Fonctions generatrices et asymptotique automatique

Bruno Salvy

INRIA

UNITÉ DE RECHERCHE
INRIA-ROCQUENCOURT

Institut National
de Recherche
en Informatique
et en Automatique

Domaine de Voluceau
Rocquencourt
B.P.105
78153 Le Chesnay Cedex
France
Tél.:(1) 39 63 55 11

Rapports de Recherche

N° 967

Programme 1

FONCTIONS GÉNÉRATRICES ET ASYMPTOTIQUE AUTOMATIQUE

Bruno SALVY

Janvier 1989

RR_0967

Generating Functions and Automatic Asymptotics

Bruno Salvy[1]

Abstract: $\Lambda\Upsilon\Omega$ (INRIA Research Report 876, July 1988) is a system designed to perform automatic average case analysis of well-defined classes of algorithms operating over "decomposable" data structures. It consists of an 'Algebraic Analyzer' System that compiles algorithms specifications into generating functions of average costs, and an 'Analytic Analyzer' System that extracts asymptotic informations on coefficients of generating functions. This report describes the Analytic Analyzer. The Algebraic Analyzer is described in a companion report by Paul Zimmermann.

Generating functions are a powerful tool in combinatorics and probability theory. In computer science, they are of use in average-case complexity analysis of algorithms. A sequence of numbers is encoded as a power series, which becomes a complex function with derivability properties in suitable domains. The usual process is the following: from the structure one first gets the generating function by the "admissible constructors" method. This phase is accomplished by the Algebraic Analyzer (Alas). Then by singularity analysis or by a direct use of Cauchy's theorem, one obtains the asymptotic behavior of the Taylor coefficients. Our aim is to automatize the latter step. Given the formal expression of a generating function, the Maple program presented here outputs the asymptotic behavior of its coefficients when ranks tend to infinity.

Fonctions Génératrices et Asymptotique Automatique

Résumé : $\Lambda\Upsilon\Omega$ (Rapport de recherche INRIA 876) est un système conçu pour réaliser automatiquement l'analyse en moyenne de classes bien définies d'algorithmes opérant sur des structures de données "décomposables". Il consiste en un 'Analyseur Algébrique' qui compile des spécifications d'algorithmes en séries génératrices de coût moyen ainsi qu'en un 'Analyseur Analytique' qui extrait des informations asymptotiques sur les coefficients de séries génératrices. Ce rapport décrit l'Analyseur Analytique. L'Analyseur Algébrique est décrit dans un autre rapport, fait par Paul Zimmermann.

Les fonctions génératrices sont un outil puissant de la combinatoire et de la théorie des probabilités. Elles trouvent leur application en informatique pour l'évaluation du coût moyen d'un algorithme. Leur intérêt tient en ce qu'elles permettent de stocker une suite de nombres dans les coefficients du développement en série d'une fonction de la variable complexe possédant de bonnes propriétés de dérivabilité dans des domaines bien choisis. Leur utilisation habituelle suit le schéma suivant : on construit de proche en proche la série génératrice de dénombrement par la méthode des constructeurs admissibles. Cette étape est accomplie par l'Analyseur Algébrique. On interprète cette série comme une fonction analytique, puis par analyse de singularités ou par utilisation directe du théorème de Cauchy on dérive le comportement asymptotique des coefficients de Taylor. Le but de ce travail est d'automatiser cette dérivation. Le programme, écrit en Maple, prend en argument l'expression formelle d'une fonction génératrice et retourne le comportement des coefficients de son développement lorsque leur rang tend vers l'infini.

[1] Laboratoire d'informatique de l'Ecole Polytechnique 91128 Palaiseau Cédex et INRIA Rocquencourt

Table des matières

Chapitre 1. Fonctions génératrices

I. Définitions 5

II. Fonctions génératrices en combinatoire 5

III. Comportements asymptotiques des coefficients 6

1. Fonctions à singularités algébrico-logarithmiques 6
   a) Théorème de transfert de O *6*
   b) Le théorème de Jüngen *6*
   c) Transfert d'une échelle étendue *7*
2. Fonctions entières et à singularités essentielles 8
   a) La méthode du point col *8*
   b) Les fonctions H-admissibles *9*
   c) Les fonctions HS-admissibles *10*
   d) L'apport de Wyman *11*
   e) D'autres résultats *13*
3. Conclusion 13

Chapitre 2. Calcul Automatique

I. Localisation des singularités 14

1. La fonction solve de Maple 15
2. Les polynômes: méthodes du calcul numérique 15
   a) La méthode de Bairstow *15*
   b) La méthode de Graeffe *18*
   c) Le théorème de Cauchy *20*
   d) La méthode de Schur-Cohn *20*
   e) Une transformation du plan *22*
   f) La méthode du résultant *22*
   g) Conclusion: le programme *22*
3. Les fonctions non polynomiales 23
   a) Cas général *23*
   b) En combinatoire *23*

II. Développement dans une échelle asymptotique étendue 24

1. Notations et définitions 24
2. Les L-fonctions 25
3. Présentation de l'échelle 25
4. Calcul automatique 26

III. Admissibilités 27

IV. La comparaison d'expressions 27
1. Cas général 27
2. Les nombres algébriques 27

Chapitre 3. Exemples d'applications

I. Le système $A\gamma\Omega$ 29
II. Nombres d'origine combinatoire 29
1. Fractions rationnelles 29
a) Les nombres de Fibonacci 29
b) Combinaisons 30
α. Sans répétitions 30
β. Avec répétitions 30
γ. Avec répétitions limitées 31
c) Compositions entières 31
α. En général 31
β. Sommants impairs 31
γ. Dénumérants 32
2. Fonctions à singularités algébriques 32
a) Les nombres de Bernoulli 32
b) Les nombres d'Euler 33
c) Permutations 33
α. Sans point fixe 33
β. Sans cycle 34
γ. Tirages à pile ou face 34
d) Dénombrements d'arbres 35
α. Les nombres de Catalan 35
β. Arbres unaires-binaires 36
e) Exemples plus compliqués 36
α. Des colliers de perles 36
β. Graphes fonctionnels binaires 37
γ. Les nuées 39
δ. Le problème de la ruine 40
ϵ. Les trains aléatoires 41
3. Fonctions à singularités logarithmiques 44
a) Rondes d'enfants 44
b) Nombres de Stirling de première espèce 45
4. Fonctions entières 46
a) La formule de Stirling 46
b) Nombre de Stirling de seconde espèce 46
c) Nombres de Bell (partitions) 47
III. Applications en informatique 49
1. En mesure du parallélisme 49
2. En complexité des systèmes de réécriture 51
a) Shuffle 52
b) Distributivité 53
3. En algorithmique 53
a) Dérivation formelle 53
b) Algorithmes de comparaison 54
c) Chaînes d'addition 55
d) Retour sur les trains aléatoires 58

Conclusion

Annexes

Annexe 1: Récapitulatif des comportements asymptotiques 61
Annexe 2: Les contours d'intégration utilisés 62
Annexe 3: $\Lambda\psi\Omega$: manuel de référence 63
Annexe 4: Plus près du programme 66
a) Installation 66
b) Utilisation courante 66
c) Le programme commenté 66

Bibliographie

Chapitre 1

Fonctions Génératrices

Chapitre 1

Fonctions génératrices

I. Définitions

Définition 1. *Si $u_n \in \mathbf{R}^I$, où $I \subset \mathbf{Z}$, la série génératrice ordinaire (s.g.o.) des u_n est la série:*

$$u(z) = \sum_{n \in I} u_n z^n.$$

On note $u_n = [z^n]u(z)$.

Définition 2. *Si $u_n \in \mathbf{R}^I$, où $I \subset \mathbf{N}$, la série génératrice exponentielle (f.g.e.) des u_n est la série:*

$$u(z) = \sum_{n \in I} u_n \frac{z^n}{n!}.$$

On note $u_n = [\frac{z^n}{n!}]u(z)$.

Il faut remarquer que ces séries peuvent très bien ne pas être définies du tout sur $\mathbf{C}$ lorsque leur rayon de convergence est nul. C'est par exemple le cas pour $u_n = (n!)^2$.

Remarque: On dit aussi fonction génératrice ordinaire (f.g.o.) ou exponentielle (f.g.e.).

Exemple: les nombres de Fibonacci

Il s'agit de la suite bien connue définie par

$$u_0 = u_1 = 1, u_{n+2} = u_{n+1} + u_n.$$

En multipliant par z^{n+2}, en sommant tout en faisant attention aux premiers termes, on obtient facilement:

$$u(z) = \frac{1}{1 - z - z^2}.$$

c'est de cette expression que nous déduirons automatiquement le comportement asymptotique des nombres de Fibonacci.

II. Fonctions génératrices en combinatoire

Dans ce cas les coefficients sont des réels positifs. De plus, une grande variété de fonctions génératrices est obtenue à partir de constructeurs "admissibles" c'est-à-dire de constructeurs sur les structures qui se traduisent directement en opérateurs sur les fonctions génératrices[Vitter-Flajolet87].

Une simple réflexion sur les coefficients donne les constructeurs admissibles suivants:

Fonctions génératrices ordinaires:

Union disjointe	$E = C \cup D$	$\longrightarrow$	$e(z) = c(z) + d(z)$
Produit cartésien	$E = C \times D$	$\longrightarrow$	$e(z) = c(z)d(z)$
Suite-finie-de	$E = \bigcup_{n \in \mathbf{N}} C^n$	$\longrightarrow$	$e(z) = (1 - c(z))^{-1}$
Ensemble-fini-de	$E = 2^C$	$\longrightarrow$	$e(z) = \exp(c(z) - \frac{c(z^2)}{2} + \frac{c(z^3)}{3} \ldots)$
Substitution	$E = C[D]$	$\longrightarrow$	$e(z) = c(d(z))$

Fonctions génératrices exponentielles:

Union disjointe	$E = C \cup D \longrightarrow$	$e(z) = c(z) + d(z)$
Produit partitionnel	$E = C * D \longrightarrow$	$e(z) = c(z)d(z)$
Complexe partitionnel(P-suite)	$E = C^{<*>} \longrightarrow$	$e(z) = (1 - c(z))^{-1}$
Complexe partitionnel abélien(P-ensemble)	$E = C^{[*]} \longrightarrow$	$e(z) = \exp(c(z))$

Ces trois derniers distinguent entre tous les renumérotages possibles de la structure obtenue, le premier étant l'analogue du produit cartésien, le deuxième des suites finies et le dernier des ensembles finis.

Exemple: si l'on veut obtenir le nombre de façons d'écrire un nombre entier comme somme de nombres impairs, on écrira la f.g.o. de l'ensemble des nombres impairs:

$$I(z) = \frac{z}{1-z^2},$$

et le résultat est la construction ensemble-fini ou suite, selon que l'on veut distinguer l'ordre des sommants ou non. Ce choix mène aux deux f.g.o.:

$$S(z) = 1 + \frac{z}{1-z-z^2} \quad \text{et} \quad E(z) = \exp(I(z) - I(z^2)/2 + I(z^3)/3 \cdots)$$

III. Comportements asymptotiques des coefficients

Alors qu'une expression exacte des coefficients des fonctions génératrices risque de nécessiter la résolution de récurrences compliquées voire insolubles et peut ne donner qu'une idée assez vague du comportement de ces coefficients, il est souvent possible d'obtenir une information asymptotique relativement aisément. Les théorèmes généraux qui suivent permettent de relier cette information asymptotique à la position et à la nature de la ou des singularités de module minimal (non nul). L'idée générale est qu'un développement de la fonction de la forme:

$$f(z) = f_1(z) + f_2(z) + \cdots + f_k(z) + O(g(z))$$

entraîne sous des hypothèses raisonnables un développement homologue (un *transfert*) pour les coefficients:

$$[z^n]f(z) = [z^n]f_1(z) + [z^n]f_2(z) + \cdots + [z^n]f_k(z) + O([z^n]g(z)).$$

Il faut donc connaître une échelle asymptotique $(f_i)_{i \in I}$ et des fonctions g pour lesquelles on maîtrise les $[z^n]f_i(z)$ ainsi que $[z^n]g(z)$, et pour lesquelles les transferts sont possibles.

1. Fonctions à singularités algébrico-logarithmiques

Dans cette section et dans la suivante, on suppose que la singularité est en 1. Les autres cas s'y ramènent par rotation et homothétie. Dans le cas où il y a plusieurs singularités (toujours en nombre fini pour les fonctions que nous avons considérées) sur le cercle de convergence, on obtiendra le résultat en sommant les développements partiels provenant de chacune d'elles.

Les singularités algébrico-logarithmiques sont celles en lesquelles la fonction s'écrit:

$$f(z) = (1-z)^{\alpha} \ln^{\beta}(\frac{1}{1-z}).$$

Ces fonctions sont traitées avec beaucoup de détail dans [Flajolet-Odlyzko88] auquel le lecteur intéressé par les démonstrations est prié de se référer.

a) Théorème de transfert de O

Théorème. *Soit f analytique sauf en 1 dans $\Delta(\phi, \eta) = \{z / |z| \leq 1 + \eta, |arg(z-1)| > \phi\}$ avec $\eta > 0$ et $0 < \phi < \pi/2$. Si lorsque z tend vers 1 dans Δ,*

$$f(z) = O\big((1-z)^{\alpha} L(\frac{1}{1-z})\big) \quad \text{où} \quad L(u) = (\ln u)^{\gamma}(\ln \ln u)^{\delta}$$

pour α, γ, δ réels, alors

$$[z^n]f(z) = O(n^{\alpha-1}L(n)).$$

b) Le théorème de Jüngen

En se limitant à $\beta \in N$, on dispose d'un théorème dû à Jüngen [Jüngen31]:

Théorème. *Le développement*

$$\sum_{n=0}^{\infty} a_n z^n = (1-z)^{-s} \log^k \frac{1}{1-z} \quad \begin{cases} k \in N \\ s \in C \end{cases}$$

est donné par les formules suivantes:

I $s \neq 0, -1, -2, \ldots$

$$a_n = \frac{n^{s-1}}{\Gamma(s)}\left[(\log n)^k \phi_0(n) + (\log n)^{k-1}\phi_1(n) + \cdots + \phi_k(n)\right]$$

$$\text{où} \quad \begin{cases} \phi_0(n) \sim 1 + \frac{c_{01}}{n} + \frac{c_{02}}{n^2} + \cdots \\ \phi_1(n) \sim c_{10} + \frac{c_{11}}{n} + \frac{c_{12}}{n^2} + \cdots \\ \vdots \end{cases}$$

II $s = 0, -1, -2, \ldots, k \in N^*$.

$$a_n = (-1)^s k\Gamma(1-s) n^{s-1}\left[(\log n)^{k-1}\phi_0(n) + (\log n)^{k-2}\phi_1(n) + \cdots + \phi_{k-1}(n)\right]$$

avec les mêmes informations sur les ϕ_i, *le développement de* $\phi_0(n)$ *étant alors convergent et même, pour* $s = 0$, $\phi_0(n) = 1$.

III $s = 0, -1, -2, \ldots$ *et* $k = 0$

$$a_n = 0 \qquad \textit{pour} \quad n > -s \quad \textit{(polynôme)}$$

Démonstration: Ce résultat s'obtient de manière constructive en introduisant de nombreuses fonctions intermédiaires qui permettent de ne pas alourdir les calculs. Le détail est donc présent dans le cœur de la procédure (voir en annexe), et voici la méthode dépouillée:
On part de

$$(1-z)^{-s}\Gamma(s) = \sum_{n=0}^{\infty} \frac{\Gamma(s+n)}{n!} z^n$$

que l'on différencie k fois *par rapport à s*. On obtient ainsi un système linéaire triangulaire de $k+1$ équations en

$$(1-z)^{-s}, (1-z)^{-s}\log\frac{1}{1-z}, \ldots, (1-z)^{-s}\log^k\frac{1}{1-z}.$$

Sa résolution donne donc une expression linéaire dont on déduit le coefficient

$$a_n = \frac{1}{\Gamma(s)n!}\left[\Gamma^{(k)}(s+n) + d_1\Gamma(k-1)(s+n) + \cdots + d_k\Gamma(s+n)\right] \qquad (E)$$

On étudie alors $\Gamma^{(\nu)}(s+n)/\Gamma(n+1)$ en partant de la formule de Stirling:

$$\Gamma(z) = \left(\frac{z}{e}\right)^z z^{-1/2}\sqrt{2\pi}\exp\left[\sum_{k=1}^{\infty}\frac{1}{z^{2k-1}}\frac{B_{2k}}{(2k)(2k-1)}\right]$$

où les B_k sont les nombres de Bernoulli. On en déduit

$$\Gamma^{(\nu)}(z) = \left(\frac{z}{e}\right)^z z^{-1/2}\left[\log^\nu(z)\psi(z) + \log^{\nu-1}(z)\psi_1(z) + \cdots + \psi_\nu(z)\right]$$

donc

$$\frac{\Gamma^{(\nu)}(n+s)}{\Gamma(n+1)} = \left(\frac{n}{e}\right)^{s-1}\frac{(1+s/n)^{n+s-1/2}}{(1+1/n)^{n+1/2}}\frac{\log^\nu(n+s)\psi(n+s) + \cdots + \psi_\nu(n+s)}{\psi(n+1)}$$

$$= n^{s-1}\left[\log^\nu(n)\Psi_0(n) + \cdots + \Psi_\nu(n)\right]$$

d'où la formule finale en reportant dans (E).

Il ne reste plus qu'à traiter $s = 0, -1, -2, \ldots$

$$\log^k\frac{1}{1-z} = k\int(1-z)^{-1}\log^{k-1}\frac{1}{1-z}dz$$

donne le développement pour $s = 0$, puis par récurrence

$$(1-z)^{-s}\log^k\frac{1}{1-z} = s\int(1-z)^{-(s+1)}\log^k\frac{1}{1-z}dz + k\int(1-z)^{-(s+1)}\log^{k-1}\frac{1}{1-z}dz.$$

c) Transfert d'une échelle étendue

Toujours dans [Flajolet-Odlyzko88], un théorème qui traite du cas β non entier:

Théorème. *Pour α et β non entiers, et*

$$f(z) = (1-z)^{\alpha}\left(\frac{1}{z}\ln\frac{1}{1-z}\right)^{\beta},$$

on a:

$$[z^n]f(z) \sim \frac{n^{-\alpha-1}}{\Gamma(-\alpha)}(\ln n)^{\beta}\left(1+\sum_{k>0}\frac{e_k^{(\alpha,\beta)}}{\ln^k n}\right)$$

$$\text{où} \quad e_k^{(\alpha,\beta)} = \Gamma(-\alpha)\frac{d^k}{ds^k}\left(\frac{1}{\Gamma(-s)}\right)\Big|_{s=\alpha}\binom{k-\beta-1}{k}.$$

Remarque: les trois théorèmes précédents contiennent le fameux théorème de Darboux [Darboux1878] comme cas particulier en prenant $\beta = 0$.

2. Fonctions entières et à singularités essentielles

Ces deux classes de fonctions se traitent par le même type de méthodes, toutes dérivées de la méthode du point col.

a) La méthode du point col

Cette méthode s'applique à des fonctions entières à croissance au moins exponentielle à l'infini, ainsi qu'aux singularités essentielles du type

$$\exp\left(\frac{1}{1-z}\right)$$

ou plus. Il s'agit, comme pour les théorèmes de transfert, d'utiliser une conséquence du théorème de Cauchy:

$$[z^n]f(z) = \frac{1}{2i\pi}\int_{|z|=R}\frac{f(z)}{z^{n+1}}dz \qquad \forall\, 0 < R < R_0$$

où R_0 est le rayon de convergence. La méthode du point col va donc se dérouler ainsi:

α) Soit $h_n(z) = \log f(z) - (n+1)\log(z)$

$$\text{on a alors} \quad [z^n]f(z) = I = \frac{1}{2i\pi}\int_{\Gamma} e^{h_n(z)}dz.$$

β) On trouve R_n tel que

$$\left(\frac{dh_n(z)}{dz}\right)\Big|_{z=R_n} = 0.$$

γ) On prouve l'existence d'un angle θ tel que

- $\int_{|z|=r,|\arg(z)|>\theta} e^{h_n(z)}dz \ll \int_{|z|=r} e^{h_n(z)}dz$
- $e^{h(z)} \sim \exp(h_n(R_n) + \frac{1}{2}h_n''(R_n)(z-R_n)^2)$ pour $|z| = r, |\arg(z)| \leq \theta$

δ) Alors

$$I \sim \frac{1}{2i\pi}\int_{|z|=r,|\arg(z)|\leq\theta]} \exp(h_n(R_n) + \frac{1}{2}h_n''(R_n)(z-R_n)^2)dz,$$

et après avoir posé $z = R_n + it$, si on peut démontrer que

$$I \sim \int_{-\infty}^{+\infty} e^{-t^2/2h_n''(R_n)}dt,$$

on obtient finalement

$$[z^n]f(z) \sim \frac{e^{h_n(R_n)}}{\sqrt{2\pi h_n''(R_n)}}.$$

Exemple: la fonction génératrice exponentielle qui compte le nombre d'entiers de valeur n, et qui peut servir de base à des constructions plus compliquées est la fonction exponentielle. On retrouve alors la formule de Stirling par la méthode du point col:

$$h_n(z) = z - (n+1)\ln z$$

$$R_n = \frac{1}{n+1} \sim \frac{1}{n}$$

$$[z^n]\exp(z) = \frac{1}{n!} \sim \frac{e^n}{n^n\sqrt{2\pi n}}.$$

L'inconvénient majeur de cette méthode tient en ce que les hypothèses nécessaires sont d'une vérification difficilement automatisable. De plus, le résultat ne donne qu'un équivalent des coefficients, alors qu'il est souvent possible d'obtenir un développement.

Heureusement, quelques auteurs se sont penchés sur la question, et ont donné des classes de fonctions restreintes mais faciles à reconnaître pour lesquelles un résultat (parfois un développement) peut être obtenu automatiquement. Les sections suivantes passent en revue une partie de ces résultats.

b) Les fonctions H-admissibles.

H signifie Hayman, du nom de l'auteur qui a introduit cette notion. Nous ne présenterons pas ici les fonctions H-admissibles dans toute leur généralité, mais simplement celles qui sont des fonctions génératrices à coefficients positifs.

Soit R le rayon de convergence (éventuellement infini). On introduit les fonctions

$$a(r) = \frac{rf'(r)}{f(r)} \qquad \text{et} \qquad b(r) = ra'(r)$$

ainsi que le point r_n défini par:

$$a(r_n) = n.$$

(On retrouve là le point R_{n-1} de la méthode du point selle, mais le point R_n donnerait exactement le même résultat.) Alors pour θ au voisinage de 0,

$$\log f(re^{i\theta}) = \log f(r) + i\theta a(r) - \frac{\theta^2}{2}b(r) + O(\theta^3),$$

et on impose les trois conditions suivantes:

1) $b(r) \longrightarrow +\infty$ pour $r \to R$

et pour une certaine fonction δ définie pour $r > R_0$ avec $0 < \delta(r) < \pi$,

2) $f(re^{i\theta}) \sim f(r)\exp(i\theta a(r) - 1/2\theta^2 b(r))$ pour $r \to R$ uniformément pour $|\theta| \leq \delta(r)$,

3)

$$f(re^{i\theta}) = \frac{o(f(r))}{\sqrt{b(r)}} \quad \text{pour} \quad r \to R$$

uniformément pour $\delta(r) \leq |\theta| \leq \pi$.

Une fonction vérifiant ces trois conditions est dite H-admissible. On a alors le résultat suivant:

Théorème. *Si $f = \sum a_n z^n$ est H-admissible, alors pour $n \to +\infty$,*

$$a_n \sim \frac{f(r_n)}{r_n^n\sqrt{2\pi b(r_n)}}.$$

La démonstration peut être trouvée dans [Hayman56]. En gros, les conditions permettant d'appliquer la méthode du point selle sont remplies et le calcul se fait bien.

Sous cette forme, la H-admissibilité ne serait pas très maniable, mais on dispose des trois théorèmes suivants qui permettent de définir récursivement une sous-classe intéressante des fonctions H-admissibles:

Théorème 1. *Si f et g sont H-admissibles, si P est un polynôme réel et si h est une fonction entière quelconque, alors sont aussi H-admissibles:*

$$\exp(f(z))$$
$$f(z)g(z)$$
$$f(z)+P(z)$$
$$P(f(z)) \quad \textit{si le coefficient dominant de } P \textit{ est positif}$$
$$f(z)P(z) \quad \textit{dans les mêmes conditions}$$
$$f(z)+h(z) \quad \textit{si pour} \quad \epsilon>0, max|h(re^{i\theta})| = O(f(r)^{1-\epsilon}) \quad \textit{quand} \quad r\to+\infty.$$

Ce qui signifie qu'une fonction H-admissible est assez grande pour imposer son comportement aux plus petites qu'elle. On dispose en outre d'un générateur de fonctions H-admissibles entières:

Théorème 2. *Si $P(z)=b_0+b_1z+\cdots+b_kz^k$, $k>0$ est un polynôme à coefficients réels et*

$$f(z)=\exp(P(z)) \qquad \textit{alors}$$

$$f\,H\text{-admissible} \Leftrightarrow \forall d\in N^*-\{1\}, \quad \begin{cases} 1)\exists m\in N, \begin{cases} d \textit{ ne divise pas } m \\ b_m\neq 0 \end{cases} \\ 2)\textit{Si } m(d) \textit{ est le plus grand tel entier,} \quad b_{m(d)}>0 \end{cases}$$

ainsi que d'une large classe de fonctions H-admissibles ayant une singularité à distance finie:

Théorème 3. *Avec α,β_1 des réels positifs, et $\beta_2,\beta_3,\ldots$ des réels quelconques,*

$$f(z)=\exp\Big[\beta_1(1-z)^{\alpha}\Big(\frac{1}{z}\ln\frac{1}{1-z}\Big)^{\beta_2}\Big(\frac{2}{z}\ln\Big(\frac{1}{z}\ln\frac{1}{1-z}\Big)\Big)^{\beta_3}\ldots\Big]$$

est H-admissible.

Ces trois derniers théorèmes nous permettent de déterminer assez facilement la H-admissibilité dans le cas des fonctions qui nous préoccupent. En outre, la section suivante apporte un nouveau progrès.

c) Les fonctions HS-admissibles.

Elles furent introduites par Harris et Schoenfeld qui, en rajoutant encore des conditions sur la fonction, obtiennent non plus un équivalent mais un développement asymptotique sur les coefficients.

Ici nous n'avons pas besoin de la HS-admissibilité dans toute sa généralité, mais simplement du résultat suivant [Odlyzko-Richmond85]:

Théorème. *Si $f(z)$ est une fonction H-admissible, $g(z)=\exp(f(z))$ est HS-admissible.*

Alors en introduisant

$$a(z)=\frac{f'(z)}{f(z)}, b_k(z)=\frac{z^k}{k!}a^{(k-1)}(z), b(z)=\frac{zb_1'(z)}{2},$$
$$u_n=b_1^{<-1>}(n+1), \beta_n=b(u_n)$$
$$\gamma_j(n)=-[b_{j+2}(u_n)+\frac{(-1)^j}{j+2}b_1(u_n)]/b(u_n),$$
$$F_k(n)=\frac{(-1)^k}{\sqrt{\pi}}\sum_{m=1}^{2k}\frac{\Gamma(m+k+1/2)}{m!}\sum_{\substack{j_1+\cdots+j_m=2k\\ j_1,\ldots,j_m\geq 1}}\gamma_{j_1}(n)\ldots\gamma_{j_m}(n),$$

on a finalement pour tout N positif, et pour $n\to\infty$,

$$[z^n]f(z)\approx\frac{f(u_n)}{2u_n^n\sqrt{\pi\beta_n}}\Big[1+\sum_{k=1}^{N}F_k(n)\beta_n^{-k}+o(\beta_n^{-N})\Big]$$

d) L'apport de Wyman.

Wyman introduit une classe de fonctions lui aussi, pour laquelle il est capable de déterminer un développement asymptotique des coefficients. Sa classe de fonctions contient en particulier toutes les fonctions du type:

$$P(z)\exp(Q(z)+G(1/z))$$

où P et Q sont des polynômes et G une fonction entière. Il s'agit donc d'une classe de fonctions assez étendue puisque l'exponentielle par exemple en fait partie alors que jusque là on n'a que la H-admissibilité et donc un équivalent des coefficients et non un développement.

Le calcul de Wyman est en fait un renforcement de la méthode du point col. Tout d'abord, l'intégrale de Cauchy peut très bien ne pas être concentrée au voisinage d'un seul point du cercle (exemple: $\exp(1/(1-x^2))$). Wyman considère donc la fonction module, dont les extrema à rayon fixé lorsque le rayon croît vers celui de la singularité parcourent des chemins qui portent les points cols à considérer. Ensuite, l'angle à prendre en compte au voisinage des points cols est raffiné. On ne souhaite plus se contenter d'un équivalent, mais on cherche un développement. Il faut donc que les parties négligées du contour d'intégration soient équivalentes à 0 dans une échelle asymptotique à préciser.

La méthode de Wyman se déroule finalement ainsi.
Soit $f(z)$ développable pour $0 \leq a \leq |z| < b \leq \infty$ sous la forme

$$f(z) = \sum_{-\infty}^{\infty} a_n z^n, a_n \in \mathbf{R},$$

et telle que, en posant $M(r,\theta) = |f(r\,e^{i\theta})|$,

$$\text{H1} \quad \max_{\theta\in[0\,2\pi]} M(r,\theta) \to \infty \quad \text{quand } r \to b.$$

Etape 1: On résout $\partial M/\partial\theta = 0$ en r, ce qui nous donne un certain nombre de solutions continues que l'on note sous forme polaire $\theta_k(r)$, avec leur module $M_k(r)$.

On choisit alors ceux de ces chemins qui vont apporter la plus grande contribution à l'intégrale. Pour ceci on commence par privilégier un chemin principal pour lequel on fait une hypothèse:
(H2) Pour r suffisamment grand,

$$\exists\theta_1(r), \quad \frac{M(r,\theta)}{M_1(r)} \text{ est uniformément borné en } \theta.$$

Les comparaisons entre les différentes parties de l'intégrale vont se faire dans l'échelle $(\ln M_1(r))^{-k}$, où k varie sur $\mathbf{N}$. Les chemins significatifs sont ceux des $\theta_k(r)$ pour lesquels on n'a pas pour r suffisamment grand

$$\frac{M_k(r)}{M_1(r)} \approx 0$$

Cette notation signifiant

$$\frac{M_k(r)}{M_1(r)} = O((\ln M_1(r))^{-n}) \quad \forall n \in \mathbf{N}.$$

Pour pouvoir poursuivre le calcul, Wyman demande aussi:
(H3) Pour r suffisamment grand:

- le nombre de chemins significatifs est constant,
- tout chemin significatif a une limite lorsque le rayon tend vers b et deux limites sont toujours différentes,
- le long d'un chemin significatif, $\partial^2 M/\partial\theta^2 < 0$.

On numérote les chemins significatifs L_k de 1 à N, L_k correspondant à $\theta_k(r)$.
Etape 2: On calcule alors les N points cols: $Z_k(n)$ est le point de L_k solution de

$$z\frac{f'(z)}{f(z)} = n,$$

dont Wyman justifie l'existence et l'unicité. On note $z_k(n) = L_k \cap \{r = |Z_1(n)|\}$. Alors pour pouvoir terminer le calcul, Wyman a besoin d'une hypothèse supplémentaire:
(H4) Pour r suffisamment grand,

- $\exists p, p_1, p_2 > 0$ tels que

$$p_1(\ln M_1(r)) \leq r\frac{f'(r)}{f(r)} \leq p_2(\ln M_1(r))^{1+p},$$

- $Z_k(n)/z_k(n) = 1 + O(1/n)$,
- dans le w-plan complexe, il existe un voisinage fixe $|w| \leq h$ dans lequel les

$$\ln f(Z_k(n)\exp(\frac{w}{n}\ln M_1(|Z_1(n)|)))$$

sont des fonctions régulières de w. De plus

$$\lim_{n\to\infty} \frac{\ln f(Z_k(n)\exp[\frac{w}{n}\ln M_1(|Z_1(n)|)]) - \ln f(Z_k(n))}{\ln M_1(|Z_1(n)|)}$$

existe uniformément en w pour $|w| \leq h$, on note cette limite $g_k(w)$ et il faut $\mathcal{R}e\, g"_k(0) \neq 0$.

Une fonction vérifiant les hypothèses H1,H2,H3,H4 est dite W-admissible, et on a:

Théorème . *Les coefficients a_n d'une fonction W-admissible f sont donnés par:*

$$a_n \approx \sum_{k=1}^{N} \frac{f(Z_k(n))}{Z_k^n(n)\sqrt{2\pi H^2 \ln f(Z_k(n))}}[1 + \sum_{m=1}^{\infty} \frac{1}{\sqrt{\pi}(\ln M_1(|Z_1(n)|))^m} \int_{-\infty}^{+\infty} b_{2m,k}(\phi)e^{-\phi^2}d\phi]$$

où $H = z\frac{d}{dz}$ et

$$b_{m,k}(\phi) = [u^m]\exp(\sum_{j=1}^{\infty} u^j \frac{[\ln M_1(|Z_1(n)|)]^{j/2} H^{j+2}\ln f(Z_k(n))(i\phi)^{j+2}}{(j+2)!(\frac{1}{2}H^2\ln f(Z_k(n)))^{\frac{j+2}{2}}})$$

Exemple: la fonction exponentielle.

Le seul chemin admissible est clairement le demi-axe réel positif. On a alors

$$z_1(n) = Z_1(n) = n$$

pour l'hypothèse (H4), on a:

- $r \leq r \leq r^{1+p}$
- $1 = 1$
- $w \mapsto n\exp w$ régulière

$$g_1(w) = \exp(w) - 1.$$

Donc la fonction est W-admissible. Alors quelque soit l'ordre désiré pour le développement de Stirling, il est facile de l'obtenir par le théorème.

