

PRATIQUE
DU
BASIC
1000D

Jean-Jacques LABARTHE

Laboratoire Aimé Cotton
Université de Paris XI

1991

Table des matières

1. Introduction 1

2. Les nombres 5

Nombres flottants	6
Nombres entiers	11
Nombres premiers et factorisation	15
Nombres modulaires	18
Nombres rationnels	19
Instabilités	25
Nombres complexes	27
Fonctions aléatoires	29

3. Calcul formel 31

Polynômes et fractions rationnelles	32
Factorisation	34
Décorticage	36
Substituer et Homogénéiser	37
Division et PGCD	38
Fonctions symétriques	39
Décomposition en éléments simples	40
Développements limités	41
Calcul matriciel	43
Extensions algébriques	47
Résolution d'équations	49
Dérivation	52
Intégration	53
Sommation en termes finis	55
Calculs modulaires	56
Géométrie plane	56
Approximation polynomiale	57

4. Quelques problèmes arithmétiques 59

L'indicateur d'Euler	60
Congruences linéaires	65
Le symbole de Legendre	68
Congruences non linéaires	70
Factorisation des entiers de Gauss	75

5. Factorisation des nombres 81

La méthode rho de Pollard	82
Factorisation de Fermat	86
La méthode de Legendre	87
Développement en fraction continue	87
Bases de facteurs	91
La méthode de Brillhart et Morrison	91
Méthode des courbes elliptiques	100
Stratégie pour factoriser les nombres	105
Mesure des performances	113

6. Accélération de la convergence des séries 115

La méthode rho	116
La transformation de Levin (symbolique)	119
La transformation de Levin (numérique)	125
Comparaison des méthodes	128

7. Tracé de courbes 131

Les procédures <code>fplot</code> et <code>axis</code>	132
La procédure <code>qplot</code>	137
Fonctions d'onde de l'hydrogène	145

8. Fonction Gamma 159

Fonction Gamma et factorielle	160
Graphe de la fonction Gamma	161
Fonction Gamma complexe	162
Trajectoires complexes	163
Description du programme <code>gamma</code>	166
La procédure <code>stirling</code>	169
La procédure <code>bernoulli</code>	170
Les polynômes et nombres de Bernoulli	170
Les polynômes et nombres d'Euler	173
Permutations d'André	175
La lemniscate de Bernoulli	178
Fonction Gamma en précision 1000	180
Fonctions digamma et polygamma	182
Constante d'Euler	191
Fonction Gamma incomplète	193

9. Etats quasistationnaires 199

Figure du potentiel	200
La fonction $W(x)$	202
Etats quasistationnaires	204
Etude de $W(x)$ au voisinage de $x = 1$	208
Etats quasistationnaires en $x = 1$	209
Largeur des états quasistationnaires en $x = 1$	211

10. Etude d'un moteur 213

 Ecriture formelle des équations 215

 Résolution numérique 219

11. Algèbres non commutatives 225

 Programme de décodage 229

 Autre algèbre 234

Appendice 237

 Solution des exercices 238

 Bibliographie 276

 Notations 277

 Index 277

1

Présentation



Introduction

Le Basic 1000d est un logiciel de calcul formel qui vise une large diffusion dans le public, en particulier chez les étudiants et lycéens. Dans ce but, le logiciel, qui fonctionne confortablement sur Atari 520 ST, utilise un langage facile à apprendre, qui n'est rien d'autre qu'un Basic. Le Basic 1000d n'est pas une simple initiation au calcul formel, mais un logiciel très performant : Les fonctions les plus importantes (traitement des fractions rationnelles, factorisation, élimination) font partie du noyau. Le noyau, écrit en langage machine, donne une rapidité fulgurante qui est du même ordre de grandeur que les systèmes de calcul formel sur mini-ordinateurs. Le logiciel peut ensuite absorber des bibliothèques, écrites en Basic 1000d, pour traiter des problèmes plus spécialisés (intégration, résolution d'équations, fonction trigonométriques, sorties graphiques, etc.).

Voici quelques mots sur l'historique du Basic 1000d. L'ancêtre du Basic 1000d, le Basic Algébrique (Basalg), a été dès 1986, le premier logiciel de calcul formel disponible sur Atari ST. Il permettait déjà la manipulation de formules mathématiques. Les progrès du Basic 1000d sur Basalg sont considérables. Tout d'abord du point de vue de la convivialité et de la facilité de programmation, le Basic 1000d n'a rien à envier aux meilleurs logiciels, alors que Basalg exigeait des acrobaties constantes, tant pour l'édition que pour l'écriture des programmes. La partie standard du langage du Basic 1000d est maintenant extrêmement complète. Elle comprend des fonctions avancées comme le tri (suivant une relation d'ordre programmable), et le traitement des ensembles (ce qui rapproche le Basic du langage Lisp). Les procédures et fonctions, avec variables locales et récursivité, permettent une programmation aisée. La réalisation d'applications particulières est très facilitée en Basic 1000d par la possibilité d'intégration d'une bibliothèque et d'un fichier d'aide en ligne.

Les progrès du point de vue mathématique sont encore plus extraordinaires. La fonction la plus utile en calcul formel, la factorisation complète des polynômes en coefficients rationnels, est maintenant disponible pour toutes les expressions. Un point nouveau du Basic 1000d, qui n'existait pas vraiment en Basalg, est la possibilité d'effectuer des calculs numériques très précis. On dispose en effet d'un type de nombres approchés, les nombres flottants, qui peuvent avoir jusqu'à 1230 chiffres significatifs. Basalg était seulement capable de manipuler des nombres exacts (même lorsque ces nombres exacts ne représentaient que des valeurs approchées comme π), qui devenaient rapidement des nombres gigantesques dans les opérations en chaîne. Les calculs numériques en Basalg étaient ainsi particulièrement éprouvants (temps très longs et souvent arrêt par dépassement mémoire). Enfin, la vitesse du Basic 1000d est en général de l'ordre de 5 à 20 fois celle de Basalg.

Le but de ce livre est de montrer par des exemples de programmes comment utiliser le Basic 1000d pour résoudre des problèmes mathématiques. Nous voulons donner une vue d'ensemble sur la partie du Basic consacrée au calcul numérique et au calcul formel. Vous pouvez commencer l'étude de ce livre après avoir lu les deux premiers chapitres du manuel de référence. De nombreux exemples utilisent les sous-programmes de la bibliothèque MATH, que nous vous recommandons donc de charger pour les tester vous-même. D'autres exemples *sont* une description détaillée de programmes de la bibliothèque MATH. Pour presque tous les exercices proposés, une solution est donnée.

Le Basic 1000d manipule de façon très simple et efficace les objets suivants :

- les nombres flottants, réels et complexes, avec une précision variable;
- les nombres entiers et fractionnaires;
- les polynômes et fractions rationnelles symboliques.

Les chapitres 2 et 3 présentent les possibilités du Basic 1000d, en ce qui concerne le traitement de ces objets, en privilégiant l'aspect mathématique, sans trop insister sur la programmation. Le Basic 1000d, à la différence des gros systèmes de calcul formel, ne manipule donc pas directement les expressions mathématiques les plus générales. Dans les chapitres suivants, nous décrivons de nombreuses applications, dans l'optique de permettre une assimilation solide du Basic. Dans le chapitre 4, nous considérons quelques exercices élémentaires de la théorie des nombres comme la résolution de congruences et la factorisation des entiers de Gauss. Le chapitre 5 traite de la factorisation des grands nombres en facteurs premiers. Le chapitre 6 étudie des méthodes de calculs de développements en série qui convergent très lentement. Dans le chapitre 7, nous décrivons des procédures de tracés de courbes. Le chapitre 8 fournit des exemples d'utilisations et une description du fonctionnement de la fonction **gamma**, et de fonctions apparentées. Le chapitre 9 considère un problème de mécanique quantique. Le chapitre 10 traite la modélisation d'un moteur asynchrone. Le chapitre 11 donne des programmes qui permettent d'étendre le Basic 1000d à des calculs dans une algèbre non-commutative.

Les exercices suivants peuvent tous être facilement résolus avec le Basic 1000d. Leur programmation, très simple, peut être effectuée en quelques minutes. Le temps d'exécution, pour la totalité des exercices, est de quelques secondes. Ce que nous vous proposons ici est de résoudre les exercices à la main, en notant soigneusement le résultat et le temps de résolution pour chacun. Lorsque vous connaîtrez le Basic 1000d, vous reprendrez cette résolution, pour vérifier les résultats et comparer les temps passés sur chaque exercice.

Exercice 1.1.

Simplifier la somme de nombres rationnels

$$\frac{70}{125} + \frac{763}{3325}. \quad (1.1)$$

Exercice 1.2.

Simplifier l'expression

$$\frac{a}{x-a} + \frac{x(a-1) + a^2 + a}{x^2 - a^2}. \quad (1.2)$$

Exercice 1.3. Racines

Déterminer les racines de l'équation

$$x^2 - 4x + 13 = 0. \quad (1.3)$$

Exercice 1.4. Equations

Résoudre le système d'équations :

$$\begin{cases} x + y = 1 \\ 98x - 14y = 2. \end{cases} \quad (1.4)$$

Exercice 1.5. Autres équationsRésoudre le système d'équations en x , y , z et t :

$$\begin{cases} x + y - z - t = a + 2 \\ x^3 + y^3 + z^3 + t^3 = a^3 + 8 \\ x^2 + y^2 + z^2 + t^2 = a^2 + 6 \\ x + y + z + t = a + 2. \end{cases} \quad (1.5)$$

Exercice 1.6. DérivationDériver par rapport à x l'expression

$$\frac{x}{\cos(x^2 + ax)}. \quad (1.6)$$

Exercice 1.7. Intégrale

Calculer l'intégrale :

$$\int \frac{1}{(x^2 + 3ax + 2a^2)^3} dx. \quad (1.7)$$

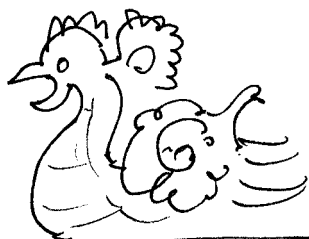
Exercice 1.8. Développement limitéCalculer le développement limité de $\exp(\sin x)$ au voisinage de $x = 0$, à l'ordre 10.

2

Les nombres



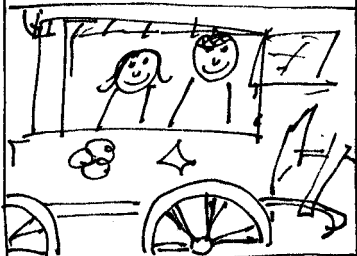
D	L	M	M	J	V	S
					1	2
3	4	5	6	7	8	9
10	11	12	13	14	15	16
17	18	19	20	21	22	23
24 31	25	26	27	28	29	30



D	L	M	M	J	V	S
		1	2	3	4	5
6	7	8	9	10	11	12
13	14	15	16	17	18	19
20	21	22	23	24	25	26
27	28	29	30	31		



D	L	M	M	J	V	S
	1	2	3	4	5	6
7	8	9	10	11	12	13
14	15	16	17	18	19	20
21	22	23	24	25	26	27
28	29					



D	L	M	M	J	V	S
					1	2
3	4	5	6	7	8	9
10	11	12	13	14	15	16
17	18	19	20	21	22	23
24	25	26	27	28	29	30

Nombres flottants

Les nombres flottants sont des nombres traités de façon approchée. A la différence des nombres exacts, qui seront décrits plus bas, ils ne sont manipulés qu’avec une précision limitée, mais cette précision peut atteindre 1230 chiffres. Par défaut, la précision est de 10 chiffres.

Le tilde

Le tilde indique que le nombre est flottant. La première ligne traite des nombres exacts, la deuxième des nombres flottants.

```
print 0.3333333333+0.6666666666
print 0.3333333333~+0.6666666666~
```

Sortie (95 ms)

```
9999999999/100000000000
1.0000000000~
```

Précision

On affiche π avec la plus grande précision permise par le Basic.

```
precision 1230
print pi
```

Sortie (4200 ms)

```
0.314159265358979323846264338327950288419716939937510582097494459230
781640628620899862803482534211706798214808651328230664709384460955058
223172535940812848111745028410270193852110555964462294895493038196442
881097566593344612847564823378678316527120190914564856692346034861045
432664821339360726024914127372458700660631558817488152092096282925409
171536436789259036001133053054882046652138414695194151160943305727036
575959195309218611738193261179310511854807446237996274956735188575272
489122793818301194912983367336244065664308602139494639522473719070217
986094370277053921717629317675238467481846766940513200056812714526356
082778577134275778960917363717872146844090122495343014654958537105079
227968925892354201995611212902196086403441815981362977477130996051870
721134999999837297804995105973173281609631859502445945534690830264252
230825334468503526193118817101000313783875288658753320838142061717766
914730359825349042875546873115956286388235378759375195778185778053217
122680661300192787661119590921642019893809525720106548586327886593615
338182796823030195203530185296899577362259941389124972177528347913151
557485724245415069595082953311686172785588907509838175463746493931925
506040092770167113900984882401285836160356370766010471018194~ E+1
```

Racine carrée

La précision des calculs flottants est fixée à 1000 chiffres. Le calcul proprement dit de $\sqrt{3}$ prend 1.3 s, et l’affichage du résultat 3 s.

```
precision 1000
```

```
print sqrt(3)
```

Sortie (4280 ms)

```
0.173205080756887729352744634150587236694280525381038062805580697945
193301690880003708114618675724857567562614141540670302996994509499895
247881165551209437364852809323190230558206797482010108467492326501531
234326690332288665067225466892183797122704713166036786158801904998653
737985938946765034750657605075661834812960610094760218719032508314582
952395983299778982450828871446383291734722416398458785539766795806381
835366611084317378089437831610208830552490167002352071114428869599095
636579708716849807289949329648428302078640860398873869753758231731783
139599298300783870287705391336956331210370726401924910676823119928837
564114142201674275210237299427083105989845947598766428889779614783795
839022885485290357603385280806438197234466105968972287286526415382266
469842002119548415527844118128653450703519165001668929441548084607127
714399976292683462957743836189511012714863874697654598245178855097537
901388066496191196222295711055524292372319219773826256163146884203285
3716682938649611917049738836395495938~ E+1
```

Fonctions transcendentes

Les noms usuels `sin`, `sinh`, etc. correspondent à des fonctions numériques flottantes. L’exemple suivant calcule et affiche `sin 1` avec 500 chiffres significatifs.

```
precision 500
```

```
print sin(1)
```

Sortie (37150 ms)

```
0.841470984807896506652502321630298999622563060798371065672751709991
910404391239668948639743543052695854349037907920674293259118920991898
881193410327729212409480791955826766606999907764011978408782732566347
484802870298656157017962455394893572924670127086486281053382030561377
218203868449667761674266239013382753397956764255565477963989764824328
690275696429120630058303651523031278255289853264851398193452135970955
962062172114814441781057601075674136648055008916726605804140078062393
07037187795626128880~
```

L’exemple suivant donne `Arcsin $\frac{1}{3}$` avec 150 chiffres significatifs.

```
precision 150
```

```
print asin(1/3)
```

Sortie (7315 ms)

```
0.339836909454121937096392513391764066388244690332458071431923962489
915888664848411460765792500197612852129763807402294474152393575686806
025683416157261~
```

Temps de calcul typiques en secondes

precision	10	11	100	200	500	1000	1230
$w * w$	0.002	0.002	0.01	0.03	0.1	0.6	0.9
$1/(w + 1)$	0.01	0.01	0.03	0.07	0.3	0.8	1.2
w^{1000}	0.02	0.02	0.14	0.44	2	9	13
\sqrt{w}	0.01	0.01	0.05	0.11	0.4	1.3	2.8
e^w	0.03	0.06	0.89	3.1	18	85	135
$\log w$	0.05	0.09	1.38	4.7	27	124	258
w^w	0.07	0.14	2.21	7.7	46	210	396
$\sin w$	0.04	0.09	1.67	6.0	39	187	362
$\tan w$	0.04	0.18	3.30	12.1	78	374	726
$\operatorname{atn} w$	0.05	0.12	2.95	11.6	78	396	880
$\operatorname{print} w$	0.03	0.03	0.14	0.30	1.0	2.9	4.1

Le nombre réel flottant $w = 17/19 \sim$ a été utilisé. Les fonctions flottantes sont calculées par polynômes en mémoire jusqu’à la précision 10 et par des développements limités et relations fonctionnelles au delà. C’est pourquoi les temps augmentent fortement en précision 11. Les calculs en précision inférieure à 10 ne sont pas recommandés, le gain en temps étant de 30% au mieux.

Formats d’affichage

La commande `format n` indique que les nombres flottants sont affichés avec $|n| - 1$ chiffres après la virgule (ou plutôt le point décimal), si $n \neq 0$. Si $n > 0$, la sortie est sous forme fixe, sans exposant. Si $n < 0$, la sortie est sous forme flottante, avec un exposant (s’il est non nul). Si $n = 0$, on obtient la représentation machine exacte des nombres affichés (un entier divisé par une puissance de 2). Après `precision p`, `format` prend la valeur $-p - 1$ (`format` est une variable d’état qui peut être lue). Les valeurs normalement utilisables de n sont limitées par la précision. Si des valeurs trop grandes de $|n|$ sont employées, les derniers chiffres n’auront pas de signification. Dans l’exemple ci-dessous, $\sqrt[3]{2}$, calculé en précision 10, est affiché d’abord en format fixe avec 5 chiffres, puis dans le format de la représentation machine exacte (on vérifie facilement que le dénominateur est $2^{\operatorname{intlg}(1125899906842624)} = 2^{50}$) et enfin en format exponentiel avec 30 chiffres après la virgule. Le calcul en précision 30 montre que seulement les 14 premiers chiffres après la virgule étaient exacts.

```
format 6
print 2^(1/3)
format 0
print 2^(1/3)
format -31
print 2^(1/3)
```

```
precision 30
print 2^(1/3)
```

Sortie (420 ms)

```
1.25992~
1418544992705693/1125899906842624~
0.125992104989486808364063108456~ E+1
0.125992104989487316476721060728~ E+1
```

Le tilde ~ peut être supprimé ou remis à l'aide des variables d'état `notilde` et `tilde`.

```
notilde
print pi^2
tilde
print pi^2
```

Sortie (70 ms)

```
0.9869604401 E+1
0.9869604401~ E+1
```

En format exponentiel (`format < 0`), la variable d'état `formatl`, qui vaut 0 par défaut, donne le nombre de chiffres devant le point décimal.

```
print exp(1)
formatl 1
print exp(1)
formatl 2
print exp(1)
```

Sortie (170 ms)

```
0.2718281828~ E+1
2.7182818285~
27.1828182846~ E-1
```

Les exposants sont multiples de la variable d'état `formatm`, qui vaut 1 par défaut.

```
formatm 3
for i=1 to 5
  print 10~^i
next
```

Sortie (190 ms)

```
10.0000000000~
0.1000000000~ E+3
1.0000000000~ E+3
10.0000000000~ E+3
0.1000000000~ E+6
```

La commande `print using` permet des possibilités supplémentaires de formatage, par exemple l'insertion d'espaces pour faciliter la lecture.

```
print using "41/333=#.### ### ###~_._._.",41/333~
```

Sortie (45 ms)

```
41/333=0.123 123 123~...
```

Intégration numérique

La fonction `romberg(a, b, f, ε)` de la bibliothèque MATH calcule numériquement l'intégrale définie

$$I = \int_a^b f(x) dx \quad (2.1)$$

par la méthode de Romberg, à la précision ϵ . L'exemple suivant calcule 20 chiffres de la constante d'Euler par :

$$\gamma = \int_0^1 \frac{1 - e^{-t} - e^{-1/t}}{t} dt \quad (2.2)$$

La fonction externe `f1` est définie par un sous-programme en Basic. La variable `tf` est locale à ce sous-programme. La fonction `f1` prend la valeur assignée à la variable `value`. La fonction `exp1(x)` qui calcule $e^x - 1$ à la précision courante, même quand x est voisin de 1, remplace avantageusement `exp(x)-1`.

```

prec=20
precision prec
print romberg(0,1,f1,10^-prec)
stop
f1:function(tf)
  tf=float(tf)
  if tf=0
    value=1~
  else
    value=-(exp1(-tf)+exp(-1/tf))/tf
  endif
return

```

Sortie (827 s)

0.57721566490153286061~

Calcul de π

La méthode suivante a été utilisée par Tamura et Kanada (1983) pour calculer $2^{34} = 16777216$ décimales de π . Nous sommes limités à 1230 chiffres significatifs par la précision maximum des calculs flottants du Basic 1000d. Après chaque itération, a et b sont remplacés par les moyennes arithmétiques et géométriques de leurs valeurs précédentes. D'après un résultat de Gauss, le nombre $(a + b)^2/4c$ converge vers π de façon très rapide, le nombre de chiffres significatifs doublant à chaque itération. En 9 itérations on obtient plus de 1230 chiffres. Les 16 millions de chiffres de π s'obtiennent avec seulement 14 itérations supplémentaires.

```

precision 1230
a=1~
b=1/sqr(2)
c=1/4~

```

```

for n=0,8
  y=a
  a=(a+b)/2
  b=sqr(b*y)
  c=c-2^n*(a-y)^2
next
q=(a+b)^2/(4*c)
d=q-pi
precision 10
print "q-pi=";d
print timer

```

Sortie (60 s)

```
q-pi= 0.1149081664~ E-1241
```

Exercice 2.1. Arrondi de l'unité

Déterminer le plus petit nombre flottant positif δ tel que la représentation machine de $1 + \delta$ diffère de 1. Pour tout nombre flottant positif $x < \delta$, $1 + x$ et 1 sont identiques pour le Basic. La valeur δ indique le nombre de chiffres de la représentation d'un nombre. L'arrondi de l'unité est de la forme $\delta = 2^{-k}$; comment l'entier k varie-t-il en fonction de la précision ?

Nombres entiers

Les nombres entiers sont traités de façon exacte, sans approximation. Leur taille est limitée, mais la limite correspond à des nombres s'écrivant avec 19000 chiffres environ.

Division

La division de 123456 par 789 donne $123456 = 156 \times 789 + 372$.

```
print divr(123456,789);modr(123456,789)
```

Sortie (25 ms)

```
156 372
```

Plus grand commun diviseur (pgcd)

L'exemple calcule le pgcd de deux nombres d'une centaine de chiffres chacun. Le résultat peut être facilement vérifié puisque le pgcd de $2^p - 1$ et $2^q - 1$ est $2^r - 1$ où r est le pgcd de p et q .

```
print gcdr(2^370-1,2^430-1)
```

Sortie (30 ms)

```
1023
```

Factorielle

Calcul et affichage de la factorielle 100!, puis division par 99!.

```

f=ppwr(100)
print f
print f/ppwr(99)

```

Sortie (620 ms)

```

93326215443944152681699238856266700490715968264381621468592963895217
599993229915608941463976156518286253697920827223758251185210916864000
00000000000000000000
100

```

Comme le plus grand entier possible est $2^{65520} - 1$, la plus grande factorielle calculable exactement est 5909!. Son calcul par :

```

w=ppwr(5909)

```

prend 199 secondes, mais l'impression en base 10 de ce nombre de 19722 chiffres nécessite 245 secondes (c'est principalement le temps de la conversion de binaire en d cimal).

Temps de calcul typiques en millisecondes

<i>p</i>	100	500	1000	5000	10000	19000
$N_p = 10^p + 1$	6	45	135	750	10700	37400
$N_p + N_p$	1	1	2	15	25	40
<code>intsqr</code> (N_p)	30	170	560	15170	59000	148000
$p!$	90	1215	4795	139800		
<code>print</code> N_p	20	190	675	15300	60500	217000
$N_p * N_4$	1	3	6	25	65	115
$N_p * N_5$	1	5	8	40	80	145
$N_p * N_{50}$	5	20	35	160	315	625
$N_p * N_{500}$	35	145	285	1390	2780	5265
$N_p * N_{5000}$	290	1390	2765	13675	27350	
$N_p \setminus N_4$	5	10	10	30	50	100
$N_p \setminus N_5$	5	15	35	150	305	575
$N_p \setminus N_{50}$	10	35	60	310	625	1180
$N_p \setminus N_{500}$	5	5	215	1745	3645	7055
$N_p \setminus N_{5000}$	5	5	5	10	18695	50835

Les temps des op rations de divisions (`divr`, `modr`, `/`) ou du calcul du pgcd (`gcdr`) de deux nombres sont voisins du temps de la multiplication de ces nombres. La fonction `intsqr`(x) calcule la partie enti re de la racine carr e $\lfloor \sqrt{x} \rfloor$.

Coefficients du binôme

La fonction `ppwr`(p, q) calcule $p^{(q)} = \Gamma(p+1)/\Gamma(p-q+1)$, pour q entier. La v_fonction `binome`(p, q) renvoie le coefficient du binôme

$$C_q^p = \binom{p}{q} = \frac{p!}{(p-q)!q!} = \frac{p^{(q)}}{q!} = \frac{p^{(p-q)}}{(p-q)!}. \quad (2.3)$$

C'est un exemple de fonction externe (écrite en Basic). Une fois que la fonction est placée dans la source ou bibliothèque, elle devient disponible, comme les fonctions internes, aussi bien en mode direct que dans les programmes. Les variables `p` et `q` sont ici locales à la fonction. La c_fonction `binome$(p, q)` renvoie une chaîne contenant le nom et la valeur du coefficient du binôme C_q^p . Tant dans le cas des v_fonctions que des c_fonctions, la valeur prise par la fonction est la valeur de la variable locale `value` (qui est de type `var` dans les v_fonctions et de type `char` dans les c_fonctions) au moment du retour par la commande `return`. Dans le calcul de la chaîne assignée à `value`, dans la fonction `binome$`, noter que les expressions p , q et `binome`(p, q) sont automatiquement converties en chaînes (la concaténation est notée par `&`).

```

print binome$(100,50)
for i=0,3
  print binome$(3,i);
next i
stop
binome$:function$(p,q)
  value="  C(" & p & ", " & q & ")=" & binome(p,q)
  return
binome:function(p,q)
  ift integerp(p) and integerp(q) q=min(q,p-q)
  value=ppwr(p,q)/ppwr(q)
  return

```

Sortie (385 ms)

```

C( 100, 50)= 100891344545564193334812497256
C( 3, 0)= 1   C( 3, 1)= 3   C( 3, 2)= 3   C( 3, 3)= 1

```

Base

Le Basic 1000d peut travailler dans une base quelconque entre 2 et 36, spécifiée par la variable d'état `base`. Les chiffres plus grands que 9 sont alors représentés par les lettres A, B, ..., sans distinction entre majuscules et minuscules. En entrée, les nombres sont normalement décodés dans la base courante (il faut rajouter 0 devant un nombre commençant par une lettre), mais ils peuvent être écrits en base 10 (resp 2, 16) indépendamment de la base courante en les faisant précéder du symbole § (resp %, \$). En sortie, l'option D (resp B, H) de la commande `print` force l'affichage en base 10 (resp 2, 16). Nous attirons votre attention sur le piège suivant. Pour revenir en base 10, on ne peut pas utiliser

base 10, qui n'a aucun effet (10 est décodé dans la base courante). On utilisera base §10 par exemple.

Le programme convertit l'écriture du nombre x de la base 20 aux bases 36 et 10 :

$$(\text{GD312D0BI4J3D})_{20} = (\text{INTERESSANT})_{36} = (68229699878119673)_{10}$$

```
base 20
x=0GD312D0BI4J3D
base §36
print x
print/d/x
```

Sortie (60 ms)

```
0INTERESSANT
68229699878119673
```

Un entier de 19724 chiffres

L'exemple suivant calcule et écrit le 94377-ième terme F_{94377} de la suite de Fibonacci $F_1 = 1, F_2 = 1, \dots, F_n = F_{n-1} + F_{n-2}$. L'écriture du nombre, qui nécessiterait 7 pages, n'a pas été reproduite ici. La variable `c$` contient la chaîne de caractères représentant F_{94377} . Après la fin du programme, on peut relire rapidement le nombre par `print/a/c$` ou l'imprimer par `lprint c$`.

La commande `vadd W1,W2` équivaut à $W1=W1+W2$. La commande `exg` échange les valeurs de deux variables. La fonction `timer` a pour valeur le nombre de secondes écoulées depuis le début du programme ou depuis `clear timer`. Noter que ce temps écoulé peut être mesuré en millisecondes par la fonction `mtimer` (mais il n'existe pas de commande `clear mtimer`, pour rappeler que `timer` et `mtimer` mesurent le même temps). En Basic 1000d, la commande `c$=W2` effectue la transformation de la valeur de `W2` en `str$(W2)`. La chaîne `c$` contient alors deux espaces devant le nombre. Pour compter le nombre de chiffres par la fonction `len` (longueur en caractères), on supprime d'abord ces espaces par la fonction `justl$`.

```
W1=0
W2=1
for I=2,94377
  vadd W1,W2
  exg W1,W2
next I
W1=timer
clear timer
c$=W2
print c$
print "Le ";I;"-ième nombre de la suite de Fibonacci s'
écrit avec"&len(justl$(c$))&" chiffres"
print "Son calcul a pris";W1;" secondes"
```

```
print "Son affichage a pris";timer;" secondes"
```

Sortie (2069 s)

```
Le 94378-ième nombre de la suite de Fibonacci s'écrit avec 19724 chiffres
```

```
Son calcul a pris 1819 secondes
```

```
Son affichage a pris 250 secondes
```

Exercice 2.2. Somme des chiffres

Ecrire un programme qui calcule la somme des chiffres d'un nombre entier.

Exercice 2.3. 1989

On dispose les chiffres de 1 à 9 dans l'ordre croissant et on intercale entre eux des signes $+$, $-$ et $*$ pour former une expression comme par exemple $12 + 3 * 4 - 5 * 6 + 789$ qui vaut 783. Déterminer les expressions de cette forme égales à 1989.

Exercice 2.4. Total=100

Avec les chiffres de 1 à 9, dans l'ordre, et en utilisant seulement des additions et soustractions, on peut obtenir 100. Par exemple :

$$123 - 45 - 67 + 89 = 100. \quad (2.4)$$

Ecrire un programme qui calcule toutes les solutions.

