \dots\\ S(B_{n1}) & S(B_{n2}) & \dots & S(B_{nn}) \end{bmatrix}\in K^{n^3,n^2+n^3} \tag{2.4}$$

On peut alors montrer que par triangularisation par opérations de lignes, la forme normale d'Hermite ${}^t[G(x),0]$, *i.e.* le pgcdd de $A(x)$ et $B(x)$ peut être calculé. Regardons sur un exemple de dimension 2 (toujours avec des instructions grenobloises) :

```
print(A,B);
                       [          2     ]
   [ x - 1  x - 1 ]  [ x - 1  x   - 1 ]
   [   x      x   ]  [   x        x   ]

G:=RightGenSylvester(A,B,x);
     [ 0  1  -1   0  0  1  -1   0 ]
     [ 0  0   1  -1  0  0   1  -1 ]
     [ 0  1   0   0  0  1   0   0 ]
     [ 0  0   1   0  0  0   1   0 ]
     [ 0  1  -1   0  1  0  -1   0 ]
     [ 0  0   1  -1  0  1   0  -1 ]
     [ 0  1   0   0  0  1   0   0 ]
```

```
     [ 0  0   1   0  0  0   1   0 ]

gausselim(G);
       [ 0  1  -1   0  0   1  -1   0 ]
       [ 0  0   1  -1  0   0   1  -1 ]
       [ 0  0   0   1  0   0   0   1 ]
       [ 0  0   0   0  1  -1   0   0 ]
       [ 0  0   0   0  0   1  -1   0 ]
       [ 0  0   0   0  0   0   0   0 ]
       [ 0  0   0   0  0   0   0   0 ]
       [ 0  0   0   0  0   0   0   0 ]
```

En utilisant comme « patron de structure » la définition de la matrice de Sylvester généralisée, on découpe la matrice résultat en deux blocs de colonnes et deux blocs de lignes :

```
print(G11,G12); print(G21,G22);
   [ 0  1  -1   0 ]  [ 0  1  -1   0 ]
   [ 0  0   1  -1 ], [ 0  0   1  -1 ]
   [ 0  0   0   1 ]  [ 0  0   0   1 ]

    [ 0  0  0  0 ]  [ 1  -1   0  0 ]
    [ 0  0  0  0 ]  [ 0   1  -1  0 ]
    [ 0  0  0  0 ], [ 0   0   0  0 ]
    [ 0  0  0  0 ]  [ 0   0   0  0 ]
    [ 0  0  0  0 ]  [ 0   0   0  0 ]
```

les dernières lignes de chaque bloc nous donnant le coefficient correspondant du pgcdd (nous l'avions déjà calculé plus haut) :

$$\begin{array}{lcl} (0,0,0,1) & \to & g_{11} = 1 \\ (0,0,0,1) & \to & g_{12} = 1 \\ (0,0,0,0) & \to & g_{21} = 0 \\ (0,1,-1,0) & \to & g_{22} = x^2 - x \end{array}$$

Donc : nous avons donné l'idée du fait que le calcul d'une forme d'Hermite d'une matrice polynomiale de dimensions et degré n se ramène à une élimination de Gauss sur une matrice constante de dimensions $\mathcal{O}(n^3)$ et pouvons, pour en déduire la complexité, utiliser les résultats concernant ce dernier problème.

Pour le calcul de la forme de Smith, on pourrait montrer que si pour Hermite, une élimination de Gauss suffit, on aurait besoin de mettre au point une élimination de type Gauss-Jordan, mais c'est un autre sujet.

2.5.4 Algorithmes probabilistes

Pour terminer ce paragraphe sur les difficultés à mettre au point des algorithmes peu onéreux, nous allons donner deux exemples d'algorithmes probabilistes. En effet si les algorithmes précédents calculent à tous les coups une solution au problème, on peut imaginer des algorithmes qui ne produisent la bonne réponse qu'avec une certaine probabilité. Si cette probabilité est aussi grande que l'on veut et que l'on peut vérifier le résultat, alors ce peut être un bon moyen d'accélérer les calculs à moindre risque. Pour un approfondissement de la question, le lecteur se reportera par exemple au livre de Brassard et Bratley [39].

Le paragraphe est illustré par l'obtention d'algorithmes pour le calcul de la forme normale de Smith (nous n'en avons pas encore donné de satisfaisant).

Une deuxième illustration de ce type d'algorithmes sera trouvée au paragraphe 3.1.5 avec la recherche d'une chaîne de caractères dans un fichier.

• **Un exemple typique.** L'exemple typique en calcul formel est donné par l'application d'un lemme de J.T. Schwartz [38]. Cet auteur a posé la base de la plupart des autres démonstrations produites sur le sujet. Soit un polynôme inconnu $P(x_1, x_2, \ldots, x_n)$ à n indéterminées et de degré n en chacune. On suppose que l'on est capable d'évaluer P en n'importe quel point donné. La question est de savoir si P est identiquement nul ou pas ? De manière déterministe, la résolution est « inenvisageable » elle demanderait en effet un nombre exponentiel d'évaluations, de tests. Il est ici naturel de résoudre le problème de manière probabiliste. En choisissant « assez » de points $X_1, \ldots X_N$ d'évaluation, on conclura que si pour tous ces points l'évaluation retourne zéro, alors le polynôme est nul. Ceci se formalise comme suit.

Lemme 2 *Soit $P(x_1, x_2, \ldots, x_n)$ un polynôme sur un corps K à n indéterminées et de degré n en chacune. Soit I un sous-ensemble fini de K de cardinal c et $X = (\chi_1, \chi_2, \ldots, \chi_n)$ un n-uplet tiré uniformément dans I^n alors :*

$$Prob(P(\chi_1, \chi_2, \ldots, \chi_n) = 0) \leq n/c.$$

Cette propriété permet de mettre au point un algorithme de test de Monte-Carlo.

Dans ce cas l'algorithme de test de nullité consiste à évaluer le polynôme en un nombre suffisament grand de points. Si toutes les évaluations donnent zéro, alors le polynôme est « décrété » nul. On peut montrer que cela peut se faire avec une probabilité d'échec tendant vers zéro quand la dimension du problème augmente. Cet algorithme sera dit de Monte-Carlo, au contraire de Las Vegas, en ce sens que l'on n'a aucun moyen (autre que celui de tester un nombre exponentiel de points) d'être sûr que le polynôme est vraiment nul.

Définition 5 *L'algorithme probabiliste est dit de Monte-Carlo s'il peut donner une réponse erronée sans indication d'échec, mais cela avec une probabilité plus petite qu'une constante (disons $1/2$).*

Ceci peut être encore perçu comme peu satisfaisant (bien que très puissant dans certaines situations). Il existe une autre classe d'algorithmes probabilistes, comme nous le verrons pour la forme de Smith, les algorithmes dits de Las Vegas. Avec ces derniers, il est possible de vérifier la réponse (ouf !).

Définition 6 *L'algorithme probabiliste est dit de Las Vegas s'il délivre, soit la bonne réponse, soit l'indication qu'il lui est impossible de donner le résultat. Un échec ne se produit qu'avec une probabilité plus petite qu'une constante (disons $1/2$).*

• **Vers Monte-Carlo.** Essayons d'appliquer ceci au calcul de la forme normale de Smith. En suivant Kaltofen, Krishnamoorthy et Saunders [32], nous avons pour cela besoin d'une nouvelle caractérisation de la forme (la forme a été définie au paragraphe 2.5.1).

Lemme 3 *Soit $A(x)$ et $S(x) = diag(s_1(x), \ldots, s_n(x))$ sa forme de Smith. Les $s_i(x)$ sont donc les polynômes invariants de $A(x)$. Notons s_i^*, $1 \leq i \leq n$, le pgcd des tous les mineurs $i \times i$ de $A(x)$. Alors les polynômes invariants sont donnés par $s_1 = s_1^*$ et $s_i = s_i^*/s_{i-1}^*$, $2 \leq i \leq n$.*

L'analogie avec le test de nullité du polynôme est « immédiate ». En effet, vouloir calculer de manière sûre la forme de Smith par cette caractérisation, par exemple calculer $s_i(x)$, amènerait à calculer tous les mineurs d'ordre i et ceux d'ordre $i-1$, or ......il y en a exponentiellement beaucoup !

Donc : on probabilise. Nous ne ferons pas la preuve de l'algorithme, mais donnons son fonctionnement. Fondamentalement, pour appliquer le lemme 2, il faut se ramener à la situation dans laquelle la matrice $A(x)$ est « générique », c'est-à-dire qu'en calculant le pgcd sur seulement quelques mineurs d'ordre i il faut que l'on ait une bonne chance de calculer la valeur correcte de s_i*. En fait, on peut faire beaucoup mieux et se limiter à calculer seulement deux mineurs d'ordre i.

On tire au hasard quatre matrices P_1, Q_1, P_2 et Q_2 constantes inversibles (on peut les prendre triangulaires à diagonale unité). Cela nous donne deux copies perturbées de $A(x)$:

$$\begin{cases} A_1(x) = P_1 A(x) Q_1, \\ A_2(x) = P_2 A(x) Q_2. \end{cases}$$

Bien sûr, par construction, $A_1(x)$ et $A_2(x)$ sont équivalentes à $A(x)$ et ont la même forme de Smith. Pour calculer s_i*, $1 \leq i \leq n$, on calcule les mineurs principaux $\Delta_i^{(1)}(x)$ et $\Delta_i^{(2)}(x)$ d'ordre i de $A_1(x)$ et $A_2(x)$. On montre qu'avec une bonne probabilité, si les coefficients de P_1, Q_1, P_2 et Q_2 sont tirés dans un ensemble assez grand en fonction de $A(x)$, alors s_i* est le pgcd de $\Delta_i^{(1)}(x)$ et $\Delta_i^{(2)}(x)$ [32]. De là on calcule les facteurs invariants, donc la forme de Smith, en n divisions supplémentaires.

```
print(A);
        [                2    ]
        [     x - 1     x  - 1 ]
        [  2                  ]
        [ x  - 4 x + 3   x - 1 ]

print(A1,A2);
  [              2                  2 ]  [              2   2     ]
  [   3 x - 4 + x      4 x - 6 + 2 x  ]  [ 2 x - 6 + 4 x   x  - 1 ]
  [    2               2              ]  [    2                   ]
  [ 3 x  - 11 x + 8  4 x  - 14 x + 10 ]  [ 2 x  - 4 x + 2   x - 1 ]

s1star:=gcd(A1[1,1],A2[1,1]);      # star signifie "etoile" en anglais
      s1star := x - 1
s2star:=gcd(det(A1),det(A2));
                3      2                 4
 s2star := - 8 x  + 2 x  + 12 x - 8 + 2 x
s2:=quo(s2star,s1star,x);
                 3      2
        s2 := 2 x  - 6 x  - 4 x + 8
smith(A,x);
     [ x - 1          0         ]
     [          3      2        ]
     [   0     x  - 3 x  - 2 x + 4 ]
```

Une nouvelle fois, nous avons ramené un problème, *a priori* plus compliqué, à un problème de calcul de déterminant. L'algorithme de calcul de la forme de Smith a ici un coût asymptotiquement égal à celui d'une élimination de Gauss sur l'anneau des polynômes. Certaines limitations peuvent se présenter bien sûr si le corps sur lequel on travaille est trop petit (*e.g.* un corps fini), mais c'est une autre histoire.

Encore une fois, ceci peut n'être considéré comme à moitié satisfaisant seulement puisque l'on ne dispose pas de moyen de vérifier que la forme calculée est effectivement celle de Smith.

• **Vers Las Vegas.** En suivant à nouveau Kaltofen, Krishnamoorthy et Saunders [33], voyons maintenant un autre algorithme probabiliste, mais de Las Vegas cette fois-ci. Si l'algorithme retourne une matrice, alors c'est la forme de Smith. On va utiliser une nouvelle caractérisation de la forme d'Hermite :

Lemme 4 *Soit $A(x)$. On note h_i, $1 \leq i \leq n$, le i-ème coefficient diagonal de sa forme d'Hermite. De même, soit h_i^*, $1 \leq i \leq n$, le pgcd des tous les mineurs $i \times i$ de $A(x)$ construits sur ses i premières colonnes. Alors les h_i sont donnés par $h_1 = h_1^*$ et $h_i = h_i^*/h_{i-1}^*$, $2 \leq i \leq n$.*

En termes plus clairs, concernant le i-ème coefficient diagonal on utilise les mineurs d'ordre i et $i-1$, pour la forme d'Hermite on regarde les mineurs sur les i premières colonnes, pour la forme de Smith on regarde sur toutes les colonnes. La clé est là : dans le cas « générique » les deux caractérisations coïncident. C'est dire que les coefficients diagonaux de la forme d'Hermite sont aussi ceux de la forme de Smith.

Comme dans le cas de l'algorithme de Monte-Carlo, fondamentalement, pour appliquer le lemme 2, il faut se ramener à la situation dans laquelle la matrice $A(x)$ est « générique ». Ici, on veut qu'en calculant la forme d'Hermite, la diagonale donne la forme de Smith !

L'algorithme consiste donc à tirer aléatoirement une matrice constante inversible P (on la prend triangulaire inférieure à diagonale unité) et on perturbe la matrice A en $A_1 = AP$ qui a la même forme de Smith. Si A_1 est bien choisie alors la forme d'Hermite de A_1 donne effectivement la forme de Smith [33].

```
print(A);
        [                2     ]
        [     x - 1     x  - 1 ]
        [  2                   ]
        [ x  - 4 x + 3   x + 2 ]

hermite(A,x);
    [               2           ]
    [ x - 1         x  - 1      ]
    [                   3     2 ]
    [   0    - 2 x + 1 + x  - 3 x  ]
```

Sur cette dernière sortie, la matrice $A(x)$ n'a pas été perturbée, le résultat n'est pas correct (ce n'est pas l'algorithme).

```
print(A1);
         [              2              2  ]
         [   3 x - 4 + x      4 x - 6 + 2 x  ]
         [    2                 2            ]
         [ 3 x  - 11 x + 11  4 x  - 14 x + 16 ]

hermite(A1,x);
             [      38  2    122       40  3    422 ]
             [ 1   --- x  + --- x - --- x  + --- ]
             [    103       309      309       309 ]
             [           3    2               4  ]
             [ 0    - 4 x  + x  + 3 x - 1 + x    ]
smith(A,x);
```

```
[ 1             0           ]
[        3    2            4 ]
[ 0  - 4 x  + x  + 3 x - 1 + x  ]
```

Voyons comment l'algorithme est de Las Vegas. On rajoute un test de vérification de la sortie. Ce test consiste à essayer de mettre à zéro les coefficients hors-diagonaux de la forme d'Hermite calculée (notons que c'est une procédure simple comparée au reste). Si l'on y parvient et que la matrice résultante est sous forme de Smith, on la retourne en résultat, sinon l'algorithme retourne *échec*.

Remarque. Au lecteur peu familier des algorithmes probabilistes qui serait encore sceptique à leur sujet, l'argument suivant n'est pas la preuve de quoi que ce soit mais ... Sous MAPLE ainsi que sous les autres systèmes de calcul formel, il est fréquent de rencontrer des algorithmes probabilistes. On pense ici à la résolution de problèmes d'arithmétique sur les entiers (*e.g.* factorisation, test de primalité), sur les polynômes (*e.g.* pgcd) ou à bien d'autres. Mais les probabilités d'échec sont en principe tellement faibles qu'un utilisateur « normal » ne se rend compte de rien.

2.6 Programmer sur des domaines plus compliqués

(Doit être complété).

2.7 Conclusion

Voilà pour une première incursion dans le domaine de l'algèbre linéaire exacte. Retenons deux ou trois choses :

- le calcul formel est un outil précieux quand il s'agit de traiter de *problèmes d'algorithmique mathématique*, nous l'avons illustré avec la manipulation de nombres algébriques. Il fait éventuellement primer les aspects qualitatifs sur les aspects quantitatifs.
- Par sa nature même, *le calcul formel manipule des données dans des domaines très variés.* Pour gagner en efficacité il faut faire appel à des *techniques sophistiquées, techniques aussi bien mathématiques qu'informatiques.*
- En algèbre linéaire comme ailleurs, *le problème de la gestion de la taille des données est crucial.*

Nous avons peu discuté des applications en vraie grandeur de ce que nous avons vu. Le domaine de l'automatique a été effleuré, une autre branche énorme serait concernée par la résolution d'équations différentielles. Ce sera pour une autre fois.

2.8 Travaux pratiques

I. FORME NORMALE DE SMITH

- **Utilisation des fonctions** MAPLE : `transpose`, `hermite`, `randmatrix`, `gausselim`.

- **Itérations.** Écrire une fonction calculant la forme de Smith par itérations successives à partir de la forme d'Hermite (cf. fin du paragraphe 2.5.2).

- **Monte-Carlo.** Écrire une fonction calculant la forme de Smith par l'algorithme de Monte-Carlo du paragraphe 2.5.4. Les mineurs seront tous calculés par une élimination de Gauss (cf. paragraphe 2.2).

- **Las Vegas.** Écrire une fonction calculant la forme de Smith par l'algorithme de Las Vegas du paragraphe 2.5.4. Ceci comprend l'écriture d'une fonction qui teste la validité du résultat.

Chapitre 3

Calcul dans les corps finis et Factorisation

Jean-Louis Roch

Dans toute la suite, K désigne un corps de caractéristique nulle ou un corps fini. $K[X]$ désigne l'anneau des polynômes à une indéterminée à coefficients dans le corps K.

$K[X]$ est un anneau principal et il est donc possible de calculer le pgcd de deux polynômes (défini à un élément inversible près) en utilisant l'algorithme d'Euclide ou ses variantes. $K[X]$ étant principal, il est aussi factoriel ; tout polynôme admet donc une unique décomposition en facteurs irréductibles.

Ainsi, en Maple, la primitive `factor` permet de factoriser un polynôme à coefficients entiers en facteurs irréductibles.

Exemple :

```
> Q := x^6 - 12*x^5 + 24*x^4 - 15*x^3 - 6*x^2 + 12*x - 8 ;
```

$$x^6 - 12\,x^5 + 24\,x^4 - 15\,x^3 - 6\,x^2 + 12\,x - 8$$

```
> factor(Q) ;
```

$$\left(2\,x^3 + 1\right)(x - 2)^3$$

Le programme par défaut utilisé par Maple pour effectuer ce calcul est basé sur l'algorithme de Berlekamp et ses variantes. Le but de ce chapitre est de présenter cet algorithme, qui tire parti du lien étroit qui existe entre $K[X]$ et l'anneau des matrices carrées, notamment lorsque K est un corps fini.

L'arithmétique dans les corps finis est particulièrement importante pour le problème de la factorisation. C'est pourquoi ce cours est structuré en deux grandes parties.

- La première présente comment il est possible de calculer dans le corps de Galois F_q avec Maple. Les propriétés élémentaires de F_q sont rappelées, ainsi que le théorème chinois des restes dans ses versions entières et polynomiales. L'arithmétique modulaire joue un rôle très important en calcul formel pour améliorer la performance de nombreux algorithmes. Nous montrons comment une telle arithmétique peut être utilisée pour rechercher toutes les occurrences d'une chaîne dans un fichier.

– La deuxième partie est consacrée à l'algorithme de Berlekamp, Le premier paragraphe explicite un algorithme permettant de se ramener au cas où le polynôme à factoriser a toutes ses racines simples. Le problème de la factorisation dans $F_p[x]$ est ensuite abordé. Il permet la construction d'un algorithme pour la factorisation d'un polynôme à coefficients entiers, en utilisant la technique de remontée modulaire de Hensel.

Ce cours constitue donc une initiation au problème de la factorisation et est illustré par l'utilisation de MAPLE. La plupart des résultats sont cités sans démonstration. Des études beaucoup plus complètes peuvent être trouvées dans [19, 13, 11, 7, 8].

Remarque
Nous nous limitons ici à la factorisation d'un polynôme à une variable. Cet algorithme peut être étendu, par homomorphisme, à la factorisation dans $K[X_1, \ldots, X_t]$ qui, bien que non principal, est un anneau factoriel.

3.1 Calcul dans un corps fini

Dans toute la suite, p désigne un nombre premier, $m \geq 1$ un entier et F_q le corps de Galois à $q = p^m$ éléments.

3.1.1 Calculer dans $\mathbf{F}_p$ avec MAPLE

Grâce à l'opérateur `mod`, il est possible de calculer la valeur modulo un entier k (*i.e.* dans l'anneau $\boldsymbol{Z}/k\boldsymbol{Z}$) d'une expression algébrique à plusieurs indéterminées.

Exemple :

```
> expr := 7 * x^3 * y + (x+9) / (y+4*z-3) ;
```

$$7\,x^3 y + \frac{x+9}{y+4\,z-3}$$

```
> expr mod 3 ;
```

$$x^3 y + \frac{x}{y+z}$$

Si k est un nombre premier p, les opérations arithmétiques sont effectuées dans le corps fini F_p, et en particulier le calcul de l'inverse.

Il est possible de spécialiser le comportement de l'opérateur `mod` pour des fonctions de l'utilisateur. Ainsi, si l'on a défini la fonction `` `mod/f` ``, alors l'expression `f(e) mod 3` est équivalente à l'appel `` `mod/f`(e, 3) ``.
MAPLE propose un certain nombre de fonctions pour effectuer des calculs d'arithmétique polynomiale et d'algèbre linéaire dans les corps finis. Ces fonctions, reconnues par `mod`, ont de manière générale leur premier caractère en majuscule. Le fonctionnement de l'opérateur `mod` dans ce cas est alors très différent selon que le nom de la fonction appelée dans l'expression de gauche commence ou non par une majuscule.

Exemple :

```
> P := x^2 - x + 1 ;
```

$$x^2 - x + 1$$

`> factor(P) ; #` Factorise P dans $\boldsymbol{Z}[x]$: P est irréductible

$$x^2 - x + 1$$

`> factor(P) mod 7 ; #` Factorise P dans $\boldsymbol{Z}[x]$ et applique l'opérateur `mod` au coefficient du résultat

$$x^2 + 6\,x + 1$$

`> Factor(P) mod 7 ; #` Factorise P dans $\boldsymbol{Z}/7\boldsymbol{Z}[x]$: P n'est pas irréductible

$$(x + 4)\,(x + 2)$$

3.1.2 Calculer dans $\mathbf{F}_q$ avec MAPLE

Grâce à la bibliothèque `GF`, MAPLE permet de calculer dans F_q. Le corps fini de Galois à $q = p^m$ éléments est isomorphe au corps $\mathrm{F}_p[x]/a(x)$ où $a(x)$ est un polynôme irréductible de degré m de $\mathrm{F}_p[x]$.
L'appel de fonction `GF(p,m)` permet de construire F_q, $q = p^m$, à partir de cette remarque. MAPLE choisit de manière arbitraire un polynôme a de degré m unitaire et irréductible dans $F_p[x]$.
L'appel `GF( p, m, a)` permet à l'utilisateur de proposer lui-même un polynôme.
L'exemple suivant montre une construction de F_9 (noté ci-dessous GF9) à partir du polynôme irréductible de degré 2 dans $\boldsymbol{Z}/3\boldsymbol{Z}[x]$: $u^2 + 1$.

Exemple :

```
> readlib(GF) :
> GF9 := GF( 3, 2, u^2 + 1 ) :
```

Il est alors possible de calculer directement dans F_9.
`GF9[0]` (resp. `GF9[1]`) désigne l'élément neutre pour « + » (resp. « × ») dans F_9.
Il est possible de passer d'un entier à l'élément associé dans F_9 (*i.e.* le polynôme correspondant) et réciproquement. L'appel de fonction `s := GF9[input](i)` où i est un entier retourne l'élément s de F_9 indicé par i (qui a un sens pour $0 \leq i < 9$). L'opération `GF9[output](s)` permet d'effectuer l'opération réciproque.
De la même façon, les appels `t := GF9[ConvertIn](b)` (de réciproque `GF9[ConvertOut]`) où b est un polynôme en la variable u permet de construire l'élément t de F_9 correspondant.

Exemple :

```
> GF9[ConvertOut]( GF9[input](0) ) ;
```

$$0$$

```
> GF9[ConvertOut]( GF9[input](1) ) ;
```

$$1$$

```
> GF9[ConvertOut]( GF9[input](2) ) ;
```

$$2$$

```
> GF9[ConvertOut]( GF9[input](3) ) ;
```

$$u$$

```
> GF9[ConvertOut]( GF9[input](4) ) ;
```

$$u+1$$

```
> GF9[ConvertOut]( GF9[input](5) ) ;
```

$$u+2$$

```
> GF9[ConvertOut]( GF9[input](6) ) ;
```

$$2\,u$$

```
> GF9[ConvertOut]( GF9[input](7) ) ;
```

$$2\,u+1$$

```
> GF9[ConvertOut]( GF9[input](8) ) ;
```

$$2\,u+2$$

```
> GF9[output]( GF9[ConvertIn]( 2*u ) ) ;
```

$$6$$

```
> GF9[input](100) ;
```

$$10000000200000001$$

```
> GF9[ConvertOut](") ;
```

$$u^4+2\,u^2+1$$

Grâce aux fonctions `GF9[‘+‘]`, `GF9[‘-‘]`, `GF9[‘*‘]`, `GF9[‘/‘]`, `GF9[‘^‘]` et `GF9[inverse]`, il est alors possible d'effectuer les opérations arithmétiques correspondantes dans le corps F_9.

Exemple :

```
> GF9[input](3) ;
```

$$10000$$

```
GF9[‘-‘] ( 10000, GF9[1] ) ;
```

$$10002$$

```
> GF9[output](") ;
```

$$5$$

3.1.3 Propriétés élémentaires

On rappelle les propriétés élémentaires suivantes dans F_q qui seront utilisées par la suite.

Proposition 1 *Soit p un nombre premier, m un entier et soit F_q le corps d'ordre $q = p^m$. Les propriétés suivantes sont vérifiées :*

$$r^q = r \qquad \forall r \in F_q, \tag{3.1}$$

$$r^{1/p} = r^{q/p} = r^{p^{m-1}} \qquad \forall r \in F_q, \tag{3.2}$$

$$(r+s)^{p^j} = r^{p^j} + s^{p^j} \quad \forall r, s \in F_q, 0 \le j \le m. \tag{3.3}$$

À partir de ces propriétés, on obtient directement le résultat suivant :

Proposition 2 *Soit $P(x) = \sum_{i=0}^{n} a_i x^{ip}$ un polynôme de $F_q[x]$. Le polynôme $Q(x) = \sum_{i=0}^{n} a_i^{1/p} x^i$ vérifie :*

$$P(x) = Q(x)^p. \tag{3.4}$$

3.1.4 Théorème chinois des restes

On rappelle ici une application du théorème chinois qui sera utilisée par la suite. On travaille dans le cas entier et dans le cas polynomial.

Proposition 3 Cas entier. *Soient m_i, $1 \leq i \leq n$, n entiers positifs premiers deux à deux, et soit $M = \prod_{i=1}^{n} m_i$.*
Tout entier u, $0 \leq u < M$ peut être représenté de manière unique par le n-uplet de résidus $(u_1, \dots, u_n)$ où $u_i = u \bmod m_i$, pour $1 \leq i \leq n$.

De plus, l'opération de calcul des résidus quand l'entier est donné – et réciproquement celle de remontée de l'entier étant donnés les résidus – peut être effectuée avec un coût de $\mathcal{O}(\log n)$ multiplications d'entiers plus petits que M (cf. [2]).
Remarque
Le résultat précédent est bien sûr conservé si l'on prend l'entier u dans un intervalle entier quelconque d'amplitude M. En particulier, si $M = 2N+1$ est impair, on considère souvent l'intervalle centré : $\{-N, \dots, 0 \dots, N\}$.

Le résultat précédent se généralise au cas polynomial.

Proposition 4 Cas polynomial. *Soit K un corps et $K[x]$ l'anneau des polynômes à coefficients dans K.*
Soient $p_i(x)$, $1 \leq i \leq n$, n polynômes de $K[x]$ de degrés respectifs d_i et premiers deux à deux.
Alors tout polynôme $u(x)$ de $K[x]$ de degré strictement inférieur à $d = \sum_{i=1}^{n} d_i$ peut être représenté de manière unique par le n-uplet de résidus $(u_1(x), \dots, u_n(x))$ où $u_i(x) = u(x) \bmod p_i(x)$, pour $1 \leq i \leq n$.

De plus les opérations de passage de la représentation modulaire à la représentation dense peuvent être calculées avec un coût de $\mathcal{O}(\log n)$ multiplications de polynômes de degrés inférieurs à d (cf. [2]).

3.1.5 L'exemple de la recherche de chaînes de caractères

L'arithmétique modulaire peut être un moyen très efficace pour résoudre certains problèmes.
Outre l'utilisation du théorème chinois des restes, elle peut permettre la construction d'algorithmes probabilistes (cf. chapitre précédent au paragraphe 2.5.4) simples et performants.
Le but de cette partie est d'exhiber un exemple d'utilisation de l'arithmétique modulaire.

i) Homomorphisme sur les entiers

Un algorithme probabiliste est toujours construit à partir d'un oracle générant des entiers aléatoires (ou des bits). Pour introduire des entiers dans un problème $\mathcal{P}$ – de décision ici [1] – défini dans un ensemble E, une méthode consiste à transformer le problème initial pour le ramener à un problème équivalent (isomorphe) $\mathcal{P}_{\mathbb{Z}}$ dans les entiers – ou les rationnels –, en construisant un homomorphisme $\Phi : E \to \mathbb{Z}$ adapté. Si le problème $\mathcal{P}_{\mathbb{Z}}$ calcule en sortie un entier $s_{\mathbb{Z}}$, la solution s au problème $\mathcal{P}$ initial doit alors s'exprimer comme un test de nullité de cet entier, par exemple $s = (s_{\mathbb{Z}} =?0)$.

1. Cette technique est généralisable, comme nous le verrons dans l'exemple du filtrage de chaînes, à des problèmes qui ne se décrivent pas initialement comme un problème de décision.

Pour obtenir un algorithme pour résoudre $\mathcal{P}$, il faut construire un algorithme $\mathcal{A}_{\mathbb{Z}}$ pour résoudre le problème $\mathcal{P}_{\mathbb{Z}}$. Pour simplifier, nous supposerons dans ce qui suit que $\mathcal{A}_{\mathbb{Z}}$ ne contient pas de branchements – ou au plus un nombre fini indépendant de la taille des entrées. Généralement, les entiers manipulés par l'algorithme $\mathcal{A}_{\mathbb{Z}}$ peuvent être grands. Par suite, même si l'algorithme $\mathcal{A}_{\mathbb{Z}}$ a un coût faible en nombre d'opérations arithmétiques sur $\boldsymbol{Z}$, il peut s'avérer très inefficace en arithmétique booléenne, si les entiers sont plus grands que $\mathcal{O}(1)^n$. Pour limiter le coût de l'arithmétique entière, on peut alors effectuer les calculs *modulairement*, *i.e.* calculer $\mathcal{A}_{\mathbb{Z}}$ dans $\boldsymbol{Z}/p\boldsymbol{Z}$, où p est un entier choisi aléatoirement (plus petit que $n^{\mathcal{O}(1)}$). p peut être choisi premier ou non, selon les besoins (notamment si l'algorithme nécessite ou non des divisions). On construit ainsi un algorithme de Monte-Carlo :

i. tirer au hasard un entier p dans $\{1, \ldots, M\}$,

ii. effectuer les calculs de l'algorithme $\mathcal{A}_{\mathbb{Z}}$ dans $\boldsymbol{Z}/p\boldsymbol{Z}$: on obtient en sortie $s_{\boldsymbol{Z}/p\boldsymbol{Z}} = s_{\boldsymbol{Z}} \bmod p$,

iii. si $s_{\boldsymbol{Z}/p\boldsymbol{Z}} \neq 0$ alors $s_{\boldsymbol{Z}} \neq 0$: on peut alors calculer sans échec la solution du problème $\mathcal{P}$. Sinon, on ne peut pas décider si $s_{\boldsymbol{Z}}$ est nul ou pas, puisque, si cet entier est un multiple non nul de p, l'algorithme modulaire renvoie $s_{\boldsymbol{Z}/p\boldsymbol{Z}} = 0$ alors que $s_{\boldsymbol{Z}} \neq 0$. On a donc ici une possibilité d'échec.

Les définitions concernant les algorithmes de Monte-Carlo et de Las Vegas ont été données au paragraphe 2.5.4. Nous montrerons, sur l'exemple des chaînes de caractères, que la probabilité d'échec de cet algorithme de Monte-Carlo peut être rendue petite lorsque la taille des entiers manipulés par l'algorithme déterministe $\mathcal{A}_{\mathbb{Z}}$ est bornée par $n^{\mathcal{O}(1)}$.

Transformation en algorithme de Las Vegas. S'il existe un algorithme pour vérifier la justesse de la solution, il peut être transformé en algorithme de Las Vegas. Par ailleurs, il est toujours possible de le transformer en un algorithme déterministe, par utilisation du théorème chinois des restes : il suffit de lancer plusieurs exécutions de l'algorithme avec suffisamment de nombres – relativement premiers ici –, pour pouvoir remonter par interpolation la solution du problème sur les entiers : mais si les entiers manipulés sont grands, on obtient un travail exponentiel par rapport à celui de l'algorithme probabiliste.

Cette technique de calcul par homomorphisme est illustrée ci-après sur le problème de la recherche de chaîne de caractères dans un fichier.

ii) Filtrage de chaînes de caractères

Soit $E = (\{0,1\}^{\star}, \wedge, \epsilon)$ l'ensemble des chaînes de caractères (codées en binaire), $\wedge$ désignant l'opération de concaténation de chaînes et ϵ la chaîne vide (E est un *monoïde*). Étant données en entrée deux chaînes $Y = (y_0, \ldots, y_n)$ et $X = (x_0, \ldots, x_m)$ (on supposera $m \leq n$), le problème consiste à trouver toutes les occurrences de la chaîne X dans Y, autrement dit l'ensemble K :

$$K = \{k \leq n / y_k \ldots y_{k+m} = X\}$$

Une étude complète de ce problème est effectuée dans [1]. Le premier algorithme optimal de complexité $\mathcal{O}(n+m)$ est dû à Boyer et Moore (cf. [4]), mais il est peu parallèle. Il est facile de construire un algorithme parallèle en introduisant de la redondance : il suffit de comparer, en parallèle pour tout k, la chaîne $Y_k = (y_k \ldots y_{k+m})$ à la chaîne X – avec un coût séquentiel $\mathcal{O}(nm)$ – puis de rassembler les résultats obtenus.

Le problème est que cet algorithme est inefficace. Le but de ce paragraphe est de montrer comment l'utilisation de calcul modulaire permet de construire un algorithme probabiliste optimal de complexité $\mathcal{O}(n)$ pour localiser toutes les occurrences de X dans Y. Cet algorithme est dû à Karp et Rabin (cf. [10],[1]). Des extensions pour des problèmes de filtrage multi-dimensionnel sont présentées dans [6].

iii) Plongement du problème dans les entiers

L'espace E de départ étant muni d'une loi $\wedge$ associative, non commutative et qui possède un élément neutre ϵ, il est nécessaire de construire un espace F – construit à partir des entiers – qui possède au moins une telle opération (et éventuellement une structure plus riche).

L'espace des matrices carrées de dimension 2, muni du produit de matrices, a une telle structure. Pour pouvoir rendre l'opération de concaténation inversible, on plonge le problème dans l'espace $\mathcal{E}$

$$\mathcal{E} = \{\mathcal{M} \in \mathcal{M}_{2,2} / \det(\mathcal{M}) = 1\}$$

des matrices de déterminant 1, ce qui permettra de les inverser. Dans la suite, on note I la matrice identité de $\mathcal{E}$. L'homomorphisme de structure Φ doit vérifier :

$$\begin{cases} \Phi(\epsilon) = I, \\ \Phi(a \wedge b) = \Phi(a)\Phi(b) \quad \forall (a,b) \in E^2. \end{cases} \tag{3.5}$$

Φ est alors complètement caractérisé par la donnée de $\Phi(<<\ 0\ >>)$ et $\Phi(<<\ 1\ >>)$. Pour que Φ soit injective – de façon à ce que deux chaînes distinctes soient associées à deux matrices distinctes – on pose :

$$\Phi('0') = \begin{bmatrix} 1 & 0 \\ 1 & 1 \end{bmatrix}, \qquad\qquad \Phi('1') = \begin{bmatrix} 1 & 1 \\ 0 & 1 \end{bmatrix}. \tag{3.6}$$

Il est facile de montrer que Φ ainsi construite est un homomorphisme injectif de structure, donc que E est isomorphe à $\Phi(E) \subset \mathcal{E}$.