Toutes les hypothèses faites sont malheureusement difficiles à vérifier automatiquement, mais un petit calcul donne des résultats plus programmables. En particulier, sont W-admissibles toutes les fonctions:

$$f(z) = \exp[\beta_1(1-z)^{\alpha}(\frac{1}{z}\ln\frac{1}{1-z})^{\beta_2}(\frac{2}{z}\ln(\frac{1}{z}\ln\frac{1}{1-z}))^{\beta_3}\ldots],$$

avec α, β_1 des réels positifs, et $\beta_2, \beta_3, \ldots$ des réels quelconques.

e) D'autres résultats.

On sait d'après des calculs dus à Macintyre et Wilson [Macintyre-Wilson54] déterminer des équivalents aux coefficients des fonctions:

$$(1-z)^{\alpha}[\log(1-z)]^{\beta}\exp[\gamma/(1-z)].$$

où α et β sont des réels positifs et γ un complexe quelconque.

On peut également traiter les fonctions

$$\exp[P(\frac{1}{1-z})]$$

avec P pseudo-polynôme quelconque (les puissances sont réelles).[Wright49]

D'autres auteurs [Häusler], [Grosswald] ont également apporté des contributions sur la question, mais souvent ces résultats compliqués ne présentent que peu d'intérêt dans le cas des fonctions génératrices issues de la combinatoire et n'ont donc pas été implantés bien qu'il n'y ait là aucune difficulté.

Il est de toute manière clair que la bonne approche de ce problème ne consiste pas à programmer un patchwork de résultats fragmentaires, mais bien plutôt à attendre un résultat plus général que l'avancement de la théorie permet d'espérer, tout en souhaitant que ce résultat ait la forme la plus programmable possible.

3. Conclusion

Les nombreux résultats présentés masquent un peu la réalité. En effet, même s'il est légitime de souhaiter un théorème général, même si certaines applications combinatoires peuvent amener à utiliser des méthodes de point col sophistiquées, la plupart des applications se réduisent en fait à l'utilisation du théorème de Darboux, c'est-à-dire à des singularités purement algébriques.

Chapitre 2

Calcul Automatique

Chapitre 2

Calcul automatique

La section précédente présentait l'ensemble des théorèmes que nous souhaitons automatiser. Nous allons maintenant examiner les différentes étapes qui mènent à une automatisation de ces calculs en essayant de préciser au maximum la classe de fonctions pour laquelle une automatisation totale est possible avec les méthodes choisies ainsi que les directions à creuser pour étendre cette classe.

Tout d'abord, il nous faut déterminer les singularités de module minimal des fonctions étudiées. Ceci suppose la résolution de polynômes, ainsi qu'un codage approprié pour les nombres algébriques, puis une généralisation approximative de ces méthodes pour des fonctions construites avec le logarithme et l'exponentielle. Une attention spéciale sera cependant apportée aux fonctions issues de la combinatoire afin d'en inclure le plus possible dans la classe résolue.

Ensuite nous parlerons du développement de la fonction au voisinage de ses singularités, de l'échelle asymptotique, du problème de la détermination analytique sur laquelle le calcul doit se faire, tout en tenant compte de la façon dont les singularités sont codées.

Enfin, plus facile, nous verrons comment il est possible de déterminer automatiquement l'appartenance d'une fonction aux classes de fonctions admissibles vues plus haut.

I. Localisation des singularités

D'après ce que nous venons de voir, la première chose à faire est de déterminer l'ensemble des singularités non nulles de module minimal. Si cet ensemble est vide, on laisse la partie *Admissibilités* qui va suivre s'occuper de la fonction. Sinon, il faut amener la fonction dans une forme permettant l'application d'un des théorèmes, et en déduire le comportement asymptotique des coefficients.

Tout d'abord, il faut décider de l'ensemble des fonctions que l'on essaie de traiter. La forme des théorèmes précédents suggère d'elle-même l'ensemble d'expressions E suivant:

$$x \in E \Longleftrightarrow \begin{cases} x \in \mathrm{R}[X] \\ \text{ou} \quad x = y \mp z \quad (y,z) \in E^2 \\ \text{ou} \quad x = yz \quad (y,z) \in E^2 \\ \text{ou} \quad x = \ln(y) \quad y \in E \\ \text{ou} \quad x = \exp(y) \quad y \in E \end{cases}.$$

Bien évidemment, on ne sait pas traiter tout cet ensemble, mais syntaxiquement, on est bien obligé d'en accepter a priori les éléments, quitte à les rejeter après un examen plus approfondi. Le problème est donc le suivant: lors de l'étude de l'arbre qui représente l'expression formelle d'un élément de E, quelles sont les caractéristiques qui vont déterminer l'existence de singularités ?

On distingue alors les différents cas pour lesquels on a besoin de connaître les singularités ou les zéros des opérandes de l'expression, qu'elle soit somme, produit ou fonction. En appelant $\mathcal{S}(f)$ l'ensemble des singularités de f et $\mathcal{Z}(f)$ l'ensemble de ses zéros, il suffit d'appliquer les règles simples suivantes:

$$\mathcal{S}(P) = \emptyset \quad \text{pour } P \in \mathrm{C}[X],$$

$$\mathcal{S}(\exp(f)) = \mathcal{S}(f),$$

$$\mathcal{S}(f^k) = \mathcal{S}(f) \quad \text{pour } k \in \mathrm{N}^*,$$

$$\mathcal{S}(\ln(f)) \subset \mathcal{S}(f) \cup \mathcal{Z}(f),$$
$$\mathcal{S}(f+g) \subset \mathcal{S}(f) \cup \mathcal{S}(g),$$
$$\mathcal{S}(f \times g) \subset \mathcal{S}(f) \cup \mathcal{S}(g),$$
$$\mathcal{S}(f^\alpha) \subset \mathcal{S}(f) \cup \mathcal{Z}(f) \quad \text{pour } \alpha \notin \mathbf{N}.$$

Ces règles ramènent en fait la détermination des singularités à celle des zéros. Nous allons maintenant voir ce que fait Maple dans ce domaine, constater quelques insuffisances et montrer comment y remédier dans la plupart des cas.

1. La fonction solve de Maple

Il existe en Maple une fonction **solve** dont le rôle est de résoudre la plupart des équations solubles de manière exacte. Cependant, sauf pour les polynômes, les solutions que donne **solve** sont les solutions *réelles* de l'équation. Ceci est justifié par la difficulté qu'il y a à répondre à une question du genre:

$$\mathbf{solve}(\exp(\mathbf{x}) = \mathbf{1})$$

dans le plan complexe. En effet, la réponse souhaitée:

$$\{2i k\pi, k \in \mathbf{Z}\},$$

est non seulement peu manipulable, mais inexprimable en Maple où le typage des variables non affectées est impossible.

Il existe en outre une fonction **fsolve** chargée de donner *une* solution réelle en flottants de l'équation à résoudre, en utilisant une méthode combinée d'interpolation parabolique, de sécante et de dichotomie (décrite dans [Gonnet77]).

En ce qui concerne les polynômes, toutes les racines peuvent être obtenues lorsque le degré du polynôme est inférieur à cinq, plus quelques cas lorsque les coefficients du polynôme sont rationnels. Si l'on veut se contenter de racines en flottants, on peut obtenir par **fsolve** toutes celles qui sont réelles.

Ainsi, dans le cas général, on n'obtiendra des racines complexes que lorsque la résolution de l'équation passe par celle d'un polynôme de degré faible. Or les racines qui nous intéressent sont *toutes* les racines de plus faible module, que les expressions soient des polynômes ou non. Il nous faut donc voir comment la prise en compte de l'origine combinatoire de nos fonctions permet de résoudre la plupart des situations que nous rencontrons en se ramenant au cas réel.

2. Les polynômes: méthodes du calcul numérique

Il s'agit ici de régler le problème posé par les polynômes de fort degré. Bien entendu, la littérature sur ce sujet est abondante et donnera souvent satisfaction à qui souhaite déterminer *toutes* les racines de ces polynômes. Cependant, il n'existe pas de "boîte noire", c'est-à-dire de programme acceptant n'importe quel polynôme en entrée et donnant en sortie ses racines, chaque méthode ayant ses cas d'exception. En outre, le problème qui nous préoccupe ici est de trouver toutes les racines de plus faible module, ce qui signifie en particulier qu'une solution sous la forme d'un couple *(partie réelle, partie imaginaire)* ne nous intéresse pas telle quelle, mais qu'il est essentiel que nous puissions décider de l'égalité des modules de racines, même si ces modules ne sont connus que de manière approchée. De plus, puisque c'est le plus petit module uniquement que nous cherchons, il serait bon d'éviter le surcroît de travail que nécessite la recherche de toutes les racines.

Dans ce but, plusieurs méthodes sont donc à étudier et à comparer.

a) La méthode de Bairstow

Les méthodes itératives du type Newton, si elles ont l'avantage d'une convergence rapide, ont l'inconvénient de ne converger à coup sûr que lorsque la valeur initiale est suffisamment proche de la valeur recherchée. La méthode de Bairstow consiste à appliquer la méthode de Newton non pas sur la fonction directement mais sur le reste de la division du polynôme par un facteur quadratique:

$$\text{On part de} \quad \begin{cases} P_n(x) = a_0 x^n + a_1 x^{n-1} + \cdots + a_n \\ Q(x) = x^2 - sx + p \end{cases},$$

alors

$$P_n(x) = P_{n-2}(x)Q(x) + R(x)$$

avec

$$\begin{cases} P_{n-2}(x) = b_0 x^{n-2} + \cdots + b_{n-2} \\ R(x) = b_{n-1}(x-s) + b_n \end{cases},$$

et les coefficients sont donnés par:

$$\begin{cases} b_0 = a_0 \\ b_1 = a_1 + sb_0 \\ b_2 = a_2 - pb_0 + sb_1 \\ \quad \vdots \\ b_{n-2} = a_{n-2} - pb_{n-4} + sb_{n-3} \\ b_{n-1} = a_{n-1} - pb_{n-3} + sb_{n-2} \\ b_n = a_n - pb_{n-2} + sb_{n-1} \end{cases}$$

On aura obtenu une solution lorsque $b_n = b_{n-1} = 0$.
C'est ce qui amène à considérer la fonction de $\mathbf{R}^2$ dans lui-même:

$$F(s,p) = \begin{pmatrix} b_{n-1} \\ b_n \end{pmatrix},$$

dont on note $f(s,p)$ et $g(s,p)$ les coordonnées. On applique alors la formule de Newton sur F , qui en notant

$$x_n = \begin{pmatrix} b_{n-1} \\ b_n \end{pmatrix},$$

s'écrit ici: $x_{n+1} = x_n - F'^{(-1)}(x_n)F(x_n)$, ou encore:

$$\begin{cases} s_{n+1} = s_n + \frac{S}{\Delta} \\ p_{n+1} = p_n + \frac{P}{\Delta} \end{cases},$$

avec

$$\begin{cases} S = f\frac{\partial g}{\partial p} - g\frac{\partial f}{\partial p} \\ P = -f\frac{\partial g}{\partial s} + g\frac{\partial f}{\partial s} \\ \Delta = \frac{\partial f}{\partial s}\frac{\partial g}{\partial p} - \frac{\partial g}{\partial s}\frac{\partial f}{\partial p} \end{cases}$$

en posant alors

$$\begin{cases} \frac{\partial}{\partial s} b_q = c_{q-1} \quad q = 1, \ldots, n-1 \\ \frac{\partial}{\partial s} b_n = c_{n-1} + b_{n-1} \end{cases},$$

les c_i sont donnés par:

$$\begin{cases} c_0 = b_0 \\ c_1 = b_1 + sc_0 \\ \quad \vdots \\ c_q = b_q + sc_{q-1} - pc_{q-2} \\ \quad \vdots \\ c_{n-2} = b_{n-2} + sc_{n-3} - pc_{n-4} \\ c_{n-1} = sc_{n-2} - pc_{n-3} \end{cases}$$

on a d'autre part

$$\frac{\partial}{\partial p} b_q = -b_{q-2} - \frac{\partial}{\partial p} b_{q-2},$$

dont on peut déduire par récurrence:

$$\frac{\partial}{\partial p} b_q = -c_{q-2} \qquad q \geq 2.$$

D'où finalement les formules dont on se sert dans la pratique:

$$\begin{cases} S = -b_{n-1}c_{n-2} + b_n c_{n-3} \\ P = b_n c_{n-2} - b_{n-1}c_{n-1} \\ \Delta = c_{n-2}^2 - c_{n-1}c_{n-3} \end{cases}$$

Test d'arrêt.

Comme pour toute méthode itérative, il faut bien décider un jour d'arrêter le calcul, en décrétant que la précision atteinte est suffisante. En fait, on peut éviter d'agir de façon totalement arbitraire, et c'est ce que le paragraphe suivant va détailler.

Si $P(z) = (z - z_1)^{\nu_1} \cdots$, si α est une approximation de z_1, et si $\rho = |\alpha|$, on veut $|\alpha - z_1| < \varepsilon\rho$. On introduit le polynôme $T(z) = \sum_{i=0}^{n} |a_i| z^{n-i}$, alors

$$\forall \rho' \geq |\alpha|, \forall k \in \{1, \ldots, n\}, \qquad T^{(k)}(\rho') \geq |P^{(k)}(\alpha)|.$$

Or

$$\begin{aligned} T(\rho + 2\varepsilon\rho) - T(\rho + \varepsilon\rho) &= \varepsilon\rho T'(\rho + \varepsilon\rho) + \cdots + \frac{(\varepsilon\rho)^n}{n!} T^{(n)}(\rho + \varepsilon\rho) \\ &\geq \left| \frac{(\alpha - z_1)^{\nu_1}}{\nu_1!} P^{(\nu_1)}(z_1) \right| + \cdots + \left| \frac{(\alpha - z_1)^n}{n!} P^{(n)}(z_1) \right|, \\ &\geq |P(\alpha)| \end{aligned}$$

donc

$$|\alpha - z_1| < \varepsilon\rho \Rightarrow T(\rho + 2\varepsilon\rho) - T(\rho + \varepsilon\rho) \geq |R(\alpha)|,$$

où R est le reste de la division de $P(z)$ par $z - \alpha$.

Le polynôme étant à coefficients réels, après avoir appliqué la méthode de Bairstow, on peut se trouver dans trois situations:

- ⋆ deux racines conjuguées (ou $-a$ et $+a$, $a \in \mathbf{R}$)
- ⋆ deux racines réelles distinctes (sauf le cas déjà mentionné)
- ⋆ une racine réelle double.

Dans le premier et le dernier cas, en utilisant la réalité de $\alpha_1 + \alpha_2 = s$ et $\alpha_1\alpha_2 = p$, le test d'arrêt s'écrit:

$$T(\rho + 2\varepsilon\rho) - T(\rho + \varepsilon\rho) \geq |\alpha_1 R_1 + R_0| \quad \text{et} \quad T(\rho + 2\varepsilon\rho) - T(\rho + \varepsilon\rho) \geq |\alpha_2 R_1 + R_0|,$$

où $P(z) = (z^2 - sz + p)Q(z) + R_1 z + R_0$. D'où, en faisant le produit:

$$\left[T(\rho + 2\varepsilon\rho) - T(\rho + \varepsilon\rho)\right]^2 \geq \left|R_0^2 + sR_0R_1 + pR_1^2\right|.$$

Dans le deuxième cas, en posant $\rho = \min(|\alpha_1|, |\alpha_2|)$, avec $|\alpha_1| \leq |\alpha_2|$ par exemple, le test s'écrit:

$$T(\rho + 2\varepsilon\rho) - T(\rho + \varepsilon\rho) \geq |\alpha_1 R_1 + R_0|.$$

Les avantages de la méthode de Bairstow.

Le principal intérêt de cette méthode tient en ce qu'elle converge vers les racines de plus petit module lorsque la valeur initiale est (0,0)[Durand71]. De plus, contrairement à la méthode de Newton-Raphson dans le plan, le domaine dans lequel peut être pris la valeur initiale est très grand. En outre, cette méthode a l'avantage de la rapidité.

Ses inconvénients.

L'inconvénient essentiel de la méthode de Bairstow est de taille lorsque l'on souhaite programmer: elle ne converge pas toujours. On essaye donc dans la mesure du possible d'éviter les cas de non convergence qui sont:

- $\Delta = 0$
  - ⋆ à cause d'une racine multiple
  - ⋆ par accident au début (p.ex. $x^n - a = 0$)
- $s_2 = p_2 = 0$ (p.ex. $x^5 + x^4 + x^3 + x^2 + x + 1 = 0$ qui n'est qu'une forme déguisée du précédent)

Le premier inconvénient peut facilement être évité: il suffit de traiter le polynôme au préalable pour qu'il n'ait que des racines simples. Ce résultat peut être atteint en n'employant la méthode de Bairstow que sur le PGCD du polynôme et de son polynôme dérivé lorsque les coefficients sont connus de manière exacte. Cependant on souhaite également avoir l'ordre de la racine de plus petit module. On procède donc de la manière suivante:

◇ on calcule $R = PGCD(Q, Q')$

◇ on garde R en mémoire et on travaille avec P/R

◇ on applique alors la méthode de Bairstow

◇ on calcule le module des racines trouvées

◇ on arrête lorsque le rapport des modules de deux racines successives dépasse un seuil fixé au préalable.

On revient alors à $Q(x)$ dont on obtient les racines avec leur ordre de multiplicité.

b) La méthode de Graeffe

Contrairement à la méthode de Bairstow, la méthode de Graeffe converge dans tous les cas. En revanche, elle ne donne que les modules des racines (ainsi que leur ordre de multiplicité).

Cette méthode consiste à séparer au mieux les racines par élévation à la puissance 2^m. Une fois écrit le polynôme $P^{(m)}$ dont les racines sont les puissances $2^m èmes$ des racines cherchées, ses coefficients donnent les fonctions symétriques des racines. Ces fonctions symétriques, lorsque m augmente, sont de plus en plus dominées par les racines de plus grand module.

Lorsqu'il n'y a qu'une racine de module maximal, la somme des racines de $P^{(m)}$ apportera donc l'information suffisante pour obtenir la racine de module maximal de P, et sinon un calcul plus détaillé tenant compte des coefficients suivants permet à nouveau d'obtenir l'information.

Le lecteur intéressé pourra consulter [Bareiss60] pour plus de détail, et voici les principales étapes du raisonnement. Soit

$$P(x) = a_0 x^n + \cdots + a_n,$$

$$\text{avec} \quad \rho_1 \geq \rho_2 \geq \cdots \geq \rho_p \geq 0,$$

les modules des racines, et $\alpha_j = \rho_{k_j} e^{i\phi_j}$ les racines (k_j croissante). Le polynôme $P^{(m)}$ de racines $\alpha_k^{(m)} = \alpha_k^{2^m}$ s'écrit par récurrence:

$$P^{(m+1)}(x^2) = (-1)^n P^{(m)}(x) P^{(m)}(-x).$$

Soient $a_j^{(m)}$ les racines de ce polynôme. Les formules de Newton donnent:

$$a_j^{(m)} = (-1)^j a_0^{(m)} \sum_{\sigma \in S_n} (\rho_{k_{\sigma(1)}} \cdots \rho_{k_{\sigma(j)}})^{2^m} \exp\left[i 2^m (\varphi_{\sigma(1)} + \cdots + \varphi_{\sigma(j)})\right],$$

la somme portant sur C_n^j termes. L'idée est que dans ces sommes, la mise à la puissance va contribuer à accroître les écarts entre les modules et les plus faibles d'entre eux vont devenir négligeables devant les plus importants.

Remarque: Si $\alpha_p, \ldots, \alpha_{p+\nu}$ sont les racines de module ρ, alors $\alpha_p^{(m)} \cdots \alpha_{p+\nu}^{(m)} = \rho^{\nu 2^m}$ (car les coefficients de P sont réels).

Propriété. *Si ν_i est le nombre de racines de P de module ρ_i, et si $l = \nu_1 + \cdots + \nu_k$ ($k \in \{1, \ldots, p\}$) alors pour $m \to \infty$ on a:*

$$a_l^{(m)} \sim (-1)^l a_0^{(m)} \left(\rho_1^{\nu_1} \rho_2^{\nu_2} \cdots \rho_k^{\nu_k}\right)^{2^m}.$$

Corollaire. *Si ρ est le module de racines de p, alors il existe j et $j+\nu$ tels que, pour m suffisamment grand et avec une précision arbitraire*

$$a_{j+\nu}^{(m)} \approx (-1)^{\nu} a_j^{(m)} \rho^{\nu 2^m},$$

où ν est le nombre de racines de p de module ρ.

Il suffit donc de trouver les bons indices et ce corollaire permet d'en déduire les modules des racines. Le signe $\approx$ de ce corollaire demande à être précisé, c'est pourquoi on introduit la définition suivante:

Définition. $a_j^{(m)}$ *est un coefficient ϵ-pivot lorsque*

$$\left| \frac{a_j^{(m)^2}}{a_j^{(m+1)}} - (-1)^j \right| < \epsilon.$$

Intérêt: Pour m suffisamment grand $a_0^{(m)}, a_{\nu_1}^{(m)}, a_{\nu_1+\nu_2}^{(m)}, \ldots, a_n^{(m)}$ sont ϵ-pivots et vérifient la condition du corollaire. Il faut cependant faire attention car même les coefficients zéro-pivots ne vérifient pas tous la condition du corollaire.

En effet, soit $j \leq \nu_1$

$$\frac{a_j^{(m)^2}}{a_j^{(m+1)}} = (-1)^j \frac{\left[\sum \rho_1^{j2^m} \exp\left(i2^m(\varphi_1+\cdots+\varphi_{\nu_1})\right)\right]^2 (1+\delta_{j,\nu_1}^{(m)})^2}{\left[\sum \rho_1^{j2^{m+1}} \exp\left(i2^{m+1}(\varphi_1+\cdots+\varphi_{\nu_1})\right)\right] (1+\delta_{j,\nu_1}^{(m+1)})}.$$

$\star$ si $\nu_1 = j$, il n'y a pas de problème, le coefficient est pivot,

$\star$ si $\nu_1 > j$, on a deux sommes de $C_{\nu_1}^j$ termes et la condition des coefficients pivots s'écrit:

$$\left(\sum \exp\left(i2^m(\varphi_1+\cdots+\varphi_{\nu_1})\right)\right)^2 = \sum \exp\left(i2^{m+1}(\varphi_1+\cdots+\varphi_{\nu_1})\right),$$

et lorsque les φ_k sont des π/s, l'égalité n'a pas de mal à arriver par accident. On obtient alors un nombre qui est bien un module de racines mais qui n'est pas le plus grand.

Pour remédier à ce problème, nous aurons besoin d'un test pour vérifier qu'un module trouvé est bien maximal.

Choix des paramètres dans la méthode de Graeffe.

Un coefficient $\epsilon_j^{(M)}$-pivot doit vérifier

$$\left| \frac{a_j^{(M)^2}}{a_j^{(M+1)}} - (-1)^j \right| < \epsilon_j^{(M)}.$$

Le problème est: étant donné un facteur de séparation η (ce qui signifie que l'on considère deux modules de racines successifs comme étant différents à partir du moment où $\rho_{k+1} < (1-\eta)\rho_k$), comment choisir M le nombre d'itérations, et comment choisir $\epsilon_j^{(M)}$?

Pour répondre à cette question, on utilise la condition sous la forme:

$$\frac{(s_j^{(M-1)})^2}{s_j^{(M)}} = 1 + \epsilon_j^{(M)},$$

où les s_j sont les sommes de Newton des racines.

Le choix de $\epsilon_j^{(M)}$: une étude du cas le pire donne l'inégalité:

$$\epsilon_j^{(M)} < \bar{\epsilon}_{n/2}^{(M)} \qquad \text{pour tout } j,$$

où:

$$\bar{\epsilon}_{n/2}^{(M)} = \frac{\left(\sum_{k=0}^{n/2} \binom{n/2}{k}^2 (1-\eta)^{k2^{M-1}}\right)^2}{\sum_{k=0}^{n/2} \binom{n/2}{k}^2 (1-\eta)^{k2^M}} - 1.$$

Nombre d'itérations: l'intérêt de minimiser M tient non seulement dans la diminution du temps de calcul, mais aussi dans l'amoindrissement de la propagation des erreurs d'arrondi. Une condition suffisante pour que l'erreur relative sur les modules soit majorée par ε est:

$$\frac{(1-\eta)^{2^m}}{2^{m-1}} < \varepsilon \frac{\sqrt{2\pi n}}{2^{n+1}} \nu,$$

formule permettant de déterminer M.

Précision des calculs: il faut tenir compte de la propagation des erreurs d'arrondi lors du calcul des $a_j^{(m)}$. Une étude du cas le pire donne:

$$H_{\max} \leq -\log_{10} \varepsilon + M \log_{10} \frac{2^{n+1}}{\sqrt{2\pi n}}.$$

Conclusion sur les paramètres dans la méthode de Graeffe: on procède finalement ainsi:

1) on choisit le facteur de séparation désiré η,
2) on en déduit M en prenant par exemple $\varepsilon = \eta/10$,
3) on calcule alors $\bar{\epsilon}_{n/2}^{(M)}$.

Remarque: on applique ici la méthode de Graeffe sur le polynôme $X^{\deg(P)}P(1/X)$, puisque c'est le plus *petit* module des racines qui est l'objet de notre quête.

c) Le théorème de Cauchy

Comme nous l'avons vu dans la section précédente, il nous faut une procédure pour tester si un module trouvé par la méthode de Graeffe est bien minimal, le théorème de Cauchy apparaît alors comme une solution naturelle. La conséquence de ce théorème qui nous intéresse s'énonce ainsi:

$$\frac{1}{2i\pi} \oint_{|z|=R} \frac{f'(z)}{f(z)} dz = Z(f,R) - P(f,R),$$

où Z est le nombre de zéros de f dans le cercle de rayon R et de centre l'origine et P est le nombre de pôles de f dans ce même cercle.

On note qu'en fait, pour des fonctions n'ayant pas de pôle, on en déduit directement une méthode de détermination des racines de plus petit module, en procédant par dichotomie.

Cependant, il nous faut modérer notre enthousiasme quant à ce resolveur très général. En effet, non seulement il ne s'applique pas dans la majorité des cas à des fonctions ayant des pôles, mais même sur des polynômes, la dichotomie qu'il nécessite rend un calcul précis d'une longueur prohibitive. De plus, alors qu'en théorie les discontinuités de l'intégrale sont brutales, les calculs numériques, avec les approximations qu'ils entraînent, montrent un résultat qui varie continûment lorsque l'on passe par les modules des zéros, et ceci suffisamment sensiblement pour que même les polynômes soient souvent insolubles par cette méthode.

d) La méthode de Schur-Cohn

Pour les polynômes à coefficients réels, il est possible de déterminer exactement le nombre de racines dans le disque unité, donc dans n'importe quel disque du plan. Le principe de la méthode est le suivant: à partir d'un polynôme P de degré n, on construit une suite de polynômes de degrés décroissants pour laquelle on

connaît une récurrence entre les nombres de zéros dans le disque unité $D(0,1)$ de deux polynômes consécutifs. La récurrence se termine lorsque l'on arrive sur un polynôme de degré zéro. Voici une présentation plus détaillée de la méthode pour des polynômes à coefficients réels. On part de

$$P(X) = P_0(X) = \sum_{i=0}^{n} a_i^{(0)} X^i,$$

et on définit le transformé de Schur

$$P_{k+1}(X) = a_0^{(k)} P_k(X) - a_{n-k}^{(k)} P_k^*(X)$$
$$\text{où} \quad P_k^*(X) = \sum a_{n-k-i}^{(k)} X^i.$$

Propriété: $d^\circ(P_k) \leq n-k$.

On note $\gamma_k = a_0^{(k)}$.

Lemme de Cohn. *Si P_k a n_k zéros dans $D(0,1)$ ouvert et si $\gamma_{k+1} \neq 0$ alors*

$$n_{k+1} = \begin{cases} n_k & si\ \gamma_k > 0 \\ n-k-n_k & si\ \gamma_k < 0 \end{cases}$$

Démonstration:

$$\gamma_{k+1} = |a_0^{(k)}|^2 - |a_{n-k}^{(k)}|^2$$

lorsque $\gamma_{k+1} <> 0$, on a donc:

$$sign(|a_0^{(k)} P_k(e^{i\theta})| - |a_{n-k}^{(k)} P_k^*(e^{i\theta})|) = sign(\gamma_{k+1}) \quad \forall \theta.$$

Donc, d'après le théorème de Rouché, lorsque γ_{k+1} est positif (*resp. négatif*), le polynôme P_{k+1} a autant de racines dans le disque unité que le polynôme P_k (*resp.* P_k^*). Et les racines de P_k^* étant les inverses des racines de P_k, on a bien le résultat annoncé.

Si $\gamma_{k+1} = 0$, on distingue alors deux cas selon que P_{k+1} est nul ou non.

Si P_{k+1} est nul, c'est que:

$$a_i^{(k)} a_0^{(k)} = a_{n-k}^{(k)} a_{n-k-i}^{(k)} \quad \forall i \in \{0..n-k\}.$$

En particulier,

$$a_0^{(k)} = \epsilon a_{n-k}^{(k)} \quad \epsilon \in \{-1,+1\}$$
$$a_i^{(k)} = \epsilon a_{n-k-i}^{(k)}.$$

Un tel polynôme est dit autoréciproque et on a alors:

Théorème de Cohn. $P(X) = \sum_{i=0}^{m} b_i X^i$ *autoréciproque a le même nombre de zéros avec ordre de multiplicité dans le disque unité ouvert que:*

$$\tilde{P}(X) = \sum_{i=0}^{m-1} (m-i) b_{m-i} X^i.$$

Démonstration: voir [Marden49]

Si P_{k+1} est non nul, on a quand même un certain nombre de relations

$$a_{n-k-i}^{(k)} = \epsilon a_i^{(k)},$$

on note alors q le premier i pour lequel cette égalité ne se produit pas, et on multiplie le polynôme par un polynôme ayant des racines de module supérieur à 1:

$$G(z) = (z^q + 2sign((a_{n-k-q}^{(k)} - \epsilon a_q^{(k)})/a_{n-k}^{(k)}))P_k(z).$$

On note que le transformé de Schur de G a pour degré $n-k$ et un terme constant positif, ce qui nous assure que le processus termine et que le nombre de racines dans le disque unité est conservé.

e) Une transformation du plan.

Voici une autre méthode pour déterminer les racines dont on ne connaît que le module. On effectue un changement de variable transformant le cercle sur lequel sont les racines cherchées en une droite. Il se trouve que le changement de variable choisi se traduit de manière agréable au niveau du polynôme, d'où l'intérêt de la méthode. Voici le détail de cette transformation:

$$\begin{aligned} z &= \rho e^{i\theta} \\ &= \rho \frac{e^{i\theta/2}}{e^{-i\theta/2}} \\ &= \rho \frac{\cos(\theta/2) + i\sin(\theta/2)}{\cos(\theta/2) - i\sin(\theta/2)}, \\ &= \rho \frac{1+iZ}{1-iZ} \end{aligned}$$

avec $Z = \mathrm{tg}(\theta/2)$, $-\pi < \theta < \pi$. On a donc

$$(1-iZ)^n P_n(\rho\frac{1+iZ}{1-iZ}) = A(Z) + iB(Z),$$

et A ainsi que B sont de degré n en Z. De plus, et c'est là que le miracle se produit, selon la parité de n, le résultat est de la forme $\Psi(Z^2) + iZ\Theta(Z^2)$ ou $Z\Psi(Z^2) + i\Theta(Z^2)$. On pose alors $U = Z^2$ *et le degré des polynômes est divisé par 2 !* (Tout simplement parce qu'une racine correspond à deux solutions conjuguées.) On cherche alors les racines réelles du PGCD de $\Psi(U)$ et $\Theta(U)$ qui correspondent forcément aux solutions du problème initial, ces dernières étant obtenues par la transformation inverse:

$$\Re(z) = \rho\frac{1-Z^2}{1+Z^2} \qquad \Im(z) = \pm\rho\frac{2Z}{1+Z^2}.$$

Les racines réelles sont quant à elles obtenues grâce à la remarque suivante:

Lorsque l'on forme $A(Z)$ et $B(Z)$,

$\star$ si les p termes de plus haut degré sont nuls, c'est que $-\rho$ est racine d'ordre p, la condition étant également nécessaire.

$\star$ si Z^q peut être mis en facteur dans A et$B$, c'est que ρ est racine d'ordre q, la condition étant également nécessaire.

Cette méthode, après essais, a été abandonnée au profit de la méthode du résultant bien plus stable numériquement.

f) La méthode du résultant.

Mettant à part le cas des racines réelles, cette méthode commence comme la méthode de Bairstow par une division du polynôme par un facteur quadratique inconnu. La différence est qu'ici, le terme constant du facteur quadratique est connu puisqu'il s'agit du carré du module des racines conjuguées cherchées. On obtient donc un polynôme en une variable dont on cherche les racines réelles (en fait pour des raisons de stabilité on se contente une fois encore de module de racines) et il ne reste plus qu'à avoir un bon test pour décider lesquelles des racines ainsi obtenues sont effectivement racines du polynôme initial. Or le test d'arrêt déjà vu pour la méthode de Bairstow est toujours valide et est donc réutilisé dans ce nouveau contexte.

g) Conclusion: le programme.

Il s'agit en fait d'une procédure qui manquait en Maple, et qui est équivalente à la procédure `allroots` de Macsyma.

Le programme se déroule finalement comme suit: on lance le `solve` de Maple pour voir s'il est possible de travailler avec des valeurs exactes, et lorsque ce n'est pas le cas, c'est-à-dire lorsque le nombre de racines obtenues est inférieur au degré du polynôme, on lance la méthode de Graeffe en lui demandant le plus petit module des racines, donc en lançant la méthode de Schur sur chaque module trouvé jusqu'à avoir le plus petit, ainsi que sa multiplicité. On teste alors l'existence de racines réelles, et si leur nombre est inférieur à la multiplicité du module, on calcule le résultant, que l'on résout par la méthode de Graeffe en calculant

cette fois-ci tous les modules donc sans utiliser la méthode de Schur et on conclut en appliquant le test vu pour la méthode de Bairstow.

Cette méthode, après l'essai d'autres combinaisons entre les différentes procédures classiques, nous apparaît comme la meilleure, en grande partie grâce à l'excellente étude qui en a été faite dans [Bareiss60] et [Bareiss65].