Exercice 2.5. cube+cube=carré

Déterminer les nombres i de deux chiffres, tel que si j est le nombre obtenu en inversant les deux chiffres de i , les propriétés suivantes soient vérifiées :

$$i \leq j \quad \text{et} \quad i^3 + j^3 \text{ est un carré.} \quad (2.5)$$

Nombres premiers et factorisation

Nombres de Mersenne

Marin Mersenne (1588-1648) affirma que les nombres $2^p - 1$ étaient premiers pour $p = 1, 2, 3, 5, 7, 13, 17, 19, 31, 67, 127, 257$, et pour aucun autre p inférieur à 257. Les nombres premiers de la forme $M_p = 2^p - 1$ sont appelés nombres de Mersenne. Pour Mersenne, 1 était un nombre premier, mais de nos jours la liste commence par M_2 . L'affirmation de Mersenne comporte plusieurs erreurs, en particulier pour $p = 257$ qui donne un nombre composé, résultat qui ne fut démontré qu'en 1922 par Kraitchik (sans détermination des facteurs). Le calcul suivant *prouve* en 8 secondes que $2^{257} - 1$ n'est pas un nombre premier.

```
print prtst(2^257-1)
```

Sortie (7990 ms)

0

Rep-unit

Les nombres de la forme $R_n = (10^n - 1)/9$ sont appelés rep-units (de l'anglais *repeated unit*, car le nombre s'écrit uniquement avec des 1). On ne connaît que 5 rep-units premiers, ceux qui s'écrivent avec 2, 19, 23, 317 et 1031 chiffres. Remarquer que les nombres de Mersenne $2^n - 1$ sont des rep-units en base 2. Le test de primalité est appliqué au rep-unit de 19 chiffres. Il ne *prouve* pas que le nombre est premier, mais indique qu'il est probablement premier, avec une probabilité d'erreur plus faible que 10^{-40} .

```
print prtst(1111111111111111111)
```

Sortie (5540 ms)

-1

Nombre premier

Le programme suivant détermine le plus petit nombre premier s'écrivant avec 20 chiffres. La fonction `prime(n)` utilise le test probabiliste `prtst` pour calculer le plus petit nombre premier $\geq n$. Comme pour `prtst`, le résultat est obtenu avec une probabilité très faible d'être faux.

```
print prime(10^19)
```

Sortie (6185 ms)

100000000000000000051

Facteurs premiers d'un nombre

La fonction `prfact$(n)` décompose l'entier n en facteurs premiers. Le résultat est renvoyé sous forme de chaîne.

```
print prfact$(1111111111111111111)
```

Sortie (300 ms)

11 * 17 * 73 * 101 * 137 * 5882353

La fonction `prfact$` fonctionne en divisant n par 2, 3, 5, 7, 11, puis par les entiers croissants jusqu'à \sqrt{u} (où u est le facteur non encore décomposé), sans diviser par les multiples de 2, 3, 5, 7 et 11. La bibliothèque MATH comprend des procédures factorisant les entiers suivant des méthodes plus performantes que nous décrirons en détail dans le chapitre 5. Voici quelques exemples. La procédure `pollard` utilise la méthode ρ de Pollard. La factorisation de $2^{67} - 1$ par cette méthode est 80 fois plus rapide que par la méthode des divisions successives de `prfact$`. Nous avons indiqué plus haut que Mersenne pensait que $2^{67} - 1$ était un nombre premier. La factorisation de ce nombre est connue depuis 1903, grâce à Cole, qui passa ses dimanches, pendant trois années, à la rechercher.

```
pollard 2^67-1
```

Sortie (695 s)

147573952589676412927= 193707721 * 761838257287

L'exemple suivant factorise un nombre de 22 chiffres par la méthode de Brillhart et Morrison. Le nombre est factorisé en 6 minutes, alors qu'on peut estimer que `prfact$` mettrait plus d'un mois pour arriver au résultat.

```
brison 5810323356986051704777
```

Sortie (374 s)

```
5810323356986051704777= 644804915323 * 9010978699
```

La méthode de factorisation de Lenstra, basée sur les courbes elliptiques, est également disponible en Basic 1000d. Voici la factorisation d'un nombre de 33 chiffres, $2^{109} + 1$, par cette méthode, qui dans ce cas, est beaucoup plus rapide que par la procédure `brison`.

```
lenstra 2^109+1,10^7
```

Sortie (734 s)

```
649037107316853453566312041152513= 3 * 104124649 *
2077756847362348863128179
```

Facteur non carré d'un entier

On décompose l'entier $x = ys^2$ sous la forme du produit d'un nombre sans facteurs premiers carrés y et d'un carré s^2 . Cette décomposition peut servir, entre autres, à simplifier la racine carrée d'un entier ($\sqrt{x} = s\sqrt{y}$). La fonction `sqr(x)` programmée ci-dessous renvoie y . La fonction `integerp(x)` teste si x est un entier, et `odd(j)` teste si j est impair. La fonction `root(z, 2)` renvoie la racine carrée de z si cette racine est rationnelle, et renvoie 0 sinon. La fonction `introot(x, 3)` renvoie la partie entière de $\sqrt[3]{x}$. Le calcul est exact, alors que le calcul par `int(x^(1/3))` serait approché. La fonction `prfact(x)`, comme `prfact$`, décompose l'entier x en facteurs premiers. Le résultat est codé sous la forme d'un polynôme. Le programme illustre les possibilités suivantes de la fonction `prfact` :

- On peut restreindre `prfact` pour rechercher les facteurs inférieurs à une valeur donnée. Ici la limite est la partie entière de $\sqrt[3]{x}$. Tous les facteurs, sauf peut-être le dernier, sont premiers. On peut aussi limiter `prfact` à la recherche du plus petit facteur premier, mais cette possibilité n'est pas utilisée ici.
- La décomposition obtenue est analysable. Le nombre de facteurs est donné par `polymn(M)` et, posant $e = \text{polym}(M, i)$, le i ème facteur est le nombre `norm(e)` élevé à la puissance `deg(e)`.

```
x=245
y=sqr(x)
print using "#=# *_ #^2",x,y,root(x/y,2)
stop
sqr:function(x)
  ift not integerp(x) erreur_non_entier
  value=1
  ift root(x,2) return
  local var e,M index i,j
  M=prfact(x,introot(x,3))
```

```

i=polymn(M)
e=polym(M,i)
j=deg(e)
if odd(j)
    e=norm(e)
    ift j=1 ift root(e,2) exitif
    value=e
endif
ift i=1 return
for i=1,i-1
    e=polym(M,i)
    ift odd(deg(e)) vmul value,norm(e)
next i
return

```

Sortie (105 ms)

245=5 * 7^2

Exercice 2.6. Premiers jumeaux

Deux nombres premiers p_1 et p_2 qui diffèrent de 2 ($p_2 - p_1 = 2$) sont appelés nombres premiers jumeaux. Exemples : 4 ± 1 , $694513810 \times 2^{2304} \pm 1$ et $1159142985 \times 2^{2304} \pm 1$. On présume qu'il existe une infinité de nombres premiers jumeaux, mais cette conjecture n'a pas été démontrée. Par contre, on sait que la série de Brun :

$$B = \left(\frac{1}{3} + \frac{1}{5}\right) + \left(\frac{1}{5} + \frac{1}{7}\right) + \left(\frac{1}{11} + \frac{1}{13}\right) + \left(\frac{1}{17} + \frac{1}{19}\right) + \cdots, \quad (2.6)$$

où les dénominateurs sont les nombres premiers jumeaux, converge alors que la somme des inverses des nombres premiers diverge.

Calculer la constante de Brun B .

Nombres modulaires

Le Basic 1000d possède des fonctions très performantes effectuant des calculs modulo un nombre premier p . Nous en verrons quelques applications dans le chapitre 4.

Reste modulaire

Les fonctions `modr` et `mods` renvoient le reste positif ou signé.

```

print modr(1989,169)
print mods(1989,169)

```

Sortie (35 ms)

130

-39

Exponentiation

La fonction `mdpwre` donne $721580^{5^{6789}} \pmod{1234577}$.

```
print mdpwre(721580,5^6789,1234577)
```

Sortie (298 s)

2

Système d'équations modulaires

La fonction `chinoiseq` de la bibliothèque MATH permet de résoudre un système de congruences. L'exemple suivant détermine une solution des congruences $15x \equiv 49 \pmod{53}$ et $12x \equiv 32 \pmod{59}$. D'après le théorème des restes chinois, cette solution est unique modulo 53×59 .

```
print chinoiseq(15,49,53,12,32,59)
```

Sortie (180 ms)

1989

Racine carrée modulaire

Gauss montra (*Disquisitiones Arithmeticae*, n° 328) que les solutions de l'équation :

$$x^2 \equiv -286 \pmod{4272943}$$

sont ± 1493445 . Le programme suivant retrouve ce résultat, en calculant une racine carrée de $-286 \pmod{4272943}$ par la fonction `prsqre` de la bibliothèque MATH.

```
print prsqre(-286,4272943)
```

Sortie (345 ms)

1493445

Nombres rationnels

Les nombres rationnels m/n avec m et n entiers sont représentés et traités de façon exacte, sans approximation. Ainsi le calcul suivant est exact.

```
print (4^30000+7/17)-4^30000
```

Sortie (440 ms)

7/17

Dans l'exemple, le nombre $4^{30000} + 7/17$ est d'abord calculé (il occupe 7512 octets en mémoire), puis le nombre 4^{30000} est calculé et retranché au nombre précédent.

30 chiffres de la constante de Neper

La somme $\sum_{i=0}^{28} 1/i!$, calculée exactement, donne une approximation de la constante e . Après `formatx 31`, le résultat est affiché sous forme décimale avec 30 chiffres après la virgule. L'ordre de grandeur de l'erreur sur cette valeur, $1/29!$, est affiché en format exponentiel à deux chiffres significatifs après `formatx -3`.

```
e=1
for i=1 to 28
  e=e+1/ppwr(i)
next i
print e
formatx 31
print e
formatx -3
print 1/ppwr(29)
formatx 0
```

Sortie (1135 ms)

```
828772446866981044847857913441/304888344611713860501504000000
2.718281828459045235360287471353~
0.11~ E-30
```

La fonction `sum` permet de calculer plus facilement la somme précédente.

```
print sum(i=0,28 of 1/ppwr(i))
```

Sortie (905 ms)

```
828772446866981044847857913441/304888344611713860501504000000
```

Somme sur des permutations

Le programme suivant calcule la somme des 24 fractions

$$S = \frac{1}{1234} + \frac{1}{1243} + \cdots + \frac{1}{4321}, \quad (2.7)$$

où les dénominateurs sont les nombres obtenus en permutant les chiffres 1, 2, 3 et 4. La fonction `nextperm(N, P(1), a)` permet d'effectuer la boucle sur les $N!$ permutations $P(1), P(2), \dots, P(N)$ de 1, 2, \dots , N . Le premier appel, avec $a = 0$, met la permutation 1, 2, \dots , N dans le tableau d'index*32 P . Les appels suivants mettent les 23 permutations suivantes, dans l'ordre lexicographique. Après la dernière permutation, la fonction `nextperm` renvoie 0, ce qui permet de terminer la boucle `while ...wend`. Le dénominateur correspondant à une permutation est calculé par la fonction `sum`.

```
N=4
index P(N)
k=nextperm(N,P(1),0)
while k
  S=S+1/sum(i=1,N of 10^(N-i)*P(i))
  k=nextperm(N,P(1))
```



```
wend
print "S=";S
```

Sortie (1040 ms)

```
S= 916852623262960725151557410259374071539911822938023989382581475/88
143836170793809085636874369597591023849058055778354106460866668
```

Convergents de π

La fonction `appr(p , k)` renvoie la fraction rationnelle f la plus simple possible qui soit une approximation à mieux que 2^{-k} de p ($|p - f| < 2^{-k}$). Elle est utilisée pour calculer les convergents de π .

```
precision 20
w=exact(pi)
print "pi=";pi;" est représenté par"&a;numr(w);"/";
justl$(prfact$(denr(w)))
format -5
forv i in (1,3,10,14,22,31)
  w=appr(pi,i)
  print justl$("appr(pi,"&justl$(i,2)&")="&w,30);
  "erreur=";abs(w-pi)
nextv
```

Sortie (875 ms)

```
pi= 0.31415926535897932385~ E+1 est représenté par
124451306656115542615260972311/2^95
appr(pi,1 )= 3          erreur= 0.1416~
appr(pi,3 )= 22/7       erreur= 0.1264~ E-2
appr(pi,10)= 333/106    erreur= 0.8322~ E-4
appr(pi,14)= 355/113    erreur= 0.2668~ E-6
appr(pi,22)= 103993/33102 erreur= 0.5779~ E-9
appr(pi,31)= 104348/33215 erreur= 0.3316~ E-9
```

Un jeu simple

Le programme suivant est une variante du jeu qui fait deviner un nombre à partir des indications *trop grand* et *trop petit*. Il s'agit de découvrir un nombre fractionnaire p/q compris entre 0 et 1 et dont le dénominateur q est inférieur à `qmax`. La boucle `do` externe permet de jouer plusieurs parties. La commande `checker` affiche le but du jeu dans les trois lignes supérieures de l'écran. A la différence d'un texte écrit par `print`, ce texte reste dans le menu et ne subit pas le défilement de l'écran. Les symboles d'édition utilisés dans `checker` sont « (obtenu par [a] C) qui centre le texte, et a (obtenu par [a] L) qui change de ligne. La variable `score` compte le nombre d'essais. On prend pour le dénominateur q un entier aléatoire $\lceil qmax/2 \rceil < q < qmax$, puis pour le numérateur p un entier aléatoire $0 < p < q$. Si p et q ont des diviseurs communs, le rapport $x = p/q$ est automatiquement simplifié lorsqu'on forme leur quotient. Le nombre à découvrir x est donc une fraction réduite avec un dénominateur dans $[2, qmax - 1]$. Le compteur `timer` est mis à zéro au début du jeu par `clear timer`. La

boucle **do** interne est parcourue jusqu'à la découverte du nombre x , et à chaque passage la variable **score** est incrémentée. La boucle **repeat ...until** attend une entrée convenable : la fonction **ratnump** teste si l'entrée y lue par **input** est bien un nombre rationnel exact; si par exemple on tape le nombre flottant 0.23~, **ratnump** prend la valeur 0 et la boucle **repeat** est reprise pour demander une autre entrée. Les entrées : 0.6, 3/5, 9/15 et $(1+2)/(2+3)$ sont admises et définissent la même fraction. Noter que le Basic calcule les expressions entrées par **input**. Dans cet exemple la forme 3/5 n'est pas vue comme un nombre rationnel, mais comme l'opération / effectuée exactement sur les nombres 3 et 5. Cette distinction est importante dans des exemples comme \$123/45 où seul le premier nombre 123 est en base 16, le deuxième 45 étant décodé en base courante.

La comparaison de x et y est effectuée à l'aide d'une structure **if ...else** qui permet de traiter les trois cas $y < x$, $y > x$ et $y = x$. En cas d'égalité, une boîte d'alerte est affichée par **message**, qui donne le score et le temps de jeu **timer** converti en minutes et secondes. Si une nouvelle partie est demandée dans la boîte d'alerte affichée par la fonction **sure?**, la commande **exit** est effectuée, ce qui fait sortir de la boucle **do ...loop** interne. Sinon, il y a arrêt du programme sur la commande **stop**.

```

qmax=12
do
  checker "<<Vous devez découvrir  a<<un nombre fraction
    naire p/qa<<entre 0 et 1 tel que q <"&qmax
  score=0
  q=gint(qmax/2)+random(qmax\2)
  p=1+random(q-1)
  x=p/q
  clear timer
do
  repeat
    print "Quel nombre proposez-vous ?"
    input y
  until ratnump(y)
  score=score+1
  print "Votre nombre"&y&" est ";
  if y<x
    print "trop petit"
  else if y>x
    print "trop grand"
  else
    print "bon !"
  y=timer
  message "Vous avez gagné|après "&score&" essais|e
    t en"&y\60&" m"&y mod 60&" s"
```

```

        ift sure?("Voulez vous rejouer?") exit
        stop
    endif
loop
loop

```

Exemple de dialogue

```

Quel nombre proposez-vous ?
INPUT >
1/2
Votre nombre 1/2 est trop petit
Quel nombre proposez-vous ?
INPUT >
1/2+1/6
Votre nombre 2/3 est trop grand
Quel nombre proposez-vous ?
INPUT >
3/5
Votre nombre 3/5 est bon !

```

Fractions égyptiennes

Le nom de fraction égyptienne pour désigner les nombres rationnels $1/n$ (n entier) vient du fait que les Egyptiens utilisaient presque uniquement ce type de fractions. Ils représentaient les autres nombres rationnels par des sommes de fractions égyptiennes, par exemple pour doubler $1/97$ ils connaissaient la relation :

$$2 \times \frac{1}{97} = \frac{1}{56} + \frac{1}{679} + \frac{1}{776}. \quad (2.8)$$

La c_fonction **egypt**(x, a_0, m) essaie de représenter le nombre rationnel x par une somme de m fractions égyptiennes :

$$x = \frac{1}{a_1} + \frac{1}{a_2} + \cdots + \frac{1}{a_m}, \quad (2.9)$$

où les entiers a_i vérifient les inégalités strictes : $a_0 < a_1 < a_2 < \cdots < a_m$. Si une telle représentation est trouvée, la fonction renvoie une chaîne contenant son écriture sous forme ASCII. S'il n'en existe pas, la fonction **egypt** renvoie la chaîne vide. Le deuxième argument a_0 de **egypt** permet l'écriture d'un programme fonctionnant de façon récursive. Si $m = 1$, on doit seulement vérifier que $x = 1/a_1$ où a_1 est un entier plus grand que a_0 . Si $m > 1$, la première fraction $1/a_1$ est la plus grande des m fractions. On doit donc avoir $a_0 < a_1 < m/x$ et le nombre $x - 1/a_1$ doit pouvoir s'écrire comme une somme de $m - 1$ fractions égyptiennes différentes de dénominateurs strictement plus grand que a_1 . La recherche est ainsi effectuée par une boucle sur les valeurs possibles de a_1 (qui est désigné par **b**), avec un appel récursif de **egypt**. L'existence d'une décomposition est testée par la fonction **len** qui renvoie la longueur d'une chaîne. L'exemple détermine des décompositions comportant le moins possible de fractions pour quelques

nombres rationnels, en appelant `egypt` avec des valeurs croissantes de m jusqu'à ce qu'une solution soit trouvée.

```

forv x in (2/3,2/5,2/7,4/5)
  m=0
  repeat
    m=m+1
    c$=egypt(x,1,m)
  until len(c$)
  print x;"=";c$
nextv
stop
egypt:function$(x,index a,m)
  if m=1
    ift not integerp(1/x) return
    ift 1/x<=a return
    value=justl$(x)
    return
  else
    local index b
    b=int(m/x)
    ift b=m/x b=b-1
    ift b<=a return
    for b=b,a+1
      value=egypt(x-1/b,b,m-1)
      if len(value)
        value=justl$(1/b)&"+"&value
      return
    endif
  next
  return
endif

```

Sortie (745 ms)

2/3=1/2+1/6

2/5=1/3+1/15

2/7=1/4+1/28

4/5=1/2+1/5+1/10

Exercice 2.7. 2/97 égyptien

Le programme `egypt` ci-dessus ne permet de déterminer qu'une seule représentation de 2/97 comme somme de trois fractions égyptiennes. Ecrire un programme donnant toutes les décompositions de 2/97 en somme de trois fractions :

$$\frac{2}{97} = \frac{1}{a_1} + \frac{1}{a_2} + \frac{1}{a_3}, \quad (2.10)$$

où les nombres a_i sont des entiers croissants : $a_1 < a_2 < a_3$.

Exercice 2.8. Stein

Soit p/q un nombre rationnel ($0 < p/q < 1$ et q impair). Ecrire un programme qui décompose ce nombre en une somme de fractions égyptiennes $\sum 1/a_i$ où le nombre a_i ($i = 1, 2, \dots$) est le plus petit nombre impair, différent des a_j ($j < i$), tel que le reste $x - \sum_{j=1}^i a_j$ soit positif ou nul. Par exemple :

$$\frac{6}{11} = \frac{1}{3} + \frac{1}{5} + \frac{1}{83} + \frac{1}{13695}. \quad (2.11)$$

On ignore si la somme $\sum 1/a_i$ est toujours finie.

Exercice 2.9. $16/64 = 1/4$

La simplification de $16/64$ en barrant les 6 donne une égalité exacte :

$$\frac{16}{64} = \frac{1\cancel{6}}{6\cancel{4}} = \frac{1}{4}. \quad (2.12)$$

Déterminer les fractions écrites avec des nombres inférieurs à 100 qui se simplifient de la même manière.

Exercice 2.10. $6729/13458 = 1/2$

La fraction $6729/13458$ utilise exactement tous les chiffres de 1 à 9. Trouver toutes les fractions de ce type qui se réduisent à $1/2$.

Instabilités

Un des domaines où le calcul formel montre son intérêt est celui de la résolution de problèmes instables. Comme exemple, nous examinons (Muller 1989 p 48) la suite (a_1, a_2, a_3, \dots) définie par récurrence :

$$\begin{aligned} a_1 &= 11/2, \\ a_2 &= 61/11, \\ a_{n+1} &= 111 - \frac{1130}{a_n} + \frac{3000}{a_n a_{n-1}}. \end{aligned} \quad (2.13)$$

La limite de cette suite est 6, mais par suite des erreurs d'arrondis et de l'instabilité de la suite, un calcul numérique en flottant donne sur tout ordinateur une série convergent vers 100. Le programme suivant (calcul en flottant) montre ce phénomène.

```
a=11/2~
b=61/11
for n=3,30
```

```

c=111-1130/b+3000/b/a
print n;c
a=b
b=c
next n

```

Sortie (2505 ms)

```

3  0.5590163934~ E+1~
4  0.5633431085~ E+1~
5  0.5674648620~ E+1~
6  0.5713329052~ E+1~
7  0.5749120910~ E+1~
8  0.5781810747~ E+1~
9  0.5811311241~ E+1~
10 0.5837604978~ E+1~
11 0.5860068323~ E+1~
12 0.5866308996~ E+1~
13 0.5642342096~ E+1~
14 0.1363757631~ E+1~
15 -0.3277186453
16 0.1077356011~ E+3~
17 0.1004263903~ E+3~
18 0.1000252547~ E+3~
19 0.1000015039~ E+3~
20 0.1000000897~ E+3~
21 0.1000000054~ E+3~
22 0.1000000003~ E+3~
23 0.1000000000~ E+3~
...
30 1.0000000000~ E+2~

```

Il est possible, en Basic 1000d d'observer la convergence vers 6 de la suite a_n en effectuant un calcul exact. La fonction $a(n)$ calcule par récurrence la valeur exacte du n -ième terme de la suite. La commande **remember** rend ce calcul presque aussi rapide qu'un calcul par itération (comme celui du programme précédent).

```

for n=1,150
  print using "n=###  an=####.#####",n,a(n)
next n
stop
a:function(n)
  remember n
  select n
  case=1
    value=11/2
  case=2

```

```

        value=61/11
    case others
        value=111-1130/a(n-1)+3000/a(n-1)/a(n-2)
    endselect
    return

```

Sortie (64 s)

```

n= 1  an=  5.5000000000
n= 2  an=  5.5454545455~
n= 3  an=  5.5901639344~
n= 4  an=  5.6334310850~
n= 5  an=  5.6746486205~
n= 6  an=  5.7133290524~
n= 7  an=  5.7491209197~
n= 8  an=  5.7818109205~
n= 9  an=  5.8113142383~
n= 10 an=  5.8376565490~
n= 11 an=  5.8609515225~
n= 12 an=  5.8813772158~
n= 13 an=  5.8991539058~
n= 14 an=  5.9145249507~
n= 15 an=  5.9277414078~
n= 16 an=  5.9390504855~
n= 17 an=  5.9486874925~
n= 18 an=  5.9568707319~
n= 19 an=  5.9637987208~
n= 20 an=  5.9696491440~
...
n=150 an=  6.0000000000~

```

Dans les systèmes d'équations, les instabilités des racines sont très fréquentes. Nous en verrons quelques exemples dans le chapitre suivant.

Nombres complexes

Les nombres complexes sont très simples à utiliser, tant en flottant qu'en exact, après déclaration du symbole représentant $\sqrt{-1}$.

Un calcul en exact

Le programme calcule le pgcd de deux entiers de Gauss a et b , $g = \text{cxgcd}(a, b)$. C'est un entier de Gauss de norme maximum qui divise a et b . Le pgcd est unique, à une unité ± 1 ou $\pm i$ près. Le littéral i (tout autre nom est

possible) représente le nombre complexe i . Lors d'une assignation, une expression complexe est simplifiée en tenant compte de $i^2 + 1 = 0$, mais dans les deux derniers `print`, les expressions a/g (comparer à la valeur affichée de a') et ga' sont affichées sans réduction.

```
complex i
a=-4717+12373*i
b=303+423*i
g=cxgcd(a,b)
ap=a/g
bp=b/g
print using "(#)_*(#)=#";g;ap;a
print using "(#)_*(#)=#";g;bp;b
print a/g
print g*ap
```

Sortie (875 ms)

```
(7*i +17)*(720*i +19)=12373*i -4717
(7*i +17)*(15*i +24)=423*i +303
[7*i +17]^-1* [12373*i -4717]
5040*i^2 +12373*i +323
```

Un calcul en flottant

Les fonctions `sqr`, `log`, `exp` et les exposants sont également définis pour les nombres complexes. Le programme affiche, en précision 20, i^i , qui est un nombre réel, puis calcule une expression g . Le nombre complexe conjugué s'obtient par `cc`, les parties réelles et imaginaires par `re` et `im` respectivement. On vérifie ensuite que $g = re^{ia}$ où le module et l'argument sont donnés par `cabs` et `carg`. Enfin, la fonction `cxint` détermine le plus proche entier de Gauss.

```
complex i
precision 20
print i^i
g=(7+8*i)^(9+10*i)*sqr(-1)*log(-10)+exp(i*2*pi/3)
print cc(g)
print re(g);im(g)
r=cabs(g)
a=carg(g)
print r;a
format -5
print cabs(1-g/r*exp(-i*a))
print cxint(g)
```

Sortie (2840 ms)

```
0.20787957635076190855~
-0.99117846400142728665~ E+6 -i*0.91193045118753488584~ E+6
-0.99117846400142728665~ E+6 0.91193045118753488584~ E+6
0.13468674379104016466~ E+7 0.23978117903832015687~ E+1
```



```
0.1846~ E-23
911930*i -991178
```

Fonctions aléatoires

Nombres flottants aléatoires

La fonction `rnd` renvoie un nombre flottant aléatoire entre 0 et 1. Le programme calcule 2^{x^y} et $(2^x)^y$ pour deux nombres x et y aléatoires.

```
x=rnd
y=rnd
print x;y
print 2^(x^y);(2^x)^y
```

Sortie (330 ms)

```
0.6904809730~ 0.3778056281~
0.1826930782~ E+1 0.1198199052~ E+1
```

Entiers aléatoires

La fonction `random(p)`, où p est un entier, renvoie un entier aléatoire entre 0 et $p - 1$. Le programme détermine, en effectuant $N = 10000$ tirages de paires d'entiers aléatoires, la probabilité pour que deux entiers $< p$ choisis au hasard soient premiers entre eux. La valeur obtenue est comparée à $6/\pi^2$, qui est la limite de cette probabilité lorsque $p \rightarrow \infty$.

```
p=10^20
N=10000
for i=1,N
    ift gcdr(random(p),random(p))=1 S=S+1
next i
print using "###.#####";float(S/N);6/pi^2
```

Sortie (394 s)

```
0.60350~ 0.60793~
```

Entier de Gauss aléatoire

La fonction `random(p, i, 1)` renvoie le nombre complexe `random(p)i + random(p)`.

```
complex i
print random(100,i,1)
```

Sortie (25 ms)

```
39*i +10
```

Permutations aléatoires

La fonction `nextperm(N, P(1), -1)` renvoie une permutation aléatoire de $1, 2, \dots, N$. La commande `randomize r`, initialise les générateurs aléatoires avec le nombre r , si $r \neq 0$, ou avec le compteur 200 Hz si $r = 0$. Le programme suivant affiche toujours la même permutation, mais il suffit de supprimer la commande `randomize` (ou utiliser `randomize 0`) pour que la permutation affichée soit pratiquement toujours différente à chaque exécution.

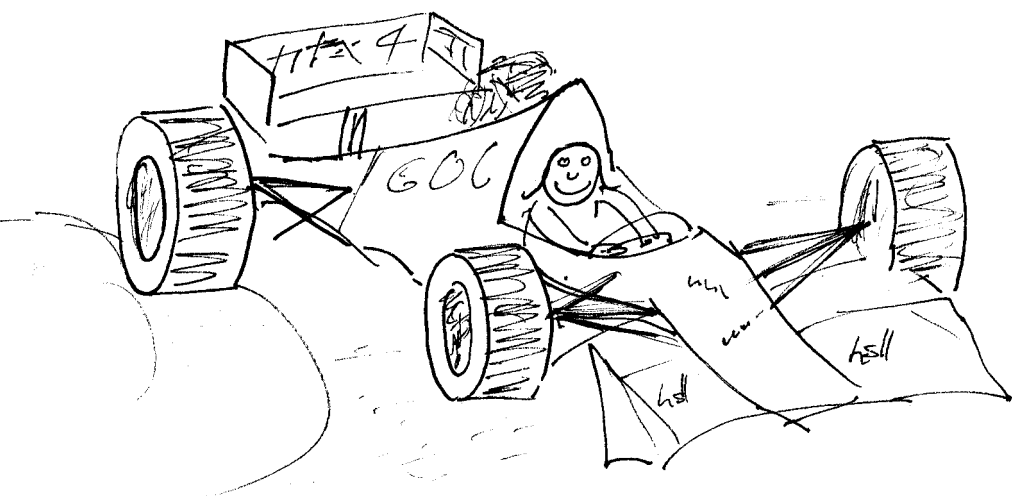
```
N=15
index P(N)
randomize 64400
k=nextperm(N,P(1),-1)
print conc$(i=1,N of P(i))
```

Sortie (140 ms)

```
13 4 7 8 12 5 1 2 15 6 9 10 3 11 14
```

3

Calcul formel



Polynômes et fractions rationnelles

Variables et littéraux

Le programme suivant est un exemple de calcul formel en Basic 1000d.

```
u=(a+b)^2
print u
type a,b,u
```

Sortie (135 ms)

```
a^2 +2*a*b +b^2
a est de type lit
b est de type lit
u est de type var
```

En Basic ordinaire, **a**, **b** et **u** sont des variables qui prennent certaines valeurs. Par exemple si **a** vaut 1 et **b** vaut 2, alors **u** vaut 9. Le même programme écrit en Basic 1000d est très différent. Le nom (ou identificateur) **a** qui apparaît à droite dans la première ligne est un littéral. Il n'a pas de valeur, c'est seulement un symbole. De même **b** est aussi un littéral. **u** qui apparaît à gauche est une variable. Sa valeur est l'expression $(a+b)^2$. Le résultat du **print** est la forme développée de l'expression. La commande **type** permet d'écrire le type des identificateurs.