Remarque. Concaténer un « 0 » à gauche d'une séquence s peut se calculer de la façon suivante :

$$\Phi('0' \wedge s) = \Phi('0')\Phi(s) = \begin{bmatrix} 1 & 0 \\ 1 & 1 \end{bmatrix} \begin{bmatrix} a & b \\ c & d \end{bmatrix} = \begin{bmatrix} a & b \\ a+c & b+d \end{bmatrix}$$

ce qui équivaut à remplacer la $2^{\text{ième}}$ ligne de $\Phi(s)$ par la somme de ses 2 lignes.
De même,

- concaténer un « 1 » à gauche d'une séquence s équivaut à remplacer la ligne 1 par la somme des 2 lignes,
- concaténer un « 0 » à droite d'une séquence s équivaut à remplacer la colonne 1 par la somme des 2 colonnes,
- concaténer un « 1 » à droite d'une séquence s équivaut à remplacer la colonne 2 par la somme des 2 colonnes.

Soit $Y_k = (y_k \dots y_{k+m})$, $M_X = \Phi(X)$ et $M_k = \Phi(Y_k)$. Les matrices M_k peuvent être calculées optimalement (en comptant le nombre d'opérations sur les entiers, et non sur les booléens).
En effet, posons $A_k = \Phi(y_0 \dots y_k)$, $0 \le k \le n$. Les matrices A_k sont ce que l'on appelle en jargon

spécialisé les préfixes – pour l'opération produit de matrices dans $\mathcal{E}$ – des matrices $\Phi(y_k)$. Un calcul de préfixes est un problème pour lequel on connaît des solutions très efficaces, aussi bien en séquentiel qu'en parallèle. Elles peuvent donc être calculées avec un coût $\mathcal{O}(n)$: la remarque précédente montre en effet que l'opération de base – le produit de deux matrices $\Phi(a)$ et $\Phi(b)$ – se ramène à 2 additions d'entiers.
Posons $A_{-1} = I$. On a alors :

$$M_k = A_{k-1}^{-1} A_{k+m} \qquad \forall\, 0 \leq k \leq n-m$$

et les matrices M_k peuvent être obtenues à partir des matrices A_k avec un coût optimal de $\mathcal{O}(n)$.

L'algorithme s'écrit finalement :

i. Calcul de $\Phi(x_1, \ldots, x_m) = M_X$.

ii. Calcul de $A_k = \Phi('y_0 \ldots y_k')$, $0 \leq k \leq n$ (calcul de préfixes).

iii. Calcul de A_k^{-1}, inverse de A_k (de déterminant 1).

iv. Calcul de $M_k = A_{k-1}^{-1} A_{k+m}$, $0 \leq k \leq n-m$.

v. Pour $k = 0, \ldots, n-m$, comparaison de M_k et M_X, pour déterminer toutes les occurrences de X dans Y.

On peut montrer, par récurrence, que si s est une chaîne de i caractères, les coefficients de $\Phi(s)$ sont bornés par 2^i. Par suite, les entiers manipulés par cet algorithme sont plus petits que 2^n, et donc très grands. La technique introduite dans cette section consiste alors à construire un algorithme probabiliste de Monte-Carlo en effectuant non pas les opérations arithmétiques exactes (trop coûteuses), mais modulo un nombre p petit, tiré au hasard, plus petit que $n^{\mathcal{O}(1)}$ (*i.e.* ayant au plus $\mathcal{O}(\log n)$ bits).

Cela revient à remplacer Φ par l'homomorphisme Φ_p :

$$\Phi_p(x) = \Phi(x) \mod p \qquad \forall x \in E.$$

Φ_p n'étant alors plus injectif, il est possible d'avoir deux chaînes a et b distinctes qui vérifient $\Phi_p(a) = \Phi_p(b)$. Dans la suite, nous allons montrer que ceci ne peut se produire qu'avec une probabilité inférieure à une constante.

L'algorithme précédent, avec Φ_p, délivre donc en sortie un sur-ensemble de K : certaines sous-chaînes de Y obtenues en sortie sont différentes de X, mais toutes les occurrences de X apparaissent en sortie. On a donc ainsi construit un algorithme de Monte-Carlo de coût $\mathcal{O}(n)$ optimal. Il reste à étudier sa probabilité d'erreur.

iv) Analyse de la probabilité d'erreur

Pour majorer la probabilité d'erreur, nous choisissons en fait dans l'algorithme précédent un nombre p premier. Cela ne change rien à l'algorithme mais permet d'établir plus facilement le calcul qui suit.

Soit p un nombre premier et Φ_p l'homomorphisme de structure induit.

Soit $\tilde{K} = \{k / Y_k \neq X\}$. Le cardinal de $\tilde{K}$ est plus petit que $(n-m)$. L'algorithme proposé filtre incorrectement (*échec*) toutes les chaînes Y_k $(k \in \tilde{K})$ qui vérifient :

$$\Phi_p(Y_k) - \Phi_p(X) = 0.$$

Posons $N_k = \Phi(Y_k) - \Phi(X)$: il y a donc échec s'il existe $k \in \tilde{K}$ tel que :

$$N_k \bmod p = 0,$$

ce qui équivaut à dire que p divise $N_{X,Y} = \prod_{k \in \tilde{K}} N_k$. La probabilité que l'algorithme de Monte-Carlo produise un échec est donc bornée par :

$$\text{Prob(échec)} = \text{Prob}(p \text{ divise } N_{X,Y}).$$

Nous avons vu que, si s est une chaîne de longueur i, les coefficients de $\Phi(s)$ sont bornés par 2^i. Par suite, $N_{X,Y} \leq 2^{nm}$.
$N_{X,Y}$ admet donc au plus nm facteurs premiers.

Soit p un nombre premier *tiré uniformément* dans l'ensemble des nombres premiers plus petits que α, où α est un entier donné (plus petit que $n^{\mathcal{O}(1)}$). Le nombre $\pi(\alpha)$ d'entiers premiers plus petits que α vérifie :

$$\frac{\alpha}{\log_e \alpha} \leq \pi(\alpha) \leq 1.3 \frac{\alpha}{\log_e \alpha}. \tag{3.7}$$

La probabilité que l'algorithme de Monte-Carlo produise un échec est donc bornée par :

$$\text{Prob(échec)} \leq \frac{nm}{\pi(\alpha)}.$$

Posons $\alpha = n^2 m$. La probabilité d'échec, lorsque p est un nombre premier tiré uniformément dans $\{2, \ldots, n^2 m\}$ est donc bornée par

$$\frac{3 \log n}{n}$$

qui tend asymptotiquement vers 0.

Exemple :

Pour $n = 10^6$ et $m = 10^2$: si l'on tire p au hasard parmi les nombres premiers plus petits que 10^{14} (codables exactement dans un flottant double précision), la probabilité d'échec de l'algorithme de Monte-Carlo proposé est donc inférieure à 10^{-4}.

v) De Monte-Carlo à Las Vegas

On suppose ici que la chaîne X n'est pas périodique : il y a donc au plus $\frac{n}{m}$ occurrences de X dans Y.

Pour rendre l'algorithme précédent déterministe, il suffit de vérifier que toutes les chaînes trouvées par l'algorithme de Monte-Carlo précédent sont bien égales à la chaîne X. La comparaison de l'égalité de deux chaînes de longueur m peut se faire optimalement en $\mathcal{O}(m)$.

Mais il se peut que l'algorithme de Monte-Carlo délivre $\mathcal{O}(n)$ occurrences, avec une très faible probabilité comme nous l'avons montré précédemment. Dans ce cas, la vérification déterministe prend une complexité $\mathcal{O}(nm)$ et rend l'algorithme inefficace.

Il est donc préférable de construire, à partir de cette vérification possible, un algorithme de Las Vegas :

i. Appliquer l'algorithme de Monte-Carlo précédent.

ii. Si l'algorithme délivre en sortie plus de $\frac{n}{m}$ occurrences alors délivrer en sortie un constat d'échec.

iii. Sinon, vérifier toutes les occurrences trouvées, et délivrer en sortie toutes les occurrences Y_k égales à X.

La complexité de cet algorithme de Las Vegas est alors $\mathcal{O}(n)$; il est donc optimal. De plus, il ne délivre un constat d'échec qu'avec une probabilité inférieure à celle d'erreur de l'algorithme de Monte-Carlo, donc très faible.

3.2 Factorisation - Algorithme de Berlekamp

Maintenant que nous avons présenté le calcul dans les corps finis avec Maple, le problème est de factoriser un polynôme, par l'algorithme de Berlekamp. Tout d'abord, le polynôme va être transformé en un polynôme n'admettant que des racines simples – en passant par sa factorisation sans carré – puis l'algorithme de factorisation sera expliqué.

Soit P un polynôme unitaire à coefficients rationnels de degré n. En multipliant tous les coefficients de P par le ppcm des dénominateurs puis en divisant les coefficients par leur pgcd, on peut se ramener à un polynôme $A = \sum_{i=0}^{n} a_i x^i$ à coefficients a_i entiers premiers dans leur ensemble[2]. Factoriser le polynôme P sur les rationnels se ramène alors à factoriser le polynôme A dans $\boldsymbol{Z}[x]$.

On définit la *hauteur* h d'un polynôme comme le plus grand module de ses coefficients :

$$h(A) = \max_{i=0}^{n} |a_i|. \tag{3.8}$$

Si le polynôme A n'est pas irréductible, il admet alors au moins un facteur à coefficients entiers de degré inférieur ou égal à $\frac{n}{2}$. À partir d'une majoration des racines du polynôme A, il est possible de borner par une constante B_d la hauteur de tout diviseur de A de degré plus petit que d. Différentes bornes sont connues (cf. [13]), notamment :

$$B_d \leq 2^d h(A) \tag{3.9}$$

qui est relativement fine.

Il est alors possible de factoriser le polynôme A dans les entiers, en énumérant tous les polynômes de degré d, $1 \leq d \leq n/2$, à coefficients entiers compris entre $-B_d$ et B_d. Par division polynomiale, il est possible de sélectionner les facteurs non triviaux de A. Si aucun diviseur de degré inférieur à $n/2$ n'est trouvé, le polynôme A est irréductible. Cet algorithme naïf est de coût exponentiel en n^2 et donc très inefficace.

Posons $B = \max_{d=0}^{n/2} B_d$ et soit p un nombre premier plus grand que $2B$. Tout facteur de A dans $\boldsymbol{Z}[x]$ est facteur de A dans $\mathrm{F}_p[x]$. À partir de la factorisation de A dans $\mathrm{F}_p[x]$ (*i.e.* modulo p), il est alors possible de construire la factorisation de A dans $\boldsymbol{Z}[x]$. C'est le principe de l'algorithme de Berlekamp que nous détaillons ici. Mais tout d'abord, la factorisation sans carré de A va être calculée, afin que l'on factorise, au paragraphe 3.2.2, un polynôme n'admettant que des racines simples.

3.2.1 Factorisation sans carré

i) Définition

Soit P un polynôme de degré n dans $K[X]$. P est dit *sans carré* si toutes ses racines sont simples.

2. *i.e.* $A = a_n P$ et $\mathrm{pgcd}(a_0, \ldots, a_n) = 1$.

La factorisation *sans carré* d'un polynôme Q est :

$$Q(x) = \prod_{i=1}^{k} a_i(x)^i \tag{3.10}$$

où chacun des a_i est un polynôme sans carré (éventuellement de degré 0), et les a_i sont premiers deux à deux, *i.e.* $\mathrm{pgcd}(a_i(x), a_j(x)) = 1$ pour tout $i \neq j$.

Exemple :

```
> Q := expand ( (2*x^3 + 1) * (x-2)^3 ) ;
```

$$2\,x^6 - 12\,x^5 + 24\,x^4 - 15\,x^3 - 6\,x^2 + 12\,x - 8$$

```
> sqrfree(Q) ;
```

$$[1, [[2\,x^3 + 1, 1], [x - 2, 3]]]$$

Calculer la factorisation sans carré d'un polynôme est donc une première étape vers la factorisation complète du polynôme.

Proposition 5 *Q est sans carré si et seulement si il est premier avec sa dérivée Q'.*

DÉMONSTRATION
Si Q est sans carré, soit $Q = P_1 \dots P_t$ sa factorisation en facteurs irréductibles (de degrés non nuls). Si Q et Q' ne sont pas premiers entre eux, il existe au moins un facteur irréductible de Q qui divise Q'. Supposons que ce facteur est P_1. Alors $Q' = P_1(P_2 \dots P_t)' + P_1'(P_2 \dots P_t)$ est un multiple de P_1. Comme P_1 est premier avec $(P_2 \dots P_t)$, P_1 divise P_1'. Donc $P_1' = 0$.
Deux cas sont alors à distinguer :

- *K est de caractéristique nulle.* $P_1' = 0$ entraîne que P_1 est de degré 0, ce qui est absurde.
- *K est de caractéristique p.* $P_1' = 0$ entraîne que les seuls termes de coefficients non nuls de P_1 sont de degré multiple de p. On a alors :

$$P_1(x) = a_0 + a_p x^p + \dots + a_{kp} x^{kp} \tag{3.11}$$

  D'après la proposition (2), on en déduit qu'il existe un polynôme R tel que $P_1 = R^p$. Ce qui est absurde puisque l'on a supposé P_1 irréductible.

Réciproquement, si Q et Q' sont premiers entre eux, il existe deux polynômes U et V tels que (Bezout) :

$$U(x)Q(x) + V(x)Q'(x) = 1. \tag{3.12}$$

Q et Q' n'ont donc pas de racines communes. Par suite, Q n'a que des racines simples et est sans carré. $\square$

ii) Calcul de la factorisation sans carré

La proposition précédente permet de construire facilement une factorisation sans carré d'un polynôme, en n'effectuant que des calculs de pgcd. Nous distinguons deux cas, selon que la caractéristique est nulle ou non, pour présenter cet algorithme.

K est de caractéristique nulle

Considérons le polynôme suivant :

$$P_1 = \frac{P}{\mathrm{pgcd}(P, P'^2)}. \tag{3.13}$$

P_1 est un diviseur de P, premier avec P/P_1 et P'. De plus, toutes les racines du polynôme P/P_1 sont racines de P', donc de multiplicité au moins 2 dans P. On en déduit donc que P_1 est le facteur sans carré correspondant aux racines de multiplicité 1 dans P.
En réitérant le calcul avec en entrée le polynôme $\mathrm{pgcd}(P, P')$, on construit le polynôme P_2 dont les racines sont simples et de multiplicité 2 dans P et ainsi de suite.

K est de caractéristique p

Si K est F_{p^m}, de caractéristique p, l'opération de dérivation peut fournir un polynôme nul. Soit $G(x) = \mathrm{pgcd}(P(x), P'(x))$. Deux cas sont alors à distinguer :

- $P'(x) \neq 0$: $\mathrm{pgcd}(P(x), P'(x))$ est alors un facteur non trivial de P. En procédant comme en caractéristique nulle, on construit le polynôme P_1 (3.13) facteur sans carré correspondant aux racines de multiplicité 1 dans P.

- $P'(x) = 0$: d'après la proposition (2), on en déduit qu'il existe un polynôme R tel que $P = R^p$. On construit alors la factorisation sans carré de P à partir de celle de R.

Comme précédemment, on peut alors construire de manière récursive les autres facteurs de multiplicités différentes de P.

iii) Coût du calcul

Dans les deux cas, la complexité en nombre d'opérations arithmétiques sur K est dominée par celle des calculs de pgcd. Lorsque les deux polynômes sont de degré n, le coût d'un tel calcul est majoré par $\mathcal{O}(n^2)$. Le coût de la décomposition sans carré est donc majoré par $\mathcal{O}(n^3)$.

3.2.2 Factorisation dans $\mathbf{F}_p[x]$

Bien que les résultats énoncés dans ce paragraphe puissent être directement étendus à tout corps fini, nous nous restreignons ici pour simplifier les notations au corps fini F_p, avec p nombre premier : les égalités sont donc valides modulo p. Les résultats obtenus sont directement généralisables au corps de Galois F_q avec $q = p^m$.

On s'intéresse donc ici à décomposer en facteurs irréductibles un polynôme P de $\mathrm{F}_p[x]$ unitaire et de degré n. Le paragraphe précédent montre que l'on peut supposer sans restriction que P est sans carré.

i) Caractérisation des facteurs irréductibles

Soit $\mathcal{V}$ l'anneau quotient $\mathrm{F}_p[x]/P(x)$. $\mathcal{V}$ est un espace vectoriel sur le corps F_p, de dimension le degré de P, *i.e.* n.

Soit $P(x) = P_1(x) \ldots P_r(x)$ la décomposition en facteurs irréductibles du polynôme P.

Soit $(s_1, \ldots s_r)$ un r-uplet de F_p^r. Les polynômes P_i étant premiers entre eux, le théorème chinois des restes permet d'écrire qu'il existe un unique polynôme U de $\mathrm{F}_p[x]$ de degré strictement inférieur à n – donc dans $\mathcal{V}$ – tel que :

$$\forall i, 1 \leq i \leq r \qquad U(x) \bmod P_i(x) = s_i \tag{3.14}$$

où $A(x) \bmod B(x)$ dénote le reste de la division polynomiale de A par B dans $\mathrm{F}_p[x]$.

De plus, soit U un polynôme vérifiant (3.14). Comme pour tout élément s de F_p on a $s^p = s$, on en déduit pour tout i, $1 \leq i \leq r$:

$$U^p(x) \bmod P_i(x) = s_i. \tag{3.15}$$

Comme il existe un unique polynôme de degré inférieur à n vérifiant (3.14), on a :

$$U^p(x) \bmod P(x) = U(x), \tag{3.16}$$

soit encore :

$$(U^p(x) - U(x)) \bmod P(x) = 0. \tag{3.17}$$

Réciproquement, posons :

$$\mathcal{W} = \{U(x) \in \mathcal{V} \ : U^p(x) \bmod P(x) = U(x)\}, \tag{3.18}$$

et soit $U(x)$ un polynôme de $\mathcal{W}$. Considérons maintenant le polynôme $\Pi(x) = x^p - x$. Π admet les p éléments de F_p comme racines distinctes. Π étant de degré p, il admet au plus p racines. On a donc l'identité :

$$\Pi(x) = x^p - x = \prod_{i=0}^{p-1}(x - i). \tag{3.19}$$

Appliquée au polynôme U vérifiant (3.17), cette identité permet d'écrire :

$$\prod_{i=0}^{p-1}(U(x) - i) \bmod P(x) = 0. \tag{3.20}$$

Autrement dit, le polynôme P divise le produit des polynômes $(U(x) - i)$, $0 \leq i \leq p-1$. Tout facteur irréductible P_i de P doit donc diviser l'un de ces polynômes.
Il existe donc $0 \leq j \leq p-1$ tel que $P_i(x)$ divise $(U(x) - j)$, soit encore :

$$U(x) \bmod P_i(x) = j. \tag{3.21}$$

Tout élément de $\mathcal{W}$ vérifie donc (3.14).

On en déduit donc le résultat suivant, dont la démonstration découle directement de ce qui précède.

Proposition 6 *$\mathcal{W}$ est isomorphe à F_p^r.*
Le nombre de facteurs irréductibles de P est donc la dimension de $\mathcal{W}$ en tant qu'espace vectoriel sur le corps F_p.
De plus, pour tout $U \in \mathcal{W}$ de degré non nul, on a :

$$P(x) = \prod_{i=0}^{p-1} \mathrm{pgcd}(U(x) - i, P(x)). \tag{3.22}$$

L'égalité (3.22) provient du fait que, d'une part, lorsque U n'est pas de degré nul, les polynômes $U(x)-i$, $0 \leq i \leq p-1$, sont premiers deux à deux et, d'autre part, leur produit est un multiple de $P(x)$.

Le calcul d'une base de $\mathcal{W}$ (en faisant appel à la proposition 1), qui est un problème d'algèbre linéaire, permet alors soit de prouver l'irréductibilité du polynôme P (lorsque cette base ne contient qu'un élément), soit de trouver un facteur non trivial de P (grâce à l'égalité (3.22)).

ii) Calcul d'une base de $\mathcal{W}$

Pour tout polynôme $U = \sum_{i=0}^{n-1} u_i x^i$ de $\mathcal{V}$, on a :

$$U(x)^p = \left(\sum_{i=0}^{n-1} u_i x^i\right)^p = \sum_{i=0}^{n-1} u_i^p x^{ip} = \sum_{i=0}^{n-1} u_i x^{ip} = U(x^p). \tag{3.23}$$

$\mathcal{W}$ peut donc être décrit de la façon suivante :

$$\mathcal{W} = \{U(x) \in \mathcal{V} \ : (U(x^p) - U(x)) \bmod P(x) = 0\}. \tag{3.24}$$

Le calcul d'une base de $\mathcal{W}$ se ramène donc au calcul de r vecteurs de la forme $(u_0, \ldots, u_{n-1})$ de F_p^n solutions d'un système linéaire.

Pour écrire ce système, calculons pour tout $0 \leq i \leq n-1$, $x^{ip} \bmod P(x)$. Soit

$$x^{ip} \bmod P(x) = \sum_{j=0}^{n-1} q_{i,j} x^j \tag{3.25}$$

et soit Q la matrice carrée de dimension n de coefficient $q_{i,j}$ en ligne $i+1$ et colonne $j+1$.

$U(x) = \sum_{i=0}^{n-1} u_i x^i$ est alors élément de $\mathcal{W}$ si et seulement si :

$$[u_0, \ldots, u_{n-1}]Q = [u_0, \ldots, u_{n-1}]. \tag{3.26}$$

Soit $M = (Q-I)^t$. Le nombre de facteurs irréductibles de P est alors $r = n - \mathrm{rang}(M)$.
De plus, une base de r polynômes de $\mathcal{W}$ peut être calculée à partir d'une base de l'espace nul de M (*i.e.* une base du noyau de M). Un tel calcul se ramène à une triangularisation de la matrice M (algorithme d'élimination).

Cette première phase de l'algorithme de Berlekamp nous a donc permis d'obtenir d'une part le nombre r de facteurs irréductibles de P (à partir du calcul du rang de la matrice M) et d'autre part une base $(U_1(x), \ldots, U_r(x))$ de $\mathcal{W}$ (à partir de l'espace nul de M). Il reste maintenant à construire les facteurs irréductibles de P à partir de cette base de $\mathcal{W}$.

On peut remarquer que le polynôme constant $U(x) = 1$ est un polynôme trivial de $\mathcal{W}$. On peut donc se ramener au cas où $U_1(x) = 1$. Nous supposerons donc dans la suite $U_1(x) = 1$.

iii) Calcul des facteurs irréductibles

Comme $U_2(x)$ est de degré non nul et strictement inférieur au degré de P, l'égalité (3.22) appliquée à U_2 fournit une factorisation non triviale du polynôme P :

$$P(x) = \prod_{i=0}^{p-1} \mathrm{pgcd}(U_2(x) - i, P(x)). \tag{3.27}$$

Deux cas sont alors à considérer. Si r facteurs sont ainsi obtenus, la décomposition de P en facteurs irréductibles est achevée.

Sinon, soit P_1 et P_2 deux facteurs irréductibles quelconques distincts de P. De par l'isomorphisme entre $\mathcal{W}$ et F_p^r, il existe nécessairement un élément i de F_p, $0 \leq i \leq n-1$, et un entier k, $2 \leq k \leq r$, tel que $(U_k(x) - i)$ est divisible par P_1 mais pas par P_2.

Soit donc $f(x)$ un facteur non trivial de P construit à partir de U_2. De la même façon que précédemment (3.27), on a l'égalité :

$$f(x) = \prod_{i=0}^{p-1} \mathrm{pgcd}(U_k(x) - i, f(x)). \tag{3.28}$$

Si f n'est pas irréductible, il existe donc forcément un indice k, $2 \leq k \leq r$, qui fournit une factorisation non triviale.

En utilisant cette égalité pour tous les facteurs f trouvés successivement pour $k = 2, 3, \ldots, r$, on obtient les r facteurs irréductibles de P. Il suffit alors d'arrêter la recherche dès que r facteurs ont été trouvés.

Remarque

Généralisation au corps de Galois $\mathbf{F}_q$ $(q = p^m)$.

La démonstration précédente s'étend directement au corps de Galois d'ordre $q = p^m$. Soient e_i, $0 \leq i \leq q-1$, les q éléments supposés indicés de F_q. Il suffit de remplacer p par q et de considérer les polynômes $(U_k(x) - e_i)$, $0 \leq i < q$, au lieu des polynômes $(U_k(x) - i)$, $0 \leq i < p$.

v) Analyse de complexité

Nous supposons ici les opérations arithmétiques dans F_p de coût constant. Dans tous les cas elles sont bornées par un facteur polylogarithmique[3] en p.

En utilisant une technique d'élévation récursive au carré, la matrice Q peut être construite avec une complexité $\mathcal{O}(n^2 \log p)$.

Le calcul d'une base de l'espace nul de la matrice M peut ensuite être effectué en $\mathcal{O}(n^3)$ par un algorithme d'élimination de type Gauss.

Le calcul du pgcd de deux polynômes de degré n peut être facilement réalisé par l'algorithme d'Euclide en $\mathcal{O}(n^2)$ opérations[4]. Dans le pire cas, n facteurs irréductibles peuvent être à calculer. On a donc finalement un calcul de coût borné par $\mathcal{O}(n^3 p)$ pour la dernière phase.

Finalement le coût du calcul est borné par $\mathcal{O}(n^3 p)$, et c'est la dernière étape, exponentielle dans la taille de la représentation binaire de p, qui est la plus coûteuse.
Le calcul est cependant de coût polynomial en le degré du polynôme à factoriser.

3.2.3 Cas des rationnels

Soit P un polynôme à coefficients rationnels de degré n que l'on désire décomposer en facteurs irréductibles. Nous avons vu qu'il est possible de se ramener au cas où le polynôme P à factoriser est sans carré et à coefficients entiers premiers deux à deux. D'après (3.9), les coefficients d'un facteur quelconque de P sont alors bornés par :

$$B = 2^n h(P). \tag{3.29}$$

3. *i.e.* $\log^{\mathcal{O}(1)} p$. Plus précisément, on a une complexité $\Omega(\log p)$ pour l'addition, $\mathcal{O}(\log p \log\log p \log\log\log p)$ pour la multiplication et $\mathcal{O}(\log p \log\log^2 p \log\log\log p)$ pour l'inversion (cf. [2]).

4. Ce calcul peut être en fait effectué en $\mathcal{O}(n \log^2 n)$ opérations (cf. [14]).

Choisissons un entier p premier et supérieur à $2B+1$. Si P admet une factorisation dans $\boldsymbol{Z}[x]$, sous la forme $P = P_1 \dots P_r$, alors il admet une factorisation dans $\boldsymbol{Z}/p\boldsymbol{Z}[x]$:

$$P \bmod p \ = (P_1 \bmod p) \dots (P_r \bmod p). \tag{3.30}$$

Mais les facteurs ($P_i \bmod p$) ne sont pas nécessairement irréductibles dans $\boldsymbol{Z}/p\boldsymbol{Z}[x]$. La réciproque est donc fausse, comme le prouve l'exemple déjà vu du polynôme $x^2 - x + 1$, irréductible dans $\mathbb{Q}[x]$ et factorisable en $(x+4)(x+2)$ dans $\mathrm{F}_7[x]$.
Si P est sans carré modulo p, sa factorisation modulo p permet de reconstruire, en regroupant si nécessaire certains des facteurs obtenus modulo p, la décomposition en facteurs irréductibles modulo p.

Nous avons vu que le coût du calcul de la factorisation dans $\mathrm{F}_p[x]$ croît linéairement avec p. La borne B sur les coefficients étant exponentielle en le degré du polynôme, le choix de $p \geq 2B+1$ entraîne un coût de factorisation modulo p exponentiel en n.
Pour éviter ce problème, la technique de remontée de Hensel permet, à partir d'une factorisation modulo un entier p premier quelconque, de construire une factorisation modulo $q = p^m$ en temps polynomial en m.

Les remarques précédentes permettent de construire la factorisation de P selon le schéma général suivant, que nous allons détailler dans ce paragraphe :

- choix d'un entier premier p petit convenable et calcul de la factorisation en facteurs irréductibles de P modulo p ;
- remontée à partir de la factorisation précédente de la factorisation la plus fine possible de P modulo p^m, en choisissant m tel que $p^m > 2B$;
- recherche dans la factorisation modulo p^m – éventuellement fausse dans $\boldsymbol{Z}[x]$ – d'un facteur non trivial de P dans $\boldsymbol{Z}[x]$.

i) Factorisation modulo p

Nous avons supposé que P est sans carré dans $\boldsymbol{Z}[x]$. Pour pouvoir appliquer l'algorithme de Berlekamp modulo p, nous devons choisir p tel que ($P \bmod p$) soit sans carré dans $\mathrm{F}_p[x]$.

D'après la proposition (5), ($P \bmod p$) est sans carré si et seulement si il est premier avec sa dérivée dans $\mathrm{F}_p[x]$, donc si le résultant [5] $\mathrm{res}(P, P')$ est non nul modulo p.

Pour choisir un p convenable, il suffit donc de calculer le résultant $\mathrm{res}(P, P')$ dans $\boldsymbol{Z}$. D'après la borne d'Hadamard, ce résultant est borné par $(nh(P))^{2n} \leq 2^{2n \log n \log h(P)}$. Il admet donc au plus $\mathcal{O}(n \log n \log h(P))$ diviseurs premiers.
En tenant compte de l'encadrement (3.7), il est donc possible de trouver en temps polynomial un nombre premier p petit – ayant au plus $\mathcal{O}(\log n + \log\log h(P))$ bits – tel que P soit sans carré modulo p.

L'algorithme de Berlekamp dans $F_p[x]$ permet alors de décomposer P en facteurs irréductibles modulo p. Soit $P_1, \dots, P_r$ les facteurs ainsi trouvés.

5. Si deux polynômes P et Q de degrés respectifs n et m sont premiers entre eux, d'après l'identité de Bezout, il existe un unique polynôme U (resp. V) de degré strictement inférieur à m (resp. n) tel que :

$$UP + VQ \equiv 1 \mod p.$$

Cette équation peut être écrite comme un système linéaire de $n+m$ équations à $n+m$ inconnues. La matrice de ce système s'appelle la matrice de Sylvester et son déterminant le résultant de P et Q.
P est premier avec Q si et seulement si ce système est inversible, donc si le résultant est non nul (voir aussi le paragraphe 2.5 du chapitre sur l'algèbre linéaire).

ii) Remontée de Hensel modulo p^m

Les facteurs P_i sont alors des facteurs de P dans $\mathrm{F}_p[x]$ et des facteurs irréductibles de P dans $\boldsymbol{Z}[x]$.

Si $r = 1$, on en déduit donc que P est irréductible.
Sinon, en groupant des facteurs irréductibles ensemble, il est possible de construire une factorisation non triviale $P = U_1V_1$ dans $\mathrm{F}_p(X)$.

L'algorithme de remontée de Hensel permet alors, à partir de cette factorisation, de construire une factorisation non triviale $P = U_kV_k$ de P qui est vérifiée modulo p^k, pour tout $k \geq 2$.

Pour simplifier, nous supposons dans un premier temps que P est un polynôme unitaire. Les facteurs P_i de P dans $\mathrm{F}_p[x]$ peuvent alors être choisis unitaires. La construction d'une factorisation $P = U_{k+1}V_{k+1}$ valide modulo p^{k+1} peut alors être effectuée par récurrence à partir de la factorisation $P = U_kV_k$ modulo p^k de la façon suivante.

Le polynôme $P - U_kV_k$ est nul modulo p^k et donc est multiple de p^k. Soit R_k le polynôme de $\boldsymbol{Z}[x]$ tel que :

$$P - U_kV_k = p^kR_k. \tag{3.31}$$

P étant sans carré dans $\mathrm{F}_p[x]$, U_k et V_k sont premiers entre eux dans $\mathrm{F}_p[x]$. D'après l'identité de Bezout, il existe donc deux polynômes A et B, avec $\deg(A) < \deg(V_k)$ et $\deg(B) < \deg(U_k)$, vérifiant :

$$A(x)U_k(x) + B(x)V_k(x) \equiv 1 \mod p. \tag{3.32}$$

Soit $T_k = R_kA \text{ div } V_k$. Construisons les deux polynômes U_{k+1} et V_{k+1} suivants :

$$\begin{array}{rcl} U_{k+1} & = & U_k + p^k(R_kB + T_kU_k), \\ V_{k+1} & = & V_k + p^k(R_kA - T_kV_k). \end{array} \tag{3.33}$$

On a alors :

$$P \equiv U_{k+1}V_{k+1} \mod p^{k+1} \tag{3.34}$$

qui construit bien une factorisation non triviale de P modulo p^{k+1}.

On peut remarquer de plus que $\deg(U_{k+1}) + \deg(V_{k+1}) = \deg(P)$ et :

$$A(x)U_{k+1}(x) + B(x)V_{k+1}(x) \equiv 1 \mod p. \tag{3.35}$$

Par suite, le calcul des coefficients de Bezout A et B n'est à faire qu'une seule fois au début de l'algorithme.

iii) De Hensel à la factorisation dans $\boldsymbol{Z}[x]$

Soit m le plus petit entier tel que $p^m > 2B$. L'itération précédente – remontée de Hensel – permet donc de construire deux polynômes U_m et V_m dont les coefficients entiers sont compris entre $-B$ et B et qui sont facteurs non triviaux de P modulo p^m.

Deux cas sont alors à considérer :

- La factorisation en U_m et V_m est valide dans $\boldsymbol{Z}[x]$. Deux facteurs non triviaux de P ont été isolés. On ré-applique la remontée de Hensel à partir des facteurs irréductibles dans $\mathrm{F}_p[x]$ de U_1 (resp. V_1) pour essayer de factoriser plus finement U_m (resp. V_m).

– $P \neq U_m V_m$ dans $\boldsymbol{Z}[x]$. La factorisation remontée n'est donc pas valide. S'il reste une autre paire (U_1', V_1') de facteurs non triviaux de P dans $\mathrm{F}_p[x]$ qui n'a pas encore été testée, on effectue une nouvelle remontée pour tenter de remonter un nouveau facteur.
  Sinon, il n'existe pas de factorisation dans $\mathrm{F}_p[x]$ compatible avec une factorisation dans $\boldsymbol{Z}[x]$; P est donc irréductible.

Remarque
Cas où le polynôme P n'est pas unitaire. Nous avons pour simplifier considéré ici le cas où P est unitaire. Dans le cas général, nous avons $P = \sum_{i=0}^{n} a_i x^i$, les coefficients a_i étant entiers et premiers dans leur ensemble.
Dans $\mathrm{F}_p[x]$, considérons alors le polynôme $Q = a_n^{-1} P$. Q a les mêmes facteurs irréductibles que P. Soient U et V deux facteurs unitaires obtenus modulo p^m après remontée de Hensel. En multipliant U et V par a_n, on construit deux facteurs non triviaux du polynôme $a_n P$ modulo p^m.
En appliquant la même technique que ci-dessus, il est donc ainsi possible

– soit de prouver l'irréductibilité du polynôme $a_n P$, et donc de P dans $\mathbb{Q}[x]$,

– soit de construire une factorisation non triviale de $a_n P$ dans $\boldsymbol{Z}[x]$, et donc une factorisation de P dans $\mathbb{Q}[x]$.

Analyse du coût.

La remontée de Hensel s'effectue en $\mathcal{O}(m)$ produits de polynômes de degrés bornés par n, donc avec une complexité bornée par $\mathcal{O}(mn \log n)$ opérations entières[6].

Mais le problème vient du nombre de paires de facteurs qu'il peut être nécessaire de tester. En effet, dans le pire cas, 2^{r-1} factorisations distinctes de P en deux facteurs modulo p sont possibles, donc un nombre exponentiel dans le pire cas.

Mais d'un point de vue pratique, le nombre moyen de facteurs irréductibles d'un polynôme de degré n dans $\mathrm{F}_p[x]$ est $r = \mathcal{O}(\log n)$ (cf. [13]). Par suite, l'algorithme de Berlekamp a « en pratique » un coût en opérations entières polynomial en n, les opérations entières étant elles-mêmes de coût $\log^{\mathcal{O}(1)} h(P)$.

Bien qu'il existe des polynômes pour lesquels l'algorithme de Berlekamp a un coût exponentiel en fonction du degré des polynômes, il se comporte donc relativement bien en pratique.

3.2.4 Compléments

Différentes améliorations peuvent être apportées à l'algorithme de Berlekamp présenté ici. Différents algorithmes ont été construits pour permettre la factorisation de polynômes à plusieurs variables et lorsque le corps de base est une extension algébrique. Le lecteur intéressé pourra consulter les survols [7, 8] qui présentent les principaux algorithmes.