3. Les fonctions non polynomiales

a) Cas général

Sur la droite réelle, et lorsqu'un intervalle de départ est connu, il existe de bonnes méthodes de résolution, et elles sont utilisées en Maple par `fsolve`. Cependant, lorsqu'aucune valeur initiale n'est connue, la convergence des méthodes usuelles n'est jamais garantie. Quant au cas complexe, il ne faut pas espérer grand chose de la méthode de Newton-Raphson à moins d'avoir une bonne approximation initiale, et l'obtention de la racine de plus faible module est totalement illusoire, sauf peut-être par quelques méthodes très lentes et à convergence incertaine utilisant le développement en série de la fonction étudiée.

b) En combinatoire

Les fonctions que nous étudions ne sont pas complètement générales. Mais avant de développer les conséquences de leurs caractéristiques, précisons qu'il n'eut pas été bon de se réduire à ces fonctions pour tout le projet, car nous aurions alors exclu de notre champ d'étude un certain nombre de fonctions génératrices utiles comme celles des nombres de Bernoulli, d'Euler ou de Stirling de première espèce.

Voici les deux particularités des fonctions étudiées que nous allons essayer d'exploiter au maximum: d'une part les coefficients de leur développement en série à l'origine sont positifs, d'autre part elles sont combinaison un nombre fini de fois des fonctions exponentielle, quasi-inverse ($Q(x) = 1/(1-x)$), quasi-logarithme ($L(x) = \ln(1/(1-x))$) et des opérateurs de somme et de produit.

Tout d'abord des propriétés nous viennent de la positivité des coefficients: si $0 < R \leq \infty$ est le rayon de convergence de la fonction,

- la fonction est réelle croissante sur le demi-axe $[0, R]$,
- son module sur tout cercle de centre l'origine et de rayon r est maximal en $z = r$,
- $z = R$ est une singularité.

Ensuite un résultat qui nous vient de l'ensemble de fonctions étudié: si la fonction a plusieurs singularités sur un même cercle, alors leurs arguments sont commensurables à π. Ceci se démontre par récurrence sur la taille de l'expression de la fonction: si la propriété est vraie sur toutes les expressions de taille inférieure à n, on considère une expression de taille $n+1$ et on discute suivant son type:

- s'il s'agit d'une somme, ses singularités sont à chercher parmi celles de ses sommants, qui sont de taille inférieure, donc la propriété est vraie;
- s'il s'agit d'un produit, le même argument est valable;
- pour une exponentielle il est toujours valable;
- pour la fonction quasi-inverse, ses singularités sont ou bien celles de l'expression argument, auquel cas la récurrence s'applique, ou bien les points qui vérifient:

$$a(z) = 1.$$

Notant

$$a(z) = \sum_{n>=0} a_n z^n \quad a_n \geq 0,$$

l'existence de plusieurs racines sur un même cercle s'écrit

$$\sum a_n r^n = \sum a_n r^n e^{i\,n\theta}.$$

Comme les coefficients sont positifs, on a alors

$$(e^{i\,n\theta})^{pgcd(k,a_k \neq 0)} = 1,$$

ce qui est ce que l'on souhaite;

- pour la fonction quasi-log, le même argument s'applique.

Ceci signifie que la recherche des singularités se ramène à la recherche de zéros sur le demi-axe réel positif. Ce n'est pas encore suffisant pour les méthodes de résolution numérique habituelles, qui ont besoin d'un intervalle de départ, mais nous avons une propriété supplémentaire qui va nous donner une borne supérieure à cet intervalle:

Proposition. *Les opérateurs correspondant au constructeur suite-finie-de pour les fonctions génératrices ordinaires et aux constructeurs P-suite-de, P-ensemble-de et P-cycle-de pour les fonctions génératrices exponentielles, font décroître au sens large le rayon de convergence de la série génératrice des objets sur lesquels on les applique.*

Démonstration: les opérateurs considérés sont

$$Q(C(z)) = \frac{1}{1-C(z)}, L(C(z)) = \ln \frac{1}{1-C(z)} \text{ et } \exp(C(z)).$$

$C(z)$ est la fonction génératrice des objets sur lesquels on applique ces constructeurs, les coefficients de son développement sont donc positifs, ce qui entraîne la croissance de $C(z)$ sur le demi-axe réel positif.

On note $O(C(z))$ la fonction composée. Si $O = Q$ ou $O = L$, les singularités sont à chercher parmi celles de C et parmi les points où C vaut 1, alors que si $O = \exp$, elles sont les singularités de C.

Soit r le rayon de convergence (fini sinon la propriété est évidente), deux cas peuvent se produire:
• $C(x)$ a une limite finie (toujours strictement positive) lorsque x tend vers r sur l'axe réel (p.ex. $C(z) = (1-\sqrt{1-z})/z$). Alors si $O = \exp$, le point r est singularité (on regarde le log), et la fonction n'en a pas de plus proche de l'origine, tandis que si $O = L$ ou $O = Q$, le point r est évidemment singularité, et la fonction en a une autre plus proche de l'origine dans le cas ou la limite de $C(x)$ en r est supérieure à 1.
• $C(x)$ tend vers l'infini lorsque x tend vers r sur l'axe réel. Alors selon la position de sa valeur en zéro par rapport à 1, la fonction composée aura une singularité en r ou avant.

Ceci permet d'utiliser des procédures de calcul flottant pour résoudre les égalités à 1 avec la certitude d'avoir une racine unique dans un intervalle connu entre les bornes duquel on a un changement de signe de la fonction, et ces conditions sont propres à satisfaire les conditions d'entrée de la plupart des resolveurs numériques et de celui de Maple en particulier.

II. Développement dans une échelle asymptotique étendue

La fonction `taylor` de Maple est capable de donner un développement d'une fonction au voisinage d'un point de la forme:

$$\sum_{i \in I} \alpha_i (x-a)^{\frac{i}{q_i}}; \qquad I \subset \mathbf{Z},$$

ceci sans tester avec beaucoup de finesse la nullité des α_i.

Cependant, vu les théorèmes que nous avons l'espoir de rendre effectifs, ce type de développement est très insuffisant: il nous faut prendre en compte les logarithmes, et il nous faut absolument savoir si les coefficients du développement sont nuls ou non puisque cela peut déterminer la nature de la singularité (et même son existence en cas d'annulation).

1. Notations et définitions

Dans toute cette partie, nous nous conformons aux notations de Erdélyi et Wyman [Erdélyi-Wyman63], et utilisons une version réduite de la notion d'échelle asymptotique généralisée telle qu'elle est présentée dans [Wyman59]. La définition générale de Wyman, même si elle est utile à l'établissement du théorème de Wyman (Ch1,III2d), se réduit en fait à cette version pour la classe des fonctions que nous manipulons.

Définition 1: si f et ϕ sont deux fonctions de la variable x définies au voisinage d'un point $\alpha \in \mathbf{R} \cup \{-\infty, +\infty\}$ et si

$$f(x)/\phi(x) \to 1 \text{ quand } x \to \alpha,$$

alors on dit que f est équivalente à ϕ en α et l'on note

$$f \sim \phi,$$

en omettant de préciser α lorsque le contexte le permet.

Définition 2: une échelle asymptotique en α est un ensemble de fonctions $\{\phi_i\}_{i\in I}$ continues et positives au voisinage de α. Cet ensemble est totalement ordonné par

$$\phi_i < \phi_j \Leftrightarrow \lim_{x\to\alpha} \frac{\phi_i(x)}{\phi_j(x)} = 0,$$

ce qui signifie en particulier que la limite existe toujours. Un développement dans l'échelle $\{\phi_i\}_{i\in I}$ d'une fonction f définie au voisinage de α s'écrira:

$$f = \sum_{i\in J\subset I} \alpha_i\phi_i + O(\phi_m),$$

où $\alpha_i \in \mathbf{C}$ et $\phi_m < \phi_i$ pour tout i dans J.

Remarque: la définition généralisée de Wyman consiste à supprimer la contrainte $\alpha_i \in \mathbf{C}$ pour autoriser comme coefficients des fonctions dont la variation est "faible" par rapport à l'échelle choisie. On perd ainsi l'unicité du développement.

2. Les L-fonctions

Définition: une L-fonction (abréviation pour fonction logarithmico-exponentielle) est une fonction de la variable réelle, définie à partir d'une certaine valeur par une combinaison finie des symboles algébriques ordinaires $(+,-,\times,\div,()^{1/n})$, et des symboles fonctionnels log() et exp(), opérant sur une variable et sur des constantes réelles. Hardy fit une étude asymptotique quasi-exhaustive de ces L-fonctions qu'il avait introduites dans [Hardy10a].

Remarque 1: plus tard [Hardy10b], il étendit cette définition en y autorisant des fonctions algébriques implicites.

Remarque 2: on voit tout l'intérêt pour nous de cette définition d'une classe de fonctions dont on peut tester les éléments simplement d'après leurs expressions.

Définition: on appelle L-ordre d'une L-fonction le nombre maximum de symboles log et exp imbriqués dans l'expression de la fonction.

Exemple: $\exp(\exp(x))$, $\exp(\log(x))$, $\log(\log(x))$ sont de L-ordre 2.

Cette définition n'est pas valide en ce sens qu'elle dépend de l'expression de la fonction. Une définition plus stricte devrait demander un minimum sur toutes les expressions de la fonction. Cependant, cette définition a deux avantages: d'une part le L-ordre d'une L-fonction est aisément calculable, d'autre part, dans toute la théorie, les théorèmes faisant intervenir le L-ordre seront vrais pour tout L-ordre de la fonction, mais seront d'autant plus précis que l'expression choisie pour représenter la fonction aura un L-ordre plus faible.

En fait, nous ne nous servons que d'un théorème très simple, que nous avons programmé. Celui-ci relie le L-ordre d'une L-fonction à sa position dans une échelle essentielle:

Théorème . *Les L-fonctions sont ultimement continues et de signe constant. Elles ont une limite en $+\infty$, et lorsqu'une L-fonction de L-ordre p tend vers l'infini, on a:*

$$f = O(e_p(x)) \text{ et } l_p(x) = O(f),$$

où l_p désigne le logarithme itéré p fois et e_p l'exponentielle itérée p fois.

Ce théorème se démontre sans difficulté par récurrence sur l'ordre.

3. Présentation de l'échelle

Nous ne présentons ici que l'échelle en $+\infty$, le cas de développements au voisinage d'un point fini s'y ramène après changement de variable.

L'échelle choisie contient $\{\ldots l_p(x), l_{p-1}(x), \ldots x, \ldots e_p(x)\ldots\}$, ainsi que les puissances rationnelles de chacun de ces termes. En outre, les L-fonctions rencontrées lors d'un calcul et n'appartenant pas déjà à cette échelle y sont insérées en tenant compte de leur croissance à l'infini. Plus précisément, l'échelle initiale est:

$$\{e_p^\alpha(x)\}_{p\in\mathbf{Z},\alpha\in\mathbf{Q}},$$

en notant $e_{-1}(x) = l(x)$. Si lors d'un calcul on rencontre x^x, qui est en dehors de cette échelle, son L-ordre est calculé (en l'occurrence 2), et on le place par dichotomie dans l'intervalle $[l_2x, e_2x]$. A partir de ce moment et pour tout le reste de la session, on lui attache une étiquette comprise entre 1 et 2, (par exemple 3/2) qui permettra de le comparer aux autres éléments de l'échelle. Lors de l'arrivée de nouveaux éléments à croissance comprise entre e_1x et e_2x, il faudra réutiliser son expression x^x le temps de positionner ces nouveaux éléments, mais ceci uniquement si d'autres expressions de la même session le nécessitent et uniquement lors de leur première apparition. Par la suite, toutes les comparaisons sont faites sur les étiquettes associées aux expressions. Il s'agit en fait d'une mémo-fonction qui construit une relation d'ordre sur les objets en faisant le minimum de comparaisons.

Notation: lorsqu'un nouvel élément se voit associer une étiquette r que l'on peut toujours choisir dans $\mathbf{Q}$, on le note (et on le manipule de manière interne) sous la forme e_r.

Nous avons alors une échelle de la forme:

$$\{e_p^\alpha(x)\}_{p\in\mathbf{Q},\alpha\in\mathbf{Q}},$$

qui n'est pas encore tout à fait suffisante pour répondre à nos besoins. Un pas de plus nous amène à l'échelle finale, celle qui est programmée:

$$\{e_p^\alpha x.d_p x\}_{\alpha\in\mathbf{Q},p\in\mathbf{Q}},$$

où

$$d_p x = \sum e_q^\alpha x.d_q x \quad q < p.$$

Cette échelle n'a de sens que pour les L-fonctions puisque sa définition repose sur la finitude du L-ordre des expressions manipulées.

Définition: on appellera rang d'un élément de l'échelle et on notera $rg(\phi)$ la plus grande valeur des indices d'exponentielle intervenant dans son expression. On appellera terme principal $e_{rg(\phi)}x$.

Exemple: la formule de Stirling s'écrit, se manipule automatiquement, sous la forme:

$$\begin{aligned}(\frac{n}{e})^n\sqrt{2\pi n}(1+\frac{1}{12n}+\cdots) &= \sqrt{2\pi}.e_{3/2}n.d_1,\\ d_1 &= e_1^{-1}n.d_0,\\ d_0 &= e_0^{1/2}n+\frac{1}{12}e_0^{-1/2}n+\cdots,\end{aligned}$$

où le rang 3/2 dépend de la session, tout en restant dans l'intervalle $]1;2[$.

4. Calcul automatique

Il s'agit d'effectuer les opérations algébriques ordinaires $(+,-,\times,\div,()^{1/n})$, ainsi que l'exponentiation et le logarithme.

Les opérations algébriques ordinaires s'effectuent en considérant les développements comme des séries où les degrés sont rationnels, où la variable est le terme principal et où les coefficients sont des développements de rang inférieur. Ces opérations reçoivent ainsi une définition récursive dont la terminaison est assurée une fois encore par la finitude du L-ordre.

L'exponentielle procède du même esprit tout en entraînant éventuellement l'insertion de nouveaux termes de rang rationnel dans l'échelle. Quant au logarithme, outre son utilisation naturelle pour les développements de séries, il possède en outre une vertu simplificatrice sur les termes auxquels il s'applique: les puissances sont transformées en produits, les produits en somme, et les seuls termes non réductibles dont la racine (au sens racine de l'arbre représentant la fonction) est un log sont les logarithmes itérés. C'est cette propriété simplificatrice des logarithmes qui permet d'ordonner de nouveaux éléments dans l'échelle.

Exemple: cet exemple est tiré de [Hardy10a] et est ici traité automatiquement: on s'intéresse à

$$f = \sqrt{x}\ln^2 x\, e^{\sqrt{\ln x}(\ln\ln x)^2\exp(\sqrt{\ln\ln x}(\ln\ln\ln x)^3)},$$

dont on cherche le comportement à l'infini. On obtient par programme:

$$f = e_0^{1/2}(x)\, e_{-1/2}(x)\, e_{-1}^2(x).$$

Ceci veut dire que la seule information que l'on a eu besoin de calculer pour installer cette expression dans l'échelle est que le dernier terme du produit croit plus vite que $\ln(x)$ mais moins vite que x.

Conclusion: l'échelle asymptotique implémentée est très vaste et couvre une grande partie des applications courantes. On peut envisager de l'étendre en y ajoutant des fonctions algébriques implicites. On peut aisément l'utiliser pour résoudre certains problèmes d'asymptotiques de fonctions implicites (comme par exemple le développement des points cols), et c'est cette direction qui nous paraît la plus intéressante, à savoir l'utilisation de cette bibliothèque Maple comme un outil pour la construction d'édifices de plus haut niveau.

III. Admissibilités

Dans le cas où aucune singularité n'a été trouvée, on étudie d'abord l'H-admissibilité de la fonction de manière récursive à partir du théorème de Hayman sur les exponentielles de polynômes. On obtient en même temps l'HS-admissibilité lorsqu'elle a lieu.

Lorsqu'il y a une singularité, la présence ou non de termes d'indice négatif de l'échelle dans le développement au voisinage de la singularité nous décide pour le choix des théorèmes de transfert ou des méthodes de point col. En ce qui concerne les théorèmes de transfert, leur application à partir des développements est immédiate. Quant aux méthodes de point col, le dernier théorème de Hayman nous donne également l'admissibilité sans aucune difficulté. La méthode de Wyman n'est pas encore utilisée dans la version actuelle.

IV. La comparaison d'expressions

1. Cas général

En calcul numérique où, de par la nature des objets manipulés, il est toujours impossible de décider l'égalité de deux nombres, on a l'habitude de se contenter d'une approximation de l'égalité, c'est-à-dire que l'on considère deux nombres comme égaux dès que leur différence est "assez faible". En calcul formel, les objets manipulés sont des expressions mathématiques exactes, et leur comparaison devient ainsi parfois possible. Toutefois il existe des obstacles de nature théorique à la comparaison. Tout d'abord, pour des expressions appartenant à une classe suffisamment étendue, c'est-à-dire construites sur les rationnels à partir des fonctions logarithme, exponentielle, valeur absolue et des opérations algébriques ordinaires, Richardson a montré dans [Richardson68] l'indécidabilité de la nullité, et par là-même de la comparaison. De plus, même pour des expressions relativement simples, en dehors de l'approche probabiliste ([Gonnet84] et [Gonnet86]), il n'existe pas d'algorithme efficace capable de traiter une sous-classe intéressante de la classe des expressions usuelles.

Il ne faut cependant pas désespérer. D'une part les algorithmes probabilistes implémentés dans Maple couvrent une classe assez large d'expressions, d'autre part, nous nous sommes rabattus sur des heuristiques, souvent très efficaces (voir l'exemple des trains aléatoires dans la partie suivante). En gros, une singularité qui n'est pas connue de manière explicite est stockée comme un couple (valeur approchée, expression dont la singularité est racine). On calcule les développements formellement; lorsqu'il est nécessaire de tester une nullité, on évalue en flottants et si le résultat n'est pas probant, on tente quelques simplifications, si ceci ne donne rien, on admet que l'expression est nulle, ce qui se traduit par l'envoi d'un message à l'utilisateur, et par la création d'un fichier où cette hypothèse est rangée. L'utilisateur peut alors aller regarder ce fichier, et si l'hypothèse lui déplaît, le modifier, le recharger et relancer le calcul qui tiendra compte de cette information.

2. Les nombres algébriques

La majorité des expressions que nous avons à traiter se compose de fractions rationnelles. Dans ce cas tout ce dont nous avons besoin est décidable, même s'il est impossible d'exprimer la racine de manière exacte lorsque le degré du dénominateur est élevé. Pour l'instant, nous nous contentons encore des heuristiques présentées dans la section précédente. Mais pour une version ultérieure, voici les outils à employer.

D'une part, il existe des techniques [Duval87] permettant de travailler avec des racines d'un polynôme sans factoriser celui-ci. C'est-à-dire qu'aussi longtemps que les opérations effectuées sont des sommes, des produits et des puissances, les tests de nullité répondent de la même façon pour toutes les racines, modulo quelques calculs de pgcd. En particulier, le développement formel d'une fraction rationnelle au voisinage d'une singularité est valable pour toutes les autres singularités (toujours à la vérification près de quelques pgcd).

D'autre part, il est possible d'ordonner deux nombres algébriques réels d'après les travaux de [Coste-Roy88] et [Roy-Szpirglas88]. En particulier, ces méthodes permettraient de comparer les plus petits modules des racines de deux polynômes, sans calculer une seule racine.

Cependant il faut remarquer que l'utilisation de ces méthodes n'est pas strictement nécessaire du point de vue des applications courantes car en général, le programme dans sa version actuelle résout les problèmes qui lui sont posés. Il serait simplement bon du point de vue de la rigueur d'avoir des outils permettant d'assurer à l'utilisateur que toute la classe des fonctions génératrices rationnelles peut être traitée.

Chapitre 3

Exemples d'Applications

Chapitre 3

Exemples d'applications

Plutôt que de prendre des fonctions au hasard, nous allons ici faire la démonstration de l'étendue du domaine d'application du programme bâti tout au long des deux parties précédentes en l'appliquant à des fonctions génératrices issues de problèmes combinatoires ou informatiques. Notre objectif prioritaire est cependant axé sur les comportements asymptotiques des coefficients des fonctions génératrices, aussi la justification de la forme des fonctions présentées ne sera-t-elle donnée que dans les cas les plus simples. Le lecteur est prié de se reporter aux ouvrages cités en bibliographie pour des détails supplémentaires sur cet aspect de la question.

On rappelle la distinction entre fonction génératrice ordinaire (f.g.o.) et fonction génératrice exponentielle (f.g.e.): dans la première les coefficients portent l'information tandis que dans la seconde ils doivent être multipliés par $n!$ où n est le rang.

I. Le système $\Lambda\gamma\Omega$

La partie II du chapitre 1 qui présentait la notion de constructeur admissible, n'est en fait qu'une petite fenêtre sur une théorie bien plus développée, et susceptible d'automatisation. Une telle automatisation a été réalisée par Paul Zimmermann à l'INRIA, et la réunion de nos deux programmes donne le système $\Lambda\gamma\Omega$(du grec $\lambda\upsilon\omega$, délier, résoudre)[Flajolet-Salvy-Zimmermann88].

Le principe externe de ce système est simple: on souhaite disposer d'un programme acceptant en entrée une spécification d'algorithme et de données et renvoyant comme sortie sa complexité. Ceci se fait en trois étapes. Tout d'abord, l'utilisateur "programme" son algorithme et spécifie sa structure de données en ADL (Algorithms Description Language). L'analyseur algébrique ALAS de P. Zimmermann transforme alors les données en fonctions génératrices de coût et les procédures en descripteurs de complexité (le lecteur intéressé par cette partie pourra consulter [Zimmermann88], et lire le manuel de référence de $\Lambda\gamma\Omega$en annexe 3). Puis ces fonctions sont traitées par le programme **equivalent** présenté dans ce rapport.

La plupart des exemples des deux parties qui vont suivre peuvent être traités entièrement par $\Lambda\gamma\Omega$. Cependant, pour éviter une succession assez lassante de listings de fichiers ADL et d'analyse par notre programme, nous ferons précéder la plupart des exemples de la partie II (exemples de nature combinatoire) d'une courte explication permettant de comprendre l'origine de la fonction génératrice utilisée pour chaque problème. Dans la partie III en revanche (exemples d'origine informatique), nous n'hésiterons pas à montrer des exemples traités entièrement automatiquement, l'auteur n'ayant plus (dans la version actuelle) qu'à mettre les titres de section.

II. Nombres d'origine combinatoire

Pour plus de détails sur les exemples classiques de cette section, voir [Comtet70], et pour un exposé systématique sur le passage des structures aux fonctions génératrices voir [Flajolet-Vitter87].

1. Fractions rationnelles

a) Les nombres de Fibonacci

Probablement la plus célèbre de toutes les suites, la suite de Fibonacci a de nombreuses interprétations. Par exemple il s'agit du nombre de couples de lapins immortels se reproduisant à partir de l'âge de deux ans pour donner naissance à une portée annuelle de un couple et issus du même couple d'ancêtres.

Sa définition

$$F_1 = F_2 = 1 \quad F_{n+2} = F_{n+1} + F_n$$

se traduit sur la f.g.o. par

$$\sum_{n>0} F_{n+2} z^{n+2} = z \sum_{n>0} F_{n+1} z^{n+1} + z^2 \sum_{n>0} F_n z^n$$

$$\text{soit} \quad F(z) - z - z^2 = zF(z) - z^2 + z^2 F(z)$$

$$\text{d'où} \qquad F(z) = \frac{z}{1 - z - z^2}$$

on obtient donc:

```
% maple
> read(equiv);# On charge le programme
> f:=z/(1-z-z^2);
```

$$f := \frac{z}{1 - z - z^2}$$

```
> equivalent(f,5);# le deuxieme parametre est l'ordre du developpement
```

$$\frac{1}{(-1/2 + 1/2\sqrt{5})^{1/2}\, 5^{1/2}\, n} + O\left(\frac{(-1/2 + 1/2\sqrt{5})^{1/2\,(-n)}}{6^n}\right)$$

b) Combinaisons

α.Sans répétitions

Il s'agit du nombre de façons de choisir n éléments parmi m. Tout le monde connaît ce nombre $\binom{m}{n}$ et sa f.g.o. (en n):

$$(1+x)^m,$$

qui ne fait que traduire la structure:

$$\prod_{x \in \{1,\ldots,m\}} \{\epsilon, \{x\}\}$$

Le comportement asymptotique est simple: pour $n > m$, le nombre est 0.

β.Avec répétitions

La structure dénombrée est alors:

$$\prod_{x \in \{1,\ldots,m\}} \{\epsilon, \{x\}, \{x,x\}, \ldots\} \quad \text{que l'on note} \quad \prod_{x \in \{1,\ldots,m\}} \{x\}^*$$

On en déduit la série:

$$f(z) = \prod_{i \in \{1,\ldots,m\}} (1 + z + z^2 + \cdots) = \frac{1}{(1-z)^m}$$

```
> f:=1/(1-z)^2;
```

$$f := \frac{1}{(1 - z)^2}$$

```
> equivalent(f,5);
                                        (- infinity)
                    1 + n + O(n                     )
```

γ.Avec répétitions limitées (à r)

On obtient de la même façon:

$$f(z) = (1 + z + z^2 + \cdots + z^r)^m = (\frac{1 - z^{r+1}}{1 - z})^m$$

```
> f:=((1-z^5)/(1-z))^20;
                                      5 20
                                (1 - z )
                        f := ------------
                                       20
                              (1 - z)
```

```
> equivalent(f);
                        No singularities found

                                    0
```

En effet, le nombre de répétions étant limité, on ne peut pas choisir plus de $r\,m$ éléments, donc asymptotiquement le résultat est nul.

c) Compositions entières

α.En général

On compte le nombre de k-uplets d'entiers $(x_1, \ldots, x_k)$ positifs tels que $\sum x_j = n$. La taille des objets est alors prise égale à leur valeur et on obtient la série génératrice de l'ensemble $\mathbf{N}^*$:

$$N(z) = 1 + z + z^2 + \cdots = \frac{1}{1 - z}$$

les coefficients de la série au carré donneront le nombre de façons d'obtenir n comme somme de deux entiers, de même pour le cube,... On en déduit la série recherchée:

$$C(z) = \frac{1}{1 - N(z)} = \frac{1 - z}{1 - 2z}$$

```
> c:=(1-z)/(1-2*z);
                                    1 - z
                            c := ---------
                                 1 - 2 z
```

```
> equivalent(c,5);
                                      n       (- infinity)  n
                          1/2 2  + O(n                    2 )
```

β.Sommants impairs

Il suffit de remplacer dans le calcul précédent la série $N(z)$ par:

$$\sum z^{2k+1} = \frac{z}{1 - z^2},$$

qui dénombre les entiers impairs. La f.g.o. est donc:

$$1 + \frac{z}{1 - z - z^2}$$

c'est-à-dire qu'en dehors du terme d'ordre 0 qui ne signifie rien, on retrouve la suite de Fibonacci.

γ.Dénumérants

La question que l'on se pose ici est: quel est le nombre de façons de faire n francs avec des pièces de $k_1, \ldots, k_p$ francs. C'est une généralisation évidente du raisonnement précédent:

$$D[k_1, \ldots, k_p] = \frac{1}{(1-z^{k_1}) \cdots (1-z^{k_p})}.$$

On peut s'intéresser notamment à:

$D[1,2]$

```
> d:=1/(1-z)/(1-z^2);
```

$$d := \frac{1}{(1-z)(1-z^2)}$$

```
> equivalent(d,5);
```

$$1/2\,n + 3/4 + 1/4\,(-1)^n + O(n^{-5})$$

$D[1,2,3]$

```
> d:=1/(1-z)/(1-z^2)/(1-z^3);
```

$$d := \frac{1}{(1-z)(1-z^2)(1-z^3)}$$

```
> equivalent(d,8);
```

$$1/12\,n^2 + 1/2\,n + \frac{47}{72} + 1/8\,(-1)^n + 2/9\,\cos(2/3\,n\,\pi) + O(n^{-7})$$

2. Fonctions à singularités algébriques

Dans cette partie, les fonctions ne sont pas rationnelles, même si certaines d'entre elles se comportent comme telles au voisinage de leur singularité.

a) Les nombres de Bernoulli

Ces nombres qui interviennent dans la formule d'Euler-MacLaurin, dans le développement de la fonction Γ d'Euler, dans celui de cotangente et donc dans ceux de nombreuses fonctions trigonométriques, ont une f.g.e. particulièrement simple qui peut leur servir de définition:

$$B(z) = \frac{z}{e^z - 1}$$

d'où

```
> b:=z/(exp(z)-1);
```

$$b := \frac{z}{\exp(z) - 1}$$

```
> equivalent(b,5);
```

$$-\frac{1}{2^n I^n Pi^n} - \frac{1}{(-1)^n 2^n I^n Pi^n} + O\left(\frac{2^{(-n)} Pi^{(-n)}}{5^n}\right)$$

On retrouve bien la formule connue:

$$\begin{cases} B_{2m+1} \sim 0 \\ B_{2m} \sim -2(2m)!(-4\pi^2)^{-m} \end{cases}$$

b) Les nombres d'Euler

Leurs propriétés ressemblent beaucoup à celles des nombres de Bernoulli. Ils ont également une interprétation combinatoire, puisque la valeur absolue des nombres d'Euler d'ordre $2n$ est le nombre de permutations totalement alternées sur un ensemble à $2n$ éléments, et voici leur f.g.e.:

$$E(t) = \frac{1}{ch(t)} = \frac{2e^t}{e^{2t}+1}$$

```
> e:=2*exp(t)/(exp(2*t)+1);
```

$$e := 2\,\frac{\exp(t)}{\exp(2\,t) + 1}$$

```
> equivalent(e,5);
```

$$2\,\frac{2^n}{I^n Pi^n Pi^n} + 2\,\frac{2^n}{(-1)^n I^n Pi^n Pi^n} + O\left(\frac{2^n Pi^{(-n)}}{7^n}\right)$$

c) Permutations

C'est le domaine privilégié d'utilisation des f.g.e. pour lesquelles on dérive de façon à peu près semblable aux f.g.o. les séries des structures (mais on ne compte pas tout à fait la même chose puisque les atomes sont étiquetés).

Les dénombrements qui suivent s'appuient sur la décomposition en cycle des permutations. Une permutation est un ensemble de cycles. En appliquant alors les règles habituelles sur les f.g.e. (un ensemble se traduit par une exponentielle et un cycle par un quasi-log), on obtient la série génératrice de S_n:

$$S(z) = \exp(\ln(\frac{1}{1-z})) = \frac{1}{1-z} = \sum z^n,$$

et on retrouve ainsi (on est en f.g.e.):

$$card(S_n) = n!$$

α.Permutations sans point fixe sur un ensemble de taille n

L'interdiction des points fixes se traduit sur la décomposition précédente par une limitation à des cycles de longueur supérieure à 1.

$$f(z) = \exp(\ln(\frac{1}{1-z}) - z) = \frac{e^{-z}}{1-z}$$

```
> p:=exp(-z)/(1-z);
```

```
                    exp(- z)
               p := ----------
                      1 - z
```

```
> equivalent(p,5);
```

$$\exp(-1) + O\left(\frac{1}{n^5}\right)$$

β.Permutations sans cycle de longueur inférieure à $k+1$.

C'est une simple généralisation du raisonnement précédent, on obtient ainsi:

$$\frac{\exp(-z - z^2/2 - \cdots - z^k/k)}{1-z}$$

```
> p:=exp(-z-z^2/2-z^3/3)/(1-z);
```

$$p := \frac{\exp(-z - 1/2\,z^2 - 1/3\,z^3)}{1-z}$$

```
> equivalent(p);
```

$$\exp(-11/6) + O(1/n)$$

γ.Tirages à pile ou face

Un problème lié aux deux précédents est le nombre de suites de pile ou face sans séquence de k piles consécutives. La fonction génératrice des probabilités est:

$$\frac{1-z^k}{1-2z+z^{k+1}}$$

d'où la probabilité de cet événement:

$$\frac{1}{2^n}[z^n]f(z)$$

```
> pf:=(1-z^2)/(1-2*z+z^3);
```

$$pf := \frac{1 - z^2}{1 - 2\,z + z^3}$$

```
> equivalent(pf,2);
```

$$\frac{1}{(-1/2 + 1/2\,5^{1/2})^n\,(-1 + 5^{1/2})} + \frac{1}{5\,(-1/2 + 1/2\,5^{1/2})^n\,(-1 + 5^{1/2})}$$

```
                                  1/2 (- n)
                      (- 1/2 + 1/2 5    )
                + O(-------------------------)
                                    3
                                   n
```

La probabilité cherchée est donc asymptotiquement:

$$\frac{5+3\sqrt{5}}{10}\,\frac{1}{(\sqrt{5}-1)^n}.$$

d) Dénombrements d'arbres

Pour des exemples plus compliqués que ceux qui vont suivre, on pourra voir [Steyaert84].

α.Les nombres de Catalan

Il s'agit du nombre d'arbres binaires à n noeuds. La structure de cet ensemble,

$$B = o + o(B, B)$$

se traduit directement sur la f.g.o.:

$$b(z) = z + zb^2(z)$$

$$\text{d'où}\quad b(z) = \frac{1-\sqrt{1-4z^2}}{2z} \qquad \text{(l'autre solution diverge)}$$

```
> b:=(1-sqrt(1-4*z^2))/2/z;
```

```
                                        2 1/2
                                1 - (1 - 4 z )
                        b := 1/2 ----------------
                                        z
```

```
> equivalent(b,5);
```

```
                1/2  n                    1/2  n                    1/2  n
               2    2                    2    2          25       2    2
- 1/2 ------------------ + 3/8 ------------------ - ---- ------------------
             n  3/2   1/2              n  5/2   1/2     64       n  7/2   1/2
          (-1)  n    Pi              (-1)  n    Pi             (-1)  n    Pi

                   1/2  n                    1/2  n                    1/2  n
      105         2    2          1659      2    2                   2    2
  + ----- ------------------ - ------ ------------------ + 1/2 ------------
      256       n  9/2   1/2     4096       n  1/2  11/2          3/2   1/2
             (-1)  n    Pi               (-1)  Pi    n           n    Pi

                1/2  n                 1/2  n                  1/2  n
               2    2         25      2    2         105      2    2
  - 3/8 ------------ + ---- ------------ - ----- ------------
             5/2   1/2       64    7/2   1/2      256    9/2   1/2
            n    Pi              n    Pi              n    Pi
```

```
                1/2  n              n
   1659        2    2             2
+ ------ ------------- + O(-------)
   4096      1/2  11/2          13/2
          Pi     n             n
```

En fait les nombres de Catalan ont une forme explicite:

$$C_{2n} = 0 \quad C_{2n+1} = \frac{1}{n+1}\binom{2n}{n},$$

le lecteur pourra donc vérifier à l'aide de la formule de Stirling et en quelques minutes pénibles l'exactitude du résultat obtenu en quelques secondes automatiquement.