Dans le programme suivant, la variable **Ms** est remplacée par sa valeur lors du calcul de l'expression assignée à **u**. Remarquer que les identificateurs **ms** et **Ms** sont différents, mais par contre **PRINT** est identique à **print** (mot clef du Basic).

```
Ms=x1^2+1
u=ms+Ms
PRINT u
type Ms,ms
```

Sortie (120 ms)

```
x1^2 +ms +1
Ms est de type var
ms est de type lit
```

Voici maintenant deux exemples d'erreurs qui produisent des diagnostics assez peu explicites. Dans le programme suivant, après exécution de la première commande, **t** est un littéral. Il est impossible de lui donner une valeur, d'où l'erreur Instruction Illégale.

```
z=2*t
t=3
```

Sortie (55 ms)

```
*ERREUR* INSTRUCTION ILLEGALE
```

```
?
2.t=3
```

Le Basic ne traite que les polynômes à exposants entiers (n ne peut pas être un littéral dans x^n). Comme le Basic n'a pu interpréter x^n ni comme un polynôme, ni comme un nombre complexe flottant, il a écrit le message d'erreur Nombre Complexe.

```
u=x^n
```

Sortie (35 ms)

```
*ERREUR* NOMBRE COMPLEXE
u=x^n?
1.u=x^n
```

Forme canonique des polynômes

Des polynômes égaux sont codés en mémoire sous la forme polynôme de la même façon. Les littéraux sont ordonnés suivant l'ordre dans lequel ils sont décodés par le Basic.

```
p=x^5+2*x+6
print p
p=x+(x+3/2+x*x^4+9/2)
print p
print 2*a+b+c
print b+a+c+a
```

Sortie (125 ms)

```
x^5 +2*x +6
x^5 +2*x +6
2*a +b +c
2*a +b +c
```

Fractions rationnelles

La forme produit de polynômes, où à chaque facteur est associé un exposant, permet au Basic 1000d de manipuler les fractions rationnelles. Les expressions sont automatiquement simplifiées, de sorte que les divers facteurs soient toujours premiers deux à deux. Dans l'exemple suivant, le numérateur et dénominateur sont simplifiés par le facteur $x - b$.

```
print (x^2 -x*a -x*b +a*b)/(x-b)^2
```

Sortie (365 ms)

```
[x -b]^~1* [x -a]
```

Plusieurs formes d'une même fraction rationnelle

À la différence de la forme polynôme, une même expression peut avoir plusieurs formes produit de polynômes. Ainsi les expressions p et q , qui sont mathématiquement égales, ont des formes différentes.

```
p=(x^2+2*x+1)^-2
q=(x+1)^-4
print p
```

```

print q
print p-q
Sortie (115 ms)
[x^2 +2*x +1]^2
[x +1]^4
0

```

Formes factorisées et développées

Après la commande `develop`, ou au début d'un programme, les expressions sont développées autant que possible. Par contre, après la commande `factor`, les expressions restent factorisées. Le programme ci-dessous donne aussi des exemples d'écriture ordonnée suivant les puissances croissantes ou décroissantes de l'exposant du littéral `x`, obtenus par la fonction `str$`.

```

p=(a*x-y*z)*(y+b*z)*(x-2*z)
print "développé",p
factor
p=(a*x-y*z)*(y+b*z)*(x-2*z)
print "reste factorisé",p
develop
p=p
print "redéveloppé",p
p=a*x^2+17/8*x*a+x
print "ordonné en x",str$(p,x)
print "ordonné en /x",str$(p,/x)
Sortie (955 ms)
développé      a*x^2*y +a*x^2*z*b -2*a*x*y*z -2*a*x*z^2*b -x*y^2*z
              -x*y*z^2*b +2*y^2*z^2 +2*y*z^3*b
reste factorisé [x -2*z]* [y +z*b]* [a*x -y*z]
redéveloppé    a*x^2*y +a*x^2*z*b -2*a*x*y*z -2*a*x*z^2*b -x*y^2*z
              -x*y*z^2*b +2*y^2*z^2 +2*y*z^3*b
ordonné en x   ( [a])*x^2+( 1/8* [17*a +8])*x
ordonné en /x  ( 1/8* [17*a +8])*x+( [a])*x^2

```

Factorisation

La fonction `formf` factorise les expressions.

```

print formf(x^3 -x^2 -3*x*a^2 -x*a +2*a^3 +2*a^2)
Sortie (690 ms)
[x -a]* [x +2*a]* [x -a -1]

```

La première instruction du programme suivant a pour effet de factoriser $x^2 - 1$ (dans la fonction `formf`) puis de redévelopper la forme obtenue (dans l'assignation `=`). Ce phénomène n'apparaît pas dans le programme précédent, où le résultat de `formf` est directement affiché. La suite du programme montre comment ne pas faire travailler `formf` pour rien. Il faut se placer en mode `factor`, dans lequel les expressions ne sont pas développées automatiquement comme en mode `develop`.

```
p=formf(x^2-1)
print p
factor
p=formf(x^2-1)
print p
```

Sortie (270 ms)

```
x^2 -1
[x -1]* [x +1]
```

Basic 1000d factorise les expressions complètement en facteurs irréductibles à coefficients rationnels. Par exemple, la factorisation $x^2 - 2 = (x - \sqrt{2})(x + \sqrt{2})$ ne peut être obtenue par `formf`. Le programme suivant factorise $x^n + 1$ pour divers n . Le premier nombre de chaque ligne donne le temps en ms de la factorisation.

```
for i=2,15
  develop
  w=1+x^i
  factor
  clear timer
  f=formf(w)
  print justr$(mtimer,5);justr$(w,10);"=";f
next i
```

Sortie (35605 ms)

```
35   x^2 +1=   [x^2 +1]
130  x^3 +1=   [x +1]* [x^2 -x +1]
310  x^4 +1=   [x^4 +1]
290  x^5 +1=   [x +1]* [x^4 -x^3 +x^2 -x +1]
765  x^6 +1=   [x^2 +1]* [x^4 -x^2 +1]
625  x^7 +1=   [x +1]* [x^6 -x^5 +x^4 -x^3 +x^2 -x +1]
910  x^8 +1=   [x^8 +1]
1065 x^9 +1=   [x +1]* [x^6 -x^3 +1]* [x^2 -x +1]
1960 x^10 +1=  [x^2 +1]* [x^8 -x^6 +x^4 -x^2 +1]
2665 x^11 +1=  [x +1]* [x^10 -x^9 +x^8 -x^7 +x^6 -x^5 +x^4 -x^3 +x^2 -x +1]
5380 x^12 +1=  [x^4 +1]* [x^8 -x^4 +1]
6030 x^13 +1=  [x +1]* [x^12 -x^11 +x^10 -x^9 +x^8 -x^7 +x^6 -x^5 +x^4 -x^3 +x^2 -x +1]
4620 x^14 +1=  [x^2 +1]* [x^12 -x^10 +x^8 -x^6 +x^4 -x^2 +1]
```

```
10315  x^15 +1=  [x +1]* [x^2 -x +1]* [x^4 -x^3 +x^2 -x +1]* [x^8 +x^
7 -x^5 -x^4 -x^3 +x +1]
```

L'exemple suivant factorise l'expression à deux littéraux

$$(x^2 + xy + y^2)^n - (x^2 - xy + y^2)^n \quad (3.1)$$

pour les valeurs $n = 2, 3, \dots, 10$.

```
for n=2,10
  print using "(n=##)",n;
  print formf((x^2+x*y+y^2)^n-(x^2-x*y+y^2)^n)
next n
```

Sortie (46 s)

```
(n= 2)  4* [y]* [x]* [x^2 +y^2]
(n= 3)  2* [y]* [x]* [3*x^4 +7*x^2*y^2 +3*y^4]
(n= 4)  8* [y]* [x]* [x^2 +y^2]* [x^4 +3*x^2*y^2 +y^4]
(n= 5)  2* [y]* [x]* [5*x^4 +5*x^2*y^2 +y^4]* [x^4 +5*x^2*y^2 +5*y^4]
(n= 6)  4* [y]* [x]* [x^2 +y^2]* [3*x^4 +7*x^2*y^2 +3*y^4]* [x^4 +5*x^
2*y^2 +y^4]
(n= 7)  2* [y]* [x]* [7*x^12 +77*x^10*y^2 +266*x^8*y^4 +393*x^6*y^6 +2
66*x^4*y^8 +77*x^2*y^10 +7*y^12]
(n= 8)  16* [y]* [x]* [x^2 +y^2]* [x^4 +3*x^2*y^2 +y^4]* [x^8 +10*x^6*
y^2 +19*x^4*y^4 +10*x^2*y^6 +y^8]
(n= 9)  2* [y]* [x]* [3*x^4 +7*x^2*y^2 +3*y^4]* [3*x^12 +45*x^10*y^2 +
186*x^8*y^4 +289*x^6*y^6 +186*x^4*y^8 +45*x^2*y^10 +3*y^12]
(n=10)  4* [y]* [x]* [x^2 +y^2]* [5*x^4 +5*x^2*y^2 +y^4]* [x^4 +5*x^2*
y^2 +5*y^4]* [x^8 +14*x^6*y^2 +31*x^4*y^4 +14*x^2*y^6 +y^8]
```

Décorticage

Degré, Ordre et Coefficients

Ordre et degré en x de l'expression :

$$p = \frac{a+b}{x^7} + \frac{7}{ax^5} + \frac{12x^{15}}{a+b+1}. \quad (3.2)$$

On sort aussi le coefficient de x^{-7} de l'expression.

```
p=(a+b)/x^7+7/a/x^5+12*x^15/(a+b+1)
print ordof(p,x);degf(p,x)
print coeff(p,x,-7)
```

Sortie (210 ms)

```
-7  15
[a +b]
```


Décorticage

Pour le polynôme w on détermine le plus grand coefficient en valeur absolue. Pour cela, on décompose le polynôme w suivant ses `polymn(w)` monômes, en utilisant la fonction `polym`, et pour chaque monôme, on extrait le coefficient par `norm`.

```
w=(17-23*x^2+y-12*y^2)^10
M=abs(norm(w))
for i=1,polymn(w)
    monome=polym(w,i)
    M=max(M,abs(norm(monome)))
next i
print M
```

Sortie (2375 ms)

```
11306291097123540
```

Substituer et Homogénéiser

Substitutions

La fonction `subs` permet de remplacer x puis y par des expressions. Dans la deuxième instruction, la substitution de s^2 est effectuée par `subsr`.

```
print subs((x+y)^2,x=x^5+y,y=-1/2)
print subsr((1+s)^4,s^2=t^3)
```

Sortie (160 ms)

```
[x^10 -2*x^5 +1]
4*s*t^3 +4*s +t^6 +6*t^3 +1
```

Homogénéisation

La fonction `homog` homogénéise p par le littéral x . L'exemple montre aussi comment déshomogénéiser par `subs`.

```
p=(12-4*y^2)/(1+5*m^4-m^8)
W=homog(p,x)
print W
print subs(W,x=1)
```

Sortie (185 ms)

```
4* [y^2 -3*x^2]* [m^8 -5*m^4*x^4 -x^8]^-1
4* [y^2 -3]* [m^8 -5*m^4 -1]^-1
```

Division et PGCD

Division

La division euclidienne des polynômes s'effectue à l'aide de `mod` et `div`.

```

p=7*x^2-x+a
q=x+b
print "quotient";div(p,q,x)
print "reste   ";mod(p,q,x)

```

Sortie (165 ms)

```

quotient  [7*x -7*b -1]
reste     a +7*b^2 +b

```

pgcd

Le pgcd de deux ou plusieurs polynômes s'obtient avec `gcd`.

```

p=(x-a)^3*(y-b)^2*(z-1)
q=(x-a)*(y-b)^3
g=gcd(p,q)
print g
print formf(g)

```

Sortie (745 ms)

```

-x*y^2 +2*x*y*b -x*b^2 +a*y^2 -2*a*y*b +a*b^2
- [y -b]^2* [x -a]

```

Division suivant les puissances croissantes

La division suivant les puissances croissantes peut être effectuée par un développement limité comme nous le verrons plus loin. On trouvera ici la fonction `quotient` qui programme directement cette division. L'appel `quotient(p, q, n, x)` effectue la division de p par q , suivant le littéral x , à l'ordre n et renvoie le quotient de la division. La fonction `litp(x)` teste si x est bien un littéral.

```

print quotient(x^2+1,x-1,10,x)
stop

```

```

quotient:function(p,q,n,x)
  ift not litp(x) erreur_arg3_non_lit
  local datai ordf(q,x) index ordq,ordp
  local datav coeff(q,x,ordq) var coefq,p1
  ift ordf(p,x)<ordq erreur_division_impossible
  while degf(value,x)<n
    ordp=ordf(p,x)
    p1=coeff(p,x,ordp)*x^(ordp-ordq)/coefq
    vadd p,-p1*q
    vadd value,p1
  end
end

```

```
wend
return
```

Sortie (745 ms)

```
-2*x^10 -2*x^9 -2*x^8 -2*x^7 -2*x^6 -2*x^5 -2*x^4 -2*x^3 -2*x^2 -x -1
```

La fonction `quotient` fonctionne avec des polynômes généralisés en x .

```
p=x/c+a
q=x^2+x+b
r=quotient(p,q,5,x)
print "r=";str$(r,/x)
print
print "p-q*r=";str$(p-q*r,/x)
```

Sortie (2080 ms)

```
r= ( [b]^-1* [a])+( - [b]^-2* [c]^-1* [c*a-b])*x+( - [b]^-3* [c]^-1
* [c*a*b -c*a +b])*x^2+( [b]^-4* [c]^-1* [2*c*a*b -c*a -b^2 +b])*x^
3+( [b]^-5* [c]^-1* [c*a*b^2 -3*c*a*b +c*a +2*b^2 -b])*x^4+( - [b]^
-6* [c]^-1* [3*c*a*b^2 -4*c*a*b +c*a -b^3 +3*b^2 -b])*x^5
```

```
p-q*r= ( - [b]^-6* [c]^-1* [c*a*b^3 -6*c*a*b^2 +5*c*a*b -c*a +3*b^3 -4
*b^2 +b])*x^6+( [b]^-6* [c]^-1* [3*c*a*b^2 -4*c*a*b +c*a -b^3 +3*b^
2 -b])*x^7
```

Fonctions symétriques

Nous considérons des expressions symétriques en a , b et c . La fonction `symf` de la bibliothèque MATH nous permet de récrire de telles expressions en fonction des fonctions symétriques $p_1 = a + b + c$, $p_2 = ab + bc + ca$ et $p_3 = abc$. Nous obtenons ainsi :

$$\begin{aligned} a^2 + b^2 + c^2 &= p_1^2 - 2p_2 \\ a^3 + b^3 + c^3 &= p_1^3 - 3p_1p_2 + 3p_3 \end{aligned} \tag{3.3}$$

...

```
var x(3),p(3),sv(3)
x(1)=a
x(2)=b
x(3)=c
p(1)=p1
p(2)=p2
p(3)=p3
sv(1)=a+b+c
```

```

sv(2)=a*b+b*c+c*a
sv(3)=a*b*c
forv i in (1,2,3,4,9)
  print using "a_^#+b_^#+c_^#=",i,i,i;
  print symf(a^i+b^i+c^i,3,x,p,sv)
nextv

```

Sortie (7575 ms)

```

a^1+b^1+c^1= p1
a^2+b^2+c^2= p1^2 -2*p2
a^3+b^3+c^3= p1^3 -3*p1*p2 +3*p3
a^4+b^4+c^4= p1^4 -4*p1^2*p2 +4*p1*p3 +2*p2^2
a^9+b^9+c^9= p1^9 -9*p1^7*p2 +9*p1^6*p3 +27*p1^5*p2^2 -45*p1^4*p2*p3
-30*p1^3*p2^3 +18*p1^3*p3^2 +54*p1^2*p2^2*p3 +9*p1*p2^4 -27*p1*p2*p3^
2 -9*p2^3*p3 +3*p3^3

```

Décomposition en éléments simples

La fonction `partfrac$` suivante décompose une fraction rationnelle en ses éléments simples, si son dénominateur se factorise en un produit de facteurs de degré 0 ou 1 en x . Les pôles en x sont obtenus par `sroot`, et la partie singulière de chaque pôle est déterminé par `psing`. L'exemple calcule la décomposition suivante de r :

$$17x + 15 + \frac{1}{x-1} + \frac{7}{x-a} - \frac{2}{(x-a)^2} \quad (3.4)$$

```

r=(17*x^4 -34*x^3*a -2*x^3 +17*x^2*a^2 +4*x^2*a -7*x^2
-2*x*a^2 +21*x*a -9*x -14*a^2 +7*a +2)/(x^3 -2*x^2*a -
x^2 +x*a^2 +2*x*a -a^2)
print partfrac$(r,x)
stop

```

```

partfrac$:function$(w,x)
  local lit p_y
  local index i
  local var R,p,q,w1
  local char c$,d$
  push factor
  factor
  R=contf(w,x)
  w1=w/R
  w=formf(denf(w1))

```

```

ift R<>1 value=R&" * ["
for i=2,factorn(w)
  p=factorp(w,i)
  if deg(p,x)=1
    q=sroot(p,x)
    p=psing(w1,x,q,p_y)
    d$="["&formd(-q)&" + "&x&" "]"
    d$=justl$(change$(str$(p,p_y),"p_y",d$))
    ift left$(d$)<>"-" d$="+ " &d$
    cadd c$," " &d$
    vadd w1,-subs(p,p_y=x-q)
  endif
next i
ift w1 cadd value,w1
cadd value,c$
ift R<>1 cadd value," "]"
factor pop
return

```

Sortie (1225 ms)

```

[17*x +15] + ( 1)*[ -1 + x ]^-1 + ( 7)*[ -a + x ]^-1+( -2
)*[ -a + x ]^-2

```

Développements limités

Fractions rationnelles

Le développement limité d'une fraction rationnelle, autour de zéro, s'obtient par `taylor`. Il est commode d'utiliser `str$` pour afficher les résultats sous la forme habituelle. Dans l'exemple, `taylor` renvoie la somme des 6 premiers termes du développement limité :

$$\frac{x^2}{1-x} = x^2(1 + x + x^2 + x^3 + x^4 + x^5 + \dots). \quad (3.5)$$

```
print str$(taylor(x^2/(1-x),5),/x)
```

Sortie (150 ms)

```
( 1)*x^2+( 1)*x^3+( 1)*x^4+( 1)*x^5+( 1)*x^6+( 1)*x^7
```

Ce développement limité est analogue à la division suivant les puissances croissantes. Le programme suivant effectue à l'aide de `taylor`, les divisions que nous avons réalisées avec la fonction externe `quotient`.

```
p=taylor((x^2+1)/(x-1),10,x)
```

```

print p
print
p=x/c+a
q=x^2+x+b
r=taylor(p/q,5,x)
print str$(r,/x)

```

Sortie (930 ms)

```

-2*x^10 -2*x^9 -2*x^8 -2*x^7 -2*x^6 -2*x^5 -2*x^4 -2*x^3 -2*x^2 -x -1

```

```

( [b]^-1* [a])+( - [b]^-2* [c]^-1* [c*a -b])*x+( - [b]^-3* [c]^-1*
[c*a*b -c*a +b])*x^2+( [b]^-4* [c]^-1* [2*c*a*b -c*a -b^2 +b])*x^3+
( [b]^-5* [c]^-1* [c*a*b^2 -3*c*a*b +c*a +2*b^2 -b])*x^4+( - [b]^-6
* [c]^-1* [3*c*a*b^2 -4*c*a*b +c*a -b^3 +3*b^2 -b])*x^5

```

Fonctions transcendentes

Les développements limités de $\sin x$, $\cos x$, $\exp x$ et $\log 1+x$ au voisinage de $x = 0$, sont donnés par les fonctions `ssin`, `scos`, `sexp` et `slog1`. Voici comment on peut obtenir les développements limités de $\tan x$, puis de $\exp(\tan x)$ et de $\log(\cos x)$, autour de $x = 0$, à l'ordre $k = 21$.

```

k=21
t=taylor(ssin(x,k)/scos(x,k),k)
print "tan x=";str$(t,/x);" + ..."
y=sexp(t,k,x)
print "exp(tan(x))=";str$(y,/x);" + ..."
y=slog1(scos(x,k)-1,k-2)
print "log(cos(x))=";str$(y,/x);" + ..."

```

Sortie (61315 ms)

```

tan x= ( 1)*x+( 1/3)*x^3+( 2/15)*x^5+( 17/315)*x^7+( 62/2835)*x^9
+( 1382/155925)*x^11+( 21844/6081075)*x^13+( 929569/638512875)*x^1
5+( 6404582/10854718875)*x^17+( 443861162/1856156927625)*x^19+( 18
888466084/194896477400625)*x^21 + ...

```

```

exp(tan(x))= ( 1)+( 1)*x+( 1/2)*x^2+( 1/2)*x^3+( 3/8)*x^4+( 37/1
20)*x^5+( 59/240)*x^6+( 137/720)*x^7+( 871/5760)*x^8+( 41641/3628
80)*x^9+( 325249/3628800)*x^10+( 3887/57600)*x^11+( 35797/691200)*
x^12+( 241586893/6227020800)*x^13+( 24362249/830269440)*x^14+( 572
1418891/261534873600)*x^15+( 342232522657/20922789888000)*x^16+( 43
15903789009/355687428096000)*x^17+( 8224154352439/914624815104000)*x
^18+( 2832484672207/426824913715200)*x^19+( 23157229065769/47424990
41280000)*x^20+( 183184249105857781/51090942171709440000)*x^21 + ...

```

```

log(cos(x))= ( -1/2)*x^2+( -1/12)*x^4+( -1/45)*x^6+( -17/2520)*x^8+( -
31/14175)*x^10+( -691/935550)*x^12+( -10922/42567525)*x^14+( -929569/
10216206000)*x^16+( -3202291/97692469875)*x^18+( -221930581/185615692
76250)*x^20 + ...

```

La fonction hypergéométrique

La fonction hypergéométrique permet le développement de nombreuses fonctions usuelles. Voici le développement de $\sqrt{1+x}$ à l'ordre 10.

```
y=shyg(-x,10,x,-1/2,1,1,-1)
print str$(y,/x);" +..."
```

Sortie (1010 ms)

```
( 1)+( 1/2)*x+( -1/8)*x^2+( 1/16)*x^3+( -5/128)*x^4+( 7/256)*x^5+
( -21/1024)*x^6+( 33/2048)*x^7+( -429/32768)*x^8+( 715/65536)*x^9+(
-2431/262144)*x^10 +...
```

Plus généralement, le développement

$$(1+x)^s = 1 + \frac{(-s)(-x)}{1} + \frac{(-s)(-s+1)(-x)^2}{2!} + \dots \quad (3.6)$$

se calcule comme dans l'exemple suivant. Alors que le calcul précédent correspondait à la valeur numérique $s = 1/2$, ici nous gardons s sous forme littérale, et le développement est effectué à l'ordre 2.

```
w=shyg(-x,2,x,-s,1,1,-1)
print "(1+x)^s=";str$(w,/x);" +..."
```

Sortie (195 ms)

```
(1+x)^s= ( 1)+( [s])*x+( 1/2* [s]* [s -1])*x^2+...
```

Développement multipolaire

On développe

$$\frac{1}{|\vec{R} - \vec{r}|} = \frac{1}{R\sqrt{1 - 2r/R \cos \theta + r^2/R^2}} \quad (3.7)$$

suivant les puissances de r .

```
y=shyg(2*x*cos_theta-x^2,4,x,1/2,1,1,-1)/R
y=subs(y,x=r/R)
print str$(y,/r);" +..."
```

Sortie (1065 ms)

```
( [R]^-1)+( [R]^-2* [cos_theta])*r+( 1/2* [R]^-3* [3*cos_theta^2
-1])*r^2+( 1/2* [R]^-4* [cos_theta]* [5*cos_theta^2 -3])*r^3+( 1/8
* [R]^-5* [35*cos_theta^4 -30*cos_theta^2 +3])*r^4 +...
```

Calcul matriciel

Lecture et écriture de matrices

Pour manipuler les matrices en Basic 1000d, définir des tableaux bidimensionnels. Il est recommandé d'utiliser les indices à partir de 0, comme ici, la raison principale n'étant pas l'économie de mémoire, mais une exigence du programme d'inversion. L'initialisation d'une matrice se fait commodément par **read** et **data**. On donne une façon d'afficher une matrice en une seule instruction, par utilisation de **conc\$** (le symbole \mathfrak{a} est $[a] \mathbb{Z}$).

```
var M(3,3)
for i=0,3
  for j=0,3
    read M(i,j)
  next j,i
data -54,-52,-48,-44
data 780,725,660,600
data -2160,-1980,-1790,-1620
data 1540,1400,1260,1137
print "M="
print conc$(i=0,3 of conc$(j=0,3 of justc$(M(i,j),10)) $\mathfrak{a}$ 
)
```

Sortie (375 ms)

M=

-54	-52	-48	-44
780	725	660	600
-2160	-1980	-1790	-1620
1540	1400	1260	1137

Déterminant

Le déterminant :

$$\begin{vmatrix} a & b & c \\ 1/c & 2/a & 3/b \\ 4 & 5 & 6 \end{vmatrix} \quad (3.8)$$

est calculé par **det**. La fonction **det** permet de faire partir les indices de 0, 1 (comme ici) ou d'une autre valeur.

```
var D(3,3)
for i=1,3
  for j=1,3
    read D(i,j)
  next j,i
data a, b, c
data 1/c, 2/a, 3/b
data 4, 5, 6
print det(D,3)
```

Sortie (320 ms)

- $[c]^{-1} [b]^{-1} [a]^{-1} [15a^2c + 6ab^2 - 29abc + 8b^2c^2]$

Autre calcul du déterminant

La fonction `det` utilise la méthode de Bareiss qui en principe a un temps de calcul polynomial suivant le nombre de lignes n . Cependant, lorsque les éléments du déterminant sont des expressions formelles compliquées, il arrive que le calcul du déterminant par la somme sur $n!$ permutations soit plus rapide. Le calcul par cette méthode est effectué par la fonction `devdet` suivante. La syntaxe de `devdet` est la même que celle de `det`, mais il est recommandé d'utiliser un tableau comme entrée (et non pas une fonction, comme c'est admis pour `det`) chaque élément de la matrice étant lu plusieurs fois. La fonction interne `nextperm` réalise la boucle sur les permutations de $1, 2, \dots, n$, tout en fournissant la parité de chaque permutation. La fonction `devdet` est utilisée pour recalculer le déterminant précédent, mais dans ce cas `det` était plus rapide.

```

var D(3,3)
for i=1,3
  for j=1,3
    read D(i,j)
  next j,i
data a, b, c
data 1/c, 2/a, 3/b
data 4 , 5 , 6
print devdet(D,3)
stop

devdet:function
local datai @2,0 index devdet_N,devdet_k,devdet_i,
  devdet_j
if @0=3
  devdet_k=(@3)-1
  devdet_N=devdet_N-devdet_k
endif
local index devdet_P(devdet_N)
devdet_i=nextperm(devdet_N,devdet_P(1),0)
while devdet_i
  vadd value,devdet_i*prod(devdet_j=1,devdet_N of @1(
    devdet_k+devdet_j,devdet_k+devdet_P(devdet_j)))
  devdet_i=nextperm(devdet_N,devdet_P(1))
wend
return

```

Sortie (500 ms)

```
- [c]^-1* [b]^-1* [a]^-1* [15*a^2*c +6*a*b^2 -29*a*b*c +8*b*c^2]
```

Polynôme caractéristique

Le polynôme caractéristique de la matrice M s'obtient en calculant $P = M - \lambda$, puis son déterminant.

```
var M(3,3),P(3,3)
```

```

for i=0,3
  for j=0,3
    read M(i,j)
  next j,i
data -310*x+244,144*x-141,192*x-120,-20*x+23
data -808*x+1030,390*x-561,480*x-552,-56*x+89
data -136*x+34,60*x-27,90*x-6,-8*x+5
data -2320*x+3898,1152*x-2085,1344*x-2136,-170*x+329
print "M="
print conc$(i=0,3 of conc$(j=0,3 of justc$(M(i,j),18))æ
)
,
'calcul de P=M-lambda
,
for i=0,3
  for j=0,3
    P(i,j)=M(i,j)
    ift i=j P(i,j)=P(i,j)-lambda
  next j,i
,
'calcul et factorisation du polynôme caractéristique
,
print "polynôme caractéristique=";formf(det(P,3,0))

```

Sortie (3290 ms)

```

M=
-310*x +244      144*x -141      192*x -120      -20*x +23
-808*x +1030     390*x -561     480*x -552     -56*x +89
-136*x +34       60*x -27       90*x -6        -8*x +5
-2320*x +3898    1152*x -2085    1344*x -2136    -170*x +329

polynôme caractéristique=  [6*x -lambda -12]* [6*x -lambda +6]* [6*x
+lambda -18]* [6*x +lambda +6]

```

Inverse

L'inverse s'obtient par la procédure `invM` de la bibliothèque `MATH`.

```

var M(3,3),M1(3,3)
for i=0,3
  for j=0,3
    read M(i,j)
  next j,i
data -54,-52,-48,-44
data 780,725,660,600
data -2160,-1980,-1790,-1620
data 1540,1400,1260,1137

```

```

print "M="
print conc$(i=0,3 of conc$(j=0,3 of justc$(M(i,j),10))æ
)
invm M,M1,3
print "M^-1"
print conc$(i=0,3 of conc$(j=0,3 of justc$(M1(i,j),10))
æ)

```

Sortie (2100 ms)

M=

-54	-52	-48	-44
780	725	660	600
-2160	-1980	-1790	-1620
1540	1400	1260	1137

M^-1

-3/2	-14/5	-12/5	-2
42	193/5	30	24
-108	-90	-683/10	-54
70	56	42	33

Extensions algébriques

Calcul conditionnel

La commande `cond`, qui définit des conditions, permet de travailler sur les nombres algébriques. Ci-dessous, pour traiter exactement une expression contenant $\sqrt{3}$, on introduit le littéral `sqrt3` et la condition `sqrt3^2=3`. Le calcul montre que $(1 + \sqrt{3})^{10} = 11584 + 6688\sqrt{3}$.

```

cond sqrt3^2=3
y=(1+sqrt3)^10
print y

```

Sortie (115 ms)

6688*sqrt3 +11584

Le programme suivant simplifie des expressions contenant $x = \sqrt{i}$ et $j = e^{2\pi i/3}$, en définissant des conditions.

```

complex i
cond x^2=i,x
f=1+2*x+3*x^2+4*x^3+5*x^4+6*x^5
print f
cond j^2=j+1

```

```
g=(x+y+z)*(x+j*y+j^2*z)*(x+j^2*y+j*z)
print g
```

Sortie (385 ms)

```
4*i*x +3*i -4*x -4
i*x -3*x*y*z +y^3 +z^3
```

Substitutions

Une autre façon d'effectuer les simplifications de l'exemple précédent consiste à effectuer des substitutions avec `subsr` et `subsrr`.

```
complex i
f=1+2*x+3*x^2+4*x^3+5*x^4+6*x^5
f=subsr(f,x^2=i)
print f
g=(x+y+z)*(x+j*y+j^2*z)*(x+j^2*y+j*z)
g=subsrr(g,j^2=-j-1,x^2=i)
print g
```

Sortie (520 ms)

```
4*i*x +3*i -4*x -4
i*x -3*x*y*z +y^3 +z^3
```

Reste modulaire

Voici encore une autre façon d'effectuer les calculs précédents. Elle utilise la fonction `mod` pour réduire les expressions. Notons que la méthode utilisant les conditions fonctionne en effectuant les réductions par `mod` à chaque assignation.

```
complex i
f=1+2*x+3*x^2+4*x^3+5*x^4+6*x^5
f=mod(f,x^2-i,x)
print f
g=(x+y+z)*(x+j*y+j^2*z)*(x+j^2*y+j*z)
g=mod(mod(g,j^2+j+1),x^2-i,x)
print g
```

Sortie (350 ms)

```
4*i*x +3*i -4*x -4
i*x -3*x*y*z +y^3 +z^3
```

Simplification des fractions rationnelles

La fonction `inv` permet de simplifier les fractions rationnelles contenant des nombres algébriques. Supposons que a et b soient des nombres algébriques définis par les équations :

$$\begin{cases} a^2 + a + 1 = 0 \\ b^2 - b - a = 0. \end{cases} \quad (3.9)$$

Toute expression rationnelle en a et b peut se simplifier en un polynôme de degré 1 en a ou b . Proposons nous le problème de simplifier ainsi l'inverse de $b + a + 1$. Le calcul suivant :

```
print inv(b+a+1,b^2-b-a,b)
```

Sortie (125 ms)

```
- [a^2 +2*a +2]^-1* [b -a -2]
```

montre que :

$$\frac{1}{b+a+1} = -\frac{b-a-2}{a^2+2a+2}. \quad (3.10)$$

On montre ensuite que l'inverse de a^2+2a+2 est $-a$ par :

```
print inv(a^2 +2*a +2,a^2+a+1)
```

Sortie (55 ms)

```
- [a]
```

Nous regroupons ces résultats, en simplifiant à l'aide de `mod`.

```
z=mod([a] * [b -a -2],a^2+a+1)
```

```
print z
```

Sortie (40 ms)

```
a*b -a +1
```

Nous avons ainsi montré que les nombres $ab - a + 1$ et $b + a + 1$ sont inverses.