Dans les corps finis, nous avons vu que l'algorithme proposé était en complexité $\mathcal{O}(q)$, donc exponentiel en fonction de la taille du codage des éléments ($\log q$). Lorsque q est grand, l'algorithme de Cantor et Zassenhaus (cf. [5, 19]) permet de limiter le nombre de pgcd à calculer en construisant les éléments de F_q susceptibles de fournir un facteur non trivial. La complexité de cet algorithme est alors linéaire en $\log q$, tout en restant polynomiale en fonction du degré du polynôme.

6. Les entiers manipulés sont ici bornés par $2^n h(P)$.

Sur les rationnels, il existe deux algorithmes différents qui permettent de factoriser en temps polynomial. Les deux sont basés sur l'algorithme LLL (du nom de ses auteurs A.K. Lenstra, H.W. Lenstra et L. Lovàsz, cf. [12]). Étant donnés n vecteurs indépendants $b_1, b_2, \ldots, b_n$ à composantes entières, le réseau engendré par ces vecteurs est le sous-groupe additif $\Sigma_{i=1}^{n} \boldsymbol{Z} b_i$. L'algorithme LLL de réduction de base permet, dans un réseau ainsi défini, de trouver un vecteur qui soit « presque le plus court du réseau » (au sens de la norme Euclidienne). Cette recherche s'effectue en temps polynomial. Pour les deux algorithmes de factorisation, le principe d'application de LLL est le même : on construit des réseaux à partir d'informations connues « facilement » à partir du polynôme à factoriser, puis on caractérise les facteurs du polynôme comme étant des « vecteurs courts » dans ces réseaux. La réduction des bases correspondantes calcule alors les facteurs.

Deux types de réseaux peuvent être utilisés :

- La première méthode (cf. [12]) reprend le schéma suivi dans ce chapitre, seule la dernière phase est modifiée. Il est possible de caractériser les facteurs irréductibles de P comme petits vecteurs dans des réseaux construits à partir de la factorisation modulo p^m.

- La deuxième méthode (cf. [9]) se base sur des approximations numériques des racines de P avec une précision suffisante. À partir des puissances de ces approximations numériques, il est possible de construire des réseaux dont les polynômes minimaux associés aux racines exactes (et qui sont donc des facteurs irréductibles de P) sont des « vecteurs courts ».

3.3 Conclusion

Les exemples présentés dans ce chapitre montrent l'importance du calcul formel et de l'algorithmique mathématique pour construire des solutions performantes à un problème.

L'arithmétique dans les corps finis joue un rôle très important en calcul formel. Elle est fondamentale dans les algorithmes les plus rapides en arithmétique entière et polynomiale. Son utilisation pour le filtrage de chaînes de caractères illustre son intérêt pour la construction d'algorithmes probabilistes efficaces.

Chapitre 4

Intégration formelle etc

J.H. DAVENPORT School of Mathematical Sciences, University of Bath, Bath BA2 7AY, Royaume-Uni,

jhd@maths.bath.ac.uk

4.1 Dérivation formelle

Avant de parler de l'intégration formelle, il nous faut parler un peu de la dérivation formelle. En analyse, la dérivée d'une fonction est normalement définie par

$$f'(x) = \lim_{h \to 0} \frac{f(x+h) - f(x)}{h},$$

et on obtient ainsi la définition d'une fonction (éventuellement partielle) f'. De cette définition découlent les propriétés suivantes : $(f+g)' = f' + g'$ et $(fg)' = f'g + fg'$, d'où on déduit algébriquement $(f-g)' = f' - g'$ et $\left(\frac{f}{g}\right)' = \frac{f'g - fg'}{g^2}$. Mais il est aussi possible de prendre $(f+g)' = f' + g'$ et $(fg)' = f'g + fg'$ comme les règles définissant une opération *formelle*, appelée une dérivation. D'abord $0' = (0+0)' = 0' + 0'$, donc $0' = 0$. On a aussi $1' = (1 \cdot 1)' = 1' \cdot 1 + 1 \cdot 1' = 2 \cdot 1'$, donc $1' = 0$. Puis $2' = (1+1)' = 1' + 1' = 0$, et de même pour tous les entiers et ensuite pour les fractions.

Quelle est la valeur de x', où x est une inconnue ? Ceci dépend de la dérivation – s'il s'agit de $\frac{\partial}{\partial x}$, alors le résultat est 1, mais s'il s'agit de $\frac{\partial}{\partial y}$ $(y \neq x)$, alors x' vaut 0. **Dans la suite, nous supposons que $' = \frac{\partial}{\partial x}$, c'est-à-dire que nous dérivons, ou calculons une primitive, par rapport à x.**

Cette définition de $'$ nous permet d'introduire $\log x$ comme « la fonction » θ telle que $\theta' = \frac{1}{x}$ et plus généralement $\log f$ comme « la fonction » θ telle que $\theta' = \frac{f'}{f}$. Nous avons dit « la fonction », car en effet cette définition est « à une constante additive près ». De même, nous pouvons définir e^x comme « la fonction » θ telle que $\theta' = \theta$, ou, plus généralement, e^f comme « la fonction » θ telle que $\theta' = f'\theta$. Ici, ces définitions sont « à une constante multiplicative près ».

Avec ces définitions (et d'autres si l'on a besoin d'autres fonctions) on a la théorie, essentiellement algorithmique, de la *dérivation formelle*, qui a été étudiée assez tôt dans le développement des ordinateurs [Kahrimanian, 1953 ; Nolan, 1953].

4.2 Intégration formelle

Dans cette section, nous décrirons la théorie et la pratique de l'intégration formelle. Les ordinateurs sont très utilisés pour l'intégration numérique, c'est-à-dire la recherche des intégrales définies. Mais le Calcul Formel permet aussi l'intégration formelle, c'est-à-dire la recherche des primitives. Ces calculs, qui ne peuvent pas se faire numériquement, sont une des grandes réussites du Calcul Formel.

4.2.1 Introduction

Nous avons dit que la dérivation formelle a été étudiée assez tôt dans le développement des ordinateurs [Kahrimanian, 1953 ; Nolan, 1953], mais les premiers pas vers l'intégration sont dus à Slagle [1961]. Pourquoi ce délai ?

La raison remonte à une grande différence entre l'intégration formelle et la dérivation formelle. La dérivation, telle qu'on l'apprend au lycée, est largement la méthode de dérivation formelle. En fait, le vrai problème pour la dérivation est la simplification du résultat, car si l'on ne simplifie pas, on obtient que la dérivée de $2x+1$ est $0x+2*1+0$ ou, si l'on représente $2x+1$ sous la forme $2x^1+1x^0$,

$$0x^1+2\left(0(\log x)x^1+1*1*x^{1-1}\right)+0x^0+1\left(0(\log x)x^0+0*1*x^{0-1}\right).$$

Par contre, l'intégration paraît être une collection aléatoire d'astuces et de cas spéciaux. Il n'existe, semble-t-il, qu'une règle générale, $\int f+g=\int f+\int g$ (et cette règle n'est pas toujours valable – voir l'équation (1) de la section « Intégration des fractions rationnelles » et la discussion associée). Pour d'autres combinaisons de fonctions que l'addition (et la soustraction) il n'existe pas de règles générales. Par exemple, le fait qu'on sache intégrer $\exp x$ et x^2 n'implique pas qu'on sache intégrer leur composition $\exp x^2$: comme on le verra plus tard, cette fonction n'a aucune intégrale plus simple que $\int \exp x^2 dx$. Ainsi on apprend plusieurs « méthodes » telles que l'intégration par parties, l'intégration par substitution, l'intégration par recherche dans les listes publiées des intégrales etc. De plus, on ne sait pas quelle méthode ou quelle combinaison de méthodes sera efficace pour une intégrale donnée. Ainsi les premières démarches (celles de Slagle) étaient conçues d'après les mêmes heuristiques que celles qu'utilisent les humains.

Cette voie a été dépassée assez tôt (dès les travaux de Moses [1967 ; 1971]) par des méthodes vraiment algorithmiques. Il existe maintenant toute une théorie de l'intégration, que nous ne pouvons que résumer succinctement. Sauf indication contraire, toute intégration dans ce chapitre est par rapport à x, tout comme $'$ désigne $\frac{\partial}{\partial x}$.

Comme la dérivation est nettement plus simple que l'intégration, il convient de poser le problème d'intégration comme « problème inverse » de la dérivation, c'est-à-dire, étant donnée une fonction a, au lieu de chercher sa primitive b, nous demandons une fonction b telle que $b'=a$.

Définition. Étant données deux classes de fonctions A et B, le *problème d'intégration* de A à B est de donner un algorithme qui, pour tout membre a de A, ou bien donne un élément b de B tel que $a=b'$, ou bien démontre qu'il n'existe aucun élément b de B tel que $a=b'$.

Par exemple, si $A=\mathbf{Q}(x)$ et $B=\mathbf{Q}(x)$, alors la réponse pour $1/x^2$ doit être $-1/x$, tandis que pour $1/x$ elle doit être « impossible ». Par contre, si $B=\mathbf{Q}(x,\log x)$, alors la réponse pour $1/x$ doit être $\log x$.

Il existe un théorème de Richardson [1968] qui démontre que le problème d'intégration pour $A=B=\mathbf{Q}(i,\pi,\exp,\log,|.|)$ (où $|.|$ désigne la valeur absolue) est insoluble. Mais ceci ne doit pas désespérer les utilisateurs d'un algorithme d'intégration : en fait le problème de décider si une constante est nulle ou non est indécidable dans ce corps. Alors, Richardson prend une constante indécidable c et considère la fonction ce^{x^2}. Comme c'est bien connu (et nous le démontrons plus tard), la fonction e^{x^2} ne peut pas être intégrée dans B, on voit que ce^{x^2} n'est intégrable que si $c = 0$, question indécidable. Désormais, nous supposons que nos classes de fonctions sont *effectives*, c'est-à-dire que tout problème d'égalité est décidable.

4.2.2 Intégration des fonctions rationnelles

Dans cette section, nous traitons le cas de $A = C(x)$, où C est un corps de constantes (décidable, d'après les remarques que nous venons de faire). Toute fonction rationnelle f peut s'écrire sous la forme $p + q/r$, où p, q et r sont des polynômes, q et r sont premiers entre eux et le degré de q est inférieur à celui de r. Il est bien connu que

$$\int f + g = \int f + \int g, \tag{1}$$

mais dans la théorie de l'intégration formelle, il faut s'en méfier un peu. Il est bien possible que $\int f + g$ ait une forme explicite, bien que $\int f$ et $\int g$ n'en aient pas. Un exemple de ce phénomène est $\int x^x + (\log x)x^x$, dont l'intégrale est x^x, tandis que ses deux composants n'ont pas de primitives (en forme finie). En fait, on ne doit utiliser (1) que si l'on sait que deux parmi les trois intégrales existent.

Un polynôme p possède toujours une intégrale finie, donc (1) est valable pour $f = p$ et $g = q/r$. Donc le problème d'intégration de $f = p + q/r$ se réduit au problème d'intégration de p (qui est tout à fait simple) et de la fonction rationnelle *propre* q/r. Le reste de cette section est consacré à l'intégration d'une fonction rationnelle propre.

La méthode naïve

Si le polynôme r se factorise en facteurs linéaires, tel que $r = \prod_{i=1}^{n} (x - a_i)^{n_i}$, on peut faire une décomposition de q/r en éléments simples :

$$\frac{q}{r} = \sum_{i=1}^{n} \frac{b_i}{(x - a_i)^{n_i}},$$

où les b_i sont des polynômes de degré inférieur à n_i. Ces polynômes peuvent être divisés par $x - a_i$, afin de donner la décomposition suivante :

$$\frac{q}{r} = \sum_{i=1}^{n} \sum_{j=1}^{n_i} \frac{b_{i,j}}{(x - a_i)^j},$$

où les $b_{i,j}$ sont des constantes.

Cette décomposition peut alors être intégrée :

$$\int \frac{q}{r} = \sum_{i=1}^{n} b_{i,1} \log(x - a_i) - \sum_{i=1}^{n} \sum_{j=2}^{n_i} \frac{b_{i,j}}{(j-1)(x - a_i)^{j-1}}.$$

Ainsi, on a démontré que toute fonction rationnelle possède une intégrale qui peut s'exprimer comme une fonction rationnelle plus une somme de logarithmes de fonctions rationnelles à coefficients constants – c'est-à-dire que l'intégrale appartient au corps $C(x, \log)$.

Bien que cette démonstration soit assez connue, elle a plusieurs défauts du point de vue algorithmique.

i. Elle demande qu'on factorise r complètement, ce qui n'est pas toujours possible sans ajouter plusieurs quantités algébriques à C. Ajouter $\sqrt{2}$ pour intégrer $\frac{1}{x^2-2}$ est assez simple, mais en général la manipulation de ces extensions algébriques est souvent très fastidieuse.

ii. Même si les extensions algébriques ne sont pas nécessaires, il est assez coûteux de factoriser un polynôme r de degré élevé.

iii. Elle nécessite aussi une décomposition complète en éléments simples.

Afin de démontrer qu'il existe plusieurs intégrales qui se laissent calculer (à la main) sans ces grandes difficultés, considérons :

$$\int \frac{5x^4+60x^3+255x^2+450x+274}{x^5+15x^4+85x^3+225x^2+274x+120}$$
$$= \log(x^5+15x^4+85x^3+225x^2+274x+120)$$
$$= \log(x+1)+\log(x+2)+\log(x+3)+\log(x+4)+\log(x+5);$$

$$\int \frac{5x^4+60x^3+255x^2+450x+275}{x^5+15x^4+85x^3+225x^2+274x+120}$$
$$= \frac{25}{24}\log(x+1)+\frac{5}{6}\log(x+2)+\frac{5}{4}\log(x+3)+\frac{5}{6}\log(x+4)+\frac{25}{24}\log(x+5)$$

(l'expression avec un seul logarithme est trop longue pour être écrite ici : elle contient le logarithme d'un polynôme de degré 120, dont les plus grands coefficients ont 68 chiffres décimaux) ;

$$\int \frac{5x^4+1}{(x^5+x+1)^2} = \frac{-1}{(x^5+x+1)};$$

$$\int \frac{5x^4+1}{(x^5+x+1)} = \log\left(x^5+x+1\right).$$

Le premier est un exemple où l'on peut factoriser le dénominateur (qui n'est que $(x+1)(x+2)(x+3)(x+4)(x+5)$), mais la décomposition en éléments simples serait assez pénible. Le deuxième montre qu'une toute petite différence peut changer largement l'intégrale. Les deux derniers ont un dénominateur qui ne se factorise pas sur $\mathbf{Q}$ et il faut ajouter quatre extensions algébriques avant qu'il ne se factorise complètement (ou une extension de degré 120 !). Ainsi, on peut se poser le problème suivant : *Trouver un algorithme d'intégration des fonctions rationnelles qui ne fasse intervenir que les quantités algébriques nécessaires pour exprimer l'intégrale.*

La méthode d'Hermite

Cette méthode [Hermite, 1872] nous permet de déterminer la partie rationnelle de l'intégrale d'une fonction rationnelle sans faire intervenir la moindre quantité algébrique. De même, elle trouve la dérivée de la somme de logarithmes qui est aussi une fonction rationnelle avec coefficients dans le même corps que q/r. Nous avons remarqué qu'un facteur du dénominateur r qui apparaît à la puissance n, apparaît à la puissance $n-1$ dans le dénominateur de l'intégrale. Ceci peut nous faire penser à une décomposition sans facteurs multiples.

Supposons donc que r ait une décomposition sans facteurs multiples de la forme $\prod_{i=1}^{n} r_i^i$. Les r_i sont alors premiers entre eux et nous pouvons construire une décomposition en éléments simples :

$$\frac{q}{r} = \frac{q}{\prod_{i=1}^{n} r_i^i} = \sum_{i=1}^{n} \frac{q_i}{r_i^i}.$$

Nous savons que chaque élément à droite a une intégrale, donc (1) est valable et il suffit d'intégrer chaque élément séparément. Ceci se fait en utilisant l'identité de Bezout appliquée à r_i et r_i' qui sont premiers entre eux et tels qu'il existe a et b satisfaisant à $ar_i + br_i' = 1$.

$$\begin{aligned}
\int \frac{q_i}{r_i^i} &= \int \frac{q_i(ar_i + br_i')}{r_i^i} \\
&= \int \frac{q_i a}{r_i^{i-1}} + \int \frac{q_i b r_i'}{r_i^i} \\
&= \int \frac{q_i a}{r_i^{i-1}} + \int \frac{(q_i b/(i-1))'}{r_i^{i-1}} - \left(\frac{q_i b/(i-1)}{r_i^{i-1}}\right)' \\
&= -\left(\frac{q_i b/(i-1)}{r_i^{i-1}}\right) + \int \frac{q_i a + (q_i b/(i-1))'}{r_i^{i-1}},
\end{aligned}$$

et nous avons pu réduire l'exposant de r_i. Nous pouvons continuer de cette façon jusqu'à ce que l'exposant devienne 1, quand l'intégrale restante est une somme de logarithmes.

La méthode de Horowitz-Ostrogradski

La méthode d'Hermite est assez bien adaptée aux calculs manuels, mais elle a le désavantage de nécessiter plusieurs sous-algorithmes (décomposition sans facteurs multiples, décomposition en éléments simples, identité de Bezout), ce qui entraîne une programmation assez compliquée. Aussi Horowitz [1969 ; 1971] a proposé la méthode suivante qui semble avoir été connue du mathématicien russe Ostrogradski.

L'objectif reste de pouvoir écrire $\int q/r$ sous la forme $q_1/r_1 + \int q_2/r_2$, où l'intégrale restante ne donne qu'une somme de logarithmes quand on la résout. D'après la méthode d'Hermite et les discussions antérieures, nous savons que r_1 a les mêmes facteurs que r, mais avec l'exposant diminué de 1, que r_2 n'a pas de facteurs multiples et que ses facteurs sont tous des facteurs de r. Les arguments standard de « décompositions sans facteurs multiples » impliquent que $r_1 = \operatorname{pgcd}(r, r')$ et que r_2 divise $r/\operatorname{pgcd}(r, r')$. On peut supposer que q_2/r_2 n'est pas écrit sous forme réduite et donc que $r_2 = r/\operatorname{pgcd}(r, r')$.

Alors

$$\begin{aligned}
\frac{q}{r} &= \left(\frac{q_1}{r_1}\right)' + \frac{q_2}{r_2} \\
&= \frac{q_1'}{r_1} - \frac{q_1 r_1'}{r_1^2} + \frac{q_2}{r_2} \\
&= \frac{q_1' r_2 - q_1 s + q_2 r_1}{r},
\end{aligned}$$

où $s = r_1' r_2 / r_1$ (la division ici est exacte).

Ainsi, notre problème se réduit à un problème purement polynomial, à savoir

$$q = q_1' r_2 - q_1 s + q_2 r_1, \tag{2}$$

où q, s, r_1 et r_2 sont connus, et q_1 et q_2 sont à déterminer. Mais les degrés de q_1 et q_2 sont inférieurs aux degrés m et n de r_1 et r_2 respectivement. Écrivons donc $q_1 = \sum_{i=0}^{m-1} a_i x^i$ et $q_2 = \sum_{i=0}^{n-1} b_i x^i$. L'équation (2) se réécrit comme un système de $m + n$ équations linéaires en $n + m$ inconnues. De plus, ce système est résoluble et l'intégration (au moins ce sous-problème) se réduit à de l'algèbre linéaire.

La partie logarithmique

Les deux méthodes exposées dans les sections précédentes peuvent réduire l'intégration d'une fonction rationnelle quelconque à l'intégration d'une fonction rationnelle (soit q/r) dont l'intégrale ne serait qu'une somme de logarithmes. Nous savons que cette intégrale peut être résolue en factorisant le dénominateur complètement, mais comme on l'a déjà vu, ceci n'est pas toujours nécessaire pour exprimer les résultats. Le vrai problème est de trouver l'intégrale sans utiliser d'autres nombres algébriques que ceux nécessaires à l'expression du résultat. Ce problème a été résolu par Rothstein [1976] et par Trager [1976] – nous donnons la résolution due à ce dernier.

Supposons que

$$\int \frac{q}{r} = \sum_{i=1}^{n} c_i \log v_i \tag{3}$$

soit une solution à cette intégrale où la partie droite utilise le moins possible d'extensions algébriques. Les c_i sont des constantes et, en général, les v_i sont des fonctions rationnelles. Comme $\log(a/b) = \log a - \log b$, on peut supposer, sans restreindre la généralité, que les v_i sont des polynômes. De plus, nous pouvons les décomposer sans facteurs multiples, ce qui n'ajoute pas d'extensions algébriques et appliquer la règle $\log \prod p_i^i = \sum i \log p_i$. En outre, comme $c \log pq + d \log pr = (c+d) \log p + c \log q + d \log r$, nous pouvons supposer que les v_i sont premiers entre eux, tout en gardant la minimalité du nombre d'extensions algébriques (bien que le nombre de logarithmes puisse changer). En outre, on peut supposer que les c_i sont tous différents.

En dérivant l'équation (3), nous trouvons

$$\frac{q}{r} = \sum_{i=1}^{n} \frac{c_i v_i'}{v_i}. \tag{3'}$$

L'hypothèse que les v_i sont sans facteurs multiples entraîne qu'aucun élément de la somme ne peut se simplifier et l'hypothèse que les v_i sont premiers entre eux implique qu'aucune annulation ne peut exister dans la somme. Ceci nécessite que les v_i soient exactement les facteurs de r, c'est-à-dire $r = \prod_{i=1}^{n} v_i$. Écrivons $u_i = \prod_{j \neq i} v_j$. Alors, on peut dériver le produit des v_i, afin de déterminer que $r' = \sum v_i' u_i$. Si nous chassons les dénominateurs de $(3')$, nous trouvons $q = \sum c_i v_i' u_i$. Ces deux expressions pour q et r' permettent les calculs suivants :

$$\begin{aligned}
v_k &= \operatorname{pgcd}(0, v_k) \\
&= \operatorname{pgcd}\left(q - \textstyle\sum c_i v_i' u_i, v_k\right) \\
&= \operatorname{pgcd}\left(q - c_k v_k' u_k, v_k\right) \\
&\text{car tous les autres } u_i \text{ sont divisibles par } v_k \\
&= \operatorname{pgcd}\left(q - c_k \textstyle\sum v_i' u_i, v_k\right) \\
&\text{pour la même raison} \\
&= \operatorname{pgcd}\left(q - c_k r', v_k\right).
\end{aligned}$$

Si $l \neq k$, nous trouvons

$$\begin{aligned}
\operatorname{pgcd}\left(q - c_k r', v_l\right) &= \operatorname{pgcd}\left(\textstyle\sum c_i v_i' u_i - c_k \sum v_i' u_i, v_l\right) \\
&= \operatorname{pgcd}\left(c_l v_l' u_l - c_k v_l' u_l, v_l\right) \\
&\text{car tous les autres } u_i \text{ sont divisibles par } v_l \\
&= 1
\end{aligned}$$

car v_l n'a pas de facteurs multiples ou de facteurs en commun avec le produit u_l des autres v_i. À l'aide de ces deux calculs, nous procéderons au calcul suivant :

$$\begin{aligned}
\operatorname{pgcd}(q-c_kr',r) &= \operatorname{pgcd}\left(q-c_kr', \textstyle\prod_{i=1}^n v_i\right)\\
&= \textstyle\prod_{i=1}^n \operatorname{pgcd}(q-c_kr',v_i)\\
&\text{car les } v_i \text{ sont premiers entre eux}\\
&= \operatorname{pgcd}(q-c_kr',v_k)\\
&\text{car tous les autres termes s'annulent}\\
&= v_k
\end{aligned}$$

Ainsi, si on connaît les c_k, on peut calculer les v_k. En outre, les c_k sont précisément les valeurs de y pour lesquelles $\operatorname{pgcd}(q-yr',r)\neq 1$. Or ces valeurs peuvent être calculées au moyen de l'idée du résultant. $\operatorname{Res}_x(q-yr',r)$ est un polynôme en y qui s'annule si et seulement si le pgcd est non trivial. Il suffit donc de calculer ce polynôme (ce qui peut se faire sans aucune extension algébrique), trouver ses racines (ce qui peut nécessiter des extensions algébriques) et, pour chacune de ses racines c_k, déterminer $v_k = \operatorname{pgcd}(q-c_kr',r)$.

4.2.3 L'intégration des fonctions plus compliquées

Dès qu'on sort du cadre des fonctions rationnelles, on trouve des fonctions dont on ne peut pas exprimer l'intégrale en termes plus simples. Par exemple, on peut dire

$$\int e^{-x^2} = \sqrt{\frac{\pi}{2}}\operatorname{erf} x,$$

mais ceci n'est qu'une réexpression de la définition de la fonction erf. On peut soulever la remarque que $\int 1/x = \log x$ n'est qu'une réexpression de la définition de la fonction log et ceci est valable. La différence est que log est « bien connue » (dans un sens que nous préciserons). Autrement dit, quelles classes B admettons-nous comme classes d'intégrales possibles ? Liouville [1835] a proposé la définition suivante qui semble tout à fait adéquate pour nos besoins actuels – voir la section « Intégration de fonctions non élémentaires » pour une généralisation.

Définition. Soit K un corps de fonctions. La fonction θ est *un générateur élémentaire* sur K si :

(a) θ est algébrique sur K, c'est-à-dire θ satisfait à une équation polynomiale avec coefficients dans K ;

(b) θ est une *exponentielle* sur K, c'est-à-dire il existe un η dans K tel que $\theta' = \eta'\theta$, ce qui est une façon algébrique de dire que $\theta = \exp\eta$;

(c) θ est un *logarithme* sur K, c'est-à-dire il existe un η dans K tel que $\theta' = \eta'/\eta$, ce qui est une façon algébrique de dire que $\theta = \log\eta$.

Définition. Soit K un corps de fonctions. Un surcorps $K(\theta_1,\ldots,\theta_n)$ de K s'appelle un *corps de fonctions élémentaires sur K* si chaque θ_i est un générateur élémentaire sur K. Une fonction est *élémentaire sur K* si elle appartient à un corps de fonctions élémentaires sur K.
Si K est omis, on entend $\mathbf{C}(x)$ le corps des fonctions rationnelles.

Cette définition regroupe aussi les fonctions trigonométriques – par exemple $\sin x$ est élémentaire (sur $\mathbf{C}(x)$), car

$$\sin x = \frac{1}{2i}\left(e^{ix}+e^{-ix}\right) = \frac{1}{2i}\left(\theta+\frac{1}{\theta}\right),$$

où θ est l'exponentielle de ix. Il en est de même pour les inverses des fonctions trigonométriques – par exemple $\tan^{-1} x = \log\left(\frac{x+i}{x-i}\right)$. Nous utilisons la notation $K(\text{elem})$ pour la classe des fonctions élémentaires sur K.

La section précédente a démontré que toute fonction rationnelle possède une intégrale élémentaire. En outre, cette intégrale est d'une forme spéciale – une fonction rationnelle (dont le calcul ne nécessite aucune extension algébrique) plus une somme de logarithmes à coefficients constants. On retrouve cette même forme dans des contextes nettement plus généraux, comme le résultat suivant le montre.

Théorème (Principe de Liouville). Soit f une fonction appartenant à un corps de fonctions K. Si f a une intégrale élémentaire sur K, elle a une intégrale de la forme suivante :

$$\int f = v_0 + \sum_{i=1}^{n} c_i \log v_i, \tag{4}$$

où v_0 appartient à K, les v_i appartiennent à $\hat{K}$, une extension de K par un nombre fini de constantes algébriques sur K et les c_i appartiennent à $\hat{K}$ et sont des constantes.
Autrement dit, si f possède une intégrale élémentaire sur K, f est de la forme suivante :

$$f = v_0' + \sum_{i=1}^{n} \frac{c_i v_i'}{v_i}. \tag{4'}$$

Ce théorème est fondamental pour la théorie de l'intégration en termes de fonctions élémentaires. Dans la suite, nous utilisons la locution « *on sait intégrer dans A avec résultat dans la classe B* » comme abréviation de la locution « il existe un algorithme qui, étant donnée une fonction a de A, donne une fonction b de B telle que $a = b'$ ou démontre qu'il n'existe aucune fonction de B ayant cette propriété ». Si B est omis, on entend la classe $A(\text{elem})$.

4.2.4 Intégration de fonctions logarithmiques

Cette section se consacre à l'étude des corps de fonctions engendrés par un *logarithme*. Dans cette section, nous travaillons dans le cadre suivant :

(i) K est un corps de fonctions, supposé effectif, tel que l'on sait intégrer dans K ;

(ii) θ est un logarithme sur K, $\theta' = \eta'/\eta$ et θ est supposé transcendant sur K ;

(iii) $K(\theta)$ a les mêmes constantes que K.

La dernière hypothèse évite les ennuis qui peuvent arriver si θ donne lieu à de nouvelles constantes. Par exemple, si $K = \mathbf{Q}(x, \log x)$ et $\theta = \log 2x$, $K(\theta)$ contient la constante $\log 2$. Cette hypothèse, aussi bien que l'hypothèse que θ soit transcendant, peut être vérifiée à l'aide du Théorème de Structure de Risch [1979]. Le but est de démontrer que l'on sait intégrer dans $K(\theta)$. Les méthodes sont largement semblables à celles de l'intégration des fonctions rationnelles, mais il existe plusieurs endroits où l'on doit faire attention.

Le lemme de décomposition

Nous pouvons écrire la fonction f de $K(\theta)$ sous la forme $p + q/r$, où p, q et r sont des polynômes de $K[\theta]$, q et r sont premiers entre eux et le degré de q est inférieur à celui de r. Est-il alors possible de traiter p et q/r séparément, c'est-à-dire d'appliquer la décomposition (1) ?

Lemme (de décomposition). Si f possède une intégrale élémentaire sur K, alors p et q/r possèdent tous deux une intégrale élémentaire sur K.
Démonstration. D'après le principe de Liouville (sous la forme de l'équation (4′)),

$$p+\frac{q}{r}=v_0'+\sum_{i=1}^{n}\frac{c_iv_i'}{v_i}.$$

Dans cette décomposition, nous pouvons supposer que $v_1,\ldots,v_k$ sont des polynômes unitaires en θ dont les coefficients appartiennent à K, tandis que $v_{k+1},\ \ldots,\ v_n$ appartiennent à K. En outre, nous pouvons écrire v_0 sous la forme $\hat{p}+\hat{q}/\hat{r}$, où $\hat{p}$, $\hat{q}$ et $\hat{r}$ sont des polynômes en θ et $\hat{q}/\hat{r}$ est une fonction rationnelle propre. Avec ces conventions et en utilisant les propriétés suivantes des dérivées : la dérivée d'un polynôme est un polynôme et la dérivée d'une fonction rationnelle propre est une fonction rationnelle propre, nous voyons que :

$$\begin{aligned}
p+\tfrac{q}{r} &= \left(\hat{p}+\tfrac{\hat{q}}{\hat{r}}\right)'+\textstyle\sum_{i=1}^{n}\frac{c_iv_i'}{v_i}\\
&= \underbrace{\hat{p}'+\sum_{i=k+1}^{n}\frac{c_iv_i'}{v_i}}_{\text{polynôme}}\\
&\quad+\underbrace{\left(\frac{\hat{q}}{\hat{r}}\right)'+\sum_{i=1}^{k}\frac{c_iv_i'}{v_i}}_{\text{fonction rationnelle propre}}
\end{aligned}$$

Comme la décomposition polynôme + fonction rationnelle propre est univoque (nous rappelons que θ est supposé transcendant), cette équation nous permet de déduire les deux équations suivantes :

$$\begin{aligned}
p &= \hat{p}'+\textstyle\sum_{i=k+1}^{n}\frac{c_iv_i'}{v_i}\\
\tfrac{q}{r} &= \left(\tfrac{\hat{q}}{\hat{r}}\right)'+\textstyle\sum_{i=1}^{k}\frac{c_iv_i'}{v_i}
\end{aligned}$$

Ces deux équations s'intègrent formellement, afin de nous donner :

$$\begin{aligned}
\textstyle\int p &= \hat{p}+\textstyle\sum_{i=k+1}^{n}c_i\log v_i\\
\textstyle\int \frac{q}{r} &= \tfrac{\hat{q}}{\hat{r}}+\textstyle\sum_{i=1}^{k}c_i\log v_i
\end{aligned}$$

Ces équations nous démontrent le résultat et nous fournissent, de plus, des précisions sur la forme des intégrales. Dans les sous-sections suivantes, nous considérerons ces deux parties séparément.

La partie polynomiale

Supposons que p soit écrit sous la forme $\sum_{i=0}^{m}A_i\theta^i$ et que $\hat{p}$ soit écrit sous la forme $\sum_{i=0}^{n}B_i\theta^i$. Les dernières équations de la démonstration du lemme de décomposition impliquent que

$$\begin{aligned}
\textstyle\sum_{i=0}^{m}A_i\theta^i &= \left(\textstyle\sum_{i=0}^{n}B_i\theta^i\right)'+\textstyle\sum_{i=k+1}^{n}\frac{c_iv_i'}{v_i}\\
&= \textstyle\sum_{i=0}^{n}B_i'\theta^i+\sum_{i=1}^{n}iB_i\theta'\theta^{i-1}+\sum_{i=k+1}^{n}\frac{c_iv_i'}{v_i}.
\end{aligned}$$

Nous rappelons que les v_i qui entrent dans cette expression appartiennent à K. Comme θ est transcendant, on peut déduire que les coefficients de chaque puissance de θ dans les deux parties doivent être égaux. Ceci implique que $n = m$ ou $n = m + 1$. Les coefficients de θ^{m+1} nous donnent l'équation $0 = B'_{m+1}$, d'où l'on déduit que B_{m+1} doit être une constante.

Les coefficients de θ^m (en supposant que $m > 0$) nous donnent l'équation :

$$A_m = B'_m + (m+1)B_{m+1}\theta'.$$

Cette équation peut se réécrire sous une forme intégrale :

$$\begin{aligned} B_m &= \int (A_m - (m+1)B_{m+1}\theta') \\ &= -(m+1)B_{m+1}\theta + \int A_m. \end{aligned}$$

Cette équation implique que A_m doit avoir une intégrale élémentaire et l'hypothèse (i) de cette section nous assure qu'il existe un algorithme qui peut trouver cette intégrale (ou démontrer que cette fonction n'a pas d'intégrale élémentaire, ce qui nous informe que p n'en a pas non plus). En outre, cette intégrale doit être d'une forme spéciale : un élément de K plus un multiple (constant) de θ. θ est le seul logarithme qui peut être ajouté à K pour que cette intégrale soit résolue. (Il est donc possible d'optimiser cette intégration de A_m mais ceci complique beaucoup la présentation, donc nous ne le faisons pas ici, bien que ce soit assez important en pratique.) Cette intégration détermine B_{m+1} et détermine B_m à une constante près. Appelons cette constante b_m.

Les coefficients de θ^{m-1} (en supposant que $m > 1$) nous donnent l'équation :

$$A_{m-1} = B'_{m-1} + m(B_m + b_m)\theta'.$$

Cette équation peut se réécrire sous une forme intégrale :

$$\begin{aligned} B_{m-1} &= \int (A_{m-1} - m(B_m + b_m)\theta') \\ &= -mb_m\theta + \int A_{m-1} - mB_m\theta'. \quad (5) \end{aligned}$$

Comme nous l'avons déjà fait pour les coefficients de θ^m, nous déduisons que $A_{m-1} - mB_m\theta'$ possède une intégrale élémentaire qui doit appartenir à K, sauf, éventuellement, un multiple constant de θ qui détermine b_m. Ainsi B_{m-1} est déterminé à une constante près, que nous appelons b_{m-1}.

Nous continuons de cette façon jusqu'aux coefficients de θ^0.

$$A_0 = B'_0 + 1(B_1 + b_1)\theta' + \sum_{i=k+1}^{n} \frac{c_i v'_i}{v_i}.$$

Cette équation peut se réécrire sous une forme intégrale :

$$\begin{aligned} B_0 + \textstyle\sum_{i=k+1}^{n} c_i \log v_i &= \int (A_0 - (B_1 + b_1)\theta') \\ &= -b_1\theta + \int A_0 - B_1\theta'. \end{aligned}$$

Nous en déduisons que $A_0 - B_1\theta'$ possède une intégrale élémentaire. Cette fois-ci, il n'est pas nécessaire que l'intégrale appartienne à K et elle peut contenir de nouveaux logarithmes. Le coefficient de θ dans cette intégrale nous donne b_1. Comme auparavant, B_0 n'est déterminé qu'à une constante près, mais cette constante est la constante d'intégration qui reste indéterminée. Ainsi, nous avons trouvé toute l'intégrale de la partie polynomiale (ou bien nous avons démontré qu'il n'en existe pas).