β.Arbres unaires-binaires

De la même façon:

$$U = o + o(U) + o(U,U)$$

$$\text{donc} \quad u(z) = z(1 + u(z) + u^2(z))$$

$$\text{d'où} \quad u(z) = \frac{1 - z - \sqrt{1 - 2z - 3z^2}}{2z}$$

```
> u:=(1-z-sqrt(1-2*z-3*z^2))/2/z;
                                                2 1/2
                                    1 - z - (1 - 2 z - 3 z )
                          u := 1/2 ---------------------------
                                                z
```

```
> equivalent(u);
                                  1/2  n          n
                                 3    3          3
                          1/2 ------------ + O(----)
                                3/2    1/2        2
                               n     Pi          n
```

e) Exemples plus compliqués

α.Des colliers de perles

On cherche à dénombrer des arbres de colliers dont les "perles" sont des colliers de perles simples. Plus précisément, on veut le nombre de tels arbres qui portent n perles. La f.g.e. se construit de proche en proche. On part de la f.g.e. des colliers de perles simples, c'est-à-dire de cycles:

$$\sum \frac{z^n}{n} = \log \frac{1}{1-z}$$

pour la f.g.e. des colliers de colliers de perles simples, il suffit d'itérer le calcul précédent:

$$\sum \frac{\log^n \frac{1}{1-z}}{n} = \log \frac{1}{1 - \log \frac{1}{1-z}}.$$

Il ne reste plus qu'à dénombrer le nombre d'arbres à n feuilles, dans la série génératrice duquel nous injecterons cette fonction. La structure

$$A = o + o(A) + o(A,A) + \cdots$$

se traduit en

$$A(z) = \frac{z}{1 - A(z)}$$

$$\text{donc} \quad A(z) = \frac{1 - \sqrt{1-4z}}{2}.$$

On en déduit donc le résultat:

$$F(z) = \frac{1}{2}\left[1 - \sqrt{1 - 4\log\frac{1}{1 - \log\frac{1}{1-z}}}\right],$$

qui permettra bien au programme d'utiliser sa structure de déplacement récursif dans les branches de l'arbre représentant la fonction.

```
> f:=1/2*(1-sqrt(1-4*L(L(z))));

                                                    1/2
                        f := 1/2 - 1/2 (1 - 4 L(L(z)))

> equivalent(f);

                                                  1/2
                (- exp(exp(-1/4)) exp(-1) + 1)    exp(5/8)
- 1/2 ------------------------------------------------------------
                                    n  3/2   1/2                1/2
      (- exp(exp(-1/4)) exp(-1) + 1)  n    Pi    exp(exp(-1/4))

                                   (- n)
     (- exp(exp(-1/4) - 1) + 1)
  + O(-------------------------------)
                      5/2
                     n
```

β.Graphes fonctionnels binaires

On considère une fonction de $\mathbf{Z}/n\mathbf{Z}$ dans lui-même. Si l'on applique cette fonction à un élément, puis à son image, puis à l'image de celui-ci,... on va finir par arriver sur un cycle. La question que l'on se pose est: à partir de combien d'itérations en moyenne se trouve-t-on sur un cycle ? L'asymptotique se fait ici sur le nombre n d'éléments de l'ensemble.

Le graphe d'une telle fonction peut se décomposer en un ensemble de cycles d'arbres plantés (la racine n'a qu'un fils). D'où la série génératrice:

$$g(z) = \exp(\ln(\frac{1}{1 - \sqrt{1-2z^2}})).$$

L'obtention de la fonction génératrice de la distance à un cycle est légèrement plus compliquée mais peut se faire automatiquement. Voici le programme ADL qui contient la spécification du type graphe fonctionnel binaire ainsi que les procédures permettant de compter la distance d'un point à un cycle:

Programme Adl:

```
type
 graphb = set(componentb);
 componentb = cycle(plantedtree);
 plantedtree = product(node,treeb);
 treeb = node | product(node,set(treeb,card=2));
 node = Latom(1);

procedure treesize(t:treeb);
case t of
```

```
 node : count;
 (node,s) : begin count;forall x in s treesize(x) end
end;

procedure pathlength(t:treeb);
begin
 treesize(t);
 case t of
 node : nil;
 (node,s) : forall x in s pathlength(x)
 end
end;

procedure plantedtreepl(p:plantedtree);
case p of
 (node,x) : begin treesize(x); pathlength(x) end
end;

procedure componentpl(c:componentb);
forall x in c
 plantedtreepl(x);

procedure graphpl(g:graphb);
forall c in g
 componentpl(c);

measure count:1;

to_analyze : graphpl;
```

Analyse:

```
%maplecaml

   CAML (sun) (V 2-5) by INRIA Fri Jan 22

#load"luo";;
Initializing maple ...
Done
() : void

#printlevel:=2;;
2 : num

#evalf:=true;;
true : bool

#analyze"binfg";;
Generating functions are exponential.
Average case analysis of function graphpl:
==========================================

Number of arguments of graphpl of size n is n! times:
```

```
                  n 1/2              (1/2 n)
               (2 )                 2
           ------------------ + O(----------)
             1/2  1/2   1/2           3/2
            2    n     Pi            n
```

Total cost of graphpl on all arguments of size n is n! times:

```
                      n 1/2          (1/2 n)
                1/4 (2 )     n + O(2       )
```

Average cost for graphpl on random inputs of size n is:

```
                                      1/2   1/2  3/2       1/2
              av_tau_graphpl_n := 1/4 2    Pi    n    + O(n   )
```

Floating point evaluation :

```
                                    3/2       1/2
                             .626655 n    + O(n   )
```

```
() : void

#quit();;
A bientot ...
```

γ.Les nuées

On part d'un ensemble de n droites du plan en position générale (deux quelconques d'entre elles ne sont pas parallèles et trois quelconques d'entre elles ne sont pas concourantes). P est l'ensemble de leurs points d'intersection. Une nuée est un ensemble de n points de P tels que trois quelconques d'entre eux n'appartiennent pas à une même droite de l'ensemble de départ.

Une façon d'arriver à la série génératrice est de se placer dans l'espace dual où les droites sont représentées par des points et les points d'intersection de deux droites par un segment entre les deux points correspondant aux droites. Le problème se traduit alors en la recherche du nombre d'ensemble de cycles non orientés de taille supérieure à 2. D'où la f.g.e.:

$$\frac{e^{-(2z+z^2)/4}}{\sqrt{1-z}}$$

On en déduit le développement:

```
> f:=exp(z/2+z^2/4)/sqrt(1-z);
```

```
                                          2
                              exp(1/2 z + 1/4 z )
                        f := --------------------
                                        1/2
                                 (1 - z)
```

```
> equivalent(f,3);
```

```
          exp(3/4)            exp(3/4)       97    exp(3/4)          1
        ------------ + 3/8 ------------ + ----- ------------ + O(------)
          1/2   1/2           3/2   1/2    128    5/2   1/2        7/2
         n     Pi            n     Pi             n     Pi         n
```

δ.Le problème de la ruine

Soient $X_1, X_2, \ldots$ des variables aléatoires indépendantes valant 1 avec la probabilité p et -1 avec la probabilité $q = 1 - p$. Le capital à l'instant n est $k + X_1 + \cdots + X_n$ où k est un entier supérieur à 1. Le jeu termine quand le capital atteint 0 ou c une constante choisie par avance. On note $a_n(k)$ la probabilité que le jeu termine au temps n par la ruine (capital nul). La f.g.o. est

$$A(z,k) = \left(\frac{q}{p}\right)^k \frac{\lambda^{c-k}(z) - \mu^{c-k}(z)}{\lambda^c(z) - \mu^c(z)},$$

avec

$$\begin{cases} \lambda(z) = \frac{1+\sqrt{1-4pqz^2}}{2pz} \\ \mu(z) = \frac{1-\sqrt{1-4pqz^2}}{2pz} \end{cases}.$$

Voici un cas particulier:

```
> pp:=1/2:
> q:=1-pp:k:=5:
> c:=10:
> l:=(1+sqrt(1-4*pp*q*z^2))/2/pp/z:
> m:=(1-sqrt(1-4*pp*q*z^2))/2/pp/z:
> a:=(q/pp)^k*(l^(c-k)-m^(c-k))/(l^c-m^c):
> equivalent(a);
```

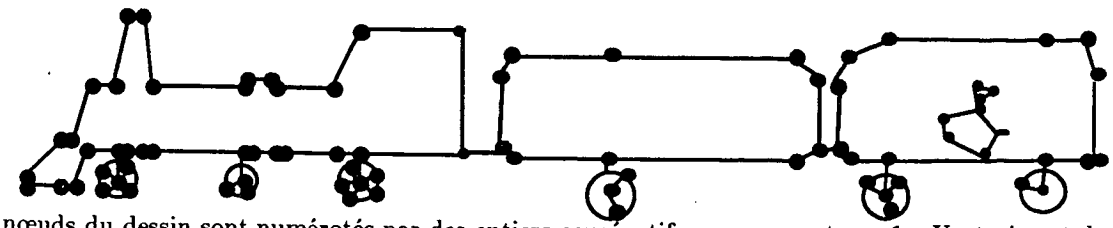
```
                  1/2
                 5
  3/10 ---------------------- - 1/2 ----------------------
                1/2       1/2 1/2           1/2       1/2 1/2
               10    (5 - 5    )           10    (5 - 5    )
 ------------------------------------------------------------
                      1/2       1/2 1/2  n
                     2    (5 - 5    )
                    (--------------------)
                             1/2
                            5

                        1/2
                       5
    - 3/10 ---------------------- + 1/2 ----------------------
                  1/2       1/2 1/2           1/2       1/2 1/2
                 10    (5 - 5    )           10    (5 - 5    )
  + ------------------------------------------------------------
                          1/2       1/2 1/2  n
                         2    (5 - 5    )
                        (- --------------------)
                                 1/2
                                5

                      1
  + O(----------------------------)
                1/2       1/2 1/2  n
               2    (5 - 5    )        2
              (--------------------)  n
                       1/2
                      5
```

ϵ.Les trains aléatoires

Cet exemple tiré de [Flajolet87] mérite une attention particulière car il permet de voir le programme à l'œuvre dans un cas où la singularité ne peut pas être connue exactement.



Les nœuds du dessin sont numérotés par des entiers consécutifs commençant par 1. Un train est donc un tel dessin dont les nœuds sont étiquetés avec les contraintes suivantes:

- un point du haut d'un wagon ou de la locomotive correspond toujours à un point du bas et réciproquement
- les roues sont des cycles (il y a donc un ordre sur les éléments)
- les passagers peuvent bouger (i.e. ils ne sont pas numérotés) à l'intérieur de *leur* wagon

La taille est alors le nombre de points du dessin. On a les égalités suivantes sur les structures:

train=loco⋆Psuitede(wagon)
loco=Psuitede(boutdecaisse) d'au moins un
boutdecaisse=haut⋆bas ou haut⋆bas⋆roue
roue=centre⋆cycle(point)
wagon=loco⋆Pensemblede(passager)
passager=tête⋆corps
tête,corps=cycle(point)
haut,bas,centre=point

qui se traduisent directement sur les f.g.e.:

train(z)=loco(z)Q(wagon(z))
loco(z)=boutdecaisse(z)Q(boutdecaisse(z))
boutdecaisse(z)=haut(z)bas(z)(1+roue(z))
roue(z)=zL(z)
wagon(z)=loco(z)exp(passager(z))
passager(z)=tête(z)corps(z)
tête(z)=corps(z)=L(z)

D'où finalement

$$train(z) = \frac{z^2 + z^3 \log \frac{1}{1-z}}{\left(1 - z^2 - z^3 \log \frac{1}{1-z}\right)\left(1 - \frac{\left(z^2+z^3 \log \frac{1}{1-z}\right) \exp(\log^2(1-z))}{1-z^2-z^3 \log \frac{1}{1-z}}\right)}$$

```
> trains:=(z^2+z^3*L(z))*Q(z^2+z^3*L(z))*Q((z^2+z^3*L(z))*Q(z^2+z^3*L(z))*exp(L(z)^2)):
> equivalent(trains);
                                    Let

s[  1  ]=   .518055
Root of    z**2/(1-z**2*(1-z*ln(1-z)))*exp(ln(1-z)**2)*(1-z*ln(1-z))-1   =0
                                 then we get :

    2
(s[1]

             2       3
  / (1 - s[1]  + s[1]  ln(1 - s[1]))
```

$$
\begin{aligned}
&/\ (2\ \frac{s[1]^2\ \exp(\ln(1 - s[1])^2)}{1 - s[1]^2 + s[1]^3\ \ln(1 - s[1])} \\
&\quad - 3\ \frac{s[1]^3\ \exp(\ln(1 - s[1])^2)\ \ln(1 - s[1])}{1 - s[1]^2 + s[1]^3\ \ln(1 - s[1])} \\
&\quad + 2\ \frac{s[1]^4\ \exp(\ln(1 - s[1])^2)}{(1 - s[1]^2 + s[1]^3\ \ln(1 - s[1]))^2} \\
&\quad - 5\ \frac{s[1]^5\ \ln(1 - s[1])\ \exp(\ln(1 - s[1])^2)}{(1 - s[1]^2 + s[1]^3\ \ln(1 - s[1]))^2} \\
&\quad + 3\ \frac{s[1]^6\ \ln(1 - s[1])^2\ \exp(\ln(1 - s[1])^2)}{(1 - s[1]^2 + s[1]^3\ \ln(1 - s[1]))^2} \\
&\quad + \frac{s[1]^6\ \exp(\ln(1 - s[1])^2)}{(1 - s[1]^2 + s[1]^3\ \ln(1 - s[1]))^2\ (1 - s[1])} \\
&\quad - \frac{s[1]^7\ \exp(\ln(1 - s[1])^2)\ \ln(1 - s[1])}{(1 - s[1]^2 + s[1]^3\ \ln(1 - s[1]))^2\ (1 - s[1])} \\
&\quad - 2\ \frac{s[1]^3\ \exp(\ln(1 - s[1])^2)\ \ln(1 - s[1])}{(1 - s[1]^2 + s[1]^3\ \ln(1 - s[1]))^2\ (1 - s[1])} \\
&\quad + 2\ \frac{s[1]^4\ \exp(\ln(1 - s[1])^2)\ \ln(1 - s[1])^2}{(1 - s[1]^2 + s[1]^3\ \ln(1 - s[1]))^2\ (1 - s[1])}
\end{aligned}
$$

$$
\begin{aligned}
&+ \frac{s[1]^4 \exp(\ln(1 - s[1])^2)}{(1 - s[1]^2 + s[1]^3 \ln(1 - s[1]))\,(1 - s[1])} \\
&- s[1]^3 \ln(1 - s[1]) \\
&/ (1 - s[1]^2 + s[1]^3 \ln(1 - s[1])) \\
&/ \left(2 \frac{s[1]^2 \exp(\ln(1 - s[1])^2)}{1 - s[1]^2 + s[1]^3 \ln(1 - s[1])}\right. \\
&\quad - 3 \frac{s[1]^3 \exp(\ln(1 - s[1])^2) \ln(1 - s[1])}{1 - s[1]^2 + s[1]^3 \ln(1 - s[1])} \\
&\quad + 2 \frac{s[1]^4 \exp(\ln(1 - s[1])^2)}{(1 - s[1]^2 + s[1]^3 \ln(1 - s[1]))^2} \\
&\quad - 5 \frac{s[1]^5 \ln(1 - s[1]) \exp(\ln(1 - s[1])^2)}{(1 - s[1]^2 + s[1]^3 \ln(1 - s[1]))^2} \\
&\quad + 3 \frac{s[1]^6 \ln(1 - s[1])^2 \exp(\ln(1 - s[1])^2)}{(1 - s[1]^2 + s[1]^3 \ln(1 - s[1]))^2} \\
&\quad \left. + \frac{s[1]^6 \exp(\ln(1 - s[1])^2)}{(1 - s[1]^2 + s[1]^3 \ln(1 - s[1]))^2\,(1 - s[1])}\right)
\end{aligned}
$$

```
                 7                  2
          s[1]  exp(ln(1 - s[1]) ) ln(1 - s[1])
      - ----------------------------------------
                   2       3              2
         (1 - s[1]  + s[1]  ln(1 - s[1])) (1 - s[1])

                   3                  2
          s[1]  exp(ln(1 - s[1]) ) ln(1 - s[1])
      - 2 ----------------------------------------
                   2       3
         (1 - s[1]  + s[1]  ln(1 - s[1])) (1 - s[1])

                   4                  2              2
          s[1]  exp(ln(1 - s[1]) ) ln(1 - s[1])
      + 2 ----------------------------------------
                   2       3
         (1 - s[1]  + s[1]  ln(1 - s[1])) (1 - s[1])

                     4                  2
              s[1]  exp(ln(1 - s[1]) )
      + ----------------------------------------))
                   2       3
         (1 - s[1]  + s[1]  ln(1 - s[1])) (1 - s[1])

        n
1 / s[1]

        1
+ O(---------)
         n
     s[1]  n
```

Or

```
                                          (- n)
                    1           .518055
.100855 ---------- + O(--------------)
                    n             n
          .518055
```

3. Fonctions à singularités logarithmiques

Dans tous les exemples précédant, les constructeurs ensembles-de ou cycles-de ne sont jamais intervenus dans la position ni dans la nature de la singularité. Les quelques exemples qui suivent montrent qu'en général il peuvent bien sûr jouer un grand rôle.

a) Rondes d'enfants

On prend n enfants dans une cour d'école, ils se rassemblent par petits groupes chacun faisant une ronde avec un enfant au milieu et les autres en cercle autour. Le problème est de trouver le nombre de configurations possibles pour cette cour de récréation. Une ronde sans enfant au centre est un cycle, le fait de mettre un enfant au milieu revient à multiplier par z, et les mettre en plusieurs rondes se traduit par une

exponentiation. Autrement dit, on dénombre des ensembles de produits d'un élément et d'un cycle, d'où la f.g.e.:

$$R(z) = \exp\Big(z \log\big(\frac{1}{1-z}\big)\Big) = (1-z)^{-z}$$

```
> equivalent((1-z)^(-z),4);
```

$$1 - 1/n - 2\frac{\ln(n)}{n^2} - 2\frac{\gamma}{n^2} - 2\frac{\ln(n)^2}{n^3} - 8\frac{\ln(n)}{n^3} - 4\frac{\gamma\ln(n)}{n^3}$$

$$+ 17\frac{1}{n^3} - 8\frac{\gamma}{n^3} - 2\frac{\gamma^2}{n^3} + 1/3\frac{\pi^2}{n^3} + O\Big(\frac{\ln(n)^3}{n^4}\Big)$$

b) Nombres de Stirling de première espèce

Ils sont définis par une f.g.e. double:

$$(1+t)^u = 1 + \sum_{1 \le k \le n} s(n,k)\frac{t^n}{n!}u^k$$

qui peut se ramener à une f.g.e. simple:

$$(1+t)^u = \exp[u\log(1+t)] = \sum_{k \ge 0} u^k \frac{\log^k(1+t)}{k!}$$

$$\text{d'où} \qquad \frac{\log^k(1+t)}{k!} = \sum_{n \ge k} s(n,k)\frac{t^n}{n!}.$$

Nombres dont la valeur absolue est le nombre de permutations à k cycles sur un ensemble à n éléments.

```
> s:=1/2*(ln(1+t))^2;
```

$$s := 1/2\,\ln(1+t)^2$$

```
> equivalent(s,5);
```

$$\frac{\ln(n)}{(-1)^n n} + \frac{\gamma}{(-1)^n n} + 4\frac{\ln(n)}{(-1)^n n^2} - 7/2\frac{1}{(-1)^n n^2} + 4\frac{\gamma}{(-1)^n n^2}$$

$$+ 10\frac{\ln(n)}{(-1)^n n^3} - \frac{181}{12}\frac{1}{(-1)^n n^3} + 10\frac{\gamma}{(-1)^n n^3} + 22\frac{\ln(n)}{(-1)^n n^4}$$

$$- 127/3\frac{1}{(-1)^n n^4} + 22\frac{\gamma}{(-1)^n n^4} + 46\frac{\ln(n)}{(-1)^n n^5} - \frac{4113}{40}\frac{1}{(-1)^n n^5}$$

```
          gamma          ln(n)
+ 46 ---------- + O(-------)
          n  5          6
       (-1)  n         n
```

(gamma est la constante d'Euler).

4. Fonctions entières

D'après le chapitre 1, ces fonctions conduisent à des méthodes de point col (sauf bien sûr pour les polynômes!). Or ces méthodes sont d'une utilisation manuelle souvent difficile, mais la grande généralité des théorèmes que nous avons utilisés dans le programme permet une obtention automatique de résultats habituellement difficiles mais classiques.

a) La formule de Stirling

Tout le monde connaît la fonction génératrice ordinaire de $1/n!$: il s'agit de $\exp(z)$ et voici ce que donne le programme dans ce cas:

```
> equivalent(exp(x));
```

```
                    No singularities found

                          exp(n)
                    --------------------
                      n  1/2   1/2  1/2
                     n 2    Pi    n
```

Précisons que la formule de Stirling ne figure pas dans le programme, elle est donc ici démontrée automatiquement, parce que les théorèmes que nous avons programmés la contiennent comme cas particulier.

b) Les nombres de Stirling de seconde espèce

$S(n,k)$ est le nombre de partitions d'un ensemble à n éléments en k sous-ensembles. On obtient donc

$$S(n,k) = \frac{1}{k!} \sum_{0 \le j \le k} (-1)^j \binom{k}{j} (k-j)^n$$

dont on finit par déduire la série génératrice:

$$S_k(z) = \frac{(e^z - 1)^k}{k!}.$$

Ou à l'inverse, on obtient la série génératrice directement par les constructeurs admissibles et on en déduit la formule générale. On demande l'asymptotique:

```
> s:=1/2*(exp(z)-1)^2;
```

```
                                        2
                    s := 1/2 (exp(z) - 1)
```

```
> equivalent(s);
```

```
                    No singularities found
```

```
1/2

                                  n              2
  (exp(1/2 n + W(- 1/2 ------------)) - 1)
                       exp(1/2 n)
```

```
                                    n       n  1/2   1/2
/ ((1/2 n + W(- 1/2 ------------)) 2    Pi    )
                    exp(1/2 n)

                              n       1/2
/ (1/2 n + W(- 1/2 ------------))
                    exp(1/2 n)

                                      n
       exp(1/2 n + W(- 1/2 ------------))
                                 exp(1/2 n)
/ (2 ---------------------------------------
                                    n
     exp(1/2 n + W(- 1/2 ------------)) - 1
                           exp(1/2 n)

   -2

                                 n                                    n       2
      (1/2 n + W(- 1/2 ------------)) exp(1/2 n + W(- 1/2 ------------))
                       exp(1/2 n)                           exp(1/2 n)

                                      n           2
        / (exp(1/2 n + W(- 1/2 ------------)) - 1)
                                 exp(1/2 n)

   + 2

                                 n                                    n
      (1/2 n + W(- 1/2 ------------)) exp(1/2 n + W(- 1/2 ------------))
                       exp(1/2 n)                           exp(1/2 n)

                                      n
        / (exp(1/2 n + W(- 1/2 ------------)) - 1))
                                 exp(1/2 n)

   ^ (1/2)
```

Où W est la fonction définie par:

$$W(x)\exp(W(x)) = x.$$

c) Nombres de Bell (partitions)

On peut s'intéresser également au nombre total de partitions d'un ensemble à n éléments:

$$d_n = \sum_{k=0}^{n} S(n,k)$$

$$\text{donc} \quad d_n z^n = \sum_{k=0}^{n} \frac{(e^z-1)^k}{k!},$$

$$\text{d'où bien sûr} \quad d(z) = \exp(e^z - 1)$$

ou directement en disant que l'on dénombre des ensembles d'ensembles non vides,

$$d(z) = \exp(e^z - 1),$$

série étudiée dans [De Bruijn81]. Le programme nous donne:

```
> equivalent(exp(exp(z)-1));
```

```
                        No singularities found

1/2

                                  1/2
   exp(exp(W(1 + n)) - 1) 2

      (1

         -2

            (-3/2

                  1/2
               Pi

                                                4
                  (1/24 exp(W(1 + n)) W(1 + n)  + 1/4 exp(W(1 + n)) W(1 + n))

                  / (W(1 + n) (exp(W(1 + n)) W(1 + n) + exp(W(1 + n))))

               + 15/4

                    1/2
                 Pi

                                                  3
                    (1/6 exp(W(1 + n)) W(1 + n)  - 1/3 exp(W(1 + n)) W(1 + n)

                    )

                       ~ 2

                                 2                                          2
                    / (W(1 + n)  (exp(W(1 + n)) W(1 + n) + exp(W(1 + n))) ))

                                                                      1/2
              / (W(1 + n) (exp(W(1 + n)) W(1 + n) + exp(W(1 + n))) Pi    )

                                                                   (- n)
         + o((1/2 W(1 + n) (exp(W(1 + n)) W(1 + n) + exp(W(1 + n))))     ))

                  n   1/2          1/2
      / (W(1 + n)  Pi    W(1 + n)    )

                                                    1/2
      / (exp(W(1 + n)) W(1 + n) + exp(W(1 + n)))
```

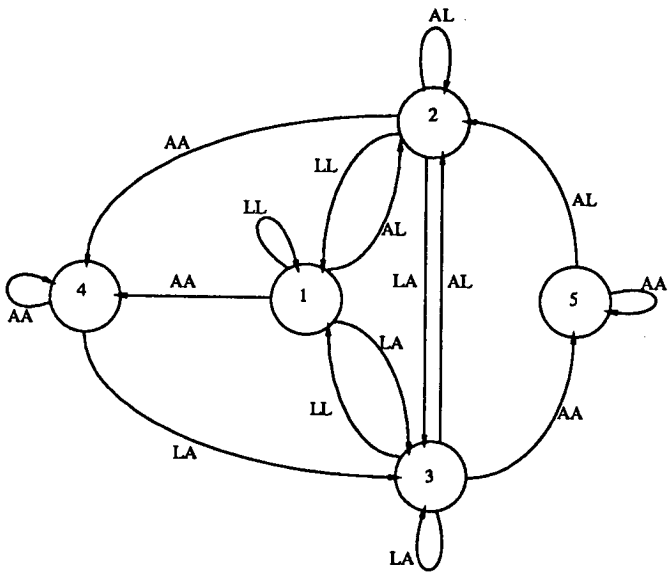
III. Applications en informatique

Dans cette partie tous les exemples sont traités automatiquement par AγΩ. On commence donc par préciser la nature du problème, que l'on traduit dans le langage Adl, et l'analyse se fait alors sans aucune intervention de l'utilisateur.

1. En mesure du parallélisme

[Beauquier-Berard-Thimonier88] introduisent une notion de mesure du parallélisme à partir de dénombrements dans des langages réguliers caractérisant les processus parallèles. Les calculs donnent donc toujours lieu à des fonctions génératrices rationnelles. L'exemple suivant va montrer que nous sommes capables de traiter automatiquement cette notion. D'autres exemples sont traités dans [Zimmermann88] de la même façon.

L'automate qui suit (voir figure) modélise un problème d'exclusion mutuelle. Il y a deux processus qui peuvent soit travailler (**L** pour labeur) soit accéder ou demander l'accès à une ressource (**A**). Les flèches entre les états de l'automate correspondent aux transitions. Leurs étiquettes sont des couples de lettres, la première décrivant l'état du premier processus et le seconde celle de l'autre. On note que le premier processus est prioritaire par rapport au second: lorsque les deux processus demandent l'accès simultanément (transition $1 \mapsto 4$), c'est le premier qui l'obtient, et on repart donc toujours par **LA**.



Pour mesurer la perte d'efficacité due à ce partage de ressource, la question que l'on se pose est: combien de temps est passé en attente d'accès ? Ce temps correspond au nombre d'apparition de la lettre **AA** sur les mots du langage décrit par cet automate.

Programme Adl:

% exclusion mutuelle avec 2 processus et une section critique

– 2 etats : (L)abour (hors s.c.) et (A)ction (dans s.c.)
– le premier processus est prioritaire
– on part de l'etat 1

etat 1 : personne n'est dans la sc, ni ne la demande
etat 2 : 1 est dans la sc, 2 ne la demandant pas
etat 3 : 2 est dans la sc, 1 ne la demandant pas
etat 4 : 1 est dans la sc, 2 la demande
etat 5 : 2 est dans la sc, 1 la demande
%

```
type h1 = () | product(LL,h1) | product(AL,h2) | product(AA,h4) | product(LA,h3);
 h2 = () | product(AL,h2) | product(AA,h4) | product(LL,h1) | product(LA,h3);
 h3 = () | product(LL,h1) | product(AA,h5) | product(LA,h3) | product(AL,h2);
 h4 = () | product(AA,h4) | product(LA,h3);
 h5 = () | product(AA,h5) | product(AL,h2);
 LL,AA,LA,AL = atom(1);

procedure ct1(h:h1);
case h of
 () : nil;
 LL(x) : ct1(x);
 AL(x) : ct2(x);
 AA(x) : begin wait2; ct4(x) end;
 LA(x) : ct3(x)
end;

procedure ct2(h:h2);
case h of
 () : nil;
 LL(x) : ct1(x);
 AL(x) : ct2(x);
 AA(x) : begin wait2; ct4(x) end;
 LA(x) : ct3(x)
end;

procedure ct3(h:h3);
case h of
 () : nil;
 LL(x) : ct1(x);
 AL(x) : ct2(x);
 AA(x) : begin wait1; ct5(x) end;
 LA(x) : ct3(x)
end;

procedure ct4(h:h4);
case h of
 () : nil;
 AA(x) : begin wait2; ct4(x) end;
 LA(x) : ct3(x)
end;

procedure ct5(h:h5);
case h of
 () : nil;
 AL(x) : ct2(x);
 AA(x) : begin wait1; ct5(x) end
end;

measure wait1 : a;  % for the first process %
    wait2 : b;      % for the second one %

to_analyze : ct1;
```

Analyse:

```
#analyze"access";;
Generating functions are ordinary.
Average case analysis of function ct1:
=====================================

Average cost for ct1 on random inputs of size n is:

2

                                                                          1/2
                        1                     1                          2
 (- b (- 1/2 ------------------ + 3/4 ------------ - 1/4 ------------)
                    1/2   1/2                    1/2                    1/2
             (2 - 3 2    ) 2            2 - 3 2               2 - 3 2

                        1                     1
   + 2 b (- 1/2 ------------------ + 1/2 ------------)
                       1/2   1/2                    1/2
                (2 - 3 2    ) 2            2 - 3 2

   - b

                                 1
     (- 1/2 ---------------------------------
                   1/2              1/2   1/2
             (2 - 3 2    ) (1 - 1/2 2    ) 2

                                1
        + 1/2 ----------------------------)
                     1/2              1/2
              (2 - 3 2    ) (1 - 1/2 2    )

                       1                     1
   - a (- 1/2 ------------------ + 1/2 ------------))
                     1/2   1/2                    1/2
              (2 - 3 2    ) 2            2 - 3 2

    1/2           1/2
   2    (1 - 1/2 2    ) n

 + O(1)
```

Cet exemple fait apparaître l'utilisation de paramètres (a et b). On s'aperçoit en prenant $a = 1$ et $b = 2$ que pendant près du quart du temps, l'un des processus attend l'autre. On peut ensuite modifier les paramètres et manipuler cette expression sous Maple ou modifier le programme Adl et recommencer pour tester de nouveaux modèles.

2. En complexité des systèmes de réécriture

[Choppy-Kaplan-Soria88] ont étudié la complexité de systèmes de réécriture dont les règles descendent dans des structures arborescentes sans jamais remonter. Un grand nombre des exemples de leur article peuvent se traiter automatiquement. Nous en avons sélectionné deux: le shuffle et le cas de fonctions mutuellement récursives, auxquels nous rajoutons un exemple de distributivité à remontée bornée emprunté à P. Zimmermann. Les autres exemples sont présentés dans [Zimmermann88].

a) Shuffle

Les termes sont des arbres binaires. Le shuffle de deux arbres correspond à la construction d'arbres dont les nœuds sont des paires, l'élément de droite provenant du premier arbre et l'élément de gauche du second.

Programme Adl:

```
type tree = a | o(tree,tree);
 a,o = atom(1);

function shuffle(x,y:tree):tree;
case x,y of
 a,t        : begin count; o(a,t) end;
 o(t1,t2),a : begin count; o(o(t1,t2),a) end;
 o(t1,t2),o(u1,u2) : begin count; o(shuffle(t1,u1),shuffle(t2,u2)) end
end;

measure count : 1;
```

Analyse:

```
#evalf:=true;;
true : bool

#analyze"shuffle";;
Generating functions are ordinary.

Average case analysis of function shuffle:
==========================================

Average cost for shuffle on random inputs of size n is:
```

$$av\_tau\_shuffle\_n := \frac{4\,\frac{2^{1/2}}{n^{1/2}\,(-1/2)\;Pi} + 4\,\frac{2^{1/2}}{n^{1/2}\,(1/2)\;Pi}}{-1/2\,\frac{2^{1/2}}{n^{1/2}\,(-1/2)\;Pi} + 1/2\,\frac{2^{1/2}}{n^{1/2}\,(1/2)\;Pi}} + O\left(\frac{1}{n^{1/2}}\right)$$

```
Floating point evaluation :
```

$$\frac{3.19154\,\frac{1}{n^{(-.500000)}} + 3.19154\,\frac{1}{n^{.500000}}}{-.398942\,\frac{1}{n^{(-.500000)}} + .398942\,\frac{1}{n^{.500000}}} + O\left(\frac{1}{n^{1/2}}\right)$$

On peut obtenir une expression plus claire en distinguant selon la parité, opération que l'utilisateur peut faire au niveau de Maple.

b) Distributivité

On applique la règle de distributivité, mais comme celle-ci pourrait entraîner une nouvelle application au niveau supérieur de l'arbre et que l'on souhaite l'exprimer en terme de descente dans un arbre, on regarde par avance k niveaux en-dessous pour traiter tous les cas qui demandent une remontée de moins de k niveaux. En faisant grandir k, on obtient des approximations de la réalité de plus en plus précises. L'exemple suivant traite le cas $k = 1$.