On peut le vérifier en simplifiant leur produit.

```
z=mod((a*b -a +1)*(b+a+1),b^2-b-a,b)
```

```
z=mod(z,a^2+a+1)
```

```
print z
```

Sortie (55 ms)

```
1
```

Résolution d'équations

Elimination

Une des fonctions les plus intéressante du Basic est `elim` qui permet d'éliminer une inconnue entre deux équations. Elle est utilisée ici pour résoudre le système d'équations :

$$\begin{cases} x + y + z = a + 3 \\ xyz = 2a \\ x^2 + y^2 + z^2 = a^2 + 5. \end{cases} \quad (3.11)$$

La solution est obtenue en éliminant y et z . Les racines en x sont les zéros de W , et se lisent de suite (1, 2 et a) sur la forme factorisée.

```
W1=elim( x+y+z-a-3, x*y*z-2*a , y)
```

```
W2=elim( x+y+z-a-3, x^2+y^2+z^2-a^2-5, y)
```

```
W=elim(W1, W2, z)
```

```
print formf(W)
```

Sortie (520 ms)

```
4* [x -2]^2* [x -1]^2* [x -a]^2
```

Système d'équations linéaires

Pour la commodité, la résolution des systèmes d'équations, à l'aide de `elim`, a été programmée dans la bibliothèque MATH. L'exemple résout d'abord le système d'équations :

$$\begin{cases} 13x + 4y = 38 \\ 13x + (4 + 10^{-7})y = 38 + 3 \times 10^{-7}, \end{cases} \quad (3.12)$$

puis le système :

$$\begin{cases} 13x + 4y = 38 \\ 13x + (4 - 10^{-7})y = 38 + 3 \times 10^{-7}. \end{cases} \quad (3.13)$$

On notera l'instabilité de ces systèmes, qui diffèrent par la très petite variation d'un seul coefficient, mais dont les solutions sont très différentes.

```
var eq1(1),z(1),vz(1)
z(0)=x
z(1)=y
eq1(0)=13*x+4*y-38
eq1(1)=13*x+4.0000001*y-38.0000003
sleq eq1,z,vz,1
print "x=";vz(0);" y=";vz(1)
eq1(1)=13*x+3.9999999*y-38.0000003
sleq eq1,z,vz,1
print "x=";vz(0);" y=";vz(1)
```

Sortie (955 ms)

```
x= 2 y= 3
```

```
x= 50/13 y= -3
```

Système non linéaire

La résolution des systèmes d'équations non linéaires, par la méthode d'élimination, a également été programmée dans la bibliothèque MATH. Voici la résolution en x et y du système d'équations :

$$\begin{cases} x + y = 1 \\ x^3 + a^2y = a^2. \end{cases} \quad (3.14)$$

```
var eq(1),z(1)
eq(0)=x+y-1
eq(1)=x^3+a^2*(y-1)
z(0)=x
z(1)=y
c$=sgeq(1,1,eq,z)
```

Sortie (1870 ms)

```

3 cas pour x
Cas 1 pour x
x= 0
y= 1
Cas 2 pour x
x= a
y= -a +1
Cas 3 pour x
x= -a
y= a +1

```

Racine

La fonction `root` renvoie une racine k -ième exacte de p , si cette racine est rationnelle.

```

p=a^3 +6*a^2*b -3*a^2 +12*a*b^2 -12*a*b +3*a +8*b^3 -12
*b^2 +6*b -1
k=3
print root(p,k)

```

Sortie (145 ms)

```
[a +2*b -1]
```

Racines d'un polynôme

L'exemple suivant est dû à Wilkinson (1959) (voir aussi Davenport et al (1987) p117). Le polynôme $W = \text{ppwr}(x + 20, 20)$, de degré 20, possède les 20 racines réelles $x = -1, -2, \dots, -20$. Cependant, une petite perturbation (inférieure à 10^{-9} en valeur relative) sur le coefficient de x^{19} donne un polynôme f qui n'a plus que 10 racines réelles. Cette instabilité rend très difficile le calcul numérique des racines. La procédure `zerop` de la bibliothèque MATH est un programme qui détermine toutes les racines réelles d'un polynôme, avec la précision en cours, même lorsque les racines sont instables. La fonction `red` a pour effet de multiplier W par une constante, rendant ses coefficients entiers et de pgcd égal à 1.

```

s_pro 40000
f=red(ppwr(x+20,20)+2^-23*x^19)
zerop f,0

```

Sortie (483 s)

```

Zéros réels de f= 8388608*x^20 +1761607681*x^19 +172931153920*x^18 +1
0543221964800*x^17 +447347234439168*x^16 +14028108264898560*x^15 +336
985244869591040*x^14 +6342720331186176000*x^13 +94877480085669019648*
x^12 +1137370949952460554240*x^11 +10968398649699241820160*x^10 +8507
9777790228273561600*x^9 +528740774622641958944768*x^8 +26116558896927
86813829120*x^7 +10122095419974470210682880*x^6 +30198816984091441338
777600*x^5 +67426052557934862488567808*x^4 +1079691968105235458555904
00*x^3 +115794329499468438700032000*x^2 +73425049924762651852800000*x

```

```
+20408661249006627717120000
```

```
Nombre de zéros distincts= 10
```

```
comptés avec leur (multiplicité)= 10
```

```
(1) -1.0000000000~
```

```
(1) -0.2000000000~ E+1~
```

```
(1) -0.3000000000~ E+1~
```

```
(1) -0.4000000000~ E+1~
```

```
(1) -0.4999999928~ E+1~
```

```
(1) -0.6000006944~ E+1~
```

```
(1) -0.6999697234~ E+1~
```

```
(1) -0.8007267603~ E+1~
```

```
(1) -0.8917250249~ E+1~
```

```
(1) -0.2084690810~ E+2~
```

Dérivation

Fractions rationnelles

La fonction `der(p, x)` dérive p suivant x . L'instruction suivante dérive $(ax + b)/(cx + d)^2$ par rapport à x .

```
print der((a*x+b)*(c*x+d)^-2,x)
```

Sortie (135 ms)

```
- [x*c +d]^-3* [a*x*c -a*d +2*b*c]
```

Le programme montre que l'équation de la tangente au point $x = 1$ de l'hyperbole $y = f(x) = 1/(x + 1)$ est :

$$y + x/4 = 3/4. \quad (3.15)$$

La commande `lit` fixe l'ordre des littéraux pour améliorer la sortie.

```
lit y
```

```
f=1/(1+x)
```

```
w=(y-subs(f,x=1))-subs(der(f,x),x=1)*(x-1)
```

```
print w
```

Sortie (100 ms)

```
y +1/4*x -3/4
```

Expressions trigonométriques

Nous avons vu plus haut comment le Basic 1000d peut être facilement employé pour traiter les nombres algébriques, par exemple $\sqrt{3}$, de façon exacte. Dans cette section, nous allons voir que le Basic 1000d peut également être

programmé pour dériver exactement des expressions mathématiques faisant intervenir les fonctions trigonométriques `sin`, `cos`, `tg` et `cotg`, comme par exemple $\sin(2+x) + \cos(\cos(x+2))$

Des programmes effectuant le traitement de telles expressions se trouvent dans la bibliothèque MATH. Le principe de la méthode utilisée est le suivant. Nous remplaçons l'expression par un ensemble de fractions rationnelles, permettant de la décrire, en introduisant des littéraux supplémentaires qui représentent des `sin` et `cos`. Ces littéraux ont pour noms `trigo_l(0,i)` pour les sinus et `trigo_l(1,i)` pour les cosinus. Détaillons l'exemple de l'expression ci-dessus. Nous posons :

```
e1=x+2
e2=trigo_l(1,0)
e3=trigo_l(0,0)+trigo_l(1,1)
```

L'expression de départ est entièrement décrite par ces trois expressions rationnelles, car on peut la retrouver comme étant égale à `e3`, après les substitutions suivantes :

```
trigo_l(1,1) → cos(e2)
trigo_l(0,0) → sin(e1)
trigo_l(1,0) → cos(e1)
```

Dans les programmes de la bibliothèque MATH, l'expression est codée par le `v_ensemble` `vset$(e1, e2, e3)` et ce codage est effectué, à partir d'une chaîne qui contient l'écriture usuelle, par la fonction `trigox`. La fonction `trigop` effectue la transformation inverse. La fonction `dertrigo` dérive, par rapport au littéral `x`, l'expression trigonométrique codée par le `v_ensemble` `t`. Elle renvoie le `v_ensemble` qui code la dérivée.

Dans l'exemple suivant, le programme affiche la dérivée de $\sin(2+x) + \cos(\cos(x+2))$.

```
c$="sin(2+x)+cos(cos(x+2))"
c$=trigox(c$)
print "La dérivée en x de ";trigop(c$)
print "est ";trigop(dertrigo(c$,x))
```

Sortie (985 ms)

```
La dérivée en x de cos( cos( x +2 ) ) +sin( x +2 )
est sin( cos( x +2 ) )*sin( x +2 ) +cos( x +2 )
```

Intégration

Par intg

La fonction interne `intg(p, x)` intègre la fraction rationnelle p suivant x . La fonction `intg` a besoin des pôles de p . Si les pôles ne sont pas rationnels, elle ne peut pas calculer l'intégrale et sort en erreur. Dans ce cas, on utilisera la bibliothèque MATH qui contient des procédures permettant d'intégrer toutes les fractions rationnelles sans restriction.

Le programme détermine l'aire s limitée par la courbe $y = f(x) = (b - x)^3(x - a)$ et l'axe des x :

$$s = \int_a^b f(x) dx. \quad (3.16)$$

Le résultat montre que $s = (b - a)^5/20$.

```
f=(b-x)^3*(x-a)
w=intg(f,x)
s=subs(w,x=b)-subs(w,x=a)
print formf(s)
```

Sortie (385 ms)

```
1/20* [b -a]^5
```

Par intg1

La procédure `intg1` de la bibliothèque MATH calcule une intégrale de toute fraction rationnelle. Le programme suivant montre que :

$$\int \frac{4x^3 - 30x^2 + 70x - 50}{x^4 - 10x^3 + 35x^2 - 50x + 24} dx = \log(x^4 - 10x^3 + 35x^2 - 50x + 24). \quad (3.17)$$

```
f=(4*x^3 -30*x^2 +70*x -50)/(x^4 -10*x^3 +35*x^2 -50*x
+24)
intg1 f,x
```

Sortie (2835 ms)

La partie logarithmique de l'intégrale est le produit de

2

et de

```
1/2 * log( x^4 -10*x^3 +35*x^2 -50*x +24 )
```

Le calcul suivant donne :

$$\int \frac{4x^3 - 30x^2 + 70x - 49}{x^4 - 10x^3 + 35x^2 - 50x + 24} dx = \frac{5}{6} \log(x - 1) + \frac{3}{2} \log(x - 2) + \frac{1}{2} \log(x - 3) + \frac{7}{6} \log(x - 4). \quad (3.18)$$

```
f=(4*x^3 -30*x^2 +70*x -49)/(x^4 -10*x^3 +35*x^2 -50*x
+24)
intg1 f,x
```

Sortie (4075 ms)

La partie logarithmique de l'intégrale est la somme de

```
7/6 * log( x -4 )
```

```

et de
5/6 * log( x -1 )
et de
3/2 * log( x -2 )
et de
1/2 * log( x -3 )

```

Comme le dénominateur des expressions à intégrer ci-dessus a toutes ses racines rationnelles, la fonction interne `intg` convient également.

```

f=(4*x^3 -30*x^2 +70*x -50)/(x^4 -10*x^3 +35*x^2 -50*x
+24)
print intg(f)

```

Sortie (1680 ms)

```

[log(x-4) +log(x-3) +log(x-2) +log(x-1)]
f=(4*x^3 -30*x^2 +70*x -49)/(x^4 -10*x^3 +35*x^2 -50*x
+24)
print intg(f)

```

Sortie (1725 ms)

```

1/6* [7*log(x-4) +3*log(x-3) +9*log(x-2) +5*log(x-1)]

```

Par contre, l'exemple suivant ne peut être traité par `intg`, le dénominateur ne se factorisant pas rationnellement. On obtient :

$$\int \frac{2x+17}{(x^2+17x+2)^2} dx = -\frac{1}{x^2+17x+2}. \quad (3.19)$$

```

f=(2*x+17)/(x^2+17*x+2)^2
intg1 f,x

```

Sortie (1365 ms)

```

La partie rationnelle de l'intégrale est
- [x^2 +17*x +2]^-1

```

Sommutation en termes finis

La fonction `dsum(f, x, a, b)` permet de sommer sur l'entier $x \in [a, b]$ (a et b désignent des entiers $a < b$) certaines expressions rationnelles f . Comme a et b peuvent contenir des littéraux, cette fonction est très différente de la sommation explicite `sum(x = a, b of f)`. Le calcul suivant montre que la somme des carrés des entiers de 1 à n est $n^3/3 + n^2/2 + n/6$.

```

print dsum(x^2,x,1,n)

```

Sortie (560 ms)

```

1/3*n^3 +1/2*n^2 +1/6*n

```

Ci-dessous, la fonction `dsum` montre que :

$$\sum_{x=1}^n \frac{1}{x(x+1)} = \frac{n}{n+1}. \quad (3.20)$$

```
print dsum(1/x/(x+1),x,1,n)
```

Sortie (580 ms)

```
[n]* [n +1]^-1
```

Le calcul suivant montre la formule :

$$2(1 + 2 + \cdots + n)^4 = (1^5 + 1^7) + (2^5 + 2^7) + \cdots + (n^5 + n^7). \quad (3.21)$$

```
print 2*dsum(x,x,1,n)^4-dsum(x^5+x^7,x,1,n)
```

Sortie (2845 ms)

```
0
```

Calculs modulaires

Nous avons déjà vu que le Basic 1000d possède des fonctions manipulant les nombres modulaires. On dispose de quelques autres fonctions très performantes, agissant sur les polynômes modulaires unilittéraux. La fonction `mdff` factorise complètement dans $\mathbf{Z}_p[x]$ le polynôme $y = x^5 + 17x^4 + 11$ pour le nombre premier $p = 23$. Elle donne un seul facteur, ce qui prouve que y est irréductible. La fonction `mdpwr` calcule la puissance n -ième ($n = -10^{50}$) du polynôme $A = 3x^3 + 5$ modulo le polynôme y et modulo le nombre premier p .

```
y=x^5 +17*x^4 +11
```

```
p=23
```

```
print "y=";mdff(y,p)
```

```
A=3*x^3+5
```

```
n=-10^50
```

```
print "A^n=";mdpwr(A,n,y,p)
```

Sortie (5935 ms)

```
y= [x^5 +17*x^4 +11]
```

```
A^n= 13*x^4 +19*x^3 +4*x^2 +10*x +17
```

Géométrie plane

Diverses fonctions et procédures de la bibliothèque MATH permettent de traiter algébriquement les problèmes élémentaires de géométrie plane. Comme exemple, le programme suivant montre que les médiatrices d'un triangle ABC sont concourantes. Les littéraux **ax** et **ay** représentent les coordonnées du sommet A. De même les coordonnées de B et C, qui peuvent être arbitraires, sont représentées par les littéraux **bx**, **by**, **cx** et **cz**. La procédure **mediatrice** détermine l'équation de la médiatrice d'un côté. Ainsi son premier appel détermine des valeurs x_1 , y_1 et z_1 telles que l'équation de la médiatrice de BC soit donnée par $x_1x + y_1y + z_1 = 0$. Les procédures utilisées pour ce calcul sont **milieu** qui détermine le milieu M (x_0/z_0 , y_0/z_0) du côté, **perpinf** qui détermine la pente y_1/x_1 de la médiatrice, et **droite** qui détermine son équation. La preuve cherchée est apportée par la fonction **aligne** qui prend une valeur nulle si et seulement si les droites données en entrée sont concourantes.

```

var x1,y1,z1,x2,y2,z2,x3,y3,z3
mediatrice bx,by,cx,cy,x1,y1,z1
mediatrice cx,cy,ax,ay,x2,y2,z2
mediatrice ax,ay,bx,by,x3,y3,z3
print aligne(x1,y1,z1,x2,y2,z2,x3,y3,z3)
stop

mediatrice:procedure(ax,ay,bx,by access x0,y0,z0)
  local var x1,y1,z1
  milieu ax,ay,1,bx,by,1,x0,y0,z0
  perpinf ax,ay,1,bx,by,1,x1,y1,z1
  droite x0,y0,z0,x1,y1,z1,x0,y0,z0
  return

```

Sortie (575 ms)

0

Approximation polynomiale

La fonction **polyappr** de la bibliothèque MATH renvoie un polynôme $w(x)$ approchant une fonction $f(x)$ sur $[0, 1]$. Considérons le problème du calcul de la fonction $y = e^t$. On désire effectuer ce calcul le plus rapidement possible, à une précision donnée, et en utilisant seulement des additions, soustractions, multiplications et divisions. La méthode habituelle utilise un polynôme qui soit une approximation de 2^x sur $[0, 1]$. On obtient ensuite y par $y = 2^x 2^p$ où $x + p = t/\log 2$, avec p entier et $x \in [0, 1]$. Le programme suivant détermine une

approximation polynomiale $w(x)$ de $\text{XXX}(x) = 2^x$ ($x \in [0, 1]$) de degré 15, puis l'erreur est déterminée par calcul de $|2^x - w(x)|$ pour des valeurs x aléatoires (arrêt par Break). Le fait que l'erreur ainsi obtenue (inférieure à 10^{-25}) soit peu différente de la valeur R renvoyée par la fonction `polyappr` indique que le polynôme $w(x)$ est voisin du meilleur polynôme possible. L'erreur sur e^t , pour $t \in [0, \log 2]$, est ainsi nettement plus faible que celle ($\approx (\log 2)^{16}/16! \approx 10^{-16}$) que donne le développement limité d'ordre 15 en $t = 0$ de e^t .

```
precision 30
L2=log(2)
w=polyappr(XXX,15,x,R)
print "timer=";timer
print "Polynôme approché à";R;" près"
print "w=";w
print "Détermination aléatoire de l'erreur"
np=0
do
  S=rnd
  S=abs(XXX(S)-fsubs(w,x=S))
  np=np+1
  if S>R
    R=S
    print using "Erreur >~ ##.####~^^^~^_ (Vérifié en #
      points)";R;np
  endif
loop
stop
XXX:function
  value=exp(L2*@1)
  return
```

```
timer= 10361
```

```
Polynôme approché à 0.895241081287921394617307114814~ E-25 près
```

```
w= 197/44381907650252309*x^15 +2535/40387356220308247*x^14 +30289/219
42547252541985*x^13 +738403/28774344051141536*x^12 +7829652/176117027
51783717*x^11 +514352761/72907177697448725*x^10 +357771966/3515120043
787111*x^9 +31991292311/24207426398220468*x^8 +382862472680/251012361
17920799*x^7 +2432843621904/15794065125174103*x^6 +23288561329795/174
66126501299056*x^5 +63355514660297/6587093389092430*x^4 +849780496490
636/15310226881081789*x^3 +6247250157603125/26005665389235078*x^2 +23
284906736464174/33593019476258881*x +11170175508047164213639311/11170
175508047164213639312
```

```
Détermination aléatoire de l'erreur
```

```
Erreur >~ 0.8954~ E-25 (Vérifié en 1243 points)
```

```
Erreur >~ 0.8954~ E-25 (Vérifié en 3316 points)
```

4

Quelques problèmes arithmétiques



Le Basic 1000d peut effectuer facilement les calculs arithmétiques sur les grands entiers et les nombres modulaires. Dans ce chapitre nous considérons quelques exercices élémentaires de la théorie des nombres et nous décrivons en détail les procédures `euler_phi`, `chinoiseq`, `legendre` et `prsq` de la bibliothèque MATH. Pour conclure le chapitre, nous donnons des programmes effectuant la décomposition en facteurs premiers des entiers de Gauss. Le problème de la factorisation des grands nombres en facteurs premiers sera examiné au chapitre suivant. Pour un exposé introductif des questions traitées dans ce chapitre nous vous conseillons le livre d'Apostol.

L'indicateur d'Euler

L'indicateur d'Euler $\varphi(n)$ est défini pour les entiers $n \geq 1$ comme étant le nombre d'entiers naturels n' inférieurs à n et premiers avec n . La fonction `euler_phi(n)` calcule $\varphi(n)$ en utilisant les propriétés suivantes de l'indicateur d'Euler :

$$\begin{aligned}\varphi(p^\alpha) &= p^\alpha - p^{\alpha-1}, & \text{pour } p \text{ premier et } \alpha \geq 1, \\ \varphi(mn) &= \varphi(m)\varphi(n), & \text{pour } m \text{ et } n \text{ premiers entre-eux.}\end{aligned}\tag{4.1}$$

La fonction $\varphi(n)$ est une fonction multiplicative (c'est ce qu'exprime la deuxième équation ci-dessus), qui peut être calculée à partir de la factorisation de n en nombres premiers. Dans `euler_phi`, le nombre n est factorisé par `prfact`. La fonction n'est donc utilisable en pratique que pour des valeurs de n assez petites (par exemple le calcul de $\varphi(100003^2)$ demande 16 secondes).

```
euler_phi:function(n)
  value=1
  ift n=1 return
  local var u,w,p index i,k
  w=prfact(n)
  for i=1,polymn(w)
    u=polym(w,i)
    p=norm(u)
    k=deg(u)-1
    vmul value,p^k*(p-1)
  next i
  return
```


Exemple

Le nombre d'entiers i ($1 \leq i \leq 100$) premiers avec 100 est obtenu directement en comptant le nombre de fois que le pgcd `gcdr(i, 100)` prend la valeur 1. Ce nombre est donné plus rapidement par `euler_phi(100) = $\varphi(100)$` .

```
print -sum(i=1,100 of (gcdr(i,100)=1));
print euler_phi(100)
```

Sortie (590 ms)

40 40

Eléments primitifs

L'ensemble des résidus modulo n , premiers avec n , muni de la loi de multiplication modulo n , forme un groupe commutatif, noté G_n . Le nombre d'éléments du groupe G_n est $\varphi(n)$, par définition de l'indicateur d'Euler. L'ordre d'un élément $a \in G_n$ est le plus petit exposant $e > 0$ tel que $a^e \equiv 1 \pmod{n}$. Le théorème de Lagrange, qui indique que l'ordre de tout élément d'un groupe fini divise l'ordre du groupe, nous donne $e|\varphi(n)$. Ce résultat donne immédiatement le théorème d'Euler,

$$a^{\varphi(n)} \equiv 1 \pmod{n}, \quad \text{où } (a, n) = 1, \quad (4.2)$$

et le théorème de Fermat ($\varphi(n) = n - 1$ si n est un nombre premier) :

$$a^{n-1} \equiv 1 \pmod{n}, \quad n \text{ premier et } (a, n) = 1. \quad (4.3)$$

On dit que $g \in G_n$ est un élément primitif si l'ordre de g est $\varphi(n)$. Le groupe G_n , identique à l'ensemble $\{1, g, g^2, \dots, g^{\varphi(n)-1}\}$, est alors un groupe cyclique. Pour $n > 2$, un élément primitif g est caractérisé par le fait que $g^{\varphi(n)/f} \not\equiv 1 \pmod{n}$ pour tout diviseur premier f de $\varphi(n)$. En utilisant cette caractérisation, la fonction `primitif(n)` détermine le plus petit élément primitif $g \geq 1$ du groupe G_n , s'il existe, et renvoie -1 sinon (Gauss a montré que g existe si et seulement si

$$n = 1, 2, 4, q^k \text{ ou } 2q^k, \quad (4.4)$$

où $q \neq 2$ est un nombre premier et $k \geq 1$). Pour $g = 2, 3, \dots$, on calcule $g^{m/f} \pmod{n}$ par `mdpwre`, les facteurs premiers f de $m = \varphi(n)$ étant déterminés à partir de la forme factorisée `prfact(m)`. Si $g^{m/f} \not\equiv 1 \pmod{n}$ pour tous les facteurs f , alors g est le plus petit élément primitif; sinon, on étudie la valeur suivante de g , première avec n , sans dépasser $n - 1$. Les cas $n = 1$ ou 2 sont traités à part. De plus, la variable `primitif10` est mise égale à -1 si 10 est un élément primitif modulo n , et mise égale à zéro sinon. Le programme détermine le plus petit élément primitif des 50 premiers nombres premiers.