La partie rationnelle et logarithmique

Les dernières équations de la démonstration du lemme de décomposition impliquent que

$$\frac{q}{r} = \left(\frac{\hat{q}}{\hat{r}}\right)' + \sum_{i=1}^{k} \frac{c_i v_i'}{v_i}.$$

Le problème est semblable à celui des fonctions rationnelles et de même la solution. En effet, la méthode d'Hermite est valable dans ce cas comme pour les fonctions rationnelles, sauf qu'il faut noter la signification du symbole $'$. Dans le cas d'un polynôme en l'indéterminée x,

$$\left(\sum_{i=0}^{n} a_i x^i\right)' = \sum_{i=1}^{n} i a_i x^{i-1},$$

tandis que pour un polynôme en l'indéterminée θ,

$$\left(\sum_{i=0}^{n} a_i \theta^i\right)' = \sum_{i=1}^{n} i a_i \theta^{i-1} + \sum_{i=0}^{n} a_i{}' \theta^i.$$

Tous les résultats de décomposition sans facteurs multiples et éléments simples restent valables et la méthode d'Hermite (ou, plus précisément, la méthode d'Hermite-Ostrowski puisque c'est Ostrowski [1946] qui a fait cette généralisation) marche.

La partie logarithmique se calcule presque de la même façon que dans le cas des fonctions rationnelles. Nous avons démontré que les coefficients des logarithmes qui, d'après le théorème de Liouville, doivent être des constantes, sont les racines du polynôme $\mathrm{Res}_\theta(q - yr', r)$. Mais dans le contexte actuel, on n'est plus assuré que ces racines soient des constantes. Prenons l'exemple de $\int dx/\log x$, où $q = 1$ et $r = \log x$.

$$\mathrm{Res}_{\log x}\left(1 - y\frac{1}{x}, \log x\right) = \left|1 - \frac{y}{x}\right| = \frac{x-y}{x}$$

et la racine de ce polynôme est $y = x$ qui n'est pas une constante. Dans ce cas, nous pouvons conclure que l'intégrale n'est pas élémentaire car une intégrale élémentaire doit avoir des coefficients constants pour les logarithmes. Si nous appliquons la même méthode à $\int dx/x \log x$, nous trouvons le polynôme $y = 1$ et l'intégrale devient $\log\log x$.

4.2.5 Intégration de fonctions exponentielles

Cette section est consacrée à l'étude de fonctions engendrées par une *exponentielle*. Le lecteur doit retrouver ici la même démarche que dans la section précédente, sauf là où les comportements différents des fonctions logarithmes et exponentielles nous obligent à faire des distinctions. Dans cette section, nous travaillons dans le cadre suivant :

(i) K est un corps de fonctions, supposé effectif, tel que l'on sache intégrer dans K et résoudre l'équation différentielle $y' + fy = g$ dans K (voir la section suivante pour la résolution de ce problème) ;

(ii) θ est une exponentielle sur K, $\theta' = \eta'\theta$ et θ est supposée transcendante sur K ;

(iii) $K(\theta)$ a les mêmes constantes que K.

La dernière hypothèse évite les ennuis qui peuvent arriver si θ donne lieu à de nouvelles constantes. Par exemple, si $K = \mathbf{Q}(x, \exp x)$ et $\theta = \exp(x+1)$, $K(\theta)$ contient la constante e. Cette hypothèse, aussi bien que l'hypothèse que θ est transcendante, peut être vérifiée à l'aide du Théorème de Structure de Risch [1979]. Le but est de démontrer que l'on sait intégrer dans $K(\theta)$. Les méthodes pour la partie rationnelle sont largement semblables à celles de l'intégration des fonctions logarithmiques, mais la partie polynomiale est assez différente.

Il faut noter que le choix de θ est un peu arbitraire. $\frac{1}{\theta}$ est également une exponentielle : celle de $-\eta$. Il est donc évident que la « partie polynomiale » doit tenir compte de cette symétrie. Dans la section précédente, nous avons utilisé les idées de décomposition sans facteurs multiples etc., en disant qu'elles sont aussi valables pour le cas de polynômes en une variable logarithmique. Elles ne sont pas valables dans le cas d'une variable exponentielle : par exemple, si $\theta = \exp x$, le polynôme $p = \theta$ est sans facteurs multiples, mais $\operatorname{pgcd}(p, p') = \theta$ et un tel pgcd non trivial nous conduit normalement à déduire que p possède un facteur multiple. Il n'est pas difficile de vérifier que ces idées sont valables si θ ne divise pas p.

Le lemme de décomposition

Nous pouvons écrire notre fonction f de $K(\theta)$ sous la forme $p + q/r$, où p est un polynôme généralisé (c'est-à-dire $\sum_{i=-m}^{n} a_i\theta^i$), q et r sont des polynômes de $K[\theta]$ tels que θ ne divise pas r, q et r sont premiers entre eux et le degré de q est inférieur à celui de r. Est-il possible de traiter p et q/r séparément, c'est-à-dire d'appliquer la décomposition (1) ?

Lemme (de décomposition). Si f possède une intégrale élémentaire sur K, alors chacun des termes de p, ainsi que q/r, possèdent une intégrale élémentaire sur K.

Démonstration. D'après le principe de Liouville (sous la forme de l'équation (4′)),

$$p + \frac{q}{r} = v_0' + \sum_{i=1}^{n} \frac{c_i v_i'}{v_i}.$$

Dans cette décomposition, nous pouvons supposer que $v_1, \ldots, v_k$ sont des polynômes unitaires en θ de degré n_i dont les coefficients appartiennent à K, tandis que $v_{k+1}, \ldots, v_n$ appartiennent à K. Nous pouvons aussi supposer qu'aucun v_i n'est divisible par θ, car $\log\theta = \eta$. En outre, nous pouvons écrire v_0 sous la forme $\hat{p} + \hat{q}/\hat{r}$, de la même manière que nous avons décomposé f. Avec ces conventions et en utilisant les propriétés que la dérivée d'un polynôme généralisé est un polynôme généralisé et que la dérivée d'une fonction rationnelle propre (telle que θ ne divise pas son dénominateur) est une fonction rationnelle propre, nous voyons que :

$$\begin{aligned} p + \tfrac{q}{r} &= \left(\hat{p} + \tfrac{\hat{q}}{\hat{r}}\right)' + \textstyle\sum_{i=1}^{n} \tfrac{c_i v_i'}{v_i} \\ &= \underbrace{\hat{p}' + \sum_{i=k+1}^{n} \frac{c_i v_i'}{v_i}}_{\text{partie « p »}} + \underbrace{\left(\frac{\hat{q}}{\hat{r}}\right)' + \sum_{i=1}^{k} \frac{c_i v_i'}{v_i}}_{\text{partie « r »}}. \end{aligned}$$

Dans la section sur les fonctions logarithmiques, nous avons remarqué que les deux parties de cette décomposition étaient un polynôme et une fonction rationnelle propre. Cette observation n'est pas valable dans le cas actuel. Supposons que v_i (avec $i \le k$) s'écrive sous la forme $\theta^{n_i} + \sum_{j=0}^{n_i-1} a_i\theta^i$. Le polynôme v_i' est aussi un polynôme de degré n_i, à savoir

$$n_i\eta'\theta^{n_i} + \left(\sum_{j=0}^{n_i-1} a_i\theta^i\right)',$$

et $c_i v_i'/v_i$ n'est donc pas une fraction rationnelle propre. Cependant,

$$\frac{c_i v_i'}{v_i} - c_i n_i \eta' = \frac{c_i(v_i' - n_i \eta' v_i)}{v_i}$$

est une fraction rationnelle propre. Nous avons donc la décomposition suivante :

$$\begin{aligned} p + \tfrac{q}{r} &= \underbrace{\hat{p}' + \sum_{i=k+1}^{n} \frac{c_i v_i'}{v_i} + \eta' \sum_{i=1}^{k} c_i n_i}_{\text{polynôme généralisé}} \\ &\quad + \underbrace{\left(\frac{\hat{q}}{\hat{r}}\right)' + \sum_{i=1}^{k} \frac{c_i(v_i' - n_i \eta' v_i)}{v_i}}_{\text{fonction rationnelle propre}} \end{aligned}$$

(où, en outre, θ ne divise pas le dénominateur de la deuxième partie).

Comme la décomposition polynôme généralisé + fonction rationnelle propre (dont θ ne divise pas le dénominateur) est univoque (nous rappelons que θ est supposé transcendant), cette équation nous permet de déduire les deux équations suivantes :

$$\begin{aligned} p &= \hat{p}' + \textstyle\sum_{i=k+1}^{n} \frac{c_i v_i'}{v_i} + \eta' \sum_{i=1}^{k} c_i n_i \\ \tfrac{q}{r} &= \left(\tfrac{\hat{q}}{\hat{r}}\right)' + \textstyle\sum_{i=1}^{k} \frac{c_i(v_i' - n_i \eta' v_i)}{v_i}. \end{aligned}$$

La deuxième équation s'intègre formellement, afin de nous donner :

$$\int \frac{q}{r} = \frac{\hat{q}}{\hat{r}} + \sum_{i=1}^{k} c_i(\log v_i - n_i \eta).$$

Dans la première équation, écrivons $p = \sum_{i=-m}^{n} A_i \theta^i$ et $\hat{p} = \sum_{i=-m'}^{n'} B_i \theta^i$, où les A_i et B_i appartiennent à K. Comme θ est transcendant, on peut déduire que les coefficients de chaque puissance de θ dans les deux parties doivent être égaux. Ceci implique que $n = n'$ et $m = m'$. Comme la dérivée de $B_i \theta^i$ est $(B_i' + i\eta' B_i)\theta^i$, nous trouvons que, pour $i \neq 0$, $A_i \theta^i = (B_i' + i\eta' B_i)\theta^i$, ce qui implique que tous ces termes-ci possèdent des intégrales. Comme la partie rationnelle possède aussi une intégrale, il faut que le terme qui reste, c'est-à-dire A_0, possède aussi une intégrale.

La partie polynomiale généralisée

Nous venons de démontrer que, pour $i \neq 0$,

$$\int A_i \theta^i = B_i \theta^i,$$

où B_i appartient à K et satisfait à l'équation différentielle $B_i' + i\eta' B_i = A_i$. Par hypothèse nous avons un algorithme qui résout ce problème dans K, ou bien en déterminant B_i, ou bien en démontrant qu'aucun B_i dans K ne satisfait à cette équation (voir la section « ÉDO en forme finie » pour les détails).

A_0 appartient à K et son intégrale peut donc être déterminée par l'algorithme d'intégration dans K. Il est à noter que nous devons retirer $\eta \sum_{i=1}^{k} c_i n_i$ de cette intégrale pour trouver l'intégrale de f, car cette somme a été ajoutée par le processus de décomposition.

La partie rationnelle et logarithmique

Les dernières équations de la démonstration du lemme de décomposition impliquent que

$$\frac{q}{r} = \left(\frac{\hat{q}}{\hat{r}}\right)' + \sum_{i=1}^{k} \frac{c_i(v_i' - n_i\eta' v_i)}{v_i}.$$

En effet, la méthode d'Hermite est valable dans ce cas comme pour les fonctions rationnelles et logarithmiques, car θ ne divise pas le dénominateur r, donc la décomposition $r = \prod_{i=1}^{n} r_i^i$ sans facteurs multiples marche et de plus, chaque r_i et sa dérivée sont premiers entre eux.

Nous pouvons donc nous ramener au cas où r n'a pas de facteurs multiples. Auparavant, ceci indiquait que l'intégrale était une somme de logarithmes, mais maintenant cela indique que

$$\int \frac{q}{r} = \sum_{i=1}^{k} c_i(\log v_i - n_i\eta).$$

La méthode que nous utiliserons pour trouver les c_i et les v_i est assez similaire à celle utilisée pour les fonctions rationnelles de x, mais nécessite des modifications techniques. Comme auparavant, nous pouvons supposer que les v_i sont des polynômes sans facteurs multiples et premiers entre eux. L'analogue de l'équation $(3')$ est

$$\frac{q}{r} = \sum_{i=1}^{n} \frac{c_i(v_i' - n_i\eta' v_i)}{v_i}. \tag{6}$$

Aucune annulation ne peut exister dans cette somme et ceci nécessite que les v_i doivent être exactement les facteurs de r, c'est-à-dire $r = \prod_{i=1}^{n} v_i$. Écrivons $u_i = \prod_{j\neq i} v_j$. Alors, on peut dériver le produit des v_i, afin de déterminer que $r' = \sum v_i' u_i$. Si nous chassons les dénominateurs de (6), nous trouvons $q = \sum c_i(v_i' - n_i\eta' v_i)u_i$.

$$\begin{aligned}
v_k &= \mathrm{pgcd}(0, v_k)\\
&= \mathrm{pgcd}\left(q - \textstyle\sum c_i(v_i' - n_i\eta' v_i)u_i, v_k\right)\\
&= \mathrm{pgcd}\left(q - c_k(v_k' - n_k\eta' v_k)u_k, v_k\right)\\
&\text{car tous les autres } u_i \text{ sont divisibles par } v_k\\
&= \mathrm{pgcd}\left(q - c_k \textstyle\sum (v_i' - n_i\eta' v_i)u_i, v_k\right)\\
&\text{pour la même raison}\\
&= \mathrm{pgcd}\left(q - c_k(r' - \eta' r \textstyle\sum n_i), v_k\right).
\end{aligned}$$

Si $l \neq k$, nous trouvons, comme pour les fractions rationnelles,

$$\mathrm{pgcd}\left(q - c_k(r' - \eta' r \sum n_i), v_l\right) = 1.$$

À l'aide de ces deux calculs, nous déterminons

$$\mathrm{pgcd}\left(q - c_k(r' - \eta' r \sum n_i), r\right) = v_k.$$

Cette formule fait intervenir $\sum n_i$, mais ceci n'est que le degré de r, que nous pouvons appeler N. Ainsi, si on connaît les c_k, on peut calculer les v_k. En outre, les c_k sont précisément les valeurs de y pour lesquelles $\mathrm{pgcd}(q - y(r' - N\eta' r), r) \neq 1$. Or ces valeurs peuvent être calculées au moyen de l'idée du résultant : $\mathrm{Res}_\theta(q - y(r' - N\eta' r), r)$ est un polynôme en y qui s'annule si et seulement si le pgcd est non trivial. Il suffit donc de calculer ce polynôme (ce qui peut se faire sans aucune extension algébrique), de trouver ses racines (ce qui peut nécessiter des extensions algébriques) et pour chacune de ses racines c_k, de déterminer $v_k = \mathrm{pgcd}(q - c_k(r' - N\eta' r), r)$. Comme dans le cas des fonctions logarithmiques, il se peut que ce polynôme ait des racines non constantes, ce qui implique que f n'a pas d'intégrale élémentaire.

4.2.6 Intégration des fonctions mixtes

Il est fort possible d'avoir des fonctions qui ne soient pas purement logarithmiques ou purement exponentielles. Mais les hypothèses sur K que nous avons énoncées nous permettent d'intégrer les fonctions mixtes, en regardant la fonction comme membre de $K(\theta)$, où K est un corps de fonctions et θ est un logarithme ou une exponentielle. Par exemple, considérons la fonction

$$\frac{-e^x \log^2 x + \log x \left(\dfrac{2(e^x+1)}{x}\right) + e^x + e^{2x}}{1 + 2e^x + e^{2x}}.$$

Le problème. Cette fonction appartient au corps de fonctions $\mathbf{Q}(x, e^x, \log x)$. Donc nous pouvons écrire $K = \mathbf{Q}(x, e^x)$, $\theta = \log x$ et appliquer la théorie de la section « Intégration de fonctions logarithmiques ». Comme élément de $K(\theta)$, cette fonction est un polynôme en θ :

$$\theta^2 \left(\frac{-e^x}{1 + 2e^x + e^{2x}}\right) + \theta \left(\frac{2}{x(1+e^x)}\right) + \frac{e^x}{1+e^x}.$$

D'après la méthode de la section « Intégration de fonctions logarithmiques – partie polynomiale », il faut intégrer le coefficient de la puissance principale de θ, c'est-à-dire $-e^x/(1+e^x)^2$. Cette intégration se déroule dans K.

Sous-problème 1. Cette intégration se déroule dans le corps $L(\phi)$, où $\phi = e^x$ et $L = \mathbf{Q}(x)$. La fonction à intégrer est une fonction rationnelle propre et de plus ϕ ne divise pas le dénominateur. Donc la théorie de la section « Intégration de fonctions exponentielles – la partie rationnelle et logarithmique » s'applique. La décomposition sans facteurs multiples est assez simple et nous devons appliquer la méthode d'Hermite à q/r^2 où $q = -\phi$ et $r = 1 + \phi$. Nous trouvons que $r' = \phi$ (le symbole $'$ désigne toujours la dérivation par rapport à x). Il faut appliquer l'identité de Bezout à r et r', ce qui est assez simple dans le cas actuel :

$$(1)r + (-1)r' = 1.$$

En substituant ces valeurs dans la méthode d'Hermite, nous trouvons une intégrale de $-q(-1)/(1+\phi)$ et un reste qui s'annule complètement. Ainsi la solution à ce sous-problème est $-\phi/(1+\phi)$.

Dans le problème original, ceci nous donne un terme de l'intégrale, c'est-à-dire $-\theta^2 e^x/(1+e^x)$. Mais la dérivée de ce terme nous donne aussi des termes en θ^1. Le coefficient de θ^1 à intégrer est, selon la formule (5) de la section « Partie polynomiale », donné par $A_1 - 2\theta' B_2$, où A_1 est le coefficient original et B_2 est la solution du sous-problème 1. Ce calcul nous donne

$$\frac{2}{x(1+e^x)} - \frac{2}{x}\frac{-e^x}{1+e^x} = \frac{2}{x}.$$

L'intégrale de cette fonction doit nous donner le coefficient de θ dans l'intégrale (plus, éventuellement, une détermination de la constante d'intégration dans le dernier sous-problème).

Sous-problème 2. Cette intégration se déroule en principe dans le corps $L(\phi)$, où $\phi = e^x$ et $L = \mathbf{Q}(x)$, mais en réalité il est assez simple de voir que la réponse est $2 \log x$.

En général, les intégrales données par les sous-problèmes doivent appartenir à K, mais il est permis, comme nous l'avons vu, qu'elles contiennent θ. C'est le cas actuel, car l'intégrale est 2θ. Ceci implique que le choix de la constante d'intégration dans le dernier sous-problème était mauvais et qu'il faut l'augmenter par $2/2 = 1$. Donc, l'état actuel de ce problème est que nous avons intégré les coefficients de θ^2 et de θ^1 et nous avons trouvé comme intégrale $\theta^2(1 - e^x/(1 + e^x))$, ce qui se simplifie en $\theta^2/(1 + e^x)$. Il reste à intégrer le coefficient de θ^0, soit $e^x/(1 + e^x)$.

Sous-problème 3. Cette intégration se déroule dans le corps $L(\phi)$, où $\phi = e^x$ et $L = \mathbf{Q}(x)$. La fonction à intégrer n'est pas une fonction rationnelle propre et doit être réécrite sous la forme $1 - 1/(1 + e^x)$. L'intégration de 1 nous donne x (plus une constante d'intégration qui est la constante d'intégration du problème). L'autre partie est une fonction rationnelle propre q/r, où $q = -1$ et $r = 1 + e^x$. Comme r n'est pas divisible par e^x et ne possède pas de facteurs multiples, son intégrale doit être une somme de logarithmes. Selon la théorie de la section « Intégration de fonctions exponentielles – partie rationnelle et logarithmique », il faut calculer

$$\mathrm{Res}_\phi(q - y(r' - N\eta' r), r),$$

où $N = 1$ (le degré de r) et $\eta = x$. Ceci se simplifie en $\mathrm{Res}_\phi(-1 + y, 1 + \phi)$, c'est-à-dire $-1 + y$. Ce polynôme a une racine, $y = 1$, qui est bien une constante. Donc l'intégrale de la partie rationnelle est $\log(1 + e^x) - x$. Le $-x$ s'annule avec le x de l'autre partie et nous avons $\log(1 + e^x)$.

Ainsi la solution du problème est

$$\log^2 x \left(\frac{1}{1 + e^x}\right) + \log(1 + e^x).$$

Cette méthode de décomposition d'une intégrale, apparemment assez compliquée, comme une série de problèmes imbriqués, mais simples, est parfaitement générale. Elle donne lieu au résultat suivant.

Théorème [Risch, 1969]. Soit $K = C(x, \theta_1, \theta_2, \ldots, \theta_n)$ un corps de fonctions, C étant un corps de constantes et chaque θ_i étant un logarithme ou une exponentielle d'un élément de $C(x, \theta_1, \theta_2, \ldots, \theta_{i-1})$ et transcendant sur $C(x, \theta_1, \theta_2, \ldots, \theta_{i-1})$. En outre, le corps de constantes de K doit être C. Alors il existe un algorithme qui, étant donné un élément f de K, ou bien donne une fonction élémentaire sur K qui est l'intégrale de f, ou bien démontre que f n'a aucune intégrale élémentaire sur K.

La démonstration se fait par récurrence sur n ($n = 0$ étant l'intégration de fonctions rationnelles), utilisant la théorie des sections « Intégration de fonctions logarithmiques » et « Intégration de fonctions exponentielles ». Ce théorème s'applique aussi aux fonctions trigonométriques (et leurs inverses), car elles peuvent être exprimées en termes d'exponentielles (ou de logarithmes).

4.2.7 Intégration de fonctions algébriques

Ceci est un problème assez difficile qui a suscité l'intérêt de plusieurs grands mathématiciens et qui est l'objet de la « Géométrie algébrique ». À la suite de plusieurs découvertes dans ce sujet, on a pu donner [Davenport, 1981] une résolution de ce problème, c'est-à-dire un algorithme qui, étant donnée une fonction f algébrique sur $\mathbf{Q}(x)$, ou bien donne une fonction élémentaire sur $\mathbf{Q}(x)$ qui est son intégrale, ou bien démontre qu'une telle fonction n'existe pas.

Cet algorithme est assez compliqué et Davenport ne l'a programmé que partiellement, dans le cas où f est définie par des racines carrées et où les problèmes de géométrie ne sont pas trop compliqués[1]. Plus récemment, Trager [1985] a donné d'autres méthodes qui semblent être plus efficaces et qui sont maintenant programmées, surtout dans Axiom. En fait, ce problème reste à la frontière de nos connaissances mathématiques et de la puissance de nos systèmes de Calcul Formel.

1. En un sens compliqué, mais précis : il faut que la fonction algébrique soit définie sur une courbe algébrique de genre au plus un s'il existe des parties logarithmiques à calculer.

Il est possible d'utiliser cet algorithme, plutôt que l'intégration de fonctions rationnelles, comme point de départ de la récurrence dans le théorème de Risch. Ceci nous donne le résultat suivant.
Théorème [Davenport, 1984a]. Soit $K = C(x, y, \theta_1, \theta_2, \ldots, \theta_n)$ un corps de fonctions, C étant un corps de constantes, y étant algébrique sur $C(x)$ et chaque θ_i étant un logarithme ou une exponentielle d'un élément de $C(x, y, \theta_1, \theta_2, \ldots, \theta_{i-1})$ et transcendant sur $C(x, y, \theta_1, \theta_2, \ldots, \theta_{i-1})$. En outre, le corps de constantes de K doit être C. Alors il existe un algorithme qui, étant donné un élément f de K, ou bien donne une fonction élémentaire sur K qui est l'intégrale de f, ou bien démontre que f n'a aucune intégrale élémentaire sur K.

Il faut noter que ce théorème ne permet pas les extensions algébriques qui dépendent des fonctions logarithmiques ou exponentielles. Ceci est un problème qui a maintenant été résolu par Bronstein [1990b] et largement implanté en Axiom. Mais cette extension aux fonctions mixtes implique une théorie et une implémentation de beaucoup d'Algèbre Géométrique et il est très difficile de suivre tous les cas.

4.2.8 Intégration de fonctions non élémentaires

Nous avons cité la définition du mot « élémentaire » due à Liouville et nous avons esquissé une théorie de l'intégration de ces fonctions dans laquelle les intégrales doivent être élémentaires également. Que se passe-t-il si nous levons cette restriction ? Tout d'abord, nous ne pouvons pas nous appuyer sur le principe de Liouville. Il a été généralisé par Singer, Saunders et Caviness [1981 ; 1985], mais cette généralisation est assez compliquée et ne s'applique qu'à une classe de fonctions assez restreinte, bien que cette base soit plus large que les fonctions élémentaires. En fait, on permet aussi, outre les logarithmes et les exponentielles, un nombre fini de fonctions de type E ou L.
Définition. Une fonction est de type E sur un corps K si elle est de la forme $\int u' G(\exp(R(u)))$, où $u \in K$ et G et R sont des fonctions rationnelles à coefficients constants. Une fonction est de type L sur un corps K si elle est de la forme $\int u' H(\log(S(u)))$, où $u \in K$ et H et S sont des fonctions rationnelles à coefficients constants, telles que le degré du numérateur de H est au plus le degré du dénominateur de H plus un.
Par exemple, la fonction $\operatorname{erf} x$ est de type E sur $\mathbf{C}(x)$, avec $u = x$ et les fonctions rationnelles $G(y) = \sqrt{2/\pi}y$ et $R(y) = -y^2$. De même,

$$\operatorname{erf} \log x = \int \frac{\sqrt{2/\pi}}{x} \exp(-\log^2 x)$$

est de type E sur $\mathbf{C}(x, \log x)$. La fonction $\operatorname{Ei}(x) = \int e^x/x$ n'est pas de type E sur $\mathbf{C}(x)$, car la fonction $G(y)$ qui serait nécessaire, soit y/x, n'est pas à coefficients constants. La fonction $\operatorname{Li}(x) = \int 1/\log x$ est de type L sur $\mathbf{C}(x)$, avec $u = x$, $H(y) = 1/y$ et $S(y) = y$. La fonction $\operatorname{Ei}(x)$ est de type L sur $\mathbf{C}(x, e^x)$, car elle peut être écrite sous la forme $\operatorname{Li}(e^x)$.

Il faut plus que le principe de Liouville pour faire de l'intégration en calcul formel – il faut aussi un algorithme. Il existe maintenant des algorithmes qui généralisent le théorème de Risch en ne permettant pas uniquement la même classe de fonctions, mais une classe plus grande d'intégrales, c'est-à-dire les fonctions « élémentaires avec erf » ou « élémentaires avec Li (et Ei) » [Cherry, 1983 ; 1985 ; Cherry et Caviness, 1984].

4.3 Solutions formelles des ÉDO linéaires

Le problème d'intégration peut être vu comme la résolution de l'équation différentielle la plus simple : $y' = f$. Dans cette section, nous donnerons quelques indications sur le comportement des solutions formelles de certaines équations différentielles linéaires. Les remarques que nous avons faites dans la dernière section sur la différence entre les solutions formelles et les solutions numériques s'appliquent ici, ainsi que les remarques sur la différence entre les approches heuristiques et algorithmiques.

4.3.1 Les équations du premier ordre

Ici nous considérons l'équation $y' + fy = g$. La sous-section précédente sur l'intégration des fonctions exponentielles a déjà introduit le problème de trouver un algorithme qui, étant données f et g appartenant à une classe A de fonctions, ou bien trouve une fonction y appartenant à une classe donnée B de fonctions, ou bien démontre qu'il n'existe aucun élément de B satisfaisant à l'équation donnée. Pour simplifier, nous considérons le cas où B est toujours la classe des fonctions élémentaires sur A.

Pour ces équations, il existe une méthode de résolution assez bien connue : on substitue $y = ze^{-\int f}$. Ceci nous donne

$$\begin{aligned} g &= y' + fy \\ &= z'e^{-\int f} - zfe^{-\int f} + fze^{-\int f} \\ &= z'e^{-\int f}. \end{aligned}$$

Ainsi $z' = ge^{\int f}$ et

$$y = e^{-\int f} \int ge^{\int f}. \tag{1}$$

En général, cette méthode ne suffit pas, algorithmiquement, pour trouver y, car l'algorithme d'intégration décrit dans la dernière section ramène cette intégrale à l'équation différentielle de départ.

Le problème de Risch

Il est donc nécessaire de trouver une méthode directe de résolution de ces équations. Risch [1969] en a trouvé une pour les cas où A est un corps de fonctions rationnelles ou une extension (par un logarithme transcendant ou une exponentielle transcendante) d'un corps sur laquelle ce problème est résoluble. Nous donnons ici l'algorithme pour le cas des fonctions rationnelles : pour les autres cas les principes sont semblables mais les détails[2] sont beaucoup plus compliqués. La résolution donnée ici est, *grosso modo*, celle de Risch [1969] : il en existe une autre, due à Davenport [1985].

Le problème peut être posé de la manière suivante : étant données deux fonctions rationnelles f et g, trouver la fonction rationnelle y telle que $y' + fy = g$ ou démontrer qu'il n'en existe pas. La fonction f satisfait à la condition que $\exp(\int f)$ n'est pas une fonction rationnelle, c'est-à-dire que f n'est pas constante et son intégrale n'est pas une somme de logarithmes à coefficients rationnels. Nous procéderons en deux étapes : réduction à un problème purement polynomial et résolution de ce problème.

Soit p un polynôme irréductible. Soit α le plus grand entier tel que p^α divise le dénominateur de y, que nous pouvons noter par $p^\alpha \parallel \mathrm{den}(y)$. Soient β et γ tels que $p^\beta \parallel \mathrm{den}(f)$ et $p^\gamma \parallel \mathrm{den}(g)$. Ainsi nous pouvons calculer les puissances de p divisant les termes de l'équation à résoudre :

$$\underbrace{y'}_{\alpha+1} + \underbrace{fy}_{\alpha+\beta} = \underbrace{g}_{\gamma}.$$

Il existe trois possibilités.

i. $\underline{\beta > 1}$. Dans ce cas, il faut que les termes en $p^{\alpha+\beta}$ et p^γ s'annulent, c'est-à-dire que $\alpha = \gamma - \beta$.

2. Les détails sont dus à Risch [1969] pour les extensions logarithmiques et exponentielles, à Davenport [1985] pour les extensions algébriques de $C(x)$ et à Bronstein [1990a] pour les autres cas de fonctions mixtes.

ii. $\underline{\beta < 1}$ (autrement dit, $\beta = 0$). Dans ce cas, il faut que les termes en $p^{\alpha+1}$ et p^γ s'annulent, c'est-à-dire que $\alpha = \gamma - 1$.

iii. $\underline{\beta = 1}$. Dans ce cas, il est possible que les termes à gauche s'annulent et que la puissance de p qui divise le dénominateur de $y' + fy$ soit inférieure à $\alpha + 1$. Si cette annulation n'a pas lieu, le résultat est bien $\alpha = \gamma - 1 = \gamma - \beta$. Supposons, donc, qu'il existe une annulation. Décomposons f et y en éléments simples par rapport à p : $f = F/p^\beta + \hat{f}$ et $y = Y/p^\alpha + \hat{y}$, où les puissances de p divisant les dénominateurs de $\hat{f}$ et $\hat{y}$ sont inférieures à $\beta - 1 = 0$ et $\alpha - 1$.

$$y' + fy = \frac{-\alpha p' Y}{p^{\alpha+1}} + \frac{Y'}{p^\alpha} + \frac{FY}{p^{\alpha+1}} + \frac{\hat{f}Y}{p^\alpha} + \frac{F\hat{y}}{p} + \hat{f}\hat{y}.$$

Pour qu'il existe une annulation dans cette équation, il faut que p divise $-\alpha p'Y + FY$. Mais p est irréductible et Y est de degré inférieur à celui de p, donc p et Y sont premiers entre eux. Ceci implique que p divise $\alpha p' + F$. Mais p' et F sont de degrés inférieurs à celui de p et le seul polynôme de degré inférieur à celui de p et divisible par p est zéro. Donc $\alpha = F/p'$.

Nous avons démontré le résultat suivant :
Lemme [Risch, 1969]. $\alpha \leq \max(\min(\gamma - 1, \gamma - \beta), F/p')$, où le dernier terme n'est valable que si $\beta = 1$ et que s'il donne lieu à un entier positif.

En fait, il n'est pas nécessaire de faire une factorisation des dénominateurs en polynômes irréductibles. Il suffit de trouver les polynômes p_i sans facteurs multiples et deux à deux premiers entre eux et des entiers non négatifs β_i et γ_i tels que $\mathrm{den}(f) = \prod p_i^{\beta_i}$ et $\mathrm{den}(g) = \prod p_i^{\gamma_i}$. Quand $\beta = 1$, il est, en principe, nécessaire de faire une factorisation complète de p, mais il suffit de trouver les racines entières de $\mathrm{Res}_x(F - yp', p)$, par un raisonnement semblable à l'algorithme de Trager pour le calcul de la partie logarithmique de l'intégrale d'une fonction rationnelle.

Nous avons donc pu borner le dénominateur de y par $D = \prod p_i^{\alpha_i}$, de sorte que $y = Y/D$ avec Y polynomial. Il est donc possible de supprimer les dénominateurs dans notre équation et de trouver l'équation $RY' + SY = T$. Soient α, β, γ et δ les degrés de Y, R, S et T. Nous avons trois possibilités[3].

i. $\underline{\beta - 1 > \gamma}$. Dans ce cas, il faut que les termes de degré $\alpha + \beta - 1$ annulent les termes de degré δ, donc $\alpha = \delta + 1 - \beta$.

ii. $\underline{\beta - 1 < \gamma}$. Dans ce cas, il faut que les termes de degré $\alpha + \gamma$ annulent les termes de degré δ, donc $\alpha = \delta - \gamma$.

iii. $\underline{\beta - 1 = \gamma}$. Dans ce cas, il se peut que les termes de degré $\alpha + \beta - 1$ à gauche s'annulent. Sinon, l'analyse précédente reste valable et $\alpha = \delta + 1 - \beta$. Pour analyser l'annulation, écrivons $Y = \sum_{i=0}^{\alpha} y_i x^i$, $R = \sum_{i=0}^{\beta} r_i x^i$ et $S = \sum_{i=0}^{\gamma} s_i x^i$. Les coefficients des termes de degré $\alpha + \beta - 1$ sont $\alpha r_\beta y_\alpha$ et $s_\gamma y_\alpha$. L'annulation est équivalente à $\alpha = -s_\gamma / r_\beta$.

Nous avons démontré le résultat suivant :
Lemme [Risch, 1969]. $\alpha \leq \max(\min(\delta - \gamma, \delta + 1 - \beta), -s_\gamma/r_\beta)$, où le dernier terme n'est valable que si $\beta = \gamma + 1$ et que s'il donne lieu à un entier positif.

La détermination des coefficients y_i de Y est un problème d'algèbre linéaire. En fait, le système d'équations est triangulaire et se résout facilement.

3. Le lecteur peut remarquer que cette analyse est très proche de l'analyse du dénominateur. Cette similarité n'est pas pur hasard – en fait l'excès du degré de Y sur celui de D est la multiplicité de $\hat{x}$ dans le dénominateur de y, après avoir fait la transformation $\hat{x} = 1/x$. Ce point est analysé par Davenport [1984a ; 1984b].

Existence de solutions

Néanmoins, la transformation de l'équation différentielle en l'équation (1) donne lieu à un résultat intéressant.

Théorème [Davenport, 1985]. Soit A une classe de fonctions, contenant f et g et supposons que l'équation $y' + fy = g$ ait une solution élémentaire sur A. Alors :

ou bien $e^{\int f}$ est algébrique sur A et dans ce cas la théorie de l'intégration doit pouvoir déterminer y ;

ou bien y appartient à A.

Démonstration. Si $e^{\int f}$ n'est pas algébrique sur A, alors cette fonction est transcendante sur A. Posons $B = A(\int f)$. Ou bien $e^{\int f}$ est algébrique sur B, ou bien elle est transcendante sur B. La seule façon pour que $e^{\int f}$ soit algébrique sur B est que $\int f$ soit une somme de logarithmes à coefficients rationnels. Mais dans ce cas, $e^{\int f}$ est algébrique sur A.

La seule possibilité qui reste à considérer est que $e^{\int f}$ soit transcendante sur B. Nous avons supposé que y est élémentaire sur A et, *a fortiori*, que y est transcendante sur B. Or le quotient de deux fonctions élémentaires est une fonction élémentaire et ainsi

$$ye^{\int f} = \int ge^{\int f}$$

est élémentaire sur B. Mais le lemme de décomposition des intégrales de fonctions exponentielles implique que cette intégrale est de la forme $he^{\int f}$, où h appartient à B. Alors $y \in B$ et y est la solution d'un problème de Risch : en fait le problème de Risch de l'équation originale $y' + fy = g$. Mais f et g appartiennent à A et la solution du problème de Risch appartient donc à la même classe.