Programme Adl:

```
type expression = x
       | plus(expression,expression)
       | expression1
       | expo(expression);
 expression1 = times(expression,expression);
 plus,times,x,expo = atom(1);

function distrib(e:expression):expression;
case e of
 plus(t,u) : plus(distrib(t),distrib(u));
 times(t,u) : distrib1(e);
 expo(t) : expo(distrib(t));
 x     : x
end;

function distrib1(e:expression1):expression;
case e of
 times(t,u):
 case t of
 plus(v,w) : plus(distrib1(times(v,u)),
       distrib1(times(w,u)));
 times(v,w): times(distrib1(t),distrib(u));
 expo(v) : times(expo(distrib(v)),distrib(u));
 x      : times(x,distrib(u))
 end
end;

measure plus,times,x,expo : 1;
```

Analyse: cet exemple n'a pas pu être analysé automatiquement. Non pas à cause de lacunes du programme (les fonctions génératrices sont de simples fractions rationnelles), mais pour des raisons physiques: les singularités sont obtenues par les formules de Cardan, et le développement des fractions au voisinage de leurs singularités fait apparaître des expressions dont la taille croît jusqu'à 1Mo. Ceci souligne tout l'intérêt qu'il peut y avoir à utiliser les méthodes de calcul sur des nombres algébriques décrites à la fin du chapitre 2.

3. En algorithmique

a) Dérivation formelle

Cet exemple est traité dans le cadre le plus général dans [Steyaert84]. On se limite ici à des fonctions composées de $\{x, \sqrt{}, +, \times, inverse\}$ où ces différents termes apparaissent avec la même fréquence qui est aussi celle des nombres réels.

Programme Adl:

```
type expression = x | const
       | root(expression)
```

```
        | plus(expression,expression)
        | times(expression,expression)
        | inverse(expression);
 root,plus,times,inverse,x,const = atom(1);

function diff(e:expression):expression;
case e of
 x      : const;
 const  : const;
 root(e1)  : begin count; count; count; diff(e1); copy(e) end;
 plus(e1,e2)  : plus(diff(e1),diff(e2));
 times(e1,e2)  : plus(times(diff(e1),copy(e2)),
        times(copy(e1),diff(e2)));
 inverse(e1)  : begin count; count; diff(e1); sqr(copy(e1)) end
end;

function copy (e:expression):expression;
case e of
 x      : x;
 const  : const;
 root(e1)  : root(copy(e1));
 plus(e1,e2)  : plus(copy(e1),copy(e2));
 times(e1,e2)  : times(copy(e1),copy(e2));
 inverse(e1)  : inverse(copy(e1))
end;

measure count,root,plus,times,inverse,x,const,sqr:1;
```

Analyse:

```
#evalf:=true;;
evalf : bool

#analyze"diff";;
Generating functions are ordinary.
Average case analysis of function diff:
=======================================
Average cost for diff on random inputs of size n is:

                                        1/2   1/2  3/2
                  av_tau_diff_n := 1/3 3    Pi    n    + O(n)

Floating point evaluation :

                                      3/2
                               1.02332 n    + O(n)
```

b) Algorithmes de comparaison

Toujours avec la même famille d'expressions, on s'intéresse au coût de recherche de la plus grande sous-expression commune à deux expressions: voici l'analyse de la formule donnée dans [Steyaert84]:

```
> tau:=2*f^2/(1-z^2*(2+4*f^2)):
> dividev(equiv(tau,4),equiv(f,4),4);
```

$$\frac{136}{25} + \frac{51}{10} 1/n - \frac{1717}{320}\frac{1}{2^n} + \frac{27489}{5120}\frac{1}{3^n} + O(\frac{1}{4^n})$$

avec n la taille totale des deux expressions (le nombre de feuilles).

c) Chaînes d'addition

Cet exemple joue un rôle particulier dans l'histoire de $\Lambda\Upsilon\Omega$: c'est le premier exemple d'une utilisation par une autre personne que les auteurs. Le problème, mentionné dans [Knuth81], est de déterminer la façon la plus économique d'élever à une puissance dont on connaît la décomposition binaire. Nous ne pouvons bien entendu pas répondre à cette question, mais tester et chiffrer des optimisations est précisément le rôle du programme.

L'algorithme naïf est le suivant: si $a = (1a_0a_1 \ldots a_n)_2$ est la puissance, le résultat cherché est:

$$x^a = \prod_{a_i=1} x^{2^{n-i}}.$$

On calcule donc par élévation au carré successives et produits des résultats intermédiaires.

Jorge Olivos (Université de Santiago du Chili) a proposé et évalué à l'aide de $\Lambda\Upsilon\Omega$ deux optimisations dans le cas où le produit, la mise au carré et la division ont sensiblement le même coût.

La première optimisation consiste à éviter les produits répétés lorsque la puissance contient plusieurs 1 successifs: on remplace

$$x^{(1\ldots1)_2} \quad \text{par} \quad \frac{x^{2^n}}{x}.$$

On remplace ainsi n multiplications par une division. Et ceci s'intègre bien dans la procédure récursive de mise à la puissance.

Programme Adl:

```
type chain = sequence(bit);
 bit = zero | one;
 zero,one = atom(1);

procedure naive(c:chain);
case c of
 () : nil;
 (zero,c1) : begin squaring; naive(c1) end;
 (one,c1) : begin squaring; multiply; naive(c1) end
end;

procedure treat(c:chain);
case c of
 () : nil;
 (zero,c1) : begin squaring; treat(c1) end;
 (one,c1) : begin squaring; treat1(c1) end
end;

procedure treat1(c:chain); % one 1 has been recognized %
case c of
 () : multiply;
 (zero,c1) : begin multiply; squaring; treat(c1) end;
 (one,c1) : begin divide; squaring; treat11(c1) end
end;

procedure treat11(c:chain); % at least two 1 have been recognized %
```

```
case c of
() : multiply;
(zero,c1) : begin squaring; multiply; treat(c1) end;
(one,c1) : begin squaring; treat11(c1) end
end;

measure multiply : 1;
   divide : 1;
   squaring : 1;

to_analyze : naive,treat;
Analyse:
   #evalf:=true;;
   true : bool

   #analyze"chains";;
   Generating functions are ordinary.
   Average case analysis of function naive:
   ========================================


   Average cost for naive on random inputs of size n is:

                          av_tau_naive_n := 3/2 n + O(1)

   Floating point evaluation :

                                  1.50000 n + O(1)

   Average case analysis of function treat:
   ========================================

   Average cost for treat on random inputs of size n is:

                         av_tau_treat_n := 11/8 n + O(1)

   Floating point evaluation :

                                  1.37500 n + O(1)
```

La seconde optimisation s'intéresse aux 0 isolés entre deux suites de 1. Dans l'optimisation précédente, on obtient alors pour une puissance 55 par exemple:

$$x^{55} = x^{(110111)_2} = \frac{x^8}{x} \cdot \frac{x^{64}}{x^{16}},$$

la puissance 8 et la puissance 16 peuvent se simplifier, c'est-à-dire que l'on peut calculer directement

$$x^{55} = \frac{x^{64}}{x^8 . x,}$$

où on a gagné une multiplication.

Programme Adl:

```
type chain = sequence(bit); % A + B = C then A = C − B %
 valid_chain = product(chain,one);
 bit = zero | one;
 zero,one = atom(1);

procedure treat(c:valid_chain);
case c of
 (c1,one) : begin treatStart(c1); divide end
end;

procedure treatStart(c:chain);
case c of
 () : nil;
 (zero,c1) : begin treat0(c1) end;
 (one,c1) : begin treat1(c1) end
end;

procedure treat0(c:chain);
case c of
 () : begin squaring; multiplyC end;
 (zero,c1) : begin squaring; treat0(c1) end;
 (one,c1) : begin squaring; treat1(c1) end
end;

procedure treat1(c:chain); % one 1 has been recognized %
case c of
 () : begin multiplyC; multiplyC end;
 (zero,c1) : begin multiplyC; squaring; treat0(c1) end;
 (one,c1) : begin squaring; multiplyB; treat11(c1) end
end;

procedure treat11(c:chain); % at least two 1 have been recognized %
case c of
 () : begin squaring; squaring; multiplyC end;
 (zero,c1) : begin squaring; treat110(c1) end;
 (one,c1) : begin squaring; treat11(c1) end
end;

procedure treat110(c:chain); % at least two 1 and one 0 have been recognized %
case c of
 () : begin multiplyB; squaring; multiplyC end;
 (zero,c1) : begin multiplyC; squaring; treat0(c1) end;
 (one,c1) : begin multiplyB; squaring; treat11(c1) end
end;

measure multiplyB : 1;
    multiplyC : 1;
    multiply : 1;
    divide : 1;
    squaring : 1;

to_analyze : treat;
```

Analyse:

```
#evalf:=true;;
true : bool

#analyze"chains2";;
Generating functions are ordinary.
Average case analysis of function treat:
========================================

Average cost for treat on random inputs of size n is:

                          av_tau_treat_n := 4/3 n + O(1)

Floating point evaluation :

                                  1.33333 n + O(1)
```

d) Retour sur les trains aléatoires

Les trains aléatoires que nous avons regardé dans la section précédente en tant que structure combinatoire, peuvent aussi être considérés comme une structure de données sur laquelle s'exécutent des programmes dont nous souhaitons évaluer la complexité. Le programme suivant recherche la position du premier wagon vide. L'analyse va donc nous donner la position moyenne du premier wagon vide dans un train aléatoire de taille n lorsque n tend vers l'infini.

Programme Adl:

```
type
 train = product(locomotive,wagons);
 wagons = sequence(wagon);
 locomotive = sequence(slice,card >= 1);
 slice = product(upper,lower)
 | product(upper,lower,wheel);
 wheel = product(center,cycle(wheel_element));
 wagon = product(locomotive,passengers);
 passengers = set(passenger);
 passenger = product(head,belly);
 head,belly = cycle(passenger_element);
 upper,lower,center,wheel_element,passenger_element = Latom(1);

function first_empty_wagon (t:train):integer;
case t of
 (loco,ws) : few(ws)
end;

function few(ws:wagons):integer;
case ws of
 () : nil;
 (w,others) : if empty(w) then count
        else begin count; few(others) end
end;

function empty(w:wagon):boolean;
case w of
 (loco,ps) : empty_set(ps)
```

```
end;

function empty_set(ps:passengers):boolean;
case ps of
 () : true;
 otherwise : false
end;

measure count:1;
    nil:0;

to_analyze : first_empty_wagon;
```

Analyse: l'analyse renvoie une formule d'une dizaine de pages dont l'évaluation flottante nous apprend qu'en moyenne et asymptotiquement, le premier wagon est toujours vide. Ceci signifie que les cas où il n'y a que la locomotive compensent exactement (au premier ordre) les cas où le premier wagon existe et est vide.

Conclusion

Il reste encore des choses à faire sur ce programme, et ce dans plusieurs directions différentes.

Tout d'abord au niveau de la théorie programmée : parmi tous les mathématiciens qui ont défini des classes de fonctions pour en dériver des développements, seul Hayman a cherché à délimiter des sous-classes dont les éléments, pris parmi les fonctions usuelles, sont reconnaissables. Il manque un tel travail sur les fonctions HS-admissibles (quoique l'article d'Odlyzko et Richmond aille dans ce sens) et sur les fonctions W-admissibles pour lesquelles la sous-classe donnée par Wyman doit pouvoir être améliorée. En ce qui concerne la classe de Wyman, il faudrait aussi chercher à approfondir le problème de la détermination automatique des chemins significatifs, car il semble qu'il y ait là un outil très puissant. De plus, le fossé apparent entre les fonctions à singularités essentielles et celles qui sont "en-dessous" demande à être précisé, ainsi que les liens entre les fonctions entières et les fonctions à singularités essentielles.

Ensuite au niveau des outils, il faut amener les objets manipulés à plus de généralité, pour permettre par exemple non pas de décider mais de dire avec une grande probabilité si une expression impliquant une racine d'une équation transcendante est nulle ou non.

Enfin, étape ultime, il faudrait automatiser les développements des coefficients de fonctions génératrices non-exprimables à l'aide des fonctions usuelles : fonctions implicites et produits infinis.

Mais tout ceci semble possible, et seul le temps peut s'avérer un obstacle.

Annexes

Annexe 1

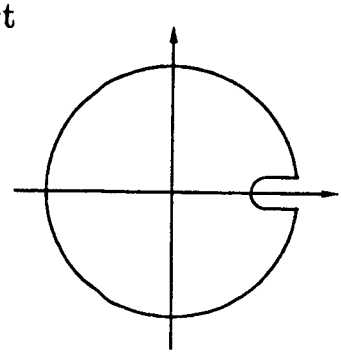
Classification des fonctions

Catégorie	Fonction	Théorème	Resultat
sing.alg.	$(x-a)^k f(x)+g(x) \quad k \in \mathbf{C}-\mathbf{N}$	Darboux	dev
sing.alg-log	$(1-x)^s \ln^k(\frac{1}{1-x}) \quad k \in \mathbf{N}, s \in \mathbf{C}$	Jüngen	dev
sing.alg-log	$(1-x)^s \ln^t(\frac{1}{1-x}) \quad t \in \mathbf{C}, s \in \mathbf{C}-\mathbf{N}$	Transfert	dev
sing.ess	$\ln^k(1-z)(1-z)^u f(z) exp(P(\frac{1}{1-z}))u \in \mathbf{C}, k \in \mathbf{N}$	Haüsler	dev
sing.ess	$\ln^v(1-z)(1-z)^u f(z) exp(\frac{c}{1-z})u \in \mathbf{C},, v \in \mathbf{C}$	Macintyre&Wilson	dev
sing.ess	$[\frac{\ln(1-z)}{z}]^v(1-z)^u f(z) \exp P(\frac{1}{1-z})\, u \in \mathbf{C},, v \in \mathbf{C}$	Wright	dev
sing.ess	$[a-\ln(1-z)]^v(1-z)^u f(z) \exp P(\frac{1}{1-z})\, u \in \mathbf{C},, v \in \mathbf{C}$	Wright	dev
sing.ess	en $exp(\frac{1}{1-z})$ au moins	Hayman	equ
sing.ess	en $exp(exp(\frac{1}{1-z}))$ au moins	Harris-Schoenfeld	dev
sing.ess	en $exp(\frac{1}{1-z})$ au moins	Wyman	dev
fct ent	En général	Point col	equ
fct ent	en $exp(z)$ au moins	Hayman	equ
fct ent	en $exp(exp(z))$ au moins	Harris-Schoenfeld	dev
fct ent	en $exp(z)$ au moins	Wyman	dev

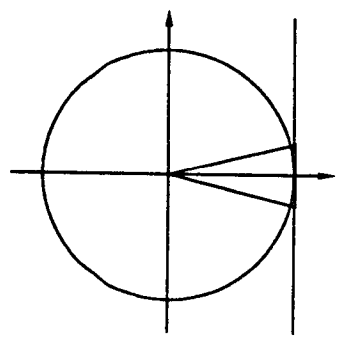
Annexe 2

Contours d'intégration

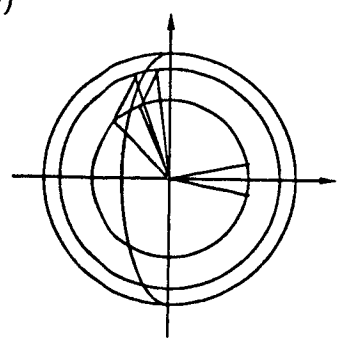
I. Théorèmes de transfert



II. Point col



III. Wyman (un exemple)



Annexe 3

$\Lambda\gamma\Omega$: manuel de référence

I. Specifying the problem in Adl

In order to get average complexity of a given algorithm, one has to specify its datas, procedures and cost measures. Hence we need a special purpose language, like Adl (Algorithm Description Language).

1. Concrete syntax

Notations :

- non-terminals are surrounded with < and >
- optional parts are surrounded with { and }

An Adl file is made of three parts:

<adl_file> ⟶ <type_decls> { <function_decls> } { <measure_decls> }

a) Type declarations

Implemented constructors are:

- the union, denoted by |,
- the product; the cartesian (or partitional) product of a, b and c is either denoted by **product**(a, c, b) or by (a, b, c) or by $a(b, c)$,
- the sequence denoted by **sequence**,
- the constructor set denoted by **set**,
- the constructor cycle denoted by **cycle**,
- the constructor unoriented cycle denoted by **ucycle**

For **sequence**, **set**, **cycle** and **ucycle**, one can specify the cardinality by an optional argument. For example,

cycle(object,**card** = **odd**)

is the set of all (oriented) cycles with an odd number of *objects*.

```
<type_decls>    ⟶  type    <type_decl>    ...    <type_decl>
<type_decl>     ⟶  <ident_list>    =    <struct>    ;
<ident_list>    ⟶  <ident>    ,    ...    ,    <ident>
<struct>        ⟶  <struct>    |    ...    |    <struct>
                ⟶  product  (  <struct>  ,  ...  ,<struct>  )
                ⟶  <constructor> ( <struct> { ,card <card_cond> } )
                ⟶  <ident>
                ⟶  atom        (        <num>        )
                ⟶  Latom       (        <num>        )
                ⟶  epsilon
<constructor>   ⟶  sequence
                ⟶  set
                ⟶  cycle
                ⟶  ucycle
<card_cond>     ⟶  =                    <num>
                ⟶  >=                   <num>
                ⟶  <=                   <num>
                ⟶  =                    even
```

```
                  ⟶  =                                          odd
```

b) Function declarations

```
<function_decls>  ⟶  <function_decl>          ...        <function_decl>
<function_decl>   ⟶  function <ident> ( <fpars> ) : <ident> ; <instruction> ;
                  ⟶  procedure <ident> ( <fpars> ) ; <instruction> ;
<fpars>           ⟶  <fpar>       ;          ...        ;          <fpar>
<fpar>            ⟶  <ident_list>          :                  <ident>
<instruction>     ⟶  case <ident_list> of <cases> { ; <otherwise> } end
                  ⟶  begin <instruction> ; ... ; <instruction> end
                  ⟶  if <condition> then <instruction> else <instruction>
                  ⟶  forall   <ident>   in   <ident>   <instruction>
                  ⟶  <expression>
<condition>       ⟶  <struct>              =              <pattern>
                  ⟶  <ident>          (          <ident>          )
<expression>      ⟶  <ident>
                  ⟶  <ident>  (  <expression>  ,  ...  ,<expression>  )
<otherwise>       ⟶  otherwise          :                  <instruction>
<cases>           ⟶  <one_case>      ;          ...      ;          <one_case>
<one_case>        ⟶  <pattern>  ,  ...  ,  <pattern>  :   <instruction>
<pattern>         ⟶  <ident>
                  ⟶  (   <pattern>   ,   ...   ,   <pattern>   )
<ident>           ⟶  any          sequence          of          letters
<num>             ⟶  an                                                  integer
```

2. Measure declarations

```
<measure_decls>   ⟶  measure <measure_decl> ... <measure_decl>
<measure_decl>    ⟶  <ident_list>    :        <cost>    ;
<cost>            ⟶  <num>
                  ⟶  <ident>
```

3. Semantic

a) Types

- there are no predefined types
- **atom**(n) (resp. **Latom**(n)) means an unlabelled (resp.labelled) atom of size n. The same file can not contain both. So either all atoms are labelled, or they are unlabelled.
- **epsilon** is the empty sequence, and is unity for **product**. Synonyms are **atom**(0), **Latom**(0), **product**(). So **product**(foo,**epsilon**) is the same as foo.
- order between type declarations is irrelevant.
- **card** is useful to restrict the number of components. For example, **cycle**(foo,**card**=odd) means all cycles with an odd number of components, which are taken in foo.

b) Functions and procedures

- order between functions or procedures is irrelevant.
- in `<function_decl>`, the first `<ident>` is the name of the function (or procedure), the second one is the function type.
- a function type is either a defined type, or **boolean**. A boolean function is called a predicate and must return either **true** or **false**. The value returned by a function is the last expression evaluated.
- all parameters are passed by value
- in `<fpar>`, `<ident>` must be one of the above defined types
- in `<instruction/case>`, all variables appearing in `<ident_list>` must have been linked somewhere above (in formal parameters, in a **forall** or in another **case**).
- in **forall** a **in** c `<instruction>`, c must be a unary constructor (**sequence, set, cycle or ucycle**).
- in the second production of `<condition>`, the first `<ident>` is the name of a boolean function
- in the second production of `<expression>`, `<ident>` is a procedure or a function name.

- in <cases>, only the first instruction corresponding to a matching pattern is executed.

c) Measures

- all names appearing in these declarations must appear in function declarations.
- when a name appears in a function, and not in measures, one assumes that its cost is 0.
- the second production of <cost> allows to handle formal costs. One has to verify there is no conflict with type names.

II. Running ΛγΩ

Once you have written an Adl program for your problem, you may want to analyze it.

1. Caml commands

First start a MapleCaml session, and load ΛγΩ:

```
% maplecaml
CAML (sun) (V 2-5) by INRIA Fri Jan 22
#load"luo";;
Initializing maple ...
/Luo/Caml/luo.lo loaded
() : void
```

then you can type:

analyze"foo";;

start analysis of file *foo.adl*.

examples();;

lists all file's names with suffix *.adl*.

order:=*p*;;

sets the number of terms in asymtotic expansions to p (default is 1).

printlevel:=k

sets the quantity of informations displayed (default is 1). Each information appears on and after its level: average cost number of inputs and total cost counting generating functions and complexity descritors (in z) solutions found by Solve (before assignation) equations produced by Alas.

evalf:=*bool*;;

enables (resp. disables) floating point evaluation of the average cost when $bool = true$ (resp. $false$, which is the default).

2. Analyzing part of the functions

If you have defined many functions in your Adl file, and you want to analyze only some of them, just add the following line at the end of it:

`to_analyze : name1, name2, ..., nameN;` where `name1`, ..., `nameN` are the names of the functions you want to analyze. Otherwise, all functions are analyzed.

Annexe 4

Plus près du programme

I. Installation

Le programme se décompose en deux parties : un fichier **equiv** qui doit être interprété à chaque exécution et une arborescence de procédures préinterprétées (au format .m de Maple) accompagnées de leur source, la racine ayant pour nom **Equiv**.

Le fichier **equiv** comporte la redéfinition de certaines procédures des librairies de Maple dont le comportement n'est pas satisfaisant ainsi qu'un appel à **readlib** sur **equivalent** qui charge le programme lui-même de façon quasi-instantanée, les différentes parties n'étant chargées qu'au fur et à mesure des besoins.

La meilleure manière de le porter est d'emporter ce fichier et cette arborescence, puis de préinterpréter à l'arrivée les différentes sources après avoir mis dans la définition de la procédure **cat2** le path du directory **Equiv**.

Eventuellement, si **equiv** et **Equiv** ne sont pas dans votre homedir, il peut être bon de mettre dans votre fichier .mapleinit quelque chose comme

```
equiv := '/udd/algo/lib/maple/equiv' :
```

II. Utilisation courante

Après avoir lancé Maple, on charge le programme par

```
read(equiv);
```

et on l'appelle sous la forme :

```
equivalent(expr,ordre)
```

où **expr** est une expression en une seule variable et **ordre**(optionnel) est une indication sur le nombre de termes du résultat (ie ordre=4 peut donner beaucoup plus de quatre termes).

equivalent retourne une expression Maple qui porte moins d'information que le format sous lequel elle est stockée lors des calculs. Ce format est accessible par la fonction **equiv** (même syntaxe), dont l'utilisateur doit se servir lorsqu'il souhaite effectuer la division de deux développements. Une utilisation type pourra être :

```
f:=expr1;
g:=expr2;
equivalent(f,4);
equivalent(g,4);
dividev(equiv(f,4),equiv(g,4));
```

III. Le programme

1. Le cœur du programme

```
cat2:=proc(x) option remember; cat('/usr/users/labos/dea/salvy/Equiv/EnCours',x) end;
##########################################################################
#
#                              EQUIVALENT
#
#  C'est le programme principal (au sens du main C).
```

```
# On lui passe deux arguments, dont le premier est la fonction
# et le second (optionnel) est l'ordre souhaite pour le developpement
# (ou plutot une indication sur la profondeur du resultat.
#
#####################################################################

equivalent:=proc()
equiv(args);
if type(",list) then printdev(") else " fi;
end:

equiv:=proc()
local sing,fct,p,i,x;
option remember;
   if assigned(n) then ERROR('Unassign n, please') fi;
   if nargs=0 or 3<nargs
      or not(type(args[1],algebraic))
      or nargs>1 and not((type(args[2],integer) and args[2]>0) or
                         (nargs=2 and type(args[2],'name')))
      or nargs=3 and not type(args[3],'name')
      then ERROR( 'invalid arguments ') fi;
   fct:=normal(args[1]);
   if nargs=3 or nargs=2 and type(args[2],integer) then p:=args[2]
   else p:=1 fi;
   map(proc(x) if type(x,'name') then x fi end,
         map( op,indets(fct)))minus{gamma,Pi,E};
   if nargs=3 then {args[3]} intersect "
   elif nargs=2 and type(args[2],'name') then {args[2]} intersect " fi;
   if nops(")=1 then
      x:=op(")
   elif nops(")=0 then RETURN([[],[0]])
   else ERROR('bad variables') fi;
   sing:=infsing(fct,x,0,p)[1];
   if "<>[] and not has(",Float(10,100)) then
      if hastype(sing,float) then printformat:=0 fi;
      equsing(eval(subs(L=proc(x)ln(1/(1-x))end,Q=proc(x)1/(1-x)end,fct)),
            map(proc(y)subsop(2=eval(subs(L=proc(x)ln(1/(1-x))end,
                                          Q=proc(x)1/(1-x)end,y[2])),y) end,
                  sing),p,x);
   else
      print('No singularities found');
      'H_HS'(fct,x);
      if "='HS' then
         HS(fct,p,x)
      elif "='H' then
         Hayman(fct,x,infinity)
      elif type(normal(fct),polynom,x) then 0
      else selle(fct,x)
      fi
   fi
end:

#####################################################################
```

```
#
#                          EQUSING
#
#  Cette procedure retourne le developpement asymptotique (lorsqu'elle
# le trouve) des coefficients de la fonction fct, dont on connait les
# singularites de plus faible module, et ce a l'ordre p.
#
#########################################################################
printformat:=1:
equsing:=proc(fct,sing,p,x)
local i,res;
   res:=[0];
   for i to nops(sing) do
      if hastype(sing[i][1],float) then
         Singularity:=sing[i];
         dev(fct,subsop(1=singul,sing[i]),p,p,x,X);
      else
         dev(fct,sing[i],p,p,x,X);
      fi;
      traperror(alglogeq(",p,p,X));
      if "<>'lasterror' then
         res:=addev(res,multbyreal(subs(singul=singularity[i],")
                        ,1/singularity[i]^n),X);
      else
         H_HS_sing("");
         if "='H' then RETURN(Hayman(fct,x,sing[i]))
         else RETURN(HS(fct,x,sing[i]))
         fi
      fi
   od;
   res:=[flat(res)];
   if p<3 and nops(")=1 and nargs=4 then
      RETURN(equsing(fct,sing,p+1,x,1))
   else
      res[nops(res)];
      for i to nops(sing) do subs(singularity[i]=1,") od;
      if "=0 then
         X[1]^(infinity)/(singularity[1])^n
      else
         "/(singularity[1])^n
      fi;
      RETURN([sing,subsop(nops(res)=",res)])
   fi;
end:


#########################################################################
#
#                          INFSING
#
#  Ce bloc est charge de determiner la ou les singularites de
# module minimal de la fonction fct de var.
#
```

```
##################################################################

infsing:=proc(fct,var,singmin,p)
local x,k,val,i,inf,j,indic,X,coefpos,l,lpol,lrac;
option remember;
   x:=var;
   if type(fct,polynom,x) then
      RETURN([[[Float(10,100),0,x]],sgncoefpol(fct,var)])
   fi;
   val:=[[[Float(10,100),0,x]],false];
   if type(fct,'+') or type(fct,'*') then
      indic:={};
      coefpos:=true;
      for i to nops(fct) do
         inf[i]:=infsing(op(i,fct),x,singmin,p);
         if coefpos and not(''[2]) then
            coefpos:=false;
            eval(subs(L=proc(x)ln(1/(1-x))end,Q=proc(x)1/(1-x)end,fct));
            indic:=[solve('',x)];
            if has('',RootOf) then
               lpol:=RootOfwhat(indic);
               for l to nops(lpol) do
                  lrac:=[solve(lpol[l])];
                  if nops('')<degree(lpol[l],_Z) then lrac:=[solvepol(lpol[l],_Z)] fi;
                  for j while j<=nops(indic) do
                     if has(indic[j],'RootOf'(lpol[l])) then
                        indic:=[op(subsop(j=NULL,indic)),subs('RootOf'(lpol[l])=lrac['k'],indic[j])$'k'=1..nops(l
                     fi
                  od
               od
            fi
         fi
      od;
      for i to nops(fct) do
         trie(inf[i][1],val[1],fct,x,singmin,p);
         if ''=[] then RETURN(infsing(fct,var,abs(val[1][1][1]),p)) fi;
         if {op(map(proc(u)op(1,u)end,''))} intersect {op(indic)}={} then
            val:=['',coefpos];
         else
            for j while j<nops(inf[i][1]) do
               if member(op(1,op(j,inf[i][1])),
                        {op(map(proc(u)op(1,u)end,''))} intersect {op(indic)}) then
                  if type(inf[i][1][j][1],float) then
                     Singularity:=inf[i][1][j];
                     dev(eval(subs(L=proc(x)ln(1/(1-x))end,Q=proc(x)1/(1-x)end,fct)),
                           proc(y,x)subsop(2=eval(subs(L=proc(x)ln(1/(1-x))end,
                                       Q=proc(x)1/(1-x)end,y[2])),y) end
                           (subsop(1=singul,inf[i][1][j]),x),
                           p,p,x,X);
                  else
                     dev(eval(subs(L=proc(x)ln(1/(1-x))end,Q=proc(x)1/(1-x)end,fct)),
                           proc(y,x)subsop(2=eval(subs(L=proc(x)ln(1/(1-x))end,
                                       Q=proc(x)1/(1-x)end,y[2])),y) end
```

```
                        (inf[i][1][j],x),
                        p,p,x,X);
                fi;
                traperror(max(op(map(
                    proc(x,X)if type(x,name) and type(x,indexed) and op(0,x)=X then
                        op(x) fi end,
                    map(op,map(listindets,{op(")})),X))));
                if type(",integer) and "=1 then
                    map(rdegree,"",X[1]);
                    if "[1]>=0 and
                        not has(false,map(proc(x) hastype(x,integer) end,")) then
                        inf[i][1]:=subsop(j=NULL,inf[i][1]);
                        j:=j-1
                    fi;
                fi;
            fi
        od;
        trie(inf[i][1],val[1],fct,x,singmin,p);
        if "=[] then RETURN(infsing(fct,var,abs(val[1][1][1]),p)) fi;
        val:=[",coefpos]
    fi
  od
elif type(fct,'^') then
  op(2,fct);
  if type(evalf("),constant) then
    infsing(op(1,fct),x,singmin,p);
    if ""<0 and type("",integer) then
      val:="
    else
      trie(infsolve(op(1,fct),x,"[2],singmin,p),"[1],fct,x,singmin,p);
      val:=[",false]
    fi
  else val:=infsing(exp(op(2,fct)*ln(op(1,fct))),x,singmin,p) fi
elif type(fct,function) then
  op(0,fct);
  if "=sqrt then
    val:=infsing(op(1,fct)^(1/2),x,singmin,p)
  elif member(",{log,ln}) then
    op(1,fct);
    infsing(",x,singmin,p);
    val:=[trie(infsolve("",x,"[2],singmin,p),"[1],fct,x,singmin,p),false]
  elif "='exp' then val:=infsing(op(1,fct),x,singmin,p)
  elif "='Q' or "='L' then # then we assume the coeff to be >=0
    infsolve(op(1,fct)-1,x,true,singmin,p);
    if not has(",Float(10,100)) then
      val:=[infsolve(op(1,fct)-1,x,true,singmin,p),true]
    else val:=[infsing(op(1,fct),x,singmin,p)[1],true]
    fi
  fi
else ERROR('Invalid expression',fct)
fi;
if val=[[Float(10,100)]] then val:=[] fi;
val
```

```
    end:

sgncoefpol:=proc(pol,x)
local i;
if not type(pol,'*') and not type(pol,'^') then
    for i from 0 to degree(pol,x) do
        if signum(coeff(pol,x,i))<>1 then RETURN(false) fi
    od;
    RETURN(true)
elif type(pol,'*') then
    for i to nops(pol) do
        if not sgncoefpol(op(i,pol),x) then RETURN(false) fi
    od;
 RETURN(true)
elif type(pol,'^') then
    RETURN(sgncoefpol(op(1,pol),x))
fi
end:

# Pour tester si une expression devient trop grosse.
sizeof:=proc(expr,n)
local i;
if type(expr,name) or type(expr,integer) then 1
elif type(expr,'+') or type(expr,'*') or type(expr,'^') or type(expr,function) then
    1;
    for i to nops(expr) while "<>false do
        sizeof(op(i,expr),n-");
        if "<>false then "+""; if "<n then " else false fi else " fi
    od;
elif type(expr,float) then 1
elif type(expr,numeric) then 1
else
    1;
    for i to nops(expr) while "<>false do
 sizeof(op(i,expr),n-");
 if "<>false then "+"" else " fi
 od;
fi
end:


'addev':='readlib('addev',cat2('/Equiv/dev/addev.m'))';
'alglogeq':='readlib('alglogeq',cat2('/Equiv/alglogeq.m'))';
'comparedev':='readlib('comparedev',cat2('/Equiv/comparedev.m'))';
'comparemodule':='readlib('comparemodule',cat2('/Equiv/comparemodule.m'))';
'comparemonome':='readlib('comparemonome',cat2('/Equiv/comparemonome.m'))';
'comparexpr':='readlib('comparexpr',cat2('/Equiv/comparexpr.m'))';
'dev':='readlib('dev',cat2('/Equiv/dev.m'))';
'dividev':='readlib('dividev',cat2('/Equiv/dev/dividev.m'))';
'expdev':='readlib('expdev',cat2('/Equiv/dev/expdev.m'))';
'flat':='readlib('flat',cat2('/Equiv/flat.m'))';
'gdev':='readlib('gdev',cat2('/Equiv/dev/gdev.m'))';
'Hayman':='readlib('Hayman',cat2('/Equiv/pointcol/Hayman.m'))';
```

```
'H_HS':='readlib('H_HS',cat2('/Equiv/pointcol/H_HS.m'))';
'HS':='readlib('HS',cat2('/Equiv/pointcol/HS.m'))';
'indexify':='readlib('indexify',cat2('/Equiv/dev/indexify.m'))';
'infsolve':='readlib('infsolve',cat2('/Equiv/infsolve.m'))';
'instanc':='readlib('instanc',cat2('/Equiv/dev/instanc.m'))';
'jungen':='readlib('jungen',cat2('/Equiv/alglogeq/jungen.m'))';
'listindets':='readlib('listindets',cat2('/Equiv/dev/listindets.m'))';
'lndev':='readlib('lndev',cat2('/Equiv/dev/lndev.m'))';
'multbyreal':='readlib('multbyreal',cat2('/Equiv/dev/multbyreal.m'))';
'nbzeros':='readlib('nbzeros',cat2('/Equiv/polsolve/nbzeros.m'))';
'ncoeftayl':='readlib('ncoeftayl',cat2('/Equiv/ncoeftayl.m'))';
'pointcol':='readlib('pointcol',cat2('/Equiv/pointcol/pointcol.m'))';
'powdev':='readlib('powdev',cat2('/Equiv/dev/powdev.m'))';
'prdev':='readlib('prdev',cat2('/Equiv/dev/prdev.m'))';
'printdev':='readlib('printdev',cat2('/Equiv/dev/printdev.m'))';
'racinesenayantmodule':='readlib('racinesenayantmodule',cat2('/Equiv/polsolve/racinesenayantmodule.m'))';
'rdegree':='readlib('rdegree',cat2('/Equiv/rdegree.m'))';
'RootOfwhat':='readlib('RootOfwhat',cat2('/Equiv/RootOfwhat.m'))';
'selle':='readlib('selle',cat2('/Equiv/pointcol/selle.m'))';
'solvepol3':='readlib('solvepol3',cat2('/Equiv/polsolve/solvepol3.m'))';
'translateof1':='readlib('translateof1',cat2('/Equiv/dev/translateof1.m'))';
'transfert':='readlib('transfert',cat2('/Equiv/alglogeq/transfert.m'))';
'trie':='readlib('trie',cat2('/Equiv/trie.m'))';

save equiv,equivalent,equsing,infsing,sgncoefpol,sizeof,printdev,cat2,'addev','alglogeq',
'comparedev','comparemodule','comparemonome','comparexpr','dev',
'dividev','expdev','flat','gdev','Hayman','H_HS',
'HS','indexify','infsolve','instanc','jungen','listindets',
'lndev','multbyreal','nbzeros','ncoeftayl','pointcol',
'powdev','prdev','printdev','racinesenayantmodule','rdegree',
'RootOfwhat','selle','solvepol3','translateof1','transfert','trie','../equivalent.m';
quit
```