Les astérisques indiquent que 10 est un élément primitif; elles correspondent à la propriété suivante. Soit p un nombre premier $p \neq 2, 5$. L'écriture décimale de $1/p$ est périodique, de période e , où e est le plus petit exposant $e > 0$ tel que $10^e \equiv 1 \pmod{p}$ (e est donc l'ordre de l'élément 10 du groupe G_p). Lorsque 10 est un élément primitif modulo p , alors cette période vaut $p - 1$. Par exemple dans $1/7 = 0.142857\ 142857\dots$ les décimales se répètent

tous les 6 chiffres. Par contre, lorsque 10 n'est pas un élément primitif modulo p , la période des décimales de $1/p$ est un diviseur propre de $p - 1$. Par exemple $1/11 = 0.09\ 09\ 09\ \dots$

```

n=2
for j=1,50,10
  print "p";
  c$="g"
  for j1=1,10
    print justr$(n,5);
    c1$=justl$(primitif(n))
    ift primitif10 c1$="*"&c1$
    c$=c$&justr$(c1$,5)
    n=prime(n+1)
  next j1
  print
  print c$
  print
next j
stop

primitif:function(n)
  primitif10=0
  local var m,g,w,f
  local index i
  if n<3
    value=n-1
    return
  endif
  value=-1
  m=euler_phi(n)
  w=prfact(m)
  g=2
  while g<n
    if gcdr(g,n)=1
      for i=1,polymn(w)
        f=norm(polym(w,i))
        ift mdpwre(g,m/f,n)=1 goto primitif_1
      next i
      value=g
      ift g>10 return
      ift gcdr(10,n)<>1 return
      for i=1,polymn(w)
        f=norm(polym(w,i))
        ift mdpwre(10,m/f,n)=1 return
      next i
    
```

```

        primitif10=-1
        return
    endif
primitif_1:g=g+1
wend
return

```

Sortie (10 s)

p	2	3	5	7	11	13	17	19	23	29
g	1	2	2	*3	2	2	*3	*2	*5	*2

p	31	37	41	43	47	53	59	61	67	71
g	3	2	6	3	*5	2	*2	*2	2	7

p	73	79	83	89	97	101	103	107	109	113
g	5	3	2	3	*5	2	5	2	*6	*3

p	127	131	137	139	149	151	157	163	167	173
g	3	*2	3	2	*2	6	5	2	*5	2

p	179	181	191	193	197	199	211	223	227	229
g	*2	*2	19	*5	2	3	2	*3	2	*6

Fonctions multiplicatives

La méthode de calcul utilisée dans `euler_phi` pour l'indicateur d'Euler s'applique, de façon plus générale, aux fonctions arithmétiques multiplicatives. La fonction `sigma(n, a)` détermine ainsi la somme des puissances a ième des diviseurs d de n :

$$\sigma_a(n) = \sum_{d|n} d^a, \quad (4.5)$$

à partir de la factorisation en nombres premiers de n et de :

$$\sigma_a(p^k) = 1 + p^a + \dots + p^{ak} = \begin{cases} (p^{a(k+1)} - 1)/(p^a - 1), & \text{si } a \neq 0, \\ a + 1, & \text{si } a = 0, \end{cases} \quad (4.6)$$

pour un nombre premier p et $k \geq 1$. Par exemple, pour $n = 6$, dont les quatre diviseurs sont 1, 2, 3 et 6, le nombre des diviseurs est donné par `sigma(6,0)` et leur somme $1 + 2 + 3 + 6 = 12$ par `sigma(6,1)` (6 est un nombre parfait, c'est-à-dire égal à la moitié de la somme de ses diviseurs). Le programme calcule ces valeurs, ainsi que $\sigma_{-2}(6)$, $\sigma_{-1}(6)$ et $\sigma_2(6)$. On calcule ensuite le nombre de diviseurs de $n = 2^{16}3^{12}5^27^2$ (quel est le plus petit entier n dont le nombre de diviseurs $\sigma_0(n)$ est 1990 ?). Le nombre 66 est un exemple de nombre dont la somme des diviseurs est un carré, et le nombre $(3 \times 11 \times 31 \times 443 \times 499)^3$ est un cube dont la somme des diviseurs est un carré.

```

print sigma(6,-2);sigma(6,-1);sigma(6,0);sigma(6,1);sig
ma(6,2)

```

```

print sigma(42664933785600,0)
print sigma(66,1)
print prfact$(sigma((3*11*31*443*499)^3,1))
stop
sigma:function(n,a)
  value=1
  ift n=1 return
  local var u,w,p index i,k
  w=prfact(n)
  for i=1,polymn(w)
    u=polym(w,i)
    p=norm(u)
    k=deg(u)
    if a
      p=p^a
      vmul value,(p^(k+1)-1)/(p-1)
    else
      vmul value,k+1
    endif
  next i
  return

```

Sortie (990 ms)

25/18 2 4 12 50

1989

144

$2^{18} * 3^2 * 5^8 * 13^2 * 37^2 * 61^2 * 157^2$

Exercice 4.1. Nombre de fractions

Quel est le nombre de fractions réduites $p/q \in (0, 1)$ ayant leur dénominateur $q \leq 100$?

Exercice 4.2. Suite aliquote

Soit $A(n) = \sigma_1(n) - n$ la somme des parties aliquotes de n (c'est-à-dire la somme des diviseurs autres que n). On appelle *suite aliquote* une suite définie par $n_{i+1} = A(n_i)$ en partant d'un entier n_1 . Par exemple, partant de $n_1 = 12$, n_2 est la somme des parties aliquotes de 12, $n_2 = 1 + 2 + 3 + 4 + 6 = 16$, n_3 est la somme des parties aliquotes de 16, $n_3 = 1 + 2 + 4 + 8 = 15$, etc., ce qui donne la suite 12, 16, 15, 9, 4, 3, 1. Souvent, les suites aliquotes se terminent ainsi sur 1. Mais parfois la suite génère un cycle de nombres qui se répètent indéfiniment. Ainsi, partant d'un nombre parfait, par exemple $n_1 = 6$, la suite donne uniquement des 6. Lorsque le cycle est de longueur 2, on a deux nombres n et n' dont la somme des parties aliquotes de l'un est égale à l'autre. On dit alors que les nombres n et n' sont amiables. Par exemple $n = 111448537712$ et $n' = 118853793424$ sont amiables. De façon plus générale, si le cycle est de longueur k , on dit que l'on a un cycle amiable de longueur k . Un exemple

merveilleux est le nombre 14316 qui donne un cycle amiable de longueur 28. Comme exercice, nous vous proposons d'écrire un programme qui recherche les cycles amiables.

Congruences linéaires

Si A et B sont des classes de résidus modulo N données, et x une classe inconnue, la congruence du premier degré

$$Ax \equiv B \pmod{N} \quad (4.7)$$

est équivalente à l'équation diophantienne

$$Ax - Ny = B. \quad (4.8)$$

Cette équation a des solutions entières en x et y si et seulement si le pgcd D de A et N divise B . Dans ce cas, l'équation (4.7) peut se récrire après division par D , en posant $a = A/D$, $b = B/D$ et $n = N/D$:

$$ax \equiv b \pmod{n}, \quad \text{où } a \text{ et } n \text{ sont premiers entre-eux.} \quad (4.9)$$

En Basic 1000d, la solution de l'équation (4.9), $x \equiv a^{-1}b \pmod{n}$, s'obtient par `prinv` ou `mdpwe`. Soit par exemple la congruence $19x \equiv 5 \pmod{140}$. Le programme suivant détermine $x \equiv 15 \pmod{140}$, puis vérifie la congruence de départ.

```
x=modr(prinv(19,140)*5,140)
print x;modr(19*x-5,140)
```

Sortie (25 ms)

```
15 0
```

Si on essaie de résoudre $2x \equiv 1 \pmod{4}$, qui n'a pas de solution, l'absence de solution apparaît avec la valeur 0 renvoyée par fonction `prinv`, ou avec une sortie erreur dans `mdpwe`.

```
print prinv(2,4)
print mdpwe(2,-1,4)
```

Sortie (10 ms)

```
0
```

```
*ERREUR* DIVISION
print mdpwe(2,-1,4)
2.print mdpwe(2,-1,4)
```

Le calcul de l'inverse $a^{-1} \pmod{p}$ par les fonctions `prinv` et `mdpwe` utilise la méthode de l'algorithme d'Euclide étendu (voir par exemple Knuth). Lorsque p est premier, l'inverse $a^{-1} \pmod{p}$ peut aussi être calculé par la

relation équivalente au théorème de Fermat (4.3) : $a^{-1} \equiv a^{p-2} \pmod{p}$. Le programme suivant, qui calcule l'inverse de 2 $\pmod{568945241}$ par les deux méthodes, montre que l'algorithme d'Euclide est plus rapide.

```
p=568945241
a=2
clear timer
print prinv(a,p);mtimer
clear timer
print mdpwre(a,p-2,p);mtimer
284472621 20
284472621 70
```

Le théorème des restes chinois

Nous considérons la généralisation de l'équation (4.9) au système de congruences :

$$\begin{aligned} c_1 x &\equiv b_1 \pmod{m_1} \\ c_2 x &\equiv b_2 \pmod{m_2} \\ &\dots \\ c_r x &\equiv b_r \pmod{m_r}, \end{aligned} \tag{4.10}$$

où les modules m_i sont premiers deux à deux ($(m_i, m_j) = 1$ pour $i \neq j$) et où le coefficient c_i de x doit être inversible modulo m_i ($(c_i, m_i) = 1$). Le théorème des restes chinois indique que, dans ces conditions, il existe alors une solution unique du système (4.10) modulo $P = m_1 \times m_2 \times \dots \times m_r$.

On détermine cette solution de la façon suivante. Puisque c_i est inversible modulo m_i , le système (4.10) équivaut à :

$$\begin{aligned} x &\equiv a_1 \pmod{m_1} \\ x &\equiv a_2 \pmod{m_2} \\ &\dots \\ x &\equiv a_r \pmod{m_r} \end{aligned} \tag{4.11}$$

où $a_i = c_i^{-1} b_i$. Posons $M_i = P/m_i$. Alors $\text{gcd}(M_i, m_i) = 1$, et M_i possède un inverse M'_i modulo m_i . Les nombres

$$N_i = M_i M'_i \tag{4.12}$$

vérifient $N_i \equiv 1 \pmod{m_i}$ et $N_i \equiv 0 \pmod{m_j}$ si $j \neq i$. Le nombre

$$x = a_1 N_1 + a_2 N_2 + \dots + a_r N_r \tag{4.13}$$

vérifie donc les congruences (4.11) et (4.10).

La solution $x \in [0, P)$ du système (4.10) est déterminée par la fonction `chinoisseq`. Les données sont dans l'ordre les nombres $c_1, b_1, m_1, c_2, b_2, m_2, \dots, c_r, b_r, m_r$. Par exemple, le programme suivant montre qu'une solution des congruences $7x \equiv 4 \pmod{211}$ et $17x \equiv 20 \pmod{49}$ est $x = 1990$.

```
print chinoisq(7,4,211,17,20,49)
```

Sortie (260 ms)

1990

Les $3r$ arguments de `chinoisq` sont placés dans le tableau de type `var` $t(2, r)$, dans l'ordre inverse (l'ordre des arguments est $t(2, r)$, $t(1, r)$, $t(0, r)$, $t(2, r - 1)$, $t(1, r - 1)$, $t(0, r - 1)$, ..., $t(0, 0)$). Le système (4.10) est alors remplacé par le système (4.11), où les nombres a_i (respectivement m_i) sont codés sous la forme du `v_ensemble` a (respectivement m) : $a = \text{vset}\$(a_1, a_2, \dots, a_r)$ et $m = \text{vset}\$(m_1, m_2, \dots, m_r)$. La solution de (4.11) est alors déterminée par la fonction `chinois(m, a)`. Dans ce calcul, la procédure `chinois1` détermine le `v_ensemble` $N = \text{vset}\$(N_1, N_2, \dots, N_r)$ formé des nombres N_i donnés par l'équation (4.12) et la fonction `chinois2` met la solution (4.13) pour x .

```
chinois1:procedure(char m access N,P)
  local datai elementn(m) index n,i var mi,MI
  P=prod(i=1,n of elementv(m,i))
  N=
  for i=1,n
    mi=elementv(m,i)
    MI=P/mi
    MI=MI*prinv(MI,mi)
    ift MI=0 ERREUR mi non premiers entre-eux
    cadd N,vset$(MI)
  next i
  return
chinois2:function
  local index i
  value=modr(sum(i=1,elementn(m) of elementv(a,i)*element
    v(N,i)),P)
  return
chinois:function(char m,a)
  local char N var P
  chinois1 m,N,P
  value=chinois2
  return
chinoisq:function(t(2,@0/3-1))
  local datai @0/3-1 char m,a index i
  for i=i,0
    cadd m,vset$(t(0,i))
    cadd a,vset$(modr(prinv(t(2,i),t(0,i))*t(1,i),t(0,i))
      )
  next i
  value=chinois(m,a)
  return
```

Notons que lorsqu'on a à résoudre plusieurs problèmes de la forme (4.11), avec les mêmes modules m_i , il est préférable de déterminer une fois pour toutes les nombres N_i . Cela s'applique au problème du comput, dans lequel on cherche le quantième de l'année pour laquelle l'indiction a_1 , le nombre d'or a_2 et le cycle solaire a_3 sont donnés. Ici $m_1 = 15$, $m_2 = 19$ et $m_3 = 28$, et l'année cherchée A est le résidu minimum modulo $P = 7980$ de

$$A \equiv 6916a_1 + 4200a_2 + 4845a_3 \pmod{7980}. \quad (4.14)$$

Le programme suivant montre comment la procédure `chinois1` permet d'obtenir l'équation (4.14).

```
char N
var P
chinois1 vset$(15,19,28),N,P
print elementv(N,1);elementv(N,2);elementv(N,3);P
```

Sortie (210 ms)

```
6916 4200 4845 7980
```

Exercice 4.3. Réduire 48828/89077

Décomposer la fraction 48828/89077 en somme de deux autres dont les dénominateurs sont 317 et 281.

Le symbole de Legendre

Si p et q sont des entiers premiers entre-eux, et si la congruence

$$x^2 \equiv p \pmod{q} \quad (4.15)$$

a des solutions x (resp. n'a pas de solution), alors on dit que p est un résidu (resp. un non-résidu) quadratique de q . Lorsque q est un nombre premier $q > 2$, le symbole de Legendre $(p|q)$ est défini par :

$$(p|q) = \begin{cases} 1, & \text{si } p \text{ est un résidu quadratique de } q; \\ -1, & \text{si } p \text{ est un non-résidu quadratique de } q; \\ 0, & \text{si } p \text{ est un multiple de } q. \end{cases} \quad (4.16)$$

Il est possible de calculer sa valeur par la formule d'Euler :

$$(p|q) \equiv p^{(q-1)/2} \pmod{q}, \quad (4.17)$$

mais une méthode plus efficace utilise une extension due à Jacobi. Soit n un entier impair factorisé en nombres premiers par $n = \pm \prod q_i^{\alpha_i}$. Le symbole de

Jacobi $(p|n)$ est défini en fonction des symboles de Legendre $(p|q_i)$ pour tout entier p par

$$(p|n) = \prod (p|q_i)^{\alpha_i}. \quad (4.18)$$

On pose aussi $(p|\pm 1) = 1$.

La fonction **legendre**(p, q) calcule le symbole de Jacobi en utilisant les propriétés suivantes de ce symbole :

$$\begin{aligned} (p|q) &= (\text{modr}(p, q)|q), \\ (mn|q) &= (m|q)(n|q), \\ (0|q) &= 0, \\ (1|q) &= 1, \\ (2|q) &= (-1)^{(q^2-1)/8}, \\ (p|q) &= (-1)^{(p-1)(q-1)/4}(q|p), \quad p \text{ et } q \text{ impairs tels que } (p, q) = 1. \end{aligned} \quad (4.19)$$

On remplace le symbole $(p|q)$ à calculer par un autre $(p|q)$ plus simple, et on itère (boucle **do ... loop**) jusqu'à ce que $p = 1$ ou $p = 0$. La deuxième équation (loi de multiplicativité complète) est utilisée pour extraire un facteur 2^α de p avant d'utiliser la dernière équation (loi de réciprocité quadratique).

```

legendre:function(p,q)
    local var v,w
    ift integerp(p) ift integerp(q) v=odd(q)
    if v=0
        print "*ERREUR* Symbole de Jacobi (" ;p;" |";q;) non
            défini"
        stop
    endif
    value=1
    do
        p=modr(p,q)
        ift p=1 return
        if p=0
            value=0
            return
        endif
        v=prfact(p,2)
        w=polym(v,1)
        if norm(w)=2
            ift odd(deg(w)) ift odd((q^2-1)/8) value=-value
            ift polymn(v)=1 return
            p=norm(polym(v,2))
        endif
    endif
    exg p,q

```

```

      ift modr(p,4)=3 ift modr(q,4)=3 value=-value
loop

```

Exemple

Lorsque q n'est pas un nombre premier, si p est un résidu quadratique, alors $(p|q) = 1$, mais la réciproque est fausse, comme on peut le voir à l'aide du programme suivant. On détermine directement tous les résidus quadratiques de $q = 15$, par le calcul de x^2 pour $x = 1, \dots, q$. En notant les résidus trouvés à l'aide du tableau **res**, on peut afficher la table qui indique si p est un résidu (R) ou non-résidu (N) quadratique (on affiche X si p n'est pas premier avec q). En comparant avec les valeurs du symbole de Jacobi $(p|q)$, on remarque que $(2|15) = (8|15) = 1$ bien que ni 2 ni 8 ne soient des résidus quadratiques de 15. Par contre, pour q premier, N est toujours associé à $(p|q) = -1$ dans la table affichée par ce programme (il suffit de changer la valeur de **q** dans la première ligne pour faire un essai).

```

q=15
index res(q)
for x=1,q
  res(modr(x^2,q))=1
next x
print "p          ";conc$(p=1,q-1 of justr$(p,3))
print "résidu?";
for p=1,q-1
  if gcdr(p,q)=1
    if res(p)
      print "  R";
    else
      print "  N";
    endif
  else
    print "  X";
  endif
next p
print
print "(p|q)  ";conc$(p=1,q-1 of justr$(legendre(p,q),3
))

```

Sortie (1515 ms)

p	1	2	3	4	5	6	7	8	9	10	11	12	13	14
résidu?	R	N	X	R	X	X	N	N	X	X	N	X	N	N
(p q)	1	1	0	1	0	0	-1	1	0	0	-1	0	-1	-1

La fonction `mdff` qui factorise le polynôme $P(x)$ modulo le nombre premier p permet de résoudre les congruences non linéaires de la forme $P(x) \equiv 0 \pmod{p}$. En effet, à chaque solution correspond un facteur de degré 1 dans la forme factorisée. Soit par exemple à résoudre l'équation :

$$3x^{19} \equiv -5 \pmod{17}. \quad (4.20)$$

Afin de ne pas entreprendre une factorisation inutilement compliquée, nous réduisons le degré de l'équation par le théorème de Fermat (4.3). La congruence (4.20) se simplifie en $3x^3 \equiv -5 \pmod{17}$. La factorisation du polynôme de degré 3 est alors effectuée en 110 ms, tandis que la factorisation du polynôme de degré 19 aurait pris 77 secondes. La forme obtenue montre qu'il y a une seule solution, $x \equiv -4 \pmod{17}$.

```
print mdff(3*x^3+5,17)
```

Sortie (110 ms)

```
3* [x +4]* [x^2 +13*x +16]
```

De façon encore plus efficace, on peut d'abord réduire la congruence $P(x) \equiv 0 \pmod{p}$ à une équation équivalente $Q(x) \equiv 0 \pmod{p}$ où $Q(x)$ se décompose complètement en facteurs linéaires. En effet, comme le polynôme $R(x) = x^p - x$ est, pour p premier, le produit modulo p de tous les facteurs linéaires de la forme $x - a$ (pour $a = 0, 1, \dots, p-1$), on peut prendre pour $Q(x)$ le pgcd de $P(x)$ et $R(x)$. En Basic 1000d, ce pgcd est donné par la fonction `mdgcd`. L'exemple suivant montre que les solutions de la congruence $x^{152} + x + 1 \equiv 0 \pmod{541}$ sont les trois résidus -412 , -138 et -130 .

```
p=541
w=x^152+x+1
w=mdgcd(x^p-x,w,p)
print w
print mdff(w,p)
```

Sortie (4760 ms)

```
x^3 +139*x^2 +139*x +138
[x +412]* [x +138]* [x +130]
```

Soit à résoudre en nombres entiers naturels ($x, y > 0$) l'équation :

$$x^3 + 1279y^2 = 1990^2. \quad (4.21)$$

Comme on doit avoir $x \leq \lfloor 1990^{2/3} \rfloor = 158$, une façon de procéder consiste à porter successivement dans (4.21) les 158 valeurs possibles de x , en examinant chaque fois si y , qui s'obtient par une racine carrée, est un entier. Cependant, on peut limiter le nombre de valeurs x à tester si on résout d'abord (4.21) modulo 1279 (ce qui fait disparaître y). La factorisation :

```
print mdff(x^3-1990^2,1279)
```

Sortie (605 ms)

```
[x +1234]* [x +982]* [x +342]
```

montre alors que seul $x = 45 \equiv -1234 \pmod{1279}$ est possible. Cette valeur donne la solution $45^3 + 1279 \times 55^2 = 1990^2$.

Racine carrée modulaire

Si a est un résidu quadratique modulo le nombre premier p , la fonction `prsq`(a, p) détermine la solution $x \in [1, p/2]$ de la congruence $a \equiv x^2 \pmod{p}$. Par exemple, soit à résoudre l'équation considérée par Gauss (*Disquisitiones Arithmeticae*, n° 328) :

$$x^2 \equiv -1365 \pmod{5428681}. \quad (4.22)$$

Le module dans (4.22) est composé : $P = m_1 m_2$, où $m_1 = 307$ et $m_2 = 17683$ sont des nombres premiers. La fonction `prsq` détermine les racines carrées $\pm a_i$ de -1365 modulo m_i (pour $i = 1, 2$). Les 4 solutions de (4.22), ± 2350978 et ± 2600262 s'obtiennent en remontant modulo $m_1 m_2$ par le théorème des restes chinois.

```
m1=307
m2=17683
P=m1*m2
a1=prsq(-1365,m1)
a2=prsq(-1365,m2)
print mods(chinoiseq(1,a1,m1,1,a2,m2),P)
print mods(chinoiseq(1,a1,m1,1,-a2,m2),P)
```

Sortie (1040 ms)

```
-2350978
-2600262
```

Description de `prsq`

La fonction `prsq` utilise une méthode empruntée à Koblitz (1987) p47, qui est un peu plus rapide que la factorisation de $x^2 - a \pmod{p}$ par `mdff`. Nous supposons $p > 2$, le cas $p = 2$ qui se limite à $a = 0, 1$ étant évident. La fonction `prfact`($p-1, 2$) donne la décomposition $p-1 = 2^\alpha s$ où s est un nombre impair ($\alpha = \alpha \geq 1$). Soit $r = a^{(s+1)/2}$. Le nombre $S = a^{-1}r^2$ vérifie l'équation :

$$S^{2^{\alpha-1}} \equiv a^{s2^{\alpha-1}} = a^{(p-1)/2} \equiv (a|p) = 1 \pmod{p}, \quad (4.23)$$

où le symbole de Legendre $(a|p)$, donné par la formule d'Euler (4.17), doit valoir 1 pour que le nombre a admette une racine carrée. Si $\alpha = 1$, l'équation (4.23) montre que r est une racine carrée de a et le problème est résolu. Sinon, nous déterminons la racine carrée de a par l'algorithme suivant.

1. Initialisation

Soit W un non-résidu quadratique de p . W s'obtient par un choix au hasard, qui est répété, dans la boucle `do ... loop`, jusqu'à ce que

$$(W^s)^{2^{\alpha-1}} \equiv (W|p) = -1 \pmod{p}. \quad (4.24)$$

L'équation (4.24) indique que le nombre $w = W^s$ est une racine primitive 2^α ième de l'unité (dans le programme W et w sont désignés par la même variable w). On pose $f = 1$.

2. Pour $i = \alpha - 2, \alpha - 3, \dots, 0$

On a ici

$$(f^2 S)^{2^i} \equiv \pm 1, \quad w^{2^{i+1}} \equiv -1 \pmod{p}. \quad (4.25)$$

Si $(f^2 S)^{2^i} \equiv -1$, alors remplacer f par fw .

Remplacer w par w^2 .

3. Solution

On a ici $f^2 S \equiv 1 \pmod{p}$, c'est-à-dire que $x = \pm rf$, qui vérifie $x^2 \equiv a \pmod{p}$, donne les solutions cherchées.

```

prsqrf: function(a,p)
    ift not prtst(p) ERREUR p non premier
    a=modr(a,p)
    value=a
    ift a=0 return
    ift a=1 return
    ift legendre(a,p)<>1 ERREUR a n'est pas un carré
    local var w,s,f
    w=prfact(p-1,2)
    s=max(norm(polym(w,2)),1)
    local datai deg(polym(w,1)) index ae
    value=mdpwre(a,(s+1)/2,p)
    if ae>1
        do
            w=random(p)
            ift legendre(w,p)=-1 exit
        loop
        w=mdpwre(w,s,p)
        f=1
        s=modr(value^2*prinv(a,p),p)
        for ae=ae-2,0
            ift mdpwre(s*f^2,2^ae,p)=p-1 f=modr(f*w,p)
            ift ae w=mdpwre(w,2,p)
        next ae
        value=modr(value*f,p)
    endif
    value=min(value,p-value)
    return

```

Autre méthode

Nous donnons une autre fonction $\text{prsqrf1}(a, p)$ qui détermine une racine carrée modulo un nombre premier p , en utilisant l'extension quadratique F_{p^2} .

On peut construire F_{p^2} de la façon suivante. Soit s un non-résidu de p . Les éléments de F_{p^2} sont les p^2 polynômes $u + vx$ où u et v sont des entiers modulo p . L'addition et la multiplication sont définies par l'addition et la multiplication des polynômes, suivies d'une réduction modulo p et modulo $x^2 - s$. L'ensemble F_{p^2} , muni de ces opérations est un corps. On dispose en Basic 1000d de fonctions permettant de calculer dans ce corps. Ainsi, $S = A + B$, $P = AB$, A^{-1} (si $A \neq 0$) et A^k (k entier relatif) sont donnés par

```
mdmod(A+B,x^2-s,p)
mdmod(A*B,x^2-s,p)
mdinv(A,x^2-s,p)
mdpwr(A,k,x^2-s,p)
```

Dans F_{p^2} , on a :

$$(A + B)^p = \sum_{k=0}^p \binom{p}{k} A^k B^{p-k} = A^p + B^p \quad (4.26)$$

(puisque les coefficients du binôme $\binom{p}{k}$ ($k = 1, 2, \dots, p-1$) sont $\equiv 0 \pmod{p}$), et

$$x^p = s^{(p-1)/2} x = -x \quad (4.27)$$

(d'après la formule d'Euler (4.17) puisque s est un non-résidu de p).

Pour calculer la racine carrée de a , on choisit un nombre w tel que $s = w^2 - a$ soit un non-résidu de p . Dans le programme, ce choix est effectué dans la boucle `do ... loop` en prenant un nombre au hasard jusqu'à ce que $s = 0$ (ce qui donne immédiatement la racine carrée w) ou que la fonction `legendre` indique que s est un non-résidu de p . Une racine carrée est alors donnée par $r = (w + x)^{(p+1)/2}$, qui est calculé par `mdpwr`. En effet, on a

$$r^2 = (w+x)^p(w+x) = (w^p+x^p)(w+x) = (w-x)(w+x) = w^2 - x^2 = w^2 - s = a, \quad (4.28)$$

où on a utilisé les équations (4.26) et (4.27) ainsi que le théorème de Fermat (4.3) (pour écrire $w^p = w$).

Les vitesses de `prsq1` et `prsq` sont très voisines, avec un léger avantage pour `prsq` en moyenne (les deux procédures sont probabilistes; on peut observer des temps très différents pour le même calcul).

```
p=2202244139
a=-374540435
print prsq1(a,p)
stop
prsq1:function(a,p)
  ift not prtst(p) ERREUR p non premier
  a=modr(a,p)
  value=a
  ift a=0 return
  ift a=1 return
```

```

ift legendre(a,p)<>1 ERREUR a n'est pas un carré
local lit x
local var w,s,f
do
  w=random(p)
  s=modr(w^2-a,p)
  ift s=0 goto prsq_s
  ift legendre(s,p)=-1 exit
loop
w=mdpwr(w+x, (p+1)/2, x^2-s, p)
prsq_s:value=min(w, p-w)
return

```

Sortie (2665 ms)

613630743

Exercice 4.4. Racine *mième*

Résoudre la congruence :

$$x^{27182817} \equiv 2 \pmod{31415971}. \quad (4.26)$$

Factorisation des entiers de Gauss

L'anneau des entiers de Gauss, c'est-à-dire les nombres $a + bi$ où a et b sont des entiers et $i = \sqrt{-1}$, a des propriétés voisines des propriétés de l'anneau des entiers \mathbf{Z} : on peut définir des nombres premiers; tout entier de Gauss se décompose en facteurs premiers et la décomposition est unique, aux unités ± 1 , $\pm i$ près. Les entiers de Gauss premiers sont les nombres suivants, définis à la multiplication par une unité ± 1 , $\pm i$ près :

- Le nombre $1 + i$.
- Les nombres $2 \pm i$, $3 \pm 2i$, $4 \pm i$, $5 \pm 2i$, \dots , c'est-à-dire tous les facteurs $a \pm bi$ de $p = a^2 + b^2$, où p est un nombre premier de la forme $p = 4n + 1$.
- Les nombres 3 , 7 , 11 , 19 , 23 , 31 , \dots , c'est-à-dire les nombres premiers de la forme $4n - 1$.

La fonction *sumsq*

Nous examinons d'abord le problème d'obtenir le facteur $p = a \pm bi$ d'un nombre premier $p \equiv 1 \pmod{4}$. Ce problème est lié au théorème de Fermat (montré par Euler en 1747) : tout nombre premier de la forme $p = 4n + 1$ est d'une seule façon la somme de deux carrés; par exemple $5 = 2^2 + 1^2$ et $13 = 3^2 + 2^2$.

Cette décomposition, $p = a^2 + b^2$, qui s'écrit aussi $p = (a + bi)(a - bi)$, donne les entiers de Gauss premiers $a \pm bi$.

La fonction `sumsq(p)`, qui exige le mode complexe, renvoie l'entier de Gauss $a + bi$, où $a \geq b > 0$ (en fait $a > b$ pour $p > 2$), qui divise p . Le nombre p doit être un nombre premier de la forme $4n + 1$, ou le nombre $2 = -i(1 + i)^2$ qui est traité à part. Dans le programme, i est une variable locale initialisée avec le littéral complexe `complex` (qui peut avoir un nom quelconque). La méthode consiste à déterminer d'abord un entier naturel A (noté `a` dans le programme) tel que $A^2 + 1 \equiv 0 \pmod{p}$. Si $z \equiv m^{2k+1} \pmod{p}$ est une puissance impaire de l'élément primitif m modulo p , alors on peut prendre $A = z^{(p-1)/4}$, puisque $A^2 \equiv m^{(p-1)/2} \equiv -1 \pmod{p}$. Le programme détermine le plus petit entier z convenable à l'aide de la boucle `do ... loop`. L'entier de Gauss cherché $a + bi$ est ensuite donné par le pgcd de $A + i$ et p , qui est calculé par la fonction `cxgcd`. L'exemple écrit la décomposition en somme de deux carrés d'un nombre premier $p = 4n + 1$.

```

complex i
p=84892074342879679333
v=sumsq(p)
print using "#_ ^2 _+ #_ ^2 = #";re(v);im(v);p
stop
sumsq:function(p)
local datav complex var i,z,a,b
if p=2
    value=1+i
    return
endif
z=1
do
    z=z+1
    a=mdpwre(z,(p-1)/4,p)
    ift modr(a^2+1,p)=0 exit
loop
z=cxgcd(a+i,p)
a=abs(re(z))
b=abs(im(z))
ift a<b exg a,b
value=a+i*b
return

```

Sortie (3315 ms)

7048680062^2 + 5933648433^2 = 84892074342879679333

La fonction `cxfact`

La c_fonction `cxfact(g)` renvoie la factorisation complète de l'entier de Gauss g sous la forme d'un `t_ensemble`. A la factorisation $g = up_1^{\alpha_1} p_2^{\alpha_2} \cdots p_r^{\alpha_r}$,

où $u = \pm 1$, $\pm i$ est une unité et où les p_i sont les entiers de Gauss premiers, correspond un `t_ensemble` formé de $r+1$ `t_éléments` : d'abord `eset$(1, mkx$(u))`, puis `eset$(α_i , mkx$(p_i))`, pour $1 \leq i \leq r$. Le $(i+1)$ ième `t_élément` a pour type l'exposant α_i et pour valeur le codage machine du nombre premier p_i . Les facteurs premiers p_i dans cette décomposition sont bien définis, et pas seulement à une unité près, par les choix suivants : $1+i$; $a \pm bi$ où $a > b > 0$ pour les facteurs de $p = 4n+1$; $q = 4n-1$. Les nombres p_i sont ordonnés par `cxcmp(p_i, p_{i+1}) > 1`. La fonction `cxcmp(x,y)` renvoie $-1, 0$ ou 1 si respectivement $x > y$, $x = y$ ou $x < y$. L'ordre est essentiellement donné par par l'ordre des normes (`cxnorm($a+bi$)` calcule la norme $N(a+bi) = a^2 + b^2$) : si $N(x) < N(y)$ on a $x < y$; si $N(x) = N(y)$, l'ordre est donné par comparaison des parties réelles ou des parties imaginaires.