En effet, ce théorème dit qu'il n'existe que deux méthodes pour calculer une solution élémentaire d'une équation différentielle du premier ordre : ou bien on trouve une solution du problème homogène, ou bien on trouve une solution dans la même classe que celle des coefficients. De plus, il suffit de rechercher les solutions homogènes algébriques sur la classe définie par les coefficients.

4.3.2 Les équations du second ordre

Toute équation du premier ordre possède, comme nous l'avons vu, une solution qui peut être exprimée en terme d'intégrales et d'exponentielles et le problème intéressant pour ces équations est de déterminer si la solution est élémentaire ou non. Pour les équations du second ordre, il n'est plus évident (ou vrai) que toute solution puisse être exprimée en terme d'intégrales et d'exponentielles. Afin de pouvoir donner des théorèmes précis, il nous faut une définition exacte de ce concept.

Définition. Soit K un corps de fonctions. La fonction θ est *un générateur liouvillien* sur K si :

- θ est algébrique sur K, c'est-à-dire si θ satisfait à une équation polynomiale avec coefficients dans K ;
- θ est une *exponentielle* sur K, c'est-à-dire s'il existe un η dans K tel que $\theta' = \eta'\theta$, ce qui est une façon algébrique de dire que $\theta = \exp \eta$;
- θ est une *primitive* sur K, c'est-à-dire s'il existe un η dans K tel que $\theta' = \eta'$, ce qui est une façon algébrique de dire que $\theta = \int \eta$.

Définition. Soit K un corps de fonctions. Un surcorps $K(\theta_1, \ldots, \theta_n)$ de K s'appelle un *corps de fonctions liouvilliennes sur* K si chaque θ_i est un générateur liouvillien sur K. Une fonction est *liouvillienne sur* K si elle appartient à un corps de fonctions liouvilliennes sur K.
Si K est omis, on sous-entend $\mathbf{C}(x)$, c'est-à-dire le corps des fonctions rationnelles.

Cette définition est plus générale que la définition de « élémentaire », car tout logarithme est une primitive. Comme nous l'avons déjà remarqué, ceci regroupe aussi les fonctions trigonométriques. Nous utilisons la notation $K(\text{liou})$ pour la classe de fonctions liouvilliennes sur K.

Théorème [Kovacic 1977 ; 1986]. Il existe un algorithme qui, étant donnée une équation différentielle linéaire du second ordre, $y'' + ay' + b = 0$ avec a et b des fonctions rationnelles en x,

ou bien trouve deux solutions liouvilliennes telles que toute solution est une combinaison linéaire à coefficients constants de ces solutions,

ou bien démontre qu'il n'existe aucune solution liouvillienne (sauf zéro).

Ce théorème et l'algorithme engendré sont assez compliqués et nous ne pouvons pas donner tous les détails ici. On sait que la transformation $z = e^{\int a/2}y$ réduit cette équation à $z'' + (b - a^2/4 - a'/2)z = 0$. Kovacic a démontré que cette équation présente quatre résolutions possibles.

i. Il existe une solution de la forme $e^{\int f}$, où f est une fonction rationnelle. Dans ce cas, l'opérateur différentiel se factorise et l'on est ramené à une équation du premier ordre, dont les solutions sont toujours liouvilliennes.

ii. Le premier cas n'est pas vérifié, mais il existe une solution de la forme $e^{\int f}$, où f satisfait à une équation quadratique ayant pour coefficients des fonctions rationnelles. Dans ce cas, l'opérateur différentiel se factorise et l'on est ramené à une équation du premier ordre dont les solutions sont toujours liouvilliennes.

iii. Les deux premiers cas ne sont pas satisfaits, mais il existe une solution non nulle liouvillienne. Dans ce cas, toute solution est une fonction algébrique.

iv. Les solutions non nulles ne sont pas liouvilliennes.

Un exemple de l'algorithme de Kovacic est donné par l'équation

$$y'' = \frac{4x^6 - 8x^5 + 12x^4 + 4x^3 + 7x^2 - 5x + 1}{4x^4} y,$$

dont une solution est

$$\begin{aligned} y &= e^{\frac{-3}{2}\log x + \log(x^2-1) + \frac{1}{2}x^2 - 1 - 1/x} \\ &= x^{-3/2}(x^2-1)e^{x^2/2 - x - 1/x}. \end{aligned}$$

La méthode de Kovacic démontre également que l'équation de Bessel, qu'on peut écrire sous la forme

$$y'' = \left(\frac{4n^2 - 1}{4x^2} - 1\right) y$$

ne possède des solutions élémentaires que si $2n$ est un entier impair et trouve les solutions quand n satisfait à cette condition. Cet algorithme a été implanté en plusieurs systèmes et semble assez pratique [Saunders, 1981]. Il a aussi eu plusieurs extensions [Duval, 1991, Duval & Loday-Richaud, 1992], et quelques personnes ont amélioré l'algorithme [van der Put & Hendriks, 1993].

Il est possible d'utiliser les mêmes méthodes, dans le cas des équations d'ordre deux, que celles que nous avons utilisées pour les équations d'ordre un, afin de résoudre les équations avec second membre (voir Davenport [1985a]) mais en fait les résultats sont beaucoup plus généraux et seront traités dans la section suivante.

4.3.3 Les équations d'ordre général

Pour les équations d'ordre général, la situation est, théoriquement, aussi bien comprise que pour les équations d'ordre deux, même si les détails pratiques des algorithmes ne sont pas aussi clairs.

Les équations homogènes

Nous traitons d'abord les généralisations de l'algorithme de Kovacic.

Théorème [Singer, 1981]. Il existe un algorithme qui, étant donnée une équation différentielle linéaire d'ordre quelconque à coefficients des fonctions rationnelles ou algébriques :

ou bien trouve une solution liouvillienne ;

ou bien démontre qu'il n'y en a pas.

Si cet algorithme trouve une solution liouvillienne, l'opérateur différentiel (qu'on peut supposer être d'ordre n) se factorise en un opérateur d'ordre un, dont la solution trouvée est une solution, et un opérateur d'ordre $n-1$, dont les coefficients sont également des fonctions rationnelles ou algébriques. Nous pouvons appliquer l'algorithme récursivement à ce dernier, afin de trouver toutes les solutions liouvilliennes.

Cet algorithme est plus général que celui de Kovacic pour les équations d'ordre deux, non seulement à cause de la généralisation des ordres, mais aussi parce qu'il permet les fonctions algébriques comme coefficients. Cependant, personne n'a jamais implanté cet algorithme qui semble assez lourd et compliqué. Singer et Ulmer [1992, 1993a, 1993b, 1993c] ont trouvé un algorithme, semblable à celui de Kovacic, pour les équations d'ordre trois.

Singer [1985] a également trouvé un algorithme semblable au précédent, mais qui cherche les solutions exprimable en termes de solutions d'équations d'ordre deux et de fonctions liouvilliennes. Ici encore, l'algorithme semble assez lourd.

Les équations inhomogènes

Ici nous considérons les généralisations du théorème de Davenport cité pour les équations d'ordre un. Il y a essentiellement trois possibilités :

i. il existe une solution élémentaire ;

ii. il existe une solution liouvillienne, mais qui n'est pas élémentaire ;

iii. il n'existe aucune solution liouvillienne.

Le théorème suivant nous permet de distinguer entre le premier cas et les deux autres, en cherchant des solutions algébriques de l'équation homogène.

Théorème 1 [Singer et Davenport, 1985 ; Davenport et Singer, 1986]. Soit A une classe de fonctions contenant les coefficients d'un opérateur différentiel linéaire L, soit g un élément de A et supposons que l'équation $L(y) = g$ ait une solution élémentaire sur A. Alors :

ou bien $L(y) = 0$ a une solution algébrique sur A,

ou bien y appartient à A.

Ce théorème est un corollaire du résultat suivant qui nous permet de distinguer entre les deux premiers cas et le dernier, en cherchant les solutions liouvilliennes (d'un type spécial) sur A.

Théorème 2 [Singer et Davenport, 1985 ; Davenport et Singer, 1986]. Soit A une classe de fonctions contenant les coefficients d'un opérateur différentiel linéaire L, soit g un élément de A et supposons que l'équation $L(y) = g$ ait une solution liouvillienne sur A. Alors :

cas 1) ou bien $L(y) = 0$ a une solution $e^{\int z}$ avec z algébrique sur A,

cas 2) ou bien y appartient à A.

En outre, il existe un algorithme pour le cas où A est un corps de fonctions algébriques qui décide du cas dans le théorème 2 et qui réduit le problème du théorème 1 à un problème de géométrie algébrique. Cet algorithme s'appuie sur l'algorithme de Singer (ou Kovacic, le cas échéant) pour les équations homogènes.

4.4 Conclusion

Nous avons vu beaucoup d'avancées théoriques (par exemple le principe de Liouville) et algorithmiques dans le calcul de primitives et des solutions formelles des ÉDO linéaires[4]. Le grand jeu des essais heuristiques peut être remplacé par une théorie systématique et des algorithmes assez puissants et, pour la plupart, bien implémentés dans les gros systèmes de calcul formel. Mais avons-nous résolu le problème? La réponse dépend un peu de ce qu'on comprend par « le problème ». Si c'est le problème algébrique, la réponse doit être « largement » – bien sûr, il existe beaucoup de choses à faire, beaucoup de thèses à écrire, et beaucoup d'algorithmes à implémenter.

Mais si l'on retourne à l'analyse d'où nous avons déduit ces idées abstraites de dérivation, intégration etc., la réponse est moins claire. Rien ne dit que nos fonctions sont continues, bien que, en étant définies en termes des opérations arithmétiques et les fonctions log et exp, elles soient continues par morceaux. Jeffrey [1993] en particulier soulève le problème de trouver une expression pour une primitive qui a le minimum de discontinuités. Par exemple, il cite

$$\int \frac{3}{5-4\cos x}dx = \begin{cases} 2\arctan(3\tan x/2) & \text{Maple, Mathematica} \\ 2\arctan\left(\frac{3\sin x}{\cos x+1}\right) & \text{Macsyma} \\ -\arctan\left(\frac{-3\sin x}{5\cos x-4}\right) & \text{Axiom.} \end{cases}$$

Aucune de ces fonctions n'est continue partout, bien qu'on ait commencé avec l'intégrale d'une fonction bornée.

Ainsi l'ordinateur peut-il rendre l'algèbre moins fastidieuse, mais, pour le moment, ne permet pas à l'étudiant d'oublier l'analyse.

4. Un certain progrès a aussi été fait dans le domaine des ÉDO non linéaires, mais ce sujet est assez différent du sujet linéaire pour que nous ne le traitions pas ici.

4.5 Bibliographie

[Bronstein, 1990a] Bronstein, M., The transcendental Risch differential equation. J. Symbolic Comp. **9** (1990) pp. 49–60.

[Bronstein, 1990b] Bronstein, M., Integration of Elementary Functions. J. Symbolic Comp., 9(1990) pp. 117–173.

[Cherry, 1983] Cherry, G.W., Algorithms for Integrating Elementary Functions in Terms of Logarithmic Integrals and Error Functions. Ph.D. Thesis, Univ. Delaware, August 1983.

[Cherry, 1985] Cherry, G.W., Integration in Finite Terms with Special Functions: the Error Function. J. Symbolic Comp. **1** (1985) pp. 283–302.

[Cherry & Caviness, 1984] Cherry, G.W. & Caviness, B.F., Integration in Finite Terms with Special Functions: A Progress Report. Proc. EUROSAM 84 [Springer Lecture Notes in Computer Science 174, Springer-Verlag, Berlin-Heidelberg-New York-Tokyo, 1984] pp. 351–359.

[Davenport, 1981] Davenport, J.H., On the Integration of Algebraic Functions. Springer Lecture Notes in Computer Science 102, Springer-Verlag, Berlin-Heidelberg-New York, 1981. Zbl. 471.14009. MR 84k:14024.

[Davenport, 1984a] Davenport, J.H., Intégration Algorithmique des fonctions élémentairement transcendantes sur une courbe algébrique. Annales de l'Institut Fourier **34** (1984) pp. 271–276. Zbl. 506.34002.

[Davenport, 1984b] Davenport, J.H., $y' + fy = g$. Proc. EUROSAM 84 [Springer Lecture Notes in Computer Science 174, Springer-Verlag, Berlin-Heidelberg-New York-Tokyo, 1984] pp. 341–350.

[Davenport, 1985] Davenport, J.H., On the Risch Differential Equation Problem. SIAM J. Comp. **15** (1986) pp. 903–918.

[Davenport & Singer, 1986] Davenport, J.H. & Singer, M.F. Elementary and Liouvillian Solutions of Linear Differential Equations. J. Symbolic Comp. **2** (1986) pp. 237–260.

[Duval, 1991] Duval, A., The Kovacic Algorithm with Applications to Special Functions. Differential Equations and Computer Algebra (ed. M. Singer) Academic Press, 1991, pp. 113–130.

[Duval & Loday-Richaud, 1992] Duval, A. & Loday-Richaud,M., Kovacic's algorithm and its application to some families of special functions. Applicable Algebra in Engineering, Communication and Computing **3** (1992) pp. 211–246.

[Hermite, 1872] Hermite, C., Sur l'intégration des fractions rationelles. Nouvelles Annales de Mathématiques, 2 Sér., **11** (1872) pp. 145–148. Ann. Scientifiques de l'École Normale Supérieure, 2 Sér., **1** (1872) pp. 215–218.

[Horowitz, 1969] Horowitz, E., Algorithm for Symbolic Integration of Rational Functions. Ph.D. Thesis, Univ. of Wisconsin, November 1969.

[Horowitz, 1971] Horowitz, E., Algorithms for Partial Fraction Decomposition and Rational Function Integration. Proc. Second Symposium on Symbolic and Algebraic Manipulation, ACM Inc., 1971, pp. 441–457.

[Jeffrey,1993] Jeffrey, D.J. Integration to obtain expressions valid on domains of maximum extent. Proc. ISSAC 1993 (ed. M. Bronstein, ACM, 1993) pp. 34–41.

[Kahrimanian, 1953] Kahrimanian, H.G., Analytic differentiation by a digital computer. M.A. Thesis, Temple U., Philadelphia, Pennsylvania, May 1953.

[Kovacic, 1977] Kovacic, J.J., An Algorithm for solving Second Order Linear Homogenous Differential Equations. Preprint, Brooklyn College, City University of New York.

[Kovacic, 1986] Kovacic, J.J., An Algorithm for solving Second Order Linear Homogenous Differential Equations. J. Symbolic Comp. **2** (1986) pp. 3–43.

[Liouville, 1835] Liouville, J., Mémoire sur l'intégration d'une classe de fonctions transcendantes. Crelle's J. **13** (1835) pp. 93-118.

[Moses, 1967] Moses, J., Symbolic Integration. Ph.D. Thesis & Project MAC TR–47, M.I.T., 1967.

[Moses, 1971] Moses, J., Symbolic Integration, the stormy decade. Comm. ACM **14** (1971) pp. 548–560.

[Nolan, 1953] Nolan, J., Analytic differentiation on a digital computer. M.A. Thesis, Math. Dept., M.I.T., Cambridge, Massachusetts, May 1953.

[Ostrowski, 1946] Ostrowski, A.M., Sur l'intégrabilité élémentaire de quelques classes d'expressions. Comm. Math. Helvet. **18** (1946) pp. 283–308.

[van der Put & Hendriks, 1993] van der Put, M. & Hendriks, P.A., A rationality result for Kovacic's algorithm. Proc. ISSAC 1993 (ed. M. Bronstein, ACM, 1993) pp. 4–8.

[Richardson, 1968] Richardson, D., Some Unsolvable Problems Involving Elementary Functions of a Real Variable. J. Symbolic Logic **33** (1968), pp. 511–520.

[Risch, 1969] Risch, R.H., The Problem of Integration in Finite Terms. Trans. A.M.S. **139** (1969) pp. 167–189. Zbl. 184,67. MR **38** (1969) #5759.

[Risch, 1979] Risch, R.H., Algebraic Properties of the Elementary Functions of Analysis. Amer. J. Math. **101** (1979) pp. 743–759. MR 81b:12029.

[Rothstein, 1976] Rothstein, M., Aspects of Symbolic Integration and Simplification of Exponential and Primitive Functions. Ph.D. Thesis, Univ. of Wisconsin, Madison, 1976. (Xerox University Microfilms 77–8809).

[Saunders, 1981] Saunders, B.D., An Implementation of Kovacic's Algorithm for Solving Second Order Linear Homogenous Differential Equations. Proceedings of the 1981 ACM Symposium on Symbolic and Algebraic Computation, ACM Inc., New York, 1981, pp. 105–108. Zbl. 486.68023.

[Singer, 1981] Singer, M.F., Liouvillian Solutions of n-th Order Homogeneous Linear Differential Equations. Amer. J. Math. **103** (1981) pp. 661–682. Zbl. 477.12016. MR 82i:12008.

[Singer, 1985] Singer, M.F., Solving Homogeneous Linear Differential Equations in Terms of Second Order Linear Differential Equations. Amer. J. Math. **107** (1985) pp. 663–696.

[Singer & Davenport, 1985] Singer, M.F. & Davenport,J.H., Elementary and Liouvillian Solutions of Linear Differential Equations. Proc. EUROCAL 85, Vol. 2 [Springer Lecture Notes in Computer Science Vol. 204, Springer-Verlag, 1985] pp. 595–596.

[Singer & Ulmer, 1992] Singer, M.F. & Ulmer, F., Liouvillian Solutions of Third Order Linear Differential Equations: New Bounds and Necessary Conditions. Proc. ISSAC 92 (ed. P.S. Wang) pp. 57-62.

[Singer & Ulmer,1993a] Singer, M.F. & Ulmer, F., Galois Groups of Second and Third Order Linear Differential Equations. J. Symbolic Comp. **16** (1993) pp. 9–36.

[Singer & Ulmer,1993b] Singer, M.F. & Ulmer, F., Liouvillian and Algebraic Solutions of Second and Third Order Linear Differential Equations. J. Symbolic Comp. **16** (1993) pp. 37–74.

[Singer & Ulmer,1993c] Singer, M.F. & Ulmer, F., On a Third Order Differential Equation whose Differential Galois Group is a Simple Group with 168 Elements. Proc. 10th AAECC, Springer Lecture Notes in Computer Science **673**, pp. 316–324.

[Singer *et al.*, 1981] Singer, M.F., Saunders, B.D. & Caviness, B.F., An Extension of Liouville's Theorem on Integration in Finite Terms. Proceedings of the 1981 ACM Symposium on Symbolic and Algebraic Computation, ACM Inc., New York, 1981., pp. 23–24. Zbl. 482.12008.

[Singer *et al.*, 1985] Singer, M.F., Saunders, B.D. & Caviness, B.F., An Extension of Liouville's Theorem on Integration in Finite Terms. SIAM J. Comp. **14** (1985) pp. 966–990.

[Slagle, 1961] Slagle, J., A Heuristic Program that Solves Symbolic Integration Problems in Freshman Calculus. Ph.D. Dissertation, Harvard U., Cambridge, Mass. May 1961.

[Trager, 1976] Trager, B.M., Algebraic Factoring and Rational Function Integration. Proceedings of the 1976 ACM Symposium on Symbolic and Algebraic Computation, ACM Inc., New York, 1976, pp. 219–226. Zbl. 498.12005.

[Trager, 1985] Trager, B.M., On the Integration of Algebraic Functions. Ph.D. Thesis, Dept. of Electrical Engineering & Computer Science, M.I.T., August 1985.

Chapitre 5

Bases de Gröbner

Nathalie REVOL

Les bases de Gröbner sont un outil d'étude et de résolution des systèmes algébriques. Elles permettent en particulier de caractériser de façon simple l'appartenance à un idéal de polynômes. Cet outil peut être utilisé, entre autres, lors de la preuve de circuits logiques, dans le cadre de la preuve automatique de théorèmes ou pour l'élimination de quantificateurs en logique. De façon très simplifiée, la base de Gröbner d'un idéal engendré par un ensemble de polynômes est un autre ensemble de polynômes, mis sous forme normalisée, qui engendre le même idéal et qui vérifie certaines propriétés. Il s'agit d'une généralisation, dans le cas de polynômes à plusieurs indéterminées, de la notion de pgcd.

Nous commencerons par rappeler, dans une première partie, les notions utilisées, telles que celle d'idéal dans un anneau de polynômes à plusieurs variables. Nous introduirons également une notion de division dans un tel anneau, la division d'Hironaka. Ceci nous permettra de définir et de caractériser une base de Gröbner, de façon non constructive.

Bien que ces notions aient été établies dès Hilbert, il a fallu attendre Buchberger, en 1965, pour disposer d'un algorithme de calcul. La version initiale de cet algorithme est simple dans son principe, cependant le nombre de calculs à effectuer s'avère rapidement rédhibitoire. De nombreux travaux de recherche portent actuellement sur la réduction de ce nombre de calculs, une grande part se révélant *a fortiori* inutile. Cet algorithme originel ainsi qu'un aperçu des variantes ultérieures feront l'objet de la deuxième partie.

Une troisième et dernière partie illustrera certaines utilisations de ces bases de Gröbner. Un premier intérêt est la recherche des solutions d'un système algébrique. La démonstration de deux théorèmes de géométrie montrera comment les bases de Gröbner interviennent en preuve automatique de théorèmes. Enfin, quelques exemples d'école montreront leur utilisation dans le domaine de la logique.

Ce chapitre ne contient pas d'exercice ; cependant toutes les démonstrations et les exemples peuvent être traités en exercice.

5.1 Polynômes à plusieurs indéterminées

Cette première partie contient les définitions des objets mathématiques manipulés ainsi que leurs propriétés importantes.

5.1.1 L'anneau des polynômes à plusieurs indéterminées sur un corps ou un anneau

Polynômes à plusieurs indéterminées

Ce paragraphe est fortement inspiré de [52, 13, 54]. Aucune démonstration n'est donnée ; les démonstrations sont en général faciles à refaire ou se trouvent dans l'un des trois ouvrages précédemment cités.

Soit $(A, +, *)$ un anneau commutatif unitaire (1 est l'unité) et soit $\{X_1, \ldots, X_n\}$ un ensemble fini.

Définition 7 : polynôme en les indéterminées $X_1, \ldots, X_n$
On appelle polynôme en les indéterminées $X_1, \ldots, X_n$ *et à coefficients dans* A *une somme formelle*

$$\sum_{i=(i_1,\ldots,i_n)\in \mathbb{N}^n} a_{i_1,\ldots,i_n} X_1^{i_1} \ldots X_n^{i_n}$$

que l'on note aussi $\sum_{i\in\mathbb{N}^n} a_i X^i$*, où la famille* $(a_i)_{i\in\mathbb{N}^n}$ *est presque nulle.*

Définition 8 : monôme
Un polynôme de la forme $aX^i = aX_1^{i_1} \ldots X_n^{i_n}$ *est appelé un monôme. Il s'agit d'un polynôme dont un seul coefficient est non nul.*

Définition 9 : $A[X_1, \ldots, X_n]$
L'ensemble des polynômes en les indéterminées $X_1, \ldots, X_n$ *est noté* $A[X_1, \ldots, X_n]$.

Soient $P = \sum_{i\in\mathbb{N}^n} a_i X^i \in A[X_1, \ldots, X_n]$, $Q = \sum_{j\in\mathbb{N}^n} b_j X^j \in A[X_1, \ldots, X_n]$. On peut définir sur $A[X_1, \ldots, X_n]$ une addition :

$$P + Q = \sum_{i\in\mathbb{N}^n} (a_i + b_i) X^i,$$

une multiplication par un scalaire $\lambda \in A$:

$$\lambda P = \sum_{i\in\mathbb{N}^n} (\lambda * a_i) X^i$$

et une multiplication interne :

$$P * Q = \sum_{k\in\mathbb{N}^n} \left(\sum_{i+j=k} (a_i * b_j) \right) X^k,$$

où $\forall i = (i_1, \ldots, i_n) \in \mathbb{N}^n$, $\forall j = (j_1, \ldots, j_n) \in \mathbb{N}^n$, $i + j = (i_1 + j_1, \ldots, i_n + j_n)$.

Muni de ces opérations, $A[X_1, \ldots, X_n]$ est une A-algèbre commutative admettant le polynôme 1 comme unité.

Remarque
$A[X_1, \ldots, X_n]$ muni de l'addition et de la multiplication par un scalaire est un A-module, et un A-espace vectoriel si A est un corps.

$\forall m \,/\, 1 \leq m \leq n$, $(A[X_1, \ldots, X_m])[X_{m+1}, \ldots, X_n]$ et $A[X_1, \ldots, X_m, X_{m+1}, \ldots, X_n]$ sont naturellement isomorphes et seront confondus.

Définition 10 : support d'un polynôme
Le support d'un polynôme $P = \sum_{i \in \mathbb{N}^n} a_i X^i \in A[X_1, \ldots, X_n]$, noté $Supp(P)$, est le sous-ensemble de $\mathbb{N}^n$ défini par

$$Supp(P) = \{i = (i_1, \ldots, i_n) \in \mathbb{N}^n / a_i = a_{i_1,\ldots,i_n} \neq 0\}.$$

Par exemple, le support du polynôme $X^4Y^2 + 5X^2Y^5 - X^7 + Y + 3 \in \boldsymbol{Z}[X,Y]$ est $\{(4;2), (2;5), (7;0), (0;1), (0;0)\}$.

Un monôme est donc un polynôme dont le support est un singleton.

Idéaux

Ce paragraphe et le suivant doivent encore beaucoup à M. Lejeune-Jalabert [52]. Nous allons définir différents types d'idéaux : idéal, idéal engendré et idéal premier.

Soit I une partie de A.

Définition 11 : idéal
I est un idéal de A ssi

- *I est un sous-groupe additif de A ;*
- *$\forall a \in A$, $\forall x \in I$, $a * x \in I$, autrement dit I est absorbant pour la multiplication*[1].

Exemple :

- A, $\{0\}$ sont des idéaux de A ;
- dans $\boldsymbol{Z}$, les $n\boldsymbol{Z}$ (qui sont les seuls sous-groupes additifs) sont les seuls idéaux.

Propriété 1
L'intersection d'une famille d'idéaux de A est un idéal de A.

Définition 12 - Théorème : idéal engendré
Soit X une partie de A. Il existe un plus petit (au sens de l'inclusion) idéal de A contenant X, c'est l'intersection de tous les idéaux de A contenant X.
On l'appelle idéal engendré par X.

Définition 13 - Théorème : idéal principal
Soit $a \in A$. L'idéal engendré par $\{a\}$, noté (a), est aA.
On l'appelle idéal principal engendré par a.

Théorème 4
Soit (I_k) une famille d'idéaux de A. J l'ensemble des sommes $\sum_k x_k$, où (x_k) est une famille presque nulle d'éléments de A vérifiant $\forall k$, $x_k \in I_k$, est l'idéal engendré par $\bigcup_k I_k$.
J est appelé somme des idéaux I_k et est noté $\sum_k I_k$.

1. La structure algébrique sous-jacente, A, étant supposée commutative, il n'y a pas lieu de distinguer idéal à droite et idéal à gauche. Tout idéal est donc bilatère.

Corollaire 1
Soit (a_k) une famille d'éléments de A. L'idéal engendré par la famille (a_k), i.e. *l'idéal engendré par la partie $\bigcup_k \{a_k\}$, est l'ensemble des sommes $\sum_k a_k u_k$, où (u_k) est une famille presque nulle d'éléments de A.*
Pour être homogène avec les notations des deux théorèmes précédents, cet idéal est noté $\sum_k a_k A$.

Définition 14 : anneau principal
Un anneau commutatif unitaire intègre A est principal ssi tout idéal de A est principal.

Définition 15 - Théorème: anneau quotient
Soit I un idéal de A, soit A/I l'ensemble quotient de A par la relation d'équivalence $x - y \in I$. Cette relation d'équivalence est compatible avec l'addition et la multiplication, et, munie des lois-quotients, A/I est un anneau, appelé anneau-quotient de A par I.

Exemple :

- On a déjà vu que les idéaux de $\boldsymbol{Z}$ sont les $n\boldsymbol{Z}$. $\boldsymbol{Z}$ est donc un idéal principal.
- Quel que soit K un corps commutatif, $K[X]$ est un anneau principal: pour tout idéal I de l'anneau $K[X]$, il existe un polynôme P (unique à un facteur multiplicatif appartenant à K près) tel que $I = (P)$.
  En particulier, si I est engendré par deux polynômes P et Q, $I = (pgcd(P, Q))$, ce qui peut être une définition du pgcd. On peut construire ce pgcd grâce à l'algorithme d'Euclide.
  Les bases de Gröbner permettent de généraliser cette notion de pgcd, comme « générateur simple » en un certain sens, au cas de l'anneau $K[X_1, \ldots, X_n]$.

Une notion importante est celle d'anneau nœthérien: l'anneau des polynômes à plusieurs variables n'est pas principal, cependant tout idéal admet une famille finie de générateurs.

Définition 16 : anneau nœthérien
Un anneau commutatif unitaire A est nœthérien ssi tout idéal de A peut être engendré par un nombre fini d'éléments.

Proposition 7
Soit A un anneau. Les trois assertions suivantes sont équivalentes :

i. A est nœthérien ;

ii. toute suite, croissante pour l'inclusion, d'idéaux de A :
$(I_n)_{n \in I\!N}$, $I_0 \subset I_1 \subset \ldots \subset I_n \subset I_{n+1} \subset \ldots$ est stationnaire à partir d'un certain rang :
$\exists r \in I\!N \;/\; \forall m \in I\!N$, $I_r = I_{r+m}$;

iii. tout ensemble non vide d'idéaux de A contient un élément maximal pour l'inclusion.

Théorème 5 : théorème de la base de Hilbert
Si A est un anneau nœthérien, alors $A[X]$ est un anneau nœthérien.

Corollaire 2

i. si A est un anneau nœthérien, alors tout anneau de polynômes $A[X_1, \ldots, X_n]$ l'est aussi ;

ii. pour tout corps K, $K[X_1, \ldots, X_n]$ est un anneau nœthérien.

Ordre dans $\mathbb{N}^n$

On travaille désormais dans $K[X_1, \ldots, X_n]$ avec K un corps commutatif. Afin de pouvoir définir une division dans $K[X_1, \ldots, X_n]$, il faut pouvoir ordonner les monômes, ou, ce qui revient au même, se doter d'un ordre total dans $\mathbb{N}^n$ l'ensemble des exposants.

Dans $K[X]$, l'ordre est induit par l'ordre sur $\mathbb{N}$:

$$X^i < X^j \text{ si et seulement si } i < j.$$

Dans $K[X_1, \ldots, X_n]$, l'ordre sur les monômes sera induit par un ordre sur les exposants - qui appartiennent à $\mathbb{N}^n$ - auquel on imposera certaines restrictions :
on suppose que l'ordre total choisi sur $\mathbb{N}^n$ vérifie

i. $\forall \alpha \in \mathbb{N}^n,\ \forall \beta \in \mathbb{N}^n,\ \text{si } \beta \neq 0\ (0 = (0, \ldots, 0)),\ \text{alors } \alpha < \alpha + \beta\,;$

ii. $\forall \alpha \in \mathbb{N}^n,\ \forall \beta \in \mathbb{N}^n,\ \forall \gamma \in \mathbb{N}^n,\ \alpha < \beta \Leftrightarrow \alpha + \gamma < \beta + \gamma.$

On notera $\alpha \leq \beta$ pour $\alpha < \beta$ ou $\alpha = \beta$.
La première condition peut être remplacée par $\forall \alpha \in \mathbb{N}^n,\ 0 = (0, \ldots, 0) \leq \alpha$.

Les ordres les plus fréquemment utilisés sont les suivants.
Ordre lexicographique $\leq_L$
$\forall \alpha = (\alpha_1, \ldots, \alpha_n) \in \mathbb{N}^n,\ \forall \beta = (\beta_1, \ldots, \beta_n) \in \mathbb{N}^n,$
$\alpha <_L \beta$ ssi $\exists j \in \{1, \ldots, n\}$ / $\forall i < j,\ \alpha_i = \beta_i$ et $\alpha_j < \beta_j$.
De façon générale, soit $L = \sum_{i=1}^n c_i x_i$ avec $c_i \in \mathbb{R}^+$ pour tout i appartenant à $\{1, \ldots, n\}$ une forme linéaire sur $\mathbb{R}^n$, on définit un ordre grâce à L par

$$\alpha \in \mathbb{N}^n,\ \beta \in \mathbb{N}^n,\ \alpha \leq \beta \text{ ssi } \left(L(\alpha) < L(\beta) \text{ dans } \mathbb{R}\right) \text{ ou } \left(L(\alpha) = L(\beta) \text{ et } \alpha \leq_L \beta\right).$$

En particulier, si $L = 0$ on retrouve l'ordre lexicographique.
Si $L = \sum_{i=1}^n x_i$, l'ordre obtenu, à savoir

$$\alpha \leq_D \beta \text{ ssi } \left(\sum_{i=1}^n \alpha_i < \sum_{i=1}^n \beta_i\right) \text{ ou } \left(\sum_{i=1}^n \alpha_i = \sum_{i=1}^n \beta_i \text{ et } \alpha \leq_L \beta\right)$$

est appelé **ordre diagonal**.
Enfin, l'ordre $\leq_I$ défini par

$$\alpha \leq_I \beta \text{ ssi } \left(\sum_{i=1}^n \alpha_i < \sum_{i=1}^n \beta_i\right) \text{ ou } \left(\sum_{i=1}^n \alpha_i = \sum_{i=1}^n \beta_i \text{ et } \exists j \in \{1, \ldots, n\} \,/\, \forall i > j,\ \alpha_i = \beta_i \text{ et } \alpha_j < \beta_j\right)$$

est appelé **ordre lexicographique inverse** ou plus souvent **ordre total**.

Exemple :

Dans $\mathbb{N}^2$, les couples inférieurs ou égaux à (5;3) sont :

– pour $\leq_L$:

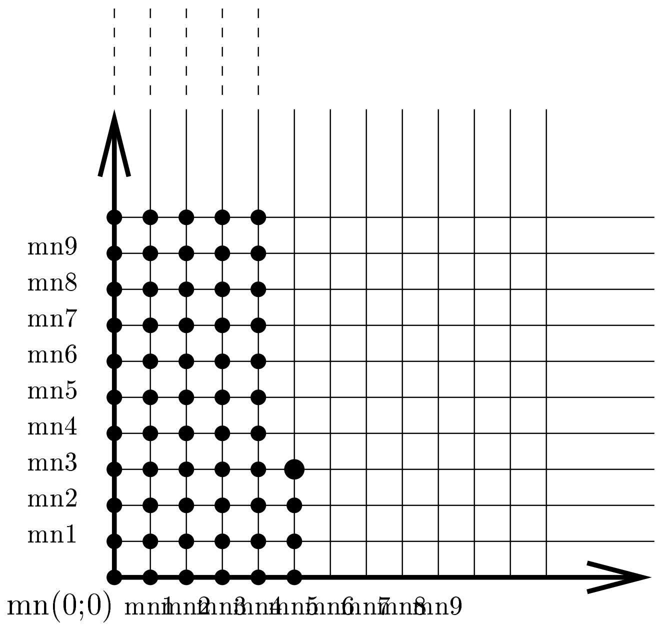


FIG. 5.1 – *Couples inférieurs à (5;3) pour l'ordre lexicographique.*

Pour cet ordre, il y a une infinité de couples inférieurs à un couple donné. Ceci est vrai pour tout $n \geq 2$.