2. Les singularités algébrico-logarithmiques

a) Mise en forme

```
#####################################################################
#
#                          ALGLOGEQ
#
#####################################################################
alglogeq:=proc(globdev,prof,p,X)
local maxi,i,result,dev,mini;
   dev:=globdev;
   maxi:=traperror(max(op(map(proc(x)if type(x,name) and type(x,indexed) then op(x) fi end, listindets(dev)))));
   if maxi='lasterror' then RETURN([0])
   elif maxi>2 then ERROR('Maybe in our next version...') fi;
   mini:=traperror(min(op(map(proc(x)if type(x,name) and type(x,indexed) then op(x) fi end, listindets(dev)))));
   if mini<0 then RETURN(H_HS_sing(globdev,p,X)) fi;
   result:=[0];
   if not type(dev,list) then
      rdegree(dev,X[2]);
      if type(",integer) and "<=0 then
         converttojungen(dev,prof,p,X);
         if not type(",list) then
```

```
            result:=jungen(dev,prof,p,X)
        else dev:=[flat(")];
            for i to nops(dev)-1 do
                result:=addev(result,jungen(dev[i],prof,p,X),X)
            od;
        result:=addev(result,Otransf(dev[nops(dev)],X),X)
        fi;
    else result:=transfert(dev,prof,X)
    fi;
  else
    dev:=[flat(dev)];
    for i to nops(dev)-1 do
        result:=addev(result,alglogeq(dev[i],prof,p,X),X)
    od;
    result:=addev(result,Otransf(dev[nops(dev)],X),X)

  fi;
  RETURN(result)
end:

Otransf:=proc(expr,X)
    expand(expr);
    if testeq(",0)<>true then
        rdegree(",X[1]);
        rdegree("",X[2]);
        if type(",integer) and "">=0 then
            if "<>0 then
                # If you read this, this line may help you :
                [X[1]^(1+"")*X[2]^("+1)]
            else [X[1]^(1+"")]
            fi;
        else [X[1]^(1+"")*X[2]^"]
        fi;
    else [0]
    fi;
end:

converttojungen:=proc(expr,p,prof,X)
local i;
  if not has(expr,X[2]) then expr
  elif not type(expr,list) then
    rdegree(expr,X[2]);
    multbyreal(powdev([X[2]*X[1]^'k'$'k'=0..p],",p,X),subs(X[2]=1,expr))
  else
    [0];
    for i to nops(expr)-1 do
      addev(",converttojungen(proc(x)
        if type(x,list) and x[nops(x)]<>0 then [op(x),0] else x fi end(
          expr[i]),p,prof,X),X)
    od;
    addev(",[expr[i]],X);
  fi
end:
```

```
H_HS_sing:=proc(dev)
   traperror(min(op(map(proc(x)if type(x,name) and type(x,indexed) then op(x) fi end, listindets(dev[1])))));
   if "=-1 then 'H'
   else 'HS'
   fi
end:

save alglogeq,Otransf,converttojungen,H_HS_sing,'../alglogeq.m';
quit
```

b) Le théorème de Jüngen

```
##################################################################
#
#                          JUNGEN
#
#  Retourne le comportement asymptotique des coefficients pour
# l'expression algebrico-logarithmique donnee en argument sous
# la forme X**r Y**k ou X represente (1-x) et Y represente
# ln(1/(1-x)).
#
##################################################################
jungen:=proc(expr,p,prof,X)
local k,s,devalg;
   if expr<>0 then
      k:=-rdegree(expr,X[2]);
      if "=Float(10,100) then k:=0 fi;
      if "<0 then ERROR('Expression en 1/ln(x)') fi;
      s:=-rdegree(expr,X[1]);
      devalg:=multbyreal(purjungen(s,k,p+1,prof,X),expand(expr/(X[1]^(-s)*X[2]^(-k))));
   else RETURN([0]);
   fi;
   RETURN(devalg)
end:

purjungen:=proc(s1,k1,p,prof,X)
local Y,n,k,s,m,res,var,j,g,al,bern,t,z,x,
      a,h,f,i,w,xx,devalg,deg,expr1,expr2,inter,uu,var2,vv;
option remember;
   k:=k1;
   s:=s1;
   if not("<=0 and type(",integer)) then
      res:={};var:={};
      for i from 0 to k do
         0;
         g:=GAMMA(z);
         for j from i by -1 to 0 do
            ""+xx[j]*binomial(i,j)*subs(z=s,g);
            g:=diff(g,z);
         od;
         res:=res union {""=a[i]};
         var:=var union {xx[i]}
      od;
```

```
    res:=(solve(res,var));
    for i while op(1,op(i,res))<>xx[k] do od;
    res:=op(2,op(i,res));
    al:=exp((1/t+s-1/2)*ln(1+s*t)-(1/t+1/2)*ln(1+t)+1-s);

    g:=sqrt(2*Pi)*exp(eval(subs(bern=bernoulli,
        sum(1/x**(2*'m'-1)*bern(2*'m')/2/'m'/(2*'m'-1),'m'=1..p+1))));
    h:=exp((x-1/2)*ln(x)-x);
    f:=g*h;
    for i from 0 to k do
        for j from 0 to i do
            normal(f/h);
            traperror(coeff(convert(taylor(subs(ln(x)=X,''),X,j+1),polynom),X,j));
            if ''<>lasterror then w[j]:=''
            elif ''=lasterror and not has('',ln(x)) then
                if j=0 then w[0]:=''
                else w[j]:=0
                fi;
            else ERROR()
            fi;
        od;
        f:=diff(f,x);
        a[i]:=subs(x=s+n+k1,sum(w['j']*(ln(x))**'j','j'=0..i))
                /subs(x=n+k1+1,w[i])
    od;
    res:=normal(al*subs(ln(s+k1+n)=Y+ln(1+(s+k1)*t),n=1/t,eval(res)));
    devalg:=[];
    for i while (nops(devalg)<=p and i<2*p) do
        for j from k by -1 to 0 while nops(devalg)<p do
            ncoeftayl(coeff(convert(taylor(res,Y,j+1),polynom),Y,j),t,i-1);
            if comparexpr('',0)<>'=' then
                devalg:=[op(devalg),(ln(n))**j*n**(s-i)*subs(t=1/n,'')];
            fi;
        od;
    od;
    if nops(devalg)=p then
#       if j>0 then
#           devalg:=[op(devalg),(ln(n))**(j-1)*n**(s-i)]
    elif nops(devalg)<p and devalg[nops(devalg)]<>0 then
        devalg:=[op(devalg),0]
    else devalg:=[op(devalg),(ln(n))**k*n**(s-i)]
    fi;
    inter:=map(proc(x,n,X)subs(ln(n)=X[2]^(-1),n=X[1]^(-1),x)end,devalg,n,X);
    expr1:=map(rdegree,'',X[1]);
    if nops('')<>nops({op('')}) then
        for i from 2 while i<=nops(expr1) do
            if expr1[i]=expr1[i-1] then
                if not type(subs(X[1]=1,inter[i-1]),list) then
                    inter:=subsop(i-1=subs(X[1]=1,[inter[i-1],inter[i]])
                            *X[1]^expr1[i],i=NULL,inter)
                else inter:=subsop(i-1=[op(subs(X[1]=1,inter[i-1])),
                        subs(X[1]=1,inter[i])]*X[1]^expr1[i],i=NULL,inter)
                fi;
```

```
                expr1:=subsop(i=NULL,expr1);
                i:=i-1
            fi
        od
    fi;
    RETURN(inter)
elif k>0 then
    jungen(s*X[1]**(-s-1)*X[2]**(-k),p,prof,X);
    jungen(k*X[1]**(-s-1)*X[2]**(-k+1),p,prof,X);
    addev("","",X);
    if nops('')<prof+1 and p<prof+3
        and ('''[nops(''')]<>0 or ""[nops("")]<>0) then
        RETURN(purjungen(s1,k1,p+1,prof,X))
    fi;
    devalg:=prdev([X[1],0],translateof1([flat('')],prof,X),X);
    if nops(devalg)<p+1 and devalg[nops(devalg)]<>0 and p<prof+3 then
        devalg:=[op(devalg),0]
    fi;
 else
    devalg:=[0];
fi;
RETURN(devalg)
end:

save jungen,purjungen,'../jungen.m';
quit
```

c) Les théorèmes de transfert

```
#####################################################################
#
#                          TRANSFERT
#
#####################################################################
transfert:=proc(dev,prof,X)
local al,k,s,gam,x;
    al:=rdegree(dev,X[1]);
    gam:=-rdegree(dev,X[2]);
    if not(type(al,integer) and al>=0) then
        RETURN(prdev([X[1]^(al+1)/GAMMA(-al)*X[2]^(-gam),0],
            subs(s=al,[1,(GAMMA(-al)*binomial(k-gam-1,k)*
                diff(1/GAMMA(-s),s$k)*X[2]^k)$k=1..prof]),X))
    else
        convert(taylor(1/taylor(GAMMA(x),x=al,prof+1),x=al,prof+1),polynom);
        RETURN(prdev([X[1]^(al+1)*X[2]^(-gam),0],
            [binomial(k-gam-1,k)*coeff('',(x-al),k)*factorial(k)*X[2]^k
                $k=1..prof+1]),X)
    fi
end:

save transfert,'../transfert.m';
quit
```

3. Les méthodes de point col

a) Admissibilités

```
#####################################################################
#
#                              H_HS
#
#  Cette procedure doit determiner si une fonction entiere est
#  H-admissible ou meme HS-admissible.
#  Elle renvoie FAIL sinon.
#
#####################################################################

H_HS:=proc(fct,var)
local j,polprd,indic,i,pol,d,autre,fonc,coeffi,x,X;
   x:=var;
   if type(fct,polynom,x) then RETURN(FAIL)
   elif type(fct,'*') then
      polprd:=1;
      indic:=true;
      for i to nops(fct) do
         op(i,fct);
         if type(",polynom,x) then polprd:=polprd*"
         elif member(H_HS(",x),{'H','HS'}) then indic:=false
         else RETURN(FAIL)
         fi
      od;
      if indic then RETURN(FAIL)
      elif comparexpr(lcoeff(expand(polprd),x),0)='>' then RETURN('H')
      else RETURN(FAIL)
      fi;
   elif type(fct,'^') then
      op(2,fct);
      if type(evalf("),constant) and (">0) and type(",integer) then
         H_HS(op(1,fct),x)
      else H_HS(exp("*ln(op(1,fct))),x)
      fi;
   elif type(fct,function) and op(0,fct)='exp' then
      pol:=op(1,fct);
      if member(H_HS(",x),{'H','HS'}) then RETURN('HS')
      elif type(",polynom,x) then
         d:=degree(pol,x);
         for i from 2 to d do
            j:=d;
            while j>1 do
               if modp(j,i)<>0 then j:=j-1
               else coeff(pol,x,j);
                  comparemodule(",0);
                  if "='=' then j:=j-1
                  elif "='<' then RETURN(FAIL)
                  elif "='>'then j:=1
                  else ERROR('Trop de variables')
                  fi
               fi
            od
         od;
```

```
        RETURN('H')
      fi
   elif type(fct,'+') then
      indic:=0;
      autre:=0;
      coeffi:=[0];
      for i to nops(fct) do
         op(i,fct);
         if not type(",polynom,x) then
            if member(H_HS(",x),{'H','HS'}) then
               dev(",[infinity,0],1,1,x,X);
               comparedev(",coeffi,X);
               if "='>' then
                  coeffi:="";
                  fonc:="";
                  autre:=0;
               elif "='=' then autre:=autre+op(i,fct)
               fi
            else autre:=autre+op(i,fct)
            fi
         fi
      od;
      if coeffi=[0] then RETURN(FAIL)
      elif autre=0 then RETURN('H')
      else print('Je vais vous proposer deux fonctions f1 et f2 pour');
         print('lesquelles j'ai besoin du renseignement suivant : ');
         print('si f2=O(f1^(1-d)) avec d>0, repondez O, sinon repondez N');
         print('f1:',fonc);
         print('f2:',autre);
         readstat('Votre reponse :');
         if "='O' or "='o' then RETURN('H')
         else RETURN(FAIL)
         fi
      fi
   else RETURN(FAIL)
   fi;
   RETURN(");
end:

save H_HS,'../H_HS.m';
quit
```

b) Le cas le pire

```
##########################################################################
#
#                              SELLE
#
#  Cette procedure applique la methode de point selle en demandant
# a l'utilisateur d'en faire lui-meme la justification.
#
##########################################################################

selle:=proc(fct,var)
local u,h,x,i;
```

```
    x:=var;
    h:=ln(fct)-(n+1)*ln(x);
    traperror([solve(diff(",x)=0,x)]);
    u:=";
    if nops(u)¿0 then
        op(1,");
        if nops(u)¿1 then
            for i from 2 to nops(u) do
                if comparexpr(op(i,u),")='>' then op(i,u) fi od fi
    else
        print('En appelant u la solution de :');
        print(cat(h,'=0 ,'));
        lprint('qui se developpe en',pointcol(fct,var,1));
        u:='u'
    fi;
    u:=";
    print('si vous admettez que les hypotheses necessaires a l'application de');
    print('la methode de point selle sont verifiees pour z=');
    print(u);
    print('alors le resultat est :');
    RETURN(subs(x=u,fct)/u^(n+1)/sqrt(2*Pi*subs(x=u,diff(diff(h,x),x))));
end:

save selle,'../selle.m';
quit
```

c) H-admissibilité

```
##########################################################################
#
#                                   Hayman
#
#  Cette procedure applique les formules de Hayman pour les fonctions
# H-admissibles.
#
##########################################################################

Hayman:=proc(fct,var,sing)
local a,b,r,i,x,X;
    x:=var;
    a:=x*diff(fct,x)/fct;
    b:=x*diff(a,x);
    r:=[traperror(solve(a=n,x))];
    if nops(r)¿0 and not has(r,W) then
        op(1,");
        for i from 2 to nops(r) do
            if comparexpr(op(i,r),")='>' then op(i,r) fi od;
        RETURN(printdev([1,multbyreal(dev(subs(x=",n=[X[1]^(-1),0],
                fct/b^(1/2)),[infinity,0],1,1,x,X),1/(2*Pi)^(1/2)/
                op(1,flat(dev(subs(n=[X[1]^(-1),0],"),[infinity,0],1,1,x,X)))^n)]))
    else
        if not has(r,W) then
            print('En appelant r la solution de :');
            lprint(cat(a,'=n'))
        else
```

```
        lprint('Le point col est r=',")
      fi;
      if sing=infinity then
        traperror(pointcol(fct,var,1));
        if "<>'lasterror' then
          lprint('qui se developpe en ',")
        fi
      fi;
      print('Le resultat est :');
      r:='r';
      RETURN(simplify(subs(x=",fct)/"^n/sqrt(2*Pi*subs(x=",b))))
  fi;
end:

save Hayman,'../Hayman.m';
quit
```

d) HS-admissibilité

```
##########################################################################
#                                  HS
#
#  Cette procedure applique les formules de Harris et Schoenfeld pour les
# fonctions HS-admissibles.
#
##########################################################################

HS:=proc(fct,p,var)
local b,g,c,u,d,i,j,k,l,res,somme,s,q,m,prod,x;
   x:=var;
   k[0]:=0;
   diff(fct,x)/fct;
   for i from 1 to 2*p+2 do
      b[i]:="*x^i/i!;
      diff("",x) od;
   c:=x*diff(b[1],x)/2;
   u:=[solve(b[1]=n+1,x)];
   traperror(op(1,"));
   if nops(u)>1 then
      for i from 2 to nops(u) do
        if comparexpr(op(i,u),")='>' then op(i,u) fi od
   elif "='lasterror' then
      print('En appelant u la solution de :');
      lprint(cat(b[1],'=n+1'));
      lprint('qui se developpe en',pointcol(fct,var,p));
      print('Le resultat est :');
      u:='u'
   fi;
   u:=";
   d:=subs(x=",c);
   for i to 2*p do g[i]:=-(subs(x=u,b[i+2]+(-1)^i/(i+2)*b[1]))/d od;
   res:=1;
   for i to p do
      somme:=0;
      for j to 2*i do
```

```
            k[1]:=2*i-j+1;
            s:=0;
            for l from 2 to j do k[l]:=1 od;
            m:=1;
            while k[1]>0 and m>0 do
               prod:=1;
               for q to j do prod:=prod*g[k[q]] od;
               s:=s+prod;
               m:=j-1;
               while k[m]=1 do m:=m-1 od;
               k[m]:=k[m]-1;
               k[m+1]:=k[j]+1;
               k[j]:=1
            od;
            somme:=somme+GAMMA(j+i+1/2)/j!*s;
         od;
         res:=res+somme*(-d)^(-i)/sqrt(Pi)
      od;
      res:=res+o(d^(-n));
      res:=subs(x=u,fct)/2/u^n/sqrt(Pi*d)*res;
      RETURN(")
end:

save HS,'../HS.m';
quit
```

e) Le développement du point col

```
pointcol:=proc(f,x,p)
printdev([[[1]],ptcol(dev(x*diff(f,x)/f,[infinity,0],1,1,x,X),
          [X[1]^(-1),0],p,x,X)])
end:

ptcol:=proc(expr1,expr2,p,x,X)
   map(proc(x,X)if type(x,name) and type(x,indexed) and op(0,x)=X then
         op(x) fi end,
      map(op,map(listindets,{op(expr1)})),X);
   if "={1} and rdegree(expr1[1],X[1])=-1 then
      RETURN(multbyreal(expr2,1/subs(X[1]=1,expr1[1]))) fi;
   map(proc(x,X)if type(x,name) and type(x,indexed) and op(0,x)=X then
         op(x) fi end,
      map(op,map(listindets,{expr1[1]})),X);
   if nops(")=0 then
      RETURN(ptcol(subs(1=NULL,expr1),addev([-expr1[1],0],expr2,X),p,x,X))
   elif "={1} then
      rdegree(expr1[1],X[1]);
      if "<>-1 then
         RETURN(ptcol(op(1,expr1),powdev(expr2,1/op(2,expr1),p,X),p,x,X))
      else
         multbyreal(expr2,1/subs(X[1]=1,expr1[1]));
         multbyreal(subsop(1=NULL,expr1),-1/subs(X[1]=1,expr1[1]));
         iter("",instanc(",X)[1],p,X,x)
      fi
   elif nops(")=1 and op(")<1 then
      RETURN(ptcol(lndev(expr1,X,p,1),lndev(expr2,X,p,1),p,x,X))
```

```
    elif nops(")=1 and op(")>1 then
        RETURN(ptcol(expdev(expr1,X,p,1),expdev(expr2,X,p,1),p,x,X))
    elif nops(expr2)>2 or expr2[nops(expr2)]<>0 then
        RETURN(ptcol(lndev([expr1[1],0],X,p,1),lndev(expr2,X,p,1),p,x,X))
    else RETURN(ptcol(lndev(expr1,X,p,1),lndev(expr2,X,p,1),p,x,X))
    fi
end:

iter:=proc(val,expr,p,X,x)
local i;
val;
for i to p do
    addev(val,dev(subs(_x=",expr),[infinity,0],p,p,x,X),X);
    if nops(")>3 then [op(1..3,")] else " fi;
od;
end:

save makecompare,iter,pointcol,ptcol,'../pointcol.m';
quit
```

4. La recherche des solutions d'équations

a) Cas général

```
##########################################################################
#
#                    INFSOLVE
#
#  Cette fonction retourne, quand elle la trouve,
#  la liste des racines de module minimal de la
#  fonction passee en argument.
#
##########################################################################

infsolve:=proc(fct,var,coefpos,minsing,p)
local x,sol,val,nbr,i,j,lpol,lrac,expr;
    x:=var;
    eval(subs(L=proc(x)ln(1/(1-x))end,Q=proc(x)1/(1-x) end,fct));
    sol:=[solve(",x)];
    if has(",RootOf) then
        lpol:=RootOfwhat(sol);
        for i to nops(lpol) do
            lrac:=[solve(lpol[i])];
            if nops(")<degree(lpol[i],_Z) then lrac:=[solvepol(lpol[i],_Z)] fi;
            for j while j<=nops(sol) do
                if has(sol[j],'RootOf'(lpol[i])) then
                    sol:=[op(subsop(j=NULL,sol)),subs('RootOf'(lpol[i])=lrac['k'],sol[j])$'k'=1..nops(lrac)]
                fi
            od
        od
    fi;
    if type(fct,polynom,x) and nops(")<>degree(fct,x) then
        if nops(indets(fct) minus {gamma,Pi,E})<2 then
            if minsing=0 then
                sol:=solvepol3(collect(fct,x),x)
            else sol:=solvepol(collect(fct,x),x,true,true)
```

```
        fi;
        else ERROR('Wrong number of variables')
      fi
   fi;
   if "<>[] then
      for i while nops(sol)>0 and comparemodule(sol[i],minsing)<>'>' do
         sol:=subsop(i=NULL,sol);i:=i-1 od;
#     for i while nops(sol)>0 and 'comparemodule'(subs(x=sol[i],fct),0)<>'=' do
#        sol:=subsop(i=NULL,sol);i:=i-1 od;
      if sol=[] then RETURN(infsolve(fct/(x-minsing),x,coefpos,minsing,p)) fi;
      val:=[sol[1]];
      nbr:=nops(sol);
      if nbr<>1 then
         for i from 2 to nbr do
            comparemodule(val[1],sol[i]);
            if "='=' then
                  val:=[op(val),sol[i]]
            elif "='>' then val:=[sol[i]]
            fi
         od;
         val
      fi;
      for i while(i<=nops(val)) do
         val:=subsop(i=[op(i,val),fct,1,x],val);
         for j from i+1 while j<=nops(val) do
            comparexpr(op(1,op(i,val)),op(j,val));
            if "='=' then
               val:=subsop(i=(subsop(3=op(3,op(i,val))+1,op(i,val))),j=NULL,val);
               j:=j-1;
            fi;
         od;
      od;
   elif not type(fct,polynom) then
      eval(subs(L=proc(x)ln(1/(1-x))end,Q=proc(x)1/(1-x) end,fct));
      fsolve(",x,0..1);
      if not type('"',numeric) then
#        if not coefpos then
#           RETURN([[Float(10,100),0,x]])
#        else
            lrac:=infsing(fct,x,minsing,p)[1];
            expr:=eval(subs(L=proc(x)ln(1/(1-x))end,Q=proc(x)1/(1-x) end,fct));
            [];
            for i to nops(lrac) do
               if comparexpr(subs(x=lrac[i][1],expr),0)='=' then
                  [op(''),lrac[i]]
               else "
               fi
            od;
            if "=[] then
               fsolve(expr,x,0..abs(op(1,op(1,lrac))));
               if not type('"',numeric) then RETURN([[Float(10,100),0,x]]) fi;
            else RETURN('')
            fi
```

```
#       fi
      fi;
      val:=[[",fct,x]];
      if coefpos then
         # for functions with positive coefficients, we can get the other roots
         map(proc(y,x) if has(y,x) then y fi end,indets(fct),x);
         map(proc(x) if type(x,'^') then op(2,x) fi end,");
         if nops(")=nops("") then
            map(proc(x) if type(x,integer) then x fi end,");
            if nops(")=nops("") then
               gcd(");
               if ">1 then
                  val:=[[val[1][1]*exp(2*I*'k'*Pi/"),fct,x]$'k'=0.."-1]
               fi
            fi
         fi
      fi
   else val:=[[Float(10,100),0]]
   fi;
val;
end:

save infsolve,'../infsolve.m';
quit
```

b) Les polynômes

α.Méthode de Graeffe

```
##################################################################
#
#                        SOLVEPOL
#
# This is a solver to be used when you do not trust
# fsolve/realpoly.
# Its arguments are :  p the polynom
#                      x its variable
#                      allroots (default true)
#                          a boolean (if false only the roots
#                              of least modulus are returned)
#                      argfrac (default false)
#                          a boolean which mean that you
#                          know the arguments of the roots are p/q*Pi
#                          and that degree(pol)<=200
#
##################################################################

solvepol:=proc()
local p,x,allroots,argfrac,moduli,i,sol,n;
p:=args[1];
x:=args[2];
n:=degree(expand(p,x),x);
if nargs>2 then allroots:=args[3] else allroots:=true fi;
if nargs>3 then argfrac:=args[4] else argfrac:=false fi;
moduli:=graeffe(expand(p),x,allroots);
sol:=[];
```

```
if allroots then
   for i to nops(moduli) while nops(sol)<n do
      racinesenayantmodule(p,moduli[i],x,argfrac);
      if "<>[] then
         sol:=[op(sol),op(")]
      elif i<nops(moduli) then moduli:=subsop(i+1=subsop(2=moduli[i+1][2]+moduli[i][2],moduli[i+1]),moduli)
      fi
   od
else
   sol:=racinesenayantmodule(p,moduli,x,argfrac)
fi;
evalf(sol)
end:

##########################################################################
#
#                              SOLVEPOL3
#
##########################################################################

solvepol3:=proc(p,x)
   RETURN(racinesenayantmodule(p,graeffe(p,x,false),x,true))
end:

##########################################################################
#
#                              GRAEFFE
#
# Cette procedure retourne le plus petit reel r tel que le
# polynome ait une racine de module r.
# Les variables utilisees sont : m le nombre d'iterations
#                                delta la precision
#                                a et b les coefficients des polynomes
#
##########################################################################

graeffe:=proc(p,x,allroots)
local a,b,i,j,l,n,m,delta,r,nonrac,eta,limite,oldDigits,k,pol;
   eta:=evalf(10^(3-Digits));
   r:=[];
   ldegree(p,x);
   if ">0 then
      pol:=expand(p/x^ldegree(p,x));
      if allroots then r:=[[0,""]] fi;
   else pol:=p
   fi;
   n:=degree(pol,x);
   if "=0 then RETURN(r) fi;
   limite:=evalf(ln(eta/10*sqrt(2*Pi*n)/2^(n+1)));
   for m to 100 while evalf(2^m*ln(1-eta)-(m-1)*ln(2))>limite do od;
   m:=min(m,20);
   oldDigits:=Digits;
   Digits:=trunc(evalf(.3*m*(n-1)-log(eta/10)/log(10)));
```

```
    delta:=evalf((sum(binomial(trunc(n/2),'k')^2*(1-eta)^('k'*2^(m-1)),
                        'k'=0..trunc(n/2)))^2
                /(sum(binomial(trunc(n/2),'k')^2*(1-eta)^('k'*2^m),
                        'k'=0..trunc(n/2)))-1);
    for i from 0 to n do b[i]:=coeff(pol,x,i) od;
    for i to m do
       a:=copy(b);
       for j from 0 to n do
          b[j]:=evalf((-1)^j*(a[j]^2+
                2*sum((-1)^l*a[j-l]*a[j+l],l=1..min(n-j,j))))
       od;
    od;
    k:=0;
     for i to n do
       traperror(abs(evalf((a[i]^2/b[i])-(-1)^i)));
       if "<>lasterror and "<=delta then
          (abs(b[i]/b[k]))^(1/(i-k)/2^m);
          if not allroots then
             if nbzeros(pol,evalf(1/("+eta)),x)=0 then
                Digits:=oldDigits;
                RETURN([evalf(1/""),i])
             fi
          else
             r:=[op(r),[evalf(1/"),i-k]];
             k:=i
          fi
       fi
    od;
    Digits:=oldDigits;
    if allroots then
       for i to nops(r) do
          for k from i+1 while k<=nops(r) do
             if evalf(r[i][1])=evalf(r[k][1]) then
                r:=subsop(i=[r[i][1],r[i][2]+r[k][2]],k=NULL,r);
                k:=k-1
             fi
          od
       od
    fi;
    r;
    end:

save solvepol,solvepol3,graeffe,'../solvepol3.m';
quit
```