Lorsque un nouveau facteur premier p^k est déterminé, on appelle la procédure `cxfact_et` qui place ce facteur dans la variable `value` (qui sera le résultat de la fonction `cxfact`) de sorte que les facteurs sont ordonnés dans l'ordre croissant. Cela est réalisé en manipulant le `t_ensemble value` à l'aide de `elementy(value,1)` et `elementv(value,1)`, qui donnent le type et la valeur du premier `t_élément`, et de `cdr$(value)` qui ôte le premier `t_élément` du `t_ensemble`. La variable `cm` contient les `t_éléments` ôtés de `value`, qui correspondent à des facteurs plus petits que p .

Pour factoriser $g = A + Bi$, on factorise d'abord le pgcd n de A et B en facteurs premiers réels. Les facteurs premiers $p \equiv 3 \pmod{4}$ de n sont également des facteurs premiers de g , un facteur 2 dans n donne $(1+i)^2$ et les facteurs premiers $p \equiv 1 \pmod{4}$ de n donnent la décomposition $(a+bi)(a-bi)$ (calculée par `sumsq`).

La partie résiduelle de g ($= g/n$ à une unité près), également notée g , est ensuite décomposée à partir de la factorisation de la norme de g , $N(g)$ (aussi notée n). A chaque facteur premier p^k de n (on a $p \not\equiv 1 \pmod{4}$), correspond le facteur $(a+bi)^k$ ou $(a-bi)^k$ de g , qui s'obtient à partir de `sumsq(p)` : si la valeur $a+bi$ ne divise pas g (ce qui est testé par `cxmod`), alors le facteur est son conjugué $a-bi$ (calculé par `cc`).

Les fonctions `cxfact_c`, `cxfact_p` et `cxfact$`

La `c_fonction` `cxfact_c(c)` transforme un `t_ensemble c` calculé par `cxfact` en chaîne de caractères affichables. La lecture du `t_ensemble c` est réalisée à l'aide de `elementy(c, k)` et `elementv(c, k)`, l'index k parcourant les valeurs de 1 à `elementn(c)`. La `v_fonction` `cxfact_c(c)`, appliquée à un `t_ensemble c` calculé par `cxfact(g)`, effectue le produit des facteurs, ce qui doit redonner g (pour vérifier les décompositions obtenues). Si on désire seulement afficher la décomposition de g , on utilisera la fonction `cxfact$(g)`, qui renvoie directement la chaîne affichable, sans conserver le `t_ensemble`.

L'exemple factorise un entier de Gauss aléatoire $A + Bi$, où $A, B \in [0, 10^5]$, obtenu par `random(105, i, 1)`.

'adjoindre `sumsq`

```

        complex i
        g=random(10^5,i,1)
        c$=cxfact(g)
        print g;"=";cxfact_c(c$)
        stop
    cxfact$:function$(g)
        value=cxfact_c(cxfact(g))
        return
    cxfact_p:function(char c)
        local index k,iz
        local var z
        value=1
        for k=1,elementn(c)
            value=value*elementv(c,k)^elementy(c,k)
        next k
        return
    cxfact_c:function$(char c)
        local char c1
        local index k,iz
        local var z,a,b
        c1=""
        for k=1,elementn(c)
            cadd value,c1
            z=elementv(c,k)
            iz=elementy(c,k)
            if z<>1
                a=re(z)
                b=im(z)
                if a<>0 and b<>0
                    cadd value,"("&justl$(a)
                    ift b>0 cadd value,"+"
                    ift b<0 cadd value,"-"
                    b=abs(b)
                    ift b<>1 cadd value,justl$(b)
                    cadd value,"i)"
                else
                    cadd value,justl$(z)
                endif
                ift iz>1 cadd value,"^"&justl$(iz)&chr$(22)
                c1=" * "
            endif
        next k
        return
    cxfact:function$(g)

```

```

local datav complex var i,n,w,u,p
local index j,k
n=gcdr(re(g),im(g))
if n>1
  g=g/n
  w=prfact(n)
  for j=polymn(w),1
    u=polym(w,j)
    p=norm(u)
    k=deg(u)
    if modr(p,4)=1
      p=sumsq(p)
      cxfact_et
      p=cc(p)
    else p=2
      p=1+i
      g=(-i)^k*g
      k=2*k
    endif
    cxfact_et
  next j
endif
n=cxnorm(g)
ift n=1 goto cxfact_1
w=prfact(n)
for j=polymn(w),1
  u=polym(w,j)
  p=norm(u)
  k=deg(u)
  p=sumsq(p)
  ift cxmod(g,p)<>0 p=cc(p)
  cxfact_et
  g=cxdiv(g,p^k)
next j
cxfact_1:value=eset$(1,mkx$(g))&value
return
cxfact_et:procedure
  local var z
  local index iz
  local char cm
  while value<>""
    z=elementv(value,1)
    iz=elementy(value,1)
    select cxcmp(p,z)

```

```

        case=0
        value=cm&eset$(iz+k,mkx$(z))&cdr$(value)
        return
    case<0
        cadd cm,eset$(iz,mkx$(z))
        value=cdr$(value)
    case>0
        value=cm&eset$(k,mkx$(p))&value
        return
    endselect
wend
value=cm&eset$(k,mkx$(p))
return
cxcmp:function(u,v)
    ift u=v return
    value=sgn(cxnorm(v)-cxnorm(u))
    ift value return
    value=sgn(re(v-u))
    ift value return
    value=sgn(im(v-u))
    return

```

Sortie (4160 ms)

```
61259*i +84897=i * (1+i) * (2-i) * (82-47i) * (257-238i)
```

5

Factorisation des nombres



Ce chapitre décrit en détail les procédures **pollard**, **brison** et **lenstra** de la bibliothèque MATH qui factorisent les nombres entiers. Dans l'exposé des méthodes utilisées nous avons été amené à décrire, dans la solution des exercices, deux autres programmes d'intérêt général : la fonction **fermat\$(n)** qui factorise n par la méthode de Fermat, et la procédure **fcont** qui effectue le développement en fraction continue d'un nombre. Nous décrivons, à la fin du chapitre, des fonctions **prfactb** et **prfactb\$**, analogues aux fonctions internes **prfact** et **prfact\$**, mais adaptées à la factorisation des grands nombres. Nous recommandons chaudement le livre de Riesel, qui est très bien écrit, aux personnes intéressées par la factorisation des nombres.

D'un point de vue pratique, nous vous conseillons la stratégie suivante pour factoriser un entier n . Si $n < 10^{10}$, utiliser simplement la fonction interne **prfact\$(n)** (ou **prfact(n)**). Ce calcul prendra moins d'une seconde en moyenne, sans jamais dépasser 16 secondes, même dans les cas les plus difficiles. Si $n > 10^{10}$, utiliser la procédure **brison(n)** ou les fonctions **prfactb** ou **prfactb\$**. En général, les nombres $n < 10^{30}$ les plus difficiles sont factorisés en moins de 4 heures, et, avec un peu de chance, la procédure traitera aussi en un temps raisonnable des nombres ayant jusqu'à 40 chiffres (par exemple $10^{37} + 1$ est factorisé en 7 heures). La procédure **brison** est en général nettement plus rapide que les procédures **pollard** et **lenstra**, mais ces dernières procédures peuvent être essayées pour les très grands nombres (plus de 30 chiffres). En effet, lorsque le nombre n contient un petit facteur premier (moins de 10 chiffres), la procédure **pollard**, ou l'appel **lenstra(n, 10¹⁰)**, permet de découvrir rapidement ce facteur. En outre, pour les très grands nombres, la méthode de Lenstra devient théoriquement plus rapide que la méthode de Brillhart et Morrison (**brison**). Par exemple, la factorisation du nombre de 33 chiffres

$$\begin{aligned} n &= 2^{109} + 1 = 649037107316853453566312041152513 \\ &= 3 \times 104124649 \times 2077756847362348863128179, \end{aligned}$$

peut être obtenue par **brison(n)** en 7 heures environ, mais, comme le deuxième plus grand facteur est de l'ordre de 10^8 , **pollard(n)** ou **lenstra(n, 10¹⁰)** donne cette factorisation en moins de 20 minutes. Il est également possible d'utiliser la commande :

```
print prfact$((2^109+1)/3,0)
```

qui donne la factorisation en 24 heures environ.

Notons qu'en 1989, sur les ordinateurs les plus rapides, on arrive à factoriser de façon générale des nombres de 100 à 120 chiffres, au prix de plusieurs heures ou plusieurs jours de calcul.

La méthode rho de Pollard

Le procédé de factorisation décrit ici a été publié en 1975 par Pollard. Il est mis en œuvre dans la procédure `pollard(n)` qui convient pour découvrir les facteurs premiers d'une douzaine de chiffres. On peut en particulier factoriser des nombres n ayant jusqu'à 25 chiffres.

Soit p un facteur premier du nombre n à factoriser (on suppose que $p \neq n$). La méthode ρ de Pollard, qui est aussi appelée la méthode Monte-Carlo de factorisation, utilise une fonction $f(x)$ de l'anneau \mathbf{Z}_n des entiers modulo n dans lui-même. En partant de x_0 , on génère par récurrence la suite de nombres

$$x_{i+1} = f(x_i), \quad i = 0, 1, 2, \dots$$

Pour que la méthode fonctionne, la suite ainsi obtenue doit ressembler à une suite de nombres pris au hasard. On désire aussi que la fonction $f(x)$ soit simple, pour que le calcul de la suite x_i soit rapide. Le choix habituel consiste à prendre un polynôme de degré 2, le plus souvent $f(x) = x^2 + 1$ (on peut montrer que la fonction linéaire $f(x) = Ax + B$ donne une suite x_i non aléatoire qui ne convient pas). Nous noterons par y_i le reste modulo p de x_i . En utilisant le fait que les nombres y_i prennent au plus p valeurs distinctes, il est facile de montrer que la suite y_i devient cyclique de période T à partir du rang a : $y_{i+T} = y_i$ pour $i \geq a$. On a toujours $T \leq p$ et $a < p$, mais si la suite x_i se comporte comme une suite aléatoire, on peut montrer que a et T sont statistiquement de l'ordre de \sqrt{p} .

Si on réussit à trouver deux nombres de la suite x_i et x_j différents modulo n , mais égaux modulo p , alors le pgcd de $x_j - x_i$ et n donne le nombre p , c'est à dire que l'on a aussi réussi à déterminer le facteur p de n .

Pour trouver ces (x_i, x_j) , on peut comparer chaque x_i pour $i = 1, 2, \dots$, avec tous les x_j précédents. Cela détermine la paire de nombres x_a, x_{a+T} , mais après $O((a+T)^2) \approx O(p)$ comparaisons. Il est cependant possible de trouver une paire (x_i, x_j) de nombres égaux modulo p beaucoup plus rapidement en $O(\sqrt{p})$ comparaisons. Le programme effectue les comparaisons suivantes :

x_0	avec	x_1
x_1	avec	x_2, x_3
x_3	avec	x_4, x_5, x_6, x_7
x_7	avec	x_8, x_9, \dots, x_{15}
\dots		\dots

Ainsi, le nombre x_m est comparé avec le nombre $x_{l(m)-1}$, où $l(m) = 2^{\text{intlg}(m)}$, jusqu'à ce que $x_m \equiv x_{l(m)-1} \pmod{p}$.

Le programme suit l'algorithme B de Knuth (page 370), en utilisant les mêmes notations.

1. `dabord_prfact`

La procédure `dabord_prfact` détermine les facteurs premiers inférieurs à 2^{16} par `prfact`. En effet, pour les petits facteurs, la méthode des divisions

successives est la meilleure. Le choix de la limite 2^{16} correspond à des diviseurs contenus dans un mot, pour lesquels la division est plus rapide (par exemple, la division `divr` de $2^{10000} + 1$ par $2^{16} + 1$ est 5 fois plus lente que la division par $2^{16} - 1$). Le résultat de cette factorisation est codée sous la forme d'un polynôme `np`. Parmi les `l=polymn(np)` facteurs, on affiche d'abord les `l - 1` premiers, qui sont de la forme p^k où p est un nombre premier : $p = \text{norm}(x)$ et $k = \text{deg}(x)$ sont obtenus à partir du i ème monôme x de `np`. On teste par `prtst` si le dernier facteur n est un nombre premier (on sait que si $n < 2^{32}$, alors n est premier). Si oui, le facteur est affiché et $n = 1$ en sortie.

2. Initialisation

Pendant le programme, le nombre à factoriser est n (on sait que n n'est pas premier), les variables `x` et `xp` contiennent x_m et $x_{l(m)-1}$, `l` contient $l(m)$ et $k = 2l - m$. En fin du programme, la variable `pollard_iter` contient l'indice m du dernier terme calculé x_m de la suite. Les valeurs initiales correspondent à $m = 1$: $l = 1$, $k = 1$, `xp` = $x_0 = 2$ et `x` = $x_1 = 2^2 + 1 = 5$.

3. Compare `x` et `xp`

On calcule le pgcd `g` de `x - xp` et n . Si `g` = 1, aller à l'étape 4. Si `g` = n , la méthode a échoué et le programme s'arrête, après affichage de `g` entre parenthèses. Sinon, `g`, qui est un facteur propre de n , est affiché (si `g` n'est pas premier, il est affiché entre parenthèses). On remplace n par n/g , puis `x` et `xp` par leurs restes modulo n . Si n est premier, le programme s'arrête; sinon on reprend cette étape 3 à son début.

4. Terme suivant de la suite

Cette étape revient à incrémenter m . Après avoir ôté 1 de `k`, si `k` = 0 (la nouvelle valeur de m vérifie $m = l(m)$) on remplace `xp` par `x`, `l` par `2*l` et `k` par 1. On donne à `x` sa valeur suivante $x^2 + 1 \pmod{n}$, puis on continue à l'étape 3.

```
pollard:procedure(n)
  var pollard_iter
  local var np,l,x,xp,k,g
  local index i
  dabord_prfact
  ift n=1 return
  x=5
  xp=2
  l=1
  k=1
pollard_3:g=gcdr(x-xp,n)
  ift g=1 goto pollard_4
  if g=n
    print "(";g;"")
    pollard_iter=2*l-k
```



```

        return
    endif
    if prtst(g)
        print g;" * ";
    else
        print "(";g;" ) * ";
    endif
    n=n/g
    x=modr(x,n)
    xp=modr(xp,n)
    if prtst(n)
        print n
        pollard_iter=2*l-k
        return
    endif
    goto pollard_3
pollard_4:k=k-1
    if k=0
        xp=x
        l=2*l
        k=1
    endif
    x=modr(x^2+1,n)
    goto pollard_3
dabord_prfact:print n;"= ";
    local var x
    np=prfact(n,2^16)
    l=polymn(np)
    if l>1
        for i=1,l-1
            x=polym(np,i)
            print justl$(norm(x));
            k=deg(x,phantom)
            if k=1
                print " * ";
            else
                print "^";justl$(k);" * ";
            endif
        next i
    endif
    x=polym(np,l)
    n=norm(x)
    k=deg(x,phantom)
    ift n>2^32 ift prtst(n)=0 return

```

```

if k=1
    print justl$(n)
else
    print justl$(n);"^";justl$(k)
endif
n=1
return

```

Example

Nous factorisons $10^{24} + 1$ par la procédure `pollard`.

```
pollard 10^24+1
```

Sortie (455 s)

[illegible]

Remarquons que ce résultat peut être déterminé plus rapidement en utilisant l'identité

$$10^{24} + 1 = (10^8 + 1)(10^{16} - 10^8 + 1),$$

qui s'obtient pour $x = 10$ à partir de la factorisation suivante.

```
print formf(x24+1)
```

Sortie (16355 ms)

$$[x^8 + 1] * [x^{16} - x^8 + 1]$$

Exercice 5.1. Variation pollard

Modifier la procédure `pollard` pour qu'on puisse préciser la fonction $f(x)$ et le premier terme x_0 de la suite dans l'appel `pollard(n , $f(x)$, x_0)`.

Factorisation de Fermat

Lorsque $n = ab$ est le produit de deux entiers impairs a et b proches de \sqrt{n} , il existe une méthode rapide de factorisation, due à Fermat. On obtient une correspondance entre les décompositions $n = ab$ et les représentations de n comme différence de deux carrés $n = s^2 - t^2$ en posant $a = s + t$, $b = s - t$. Lorsque n est impair, s et t sont des entiers. Par exemple $77 = 7 \times 11 = 9^2 - 2^2$. La méthode de Fermat consiste à rechercher les représentations $n = s^2 - t^2$ en essayant successivement si, pour $t = 0, 1, 2, \dots$, le nombre $n + t^2$ est un carré parfait.

Exercice 5.2. Fermat

Programmer la méthode de Fermat, et l'appliquer aux nombres

$w = 284620201979$ et $x = 226077651799$.

La méthode de Legendre

Nous allons exposer maintenant les idées qui sont à la base du procédé de factorisation de Brillhart et Morrison, qui est l'un des plus performants connus à ce jour. La méthode de factorisation de Legendre est la généralisation suivante de la méthode de Fermat. Si on trouve des entiers s et t vérifiant

$$s^2 \equiv t^2 \pmod{n}, \quad s \not\equiv \pm t \pmod{n},$$

alors on obtient des facteurs de n en calculant le pgcd de n et $s \pm t$. En effet, comme n divise $s^2 - t^2 = (s - t)(s + t)$ sans diviser $s - t$ ou $s + t$, le pgcd de n et $s + t$ doit être un facteur de n autre que 1 ou n . Examinons comment on peut chercher des solutions de $s^2 \equiv t^2 \pmod{n}$ pour $n = 5177$. En prenant des entiers b proches de $\sqrt{n} \approx 71.9$, on obtient des équivalences $b^2 \equiv r \pmod{n}$, où le nombre $r = \text{mods}(b^2, n)$ est petit. En combinant :

$$72^2 \equiv 7 \pmod{5177}, \quad 75^2 \equiv 448 = 2^6 \times 7 \pmod{5177},$$

on obtient :

$$5400^2 = (72 \times 75)^2 \equiv (2^3 \times 7)^2 = 56^2 \pmod{5177},$$

qui conduit à la découverte des facteurs $167 = \text{pgcd}(5400 - 56, 5177)$ et $31 = \text{pgcd}(5400 + 56, 5177)$.

Pour généraliser la méthode, nous devons trouver des petits résidus quadratiques. Le procédé qui consiste à utiliser des entiers b proches de \sqrt{kn} (k entier) comme ci-dessus nécessite des calculs avec des nombres de l'ordre de kn . Le développement en fraction continue de \sqrt{kn} , où k est entier, permet également d'obtenir des petits résidus quadratiques, mais de façon beaucoup plus efficace car les calculs se font sur des entiers de l'ordre de grandeur de \sqrt{kn} . La section suivante expose les résultats de la théorie des fractions continues qui nous seront nécessaires.

Développement en fraction continue

On construit le développement en fraction continue d'un nombre réel x_0 de la façon suivante. Soit $a_0 = \lfloor x_0 \rfloor$ la partie entière de x_0 . Pour $n = 1, 2, 3, \dots$, tant que $x_{n-1} \neq a_{n-1}$, on pose $x_n = 1/(x_{n-1} - a_{n-1})$ et $a_n = \lfloor x_n \rfloor$. Les entiers

a_n sont appelés quotients incomplets. On peut écrire pour chaque n ci-dessus ($n \geq 1$) :

$$x_0 = a_0 + \frac{1}{a_1 + \frac{1}{a_2 + \frac{1}{a_3 + \frac{1}{\cdots + \frac{1}{a_{n-1} + \frac{1}{x_n}}}}}}. \quad (5.1)$$

Un résultat classique est que la suite a_n se termine si et seulement si x_0 est un nombre rationnel. Pour la démonstration de ce résultat, ainsi que pour une étude plus approfondie des fractions continues, nous vous renvoyons au livre de Hua Loo Keng. Nous supposons que x_0 est un nombre irrationnel. Lorsqu'on remplace x_n par a_n dans la fraction (5.1), on obtient la fraction :

$$\frac{p_n}{q_n} = a_0 + \frac{1}{a_1 + \frac{1}{a_2 + \frac{1}{a_3 + \frac{1}{\cdots + \frac{1}{a_n}}}}}, \quad (5.2)$$

appelée convergent (ou réduite) d'ordre n de x_0 . Les convergents peuvent se calculer par récurrence :

$$\begin{cases} p_{-1} = 1 \\ q_{-1} = 0 \end{cases} \quad \begin{cases} p_0 = a_0 \\ q_0 = 1 \end{cases} \quad \begin{cases} p_1 = a_1 a_0 + 1 \\ q_1 = a_1 \end{cases} \quad \begin{cases} p_n = a_n p_{n-1} + p_{n-2} \\ q_n = a_n q_{n-1} + q_{n-2} \end{cases} \quad (5.3)$$

Les fractions p_n/q_n sont alors sous forme réduite, et on a pour tout n :

$$p_n q_{n-1} - p_{n-1} q_n = (-1)^{n-1}. \quad (5.4)$$

Les convergents p_n/q_n ont pour limite x_0 lorsque $n \rightarrow \infty$. Une propriété remarquable est que la fraction p_n/q_n est, parmi les fractions p/q telles que $p \leq p_n$ et $q \leq q_n$, celle qui est la plus proche du nombre x_0 .

Exercice 5.3. Fraction continue

Ecrire un programme effectuant la décomposition en fraction continue de π , e et $\sqrt{7}$.

Développement d'une racine carrée

Un théorème dû à Lagrange indique que la suite des quotients incomplets devient périodique à partir d'un certain rang si et seulement si x_0 est un nombre irrationnel quadratique, c'est-à-dire un nombre irrationnel de la forme $x_0 = s + t\sqrt{d}$, où s , t et d sont rationnels.

Le théorème suivant permet de déterminer le développement en fraction continue de la racine carrée d'un nombre entier \sqrt{d} , où d n'est pas un carré parfait. Ce développement devient périodique d'après le théorème de Lagrange.

Théorème

Dans le développement en fraction continue de $x_0 = \sqrt{d}$, les nombres x_n sont de la forme :

$$x_n = \frac{\sqrt{d} + U_n}{V_n}, \quad U_n^2 \equiv d \pmod{V_n}, \quad (5.5)$$

où U_n et V_n sont des entiers positifs ($0 \leq U_n < \sqrt{d}$ et $0 < V_n < 2\sqrt{d}$) qui peuvent se calculer par récurrence par :

$$\begin{aligned} a_n &= [x_n] = \left\lfloor \frac{\lfloor \sqrt{d} \rfloor + U_n}{V_n} \right\rfloor \\ U_{n+1} &= a_n V_n - U_n \\ V_{n+1} &= V_{n-1} + a_n(U_n - U_{n+1}), \end{aligned} \quad (5.6)$$

en partant de

$$\begin{aligned} U_0 &= 0, \quad a_0 = \lfloor \sqrt{d} \rfloor, \quad U_1 = a_0, \\ V_0 &= 1, \quad V_1 = d - a_0^2. \end{aligned} \quad (5.7)$$

Les numérateurs et dénominateurs des convergents (5.3) vérifient la relation :

$$p_{n-1}^2 - dq_{n-1}^2 = (-1)^n V_n. \quad (5.8)$$

Démonstration

A partir de $x_0 = \sqrt{d}$ et $x_1 = 1/(\sqrt{d} - a_0)$, on obtient les équations (5.7) et la validité de (5.5) pour $n = 0$ et 1. On raisonne ensuite par récurrence, en supposant (5.5) vrai pour n . Puisque $1/x_{n+1} = x_n - a_n$, on doit montrer qu'il existe U_{n+1} et V_{n+1} tels que :

$$\frac{V_{n+1}}{\sqrt{d} + U_{n+1}} = \frac{\sqrt{d} + U_n}{V_n} - a_n, \quad (5.9)$$

et

$$U_{n+1}^2 \equiv d \pmod{V_{n+1}}. \quad (5.10)$$

L'équation (5.9) équivaut à :

$$V_n V_{n+1} = d + U_n U_{n+1} - a_n V_n U_{n+1}, \quad (5.11)$$

$$U_{n+1} + U_n - a_n V_n = 0. \quad (5.12)$$

En éliminant U_n entre (5.11) et (5.12), il vient :

$$d - U_{n+1}^2 = V_n V_{n+1}. \quad (5.13)$$

L'équation (5.12) montre que U_{n+1} est un entier tel que $U_{n+1}^2 \equiv U_n^2 \pmod{V_n}$. L'hypothèse de récurrence (5.5) donne alors $U_{n+1}^2 \equiv d \pmod{V_n}$, ce qui montre que V_{n+1} , tiré de l'équation (5.13), est un nombre entier. L'équation (5.13) montre également que la congruence dans (5.5) tient pour $n+1$. La formule (5.6) pour le calcul de V_{n+1} s'obtient en retranchant les équations (5.13) écrites pour n et $n \rightarrow n-1$:

$$V_n V_{n+1} - V_n V_{n-1} = U_n^2 - U_{n+1}^2 = (U_n + U_{n+1})(U_n - U_{n+1}),$$

puis en utilisant (5.12). Sous cette forme, il n'y a pas besoin de division pour calculer V_{n+1} .

Nous démontrons l'équation (5.8) comme suit. La relation :

$$x_0 = \sqrt{d} = \frac{x_n p_{n-1} + p_{n-2}}{x_n q_{n-1} + q_{n-2}}, \quad (5.14)$$

s'obtient en remplaçant a_n par x_n dans les équations (5.2) et (5.3), puis en comparant avec (5.1). En portant la valeur de x_n donnée par (5.5) dans cette équation (5.14), il vient puisque \sqrt{d} est irrationnel :

$$p_{n-1} = q_{n-1} U_n + q_{n-2} V_n, \quad (5.15)$$

$$dq_{n-1} = p_{n-1} U_n + p_{n-2} V_n. \quad (5.16)$$

En éliminant U_n entre (5.15) et (5.16) il vient :

$$p_{n-1}^2 - dq_{n-1}^2 = (p_{n-1} q_{n-2} - p_{n-2} q_{n-1}) V_n, \quad (5.17)$$

qui donne la relation cherchée (5.8) d'après l'équation (5.4).

Il nous reste à montrer les inégalités $0 \leq U_n < \sqrt{d}$ et $0 < V_n < 2\sqrt{d}$. La relation (5.8) et le fait que la suite $p_n/q_n - \sqrt{d}$ est alternée montre que V_n a un signe constant, qui est donc positif puisque $V_0 > 0$. L'équation (5.13) montre alors que $|U_n| < \sqrt{d}$. La relation $x_n > 1$ montre, en utilisant la forme (5.5) de x_n , que $V_n < \sqrt{d} + U_n < 2\sqrt{d}$. Pour montrer que $U_n \geq 0$ nous distinguons deux cas. Si $V_{n-1} < \sqrt{d}$, puisque d'après (5.13) :

$$1 < x_n = \frac{\sqrt{d} + U_n}{V_n} = \frac{V_{n-1}}{\sqrt{d} - U_n},$$

les inégalités $\sqrt{d} > V_{n-1} > \sqrt{d} - U_n$ impliquent que U_n est positif.

Si $V_{n-1} > \sqrt{d}$, nous utilisons (5.12) pour écrire :

$$U_{n-1} + U_n = a_{n-1} V_{n-1} \geq V_{n-1} > \sqrt{d}$$

d'où le résultat puisque $U_{n-1} > -\sqrt{d}$.

Application

Dans la section précédente, nous avons vu que la factorisation d'un entier n par la méthode de Legendre nécessite la recherche de congruences de la forme $P^2 \equiv t \pmod{n}$, où le résidu t est petit devant n . Le développement en fraction continue de \sqrt{n} ou de \sqrt{kn} , où k est entier, permet d'obtenir de telles

congruences. En effet, d'après (5.8), les numérateurs p_{n-1} des réduites vérifient l'équation $p_{n-1}^2 \equiv (-1)^n V_n \pmod{n}$ où les V_n sont petits (inférieurs à $2\sqrt{kn}$).