– pour $\leq_D$ et $\leq_I$:

mnPour $\leq_D$:

mnPour $\leq_I$:

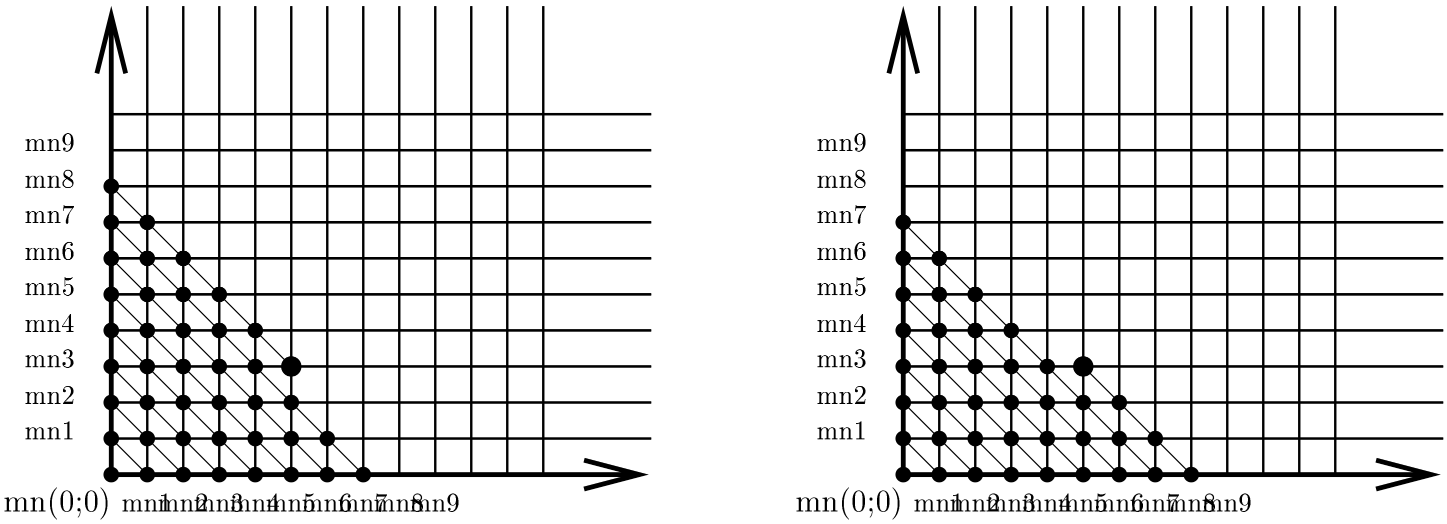


FIG. 5.2 – *Couples inférieurs à (5;3) pour l'ordre diagonal et l'ordre lexicographique inverse.*

Pour ces ordres en revanche, il n'y a qu'un nombre fini d'éléments de $\mathbb{N}^n$ inférieurs à un élément donné de $\mathbb{N}^n$.

La définition d'un ordre sur $\mathbb{N}^n$ nous permet de définir un ordre sur les monômes (on garde le même symbole pour dénoter la relation d'ordre) :

$$aX^i = aX_1^{i_1} \dots X_n^{i_n} \leq bX^j = bX_1^{j_1} \dots X_n^{j_n} \Leftrightarrow i = (i_1, \dots, i_n) \leq j = (j_1, \dots, j_n)$$

(a et b sont supposés tous les deux non nuls).

Exemple :

L'écriture du polynôme $P = X^4Y^2 + 5X^2Y^5 - X^7 + Y + 3 \ \in K[X,Y]$ en classant ses monômes dans l'ordre décroissant varie selon l'ordre choisi.

- Pour $\leq_L$, $P = -X^7 + X^4Y^2 + 5X^2Y^5 + Y + 3$.
- Pour $\leq_D$, $P = -X^7 + 5X^2Y^5 + X^4Y^2 + Y + 3$.
- Pour $\leq_I$, $P = 5X^2Y^5 - X^7 + X^4Y^2 + Y + 3$.

Définition 17 : exposant (privilégié) - partie initiale
Soit $P \in K[X_1, \dots, X_n]$, $P = \sum_{i \in Supp(P)} a_i X^i$.
L'exposant de P*, ou exposant privilégié de* P*, est le plus grand élément de* $Supp(P)$ **pour l'ordre choisi**. *On le note* $exp(P)$.
La partie initiale de P *(ou terme dominant* - leading term *ou* leading monomial *en anglais* - grobner[leadmon] *en Maple) est le monôme* $a_{exp(P)}X^{exp(P)}$*, on la note* $init(P)$.

On prend parfois pour convention $exp(0) = -\infty$.

Ces notions correspondent respectivement à celles de degré et de monôme de tête d'un polynôme à une seule indéterminée. Le terme de « degré » correspond à la somme $i_1 + \dots + i_n$ (ou $L((i_1, \dots, i_n))$) et le terme de « monôme de tête » est souvent utilisé même pour les polynômes à plusieurs indéterminées.

Exemple :

$P = X^4Y^2 + 5X^2Y^5 - X^7 + Y + 3 \ \in K[X,Y]$,

- pour $\leq_L$, $exp(P) = (7;0)$, $init(P) = -X^7$,
- pour $\leq_D$, $exp(P) = (7;0)$, $init(P) = -X^7$,
- pour $\leq_I$, $exp(P) = (2;5)$, $init(P) = 5X^2Y^5$.

Les restrictions imposées à l'ordre choisi sur $\mathbb{N}^n$ permettent d'assurer les propriétés suivantes :

i. $\forall P \in K[X_1, \dots, X_n]$, $\forall Q \in K[X_1, \dots, X_n]$, $exp(P + Q) \leq max(exp(P), exp(Q))$;

ii. $\forall P \in K[X_1, \dots, X_n]$, $\forall Q \in K[X_1, \dots, X_n]$, $exp(P * Q) = exp(P) + exp(Q)$ (c'est encore vrai si K est seulement un anneau intègre, mais sans la condition d'intégrité on a une inégalité : $exp(P * Q) \leq exp(P) + exp(Q)$), et $init(P * Q) = init(P) * init(Q)$).

On peut maintenant définir un ordre partiel sur $K[X_1, \dots, X_n]$:
$\forall P \in K[X_1, \dots, X_n]$, $\forall Q \in K[X_1, \dots, X_n]$,

$$P < Q \text{ ssi } X^{exp(P)} < X^{exp(Q)} \text{ ou si } X^{exp(P)} = X^{exp(Q)} \text{ et } P - init(P) < Q - init(Q).$$

5.1.2 Parties stables de $\mathbb{N}^n$, frontières, escaliers, bases de Gröbner

Dans cette partie, les bases de Gröbner seront définies de façon non constructive. Toutes les notions et propriétés seront établies sur $\mathbb{N}^n$, l'ensemble des exposants.

Dans ce paragraphe, n est un entier strictement positif.

Définition 18 : partie stable de $\mathbb{N}^n$
On appelle partie stable de $\mathbb{N}^n$ tout sous-ensemble non vide E de $\mathbb{N}^n$ tel que

$$\forall \alpha \in E,\ \forall \beta \in \mathbb{N}^n,\ \alpha + \beta \in E,$$

ce que l'on peut aussi écrire $E + \mathbb{N}^n \subset E$.

Définition 19 : frontière
On appelle frontière d'une partie stable E de $\mathbb{N}^n$ tout sous-ensemble ∂E de E tel que

$$\forall \alpha \in E,\ \exists \alpha_0 \in \partial E,\ \exists \beta \in \mathbb{N}^n \ /\ \alpha = \alpha_0 + \beta,$$

ce que l'on peut aussi écrire

$$E = \bigcup_{\alpha \in \partial E} \alpha + \mathbb{N}^n.$$

Exemple :

Les parties stables de $\mathbb{N}$ sont les ensembles de la forme $\alpha + \mathbb{N}$: il est facile de vérifier que $\alpha + \mathbb{N}$ est une partie stable. Réciproquement, soit E une partie stable de $\mathbb{N}$, E admet un minorant α, donc $E \subset \alpha + \mathbb{N}$. De plus (par définition d'une partie stable), $\forall \beta \in \mathbb{N}$, $\alpha + \beta \in E$, ie $\alpha + \mathbb{N} \subset E$, donc $E = \alpha + \mathbb{N}$.

Dans $\mathbb{N}$, toute partie stable (de la forme $\alpha + \mathbb{N}$ donc) admet une frontière de cardinal fini : tout sous-ensemble fini contenant α est clairement une telle frontière. Cette propriété est vraie pour tout $n \geq 1$ et fait l'objet de la proposition suivante.

Proposition 8
Toute partie stable E de $\mathbb{N}^n$ admet une frontière ∂E de cardinal fini (ou frontière finie par abus de langage).

Démonstration
Cette preuve s'effectue par récurrence sur n.
Le cas $n = 1$ est traité dans l'exemple précédent.
Supposons que la proposition soit vraie pour $n - 1$. Soit

$$\begin{array}{cccc} \pi : & \mathbb{N}^n & \to & \mathbb{N}^{n-1} \\ & (\alpha_1, \ldots, \alpha_{n-1}, \alpha_n) & \mapsto & (\alpha_1, \ldots, \alpha_{n-1}) \end{array}$$

la projection sur les $n-1$ premiers facteurs de $\mathbb{N}^n$. Soit E une partie stable de $\mathbb{N}^n$, $\pi(E)$ est une partie stable de $\mathbb{N}^{n-1}$: si $(\alpha_1, \ldots, \alpha_{n-1}) \in \pi(E)$ et $(\beta_1, \ldots, \beta_{n-1}) \in \mathbb{N}^{n-1}$, $\exists \alpha_n \in \mathbb{N} / (\alpha_1, \ldots, \alpha_{n-1}, \alpha_n) \in E$; on a alors $(\alpha_1 + \beta_1, \ldots, \alpha_{n-1} + \beta_{n-1}, \alpha_n + 0) \in E$ et $(\alpha_1 + \beta_1, \ldots, \alpha_{n-1} + \beta_{n-1}) \in \pi(E)$.

Par hypothèse de récurrence, il existe une frontière finie $\partial\pi(E)$ de $\pi(E)$. Il existe donc un sous-ensemble fini F de E tel que $\pi(F) = \partial\pi(E)$ est une frontière finie de $\pi(E)$.
Notons M le maximum des dernières composantes des éléments de F :

$$M = \max_{\alpha=(\alpha_1,\ldots,\alpha_{n-1},\alpha_n)\in F} \alpha_n.$$

Tout d'abord, pour tout $\beta = (\beta_1, \ldots, \beta_{n-1}) \in \pi(F)$, pour tout $\beta_n \geq M$, on sait que $\beta' = (\beta_1, \ldots, \beta_{n-1}, \beta_n) \in E$: $\beta = \pi(\bar\beta)$ avec $\bar\beta \in E$ et $\bar\beta_n \leq M$, donc $\beta' = \bar\beta + (0, \ldots, 0, M - \bar\beta_n) \in E$. Ceci signifie que pour tout $\beta = (\beta_1, \ldots, \beta_{n-1}) \in \pi(E)$ et pour tout $\beta_n \geq M$, $\beta' = (\beta_1, \ldots, \beta_{n-1}, \beta_n) \in E$.
On s'intéresse donc aux intersections successives de E avec les sous-ensembles de $I\!N^n$ de la forme $I\!N^{n-1} \times i$, $0 \leq i < M$. Pour tout $i \in \{0, \ldots, M-1\}$, on détermine $F_i \subset I\!N^n$ le sous-ensemble de cardinal fini de E tel que $\pi(F_i)$ est une frontière de $\pi(E \cap (I\!N^{n-1} \times i))$ - qui existe par hypothèse de récurrence. On obtient ∂E une frontière finie de E comme union finie d'ensembles de cardinaux finis :

$$\partial E = F \cup \bigcup_{0\leq i<M} F_i.$$

Vérifions que ∂E est une frontière de E : $\forall\alpha = (\alpha_1, \ldots, \alpha_{n-1}, \alpha_n) \in E$, $\exists\beta \in F$ tel que $\pi(\alpha) \in \pi(\beta) + I\!N^{n-1}$, donc $\alpha_1 - \beta_1 \geq 0, \ldots, \alpha_{n-1} - \beta_{n-1} \geq 0$.
Si $\alpha_n \geq M$, on a déjà vu que $\alpha \in \beta + I\!N^n$: $\alpha_n \geq M \geq \beta_n$ donc $\alpha_n - \beta_n \geq 0$.
Si $\alpha_n < M$, alors $\alpha \in E \cap (I\!N^{n-1} \times \alpha_n)$ et $\pi(\alpha) = (\alpha_1, \ldots, \alpha_{n-1}) \in \pi(E \cap (I\!N^{n-1} \times \alpha_n))$ donc $\exists\gamma \in F_{\alpha_n}$ / $(\alpha_1, \ldots, \alpha_{n-1}) \in \pi(\gamma) + I\!N^{n-1}$ et donc $\alpha = (\alpha_1, \ldots, \alpha_{n-1}, \alpha_n) \in \gamma + I\!N^n$. □

Définition 20 - Proposition : escalier
Toute partie stable E de $I\!N^n$ possède une unique frontière de cardinal (fini) minimal, qui est appelée son escalier.

DÉMONSTRATION
Soit q le minimum des cardinaux des frontières finies de E. Supposons que l'on ait deux frontières F et G de cardinal q, $F = (f_1, \ldots, f_q)$ et $G = (g_1, \ldots, g_q)$. Par définition, $E = \bigcup_{i=1}^q (f_i + I\!N^n) = \bigcup_{i=1}^q (g_i + I\!N^n)$.
Comme $f_i \in E$, $\exists\sigma(i)/f_i \in \left(g_{\sigma(i)} + I\!N^n\right)$. L'application

$$\begin{array}{cccc} \sigma : & \{1, \ldots, q\} & \to & \{1, \ldots, q\} \\ & i & \mapsto & \sigma(i) \end{array}$$

est surjective, car sinon E, qui est égal à $\bigcup_{i=1}^q (f_i + I\!N^n)$, serait inclus dans $\bigcup_{i=1}^q \left(g_{\sigma(i)} + I\!N^n\right)$. Ce dernier ensemble définirait donc une frontière de cardinal $< q$, ce qui contredirait la minimalité de q. On peut donc supposer (à une réindexation des g_i près) que $\sigma = I_d$. On a donc $f_i \in (g_i + I\!N^n)$. On a également $\exists\tau(i)/g_i \in \left(f_{\tau(i)} + I\!N^n\right)$ et donc $f_i + I\!N^n \subset f_{\tau(i)} + I\!N^n$. Si $i \neq \tau(i)$, alors $F \setminus \{f_i\}$ serait encore une frontière de E, de cardinal $q-1$, ce qui contredirait la minimalité de q. Finalement, $i = \tau(i)$ et $f_i = g_i$, ce qui prouve l'unicité de l'escalier. □

La dernière proposition servira à définir une division dans les anneaux de polynômes à plusieurs indéterminées (au §5.2). Elle correspond au fait que, comme $K[X_1, \ldots, X_n]$ est un anneau nœthérien, il n'existe pas de suite infinie strictement croissante d'idéaux (cf. proposition 7).

Proposition 9
Il n'existe pas de suite infinie strictement décroissante de $I\!N^n$. Autrement dit, toute suite décroissante est stationnaire à partir d'un certain rang.

DÉMONSTRATION
(par l'absurde).
Soit $(\alpha_p)_{p\in\mathbb{N}}$ une suite strictement décroissante d'éléments de $\mathbb{N}^n$. Soit $E = \bigcup_{p\in\mathbb{N}} (\alpha_p + \mathbb{N}^n)$, E est une partie stable de $\mathbb{N}^n$, donc E admet une frontière finie ∂E. $\exists \beta \in \partial E \;/\; \beta + \mathbb{N}^n$ contient une sous-suite infinie extraite de (α_p) : $(\alpha_{\sigma(p)})_{p\in\mathbb{N}}$. L'ordre vérifiant $\forall\alpha \in \mathbb{N}^n, \forall\beta \in \mathbb{N}^n, \alpha \leq \alpha+\beta$, alors $\beta \leq \alpha_{\sigma(p)}$, $\forall p \in \mathbb{N}$. D'autre part, $\beta \in E$, donc $\exists q \;/\; \beta \in (\alpha_q + \mathbb{N}^n)$. On a donc $\forall p \in \mathbb{N}$, $\alpha_q \leq \alpha_{\sigma(p)}$. On peut prendre p assez grand pour que $q < \sigma(p)$, mais alors comme la suite $(\alpha_p)_{p\in\mathbb{N}}$ est strictement décroissante, on aurait aussi $\alpha_q > \alpha_{\sigma(p)}$, ce qui est une contradiction. □

Remarque
En aucun cas cette proposition n'entraîne que $\forall\alpha \in \mathbb{N}^n$, $\{\beta \in \mathbb{N}^n \;/\; \beta < \alpha\}$ est un ensemble fini. Pour s'en convaincre, voir l'exemple 5.1.1 avec l'ordre lexicographique.

Bases de Gröbner

Si l'on considère maintenant un idéal de polynômes à plusieurs indéterminées, l'ensemble des exposants de ses éléments jouent le rôle de partie stable de $\mathbb{N}^n$. Transposer les notions de frontière et d'escalier conduit à définir les bases de Gröbner.

Soient K un corps, n un entier strictement positif et I un idéal de $K[X_1, \ldots, X_n] \neq (0)$. On note $exp(I) = \{exp(P),\ P \in I\}$. Cet ensemble est une partie stable de $\mathbb{N}^n$:
$\forall\alpha \in exp(I)$, $\exists P \in I \;/\; \alpha = exp(P)$, $\forall\beta \in \mathbb{N}^n$, $\alpha + \beta = exp(X^\beta P)$ et $X^\beta P \in I$ (puisque par définition un idéal est absorbant pour la multiplication), donc $\alpha + \beta \in \exp(I)$.

On appelle escalier de I et on note $E(I)$ l'escalier de $exp(I)$.

Définition 21 : base de Gröbner
Une base de Gröbner (ou base standard) de I est un ensemble de polynômes $\{f_1, \ldots, f_s\}$ de I tels que $\{exp(f_i),\ 1 \leq i \leq s\}$ est une frontière (finie) de $exp(I)$. Autrement dit, $exp(I) = \bigcup_{i=1}^s (exp(f_i) + \mathbb{N}^n)$.

Définition 22 : base de Gröbner minimale
On appelle base de Gröbner minimale une base de Gröbner de cardinal minimal, i.e. *telle que $\{exp(f_i),\ 1 \leq i \leq s\} = E(I)$.*

Une base de Gröbner est un ensemble fini de polynômes et non pas une suite (l'ordre n'a pas d'importance).

5.2 Division et pseudo-division d'Hironaka

Nous verrons à la fin de cette section qu'une base de Gröbner d'un idéal est un système simple de générateurs. Cette notion correspond, dans le cas des polynômes à une seule indéterminée, à celle de pgcd des générateurs de l'idéal. Tout comme le calcul d'un pgcd requiert une division dans $K[X]$, le calcul d'une base de Gröbner nécessite une division dans $K[X_1, \ldots, X_n]$: la division d'Hironaka.

5.2.1 Division d'Hironaka

Définition 23 : partition
Soit $F = (f_1, \dots, f_s)$ une famille finie de polynômes de $K[X_1, \dots, X_n]$. On associe à F une partition de $\mathbb{N}^n$: $\Delta_1, \dots, \Delta_s, \bar{\Delta}$ définie par :

$$\begin{array}{rl} & \Delta_1 = exp(f_1) + \mathbb{N}^n, \\ & \Delta_2 = (exp(f_2) + \mathbb{N}^n) \setminus \Delta_1, \\ & \vdots \\ & \Delta_i = (exp(f_i) + \mathbb{N}^n) \setminus \left(\bigcup_{j=1}^{i-1} \Delta_j \right), \quad 1 \leq i \leq s. \\ \textit{On note} & \Delta = \bigcup_{i=1}^{s} \Delta_i \\ & \bar{\Delta} = \mathbb{N}^n \setminus \Delta. \end{array}$$

Remarque

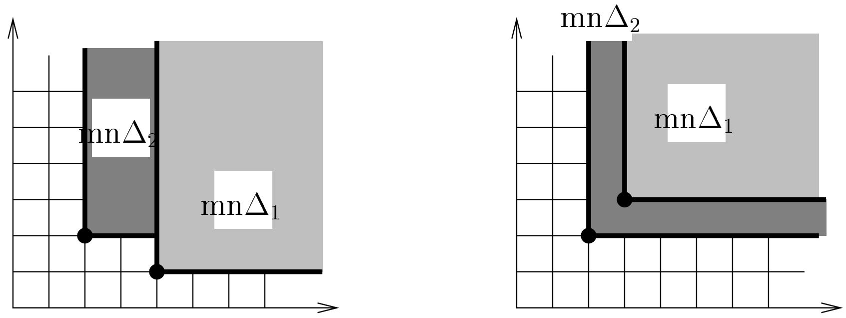
- Puisque la notion d'exposant dépend de l'ordre choisi sur $\mathbb{N}^n$, une partition dépend également de cet ordre ;
- les Δ_i dépendent de l'ordre dans lequel les polynômes sont considérés, cependant Δ, l'union des Δ_i, est indépendante de cet ordre.

Exemple :

Dans $K[X, Y]$,

$$\begin{array}{l} f_1 = X^3Y^3 + 2X^4Y + X^2 + 1 \\ f_2 = XY^3 + X^2Y^2 + Y^3 + 1 \\ f_3 = X^4Y^2 \end{array}$$

mnPour $\leq_L$ mnPour $\leq_D$



mn$exp(f_1) = (4; 1)$ $exp(f_2) = (2; 2)$ $exp(f_3) = (4; 2)$mn$exp(f_1) = (3; 3)$ $exp(f_2) = (2; 2)$ $exp(f_3) = (4; 2)$

mn$\Delta_3 = \emptyset$ mn$\Delta_3 = \emptyset$

FIG. 5.3 – *Partitions associées à $F = (f_1, f_2, f_3)$ pour les ordres $\leq_L$ et $\leq_D$.*

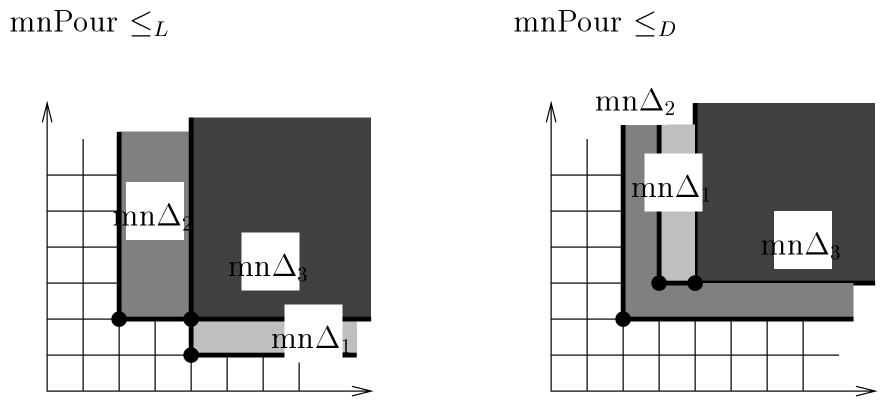


FIG. 5.4 – *Partitions associées à $G = (f_3, f_1, f_2)$ pour les ordres $\leq_L$ et $\leq_D$.*

Définition 24 - Théorème d'Hironaka
Soit $F = (f_1, \ldots, f_s) \in K[X_1, \ldots, X_n]$.
Soit Δ_F la partition de $I\!N^n$ associée à F. $\forall f \in K[X_1, \ldots, X_n]$, $\exists! H = (h_1, \ldots, h_s, h)$ avec $h_i \in K[X_1, \ldots, X_n], 1 \leq i \leq s$, $h \in K[X_1, \ldots, X_n]$ tel que

$$f = \sum_{i=1}^{s} f_i h_i + h \text{ avec } exp(f_i) + Supp(h_i) \subset \Delta_i \text{ et } Supp(h) \in \bar{\Delta} \text{ si } h \neq 0.$$

On appelle h le reste de la division de f par F et on le note $f\mathcal{R}F$. On dit aussi que f se réduit en h modulo F.

DÉMONSTRATION
L'existence sera démontrée par un algorithme de construction de H à la fin de ce paragraphe.
Pour montrer l'unicité, démontrons que deux solutions vérifiant la définition sont égales. Soient $H = (h_1, \ldots, h_s, h)$ et $H' = (h'_1, \ldots, h'_s, h')$ vérifiant la définition. On aurait donc

$$0 = \sum_{i=1}^{s} f_i(h_i - h'_i) + (h - h').$$

Montrons par absurde que $h - h' = 0$. Si $h - h' \neq 0$, alors comme $Supp(h) \subset \bar{\Delta}$ et $Supp(h') \subset \bar{\Delta}$, il en est de même pour $Supp(h - h')$: $Supp(h - h') \subset \bar{\Delta}$. En particulier $exp(h - h') \in \bar{\Delta}$. D'autre part, $h - h' = \sum_{i=1}^{s} f_i(h'_i - h_i)$ et donc $exp(h - h') = exp\left(\sum_{i=1}^{s} f_i(h'_i - h_i)\right) = \max_i\left(exp(h'_i - h_i)f_i\right)$, donc il existe un indice i_0 tel que $exp(h - h') = exp(h'_{i_0} - h_{i_0})f_{i_0} \in \Delta_{i_0}$, ce qui contredit le fait que $\Delta_{i_0} \cap \bar{\Delta} = \emptyset$, donc $h - h' = 0$.
On montre maintenant par absurde et de la même manière que $h_1 - h'_1 = 0$: si $h_1 - h'_1 \neq 0$, comme $(h_1 - h'_1)f_1 = \sum_{i=2}^{s} f_i(h'_i - h_i)$, il existerait $i_1 \in \{2, \ldots, s\}$ tel qu'on ait à la fois $exp\left((h_1 - h'_1)f_1\right) \in \Delta_1$ et $exp\left((h_1 - h'_1)f_1\right) \in \Delta_{i_1}$, ce qui contredit la définition de la partition $(\Delta_1, \ldots, \Delta_s, \bar{\Delta})$. De proche en proche, on montre ainsi que $h_i = h'_i$, $\forall i \in \{1, \ldots, s\}$. $\square$

Remarque

- H dépend de l'ordre des f_i. Par exemple, si $f_1 = X_1^3$, $f_2 = X_1 r X_2 - X_2^3$ et $f = X_1^3 X_2$, alors $f\mathcal{R}(f_1, f_2) = 0$ et $f\mathcal{R}(f_2, f_1) = X_1 X_2^3$. En effet, $f = X_2 f_1 + 0 f_2 + 0$ et $f = X_1 f_2 + 0 f_1 + X_1 X_2^3$.

– On a $exp(h) \leq exp(f)$ et $\forall i \in \{1, \ldots, s\}$, $exp(h_i) + exp(f_i) \leq exp(f)$.

5.2.2 Division d'Hironaka et base de Gröbner

Les bases de Gröbner permettent de rendre la division d'Hironaka indépendante de l'ordre sur les polynômes et même des polynômes choisis ; cette division ne dépend plus que de l'idéal engendré par les polynômes.

Définition 25 - Théorème : division d'un polynôme par un idéal
Le reste de la division de $f \in K[X_1, \ldots, X_n]$ par une famille de polynômes dont l'ensemble sous-jacent est une base de Gröbner d'un idéal I ne dépend ni de cette base de Gröbner ni de l'ordre choisi sur ses éléments.
On l'appelle reste de la division de f par I et on le note $f\mathcal{R}I$. Ce reste possède donc les propriétés suivantes :

i. *si $f\mathcal{R}I = \sum_\alpha c_\alpha X^\alpha$, alors $c_\alpha \neq 0 \Rightarrow \alpha \notin exp(I)$;*

ii. *$exp(f\mathcal{R}I) \leq exp(f)$;*

iii. *$(f\mathcal{R}I)\mathcal{R}I = f\mathcal{R}I$.*

Démonstration
Soient deux familles de polynômes $(f_1, \ldots, f_s)$ et $(f'_1, \ldots, f'_{s'})$ dont les ensembles sous-jacents sont des bases de Gröbner de I. Soient $\Delta_1, \ldots, \Delta_s, \bar{\Delta}$ et $\Delta'_1, \ldots, \Delta'_{s'}, \bar{\Delta}'$ les partitions de $I\!N^n$ associées à ces familles. On a

$$exp(I) = \bigcup_{i=1}^{s} (exp(f_i) + I\!N^n) = \bigcup_{i=1}^{s} \Delta_i = \bigcup_{i=1}^{s'} (exp(f'_i) + I\!N^n) = \bigcup_{i=1}^{s'} \Delta'_i.$$

Il s'ensuit que $\bar{\Delta} = \bar{\Delta}'$.
La division de f par $(f_1, \ldots, f_s)$ s'écrit

$$f = \sum_{i=1}^{s} h_i f_i + f\mathcal{R}(f_1, \ldots, f_s)$$

et la division de f par $(f'_1, \ldots, f'_{s'})$ s'écrit

$$f = \sum_{i=1}^{s'} h'_i f'_i + f\mathcal{R}(f'_1, \ldots, f'_{s'}).$$

Les restes de ces divisions vérifient :

$$\begin{array}{ll} \text{si} & f\mathcal{R}(f_1, \ldots, f_s) = \sum_\alpha c_\alpha X^\alpha, \text{ alors } c_\alpha \neq 0 \Rightarrow \alpha \in \bar{\Delta} \\ \text{si} & f\mathcal{R}(f'_1, \ldots, f'_{s'}) = \sum_\alpha c'_\alpha X^\alpha, \text{ alors } c'_\alpha \neq 0 \Rightarrow \alpha \in \bar{\Delta}' = \bar{\Delta}. \end{array}$$

Soit $g = f\mathcal{R}(f_1, \ldots, f_s) - f\mathcal{R}(f'_1, \ldots, f'_{s'})$, de la remarque ci-dessus on déduit que $exp(g) \in \bar{\Delta}$. Mais g peut aussi s'écrire $\sum_{i=1}^{s'} h'_i f'_i - \sum_{i=1}^{s} h_i f_i$, donc $g \in I$, donc $exp(g) \in exp(I) = I\!N^n \setminus \bar{\Delta}$. Il y a donc contradiction. Donc $g = 0$. □

L'un des intérêts des bases de Gröbner est de rendre le reste de la division indépendant de l'ensemble particulier de polynômes choisis, mais dépendant uniquement de l'idéal engendré par ces polynômes (puisque nous verrons bientôt qu'une base de Gröbner est un système de générateurs) ; un autre avantage est de permettre une caractérisation aisée de l'appartenance d'un polynôme à un idéal : un polynôme appartient à un idéal ssi le reste de sa division par l'idéal est nul.

Théorème 6
Pour tout polynôme $f \in K[X_1, \ldots, X_n]$,

$$f \in I \Leftrightarrow f\mathcal{R}I = 0.$$

DÉMONSTRATION
$\Rightarrow$: soit $\{f_1, \ldots, f_s\}$ une base standard de I et soit un ordre sur ses éléments, on a $f\mathcal{R}I = f\mathcal{R}\{f_1, \ldots, f_s\} = f - \sum_{i=1}^{s} f_i h_i$. Si $f \in I$, $f\mathcal{R}I \in I$. Si $f\mathcal{R}I \neq 0$, alors $exp(f\mathcal{R}I) \in exp(I)$, ce qui contredit l'une des propriétés énoncées dans le théorème précédent. Donc $f\mathcal{R}I = 0$.
$\Leftarrow$: si $f\mathcal{R}I = 0$, alors f peut s'écrire sous la forme $f = \sum_{i=1}^{s} h_i f_i$, donc $f \in I$. □

Ce théorème entraîne qu'une condition nécessaire et suffisante pour qu'un ensemble de polynômes $\{f_1, \ldots, f_s\}$ de I soit une base de Gröbner est que pour tout $f \in I$, f se réduise en 0 modulo $(f_1, \ldots f_s)$.

Corollaire 3
Les classes modulo I des $X^i, i \in \mathbb{N}^n \backslash exp(I)$ forment une base du K-espace vectoriel $K[X_1, \ldots, X_n]/I$. En particulier, pour que $K[X_1, \ldots, X_n]/I$ soit de dimension finie sur K, il faut et il suffit que $\mathbb{N}^n \setminus exp(I)$ soit fini et alors

$$dim_K K[X_1, \ldots, X_n]/I \leq \sharp\left(\mathbb{N}^n \setminus exp(I)\right).$$

De plus, le cardinal de $\mathbb{N}^n \setminus exp(I)$ est un majorant du nombre de « racines » communes à tous les polynômes appartenant à I ... mais comme trouver ces racines est l'une des applications des bases de Gröbner, ce résultat sera détaillé dans la partie 5.4 : « Applications ».

5.2.3 Algorithme de division

L'algorithme de division qui suit servira de démonstration constructive de l'existence des polynômes définis par le théorème d'Hironaka.

Cet algorithme prend en entrée un polynôme f et une liste de polynômes $F = (f_1, \ldots, f_s)$ et produit en sortie la liste $H = (h_1, \ldots, h_s, h)$.

On suppose que l'on dispose d'une fonction `decompose` qui prend pour arguments f et F et qui renvoie la liste $[f', f'']$ tels que $f = f' + f''$, $Supp(f') \subset \Delta$ et $Supp(f'') \subset \bar{\Delta}$ et d'une fonction `delta` qui prend pour arguments un monôme *monome* et F et qui renvoie l'indice i tel que $exp(monome) \in \Delta_i$.

L'algorithme de division d'Hironaka est le suivant :

Hironaka

Entrée
    f, $F = (f_1, \ldots f_s)$
Résultat
    $H = [h_1, \ldots, h_s, h]$.

Initialisation
    $h \leftarrow 0$
    $h_i \leftarrow 0,\ 1 \leq i \leq s$
Boucle
    tant que $f \neq 0$ faire
        décomposer f en f' et f''
        $h \leftarrow h + f''$
        si $f' \neq 0$ alors
            déterminer i tel que $exp(f') \in \Delta_i$
            $h_i \leftarrow h_i + \frac{init(f')}{init(f_i)}$
            $f \leftarrow f - \frac{init(f')}{init(f_i)} * f_i - f''$
        sinon
            $f \leftarrow f - f''$
        finsi
    fintantque
Fin.

Pour écrire ces fonctions en Maple, on a besoin d'une fonction qui rend le monôme de tête d'un polynôme. La fonction leadmon du package grobner convient ... presque ! En effet cette fonction, qui prend comme arguments un polynôme f, la liste de ses indéterminées et éventuellement un ordre (la valeur par défaut est tdeg, c'est-à-dire l'ordre lexicographique inverse, qui est désigné en Maple sous le nom d'ordre total), renvoie une liste constituée du coefficient $a_{exp(f)}$ et du monôme unitaire $X^{exp(f)}$. Je vous propose donc la fonction lm suivante :

```
# fonction qui donne le monome de tete
# grobner[leadmon] renvoie [coeff, x^exposant]

# Cette fonction reprend le squelette de la fonction grobner[leadmon],
# ce qui explique l'origine des tests et des messages d'erreur,
# mais sans faire les tests qui seront de toute facon effectues par
# leadmon (pour verifier que indet est bien une liste d'indeterminees
# et que poly est bien un polynome en les indeterminees indet)

lm := proc(poly, indet, order)

  local termorder;

  if nargs < 1 then
    ERROR(`too few arguments`)
  fi;
  if nargs = 3 then
    termorder := order
  elif nargs = 2 then
    indet := indets(poly);
    termorder := tdeg
  else
    ERROR(`too many arguments`);
  fi;

  RETURN(normal(convert(grobner[leadmon](poly,indet,termorder), `*`)))
end;
```

Écrire ces trois fonctions decompose, delta et Hironaka en Maple.

L'algorithme de division `Hironaka` se termine, sinon on construirait une suite de polynômes $(f^{(i)})$, formée par les valeurs successives de f au cours de l'algorithme, telle que la suite des exposants privilégiés de ces polynômes soit strictement décroissante. Or on sait qu'il n'existe pas de suite strictement décroissante dans $\mathbb{N}^n$.

Au passage, le nombre d'étapes est borné par le nombre d'éléments de $\mathbb{N}^n$ inférieurs à $exp(f)$; or il existe un nombre fini de tels éléments dans le cas de l'ordre diagonal ou de l'ordre lexicographique inverse, mais ce n'est pas le cas pour l'ordre lexicographique.