β.Méthode du résultant

```
racinesenayantmodule:=proc(p,modul,x,argfrac)
local sol,epsilon,res,pol,i,j,oldDigits,conv,n,maxi,a,ww,qq,r,ordre,ray,c;
pol:=expand(p);
n:=degree(pol,x);
r:=op(1,modul);
ordre:=op(2,modul);
maxi:=maxdenom[n];
epsilon:=10^(-Digits+2);
```

```
sol:=[];
# test wether +/- r is a root :
    if testrac(pol,x,r,-r,0,0,epsilon,a,ww,qq) then
        multiplicity(pol,x,r,-r,0,0,epsilon,a,ww,qq);
        sol:=[r$"]
    fi;
    if testrac(pol,x,r,r,0,0,epsilon,a,ww,qq) then
        multiplicity(pol,x,r,r,0,0,epsilon,a,ww,qq);
        sol:=[op(sol),-r$"]
    fi;
# Find other possible roots :
    if nops(sol)<n then
        res:=map(proc(u) op(1,u),-op(1,u) end,
                graeffe(expand(Resultant(pol,r,x)),x,true));
        for i to nops(res) while nops(sol)<ordre do
            if testrac(pol,x,r,res[i],r^2,res[i],epsilon,a,ww,qq) then
                if evalf(4*r^2-res[i]^2)>=0 then
                    multiplicity(pol,x,r,res[i],r^2,res[i],epsilon,a,ww,qq);
                    if argfrac then
                        evalf(arctan((4*r^2-res[i]^2)^(1/2)/2,-res[i]/2)/Pi);
                        convert(",confrac,conv);
                        for j from 2 to nops(conv) while denom(conv[j])<=maxi do od;
                        if conv[j-1]<>0 and conv[j-1]<>1 then
                            sol:=[op(sol),r*exp(I*Pi*conv[j-1])$"""",
                                          r*exp(-I*Pi*conv[j-1])$""""]
                        fi;
                        conv:='conv';
                    else
                        4*r^2-res[i]^2;
                        if evalf(abs("))>10^(2-Digits) then
                            sol:=[op(sol),-res[i]/2+I*"^(1/2)/2$"",
                                          -res[i]/2-I*"^(1/2)/2$""]
                        fi
                    fi
                fi
            fi
        od
    fi;
sol
end:

testrac:=proc(pol,x,r,w,q,p,epsilon,a,ww,qq)
local k,n;
n:=degree(pol,x);
[coeff(pol,x,n-'i')$'i'=0..n];
subs(a=",ww=w,qq=q,recurrence(a,ww,qq,n-1));
subs(a="",ww=w,qq=q,recurrence(a,ww,qq,n));
sum(abs(coeff(pol,x,'i'))*x^'i','i'=0..n);
RETURN(evalb(evalf(abs(subs(x=r*(1+2*epsilon),"))-abs(subs(x=r*(1+epsilon),"))
    -abs(""+1/2*p*""""))>0))
end:

recurrence:=proc(a,w,q,k)
```

```
option remember;
if k=-1 then 0
elif k=0 then a[1]
else expand(a[k+1]-w*recurrence(a,w,q,k-1)-q*recurrence(a,w,q,k-2))
fi
end:

Resultant:=proc(pol,ray,x)
local n,k,q,b,a,r,c,res,z,j,sol;
    Digits:=2*Digits;
    n:=degree(pol,x);
    q:=ray*ray;
    for j from 0 to n do c[j]:=coeff(pol,x,n-j) od;
    b[0]:=1;
    b[-1]:=0;
    c[-1]:=0;
    c[n+1]:=0;
    r[-1]:=0;
    for k from 0 to n do
        b[k+1]:=expand(x*b[k]-q*b[k-1]);
        a[k]:=(-1)^k*sum((c['j']*c['j'+k]-c['j'-1]*c['j'+k+1])*q^(n-'j'-k),'j'=0..n-k);
        r[k]:=r[k-1]+a[k]*b[k]
    od;
    Digits:=Digits/2;
    r[n]
end:

multiplicity:=proc(pol,x,r,w,q,p,epsilon,a,ww,qq)
local m;
gcdex(pol,diff(pol,x),x);
for m while degree(",x)>0 and testrac(",x,r,w,q,p,epsilon,a,ww,qq) do
    gcd(",diff(",x))
od;
m
end:

maxdenom := table([(42)=150,(163)=660,(158)=630,(176)=690,(75)=270,(135)=510,(
126)=462,(58)=210,(11)=30,(91)=330,(178)=690,(27)=90,(111)=420,(107)=420,(43)=
150,(148)=630,(195)=840,(184)=690,(76)=270,(59)=210,(167)=660,(12)=42,(92)=330,
(172)=660,(28)=90,(138)=510,(144)=630,(108)=420,(44)=150,(166)=660,(77)=270,(60
)=210,(13)=42,(93)=330,(121)=462,(29)=90,(177)=690,(109)=420,(139)=510,(45)=150
,(124)=462,(78)=270,(61)=210,(14)=42,(160)=660,(94)=330,(191)=690,(30)=90,(127)
=462,(197)=840,(46)=150,(180)=690,(79)=270,(151)=630,(120)=462,(62)=210,(183)=
690,(15)=42,(155)=630,(95)=330,(133)=510,(31)=90,(147)=630,(196)=840,(47)=150,(
179)=690,(168)=660,(80)=330,(63)=210,(16)=60,(96)=420,(32)=120,(65)=240,(131)=
510,(140)=510,(162)=660,(48)=210,(1)=2,(81)=330,(156)=630,(193)=840,(64)=240,(
17)=60,(97)=420,(153)=630,(170)=660,(33)=120,(188)=690,(66)=240,(154)=630,(49)=
210,(2)=6,(143)=510,(194)=840,(82)=330,(18)=60,(98)=420,(134)=510,(34)=120,(150
)=630,(142)=510,(67)=240,(50)=210,(152)=630,(3)=6,(83)=330,(175)=660,(190)=690,
(19)=60,(99)=420,(35)=120,(174)=660,(68)=240,(164)=660,(182)=690,(51)=210,(157)
=630,(4)=12,(161)=660,(84)=330,(173)=660,(20)=66,(118)=420,(100)=420,(199)=840,
(36)=126,(113)=420,(145)=630,(69)=240,(149)=630,(52)=210,(5)=12,(85)=330,(21)=
66,(185)=690,(119)=420,(101)=420,(37)=126,(70)=240,(53)=210,(6)=18,(86)=330,(22
```

```
)=66,(102)=420,(38)=126,(114)=420,(187)=690,(71)=240,(54)=210,(7)=18,(130)=510,
(87)=330,(23)=66,(115)=420,(103)=420,(39)=126,(189)=690,(72)=270,(171)=660,(55)
=210,(8)=30,(136)=510,(88)=330,(116)=420,(24)=90,(146)=630,(165)=660,(128)=510,
(104)=420,(40)=150,(117)=420,(73)=270,(56)=210,(9)=30,(198)=840,(89)=330,(192)=
840,(181)=690,(200)=840,(137)=510,(25)=90,(141)=510,(105)=420,(112)=420,(41)=
150,(186)=690,(74)=270,(110)=420,(57)=210,(10)=30,(132)=510,(90)=330,(129)=510,
(123)=462,(125)=462,(159)=630,(169)=660,(26)=90,(106)=420,(122)=462]);

save racinesenayantmodule,testrac,recurrence,Resultant,multiplicity,maxdenom,'../racinesenayantmodule.m';
quit
```

γ.Méthode de Schur

```
# For polynomials with real coefficients.
nbzeros:=proc(p,r,x)
    Digits:=Digits+4;
    nbezerounitdisk(subs(x=x*r,p),x);
    Digits:=Digits-4;
    ""
end:

nbezerounitdisk:=proc(p,x)
local n,k;
    [collect(p,x),0];
    n:=degree(p,x);if n=0 then if p=0 then RETURN(1) else RETURN(0) fi fi;
    [0,0];
    for k to n while (abs("")>10^(-Digits+2)) do
        [schur("""[1],x),"""[1]];
        proc(u) if hastype(u,float) then evalf(u) else u fi end(coeff("[1],x,0));
        if ">=0 then """ else ["""[1]+(-1)^"""[2]*(n+1-k),"""[2]+1] fi
    od;
    if ""=0 then
        if """[1]=0 then
            "[1]+(-1)^"[2]*nbezerounitdisk(cohn("""[2],x),x);
        else
            "[1]+(-1)^"[2]*nbezerounitdisk(marden("""[2],x),x);
        fi
    else
        "[1]
    fi
end:

schur:=proc(p,x)
    degree(p,x);
    coeff(p,x,0);
    coeff(p,x,"");
    sum(("*coeff(p,x,'i')-"*coeff(p,x,"""-'i'))*x^'i','i'=0.."""-1)
end:

marden:=proc(p,x)
local n,u,q;
    n:=degree(p,x);
    if abs(coeff(p,x,0))=abs(coeff(p,x,n)) then
        u:=signum(coeff(p,x,0))*signum(coeff(p,x,n));
        for q while(coeff(p,x,q)=u*coeff(p,x,n-q)) do od;
```

```
    else q:=0
    fi;
    schur((x^q+2*signum(coeff(p,x,n-q)-u*coeff(p,x,q))*signum(coeff(p,x,n)))*p,x)
end:

# ne sert que si l'on cherche aussi les racines sur le cercle
cohn:=proc(p,x)
    degree(p,x);
    sum(coeff(p,x,'j')*'j'*x^("-'j'),'j'=0..")
end:

save nbzeros,nbezerounitdisk,schur,cohn,marden,'../nbzeros.m';
quit;
```

5. Développement au voisinage d'un point

a) Point d'entrée

```
#####################################################################
#
#                              DEV
#
#  Cette procedure prend en argument :
#                    fct une fonction dont on connait
#                    a   une singularite([a,P(x)] avec P(a)=0)
#                    n   l'ordre du developpement souhaite.
#                    p   sa profondeur(le nbe de termes non nuls)
#  On retourne un developpement de la fonction dans l'echelle
#  (1-x)**s log(1/(1-x))**k ... representee sous la forme
#  X[1]**s X[2]**k ...,
#  a laquelle on se ramene par transformation affine, le
#  developpement etant un array dont le n+1eme element est le O(reste),
#  et 0 si le developpement est exact.
#
#####################################################################

dev:=proc(fct,a,p,n,var,X)
local i,inter,deve,x,ind,prod;
option remember;
    x:=var;
    if type(fct,list) then RETURN(fct)
    elif type(fct,polynom) then
        ind:=true;
        if op(1,a)<>singul or (op(2,a)<>fct and op(2,a)<>-fct) then
            op(1,a);
            if "<>0 and not has(a,infinity) then
            inter:=collect(subs(x="-X[1]*",convert(
                        taylor(fct,x=",degree(fct,x)+1),polynom)),X[1]);
            elif "=0 then inter:=collect(subs(x=X[1],fct),X[1]) # utilisation parasite
            elif op(1,a)=infinity then
                ind:=false;
                inter:=collect(subs(x=X[1],expand(x^degree(fct,x)*subs(x=1/x,fct))),X[1])
            elif op(1,a)=-infinity then
                ind:=false;
                inter:=collect(subs(x=X[1],expand(x^degree(fct,x)*subs(x=-1/x,fct))),X[1])
            fi;
```

```
    else
        inter:=subs(x=op(1,a),collect(expand(subs(x=x-x*X[1],fct)),X));
        for i from 0 to op(3,a)-1 do
            inter:=subs(coeff(inter,X[1],i)=0,inter)
        od;
    fi;
    deve:=[];degree(inter,X[1]);
    for i from 0 to degree(inter,X[1])
        while (nops(deve)<=n+1 or op(2,a)=fct or op(2,a)=-fct) do
        evalc(coeff(inter,X[1],i));
        if comparemodule('',0)<>'=' then deve:=[op(deve),"*X[1]**i] fi;
    od;
    if nops(deve)<n+1 then deve:=[op(deve),0] fi;
    if ind then " else deve:=[deve['i']/X[1]^degree(fct,x)$'i'=1..nops(deve)] fi;
elif type(fct,'+') then
    deve:=[0];
    for i to nops(fct) while (deve[nops(deve)]=0 or nops(deve)>p) do
        deve:=addev(deve,dev(op(i,fct),a,p,n,x,X),X)
    od;
    if fct=op(2,a) or fct=-op(2,a) then
        if rdegree(deve[1],X[1])=0 then deve:=subsop(1=NULL,deve) fi fi;
     if deve[nops(deve)]<>0 and nops(deve)<=p and n<p+3 then
        deve:=dev(fct,a,p,n+1,x,X) fi;
elif type(fct,'*') then
    deve:=[1,0];
    for i to nops(fct) while (deve[nops(deve)]=0 or nops(deve)>p) do
        deve:=prdev(deve,dev(op(i,fct),a,p,n,x,X),X)
    od;
    if fct=op(2,a) or fct=-op(2,a) then
        if rdegree(deve[1],X[1])=0 then deve:=subsop(1=NULL,deve) fi fi;
    if deve[nops(deve)]<>0 and nops(deve)<=p and n<p+3 then
        deve:=dev(fct,a,p,n+1,x,X) fi;
elif type(fct,'^') then
    op(2,fct);
    if type(evalf(''),constant) then
        powdev(dev(op(1,fct),a,p,n,x,X),'',n,X);
        if ''[nops('')]=0 or nops('')>p or n>p+2 then
            deve:='';
        else deve:=dev(fct,a,p,n+1,x,X)
        fi;
    else RETURN(dev(exp("*ln(op(1,fct))),a,p,n,x,X))
    fi
elif type(fct,function) then
    op(0,fct);
    if "='sqrt' then dev(op(1,fct)^(1/2),a,p,n,x,X)
    elif member('',{ln,log}) then
        if not has(a,infinity) then
            deve:=lndev(dev(op(1,fct),a,p,n+1,x,X),X,n);
        else
            deve:=lndev(dev(op(1,fct),a,p,n+1,x,X),X,n,1);
        fi
    elif "='exp' then
        if type(op(fct),'*') then
```

```
            prod:=1;ind:=0;
            for i to nops(op(fct)) do
               if not has(op(i,op(fct)),x) then prod:=prod*op(i,op(fct))
               elif type(op(i,op(fct)),function) and
                     op(0,op(i,op(fct)))='ln' then
                  if ind=0 then ind:=i else ind:=-1 fi;
               else ind:=-1
               fi;
            od;
            if ind>0 then
         RETURN(dev(op(op(ind,op(i,fct)))^prod,a,p,n,x,X)) fi;
         fi;
         inter:=dev(op(1,fct),a,p,n,x,X);
         if not has(a,infinity) then
            traperror(expdev(",X,n));
         else
            traperror(expdev(",X,n,1));
         fi;
         if "<>'lasterror' then
            if "[nops(")]<>0 and nops(")<=p and n<p+3 then
               deve:=dev(fct,a,p,n+1,x,X)
            else deve:="
            fi;
         elif n<p+10 then
            # sometimes trying a little further may help
            RETURN(dev(fct,a,p,p+5,x,X))
         else ERROR('I cannot develop',fct,'around',op(1,a))
         fi;
      else ERROR('Invalid expression',fct)
      fi;
   fi;
   deve
end:

save dev,'../dev.m';
quit
```

b) Opérations sur les développements

```
#########################################################################
#
#                                ADDEV
#
#  Cette procedure additionne deux developpements de fonctions.
#
#########################################################################

addev:=proc(u,v,X)
local i,j,somme,mini;
option remember;
   traperror(min(op(map(
      proc(x,X)if type(x,name) and type(x,indexed) and op(0,x)=X then
         op(x) fi end,
      map(op,map(listindets,{op(u)})),X))));
   traperror(min(op(map(
```

```
        proc(x,X)if type(x,name) and type(x,indexed) and op(0,x)=X then
            op(x) fi end,
        map(op,map(listindets,{op(v)})),X))));
    mini:=min(",""");
    if not type("",rational) and not type("""",rational) then
        if not type(u,list) and not type(v,list) then
            if u+v=0 then RETURN([0]) else RETURN([1]) fi fi;
        if u=0 or u=[0] then RETURN(v)
        elif v=[0] or v=0 then RETURN(u)
        elif nops(u)=1 or nops(v)=1 then
            RETURN([1])
        else op(1,u)+op(1,v);
            if comparexpr(",0)='=' then RETURN([0])
            else RETURN([",0])
            fi;
        fi;
    fi;
    if not type("",rational) and type("""",rational) then
            mini:=""""
    elif not type("""",rational) then mini:=""
    fi;
    if nops(u)=1 and type(op(u),list) then
        u;
        while nops(")=1 and type(op("),list) do op(") od;
        RETURN(addev(",v,X))
    fi;
if nops(v)=1 and type(op(v),list) then
v;
while nops(")=1 and type(op("),list) do op(") od;
RETURN(addev(u,",X))
fi;
    i:=1;
    j:=1;
    somme:=[];
    while (i<=nops(u) and j<=nops(v)) do
        rdegree(u[i],X[mini]);
        if "=Float(-10,100) then
            Float(10,100)
        else " fi;
        rdegree(v[j],X[mini]);
        if "=Float(-10,100) then
            Float(10,100)
        else " fi;
        comparexpr("""","");
        if "='>' then
            somme:=[op(somme),v[j]];
            j:=j+1
        elif "='<' then
            somme:=[op(somme),u[i]];
            i:=i+1;
        else
proc(x,y) if x=[0]or x=[1] then x
                    elif y then subsop(nops(x)=NULL,x)
```

```
                    else x fi;
 if nops('')=1 then op('') else '' fi end(
          addev(op(map(proc(x)
                  if not op(2,x) then
    if type(op(1,x),list) then
                          [op(op(1,x)),0] else [op(1,x),0] fi
                  else
                     if type(op(1,x),list) then
       op(1,x) else [op(1,x)] fi
                  fi
 end,[[subs(X[mini]=1,u[i]),evalb(i=nops(u))],
 [subs(X[mini]=1,v[j]),evalb(j=nops(v))]])),X),
          evalb(i<nops(u) and j<nops(v)));
          if '<>0 then
                somme:=[op(somme),"*X[mini]^""""]
          elif i=nops(u) and j=nops(v) then
             if u[i]=0 or u[i]=[0] then
                somme:=[op(somme),0]
             else
                somme:=[op(somme),u[i]]
             fi
          fi;
          i:=i+1;
          j:=j+1;
       fi
    od;
    if (i>nops(u) and u[nops(u)]=0) then
       for i from j+1 to nops(v) do somme:=[op(somme),v[i]] od;
    elif (j>nops(v) and v[nops(v)]=0) then
       for j from i+1 to nops(u) do somme:=[op(somme),u[j]] od;
    fi;
    RETURN(somme)
end:

save addev,'../addev.m';
quit
##########################################################################
#
#                              PRDEV
#
#  Cette procedure retourne le produit des developpements des
# fonctions passes en argument.
#
##########################################################################

prdev:=proc(U,V,X)
local maxi,reste,degu,degv,deg,i,j,fini,l,inter,produit,degy,liste,mini,mini1,mini2,u,v;
option remember;
    u:=U;v:=V;
    if not type(u,list) or not type(v,list) then
       RETURN(u*v)
    fi;
    mini1:=traperror(min(op(map(
```

```
proc(x,X)if type(x,name) and type(x,indexed) and op(0,x)=X then
            op(x) fi end,map(op,map(listindets,{op(u)})),X))));
  mini2:=traperror(min(op(map(
proc(x,X)if type(x,name) and type(x,indexed) and op(0,x)=X then
            op(x) fi end,map(op,map(listindets,{op(v)})),X))));
  mini:=proc(x,y) if type(x,rational) and type(y,rational) then min(x,y)
            elif type(x,rational) then x
            elif type(y,rational) then y
            else false
            fi end(",");
  if not type(mini,rational) then
     if u=[0] or v=[0] then RETURN([0])
     elif nops(u)=1 or nops(v)=1 then RETURN([1])
     else RETURN([op(1,u)*op(1,v),0])
     fi;
  elif type("",rational) and "<"" then
     if v[nops(v)]=0 then
        v:=[subsop(nops(v)=0,v),0]
     else
v:=[v]
     fi
  elif type("""",rational) and "<""" then
     if u[nops(u)]=0 then
        u:=[subsop(nops(u)=NULL,u),0];
     else
u:=[u]
     fi
  fi;
  fini:=false;
  degu:=[];
  for i to nops(u) while u[i]<>0 do
     degu:=[op(degu),rdegree(u[i],X[mini])] od;
  degv:=[];
  for i to nops(v) while v[i]<>0 do
     degv:=[op(degv),rdegree(v[i],X[mini])] od;
  if nops(degu)=0 or nops(degv)=0 then
     produit:=[0];
     RETURN(produit);
  fi;
  if nops(degu)=1 and nops(degv)=1 then
#    coeff(op(1,u),X[mini],op(degu));
#    coeff(op(1,v),X[mini],op(degv));
     subs(X[mini]=1,op(1,u));
     subs(X[mini]=1,op(1,v));
     X[mini]^(op(degu)+op(degv));
     if not type("",list) and not type("""",list) then
        if nops(u)=2 and nops(v)=2 then
           RETURN(["""*""*",0])
        else RETURN(["])
        fi
     else
        if type("""",list) then
           if """[nops("""")]=0 then """
```

```
            else [proc(x) x;while whattype(")=list do op(") od;" end("""),0] fi
        else ["""",0] fi;
        if type("""",list) then
if """"[nops("""")]=0 then """"
            else [proc(x) x;while whattype(")=list do op(") od;" end("""),0] fi
        else ["""",0] fi;
        proc(x)subsop(nops(x)=NULL,x);if nops(")=1 then op(") else " fi end(
            prdev("","",X))*"""";
        if nops(u)=2 and nops(v)=2 then
            RETURN(["",0])
        else RETURN(["])
        fi;
    fi;
fi;
deg:=[[1,1,degu[1]+degv[1]]];
# On gere la liste des couples de positions donnant un degre.
produit:=[];
while nops(deg)>0 do
    j:=deg[1][3];#j est le min(degre en X[mini]) courant
    degy:=[Float(10,100)];# la liste des positions des min(degre en X[mini])
    i:=1;# la position du min des degres en X[mini]
    for l from 2 to nops(deg) do # calcul de degy et de i
        deg[l][3];
        if "<j then
            j:="";
            i:=l
        elif "=j then
            if degy[1]=j then degy:=[op(degy),l]
            else degy:=[j,i,l]
            fi
        fi
    od;
    if degy[1]<>j or (i=1 and nops(degy)=1) then
        # il y a un seul minimum et il est en i
        deg[i];
#       if ("[1]<nops(u) or degu=[0]) and
#           ("[2]<nops(v) or degv=[0]) then
        if "[1]<nops(u) and "[2]<nops(v) then
            produit:=[op(produit),
                    proc(x) if x=[0]then[0]else subsop(nops(x)=NULL,x) fi;
if "=[0] then NULL elif nops(")=1 then op(") else " fi end
                (prdev(op(map(proc(x)
                        if type(x,list) then
                            x;while whattype(")=list do op(") od;[",0]
                        else [x,0] fi
                        end,[u["[1]],v["[2]]])),X))];
            deg:=subsop(i=NULL,deg);
            if """"[1]<nops(degu) and """"[2]<nops(degv) then
                [["""[1]+1,""""[2],degu[""""[1]+1]+degv[""""[2]]],
                [""""[1],""""[2]+1,degu[""""[1]]+degv[""""[2]+1]]];
                deg:=[op(deg),op({"[1],"[2]} minus {"[1],"[2]} intersect {op(deg)})]
            elif """"[1]<nops(degu) then
                deg:=[op(deg),[""""[1]+1,""""[2],degu[""""[1]+1]+degv[""""[2]]]]
```

```
            elif """[2]<nops(degv) then
                deg:=[op(deg),["""[1],"""[2]+1,degu["""[1]]+degv["""[2]+1]]]
            else RETURN([op(produit),0])
            fi;
         else RETURN([op(produit),
proc(x) if x=[0]then[0]else subsop(nops(x)=NULL,x) fi;
if "=[0] then NULL elif nops(")=1 then op(") else " fi end
(prdev(op(map(proc(x)
if type(x,list) then
                                x;while whattype(")=list do op(") od;[",0]
                            else [x,0] fi
                            end,[u["[1]],v["[2]]])),X))])
         fi;
      else # il y a plusieurs min
         i:=[degy[2]];
         deg[degy[2]];
         j:=proc(x) if x=[0] then [0] else subsop(nops(x)=NULL,x) fi end
              (prdev(op(map(proc(x)
if type(x,list) then
x;while whattype(")=list do op(") od;[",0]
                            else [x,0] fi
end,
                    [subs(X[mini]=1,u["[1]]),subs(X[mini]=1,v["[2]])])),X));
         if ""[1]=nops(u) or ""[2]=nops(v) then fini:=true fi;
         for l from 3 to nops(degy) do
            deg[degy[l]];
            j:=proc(x) if x=[0] then [0] else subsop(nops(x)=NULL,x) fi;
                  if nops(")=1 then op(") else " fi end
                  (addev(proc(x)if type(x,list)then[op(x),0]else[x,0]fi end(
                              j),
                        [op(prdev(op(map(proc(x)
                              if type(x,list) then
                                 x;while whattype(")=list do op(") od;[",0]
                              else [x,0] fi
                              end,[subs(X[mini]=1,u["[1]]),
                                    subs(X[mini]=1,v["[2]])])),X))],X));
            i:=[op(i),degy[l]];# la liste de ceux qui seront a supprimer
            if """[1]=nops(u) or """[2]=nops(v) then fini:=true fi
         od;
         if not fini then
            if j<>[0] and j<>0 then
               produit:=[op(produit),j*X[mini]^degy[1]];
            fi;
            inter:=copy(deg);
      for l from nops(i) by -1 to 1 do
               op(l,i);
               inter["];
               deg:=subsop(""=NULL,deg);
               if ""[1]<nops(degu) and ""[2]<nops(degv) then
                  [""[1]+1,""[2],degu[""[1]+1]+degv[""[2]]];
                  ["""[1],"""[2]+1,degu["""[1]]+degv["""[2]+1]];
                  deg:=[op(deg),op({","} minus {","} intersect {op(deg)})]
                  elif ""[1]<nops(degu) then
```

```
                        [""[1]+1,""[2],degu[""[1]+1]+degv[""[2]]];
                        if not member(",{op(deg)}) then
                            deg:=[op(deg),"] fi
                    elif ""[2]<nops(degv) then
                        [""[1],""[2]+1,degu[""[1]]+degv[""[2]+1]];
                        if not member(",{op(deg)}) then
                            deg:=[op(deg),"] fi
                    else
                        RETURN([op(produit),0])
                    fi;
                od;
            else
                if not type(j,list) then j:=[j] fi;
                if j=[0] then
                    j:=[X[mini]];
                    for l from 2 to nops(degy) do
                        deg[degy[l]];
                        j:=mindev(j,proc(x)subsop(nops(x)=NULL,x)end(
                            prdev(op(map(proc(x)
 if type(x,list) then
 x;while whattype(")=list do op(") od;[",0]
 else [x,0] fi
 end,
                                    [subs(X[mini]=1,u["[1]]),
                                     subs(X[mini]=1,v["[2]])])),X)),X);
                    od;
                    produit:=[op(produit),proc(x)if nops(x)=1 then op(x) else x fi
                            end(j)*X[mini]^degy[1]];
                else
                    if member(nops(u),map(proc(x)op(1,x)end,
                                {deg[op('j',i)]$'j'=1..nops(i)})) then
                        j:=nettoie(j,reduit(subs(X[mini]=1,u[nops(u)]),X),X) fi;
                    if member(nops(v),map(proc(x)op(2,x)end,
                                {deg[op('j',i)]$'j'=1..nops(i)})) then
                        j:=nettoie(j,reduit(subs(X[mini]=1,v[nops(v)]),X),X) fi;
                    if nops(j)=1 then j:=op(j) fi;
                    produit:=[op(produit),j*X[mini]^degy[1]]
                fi;
                RETURN(produit)
            fi
        fi
    od;
    evalc(simplify(produit))
end:


nettoie:=proc(globdev,lim,X)
local mini,deg,dev;
    dev:=globdev;
    mini:=traperror(min(op(map(
        proc(x,X)if type(x,name) and type(x,indexed) and op(0,x)=X then
            op(x) fi end,map(op,map(listindets,{op(dev)})),X))));
    if not type(",rational) then
```

```
        if subs(0=NULL,dev)=[] then RETURN(dev)
        else RETURN(1)
        fi;
    fi;
    deg:=rdegree(lim,X[mini]);
    while rdegree(dev[nops(dev)],X[mini])>deg do
        dev:=subsop(nops(dev)=NULL,dev) od;
    subsop(nops(dev)=X[mini]^rdegree(dev[nops(dev)],X[mini])*
            nettoie([subs(X[mini]=1,dev[nops(dev)])],subs(X[mini]=1,lim),X),dev)
end:

# L'arete droite
reduit:=proc(tree,X)
    if not type(tree,list) and not(type(tree,'*')) then tree
    elif type(tree,'*') then map(reduit,tree,X)
    else reduit(tree[nops(tree)],X)
    fi
end:

mindev:=proc(u,v,X)
    traperror(min(op(map(
 proc(x,X)if type(x,name) and type(x,indexed) and op(0,x)=X then
              op(x) fi end,map(op,map(listindets,{op(u)})),X))));
    traperror(min(op(map(
 proc(x,X)if type(x,name) and type(x,indexed) and op(0,x)=X then
              op(x) fi end,map(op,map(listindets,{op(v)})),X))));
    if not type("",rational) then
        if not type(",rational) then 1 else u fi
    elif not type(",rational) then v
    else
        if ""<" then v
        elif "<"" then u
        else
            rdegree(u[1],X["]);
            rdegree(v[1],X[""]);
            if ">"" then u
            elif "<"" then v
            else [X[""""]^"*proc(x) if nops(x)=1 then op(x) else x fi end (
                    mindev(subs(X[""""]=1,u[1]),subs(X[""""]=1,v[1]),X))]
            fi
        fi
    fi
end:

save prdev,mindev,reduit,nettoie,'../prdev.m';
quit
##########################################################################
#
#                              POWDEV
#
##########################################################################
powdev:=proc(u,p,n,X)
local deve3,deve4,expr,i,inter,j,v,x,mini,premier,norm;
```

```
    mini:=traperror(min(op(map(proc(x,X)
        if type(x,name) and type(x,indexed) and op(0,x)=X then op(x) fi end,
        map(op,map(listindets,{op(u)})),X))));
    if not type(mini,rational) then
        if nops(u)=1 then RETURN([op(u)^p])
        elif listindets({op(1,u)}) minus {constants}={} then
            RETURN([evalc(op(1,u)^p),0])
        else RETURN([op(1,u)^p,0])
        fi;
    fi;
    # First term
    premier:=proc(x,y) if y=1 then x
                       elif x[nops(x)]=0 then
                         if nops(x)=2 then [x[1]*y,0] else
                             [subsop(nops(x)=NULL,x)*y,0] fi
                       elif nops(x)=1 then [x[1]*y]
                       else [x*y]
                       fi end(
                proc(x,exact,p)
                       if type(x,list) and nops(x)=1 then op(x) else x fi;
                       if not exact then
                          if nops('')>p then ['x'[i]$i=1..p]
                          else '' fi
                       else '' fi end(
                   powdev(proc(x)if type (x,list) then
                              [op(x),0] else [x,0] fi end(
                              subs(X[mini]=1,op(1,u))),
                          p,n,X),
                   evalb(not(type(u,list)) or u[nops(u)]=0),
                   max(2,n)),
                X[mini]^(p*rdegree(op(1,u),X[mini])));
    # the others
    if op(2,u)=0 then RETURN(premier)
    else
        norm:=[op(prdev(powdev(proc(x)if type (x,list) then
         [op(x),0] else [x,0] fi end(
                                        op(1,u)),
                                    -1,n,X),subsop(1=NULL,u),X))];
        expr:=convert(taylor((1+x)^p,x=0,n+2),polynom);
        j:=degree(expr,x);
        deve4:=[1,0];
        deve3:=[1,0];
        for i to j while(deve4[nops(deve4)]=0 or nops(deve4)<=n) do
            deve3:=prdev(deve3,norm,X);
            while nops(deve3)>n+1 do deve3:=subsop(nops(deve3)=NULL,deve3) od;
            if i=j and nops(deve3)>1 and not(type(p,integer) and p>=0)then
                deve3:=subsop(nops(deve3)=NULL,deve3) fi;
            inter:=coeff(expr,x,i);
            multbyreal(deve3,inter);
            deve4:=addev(deve4,'',X)
        od;
    fi;
RETURN(prdev(premier,deve4,X));
```

```
end:

save powdev,'../powdev.m';
quit
#############################################################
#
#                        LNDEV
#
#############################################################
lndev:=proc()
local deve3,lvar,deve4,expr,i,interj,liste,norm,premier,prod,x,nbe,u,X,n;
u:=args[1];X:=args[2];n:=args[3];
   lvar:=map(proc(x,X)if type(x,name) and type(x,indexed) and op(0,x)=X then
      op(x) fi end, map(op,map(listindets,{op(u)})),X);
   nbe:=nops(");
   if nbe=0 then
      comparexpr(u[1],0);
      if "='<' then ERROR('ln of a negative number')
      elif "='=' then ERROR('ln(0)')
      elif u=[1,0] then RETURN([0])
      else RETURN([ln(u[1]),0])
      fi;
   elif nbe=1 and nops(u)=2 and op(2,u)=0 then
      rdegree(u[1],X[op(lvar)]);
      # Here is hidden X[n+1]=f(X[n]) :
      if op(lvar)>0 and nargs=3 then
         if not type(op(lvar),integer) then
            ERROR('Invalid function')
         else
            RETURN(addev(lndev(subs(X[op(lvar)]=1,u),X,n),
                 prdev([lncoeff(op(lvar))*"*X[op(lvar)+1]^(-1),0],
                       [-1,X[1],0],X),X))
         fi
      else
         RETURN(addev(lndev(subs(X[op(lvar)]=1,u),args['i']$'i'=2..nargs),
                           [-"*X[op(lvar)+1]^(-1),0],X))
      fi
   else
    # first term
    liste:=[1,0];
    prod:=1;
      if type(op(1,u),list) or type(op(1,u),'*') then
       for i to nops(op(1,u)) do
          if type(op(i,op(1,u)),list) then liste:=op(i,op(1,u))
          else prod:=prod*op(i,op(1,u))
         fi;
       od
      else prod:=op(1,u) fi;
      [0];
      if type(prod,'*') then
         for i to nops(prod) do
            addev(",lndev([op(i,prod),0],args['i']$'i'=2..nargs),X)
         od;
```

```
        else lndev([prod,0],args['i']$'i'=2..nargs)
        fi;
    premier:=addev(lndev(liste,args['i']$'i'=2..nargs),",X);
    norm:=[op(prdev('powdev'([op(1,u),0],-1,n,X),subsop(1=NULL,u),X))];
    expr:=convert(taylor(ln(1+x),x=0,n+2),polynom);
 j:=degree(expr,x);
    deve3:=[1,0];
    deve4:=[0];
    for i to j do
        deve3:=prdev(deve3,norm,X);
        if i=j and nops(deve3)>1 then
            deve3:=subsop(nops(deve3)=NULL,deve3) fi;
        inter:=coeff(expr,x,i);
        multbyreal(deve3,inter);
        deve4:=addev(deve4,",X)
    od;
 addev(premier,",X);
 fi;
end:

lncoeff:=proc(n)
option remember;
if n=1 then RETURN(1) fi;
if n=2 then RETURN(1/2) fi;
if n=3 then RETURN(5/12) fi;
if n=4 then RETURN(47/120) fi;
if n=5 then RETURN(12917/33840) fi;
if n=6 then RETURN(329458703/874222560) fi;
ERROR('If you need more than 6 nested log, make me a mail for some more coefficients');
end:

save lndev,lncoeff,'../lndev.m';
quit

##########################################################################
#
#                          EXPDEV
#
##########################################################################
expdev:=proc()
local i,v,u,U,X,n;
U:=args[1];X:=args[2];n:=args[3];
   u:=[flat(U)];
   if u[nops(u)]<>0 then
      traperror(min(op(map(proc(x,X)
          if type(x,name) and type(x,indexed) and op(0,x)=X then op(x) fi end,
             map(op,map(listindets,{u[nops(u)]})),X))));
      if "=lasterror then [1]
      else rdegree(u[nops(u)],X["]);
         if ">0 then [1,u[nops(u)]]
         else ERROR('not a convergent development')
         fi;
      fi;
```

```
    else [1,0] fi;
    v:=";
    if nops(u)>2 then
       for i to nops(u)-1 do
          prdev(",#proc(x) if x[nops(x)]=0 then subsop(nops(x)=NULL,x) else x fi;end
                   (expdev(proc(x) if type(x,list) then
                   [op(x),0] else [x,0] fi end(op(i,u)),
                      args['i']$'i'=2..nargs)),X)
       od;
    elif type(u[1],list) then
       for i to nops(u[1]) do
          prdev(",#proc(x) if x[nops(x)]=0 then subsop(nops(x)=NULL,x) else x fi;end
                   (expdev(proc(x) if type(x,list) then
                   [op(x),0] else [x,0] fi end(op(i,u[1])),
                      args['i']$'i'=2..nargs)),X)
       od;
    else
       traperror(min(op(map(proc(x,X)
          if type(x,name) and type(x,indexed) and op(0,x)=X then op(x) fi end,
             map(op,map(listindets,{u[1]})),X))));
       if "=lasterror then [exp(op(1,u)),0]
       else rdegree(u[1],X["]);
          if ">0 then
             [1,0];
             for i to n do
                addev(",map(multbyreal,powdev([u[1],0],i,n,X),1/i!),X)
             od;
             subsop(nops(")=NULL,");
          elif "=-1 and not has(X,coeff(u[1],X[""],-1)) and type("",integer)then
             if "">1 and nargs=3 then
                [X[""-1]^(coeff(u[1],X[""],-1)*(-lncoeff(""-1)))*
                   (X[1]/X[""])^'k'$'k'=0..n]
             else [X[""-1]^(-coeff(u[1],X[""],-1)),0]
             fi;
          else [indexify(exp(u[1]),X),0]
          fi;
       fi;
    prdev(v,",X);
    fi;
end:

save expdev,'../expdev.m';
quit
#####################################################################
#
#                          TRANSLATEOF1
#
#  This is used only for developments in X[1] and X[2].
#
#####################################################################
translateof1:=proc(expr,prof,X)
local i;
   if not has(expr,X) then
```

```
    if type(expr,list) then expr elif expr=0 then [0] else [expr,0] fi
  elif type(expr,'*') and hastype(expr,list) then
    prdev(translateof1(op(1,expr),prof,X),
          translateof1(op(2,expr),prof,X),X);
    for i from 3 to nops(expr) do
      prdev('',translateof1(op(i,expr),prof,X),X)
    od
  elif type(expr,list) then
    [0];
    for i to nops(expr) -1 do
      addev('',translateof1(proc(x)
        if type(x,list) and x[nops(x)]<>0 then [op(x),0] else x fi end(
        expr[i]),prof,X),X)
    od;
    addev('',[expr[i]],X)
  else
    prdev([subs(X[1]=1,X[2]=1,expr),0],
          prdev(powdev([X[1]^(-1),-1,0],-rdegree(expr,X[1]),prof,X),
                powdev(addev([X[2]^(-1),0],
                             lndev([1,-X[1],0],X,prof),X),
                       -rdegree(expr,X[2]),prof,X),X),X)
  fi
end:

save translateof1,'../translateof1.m';
quit
##################################################################
#
#                        LISTINDETS
#
##################################################################
listindets:=proc(expr)
  traperror(indets(expr));
  if ''='lasterror' then
    convert(map(op,map(listindets,convert(expr,list))),set)
  else ''
  fi;
end:

save listindets,'../listindets.m';
quit
dividev:=proc(dev1,dev2,p)
local sing1,sing2,res1,res2,i;
  sing1:=op(1,dev1);
  res1:=op(2,dev1);
  if has(res1[nops(res1)],infinity) then res1:=subsop(nops(res1)=0,res1) fi;
  sing2:=op(1,dev2);
  res2:=op(2,dev2);
  if has(res2[nops(res2)],infinity) then res2:=subsop(nops(res2)=0,res2) fi;
  if nops(sing1)=1 and nops(sing2)=1 then
    if op(1,sing1[1])=op(1,sing2[1]) then
      if hastype(sing1,float) then
        Singularity:=sing1[1];
```

```
            RETURN([sing1,subs(singul=singularity[1],
                proc(x) subsop(nops(x)=subs(singul=1,x[nops(x)]),x)end(
                [flat(proc(x) if sizeof(x,50)<>false then simplify(x) else x fi
                end(
                    prdev(subs(singularity[1]=singul,res1),
                    powdev(subs(singularity[1]=singul,res2),-1,p,X),X))])])]);
        else
            RETURN([sing1,[flat(proc(x) if sizeof(x,50)<>false then
                simplify(x) else x fi end(
                prdev(subs(singularity[1]=1,res1),
                powdev(subs(singularity[1]=1,res2),-1,p,X),X))]]);
        fi;
    elif not hastype([sing1,sing2],float) then
        RETURN([[op(sing1),op(sing2)],
            [flat(proc(x) if sizeof(x,50)<>false then
            simplify(x) else x fi end(prdev(res1,
            powdev(subs(singularity[1]=singularity[2],res2),-1,p,X),X))]])
    elif not hastype(sing1,float) then
        Singularity:=sing2[1];
        RETURN([sing2,proc(x)
            subs(singul=singularity[1],
                subsop(nops(x)=subs(singul=1,x[nops(x)]),x)) end(
                [flat(proc(x) if sizeof(x,50)<>false then simplify(x) else x fi
                end(
                subs(singularity[1]=sing1[1][1],prdev(res1,
                powdev(subs(singularity[1]=singul,res2),-1,p,X),X))))])])
    elif not hastype(sing2,float) then
        Singularity:=sing1[1];
        RETURN([sing1,proc(x)
            subs(singul=singularity[1],
                subsop(nops(x)=subs(singul=1,x[nops(x)]),x)) end(
            [flat(proc(x)if sizeof(x,50)<>false then simplify(x) else x fi end(
            subs(singularity[1]=sing2[1][1],prdev(subs(singularity[1]=singul,
                res1),powdev(res2,-1,p,X),X))))])])
    else# a refaire avec des outils plus puissants
        Singularity:=sing2[1];
        RETURN([[op(sing2),op(sing1)],subs(singul=singularity[2],
            [flat(proc(x) if sizeof(x,50)<>false then simplify(x) else x fi
            end(prdev(res1,
            powdev(subs(singularity[1]=singul,res2),-1,p,X),X))])])
    fi
elif not hastype([sing1,sing2],float) then
    if map(proc(x)op(1,x)end,{op(sing1)})=
        map(proc(x)op(1,x)end,{op(sing2)}) then
        RETURN([sing1,[flat(proc(x) if sizeof(x,50)<>false then
            simplify(x) else x fi end
            (prdev(res1,powdev(res2,-1,p,X),X))]])
    else
        for i to nops(sing2) do
            res2:=subs(singularity[i]=singularity[i+nops(sing1)],res2)
        od;
        RETURN([[op(sing1),op(sing2)],
            [flat(proc(x)if sizeof(x,50)<>false then simplify(x) else x fi end(
```

```
                prdev(res1,powdev(res2,-1,p,X),X)))]])
      fi
   else # a revoir aussi
    if map(proc(x)op(1,x)end,{op(sing1)})=
          map(proc(x)op(1,x)end,{op(sing2)}) then
 RETURN([sing1,[flat(proc(x) if sizeof(x,50)<>false then
 simplify(x) else x fi end
 (prdev(res1,powdev(res2,-1,p,X),X)))]])
 else
 for i from nops(sing2) by -1 to 1 do
 res2:=subs(singularity[i]=singularity[i+nops(sing1)],res2)
 od;
 RETURN([[op(sing1),op(sing2)],
 [flat(proc(x)if sizeof(x,50)<>false then simplify(x) else x fi end(
 prdev(res1,powdev(res2,-1,p,X),X)))]])
 fi
   fi
end:

save dividev,'../dividev.m';

quit

multbyreal:=proc(tree,r)
if type(tree,list) then
   map(multbyreal,tree,r)
elif type(tree,'*') then
 if type(op(1,tree),list) then
       multbyreal(op(1,tree),r)*op(2,tree)
 elif type(op(2,tree),list) then
 multbyreal(op(2,tree),r)*op(1,tree)
   else r*tree
 fi
else tree*r
fi;
end:

save multbyreal,'../multbyreal.m';
quit
```