Bases de facteurs

Dans l'exemple de la factorisation de $n = 5177$ par la méthode de Legendre, nous avons effectué le produit des deux congruences $72^2 \equiv 7 \pmod{5177}$ et $75^2 \equiv 448 = 2^6 \times 7 \pmod{5177}$ pour obtenir $5400^2 \equiv 56^2 \pmod{5177}$. Cette idée, due à Maurice Kraitchik, de combiner divers résidus pour former des carrés, se prête à une automatisation de la façon suivante. On se donne une *base de facteurs* $B = \{p_0 = -1, p_1, \dots, p_m\}$, formée du nombre -1 et de m nombres premiers distincts p_i ($i \geq 1$). On dit que b est un B -nombre si le résidu minimum signé de b^2 , $\text{mods}(b^2, n)$, est un produit de facteurs de B . Pour l'exemple ci-dessus et $B = \{-1, 2, 7\}$, 72 et 75 sont des B -nombres. Si nous trouvons suffisamment de B -nombres b_i ($i = 1, \dots, k$) :

$$b_i^2 \equiv (-1)^{e_{0i}} p_1^{e_{1i}} p_2^{e_{2i}} \dots p_m^{e_{mi}} \pmod{n}, \quad (5.18)$$

alors il sera possible de former des produits

$$(b_\alpha b_\beta \dots b_\mu)^2 \equiv (-1)^{e_0} p_1^{e_1} p_2^{e_2} \dots p_m^{e_m} \pmod{n}, \quad (5.19)$$

où les exposants $e_k = e_{k\alpha} + e_{k\beta} + \dots + e_{k\mu}$ sont tous pairs. On aura ainsi trouvé une solution de $s^2 \equiv t^2 \pmod{n}$, et si $s \not\equiv \pm t \pmod{n}$, alors la méthode de Legendre donne une factorisation de n . En écrivant $s = \prod b_i^{\xi_i}$, où les inconnues ξ_i valent 0 ou 1, on voit que trouver (5.19) revient à résoudre le système d'équations linéaires modulo 2 :

$$\sum_{i=0}^k \xi_i e_{ji} = e_j \equiv 0 \pmod{2}. \quad (5.20)$$

La méthode de Brillhart et Morrison

Principe

Soit n un nombre impair composé à factoriser et k un entier $k > 0$ sans facteur carré. On se donne un nombre premier p_m et un entier P_0 tel que $P_0 < p_m p'$, où p' est le plus petit nombre premier plus grand que p_m . Nous avons vu que le développement en fraction continue de \sqrt{kn} permet d'obtenir des congruences $P^2 \equiv t \pmod{n}$, où les résidus t sont petits. Ces résidus,

$$t = (-1)^s p_1^{\alpha_1} p_2^{\alpha_2} \cdots p_m^{\alpha_m} \rho, \quad (5.21)$$

sont factorisés en un produit de nombres premiers inférieurs à p_m , du signe $(-1)^s$ et du facteur ρ . Si $\rho \leq P_0$, alors, puisque $P_0 < p_m p'$, ρ est également un nombre premier, et on conserve la congruence $P^2 \equiv t \pmod{n}$. Par contre, si $\rho > P_0$, la congruence est rejetée. La base de facteurs est définie au fur et à mesure que les congruences sont obtenues, en partant de $B = \{p_0 = -1, p_1 = 2\}$, puis en rajoutant à B les facteurs de la décomposition de t qui n'y sont pas encore. Pour chaque nouvelle congruence retenue, on examine si on peut la combiner aux précédentes pour former une congruence $S^2 \equiv T^2 \pmod{n}$. Si oui, et si $S \not\equiv \pm T \pmod{n}$, alors la factorisation est obtenue, mais si $S \equiv \pm T \pmod{n}$, la congruence n'est pas utile et est rejetée.

Description de la fonction `brison_1`

La c_fonction `brison_1`(n, k, p_m) renvoie une chaîne de caractères donnant une décomposition de n en deux facteurs. Par suite du codage utilisé, le nombre premier p_m doit être inférieur à 2^{15} . Il en est de même du nombre P_0 (P_0 est calculé à partir de p_m). Les noms des variables, des index et du littéral utilisés sont tous déclarés, mais de façon globale (comme beaucoup de mémoire est utilisée, avec des variables locales il faudrait redéfinir les variables d'état de structure du Basic). Les points B1, B2, ... de la description suivante correspondent aux labels `brison_1`, `brison_2`, ... du programme.

B1. Initialisation

Le développement en fraction continue de \sqrt{kn} utilise les variables $d = kn$, $r = \lfloor \sqrt{d} \rfloor$ (calculée par la fonction `intsqr`), $rp = 2 \lfloor \sqrt{d} \rfloor$, $u = \lfloor \sqrt{d} \rfloor + U_{i+1}$, $up = \lfloor \sqrt{d} \rfloor + U_i$, $v = V_i$, $vp = V_{i-1}$, $a = a_i$, $P \equiv p_i \pmod{n}$, $Pp \equiv p_{i-1} \pmod{n}$ (p_i et p_{i-1} désignent dans ces relations les numérateurs des réduites (5.2), et non pas les facteurs de la base B) et l'index $s \equiv i \pmod{2}$. Ces égalités tiennent pour $i \geq 1$; les valeurs initiales, pour $i = 0$, diffèrent de (5.7) pour simplifier le calcul. Au point B2, l'indice $i = \text{iter}$ vaut $i = 0, 1, 2, \dots$ avec une nouvelle valeur à chaque passage.

Dans la congruence

$$x^2 \equiv t \pmod{n}, \quad (5.22)$$

si t est donné par (5.21), la forme factorisée (5.21) est codée à l'aide du polynôme en z :

$$e = s + \alpha_1 z^{p_1} + \alpha_2 z^{p_2} + \cdots + \alpha_m z^{p_m} + z^p, \quad (5.23)$$

dont les coefficients sont les exposants de la factorisation. Les limitations $p_m < 2^{15}$ et $P_0 < 2^{15}$ sont dues au fait que les exposants du polynôme (5.23) doivent être inférieurs à 2^{15} . Avec ce codage, au produit de deux factorisations de la forme (5.21) est associé la somme des polynômes (5.23) correspondants. La condition que la factorisation (5.21) soit un carré s'exprime simplement par $e \equiv 0 \pmod{2}$. On conserve en mémoire un système de jk congruences indépendantes $x_i^2 \equiv t_i \pmod{n}$. Les variables $\mathbf{x}(i)$ et $\mathbf{E}(i)$, pour $0 \leq i < jk$ donnent x_i et le codage (5.23) de t_i . Le tableau \mathbf{b} permet d'effectuer l'élimination de Gauss sur le système (5.20) (modulo 2) : à chaque congruence i correspond un facteur p_j de la base B apparaissant avec un exposant impair α_j ; cette correspondance est donnée par $i = \mathbf{b}(p_j)$; les facteurs p_k de la base non associés sont caractérisés par $\mathbf{b}(p_k) = -1$. Au début, tous les éléments du tableau \mathbf{b} sont initialisés à -1 par la commande `copy`.

B2. Avancer \mathbf{u} , \mathbf{v} , \mathbf{s}

On calcule les nouvelles valeurs de \mathbf{v} et \mathbf{u} par les formules de récurrence (5.6). D'après l'équation (5.8), on vient de trouver la congruence $\mathbf{P}^2 \equiv (-1)^s \mathbf{v} \pmod{n}$. La décomposition (5.21) de $t = (-1)^s \mathbf{v}$ en facteurs premiers inférieurs à p_m est effectuée par la fonction `prfact`. Le facteur non décomposé ρ est donné par `norm(polym(tf, l))`. Si $\rho > P_0$, aller en B6. Si $\rho \leq P_0$, une congruence vient d'être trouvée. La factorisation (5.21) de t est codée dans la variable \mathbf{e} par un polynôme en z de la forme (5.23). Il est plaisant de remarquer qu'on passe de la sortie de `prfact` au codage (5.23) en échangeant les coefficients et exposants de chaque monôme (mis à part le terme constant s qui donne le signe). Noter que \mathbf{t} représente $|t|$ et que le cas $t = \pm 1$ est traité à part pour éviter une erreur dans la fonction `prfact`.

B3. Traitement de la congruence

La congruence est combinée avec le système de congruences. La combinaison en cours, de la forme (5.22), est donnée par \mathbf{xx} (le nombre x) et \mathbf{e} (le polynôme (5.23) qui code la factorisation de t).

B4. Dépendance linéaire.

Si $\mathbf{e} \equiv 0 \pmod{2}$ (le résidu t est un carré) aller en B5. Si dans t le plus grand facteur d'exposant impair, p_{kf} , est déjà associé à la congruence i (donnée par $i = \mathbf{b}(p_{kf})$), remplacer la congruence (5.22) par son produit avec cette congruence i et reprendre l'étape B4. Sinon, la congruence (5.22), qui est indépendante des précédentes, est associée au facteur p_{kf} et ajoutée au système des congruences, puis on continue en B6.

B5. Essai de factorisation

Ici le résidu t de x^2 est un carré. On forme une de ses racines carrées y en utilisant les exposants codés dans \mathbf{e} . Si $\mathbf{xx} \equiv \pm y \pmod{n}$ continuer en B6. Si $\mathbf{xx} \not\equiv \pm y \pmod{n}$, une factorisation de n est obtenue. Le programme se termine de façon analogue à la procédure `fermat$`, en plaçant entre parenthèses les facteurs non premiers.

B6. Avancer P

La variable `iter` compte le nombre de congruences générées par le développement en fraction continue. Si `v` vaut 1, la méthode échoue. La nouvelle valeur de `P` est calculée par l'équation (5.3), écrite modulo n , puis on continue en B2.

```
brison_1: function$(n, index k, pm)
    index i, mm, P0, jk, j, s, kf
    var np, d, r, rp, u, up, v, vp, P, Pp, a
    var t, tf, l, pc, xx, y
    lit z
    mm=2000
    P0=32749
    index*16 b(P0)
    var E(mm), e, x(mm), iter
    P0=min(P0, pm*prime(pm+1)-1)
    b(0)=-1
    copy b(0), P0, 1, b(1), 1
    iter=0
    jk=0
    d=k*n
    r=intsqr(d)
    rp=2*r
    u=rp
    up=rp
    v=1
    vp=d-r^2
    P=r
    Pp=1
    a=0
    s=0
brison_2: vadd vp, a*(up-u)
    exg v, vp
    a=divr(u, v)
    up=u
    u=rp-modr(u, v)
    s=1-s
    t=v
    e=s
    ift t=1 goto brison_3
    tf=prfact(t, pm)
    l=polymn(tf)
    ift norm(polym(tf, l))>P0 goto brison_6
    for i=1, 1
```

```

        pc=polym(tf,i)
        vadd e,deg(pc)*z^norm(pc)
    next i
brison_3:xx=P
brison_4:pc=mod(e,2)
        kf=deg(pc)
        ift kf=0 ift pc=0 goto brison_5
        if b(kf)>=0
            i=b(kf)
            xx=modr(xx*x(i),n)
            e=e+E(i)
            goto brison_4
        else
            b(kf)=jk
            x(jk)=xx
            E(jk)=e
            jk=jk+1
            goto brison_6
        endif
brison_5:y=1
        for i=1,polymn(e)
            pc=polym(e,i)
            j=deg(pc)
            ift j y=modr(y*mdpwre(j,norm(pc)/2,n),n)
        next i
    endif
    ift xx+y=n goto brison_6
    ift xx=y goto brison_6
    np=gcdr(xx-y,n)
    'print "(iter=";iter;"  jk=";jk;"  timer=";timer;")";
    push$ ""," * "
    for i=1,2
        if prtst(np)
            cadd value,justl$(np)&pop$
        else
            cadd value,("&justl$(np)&")"&pop$
        endif
        np=n/np
    next i
    return
brison_6:iter=iter+1
        ift v=1 return'échec_brison
        Pp=modr(a*P+Pp,n)
        exg P,Pp

```

```
goto brison_2
```

Exemple

Dans la factorisation de 5063, on obtient au point B3 les congruences modulo 5063 :

$$71^2 \equiv -2 \times 11$$

$$427^2 \equiv 61$$

$$925^2 \equiv -2 \times 11.$$

Le programme effectue le produit de la première congruence avec la dernière, ce qui donne $4919^2 \equiv 22^2 \pmod{5063}$. C'est cette congruence qui permet d'obtenir ensuite le facteur $\text{gcdr}(4919 - 22, 5063) = 83$.

```
print brison_1(5063,1,19)
```

Sortie (1545 ms)

```
83 * 61
```

Exemple

Le développement en fraction continue de $\sqrt{2(2^{67} - 1)}$ donne les quotients incomplets :

$$2^{34} - 1, \quad (1, \quad 2^{34} - 2, \quad 1, \quad 2^{35} - 2),$$

avec ensuite répétition des 4 valeurs entre parenthèses. Par suite de ce cycle très court, la méthode échoue (**brison_1** détecte l'échec et renvoie la chaîne vide; *v* vaut 1).

```
print brison_1(2^67-1,2,541);" v=";v
```

Sortie (3065 ms)

```
v= 1
```

Exercice 5.4. $\sqrt{2(2^{67} - 1)}$

Montrer la propriété de la fraction continue de $\sqrt{2(2^{67} - 1)}$ indiquée ci-dessus.

Choix de p_m et k

La rapidité de la méthode de Brillhart et Morrison dépend du choix des paramètres p_m et k , donnés en arguments dans **brison_1**, et aussi de P_0 . En agissant sur ces paramètres, on modifie la taille de la base B et les congruences générées. La partie la plus lente du programme est l'appel **prfact**(t, p_m) pour chaque résidu. Si on augmente p_m et P_0 , le nombre de congruences utiles augmente, et la factorisation est obtenue avec moins d'itérations, mais chaque itération est plus chère. Divers essais nous ont montré que les valeurs de p_m entre 400 et 1000 donnent les meilleurs temps pour factoriser des nombres de 15 à 30 chiffres. Ces temps, qui restent assez voisins pour les divers choix de p_m dans ces limites, ne dépendent pas trop de la valeur de P_0 .

Pour optimiser la valeur de k , nous nous servons de l'analyse de Knuth. En désignant par $f(p^\alpha, kn)$ la probabilité que p^α (p nombre premier) divise le

résidu t (équation (5.21)), on choisit la valeur de k qui maximise

$$F = -\frac{1}{2} \log k + \sum_{p \leq p_m} \sum_{\alpha} f(p^\alpha, kn) \log p. \quad (5.24)$$

En effet, F est la valeur moyenne de \sqrt{n}/ρ , et plus cette valeur est grande, plus on obtiendra de congruences utiles. Pour calculer $f(p^\alpha, kn)$, on suppose que dans l'équation (5.8) réécrite modulo p^α : $t \equiv A^2 - knB^2 \pmod{p^\alpha}$, les restes modulaires A et B du numérateur et dénominateur d'une réduite prennent toutes les valeurs premières entre-elles avec la même probabilité. Nous nous limitons à des petites valeurs de k ($k \leq 23$) sans facteur carré et nous supposons que n n'a pas de petits diviseurs ($\leq k$) autres que 1. Nous avons à envisager seulement les cas suivants. Si p divise k , on a $f(p, kn) = 1/(p+1)$ et $f(p^\alpha, kn) = 0$ si $\alpha > 1$. Si $p > 2$ ne divise pas k , on a $f(p^\alpha, kn) = 2/(p^{\alpha-1}(p+1))$ si kn est un résidu quadratique de p et $f(p^\alpha, kn) = 0$ sinon. Si $kn \equiv 5 \pmod{8}$ nous avons $f(2, kn) = f(4, kn) = 1/3$ et $f(2^\alpha, kn) = 0$ pour $\alpha > 2$. Si $kn \equiv 1 \pmod{8}$ nous avons $f(2, kn) = f(4, kn) = 1/3$ et $f(2^\alpha, kn) = 1/(2^{\alpha-3}3)$ pour $\alpha > 2$. Si $kn \not\equiv 1, 5 \pmod{8}$ nous avons $f(2, kn) = 1/3$ et $f(2^\alpha, kn) = 0$ pour $\alpha > 1$.

La fonction **kmult**

La valeur de k qui rend (5.24) maximum est déterminée par la fonction **kmult**. Les données d'entrée, $\mathbf{pm} = p_m$ et n , doivent être initialisées avant l'appel de **kmult**. Le tableau **p** est rempli avec les m premiers nombres entiers, m étant tel que $p_m = \mathbf{p}(m)$. On envisage seulement les valeurs $k = 1, 2, 3, \dots, \mathbf{km}$ sans facteur carré (après analyse de $\mathbf{y} = \mathbf{prfact}(k)$, les k ayant des facteurs carrés sont sautés). Pour chaque valeur de k , l'expression (5.24), F , est calculée en utilisant la fonction **legendre** pour déterminer si kn est un résidu quadratique de p . Le facteur de $\log p$ est $\sum_{\alpha} f(p^\alpha, kn)$. La plus grande valeur de F est conservée dans **Fm**.

```
kmult:function
    push precision2
    precision 10
    local index i,j,k,m,km
    m=200
    km=23
    local index p(m)
    local var y,F,Fm,np
    j=1
    for i=1,m
        j=prime(j+1)
        p(i)=j
        ift j>pm exit
    next i
    m=i-1
    ift p(m)<>pm err_kmult
```

```

y=phantom
for k=1,km
  for i=1,polymn(y)
    ift deg(polym(y,i))>1 goto kmult_1
  next i
  np=k*n
  select modr(np,8)
  case =1
    F=4~/3
  case =5
    F=2~/3
  case others
    F=1~/3
  endselect
  F=F*log(2)-log(k)/2
  for i=2,m
    if modr(k,p(i))
      ift legendre(np,p(i))=1 F=F+2*p(i)/(p(i)^2-1)*log
        (p(i))
      else
        F=F+log(p(i))/(1+p(i))
      endif
    next i
    if F>Fm
      value=k
      Fm=F
    endif
  kmult_1:y=prfact(k+1)
next k
precision2 pop
return

```

Exemple

Examinons la factorisation du nombre

$$n = \frac{10^{22} + 1}{89 \times 101} = 1052788969 \times 1056689261.$$

La table donne pour les valeurs de $k < 23$, la valeur de l'expression (5.24), F , calculée pour $p_m = 661$ et le temps de factorisation en secondes de n par la fonction `brison_1(n, k, p_m)`. Il apparaît clairement que le temps de calcul est d'autant plus court que F est grand. Ainsi, pour $k = 5$, on obtient la plus grande valeur de F et le meilleurs temps de calcul.

k	F	temps	k	F	temps
1	4.380	257	13	4.184	284
2	4.157	329	14	5.316	141
3	4.588	217	15	3.875	267
5	7.211	91	17	3.631	365
6	4.757	214	19	4.307	176
7	3.962	214	21	4.352	193
10	2.849	554	22	3.152	276
11	3.575	429	23	4.023	262

La procédure **brison**

La factorisation de n par la procédure **brison** n commence, comme dans le programme **pollard**, par l'appel de **dabord_prfact** qui détermine les facteurs premiers de n inférieurs à 2^{16} . La procédure **brison** utilise, indépendamment de n , le 121-ième nombre premier $p_m = 661$ et la valeur maximum possible de P_0 . La fonction **kmult** permet de choisir la valeur de k , mais comme son appel prend de 5 à 10 minutes, la procédure **brison** n'optimise pas k (on pose $k = 1$) si le nombre à factoriser est inférieur à 2^{79} , car pour ces nombres le temps de calcul est en général inférieur à 20 minutes. Ainsi pour le nombre examiné ci-dessus, $n = (10^{22} + 1)/89 \times 101$, la fonction **kmult** demande 331 secondes pour obtenir la valeur optimale $k = 5$, alors que l'appel direct de **brison_1** avec $k = 1$ prend seulement 257 secondes. On peut cependant forcer l'appel de **kmult** avec la forme **brison**($n, 0$), ou au contraire l'inhiber en donnant $k > 0$ dans **brison**(n, k). La boucle **do ... loop** est utile dans le cas peu fréquent où **brison_1** conduit à un échec (on augmente alors la valeur de k).

```
brison:procedure(n)
  char c$
  index k,pm
  dabord_prfact
  ift n=1 return
  pm=661
  if @0=2
    k=@2
    ift k<1 k=kmult
  else
    k=1
    ift n>2^79 k=kmult
  endif
  do
    c$=brison_1(n,k,pm)
```

```
  ift len(c$) exit
  k=k+1
loop
print c$
return
```

Exemple

La factorisation de $10^{37} - 1$, le programme **brison** ayant choisi la valeur $k = 23$, est obtenue en 209761 itérations; 323 congruences indépendantes sont en mémoire à la fin du calcul.

```
brison 10^37+1
print "k=";k;"  iter=";iter;"  jk=";jk
```

Sortie (25902 s)

[illegible]

```
k= 23  iter= 209761  jk= 323
```

Méthode des courbes elliptiques

Récemment Lenstra a découvert une méthode de factorisation utilisant les courbes elliptiques, qui pour les grands nombres est plus efficace que les autres méthodes connues. Nous donnons ici seulement les propriétés des courbes elliptiques utilisées par le programme, et nous renvoyons le lecteur aux livres de Koblitz pour les démonstrations et un exposé plus approfondi. Soit $K = \mathbf{F}_p$ le corps des entiers modulo un nombre premier $p > 3$ (ou $K = \mathbf{Q}$ le corps des nombres rationnels). Soit $a, b \in K$ et $x^3 + ax + b$ un polynôme sans racines multiples. Une courbe elliptique est l'ensemble des points (x, y) avec $x, y \in K$ vérifiant l'équation :

$$y^2 = x^3 + ax + b, \quad (5.25)$$

et d'un point O appelé point à l'infini.

L'ensemble des points d'une courbe elliptique forme un groupe commutatif d'élément neutre O pour la loi de composition (notée $+$) définie comme suit. Si $P = (x_1, y_1)$ et $Q = (x_2, y_2)$ avec $x_1 \neq x_2$, on définit le point $P + Q = (x_3, y_3)$ de la courbe elliptique par :

$$\begin{aligned} x_3 &= \left(\frac{y_2 - y_1}{x_2 - x_1} \right)^2 - x_1 - x_2 \\ y_3 &= -y_1 + \left(\frac{y_2 - y_1}{x_2 - x_1} \right) (x_1 - x_3). \end{aligned} \quad (5.26)$$

Si $P = Q = (x_1, y_1)$ on définit le point $P + Q = 2P = (x_3, y_3)$ par :

$$\begin{aligned} x_3 &= \left(\frac{3x_1^2 + a}{2y_1} \right)^2 - 2x_1 \\ y_3 &= -y_1 + \left(\frac{3x_1^2 + a}{2y_1} \right) (x_1 - x_3). \end{aligned} \quad (5.27)$$

Si $P = (x_1, y_1)$ et $Q = (x_1, -y_1)$ on pose $P + Q = O$. On pose aussi $P + O = O + P = P$.

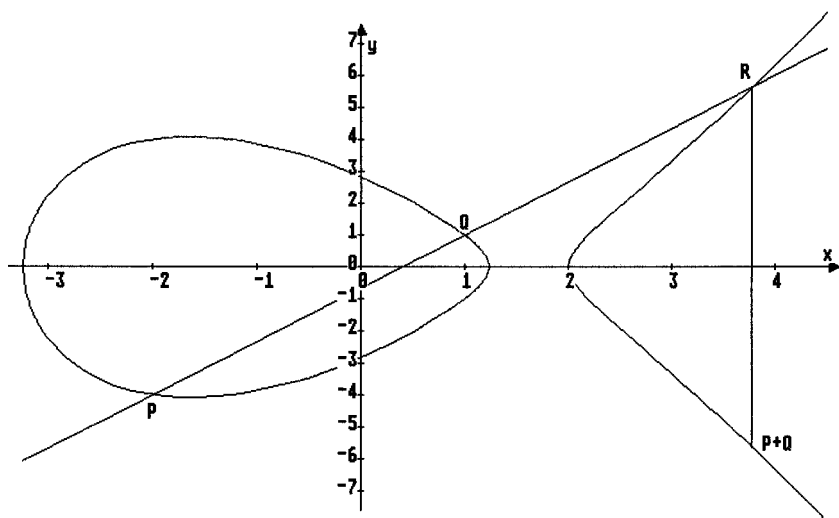


Figure 5.1. La courbe elliptique $y^2 = x^3 - 8x + 8$

La loi de composition correspond à la construction géométrique illustrée sur la figure 1 pour la courbe elliptique $y^2 = x^3 - 8x + 8$ et les points $P = (-2, -4)$, $Q = (1, 1)$. La droite PQ recoupe la courbe elliptique au point $R = (34/9, 152/27)$. La somme $P + Q$ est le symétrique de R par rapport à l'axe Ox . La loi de composition pour divers cas particuliers ($P = Q$, ou $P = O$, ou $Q = O$ ou PQ parallèle à l'axe Oy) s'obtient par passage à la limite, sachant que le point O est le point à l'infini dans la direction Oy .

Pour $K = \mathbf{F}_p$, le nombre de points N de la courbe elliptique est fini et vérifie (théorème de Hasse) :

$$|N - p - 1| < 2\sqrt{p}. \quad (5.28)$$

L'ordre d'un point P de la courbe elliptique est le plus petit entier k tel que $kP = O$ (kP désigne la somme de k fois le point P). D'après le théorème de

Lagrange (voir le chapitre 4), k divise N .

Soit p un facteur premier de l'entier n à factoriser (nous supposons que 2 et 3 ne divisent pas n). Nous ne connaissons pas p , aussi examinons ce qu'il se passe lorsqu'on essaie d'utiliser la loi de composition (5.26) ou (5.27) pour additionner P et Q modulo n . Si le dénominateur $x_2 - x_1$ (ou $2y_1$) est premier avec n , alors on peut calculer son inverse modulo n et obtenir $P + Q \pmod{n}$. Par contre ce calcul échoue si $x_2 - x_1$ n'est pas premier avec n , mais c'est là le cas le plus intéressant, car en général le pgcd de $x_2 - x_1$ et n est alors un facteur non trivial de n . Or on peut montrer que le calcul de $P + Q \pmod{n}$ échoue lorsque $P + Q = O \pmod{p}$. Si en partant d'un point P on calcule divers multiples sP modulo n , on arrivera à un échec lorsque s est un multiple de l'ordre du point P de la courbe elliptique sur \mathbf{F}_p . La méthode de factorisation de Lenstra est basée sur cette remarque.

La méthode de Lenstra

1. Choisir une courbe elliptique et un point P sur cette courbe.
2. Soit C un nombre et $p_1 = 2, p_2 = 3, \dots, p_m$ les nombres premiers inférieurs à un nombre B . Soit $\alpha_i = \lfloor \log C / \log p_i \rfloor$ le plus grand entier tel que $p_i^{\alpha_i} \leq C$ et :

$$s = \prod_{i=1}^m p_i^{\alpha_i}. \quad (5.29)$$

Calculer sP modulo n . Si ce calcul est possible, reprendre à l'étape 1, avec une autre courbe elliptique. Si par contre, au cours du calcul d'une somme $P + Q$ ou $2P$, $\text{gcdr}(x_2 - x_1, n)$ ou $\text{gcdr}(2y, n)$ est différent de 1, alors ce pgcd donne en général un facteur de n . Ce cas se produit si l'ordre N de la courbe elliptique modulo p peut s'écrire comme un produit (5.29). D'après le théorème de Hasse (5.28), cela aura lieu si $p + 1 + 2\sqrt{p} < C$ et si N est un produit de nombres premiers inférieurs à B . Comment choisir une valeur de B qui minimise le temps de factorisation ? Les diverses courbes elliptiques modulo p obtenues en faisant varier a et b ont des ordres N qui se répartissent pratiquement de façon aléatoire entre les limites $p + 1 \pm 2\sqrt{p}$ données par le théorème de Hasse (5.28). On peut en déduire que la probabilité que N soit décomposable en facteurs inférieurs à B est approximativement x^{-x} avec $x = \log p / \log B$. Le choix de la borne $B = \sqrt{C}$, qui donne environ 1/4 pour cette probabilité, est satisfaisant, d'après nos essais, pour des nombres n de 15 à 20 chiffres.

Exercice 5.5. Racine multiple

Montrer que la condition que $x^3 + ax + b$ soit un polynôme sans racines multiples s'écrit $4a^3 + 27b^2 \neq 0$.

Exercice 5.6. Courbe elliptique

Ecrire un programme traçant la courbe elliptique $y^2 = x^3 - 8x + 8$.

La procédure lenstra

La procédure `lenstra`(n [, $C1$ [, a]]) peut être utilisée avec un deuxième argument $C1$ (par défaut $C1 = \lfloor \sqrt{n} \rfloor$, mais en général $C1 = \sqrt[3]{n}$ suffit) qui donne

une borne supérieure des facteurs p de n cherchés. La borne C correspond à cette limite $C1 : C = C1 + 2\sqrt{C1}$. Le troisième argument, a , qui doit être un entier*32, permet d'imposer la première courbe elliptique utilisée ($a = -1$ par défaut). La factorisation de n par la procédure **lenstra** commence, comme dans les programmes **pollard** et **brison**, par l'appel de **dabord_prfact** qui détermine les facteurs premiers de n inférieurs à 2^{16} .

On utilise la courbe elliptique $y^2 = x^3 + ax - a$ et le point $P = (x_1, y_1) = (1, 1)$ de cette courbe. Ultérieurement, si cette courbe n'a pas produit une factorisation, la courbe avec a remplacé par $a + 1$ sera utilisée. Si $4a^3 + 27a^2 \equiv 0 \pmod{n}$, la valeur de a est rejetée (voir l'exercice 5.5.).