Exemple :

L'ordre choisi est l'ordre diagonal $\leq_D$.

$$\begin{array}{lcl} f_1 & = & XY^3 + X^2Y + 1 \\ f_2 & = & X^2Y + XY \\ f & = & X^2Y^3 + 2X^3Y + 3XY \end{array}$$

En guise d'exercice, dessinez la partition de $\mathbb{N}^2$ associée à $\{f_1, f_2\}$.
h, h_1 et h_2 sont initialisés à 0.
$1^{ième}$ **étape :**

$$\begin{array}{lcll} f' & = & X^2Y^3 + 2X^3Y & \qquad exp(f') \in \Delta_1, \quad init(f') = X^2Y^3 \\ f'' & = & 3XY & \\ h & \leftarrow & 3XY & \\ h_1 & \leftarrow & X & \\ f & \leftarrow & (X^2Y^3 + 2X^3Y + 3XY) - (X^2Y^3 + X^3Y + X) - 3XY & \\ & = & X^3Y - X & \end{array}$$

$2^{ième}$ **étape :**

$$\begin{array}{lcll} f' & = & X^3Y & \qquad exp(f') \in \Delta_2, \quad init(f') = X^3Y \\ f'' & = & -X & \\ h & \leftarrow & 3XY - X & \\ h_2 & \leftarrow & X & \\ f & \leftarrow & (X^3Y - X) - (X^3Y + X^2Y) - (-X) & \\ & = & -X^2Y & \end{array}$$

$3^{ième}$ **étape :**

$$\begin{array}{lcll} f' & = & -X^2Y & \qquad exp(f') \in \Delta_2, \quad init(f') = -X^2Y \\ f'' & = & 0 & \\ h & \leftarrow & h & \\ h_2 & \leftarrow & h_2 - 1 & \\ & = & X - 1 & \\ f & \leftarrow & (-X^2Y) + (X^2Y + XY) & \\ & = & XY & \end{array}$$

$4^{ième}$ **étape :**

$$\begin{array}{rcl} f' & = & 0 \\ f'' & = & XY \\ h & \leftarrow & h + XY \\ & = & 4XY - X \\ f & \leftarrow & XY - XY \\ & = & 0 \end{array}$$

Bilan :

$$\begin{array}{rcl} h_1 & = & X \\ h_2 & = & X - 1 \\ h & = & 4XY - X \\ f & = & Xf_1 + (X-1)f_2 + (4XY - X) \end{array}$$

5.2.4 Pseudo-division d'Hironaka

Dans le cas où l'ensemble des coefficients n'est pas un corps mais un anneau, il est possible d'établir un analogue de la division d'Hironaka. Le problème provient du fait qu'il n'est pas toujours possible de diviser les coefficients des monômes de tête de f' et f_i. Pour pallier à cet inconvénient, on multiplie les polynômes par ce qu'il faut pour que les coefficients de leurs nouveaux monômes de tête soient le ppcm des précédents.

La définition plus précise d'une pseudo-division d'un polynôme $f \in A[X_1, \ldots, X_n]$ par une famille $F = (f_1, \ldots f_s)$ de polynômes appartenant à $A[X_1, \ldots, X_n]$ est la suivante :

Définition 26 : pseudo-division d'Hironaka
Soit $F = (f_1, \ldots, f_s) \in A[X_1, \ldots, X_n]$. Soit Δ_F la partition de $\mathbb{N}^n$ associée à F. $\forall f \in A[X_1, \ldots, X_n], \exists H = (h_1, \ldots, h_s, h)$ avec $h_i \in K[X_1, \ldots, X_n], 1 \leq i \leq s$, $h \in K[X_1, \ldots, X_n]$ et $\exists \alpha \in A$ tels que

$$\alpha f = \sum_{i=1}^{s} f_i h_i + h \text{ avec } exp(f_i) + Supp(h_i) \subset \Delta_i \text{ et } Supp(h) \in \bar{\Delta} \text{ si } h \neq 0.$$

L'algorithme devient, en notant lcoeff(f_i) le coefficient de $init(f_i)$:

Pseudo-Hironaka

Entrée
    f, $F = (f_1, \ldots, f_s)$
Résultat
    $H = [h_1, \ldots, h_s, h]$.
Initialisation
    $h \leftarrow 0$
    $h_i \leftarrow 0,\ 1 \leq i \leq s$
Boucle
    tant que $f \neq 0$ faire
        décomposer f en f' et f''
        $h \leftarrow h + f''$
        si $f' \neq 0$ alors
            déterminer i tel que $exp(f') \in \Delta_i$
            $h_i \leftarrow \text{lcoeff}(f_i) * h_i + \frac{init(f')}{X^{exp(f_i)}}$
            $h_j \leftarrow \text{lcoeff}(f_i) * h_j,\ \forall j \neq i$
            $h \leftarrow \text{lcoeff}(f_i) * h$

$f \leftarrow \mathrm{lcoeff}(f_i) * f - \frac{init(f')}{X^{exp(f_i)}} * f_i - f''$

sinon

$f \leftarrow f - f''$

finsi

finsi

fintantque

Fin.

Cette pseudo-division permet de généraliser l'algorithme de Buchberger du §5.3.2 dans le cadre d'un calcul de base de Gröbner de polynômes à coefficients dans un anneau. Cette généralisation ne sera pas explicitée, il suffira de remplacer toutes les divisions de la même façon que dans cet algorithme.

5.3 Algorithme de Buchberger

5.3.1 S-polynôme ou syzygie

Dorénavant, le coefficient de tête d'un polynôme f, *i.e.* le coefficient de sa partie initiale, sera appelé *lcoeff*(f) (pour *leading coefficient*). C'est d'ailleurs le nom de la fonction Maple qui prend en entrée un polynôme, la liste de ses indéterminées et éventuellement un ordre, et qui renvoie ce coefficient. Le monôme unitaire $\frac{init(f)}{\mathrm{lcoeff}(f)}$ sera appelé *lterm*(f) (pour *leading term*). C'est encore le nom de la fonction Maple qui prend en entrée un polynôme, la liste de ses indéterminées et éventuellement un ordre, et qui renvoie ce monôme.

Définition 27 *: S-polynômes ou syzygies*
Soient f et $g \in K[X_1, \ldots, X_n]$, si $exp(f) = (\alpha_i)_{1 \leq i \leq n}$ et $exp(g) = (\beta_i)_{1 \leq i \leq n}$, soit[2] *$\gamma = \sup(exp(f), exp(g)) = (\max(\alpha_i, \beta_i))_{1 \leq i \leq n}$, le S-polynôme de f et g, noté fSg, est défini par :*

$$fSg = \mathrm{lcoeff}(g) * X^{\gamma - exp(f)} * f - \mathrm{lcoeff}(f) * X^{\gamma - exp(g)} * g.$$

Exemple :

Dans $\mathbb{Q}[X, Y]$ muni de l'ordre lexicographique, soient $f = 6X^3Y + 1$ et $g = 8XY^2 + 3XY + 2X$, $\gamma = \sup((3,1),(1,2)) = (3,2)$ et

$$fSg = 8 * Y * f - 6 * X^2 * g = 48X^3Y^2 + 8Y - 48X^3Y^2 - 18X^3Y - 12X^3 = -18X^3Y - 12X^3 + 8Y.$$

On note que ce nouveau polynôme est construit de façon à faire disparaître le monôme d'exposant γ qui est formé pendant cette combinaison linéaire, de sorte que l'exposant de fSg est strictement inférieur à γ.

Théorème 7 : caractérisation d'une base de Gröbner
Un système de générateurs $\{f_1, \ldots, f_s\}$ d'un idéal I est une base de Gröbner de I ssi, en choisissant un ordre sur les f_i (par exemple $(f_1, \ldots, f_s)$), pour tout couple (i, j) tel que $i < j$,

$$(f_i S f_j)\, \mathcal{R}(f_1, \ldots, f_s) = 0$$

(les f_i sont tous supposés non nuls).

2. γ n'est pas le maximum des deux éléments de $\mathbb{N}^n$, mais le nouvel élément de $\mathbb{N}^n$ construit en prenant le maximum composante par composante ; le maximum de deux éléments α et β de $\mathbb{N}^n$ sera noté $\max(\alpha, \beta)$.

Démonstration
C'est une condition nécessaire d'après la proposition 6 puisque $f_iSf_j \in I$.

Montrons qu'elle est suffisante : il s'agit de voir que pour tout $f \in I$, $f\mathcal{R}I = 0$.
Soit donc $f \in I$ et soit $g = f\mathcal{R}(f_1,\ldots,f_s)$. Supposons que g n'est pas nul. Comme f peut s'écrire $f = \sum_{i=1}^s f_ih_i + g$, on a $g = f - \sum_{i=1}^s f_ih_i$ et $g \in I$. Puisque $\{f_1,\ldots,f_s\}$ est un système de générateurs de I, $\exists H_1,\ldots,H_s \,/\, g = \sum_{i=1}^s f_iH_i$.
Soit $M = \max_{i/H_i\neq 0} exp(f_iH_i)$. Le squelette de la preuve va consister à trouver une nouvelle écriture de g, encore sous la forme $\sum_{i=1}^s f_iH_i''$, dans laquelle $M'' = \max_{i/H_i''\neq 0} exp(f_iH_i'')$ est strictement inférieur à M, ce qui signifierait que l'on peut construire une suite infinie strictement décroissante d'éléments de $\mathbb{N}^n$, ce qui contredit la proposition 9. Cette construction est un peu technique et peut être omise en première lecture.
Tout d'abord, il existe au moins deux indices j et k sur lesquels le maximum est atteint : $M = exp(f_jH_j) = exp(f_kH_k)$, sinon, si le seul indice était j, on aurait $exp(g) = exp(f_jH_j) = exp(f_j) + exp(H_j) \in (exp(f_j) + \mathbb{N}^n)$, ce qui contredit le fait que, par définition de la division, $exp(g) \in \mathbb{N}^n \setminus (\bigcup_{i=1}^s (exp(f_i) + \mathbb{N}^n))$.
Soient donc j et k tels que $M = exp(f_jH_j) = exp(f_kH_k)$. Notons $\alpha = (\alpha_1,\ldots,\alpha_n) = exp(f_j)$, $\beta = (\beta,\ldots,\beta_n) = exp(f_k)$, $u = (u_1,\ldots,u_n) = exp(H_j)$ et $v = (v_1,\ldots,v_n) = exp(H_k)$. Soit $\gamma = \sup(\alpha,\beta)$, il est facile de vérifier que $\forall i \in \{1,\ldots,n\}$, $u_i \geq \gamma_i - \alpha_i$ et $v_i \geq \gamma_i - \beta_i$ (car $\alpha + u = \beta + v = M$), et que $u_i - (\gamma_i - \alpha_i) = v_i - (\gamma_i - \beta_i)$. Notons $\delta_i = u_i - (\gamma_i - \alpha_i) = v_i - (\gamma_i - \beta_i)$ et $\delta = (\delta_1,\ldots,\delta_n)$ (on note $\delta = u + \alpha - \gamma = v + \beta - \gamma$), on a :

$$\begin{array}{rcl} X^\delta(f_jSf_k) & = & X^\delta\left(\mathrm{lcoeff}(f_k)X^{\gamma-\alpha}f_j - \mathrm{lcoeff}(f_j)X^{\gamma-\beta}f_k\right) \\ & = & \mathrm{lcoeff}(f_k)X^{u+\alpha-\gamma+\gamma-\alpha}f_j - \mathrm{lcoeff}(f_j)X^{v+\beta-\gamma+\gamma-\beta}f_k \\ & = & \mathrm{lcoeff}(f_k)X^uf_j - \mathrm{lcoeff}(f_j)X^vf_k \\ & = & \mathrm{lcoeff}(f_k)lterm(H_j)f_j - \mathrm{lcoeff}(f_j)lterm(H_k)f_k \end{array}$$

Réécrivons g :

$$\begin{array}{rcl} g & = & \sum_{i=1}^s f_iH_i \\ & = & init(H_j)f_j + init(H_k)f_k + (H_j - init(H_j))f_j + (H_k - init(H_k))f_k + \sum_{i\notin\{j,k\}} f_iH_i \end{array}$$

On remplace $init((H_j)f_j$ grâce à la formule donnée pour $X^\delta(f_jSf_k)$:

$$lterm(H_j)f_j = \frac{1}{\mathrm{lcoeff}(f_k)}X^\delta(f_jSf_k) + \frac{\mathrm{lcoeff}(f_j)}{\mathrm{lcoeff}(f_k)}lterm(H_k)f_k,$$

ce qui donne :

$$\begin{array}{rcl} g & = & \frac{\mathrm{lcoeff}(H_j)}{\mathrm{lcoeff}(f_k)}X^\delta(f_jSf_k) + \frac{\mathrm{lcoeff}(H_j)*\mathrm{lcoeff}(f_j)}{\mathrm{lcoeff}(f_k)}lterm(H_k)f_k + init(H_k)f_k \\ & & +(H_j - init(H_j))f_j + (H_k - init(H_k))f_k + \sum_{i\notin\{j,k\}} f_iH_i \\ & = & \frac{\mathrm{lcoeff}(H_j)}{\mathrm{lcoeff}(f_k)}X^\delta(f_jSf_k) + \left[1 + \frac{\mathrm{lcoeff}(H_j)*\mathrm{lcoeff}(f_j)}{\mathrm{lcoeff}(H_k)*\mathrm{lcoeff}(f_k)}\right] lterm(H_k)f_k \\ & & +(H_j - init(H_j))f_j + (H_k - init(H_k))f_k + \sum_{i\notin\{j,k\}} f_iH_i \end{array}$$

De cette dernière expression, on tire que

$$\begin{array}{lclcl} exp(X^\delta(f_jSf_k)) & = & \delta + exp(f_jSf_k) \quad < \quad \delta+\gamma & = & M \\ exp\left(\left[1 + \frac{\mathrm{lcoeff}(H_j)*\mathrm{lcoeff}(f_j)}{\mathrm{lcoeff}(H_k)*\mathrm{lcoeff}(f_k)}\right] lterm(H_k)f_k\right) & \leq & exp(lterm(H_k)f_k) & = & M \\ exp([H_j - init(H_j)]f_j) & < & exp(H_jf_j) & = & M \\ exp([H_k - init(H_k)]f_k) & < & exp(H_kf_k) & = & M \\ exp(sum_{i\notin\{j,k\}}f_iH_i) & \leq & M. & & \end{array}$$

Comme par hypothèse, $(f_j S f_k)\mathcal{R}(f_1, \dots, f_s) = 0$,

$$(f_j S f_k) = \sum_{i=1}^{s} f_i h'_i \text{ avec } exp(f_i h'_i) \leq exp(f_j S f_k) < \gamma.$$

On remplace $(f_j S f_k)$ par $\sum_{i=1}^{s} f_i h'_i$ dans la dernière expression de g - on remarque au passage que $exp(X^\delta \sum_{i=1}^{s} f_i h'_i) < M$ - et on obtient une nouvelle expression pour g, sous la forme $\sum_{i=1}^{s} f_i H'_i$, dans laquelle

- soit $\max(exp(f_i H'_i) < M$,
- soit il y a au moins un indice de moins, j, sur lequel ce maximum est atteint ; dans ce cas, en réitérant ce processus, on finit par obtenir une nouvelle écriture de g sous la forme $\sum_{i=1}^{s} f_i H''_i$ telle que $\max(exp(f_i H''_i) < M$.

On peut donc construire une suite de $\mathbb{N}^n$ infinie et strictement décroissante, ce qui contredit la proposition 9, donc l'hypothèse $g \neq 0$ était absurde.
□

On remarque que ce théorème est encore vrai si la structure algébrique sous-jacente (K) est un anneau : toutes les divisions qui interviennent dans la preuve sont exactes.

Remarque

On se donne toujours (quand on veut effectuer des calculs) un idéal par un système de générateurs.

5.3.2 Algorithme de Buchberger

L'idée de l'algorithme de Buchberger est basée sur le théorème précédent : on se donne en entrée un système de générateurs de I, on forme tous les S-polynômes à partir de cette famille et on les « réduit » (ou divise) par les polynômes de cette famille. Si un S-polynôme ne se réduit pas à 0, on l'ajoute à la famille et on réitère ce processus jusqu'à ce que tous les S-polynômes que l'on peut fabriquer se réduisent à 0 modulo les polynômes de la nouvelle famille (augmentée par rapport à la famille originelle).

L'algorithme de Buchberger de construction d'une base de Gröbner est le suivant[3] :

Buchberger

Entrées
    $F = (f_1, \dots, f_r)$ un système de générateurs de l'idéal I.
Résultat
    $G = (g_1, \dots, g_s)$ une base standard de I.
Initialisation
    $G \leftarrow F$.
Boucle
    si G ne contient qu'un seul élément alors
        renvoyer G
    sinon
        former C l'ensemble des couples d'éléments de G
    finsi

3. Buchberger a mis au point cet algorithme pendant qu'il était en thèse sous la direction de Gröbner.

tant que $C \neq \emptyset$ faire
pour tout $(f_i, f_j) \in C$ faire
supprimer (f_i, f_j) de C
former $f_i S f_j$
calculer $(f_i S f_j) mod G$
si $(f_i S f_j) mod G \neq 0$ alors
rajouter $(f_i S f_j) mod G$ à G
rajouter à C
l'ensemble des couples formés par les éléments de G et $(f_i S f_j) mod G$
finsi
finboucle
fintantque
Fin.

Ce processus s'arrête, sinon on construirait une suite infinie $(f_p)_{p \in \mathbb{N}}$ d'éléments de I tous non nuls. Soit alors $E = \bigcup_{p \in \mathbb{N}} (exp(f_p) + \mathbb{N}^n)$. E est une partie stable de $\mathbb{N}^n$, donc E admet une frontière finie $\partial E = \{\alpha_1, \ldots, \alpha_t\}$. Pour tout $i \in \{1, \ldots, t\}$, $\exists \sigma(i)$ tel que $\alpha_i \in \exp(f_{\sigma(i)}) + \mathbb{N}^n$. Soit $T = \max_{1 \leq i \leq t} \sigma(i)$, on a donc $E \subset \bigcup_{1 \leq p \leq T} (exp(f_p) + \mathbb{N}^n)$. Trivialement, $\bigcup_{1 \leq p \leq T} (exp(f_p) + \mathbb{N}^n) \subset E$. Donc $E = \bigcup_{1 \leq p \leq T} (exp(f_p) + \mathbb{N}^n)$, or $f_{T+1} \neq 0$ et il existe i, j appartenant à $\{1, \ldots, T\}$ tels que $f_{T+1} = (f_i S f_j) \mathcal{R}(f_1, \ldots, f_T)$, donc d'après la définition de la division, $exp(f_{T+1}) \in \mathbb{N}^n \setminus \bigcup_{1 \leq p \leq T} (exp(f_p) + \mathbb{N}^n)$. Cependant $exp(f_{T+1}) \in E$, ce qui est une contradiction.

Il est à noter que l'on ne dispose d'aucune information sur le nombre de calculs à effectuer, par exemple on ne sait pas comment évoluent les exposants des polynômes calculés.

Remarque
Cependant la complexité de cet algorithme a été établie récemment [49], confirmant les complexités observées expérimentalement : la complexité de cet algorithme, *i.e.* le nombre d'opérations élémentaires effectuées ou de façon équivalente le temps de calcul, est doublement exponentiel en n. Plus précisément, le maximum des degrés des polynômes de la base de Gröbner est inférieur au maximum des degrés des polynômes de la famille initiale, élevé à la puissance a^n où $a < \sqrt{3}$. Cette borne donne une idée du nombre de polynômes qui peuvent apparaître. Une telle complexité est réellement monstrueuse et habituellement un algorithme de complexité exponentielle est considérée comme impraticable : la taille des problèmes tractables est très limitée. Comme il s'avère qu'il n'y a pas d'autre alternative pour ce problème, cet algorithme est tout de même utilisé. Une telle complexité doit inciter à bien réfléchir au problème à résoudre et à vérifier soigneusement qu'il n'existe pas d'autre solution possible !

Pour passer d'une base de Gröbner à une base minimale, on réduit tous les éléments de la base par rapport à tous les autres et on supprime ceux qui se réduisent en 0.

Obtention d'une base de Gröbner minimale
Entrée
$F = (f_1, \ldots, f_r)$ une base de Gröbner de I.
Résultat
$G = (g_1, \ldots, g_s)$ une base minimale.
Initialisation
$G \leftarrow F$
Boucle
pour tout $g \in G$ faire

si $g\mathcal{R}(G \setminus \{g\}) = 0$ alors
$G \leftarrow G \setminus \{g\}$
sinon si $g\mathcal{R}(G \setminus \{g\}) \neq g$ alors
$G \leftarrow G \setminus \{g\} \cup \{g\mathcal{R}(G \setminus \{g\})\}$
finsi
finboucle
Fin.

La preuve de la correction de cet algorithme est laissée en exercice.

Autre exercice : écrire en Maple les fonctions Spol, Buchberger et basemin qui calculent respectivement le S-polynôme de deux polynômes fournis en entrée, une base de Gröbner d'une famille de polynômes et une base minimale à partir d'une base de Gröbner.

5.3.3 Exemple

Dans $\mathbb{Q}[X_1, X_2]$, avec l'ordre lexicographique,

$$\begin{aligned} f_1 &= X_1^2 + X_1X_2 + 2X_1 + X_2 - 1 \\ f_2 &= X_1^2 + 3X_1 - X_2^2 + 2X_2 - 1 \\ F &= (f_1, f_2) \end{aligned}$$

$G = F$.
On forme le S-polynôme $f_1Sf_2 = f_1 - f_2 = X_1X_2 - X_1 + X_2^2 - X_2$. Ce polynôme ne se réduit pas modulo (f_1, f_2), on ajoute donc $f_3 = X_1X_2 - X_1 + X_2^2 - X_2$ à G.
$C = \{(f_1, f_3), (f_2, f_3)\}$.
$f_1Sf_3 = X_2f_1 - X_1f_3 = X_1^2 + 3X_1X_2 + X_2^2 - X_2$, qui se réduit modulo (f_1, f_2, f_3) en $-X_2^2 + 1$ (il est égal à $f_1 + 2f_3 - X_2^2 + 1$).
Soit $f_4 = -X_2^2 + 1$, on enlève (f_1, f_3) de C et on ajoute f_4 à G et (f_1, f_4), (f_2, f_4), (f_3, f_4) à C qui est maintenant égal à $\{(f_2, f_3), (f_1, f_4), (f_2, f_4), (f_3, f_4)\}$.
$f_2Sf_3 = X_2f_2 - X_1f_3 = X_1^2 - X_1X_2^2 + 4X_1X_2 - X_2^3 + 2X_2^2 - X_2$, qui se réduit modulo (f_1, f_2, f_3, f_4) en 0 : $f_2Sf_3 = f_1 + (-X_2 + 2)f_3 + f_4$.
$C = \{(f_1, f_4), (f_2, f_4), (f_3, f_4)\}$.
$f_1Sf_4 = -X_2^2f_1 - X_1^2f_4 = -X_1^2 - X_1X_2^3 - 2X_1X_2^2 - X_2^3 + X_2^2$, qui se réduit modulo (f_1, f_2, f_3, f_4) en 0 : $f_1Sf_4 = -f_1 - (X_2^2 + 3X_2 + 2)f_3 - (X_2^2 + X_2 + 1)f_4$.
$C = \{(f_2, f_4), (f_3, f_4)\}$.
$f_2Sf_4 = -X_2^2f_2 - X_1^2f_4 = -X_1^2 - 3X_1X_2^2 + X_2^4 - 2X_2^3 + X_2^2$, qui se réduit modulo (f_1, f_2, f_3, f_4) en 0 : $f_2Sf_4 = -f_1 - (3X_2 + 2)f_3 - (X_2^2 + X_2 + 1)f_4$.
$C = \{(f_3, f_4)\}$.
$f_3Sf_4 = -X_2f_3 - X_1f_4 = X_1X_2 - X_1 - X_2^3 + X_2^2$, qui se réduit modulo (f_1, f_2, f_3, f_4) en 0 : $f_3Sf_4 = f_3 + X_2f_4$.
$C = \emptyset$, donc l'algorithme se termine et la base de Gröbner de (F) est $G = \{f_1, f_2, f_3, f_4\}$:

$$\begin{aligned} f_1 &= X_1^2 + X_1X_2 + 2X_1 + X_2 - 1 \\ f_2 &= X_1^2 + 3X_1 - X_2^2 + 2X_2 - 1 \\ f_3 &= X_1X_2 - X_1 + X_2^2 - X_2 \\ f_4 &= -X_2^2 + 1 \end{aligned}$$

5.3.4 Problèmes et améliorations

Expérimentalement, on constate que la division d'Hironaka est très coûteuse. Des efforts pour améliorer les sous-procédures les plus souvent appelées, telles que Im, consistent en un stockage adapté des polynômes et l'implantation d'une arithmétique rapide. On a par exemple intérêt à stocker les polynômes sous forme ordonnée, leur partie initiale étant alors le premier monôme. Nous n'entrerons pas dans le détail de l'arithmétique rationnelle et polynomiale, elle pourrait faire l'objet d'un autre chapitre.

Un autre aspect qui intervient dans le temps de calcul d'une division d'Hironaka est la façon dont sont conservés les coefficients, pour les polynômes appartenant à $Q[X_1, \ldots, X_n]$: soit on travaille toujours avec des coefficients réduits (*i.e.* le numérateur et le dénominateur de chaque coefficient sont premiers entre eux), mais les calculs de pgcd que cela entraîne sont coûteux, soit on ne réduit pas les coefficients, mais ils ont tendance à « exploser »[4] en cours de calcul, quitte à redevenir petits à la fin. Ce phénomène d'explosion des coefficients est caractéristique du calcul formel.
Une solution envisageable consiste à pré-multiplier chaque polynôme par le ppcm des dénominateurs de ses coefficients pour le transformer en un polynôme appartenant à $\boldsymbol{Z}[X_1, \ldots, X_n]$ et à effectuer tous les calculs dans $\boldsymbol{Z}$, en utilisant la pseudo-division d'Hironaka. Cela n'empêche évidemment pas les coefficients d'exploser.

On remarque aussi que l'ordre choisi influe sur la vitesse de calcul. L'ordre lexicographique est évidemment à proscrire puisque la complexité de la division n'est pas connue pour cet ordre. En pratique, il y a terminaison des algorithmes de division et de Buchberger avec cet ordre, mais l'ordre total est (expérimentalement) celui qui permet les calculs les plus rapides. Savoir dans quel ordre classer les indéterminées joue également sur le temps de calcul (la stratégie de Boge, Gebauer et Kredel [42] permet de les réordonner).

Une autre constatation expérimentale est d'une part qu'un calcul de base de Gröbner est très long et très gourmand en mémoire (sauf pour de petits exemples tels que ceux présentés dans ce chapitre) et d'autre part que l'on calcule un très grand nombre de S-polynômes qui se réduisent en 0. Ce phénomène est déjà sensible sur l'exemple de la section 5.3.3. De tels S-polynômes sont dits inutiles, les autres étant appelées « paires critiques ». Cependant la division d'Hironaka est effectuée sur un S-polynôme même si ce S-polynôme est inutile. Une grande partie des efforts de recherche portent sur la détection *a priori*, *i.e.* avant d'effectuer les calculs, des S-polynômes inutiles.

Un des critères proposés par Buchberger [44], le critère 1, s'énonce ainsi : soit $G = \{g_1, \ldots, g_r\}$, si la condition

$$\begin{array}{l} \forall k \in \{1, \ldots, r\}, \forall u_1, \ldots, u_k \in \{1, \ldots, r\} \\ \text{en notant } u_1 = i \text{ et } u_k = j, \text{ on a} \\ ppcm(init(f_{u_1}), \ldots, init(f_{u_k})) \leq ppcm(init(f_i), init(f_j)) \\ \text{et } \forall l \in \{1, \ldots, k\},\ (f_{u_l}, f_{u_{l+1}}) \notin C \end{array}$$

n'est pas vérifiée, alors le S-polynôme $(g_i S g_j)$ est inutile.
Ce critère est invérifiable en pratique, mais on utilise souvent une forme simplifiée (qui consiste à vérifier une partie de cette condition pour $k = 2$ seulement) :
soient g_i et g_j deux polynômes, soient m_i et m_j leurs parties initiales et soit r_i et r_j leurs restes (ou leur queue - *tail* en anglais), $r_i = g_i - m_i$ et $r_j = g_j - m_j$,

4. En informatique, on s'intéresse au nombre de chiffres des coefficients plutôt qu'à leur valeur : en effet la complexité des opérations arithmétiques est directement liée à ce nombre de chiffres, linéairement pour l'addition par exemple.

si $D = pgcd(m_i, m_j)$ divise r_i et r_j, alors le S-polynôme $g_i S g_j$ est inutile. Une version encore plus simplifiée de ce critère remplace le test « si $D = pgcd(m_i, m_j)$ divise r_i et r_j » par « si $D = 1$ ».

Les deux autres critères de Buchberger ayant des énoncés encore plus complexes, ils seront passés sous silence. Le lecteur intéressé peut consulter [44].

Des heuristiques permettent en général de diminuer le temps de calcul. Les voici en vrac :

- **classement des paires à traiter :** Buchberger [44] préconise de classer les couples de polynômes de C par ordre croissant des ppcm des monômes de tête. En cas d'égalité, on choisit la paire (f_i, f_j) ayant le plus petit j (le polynôme f_j est le plus ancien). Cette stratégie est appelée « stratégie normale » de sélection. L'utilisation de cette stratégie permet une efficacité maximale des critères qu'il propose, en réduisant les pgcd D à examiner. De plus, obtenir rapidement des polynômes avec de petits monômes de tête permet de diminuer beaucoup la taille des monômes lors de la réduction.

- **choix de l'ordre :** si, pour des raisons diverses (cf. §5.4.1), on a besoin de la base de Gröbner pour l'ordre lexicographique, on peut calculer la base de Gröbner en utilisant l'ordre total et, si la dimension de l'ensemble des zéros des polynômes est nulle, passer à la base désirée grâce à un changement linéaire de base [47, 51]. Cette procédure fonctionne en temps polynomial, ce qui est négligeable comparé au temps de calcul de la base de Gröbner. On peut également travailler avec l'ordre lexicographique, mais en utilisant la stratégie normale et en comparant pour l'ordre total les ppcm des monômes de tête.

- **pré-réduction de la famille donnée en entrée :** on peut ajouter un pré-calcul qui consiste à réduire chaque polynôme par rapport aux autres polynômes de la famille, en utilisant l'algorithme de passage d'une base à une base minimale (cet algorithme ne requiert pas que la famille donnée en entrée soit une base de Gröbner). Cela permet à la fois de diminuer le nombre de polynômes et surtout la taille des monômes. Le surcoût dû à ce pré-calcul est en général compensé par une diminution effective de la durée des calculs de la base de Gröbner... sauf pour de petits exemples.

- **classement des polynômes :** classer les polynômes de la famille par ordre décroissant peut permettre d'accélérer la décomposition de f en $f' + f''$ dans la division d'Hironaka.

- **division totale d'Hironaka :** l'idée de cette nouvelle division est de réduire tous les monômes de f et pas seulement le monôme de tête lors de la division d'Hironaka, pour la même raison : on obtient plus rapidement des monômes petits. On trouve parfois dans la littérature le vocable « base rédondante » pour désigner une base non minimale, « base irrédondante » pour une base minimale et « base réduite » pour une base minimale totalement réduite.

- **homogénéisation des polynômes :** [48] il s'avère que, lorsque l'on homogénéise les polynômes de départ (en multipliant chacun de leurs monômes par une puissance adéquate d'une nouvelle indéterminée), les calculs sont en général plus rapides, même si l'ordre utilisé est l'ordre lexicographique ; on « déshomogénéise » les polynômes obtenus et on les réduit pour aboutir à une base de Gröbner du système originel. Cependant, la base de Gröbner du système homogène peut être beaucoup plus grande que la base de Gröbner cherchée (et donc avoir nécessité plus de calculs).

Un algorithme de trace [56] peut être combiné aux stratégies précédentes. L'idée est d'effectuer les calculs modulo un nombre premier si on travaille dans $\boldsymbol{Z}$, ou de façon générale modulo un idéal premier de K, et de conserver une trace de ces calculs : quels sont les S-polynômes utiles et quels sont les polynômes de la famille impliqués dans la réduction d'un S-polynôme utile. On effectue ensuite les calculs dans l'anneau de départ ($\boldsymbol{Z}$ ou K), mais seuls les calculs qui ont été tracés sont effectués. On conserve également les monômes de tête des polynômes de la famille afin de vérifier que ce sont les mêmes pour les deux calculs ; si ce n'est pas le cas le second calcul renvoie un échec. On obtient ainsi un algorithme (probabiliste ou déterministe selon la variante choisie). La trace est dite, selon les cas,

– correcte : le second calcul ne renvoie pas d'échec et son résultat est une base de Gröbner ;

– valide : le second calcul ne renvoie pas d'échec mais le résultat n'est pas une base de Gröbner ;

– non valide : le second calcul renvoie un échec.

Si on calcule dans $\boldsymbol{Z}[X_1, \ldots, X_n]$ modulo un nombre premier p, on dit que ce nombre premier est

– bon si la trace est correcte : le calcul effectué en reproduisant dans $\boldsymbol{Z}[X_1, \ldots, X_n]$ les calculs utiles effectués dans $\boldsymbol{Z}/p\boldsymbol{Z}[X_1, \ldots, X_n]$ a fourni une base de Gröbner ;

– mauvais si la trace est non valide : un S-polynôme utile dans $\boldsymbol{Z}/p\boldsymbol{Z}[X_1, \ldots, X_n]$, calculé dans $\boldsymbol{Z}[X_1, \ldots, X_n]$, a son monôme de tête multiple de p ;

– malchanceux si la trace est valide mais non correcte : un S-polynôme a été abusivement décrété inutile dans $\boldsymbol{Z}/p\boldsymbol{Z}[X_1, \ldots, X_n]$, il n'a donc pas été calculé dans $\boldsymbol{Z}[X_1, \ldots, X_n]$; or s'il avait été calculé, il serait un polynôme dont tous les coefficients sont multiples de p. Ce cas est donc le plus difficile à détecter puisque le second calcul ne signale aucun problème.

Il existe des tests probabilistes ou déterministes pour vérifier si le résultat est une base de Gröbner. On peut également calculer la probabilité qu'a un grand nombre premier, ou un ensemble de nombres premiers, d'être bon(s).

Cependant, l'algorithme de trace ne dispense pas d'utiliser une heuristique qui permet de réduire le nombre de S-polynômes inutiles, lors du calcul modulaire. Une stratégie est à présent reconnue comme la meilleure, la stratégie dite « du sucre[5] », due à Traverso *et al.* [48]. Cette stratégie vise à conserver les avantages liés à l'homogénéisation, sans les inconvénients : on calcule directement la base de Gröbner correspondant au système donné en entrée.
L'algorithme du sucre est celui de Buchberger où l'on applique la stratégie normale, mais en simulant l'algorithme d'homogénéisation : on calcule le degré total du S-polynôme homogénéisé et on choisit la paire dont le degré, appelé « sucre », est le plus petit. En cas d'égalité on utilise la stratégie normale.
Le double sucre n'effectue une réduction d'un monôme dit intermédiaire (*i.e.* qui n'est pas le monôme de tête) que si cela n'augmente pas le sucre du polynôme.

Cette stratégie du sucre est actuellement celle qui permet d'obtenir les meilleurs temps de calcul, et de loin. En effet, dans les pires cas elle n'est pas plus longue que les autres stratégies, et

5. Cette stratégie n'est pas nommée ainsi parce qu'elle permet de sucrer des calculs, mais parce qu'elle donne l'arôme de l'algorithme d'homogénéisation.

selon les exemples elle est souvent de 2 à 10 fois plus rapide.

En résumé un certain consensus s'est dégagé quant aux heuristiques à utiliser, basé essentiellement sur des constatations expérimentales : effectuer des divisions totales d'Hironaka, en classant les polynômes par ordre d'arrivée dans la famille afin de réduire préférentiellement avec les plus vieux monômes, et enfin utiliser la stratégie du sucre.

5.4 Applications

Cette section contient l'énoncé du TP.

Les applications des bases de Gröbner, outre un test d'appartenance à un idéal, concernent la résolution de systèmes algébriques, la preuve automatique de théorèmes en géométrie et la preuve de circuits logiques.

Rappel : programmer la division d'Hironaka (cf. §5.2), le calcul d'un S-polynôme (cf. §5.3.1) et la réduction d'une base standard en base minimale (cf. §5.3.2).

Pour constater expérimentalement un phénomène de grossissement des coefficients, essayez la commande gbasis sur l'exemple suivant :

```
p1 := 2*x*z - z^2 - 5;
p2 := x*y + y^2*z + 3;
p3 := 8*x^3 - 3*y^2;
F  := [p1, p2, p3];
X  := [x, y, z];
gbasis(F, X, plex);
```

puis changez l'ordre des variables. Il n'aura pas échappé à votre perspicacité qu'il s'agit de l'exemple donné dans l'aide en ligne au sujet de gbasis (en échangeant x et z), avec l'ordre lexicographique et $z > y > x$.

5.4.1 Recherche des solutions d'un système algébrique

On a vu que les bases de Gröbner sont une généralisation aux polynômes à plusieurs variables de la notion de pgcd : dans le cas d'une seule indéterminée, un polynôme f appartient à l'idéal I engendré par $\{f_1, \ldots, f_s\}$ ssi le reste de la division de f par $g = pgcd(f_1, \ldots, f_s)$ est nul ; dans le cas de plusieurs indéterminées, un polynôme f appartient à l'idéal I engendré par $\{f_1, \ldots, f_s\}$ ssi le reste de la division par G la base de Gröbner de $(f_1, \ldots, f_s)$ est nul. Les bases de Gröbner permettent donc de tester l'appartenance d'un polynôme à un idéal.