6. Procédures de comparaison

```
#####################################################################
#
#                          COMPAREXPR
#
#  De meme que pour comparemodule, il s'agit ici de comparer
#  deux expressions formelles.
#
#####################################################################
comparexpr:=proc(Expr1,Expr2)
local a,b,res,t,test,i,X,expr1,expr2;
expr1:=Expr1;expr2:=Expr2;
traperror(testeq(expr1-expr2));
if "=true then RETURN('=') fi;
```

```
if has(expr1,I) then expr1:=evalc(expr1) fi;
if has(expr2,I) then expr2:=evalc(expr2) fi;
if has({expr1,expr2},infinity) then
   if expr1=infinity then RETURN('>')
   elif expr1=-infinity then RETURN('<')
   elif expr2=infinity then RETURN('<')
   elif expr2=-infinity then RETURN('>')
   fi
fi;
if not has(expr1,singul) and not has(expr2,singul) then
   evalf(expr1-expr2);
   if type(",numeric) then
      if abs(")>Float(10,-Digits+3) then
         if ">0 then RETURN('>') else RETURN('<') fi fi;
      Digits:=Digits+8;
      evalf(expr1-expr2);
      Digits:=Digits-8;
      if abs("")<Float(10,-Digits-3) then
         if expr2<>0 then
            RETURN(comparexpr(simplify(expr1-expr2),0))
         elif not type(expr1,float) and expr1<>0 then
            if type(expr1,'*') then
               '>';
               for i to nops(expr1) do
                  comparexpr(op(i,expr1),0);
                  if "='=' then RETURN('=')
                  elif "='<>' or ""='<>' then '<>'
                  elif "='<' then
                     if ""='<' then '>'
                     else '<'
                     fi;
                  else if ""='<' then '<'
                     else '>'
                     fi;
                  fi;
               od;
               RETURN(");
            elif type(expr1,'^') then RETURN(comparexpr(op(1,expr1),0))
            elif type(expr1,fraction) then
                  if op(1,expr1)>0 then '>' else '<' fi;
                  RETURN(");
            else simplify(expr1);
               if "<>expr1 then RETURN(comparexpr(",0))
               else true;
                  while (") do
                     lprint('A mon avis',expr1 ,'=' ,expr2);
                     print(' mais j'ai besoin d'aide ');
                     readstat(' (''='ou''<''ou''>'') :');
                     not member(",{'=','<','>','<>'});
                     if not " then RETURN("") fi;
                  od;
               fi;
         fi;
```

```
        else RETURN('=')
        fi;
    elif ""<0 then RETURN('<')
    else RETURN('>')
    fi;
else
    a:={op(indets(expr1) minus {gamma,Pi,E})};
    if nops(")<>0 then
        a:=map(proc(x) if type(x,'name')then x fi end,")
    fi;
    b:={op(indets(expr2) minus {gamma,Pi,E})};
    if nops(")<>0 then
        b:=map(proc(x) if type(x,'name')then x fi end,")
    fi;
    if nops(a union b)=0 then
        if has(a,I) or has(b,I) then
            if comparexpr(coeff(expr1,I,1),coeff(expr2,I,1))='=' and
                comparexpr(coeff(expr1,I,0),coeff(expr2,I,0))='=' then
                RETURN('=');
            fi;
        fi;
        RETURN('<>');
    elif nops(a)<=1 and nops(b)<=1 then
        0.;
        if not member(n,{op(a)} union {op(b)}) then
            subs(op(a)=t,expr1)-subs(op(b)=t,expr2);
        elif {n}={op(a)} union {op(b)} then
            normal(subs(n=1/t,expr1-expr2));
        else ERROR('On essaye de comparer des pommes et des tomates')
        fi;
        if evalf(")=0. then
            res:='='
        elif type(",'^') and type(op(2,"),integer) then
            comparexpr(op(1,"),0);
            if member(",{'=','>'}) then RETURN(")
            elif op(2,"") mod 2 = 0 then
                RETURN('>');
            else RETURN(")
            fi;
        elif type(",'*') then
            res:=[op(")];
            '>';
            for i to nops(res) do
                comparexpr(op(i,res),0);
                if "='=' then RETURN('=')
                elif "='<>' or ""='<>' then '<>'
                elif "='<' then
                    if ""='<' then '>'
                    else '<'
                    fi;
                else if ""='<' then '<'
                    else '>'
                    fi;
```

```
            fi;
        od;
        RETURN(");
    elif type(",function) and op(0,")=abs then
        comparexpr(op("),0);
        if "<>'=' then RETURN('>') else RETURN('=') fi;
    else
        subs(abs=proc(x) x end,");
        if type(",polynom) then
            op(1,dev(",[0,t,1],1,1,t,X));
            evalf(normal("/X[1]**rdegree(",X[1])));
        else
            evalf(convert(subs(O=proc()0 end,taylor(",t,25)),polynom));
            if type(",'+') then op(1,") fi;
            normal("/t**rdegree(",t));
        fi;
        if not type(",numeric) then RETURN('<>') fi;
        if ">0 then res:='>'
        elif "<0 then res:='<'
        elif simplify(expr1-expr2)=0 then RETURN('=')
         else test:=true;
            while (test) do
                lprint('A mon avis',expr1 ,'=' ,expr2);
                print(' mais j'ai besoin d'aide ');
                res:=readstat(' (''='ou''<'ou''>'') :');
                if member(res,{'=','<','>'}) then test:=false fi;
            od;
        fi;
    fi;
elif nops(a)=0 or nops(b)=0 then res:='<>'
elif expr2<>0 then comparexpr(expr1-expr2,0)
elif type(expr1,'*') then
    res:=[op(")];
    '>';
    for i to nops(res) do
        comparexpr(op(i,res),0);
        if "='=' then RETURN('=')
        elif "='<>' or ""='<>' then '<>'
        elif "='<' then
            if ""='<' then '>'
            else '<'
            fi;
        else if ""='<' then '<'
            else '>'
            fi;
        fi;
    od;
    RETURN(");
else test:=true;
    while (test) do
        lprint('Je ne sais pas comparer',expr1 ,'et' ,expr2);
        res:=readstat('. Votre avis? (''='ou''<'ou''>'') :');
        if member(res,{'=','<','>'}) then test:=false fi od
```

```
        fi
    fi;
    res
else
    Digits:=Digits+3;
    evalf(subs(singul=Singularity[1],expr1)-subs(singul=Singularity[1],expr2));
    Digits:=Digits-3;
    if not has("",I) then
        if abs("")>Float(10,-Digits+2) then
            if "">0 then RETURN('>') else RETURN('<') fi
        fi;
    else
        if abs("")>Float(10,-Digits+2) then RETURN('<>') fi
    fi;
    subs(singul=Singularity[nops(Singularity)],expr1-expr2);
    if "=Singularity[2] or "=-Singularity[2] then RETURN('=') fi;
    if has(",Singularity[2]) then
        RETURN(comparexpr(subs(Singularity[nops(Singularity)]=singul,
                        subs(Singularity[2]=0,")),0)) fi;
    if has(Singularity[2],") then # assume we have simplified the problem
        Singularity[2]:=";
        RETURN('=')
    else
        admit(expr1-expr2,'comparexpr');
        RETURN('=')
    fi
fi
end:

admit:=proc(expr,procedure)
    if not assigned(admitcount) then
        admitcount:=0;
    else admitcount:=admitcount+1
    fi;
    print('Warning :  assumption made, see file equivassume');
    writeto(cat('equivassume',admitcount));
    if procedure='comparexpr' then
        if not has(expr,singul) then
            lprint('comparexpr(',expr,',0):=''=';')
        else
            lprint('comparexpr(',expr,',0):=''=';# where singul is a root of ',Singularity[2])
        fi
    fi;
    writeto(terminal);
    end:


save comparexpr,admit,'../comparexpr.m';
quit
##########################################################################
#
#                          COMPAREMODULE
#
```

```
# Cette procedure compare le module de deux expressions
# formelles. Elle retourne >, < ou = et demande eventuellement
# de l'aide en cas de probleme.
#
#####################################################################
comparemodule:=proc(expr1,expr2,justdiffneeded)
if not hastype(expr1,float) and not hastype(expr2,float) and testeq(expr1-expr2)=true then
   RETURN('=') fi;
if not has(expr1,singul) and not has(expr2,singul) then
   if expr2<>0 then
      comparexpr(abs(expr1),abs(expr2))
   else
      evalc(expr1);
      comparexpr(coeff(",I,1),0);
      if "='=' then comparexpr(abs(coeff("",I,0)),0) else '<>' fi;
   fi;
   if "='<>' and expr2<>0 then
      if nargs<3 or not justdiffneeded then
         true;
         while(") do
         print('Je ne sais pas comparer les modules de',expr1,'et',expr2,'.');
         readstat('Votre avis (''<''ou''>''ou''='') :');
         not member(",{'=','<','>','<>'});
         if not " then RETURN("") fi;
         od;
      else RETURN('<>')
      fi
   elif "='<>' and expr2=0 then RETURN('>')
   else " fi;
else
   evalf(abs(subs(singul=Singularity[1],expr1))-abs(subs(singul=Singularity[1],expr2)));
   if abs(")>Float(10,-Digits+3) then
         if ">0 then RETURN('>') else RETURN('<') fi
   else
      admit(abs(expr1)-abs(expr2),comparexpr);
      RETURN('=')
   fi
fi
end:


save comparemodule,'../comparemodule.m';
done
comparedev:=proc(Dev1,Dev2,X)
local i,dev1,dev2;
if not type(Dev1,list) then dev1:=[Dev1,0] else dev1:=Dev1 fi;
if not type(Dev2,list) then dev2:=[Dev2,0] else dev2:=Dev2 fi;
if nops(dev1)=2 and dev1[2]=0 and nops(dev2)=2 and dev2[2]=0 then
   if (not has(dev1,X) or type(dev1[1],name) or
      (type(dev1[1],'^') and not has(op(2,dev1[1]),X) and
         type(op(1,dev1[1]),name))) and
      (not has(dev2,X) or type(dev2[1],name) or
         (type(dev2[1],'^') and not has(op(2,dev2[1]),X) and
```

```
            type(op(1,dev2[1]),name))) then
        if not has(dev1[1],X) then
            if not has(dev2[1],X) then RETURN(comparexpr(dev1[1],dev2[1]))
            elif type(dev2[1],name) or op(2,dev2[1])>0 then RETURN('>')
            else RETURN('<')
            fi
        elif not has(dev2[1],X) then
            if type(dev1[1],name) or op(2,dev1[1])>0 then RETURN('<')
            else RETURN('>')
            fi
        else
            if type(dev1[1],name) then
                [op(dev1[1]),1]
            else [op(op(1,dev1[1])),op(2,dev1[1])]
            fi;
            if type(dev2[1],name) then
                [op(dev2[1]),1]
            else [op(op(1,dev2[1])),op(2,dev2[1])]
            fi;
            if ""[2]*"[2]<0 then
                if ""[2]<0 then RETURN('>') else RETURN('<') fi
            elif "[2]*"[1]<"[2]*""[1] then RETURN('>')
            elif "[2]*"[1]>"[2]*""[1] then RETURN('<')
            elif "[2]<""[2] then RETURN('>')
            elif "[2]>""[2] then RETURN('<')
            else RETURN('=')
            fi
        fi
    elif (type(dev1[1],'^') and has(op(2,dev1[1]),X)) or
            (type(dev1[1],'^') and has(op(2,dev1[1]),X)) then
        RETURN(comparedev(lndev(dev1,X,1),lndev(dev2,X,1),X))
    elif (type(dev1[1],'*') and has(op(1,dev1[1]),X) and
            has([op('i',dev1[1])$'i'=1..nops(dev1[1])],X)) or
            (type(dev1[1],'*') and has(op(1,dev1[1]),X) and
            has([op('i',dev1[1])$'i'=1..nops(dev1[1])],X)) then
        RETURN(comparedev(lndev(dev1,X,1),lndev(dev2,X,1),X))
    fi
else
    '=';
    for i while ("='=' and i<=nops(dev1) and i<=nops(dev2)) do
        traperror(min(op(map(
            proc(x,X)if type(x,name) and type(x,indexed) and op(0,x)=X then
                op(x) fi end,
            map(op,map('listindets',{op(dev1[i])})),X))));
        traperror(min(op(map(
            proc(x,X)if type(x,name) and type(x,indexed) and op(0,x)=X then
                op(x) fi end,
            map(op,map('listindets',{op(dev2[i])})),X))));
        if not type("",integer) and not type('',integer) then
            comparexpr(dev1[i],dev2[i])
        elif not type('',integer) then
            if dev2[i]<>0 then
                if rdegree(dev1[i],X[""])<0 then '>' else '<' fi
```

```
            else '>'
            fi
        elif not type("",integer) then
            if dev1[i]<>0 then
                if rdegree(dev2[i],X["])<0 then '<' else '>' fi
            else '<'
            fi
        elif ""<" then
            if rdegree(dev1[i],X[""])<0 then '>' else '<' fi
        elif "">" then
            if rdegree(dev2[i],X["])<0 then '<' else '>' fi
        else
            rdegree(dev1[i],X["])-rdegree(dev2[i],X["]);
            if signum(")=-1 then '>'
            elif "<>0 then '<'
            else comparedev(subs(X[""]=1,dev1[i]),subs(X[""]=1,dev2[i]),X)
            fi
        fi
    od;
    RETURN(")
fi
end:

save comparedev,'../comparedev.m':
quit
##########################################################################
#
#                    COMPAREMONOME
#
#  Cette procedure fait des comparaisons sur l'echelle n^r*logn^s
#
##########################################################################

comparemonome:=proc(m1,m2)
    rdegree(m1,n);
    rdegree(m2,n);
    if """>" then '>'
    elif ""<" then '<'
    else rdegree(m1,logn);
        rdegree(m2,logn);
        if """>" then '>'
        elif ""<" then '<'
        else '='
        fi
    fi
end:

save comparemonome,'../comparemonome.m';
quit
```

7. Divers utilitaires

```
flat:=proc(dev)
if (not type(dev,list)) and (not type(dev,'*')) then
    dev
```

```
elif type(dev,'*') then
   if type(op(1,dev),list) then op(map(proc(u,v) u*v end,[flat(op(1,dev))],op(2,dev)))
   elif type(op(2,dev),list) then op(map(proc(u,v) u*v end,[flat(op(2,dev))],op(1,dev)))
   else dev
   fi
else op(map(flat,dev))
fi
end:

save flat,'../flat.m';
quit
##########################################################################
#
#                              NCOEFTAYL
#
#  Cette procedure retourne le coefficient de degre n du developpement
# en serie de Taylor. Il devrait y avoir moins de probleme qu'avec
# la procedure standard Maple.
#
##########################################################################

ncoeftayl:=proc(expr,var,ordre)
   taylor(expr,var,ordre+2);
   convert(",polynom);
   coeff(",var,ordre);
end:

save ncoeftayl,'../ncoeftayl.m';
quit
##########################################################################
#
#                          RDEGREE
#
#  Calcule le degre d'une expression par rapport a une variable meme
# quand Maple ne reconnait pas cette expression comme un polynome.
#
##########################################################################

rdegree:=proc(expression,var)
local i,expr;
      if whattype(expression)=series then
         expr:=convert(expression,polynom)
      else expr:=expression
      fi;
      has(expr,var);
      if not " then
         if expr=0 then Float(-10,100)
         else 0 fi;
      elif type(expr,polynom) then
         degree(expr,var)
      elif type(expr,'^') then
         if op(1,expr)=var then op(2,expr)
         else op(2,expr)*rdegree(op(1,expr),var)
```

```
        fi
    elif type(expr,'*') then
        0;
        for i to nops(expr) do
            "+rdegree(op(i,expr),var)
        od;
    elif type(expr,'+') then
        Float(-10,100);
        for i to nops(expr) do
            rdegree(op(i,expr),var);
            if ">"" then " else "" fi;
        od
    else 0;# et on essaie de ne pas y envoyer n'importe quoi
    fi;
end:

save rdegree,'../rdegree.m';
quit
##########################################################################
#
#                              TRIE
#
#  Cette procedure, a laquelle on passe deux listes de
# points du plan complexe homogenes en module, doit en
# deduire la liste reunion en fonction des modules en
# question. Ceci est fait relativement a la fonction
# car deux singularites au meme point peuvent s'annuler.
#
##########################################################################

trie:=proc(u1,u2,fct,var,minsing,p)
local res,u,v,i,j,x,X;
    x:=var;u:=u1;v:=u2;
    if has(u1,Float(10,100)) then RETURN(u2) fi;
    if has(u2,Float(10,100)) then RETURN(u1) fi;
    if u1=[] then RETURN(u2)
    elif u2=[] then RETURN(u1) fi;
    if nops(u)=1 then op(u) else u[1] fi;
    if nops(v)=1 then op(v) else v[1] fi;
    if comparemodule(""[1],minsing)<>'>' then RETURN(v)
    elif comparemodule("[1],minsing)<>'>' then RETURN(u)
    fi;
    comparemodule(op(1,""),op(1,"));
    if "='<' then RETURN(u)
    elif "='>' then RETURN(v)
    elif "='=' then
        res:=v;
        for i to nops(u) do
            if member(op(1,op(i,u)),{op(map(proc(t)op(1,t)end,res))},'j') then
#               if type(u[i][1],float) then
#                   Singularity:=[op(u[i]),x];
#                   dev(eval(subs(L=proc(x)ln(1/(1-x))end,Q=proc(x)1/(1-x)end,fct)),
#                           proc(y,x)subsop(2=eval(subs(L=proc(x)ln(1/(1-x))end,
```

```
#                                                   Q=proc(x)1/(1-x)end,y[2])),y) end
#                   (subsop(1=singul,u[i]),x),
#               p,p,x,X);
#           else
#               dev(eval(subs(L=proc(x)ln(1/(1-x))end,Q=proc(x)1/(1-x)end,fct)),
#                       proc(y,x)subsop(2=eval(subs(L=proc(x)ln(1/(1-x))end,
#                                                   Q=proc(x)1/(1-x)end,y[2])),y) end
#                   (u[i],x),
#               p,p,x,X);
#           fi;
#           traperror(max(op(map(
#                   proc(x,X)if type(x,name) and type(x,indexed) and op(0,x)=X then
#                       op(x) fi end,
#                   map(op,map('listindets',{op(")})),X))));
#           if type(",integer) and "=1 then
#               map(rdegree,"",X[1]);
#               if "[1]>=0 and
#                   not has(false,map(proc(x) hastype(x,integer) end,")) then
#                   res:=subsop(j=NULL,res);
#               fi;
#           fi;
        else res:=[op(res),op(i,u)] fi
    od;
    res
  else ERROR('Trop de variables')
  fi;
  RETURN(");
end:


save trie,'../trie.m';
quit
```

Bibliographie

Bibliographie

E. H. BAREISS [1960]. "Resultant Procedure and the Mechanization of the Graeffe Process", *JACM* **7**, 1960, 346-386.

E. H. BAREISS [1967]. "The Numerical Solution of Polynomial Equations and the Resultant Procedures", in *Mathematical Methods for Digital Computers* J. Wiley, 1967, 185-214.

J. BEAUQUIER, B. BÉRARD, L. THIMONNIER [1986]. "On a concurrency measure", *Rapport de recherche LRI* **306**, 1986.

E. A. BENDER [1974]. "Asymptotic Methods in Enumeration", *SIAM Rev.* **16-4**, Oct1974, 485-515.

W. S. BROWN [1969]. "Rational Exponential Expressions and a Conjecture Concerning π and e", *Amer. Math. Monthly* **76**, 1969, 28-34.

N. G. DE BRUIJN [1981]. *Asymptotic Methods in Analysis.* Dover, New York, 1981.

B. F. CAVINESS [1970]. "On Canonical Forms and Simplification", *JACM* **17-2**, Apr70, 385-396.

B.W. CHAR, K.O. GEDDES, G.H. GONNET AND S.M. WATT [1985]. "MAPLE : Reference Manual", University of Waterloo, 1985.

C. CHOPPY, S. KAPLAN AND M. SORIA [1987] "Algorithmic complexity of term rewriting systems", Proc. of the 2nd R.T.A. Conf., LNCS **256**, 1987.

C. CHOPPY, S. KAPLAN AND M. SORIA [1988] " Complexity Analysis of term rewriting systems", to appear in *Theoretical Computer Science*

L. COMTET [1970]. *Analyse Combinatoire* PUF, 1970.

M. COSTE AND M. F. ROY [1988]. "Thom's lemma, the coding of real algebraic numbers and the topology of semi-algebraic sets", *J. of Symbolic Computation*, Jan 1988.

M. G. DARBOUX [1878]. "Mémoire sur l'approximation des fonctions de très grands nombres, et sur une classe étendue de développements en série", *Journal de Math. Pures et Appl.*, Fev1878, 5-66 et 377-416.

N. G. DE BRUIJN [1981]. *Asymptotic Methods in Analysis* Dover Publications NY, 1981.

E. DURAND [1971]. *Solutions numériques des équations algébriques* Masson et cie, Tomel, 1971.

D. DUVAL [1987]. "Diverses questions relatives au calcul formel avec des nombres algébriques" *Thèse à l'Université de Grenoble*, 1987.

A. ERDÉLYI AND M. WYMAN [1963]. "The Asymptotic Evaluation of Certain Integrals", *Arch. Rational Mech. Anal.* **14**, 1963, 217-260.

PH. FLAJOLET [1985]. "Mathematical Methods in the Analysis of Algorithms and Data Structures," INRIA, Research Report 400, 1985. To appear in *A Graduate Course in Computer Science*, Computer Science Press, 1987.

P. FLAJOLET [1985]. "Elements of a General Theory of Combinatorial Structures", in *Proc. FCT Conf., Lecture Notes in Comp. Sc*, Springer Verlag, 1985, 112-127.

P. FLAJOLET AND A. M. ODLYZKO [1987]. "Singularity Analysis of Generating Functions", preprint, 1987.

P. FLAJOLET, B. SALVY AND P. ZIMMERMANN [1988]. "$\Lambda\Upsilon\Omega$: An Assistant Algorithms Analyser", in *Proc. AAECC'6, Rome, July 1988, Lecture Notes in Computer Science*, 1988, to appear.

G. GONNET [1977]. "On the Structure of Zero-Finders", *BIT* **17-2**, 1977, 170-183.

G. GONNET [1984]. "Determining Equivalence of Expressions in Random Polynomial Time", *Proc. of the 16th ACM Symposium on the Theory of Computing*, 1984, 334-341.

G.GONNET [1986]. "New Results for Determination of Equivalence of Expressions", *Proc. of the 1976 ACM Symposium on Symbolic and Algebraic Computation*, 1976, 127-131.

E. GROSSWALD [1966]. "Generalization of a Formula of Hayman and its Application to the Study of Riemann's Zeta Function", *Illinois J. Math.* **10**, 1966, 9-23.

G. H. HARDY [1910]. "Orders of Infinity", *Cambridge Tracts in Mathematics* **12**,1910.

G. H. HARDY [1911]. "Properties of logarithmico-exponential functions", *Proc. of the London Math. Soc.* **10-2**, 1911, 54-90.

B. HARRIS AND L. SCHOENFELD [1968]. "Asymptotic Expansions for the Coefficients of Analytic Functions", *Illinois J. Math.* **12**, 1968, 264-277.

L. HÄUSLER [1930]. "Über das asymptotische Verhalten der Taylorkoeffizienten einer gewissen Funktionklasse", *Math. Zeitschrift* **32**, 1930, 115-146.

W. K. HAYMAN [1956]. "A Generalization of Stirling's Formula", *J. Reine und Angewandte Mathematik* **196**, 1956, 67-95.

P. HENRICI [1977]. *Applied and Computational Complex Analysis.* Three Volumes. Wiley, New York, 1977.

R. JÜNGEN [1931]. "Sur les séries de Taylor n'ayant que des singularités algébrico-logarithmiques sur leur cercle de convergence", *Comment. Math. Helv.* **3**, 1931, 266-306.

D. E. KNUTH [1981]. *The Art of Computer Programming.* Volume 2: *Semi-Numerical Algorithms.* Addison-Wesley, Reading, MA, second edition 1981.

J. MACINTYRE AND R. WILSON [1954]. "Operational Methods and the Coefficients of Certain Power Series", *Math. Annalen.* **127**, 1954, 243-250.

M. MARDEN [1966]. "The Geometry of the Zeros of a Polynomial in a Complex Variable", *AMS Surveys* **3**, 1966, 2nd ed.

J. MOSES [1971]. "Algebraic Simplification: a Guide for the Perplexed", *Communications of the ACM* **14-8**, Aug71, 527-537.

A. M. ODLYZKO AND L. B. RICHMOND [1985]. "Asymptotic Expansions for the Coefficients of Analytic Generating Functions", *Aequationes Mathematicae* **28**, 1985, 50-63.

D. RICHARDSON [1968]. "Some Undecidable Problems Involving Elementary Functions of a Real Variable", *Journal of Symbolic Logic* **4-33**, 1968, 514-520.

R. H. RISCH[1975]. "Algebraic Properties of the Elementary Functions of Analysis", *American Journal of Mathematics* **4-101**, 1975, 743-759.

M. F. ROY AND A. SZPIRGLAS[1988]. "Complexity of the computation on real algebraic numbers",

J. M. STEYAERT [1984]. "Structure et complexité des algorithmes" *Thèse à l'Université de Paris 7*, 1984.

J. VITTER AND PH. FLAJOLET [1987]. "Average Case Analysis of Algorithms and Data Structures", in *A Handbook of Theoretical Computer Science*, North Holland Pub. Comp., 1987, to appear.

E. M. WRIGHT [1949]. "On the Coefficients of Power Series having Exponential Singularities", *J.London Math. Soc.* **24**, 1949, 304-309.

M. WYMAN [1959]. "The Asymptotic Behavior of the Laurent Coefficients", *Canad. J. Math.* **11**, 1959, 534-555.

P. ZIMMERMANN [1988]. "Alas : un système d'analyse algébrique", *Rapport de stage de DEA, Université de Paris VII*, 1988.

Imprimé en France
par
l'Institut National de Recherche en Informatique et en Automatique