Une autre application est la résolution de systèmes algébriques $f_1 = 0, \ldots, f_s = 0$ avec $\forall i \in \{1, \ldots, s\}$, $f_i \in K[X_1, \ldots, X_n]$. On suppose dorénavant que K est un corps algébriquement clos. Dans cette optique, on peut considérer les bases de Gröbner comme une généralisation de l'élimination de Gauss pour les systèmes linéaires : on peut considérer la i^e équation comme $f_i = 0$ avec f_i polynôme de degré 1. Lors de l'élimination de Gauss, on effectue un pivotage pour éliminer x_1 des équations autres que la première (par des calculs de S-polynômes entre f_1 et f_i, $2 \leq i \leq n$), et de proche en proche on obtient un système où le dernier polynôme g_n ne contient que l'indéterminée x_n, l'avant-dernier ne contient que x_{n-1} et x_n,... On résout alors l'équation $g_n = 0$, on obtient une valeur pour x_n que l'on substitue dans le polynôme précédent et en remontant on obtient la solution du système. Il en va presque de même pour les bases de Gröbner réduites calculées avec l'ordre lexicographique. Les deux différences majeures sont d'une part que chaque équation fait intervenir un polynôme p de degré non nécessairement égal à 1, donc l'équation $p = 0$ peut avoir plusieurs solutions qu'il faudra substituer successivement dans le reste du système, d'autre part on peut avoir plusieurs équations comportant les mêmes variables. Il faut donc les satisfaire simultanément, *i.e.* ne garder que l'intersection de leurs solutions, que l'on substitue dans les autres équations.

Par exemple, la base de Gröbner réduite de l'exemple du §5.3.3 est la suivante :
f_1 se réduit en 0 modulo (f_2, f_3, f_4),
f_2 se réduit en $X_1^2 + 3X_1 + 2X_2 - 2$ modulo (f_3, f_4),
f_3 se réduit en $X_1X_2 - X_1 - X_2 + 1$ modulo (f_2, f_4),
f_4 reste inchangé. La base de Gröbner réduite est donc $G = \{X_1^2 + 3X_1 + 2X_2 - 2, X_1X_2 - X_1 - X_2 + 1, -X_2^2 + 1\}$.
On remarque que X_1 n'apparaît pas dans f_4 qui est un polynôme en X_2 seulement. On le factorise : $f_4 = -(X_2-1)(X_2+1)$. ses racines sont de la forme $(\lambda, 1)$ et $(\mu, -1)$. Remplaçons dans f_3 et f_2 :

- $(\lambda, 1)$:
  $f_3(\lambda, 1) = \lambda - \lambda - 1 + 1 = 0$: f_3 a $(\lambda, 1)$ pour zéro.
  $f_2(\lambda, 1) = \lambda^2 + 3\lambda = \lambda(\lambda + 3)$: f_2 a $(0, 1)$ et $(-3, 1)$ pour zéros.
- $(\mu, -1)$:
  $f_3(\mu, 1) = -\mu - \mu + 1 + 1 = -2(\mu - 1)$: f_3 a $(1, -1)$ pour zéro.
  $f_2(\mu, 1) = \mu^2 + 3\mu - 2 - 2 = (\mu + 4)(\mu - 1)$: f_2 a $(4, -1)$ et $(1, -1)$ pour zéros.

Les solutions du système sont donc $\{(0; 1), (-3; 1), (1; -1)\}$.

On connaît un majorant du nombre de solutions de ce système (de façon théorique au moins) :

$$Card(\mathbb{N}^n \setminus exp(I)) \geq \text{ nombre de solutions.}$$

Plus généralement, on peut montrer que, si le système à résoudre est $f_1 = 0, \ldots, f_s = 0$, et si G, la base de Gröbner de $\{f_1, \ldots, f_s\}$ avec l'ordre lexicographique et $X_1 > \ldots > X_n$, est sous forme fortement triangulaire, alors le système a un nombre fini de solutions. Cela signifie qu'il existe dans G un polynôme g_n en X_n seulement, un polynôme g_{n-1} en X_n et X_{n-1} seulement contenant un monôme de la forme $a_{n-1}X_{n-1}^{d_{n-1}}$ avec a_{n-1} et d_{n-1} non nuls,..., un polynôme g_1 en $X_1, \ldots, X_n$ contenant un monôme de la forme $a_1X_1^{d_1}$ avec a_1 et d_1 non nuls. On peut donc résoudre g_n en X_n, substituer une des solutions dans les polynômes en X_n et X_{n-1}, résoudre en X_{n-1}... On a un système triangulaire en les indéterminées, mais ce système n'est pas linéaire.

Définition 28 : forme fortement triangulaire
Soit K un corps et soit G un sous-ensemble fini de $K[X_1, \ldots, X_n]$. Soit $(G_0, \ldots, G_n)$ la partition de G définie par :

$$\begin{array}{lcl}
G_0 & = & G \cap K[X_1, \ldots, X_n] \setminus K[X_2, \ldots, X_n] \\
G_1 & = & G \cap K[X_2, \ldots, X_n] \setminus K[X_3, \ldots, X_n] \\
\vdots & & \\
G_i & = & G \cap K[X_{i+1}, \ldots, X_n] \setminus K[X_{i+2}, \ldots, X_n] \\
\vdots & & \\
G_{n-1} & = & G \cap K[X_n] \setminus K \\
G_n & = & G \cap K
\end{array}$$

G est sous forme fortement triangulaire ssi

i. $G_n = G \cap K = (0)$.

ii. pour tout $i \in \{0, \ldots, n-1\}$, il existe $g_i \in G_i$ contenant un monôme de la forme aX_{i+1}^d avec $a \in K \setminus \{0\}$ et $d > 0$.

L'exemple traité ci-dessus vérifie ces conditions.

Le *Nullstellensatz* de Hilbert nous dit qu'un système n'admet pas de solutions ssi G_n contient un polynôme constant non nul.

Théorème 8 Nullstellensatz *de Hilbert*
Si K est un corps algébriquement clos et si I est un idéal de $K[X_1, \ldots, X_n]$,

$$I = K[X_1, \ldots, X_n] \text{ ssi } \mathcal{Z}(I) = \{\lambda = (\lambda_1, \ldots, \lambda_n) \in K^n \;/\; \forall f \in I, f(\lambda) = 0\} \text{ est égal à } \emptyset.$$

$\mathcal{Z}(I)$ est appelé ensemble des zéros de I.

Ce théorème est l'une des formulations possibles du *Nullstellensatz.*

Un test simple de l'existence de solutions consiste donc à vérifier si une base de Gröbner de I contient ou non un polynôme constant non nul. On aimerait avoir une indication plus fine sur l'ensemble des zéros de I, en particulier savoir si cet ensemble est fini afin de le déterminer complètement si c'est possible. Après avoir testé si I admet des solutions grâce au critère précédent, il suffit d'examiner si la base de Gröbner est sous forme fortement triangulaire.

On appelle système algébrique un ensemble fini d'équations de la forme $f_i = 0$ où $f_i \in K[X_1, \ldots, X_n]$, $1 \leq i \leq s$.

Théorème 9 *Soit F un système algébrique, F admet un nombre fini de solutions (*i.e. *$\mathcal{Z}(I)$ est de cardinal fini) ssi une base de Gröbner de $(f_1, \ldots, f_s)$ pour l'ordre lexicographique est sous forme fortement triangulaire.*

Le lecteur intéressé par la démonstration de ces résultats pourra se reporter aux ouvrages de Mishra ou de Becker et Weispfenning [53, 41].

L'algorithme de résolution de ce système algébrique (l'équivalent de la remontée triangulaire de Gauss) est le suivant :

Zéros

Entrées :

$F = \{f_1, \ldots, f_s\} \subset K[X_1, \ldots, X_n]$

K corps algébriquement clos

Résultat :

les zéros de F si F admet un nombre fini de solutions dans K^n

Initialisation et élimination des cas particuliers

```
G := gbasis (F, [X_1, ... , X_n], plex);
```

$G = (G_0, \ldots, G_n)$

si G_n contient un polynôme constant non nul alors

rendre un échec 'pas de solutions'
sinon si G n'est pas sous forme fortement triangulaire alors
rendre un échec 'nombre de solutions non fini'
sinon
Boucle
$H := \{g \in G_{n-1}\}$
p_{n-1} := pgcd des polynômes $\in H$
$X_{n-1} := \{(\lambda_n) \;/\; p_{n-1}(\lambda_n) = 0\}$
pour $i := n-1$ à 1 par pas de -1 faire
$X_{i-1} := \emptyset$
pour tout $(\lambda_{i+1}, \ldots, \lambda_n) \in X_i$ faire
$H := \{g(X_i, \lambda_{i+1}, \ldots, \lambda_n), g \in G_{i-1}\}$
p_{i-1} := pgcd de tous les polynômes inH
$X_{i-1} := X_{i-1} \cup \{(\lambda_i, \lambda_{i+1}, \ldots, \lambda_n) \;/\; p_{i-1}(\lambda_i) = 0\}$
finpour
finsi
renvoyer(X_0)
finsi
Fin.

Cet algorithme utilise de façon implicite une procédure d'extraction des racines d'un polynôme à une seule variable ainsi qu'une arithmétique sur une extension algébrique de dimension finie d'un corps K si K n'est pas algébriquement clos (et K est souvent égal à $\mathbb{Q}$ en calcul formel). En Maple, on fera appel à la fonction solve.

« Vérifiez » expérimentalement que dans le cas d'une seule indéterminée, l'algorithme de Buchberger calcule le pgcd de la famille de polynômes donnés en entrée.

« Vérifiez » expérimentalement que dans le cas d'un système linéaire, l'algorithme de Buchberger correspond à l'algorithme d'élimination de Gauss.

Écrivez cet algorithme Zeros et testez-le sur l'exemple du §5.3.3 ainsi que sur d'autres exemples à votre guise.

5.4.2 Preuve automatique de théorèmes en géométrie

L'utilisation des bases de Gröbner pour la preuve automatique de théorèmes en géométrie ne relève pas du domaine des techniques logiques de preuve : en effet aucun mode de déduction logique tel que le *modus ponens* par exemple n'est mis en œuvre. De plus, ce mode de preuve repose sur une formulation algébrique du problème à traiter et il n'existe pas à ma connaissance de traducteurs automatisés. Cet avertissement donné, regardons sous quelle forme doit se présenter l'affirmation à prouver : on doit avoir une formule du type :

$$(f_1 = 0 \wedge f_2 = 0 \wedge \ldots \wedge f_s = 0) \Rightarrow (g = 0)$$

avec $f_1, \ldots, f_s$ et g des polynômes en n indéterminées.

On appelle la conjonction $f_1 = 0 \wedge f_2 = 0 \wedge \ldots \wedge f_s = 0$ les prémisses de l'affirmation et $g = 0$ sa conclusion.

L'algorithme de Wu [57] de preuve de théorèmes de géomètrie fonctionne comme suit. On calcule une base de Gröbner de l'idéal I engendré par $\{f_1, \ldots, f_s\}$: $G = \{g_1, \ldots, g_r\}$. Si G contient un polynôme constant, alors le théorème est faux (les prémisses se contredisent), sinon on réduit g par rapport à I. Si g se réduit en 0, alors le théorème est génériquement vrai (*i.e.* vrai partout sauf sur un ensemble de mesure nulle). Enfin si g ne se réduit pas en 0, l'algorithme de Wu renvoie la réponse « Non confirmé ».

Il est même possible de connaître les équations de l'ensemble de mesure nul sur lequel le théorème peut ne pas être vrai, et que l'on appelle ensemble des cas dégénérés (parce qu'ils sont souvent les équations de ce que l'on a coutume d'appeler cas dégénérés, tels qu'un triangle aplati,...) : si g se réduit en 0, on a une écriture de la forme

$$init(g_r)^{\alpha_r} \ldots init(g_1)^{\alpha_1} g = \sum_{i=1}^{r} h_i g_i,$$

la clause $init(g_r) = 0 \wedge \ldots \wedge init(g_1) = 0$ est appelée condition de dégénérescence associée au théorème.

Les limites de cette technique de preuve sont d'une part que l'algorithme ne marche pas pour n'importe quel corps : si on a $(x^2 + y^2 = 0) \Rightarrow (y = 0)$, ce théorème est vrai dans $\mathbb{R}^2$ et faux dans $\mathbb{C}^2$, mais l'algorithme de Wu répondra faux, et d'autre part l'impossibilité de traiter des inégalités, ce qui interdit toutes les notions comme « être à l'intérieur », « être à l'extérieur » d'un objet ou « être entre » deux objets.

Démontrer que dans un parallélogramme $ABCD$, l'intersection des diagonales est le milieu de $[AC]$. On prendra pour origine A et comme axes AB et un axe orthogonal.

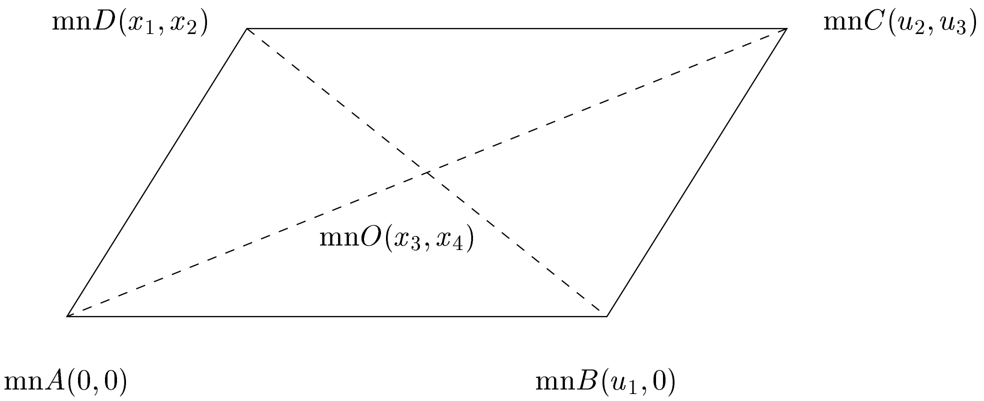


FIG. 5.5 – *Illustration du problème.*

Théorème de Simson : dans le plan affine euclidien, soient trois points non alignés A, B et C. Soit M le centre du cercle $\mathcal{C}$ circonscrit au triangle (A, B, C). Soit D un point quelconque de ce cercle.
Soit E la projection orthogonale de D sur la droite AB.
Soit F la projection orthogonale de D sur la droite BC.

Soit G la projection orthogonale de D sur la droite AC.
Montrer que E, F et G sont alignés.
En choisissant comme repère le repère orthogonal, d'origine E, d'axe horizontal AB, établir une formulation algébrique de ce problème.

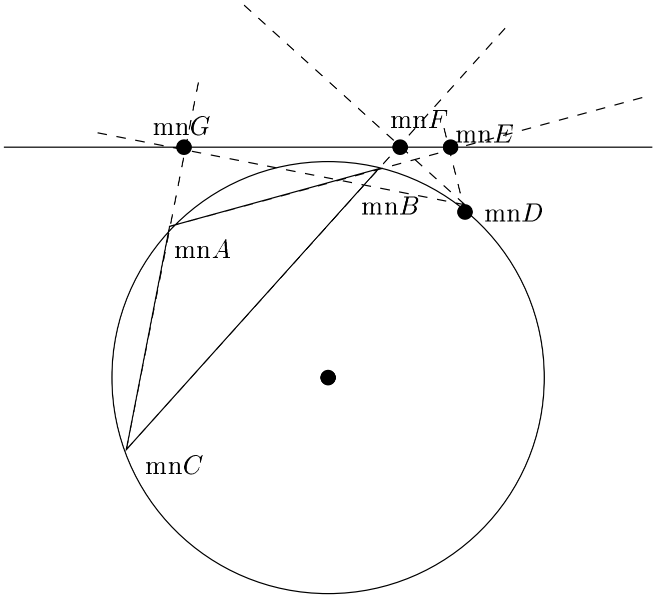


FIG. 5.6 – *Illustration du théorème de Simson.*

Olympiade mathématique internationale : [50]
Soit $ABCD$ un carré et soient K, L, M et N tels que ABK, BCL, CDM et DAN sont des triangles équilatéraux tous soit intérieurs soit extérieurs au carré. Montrer que les milieux de $[KL]$, $[LM]$, $[MN]$ et $[NK]$ ainsi que les intersections $CL \cap BK$, $CM \cap DN$, $BK \cap AN$, $BL \cap CM$, $AN \cap DM$, $AK \cap BL$, $DM \cap CL$ et $DN \cap AK$ forment un polygone régulier à 12 sommets.

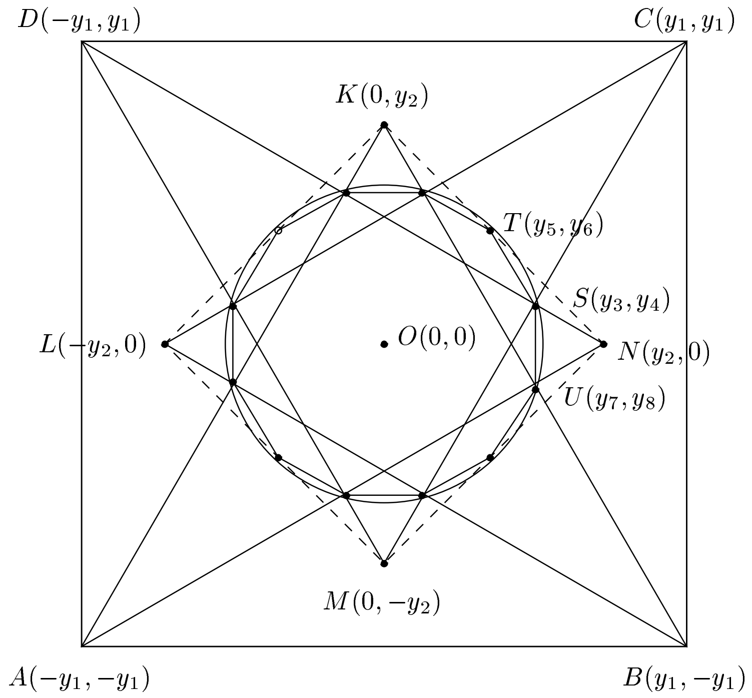


FIG. 5.7 – *Olympiade mathématique internationale.*

5.4.3 Preuve de circuits logiques

Par « preuve de circuits logiques » on entend que l'on veut vérifier si un circuit électronique, donné par son schéma logique, effectue correctement la ou les fonction(s) qui lui sont demandée(s).

On extrait, à partir du schéma logique d'un circuit, tel que celui de la figure 5.8, les fonctions booléennes représentant les sorties de ce circuit en fonction de ses entrées, en utilisant de nouvelles variables correspondant à chaque porte logique. On écrit également les fonctions supposées calculées par le circuit comme des fonctions booléennes. Il s'agit alors de tester l'égalité de fonctions booléennes. La majorité des techniques de test sont à l'heure actuelle basées sur une mise sous forme canonique mais cependant condensée des fonctions à comparer. La technique proposée n'est donc pas représentative du domaine de la preuve des circuits logiques, mais bien du champ d'application des bases de Gröbner.
Le calcul dans l'algèbre de Boole $\mathcal{B} = (\{Vrai, Faux\}, \vee, \wedge, \neg)$ n'est pas possible : il manque l'opération de soustraction, qui permet de construire les S-polynômes. On utilise la transformation de Stone qui permet de se placer dans un anneau.

Transformation de Stone

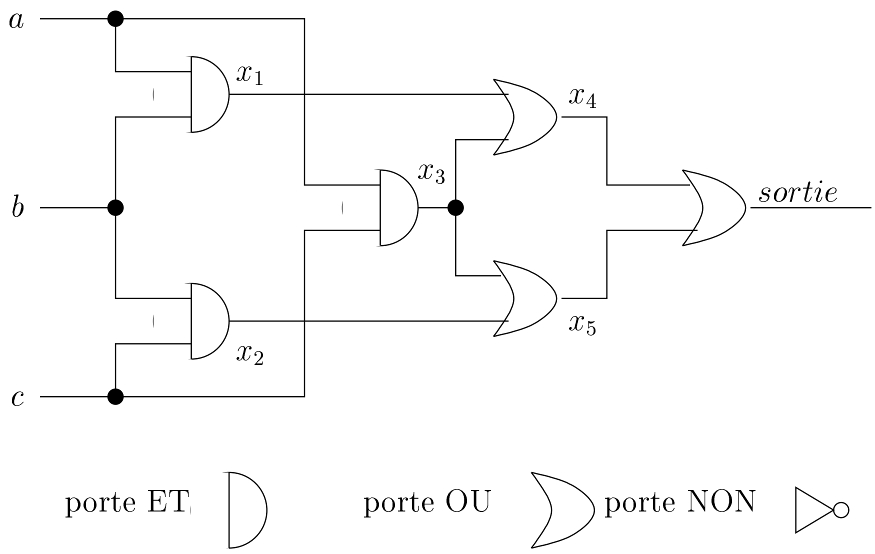


FIG. 5.8 – *schéma logique d'un circuit.*

Elle permet de passer de l'algèbre de Boole à l'anneau $\boldsymbol{Z}/2\boldsymbol{Z}$:

$$(S_0)\quad \left\{\begin{array}{llll} + & a+b & = & (a\wedge\neg b)\vee(\neg a\wedge b) \\ & a*b & = & a\wedge b \end{array}\right.$$

et réciproquement

$$(S_1)\quad \left\{\begin{array}{llll} \vee & a\vee b & = & a+b+a*b \\ \wedge & a\wedge b & = & a*b \\ \neg & \neg a & = & 1-a \end{array}\right.$$

À partir du circuit, on écrit les fonctions calculées par chacune de ses portes. On lit sur le circuit :

$$(S)\quad \left\{\begin{array}{lll} x_1 & = & a\wedge b \\ x_2 & = & b\wedge c \\ x_3 & = & a\wedge c \\ x_4 & = & x_1\vee x_3 \\ x_5 & = & x_2\vee x_3 \\ sortie & = & x_4\vee x_5 \end{array}\right.$$

On a $sortie = F(a,b,c,x_1,\ldots,x_5)$. On aimerait que ce circuit calcule $G(a,b,c) = ab\vee bc\vee ac$ (ce qui, sur cet exemple choisi pour être faisable à la main, saute aux yeux, mais habituellement un circuit comporte beaucoup plus de portes logiques et son dessin a été modifié pour obtenir une surface moindre). On applique la transformation de Stone au système, à F et à G :

$$(S')\quad \left\{\begin{array}{lll} x_1+ab & = & 0 \\ x_2+bc & = & 0 \\ x_3+ac & = & 0 \\ x_4+x_1+x_3+x_1x_3 & = & 0 \\ x_5+x_2+x_3+x_2x_3 & = & 0 \end{array}\right.$$

$$\begin{array}{lll} F(a,b,c,x_1,x_2,x_3,x_4,x_5) & = & x_4+x_5+x_4x_5 \\ G(a,b,c) & = & ab+bc+ac+abc \end{array}$$

et on travaille dans l'anneau $\boldsymbol{Z}/2\boldsymbol{Z}[a,b,c,x_1,x_2,x_3,x_4,x_5]$. Il s'agit de prouver que dans $\boldsymbol{Z}/2\boldsymbol{Z}[a,b,c,x_1,x_2,x_3,x_4,x_5]$, $F=G$ (ou $F+G=0$) sachant que les x_i vérifient le système (S').
Soit I l'idéal engendré par $\{x_1+ab, x_2+bc, x_3+ac, x_4+x_1+x_3+x_1x_3, x_5+x_2+x_3+x_2x_3\}$. On sait que cela revient à montrer que $F+G$ appartient à l'idéal $J=(\boldsymbol{Z}/2\boldsymbol{Z})[a,b,c,x_1,x_2,x_3,x_4,x_5]/I$. On peut montrer que c'est équivalent à tester si $F+G$ appartient à $(x_1+ab, x_2+bc, x_3+ac, x_4+x_1+x_3+x_1x_3, x_5+x_2+x_3+x_2x_3, a^2+a, b^2+b, c^2+c, x_1^2+x_1, \ldots, x_5^2+x_5)$. Ces derniers polynômes permettent de simuler dans $\boldsymbol{Z}/2\boldsymbol{Z}$ le calcul booléen (et plus précisément l'idempotence de l'addition). Il s'avère que, pour notre exemple, cet ensemble de polynômes forme une base (non minimale) de l'idéal qu'il engendre. Reste à réduire $F+G$ modulo cet ensemble : on prend l'ordre lexicographique avec $x_5>x_4>x_3>x_2>x_1>c>b>a$, ce qui conduit[6] à $F+G\to^* 0$.

De façon générale, on écrit un système polynomial avec une variable par porte logique. On a pour variables $e_1,\ldots,e_m$ les entrées, $x_1,\ldots,x_k$ les portes logiques et $s_1,\ldots,s_n$ les sorties, et $p_1,\ldots,p_k$ sont les polynômes associés aux x_i. On a les polynômes F_i correspondant aux sorties du circuit et G_i les polynômes correspondant aux fonctions désirées pour ces sorties. On applique la transformation de Stone à tous ces polynômes pour se placer dans $\boldsymbol{Z}/2\boldsymbol{Z}[e_1,\ldots,e_m,x_1,\ldots,x_k,s_1,\ldots,s_n]$. On teste alors pour chaque $i\in\{1,\ldots,n\}$ si F_i+G_i appartient à l'idéal engendré par $\{p_1,\ldots,p_k,e_1^2+e_1,\ldots,e_m^2+e_m,x_1^2+x_1,\ldots,x_k^2+x_k\}$.

Cet exposé doit beaucoup à la thèse de P. Sénéchaud [55].

Voici deux exemples plus ludiques à traiter, tirés du livre de R.M. Smullyan, *Quel est le titre de ce livre ?* (Dunod, 1981).

Exemple :

L'action se déroule sur une île dont les habitants, cannibales au demeurant, sont soit des Purs, soit des Pires. Les Purs disent toujours la vérité et les Pires mentent toujours. De plus, certains habitants (indépendamment d'être Purs ou Pires) ont la caractéristique d'être des Loups-Garous. Vous êtes prisonnier de trois habitants de l'île dont vous savez qu'un seul est un Loup-Garou. Ils vous disent :
A : « Je suis un Loup-Garou ».
B : « Je suis un Loup-Garou ».
C : « Un au plus de nous trois est un Pur ».
Si vous devinez qui ils sont, ils vous laissent la vie sauve. C'est le moment ou jamais de faire appel aux bases de Gröbner !

Exemple :

Les coffres de la ville de Venise sont gravés par deux familles : les Bellini et les Cellini.
Chaque fois qu'un Cellini grave un coffre, l'inscription qu'il a portée sur le coffre est fausse et chaque fois qu'un Bellini grave un coffre, l'inscription portée est vraie.
Les deux familles participèrent à la décoration de paires de coffres dont l'un était d'or et l'autre d'argent. Chaque coffre était alors l'œuvre d'un seul homme et une paire pouvait être faite par deux hommes de familles différentes.
On a la paire suivante[7] :

- sur le coffre d'or est gravée l'inscription : « Les deux coffres de cette paire furent faits par des membres de la famille Cellini. »

6. La notation $\to^*$ signifie que toutes les réductions possibles sont appliquées.
7. Votre grande perspicacité vous permet même de repérer que ces informations sont redondantes !

- sur le coffre d'argent est gravée l'inscription : « Les deux coffres ne furent pas faits par des membres de la même famille. »

Le problème est alors de déterminer qui a gravé les coffres.

Conclusion

Ce cours est destiné à illustrer le type de calculs effectués en calcul formel et leurs applications. Vous avez déjà dû vous apercevoir que le calcul exact est très coûteux et que les problèmes traités sont de petite taille comparés aux problèmes résolus numériquement. Cette caractéristique est portée à son paroxysme dans le cas de l'algorithme de Buchberger, à cause de sa complexité doublement exponentielle.

Bibliographie

[1] A-V. Aho. Algorithms for finding patterns in strings. In J. van Leuwen, editor, *Algorithms and Complexity*, pages 253–300. Elsevier, 1990.

[2] A.V. Aho, J.E. Hopcroft, and J.D. Ullman. *The design and analysis of computer algorithms.* Addison-Wesley, 1974.

[3] D. Bini and V. Pan. *Numerical and Algebraic Computations with Matrices and Polynomials.* Birkhäuser, 1992.

[4] R.S. Boyer and J.S. Moore. A Fast String Searching Algorithm. *Communications ACM*, 20(10):62–72, 1977.

[5] D.G. Cantor and H. Zassenhaus. A new algorithm for factoring polynomials over a finite field. *Math. Computation*, 36:587–592, 1981.

[6] J. Jájá. *An Introduction to Parallel Algorithms.* Addison-Wesley, 1992.

[7] E. Kaltofen. Polynomial factorization 1982-1986. In D.V. Chudnovsky and R.D. Jenks, editors, *Lecture Notes in Pure and Applied Math.*, number 125, pages 285–309. Marcel Dekker, Inc., 1990.

[8] E. Kaltofen. Polynomial factorization 1987-1991. RR 92-1, RPI, Rensselaer Polytechnic Institute, 1992.

[9] R. Kannan, H.W. Lenstra, and L. Lovász. Polynomial factorization and nonrandomness of bits of algebraic and some transcendental numbers. *Math. Comp.*, (50):235–250, 1988.

[10] R.M. Karp and M.O. Rabin. Efficient Randomized Pattern-Matching Algorithms. *IBM J. Research Develop.*, 31(2):249–260, 1987.

[11] D.E. Knuth. *Seminumerical Algorithms.* Addison-Wesley, 1981.

[12] A.K. Lenstra, H.W. Lenstra, and L. Lovász. Factoring polynomials with rational coefficients. *Math. Annalen*, 261:513–534, 1982.

[13] M. Mignotte. *Mathématiques pour le calcul formel.* Presses Universitaires de France, 1989.

[14] R.T. Moenck. Fast computation of gcd. In *5th. ACM Symp. Theory Comp.*, pages 142–151, 1973.

[15] D. Bini and V. Pan. *Polynomial and matrix computations.* Birkhäuser, 1994.

[16] E.H. Bareiss. Computational solution of matrix problems over an integral domain. *J. Inst. Math. Appl.*, 10:68–104, 1972.

[17] J. Davenport, Y. Siret, and E. Tournier. *Calcul Formel. Systèmes et algorithmes de manipulations algébriques.* Masson, 1987.

[18] R. Loos. Generalized polynomial remainder sequences. In G.E. Collins, B. Buchberger and R. Loos editors, *Computer Algebra – Symbolic and Algebraic Computation.* Springer-Verlag, 1982.

[19] K.O. Geddes, R. Czapor, and G. Labahn. *Algorithms for computer algebra.* Kluwer Academic Press, 1992.

[20] J.D. Dixon. Exact solution of linear equations using p-adic expansions. *Numer. Math.*, 40:137–141, 1982.

[21] R.T. Gregory and E.V. Krishnamurthy. *Methods and applications of error-free computations.* Springer-Verlag, 1984.

[22] C. Moler, 1993. Communication about Matlab test example *gallery(3).*

[23] F.R. Gantmacher. *Théorie des matrices.* Dunod, Paris, France, 1966.

[24] I. Gohberg, P. Lancaster, and L. Rodman. *Invariant subspaces of matrices with applications.* Wiley-Interscience, 1986.

[25] I. Gohberg, P. Lancaster, and L. Rodman. *Matrix polynomials.* Academic Press, New York, 1982.

[26] M. Newman. *Integral Matrices.* Academic Press, 1972.

[27] C.C. MacDuffee. *The theory of matrices.* Chelsea, New-York, 1956.

[28] T. Kailath. *Linear systems.* Prentice Hall, 1980.

[29] P. Ozello. *Calcul exact des formes de Jordan et de Frobenius d'une matrice.* PhD thesis, Université Scientifique et Médicale de Grenoble, France, 1987.

[30] H. Lüneburg. *On rational form of endomorphism: a primer to constructive algebra.* Wissenschaftsverlag, Mannheim, 1987.

[31] P. Van Dooren. The computation of the Kronecker canonical form of a singular pencil. *Linear Algebra and its Applications*, 27:103–140, 1979.

[32] E. Kaltofen, M.S. Krishnamoorthy, and B.D. Saunders. Fast parallel computation of Hermite and Smith forms of polynomials matrices. *SIAM J. Alg. Disc. Meth.*, 8 4, pp 683-690, 1987.

[33] E. Kaltofen, M.S. Krishnamoorthy, and B.D. Saunders. Parallel algorithms for matrix normal forms. *Linear Algebra and its Applications*, 136:189–208, 1990.

[34] S.E. Labhalla, H. Lombardi, and R. Marlin. Algorithmes de calcul de la réduction d'Hermite d'une matrice à coefficients polynomiaux. To appear in *Theoretical Computer Science*, 1995.

[35] G. Villard. Computation of the Smith normal form of polynomial matrices. In *International Symposium on Symbolic and Algebraic Computation, Kiev, Ukraine*. ACM Press, pp 209–217, July 1993.

[36] R. Kannan. Solving systems of linear equations over polynomials. *Theoretical Computer Science*, 39:69–88, 1985.

[37] S. Kung, T. Kailath, and M. Morf. A generalized resultant matrix. *IEEE Transf. Automat. Control.*, (23):1043–1046, 1978.

[38] J.T. Schwartz. Fast probabilistic algorithms for verification of polynomial identities. *J. ACM*, 27:701–717, 1980.

[39] G. Brassard and P. Bratley. *Algorithmique : conception et analyse*. Masson, 1988.

[40] J.-M. Muller. *Arithmétique des ordinateurs*. Masson, 1989.

[41] T. Becker and V. Weispfenning. *Gröbner Bases - A computational approach to commutative algebra*. Springer Verlag, 1993.

[42] W. Böge, R. Gebauer, and H. Kredel. Some examples for solving systems of algebraic equations by calculating Gröbner bases. *Journal of Symbolic Computing*, 2(1):83–98, 1986.

[43] B. Buchberger. *Ein Algorithmus zum Auffinden des Basiselemente des Restklassenringes nach einem nulldimensionalen Polynomialideal*. PhD thesis, University of Innsbruck, Austria, 1965.

[44] B. Buchberger. A criterion for detecting unnecessary reduction in the construction of Gröbner bases. In Lecture Notes in Computer Science, editor, *EUROSAM 79, Marseille*, volume 72, pages 3–21, 1979.

[45] S.-C. Chou, W.F. Shelter, and J.-G. Yang. Characteristic sets and Gröbner bases in geometry theorem proving. Raisonnement géométrique assisté par ordinateur, support de cours tome I, INRIA Sophia Antipolis, juin 1987.

[46] D. Cox, J. Little, and D. O'Shea. *Ideals, Varieties and Algorithms*. Springer Verlag.

[47] J.-C. Faugère. *Résolution des systèmes polynomiaux*. PhD thesis, Paris 6, 1994.

[48] A. Giovini, T. Mora, G. Niesi, Robbiano, and C. Traverso. "One sugar cube, please" or "Selection strategies in Buchberger algorithm". In S.M. Watt, editor, *ISSAC'91*, pages 49–54. ACM Press, 1991.

[49] M. Giusti. Some effectivity problems in polynomial ideal theory. Lecture Notes in Computer Science, volume 174, pages 159–171, 1984.

[50] B. Kutzler and S. Stifter. On the application of Buchberger's algorithm to automated geometry proving. *Journal of Symbolic Computation*, 2(4):389–397, December 1986.

[51] D. Lazard. Solving zero-dimensional algebraic systems. *Journal of Symbolic Computations*, 13(2):117–131, February 1992.

[52] M. Lejeune-Jalabert. Effectivité de calculs polynomiaux. Cours de DEA de l'Université Joseph Fourier - Grenoble, 1984-1985.

[53] B. Mishra. *Algorithmic algebra.* Texts and Monographs in Computer Science. Springer Verlag, 1993.

[54] E. Ramis, C. Deschamps, and J. Odoux. *Cours de Mathématiques Spéciales - 1 : algèbre.* Masson, 1985.

[55] P. Sénéchaud. *Bases de Gröbner booléennes : méthodes de calcul - applications - parallélisation.* PhD thesis, INPG Grenoble, 1990.

[56] C. Traverso. Gröbner trace algorithms. In P. Gianni, editor, *ISSAC'88*, Lecture Notes in Computer Science, volume 358, pages 125–138, 1988.

[57] W.T. Wu. Basic principles of mechanical theorem proving in elementary geometry. *Journal of Automated Reasoning*, 2:219–252, 1986.

Index