Le calcul de sP est effectué en multipliant P , α_1 fois par $p_1 = 2$, puis α_2 fois par $p_2 = 3, \dots$ La procédure **elliptic_mul**(p) remplace le point $P = (x_1, y_1)$ par le point pP . Cela est effectué en $O(\log p)$ opérations en utilisant la multiplication par 2 autant que possible. Par exemple $17P$ s'obtient par $P + 2(2(2(2P)))$ en 4 multiplications par 2 et une addition. La procédure **elliptic_double** remplace le point $P = (x_1, y_1)$ par le point $2P$, et la procédure **elliptic_sum** remplace le point $Q = (x_2, y_2)$ par le point $Q + P$. Dans ces procédures l'inversion du dénominateur x des formules (5.26) et (5.27) est effectuée par la fonction **prinv**(x, n). Si cette fonction renvoie 0, c'est que x et n ne sont pas premiers entre-eux; on effectue alors un retour au label **lenstra_d** du programme principal par l'intermédiaire du label **lenstra_dd** (il faut effectuer deux **return**). La variable **d** contient alors un facteur de n , qui est affiché, et la procédure continue avec n remplacé par n/d , si ce nombre n'est pas premier.

```
lenstra:procedure
  push @1
  if @0>1
    push int(@2)
    if @0=3
      push @3
    else
      push -1
    endif
  else
    push intsqr(stack(0)), -1
  endif
  local datai pop index a,m
  local datav pop,pop var C1,n
  local var np,l,k,pm
  local index i
  dabord_prfact
  ift n=1 return
  local char c$
  local var B,C,d,x1,y1,x2,y2,x,x0,LC,p
lenstra_1:if prtst(n)
```

```

        print c$;n
        return
    endif
    C=min(intsqr(n),C1)
    C=C+intsqr(4*C)+2
    LC=log(C)
    B=intsqr(C)
    for a=a,2^31-1
        d=(4*a+27)*a^2
        d=gcd(d,n)
lenstra_d:ift d=n next a
        if d>1
            if prtst(d)
                print c$;d;
            else
                print c$;using "  (#)";d;
            endif
            c$="  *"
            n=n/d
            goto lenstra_1
        endif
        x1=1
        y1=1
        p=2
        repeat
            m=int(LC/log(p))
            for m=1,m
                elliptic_mul p
            next m
            p=prime(p+1)
        until p>B
    next a
elliptic_mul:procedure(p)
    if p=2
        elliptic_double
        return
    else
        x2=x1
        y2=y1
        p=divr(p,2)
        while p
            elliptic_double
            ift odd(p) elliptic_sum
            p=divr(p,2)
        endwhile
    endif
endprocedure

```

```

        wend
        x1=x2
        y1=y2
        return
    endif
lenstra_dd:d=gcdr(x,n)
    return lenstra_d
elliptic_d1:ift y2<>y1 return lenstra_dd
elliptic_double:x=2*y1
    d=prinv(x,n)
    ift d=0 return lenstra_dd
    x=modr((3*x1^2+a)*d,n)
    x0=modr(x^2-2*x1,n)
    y1=modr(-y1+x*(x1-x0),n)
    x1=x0
    return
elliptic_sum:x=x2-x1
    ift x=0 goto elliptic_d1
    d=prinv(x,n)
    ift d=0 return lenstra_dd
    x=modr((y2-y1)*d,n)
    x2=modr(x^2-x1-x2,n)
    y2=modr(-y1+x*(x1-x2),n)
    return

```

Exemple

Les bornes utilisées pour factoriser $n = 2^{79} - 1$, (ou plus exactement $n/2687$ après division par le petit facteur 2687 dans `dabord_prfact`) sont $C = 14998854349$ et $B = 122469$. Le facteur 202029703 est obtenu avec la première courbe elliptique envisagée ($a = -1$) au moment de la multiplication par le nombre premier $p = 4567$.

```
lenstra 2^79-1
```

Sortie (2325 s)

```
604462909807314587353087= 2687 * 202029703 * 1113491139767
```

Stratégie pour factoriser les nombres

Parmi les méthodes décrites dans ce chapitre, celle de Brillhart et Morrison est en général la plus efficace. Mais, avec assez peu d'essais, on s'aperçoit vite qu'il est très avantageux de rechercher les petits facteurs des très grands nombres

avant de lancer cette méthode. Nous avons observé qu'il vaut mieux employer la méthode ρ de Pollard pour cette tâche, plutôt que la méthode des courbes elliptiques, qui est moins rapide en moyenne, malgré de beaux succès. Nous décrivons dans cette section des fonctions **prf**(n), **prfactb**(n) et **prfactb**\$(n), qui factorisent l'entier n , en utilisant une combinaison de plusieurs méthodes, avec l'espoir d'améliorer l'efficacité. Les facteurs de n inférieurs à 2^{16} sont recherchés par la méthode des divisions successives à l'aide de **prfact**. Si le facteur résiduel est composé (cela est testé par **prtst**), on entreprend une recherche par la méthode ρ de Pollard, mais pour un temps limité de 1 minute à 1 heure suivant la taille des nombres. Cela permet en général d'obtenir les facteurs de taille moyenne (jusqu'à 10 chiffres). Ensuite, si nécessaire, la méthode de Brillhart et Morrison est employée jusqu'à la réduction complète en facteurs premiers.

La combinaison des méthodes proposée résulte d'une étude pour des nombres de moins de 35 chiffres environ. Pour plus de chiffres, peut-être aurait-on intérêt à essayer la méthode des courbes elliptiques. Il faut reconnaître qu'il n'est pas très aisé de déterminer la meilleure stratégie, pour des nombres de 30 chiffres et plus, par suite du temps considérable que prend chaque factorisation. Le point délicat est l'estimation du temps qui doit être alloué à la méthode ρ . Nous prenons pour cette valeur la formule $t = 75(\text{intlgn}(n) - 77)$ qui donne un temps en secondes petit devant le temps nécessaire pour factoriser $n > 2^{80}$ par la méthode de Brillhart et Morrison.

Description des programmes

Les fonctions **prfactb** et **prfactb**\$, qui ont une sortie analogue aux fonctions internes **prfact** et **prfact**\$, peuvent être utilisées en lieu et place de ces dernières dans le cas des grands nombres. Ces programmes utilisent seulement des noms de variables locaux, mais comme beaucoup de mémoire est nécessaire, il faut modifier les variables d'état de structure par :

```
s_var 5000
```

```
s_pro 100000
```

préalablement à leurs appels.

La c_fonction **prf** renvoie le résultat de la factorisation sous la forme d'un t_ensemble. A la factorisation $p_1^{\alpha_1} p_2^{\alpha_2} \cdots p_r^{\alpha_r}$, où $p_1 < p_2 < \cdots < p_r$, correspond le t_ensemble formé des r t_éléments **eset**\$(α_i , **mkx**\$(p_i)), pour $1 \leq i \leq r$. Le i ème t_élément a pour type l'exposant α_i et pour valeur le codage machine du nombre premier p_i .

Description de prfactb\$

Le programme transforme l'écriture du polynôme **prfactb**(n) pour améliorer la présentation. Les étoiles sont supprimées, et les signes + sont transformés en * par la fonction **change**\$.

Description de prfactb

Pour comprendre les détails de fonctionnement de ce programme vous devez connaître les codages internes du Basic. La fonction **prfactb** transforme

le `t_ensemble` calculé par `prf` en un polynôme, mais, comme ce polynôme est illégal, il est créé de toutes pièces dans la variable de type `char` `cp`. L'en-tête est formé des mots 0, 1, 0 et $r - 1$ qui indiquent successivement : le type est un polynôme; il y a un seul littéral; le littéral est `phantom`; il y a r monômes. Pour chaque monôme, le premier mot, qui est l'exposant α_i , est suivi du codage de l'entier p_i (obtenu en supprimant par `mid$` les 6 premiers octets du codage machine de l'expression p_i ; l'option `develop` est ici nécessaire). Pour renvoyer le codage `cp` avec le type `var` et non `char`, il n'est pas possible d'utiliser la conversion

```
value=cvx(cp)
```

car, comme l'expression est illégale (par exemple les exposants ne sont pas ordonnés), `cvx` sortirait en erreur. L'assignation de la chaîne `cp` à la variable de type `var`, `value`, est effectuée en traitant `value` avec le type `char` par l'intermédiaire de `charn`.

Description de `prf`

Les petits facteurs premiers inférieurs à $nlim = 2^{16}$, déterminés par `prfact`, sont placés dans le `t_ensemble` `c1`. Reste à factoriser n , élevé à la puissance `en` (après `prfact`, on a `en = 1`). Bien sûr, si $n < 2^{32}$, n est premier; la factorisation est terminée, et on sort au label `prf_1`. Cette sortie est également prise si le test probabiliste `prtst` indique que n est premier. La procédure `prf_q` examine si le facteur n est un carré; si oui n est remplacé par sa racine et `en` est doublé. L'utilité de cette transformation est d'éviter une division par zéro qui apparaîtrait lors du développement en fraction continue de l'entier \sqrt{n} .

Au cours du programme on emploie deux `t_ensembles` `cp` et `cm`. Le `t_ensemble` `cp` contient les facteurs qui ont été indiqués comme étant premiers par le test `prtst`, et le `t_ensemble` `cm`, les facteurs non encore testés. Chaque nouveau facteur premier n^{en} est placé dans `cp` par la procédure `prf_et`, de sorte que les facteurs sont ordonnés dans l'ordre croissant. Cela est réalisé en manipulant le `t_ensemble` à l'aide de `elementy(cp,1)` et `elementv(cp,1)`, qui donnent le type et la valeur du premier `t_élément`, et de `cdr$(cp)` qui ôte le premier `t_élément` du `t_ensemble`.

La partie suivante du programme est presque identique à la procédure `pollard`, mais employée maintenant pendant un temps limité; `timer` ne doit pas dépasser la limite `prf_timer`. Lorsqu'un facteur g est obtenu, il est placé dans le `t_ensemble` `cm`. Si le facteur résiduel n/g , mis dans `n`, est premier, alors ce facteur premier est placé dans le `t_ensemble` `cp`, par la procédure `prf_et`, puis la méthode ρ est interrompue.

La partie du programme qui commence au label `prf_6` applique la méthode de Brillhart et Morrison au nombre n , que l'on sait composé. Cette partie est identique aux programmes `brison` et `brison_1`, mis à part que les noms des labels ont été modifiés (par exemple `prf_b1` au lieu de `brison_1`), et que la fonction `prf_b1` se contente de renvoyer le facteur de n trouvé, sans tester s'il est premier, les deux facteurs étant alors placés dans le `t_ensemble` `cm`. L'entrée en `prf_6` se fait soit après interruption de la méthode ρ , soit à partir des facteurs

du `t_ensemble cm` lorsque ces facteurs sont composés. La boucle sur les facteurs du `t_ensemble cm` est effectuée par la partie du programme `prf_5 ... prf_7`. Le premier `t_élément` du `t_ensemble cm` est ôté par `elementy`, `elementv` et `cdr`\$. Si le facteur est premier, il est placé dans le `t_ensemble cp` par l'appel de `prf_et`. Lorsque le `t_ensemble cm` devient vide, le programme se termine en renvoyant l'union des très petits facteurs `c1` et des grands facteurs `cp`.

'Utilise `kmult` et `legendre` de la bibliothèque `MATH`

```
prfactb$:function$(n)
  value=change$(prfactb(n),"*","","+","* ")
  return
prfactb:function(n)
  push develop
  develop
  local char c,cp
  local index i
  c=prf(n)
  i=elementn(c)
  cp=mkl$(1)&mki$(0)&mki$(i-1)
  for i=1,i
    cadd cp,mki$(elementy(c,i)),mid$(mkx$(elementv(c,i)),
      7)
  next i
  charn(2^15-1-varnum(value))=cp
  develop pop
  return
prf:function$(x)
  local var w,u,p,n,np,l,xp,k,g
  local index i,en,prf_timer,pm,intlim,nlim
  local char c1,cp,cm
  nlim=2^16
  intlim=79
  w=prfact(x,nlim)
  i=polymn(w)
  if i>1
    for i=1,i-1
      u=polym(w,i)
      cadd c1,eset$(deg(u),mkx$(norm(u)))
    next i
  endif
  u=polym(w,i)
  n=norm(u)
  en=deg(u)
  prf_q
  if n<nlim^2
```



```

prf_1:value=c1&eset$(en,mkx$(n))
    return
endif
ift prtst(n) goto prf_1
prf_timer=timer+max(min(75*(intlgn)-77),2000),75)
x=5
xp=2
l=1
k=1
prf_3:g=gcdr(x-xp,n)
    ift g=1 goto prf_4
    ift g=n goto prf_6
    cadd cm,eset$(en,mkx$(g))
    n=n/g
    prf_q
    if prtst(n)
        prf_et
        goto prf_7
    endif
    ift timer>prf_timer goto prf_6
    x=modr(x,n)
    xp=modr(xp,n)
    goto prf_3
prf_4:k=k-1
    if k=0
        ift timer>prf_timer goto prf_6
        xp=x
        l=2*l
        k=1
    endif
    x=modr(x^2+1,n)
    goto prf_3
prf_5:n=elementv(cm,1)
    en=elementy(cm,1)
    cm=cdr$(cm)
    prf_q
    if prtst(n)
        prf_et
    else
prf_6:pm=661
        k=1
        ift intlgn>intlml k=kmult
        do
            np=prf_b1(n,k,pm)

```

```

        ift np exit
        k=k+1
    loop
        cadd cm,eset$(en,mkx$(np),en,mkx$(n/np))
    endif
prf_7:ift cm<>" goto prf_5
    value=c1&cp
    return
prf_q:do
    np=root(n,2)
    ift np=0 return
    n=np
    en=2*en
    loop
prf_et:procedure
    local var ia,a
    local char cpp
    while cp<>"
        a=elementv(cp,1)
        ia=elementy(cp,1)
        if a=n
            cp=cpp&eset$(en+ia,mkx$(n))&cdr$(cp)
            return
        else a<n
            cadd cpp,eset$(ia,mkx$(a))
            cp=cdr$(cp)
        else
            cp=cpp&eset$(en,mkx$(n))&cp
            return
        endif
    wend
    cp=cpp&eset$(en,mkx$(n))
    return
prf_b1:function(n,index k,pm)
    local index i,mm,P0,jk,j,s,kf
    local var d,r,rp,u,up,v,vp,P,Pp,a
    local var t,tf,l,pc,xx,y
    local lit z
    mm=2000
    P0=32749
    local index*16 b(P0)
    local var E(mm),e,x(mm)
    P0=min(P0,pm*prime(pm+1)-1)
    b(0)=-1

```

```

copy b(0),P0,1,b(1),1
jk=0
d=k*n
r=intsqr(d)
rp=2*r
u=rp
up=rp
v=1
vp=d-r^2
P=r
Pp=1
a=0
s=0
prf_b2:vadd vp,a*(up-u)
exg v,vp
a=divr(u,v)
up=u
u=rp-modr(u,v)
s=1-s
t=v
e=s
ift t=1 goto prf_b3
tf=prfact(t,pm)
l=polymn(tf)
ift norm(polym(tf,l))>P0 goto prf_b6
for i=1,1
    pc=polym(tf,i)
    vadd e,deg(pc)*z^norm(pc)
next i
prf_b3:xx=P
prf_b4:pc=mod(e,2)
kf=deg(pc)
ift kf=0 ift pc=0 goto prf_b5
if b(kf)>=0
    i=b(kf)
    xx=modr(xx*x(i),n)
    e=e+E(i)
    goto prf_b4
else
    b(kf)=jk
    x(jk)=xx
    E(jk)=e
    jk=jk+1
    goto prf_b6

```

```

endif
prf_b5:y=1
for i=1,polymn(e)
    pc=polym(e,i)
    j=deg(pc)
    ift j y=modr(y*mdpwr(e,j,norm(pc)/2,n),n)
next i
endif
ift xx+y=n goto prf_b6
ift xx=y goto prf_b6
value=gcdr(xx-y,n)
return
prf_b6:ift v=1 return
Pp=modr(a*P+Pp,n)
exg P,Pp
goto prf_b2

```

Exemple

Le programme factorise les nombres écrits avec les décimales de π . Au début, nous modifions `s_var` et `s_pro` pour allouer suffisamment de mémoire à la fonction `prf`.

```

'adjoindre prfactb, prfactb$ et prf
s_var 5000
s_pro 100000
precision 100
wpi=exact(pi)
precision 10
for chif=1,100
    n=int(wpi*10^(chif-1))
    clear timer
    print justr$(chif,2);n;"=";
    c$=prfactb$(n)
    print using " (temps=#_ s)";timer
    print c$
next chif

...
36 314159265358979323846264338327950288= (temps=55168 s)
    2^4 * 31 * 52468317311183 * 12071772988002892141
37 3141592653589793238462643383279502884= (temps=92561 s)
    2^2 * 3 * 7 * 19 * 8581666150511 * 229374624355487716489
38 31415926535897932384626433832795028841= (temps=33 s)
    31415926535897932384626433832795028841
39 314159265358979323846264338327950288419= (temps=41893 s)
    13 * 19 * 1709 * 102685743415511 * 7247708286803210623

```

```

40 3141592653589793238462643383279502884197= (temps=32 s)
    59 * 53247333111691410821400735309822082783
41 31415926535897932384626433832795028841971= (temps=78722 s)
    3 * 29429 * 780104111 * 456142508343108182574711203
...

```

Exercice 5.7. Suite aliquote 2

Reprendre, en utilisant la fonction `prfactb`, l'étude des suites aliquotes (voir l'exercice 3.2.). On pourra par exemple étudier la suite $A^i(276)$.

Mesure des performances

Pour conclure ce chapitre, nous allons évaluer les performances des programmes étudiés en factorisant n nombres de k chiffres pris au hasard. Le programme ci-dessous concerne la méthode de Brillhart et Morrison; en remplaçant la ligne

```
brison NX
```

par

```
pollard NX
```

ou

```
lenstra NX
```

ou

```
print prfactb$(NX)
```

ou même

```
print prfact$(NX)
```

le lecteur pourra étudier les performances des autres méthodes.

Pour $k = 18, 20, \dots$, on affiche n ($n = 10$), le temps moyen et le temps maximum de factorisation. Ces temps sont écrits en heures, minutes et secondes par la fonction `hms.ms$(x)` à partir du temps en millisecondes x . Comme la procédure `brison` utilise les identificateurs `k`, `t`, etc. de façon globale, nous évitons d'employer ces noms (par exemple k est désigné par `kex`). A chaque valeur de k correspond une ligne d'affichage, dont le numéro est placé dans la pile par `push cursl`. Après chaque factorisation, le curseur est remis sur cette ligne par `cursl stack(0)`, puis l'écran en dessous du curseur est effacé par impression de la séquence `chr$(27)`, "J" avant d'écrire les temps mesurés.

```

var tot,tmoy,tmax
cls
print " k      n      moyenne          maximum"
forv kex in ([18,34,2])
    tot=0

```

```

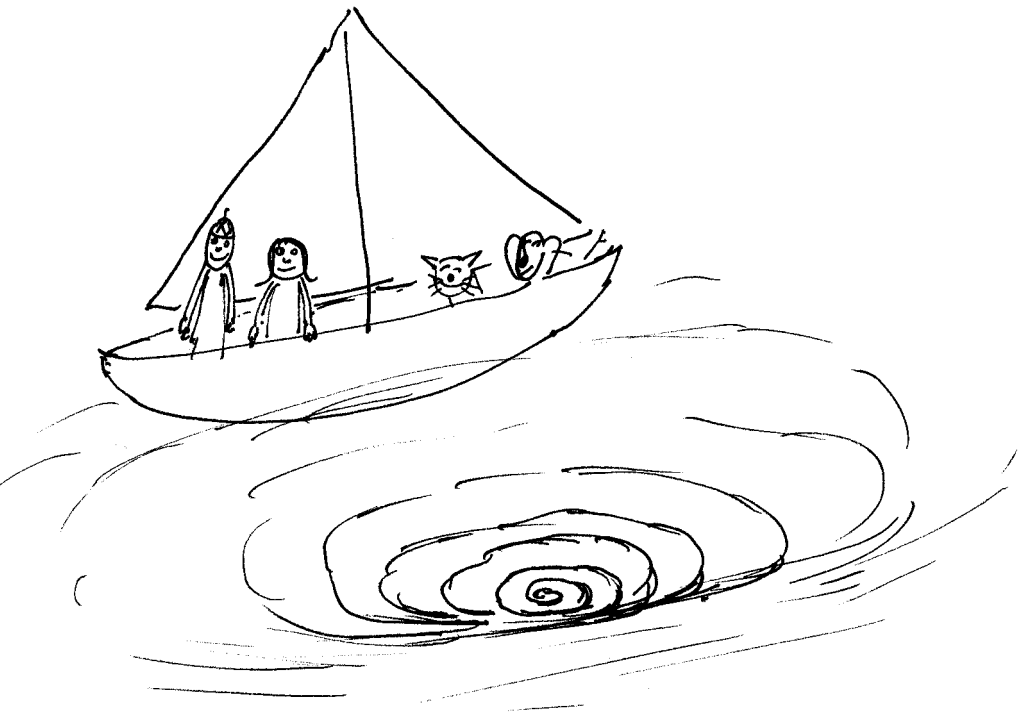
tmax=0
push cursl
nexm=10
for nex=1,nexm
  NX=random(9*10^(kex-1))+10^(kex-1)
  clear timer
  brison NX
  tmx=mtimer
  vadd tot,tmx
  tmax=max(tmx,tmax)
  tmoy=tot\nex
  cursl stack(0)
  print chr$(27)&"J";justl$(kex,2);justl$(nex,4);
  print justl$(hms.ms$(tmoy),16);justl$(hms.ms$(tmax)
    ,16)
next nex
ift pop
nextv
stop
hms.ms$:function$(x)
  local index i
  local char c$
  value="."&justl$(modr(x,1000),3,48)
  x=x\1000
  for i=1,2
    c$=justl$(modr(x,60))
    x=x\60
    value=c$&value
    ift x=0 return
    value=" "&chr$(48,2-len(c$))&value
  next i
  value=justl$(x)&value
return

```

k	n	moyenne	maximum
18	10	1 04.592	4 20.595
20	10	1 18.010	7 47.550
22	10	1 38.803	5 18.185
24	10	5 54.703	35 28.075
26	10	6 29.789	46 25.810
28	10	11 07.734	1 01 36.715
30	10	53 25.296	2 15 44.525
32	10	1 55 13.836	3 54 05.950
34	10	6 39 06.058	23 24 22.950

6

Accélération de la convergence des séries



Il arrive assez souvent d'être confronté au calcul de développements en série qui convergent très lentement, ou même qui divergent. Par exemple, pour calculer $\log(1+x)$ on dispose de la série :

$$\log(1+x) = x - x^2/2 + x^3/3 + \dots \quad (6.1)$$

mais la convergence est trop lente du point de vue pratique pour $x = 1$, et la série diverge pour $x > 1$. Euler a proposé une méthode pour calculer de telles sommes. Si :

$$s = a_1 - a_2 + a_3 - a_4 + \dots \quad (6.2)$$

la transformation d'Euler consiste à calculer s par :

$$s = \sum_{k=0}^{\infty} \frac{(-1)^k \Delta^k a_1}{2^{k+1}}, \quad (6.3)$$

$$\Delta^0 a_i = a_i, \quad \Delta^{k+1} a_i = \Delta^k a_{i+1} - \Delta^k a_i. \quad (6.4)$$

Dans le cas de la série (6.1), le procédé de sommation d'Euler permet d'obtenir 10 chiffres exacts avec 15 termes (voir l'exercice 6.1.). On dit alors que la transformation (6.3) accélère la convergence de la série (6.1). Dans ce chapitre nous considérons quelques méthodes d'accélération de la convergence des séries, qui sont beaucoup plus efficaces et s'appliquent à une plus grande variété de séries que la transformation d'Euler. Les programmes décrits donnent des exemples de calculs en grande précision (constante d'Euler γ , constante de Gompertz) et de calcul formel (détermination d'une approximation rationnelle de $\log(1+x)$). Pour un exposé détaillé des méthodes d'accélération de la convergence nous vous conseillons le livre de Brezinski (1977).

Exercice 6.1. Euler

Calculer la somme $1/8 - 1/9 + 1/10 - 1/11 + \dots$ avec 10 chiffres exacts en utilisant la transformation d'Euler et en déduire la somme $S = 1 - 1/2 + 1/3 - 1/4 + \dots$.

La méthode rho

L'exemple suivant calcule la constante d'Euler γ comme somme de la série $a_1 + a_2 + \dots$ où $a_1 = 1$ et $a_n = 1/n + \log(1 - 1/n)$. La convergence de la série est très lente, aussi utilisons nous la transformation ρ qui est très efficace pour l'accélérer (Bhowmick et al 1989). Le principe de la méthode consiste à calculer les $2p+1$ sommes partielles $s_1, s_2, \dots, s_{2p+1}$, où $s_n = \sum_{k=1}^n a_k$, puis à déterminer une fonction rationnelle $f(x)$ dont le numérateur et le dénominateur

sont des polynômes de degré p et qui passe par les $2p + 1$ points $(x_1, s_1), (x_2, s_2), \dots, (x_{2p+1}, s_{2p+1})$. La valeur de la série est obtenue par la valeur $f(\infty)$ (voir Brezinski). L'algorithme effectuant la transformation ρ consiste à définir les nombres $\rho_k^{(n)}$ à partir de la série a_n par :

$$\begin{aligned} \rho_{-1}^{(n)} &= 0, \quad \rho_0^{(0)} = 0, \quad \rho_0^{(n)} = \rho_0^{(n-1)} + a_n \\ \rho_k^{(n)} &= \rho_{k-2}^{(n+1)} + \frac{k}{\rho_{k-1}^{(n+1)} - \rho_{k-1}^{(n)}} \end{aligned} \quad (6.5)$$

Pour chaque valeur paire de k , les suites $\rho_k^{(n)}$ convergent vers γ .

La transformation ρ est effectuée dans la fonction **rho_serie**, de syntaxe suivante :

rho_serie(N , a)

V_fonction Transformation ρ

N

entier ($N \geq 3$)

a

tableau de variables déclaré par **var**

La fonction **rho_serie** attend en entrée un tableau de variables, contenant les N premières valeurs a_1, \dots, a_N de la série à sommer (la valeur a_i étant placée dans la variable **a(i)** et la variable **a(0)** étant inutilisée). Elle renvoie $\rho_{N-1}^{(1)}$ (N est éventuellement diminué d'une unité s'il n'est pas impair) ainsi qu'une estimation de l'erreur sur la somme de la série. L'estimation de l'erreur, donnée par $|\rho_{N-3}^{(1)} - \rho_{N-1}^{(1)}|$ est ici plutôt pessimiste. Dans le cas étudié, il semble que prendre pour la précision des calculs la valeur N soit un bon choix. Pour $N = 199$ on obtient γ avec 149 chiffres significatifs.

L'utilisation de **log1(x)** au lieu de **log(1 + x)** permet d'améliorer la précision de a_n . En Basic 1000d, il est facile d'estimer l'erreur sur un calcul simplement en refaisant ce calcul avec une précision plus grande. De cette façon, on montre qu'en précision 200, l'erreur sur a_{200} est 10^{-209} , alors que l'erreur serait 10^{-207} si le calcul avait été effectué par $1/200 + \log(1 - 1/200)$. Le calcul des $\rho_k^{(n)}$ est effectué à **kpn** = $k + n$, puis k croissants. Dans la structure **select**, la valeur $\rho_k^{(n)}$ est placée dans **S(k)**. Le calcul de cette valeur utilise $R(k) = \rho_k^{(n-1)}$, $R(k-1) = \rho_{k-1}^{(n)}$, $R(k-2) = \rho_{k-2}^{(n+1)}$ et $S(k-1) = \rho_{k-1}^{(n+1)}$. Avant d'augmenter la valeur de **kpn**, les valeurs **S(0)**, **S(1)**, ..., **S(k)** sont recopiées dans **R(0)**, **R(1)**, ..., **R(k)** respectivement par la commande **copy**. Dans le sous-programme **rho_serie**, **@1** est remplacé par le premier argument de l'appel, et **@0** par le nombre d'arguments. Le nom **a** dans **rho_serie**, défini comme deuxième argument d'appel, est un nom local qui accède au tableau **a** du programme principal (passage d'argument par référence). Si un troisième argument arbitraire

est donné lors de l'appel de `rho_serie`, @0 prend la valeur 3 et la fonction écrit les nombres $\rho_k^{(1)}$ pour k pair, ce qui montre comment converge la série.

```

N=199
precision N
var a(N)
a(1)=1~
for n=2,N
    a(n)=1/n+log1(-1/n)
next n
w=rho_serie(N,a)'ou w=rho_serie(N,a,) pour plus de
détails
print "gamma=";w
print using "erreur=##.###^~^~^~^",ER
stop

rho_serie:function(N, access a(@1))
    ift even(N) N=N-1
    local var R(N-1),S(N-1)
    local index n,k,kpn
    R(0)=a(1)
    for kpn=2,N
        for k=0,kpn-1
            n=kpn-k
            select k
            case=0
                S(k)=R(k)+a(n)
            case=1
                S(k)=1/(S(k-1)-R(k-1))
            case others
                S(k)=R(k-2)+k/(S(k-1)-R(k-1))
            endselect
        next k
        if odd(kpn)
            ER=abs(value-S(k-1))
            value=S(k-1)
            ift @0=3 print kpn;value;using "####.###^~^~^~^",ER
            endif
        copy S(0),k,1,R(0),1
    next kpn
    return

```

Sortie (2465 s)

```

gamma= 0.577215664901532860606512090082402431042159335939923598805767
234884867726777664670936947063291746749514631447249807082480960504014
486542836224173997643065324523824208512714030559157515147009481158916
35~

```

erreur= 0.393- E-148

La transformation de Levin (symbolique)

La transformation ρ est très efficace pour calculer la constante d'Euler γ , mais elle donne de très mauvais résultats sur la série (6.1). En 1973, Levin a introduit une classe de transformations qui permettent une accélération remarquable de la convergence dans un grand nombre de cas, en particulier sur des séries alternées comme (6.1). Nous nous proposons d'écrire des programmes qui effectuent ces transformations, d'abord sur des séries littérales. En entrée, on donnera un développement limité comme par exemple les trois termes à droite de l'équation (6.1). En sortie, on obtiendra une fraction rationnelle en x qui, dans les cas favorables, représente beaucoup mieux la somme de la série.

Nous introduisons la transformation de Levin en suivant les notations de Grotendorst (1989). Soit la suite s_n ($n = 0, 1, \dots$) des sommes partielles :

$$s_n = \sum_{k=0}^n a_k \quad (6.6)$$

qui converge pour $n \rightarrow \infty$ vers la limite s . Examinons d'abord le cas particulier des suites s_n qui vérifient pour tout n l'équation :

$$s_n = s + R(n) \sum_{i=1}^k \frac{c_i}{(n+1)^{i-1}} \quad (6.7)$$

où les c_i ($i = 1, 2, \dots, k$) sont des constantes et où $R(n)$ est une fonction de n , non précisée pour le moment. Si s_n vérifie l'équation (6.7), on peut former un système d'équations linéaires suivant les $k+1$ inconnues s, c_1, c_2, \dots, c_k en écrivant les $k+1$ équations (6.7) pour $n = 0, 1, \dots, k$. La résolution en s du système par la règle de Cramer donne :

$$s = \frac{\begin{vmatrix} s_0 & R(0) & R(0)/1 & \dots & R(0)/1^{k-1} \\ s_1 & R(1) & R(1)/2 & \dots & R(1)/2^{k-1} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ s_k & R(k) & R(k)/(k+1) & \dots & R(k)/(k+1)^{k-1} \end{vmatrix}}{\begin{vmatrix} 1 & R(0) & R(0)/1 & \dots & R(0)/1^{k-1} \\ 1 & R(1) & R(1)/2 & \dots & R(1)/2^{k-1} \\ \vdots & \vdots & \vdots & \ddots & \vdots \end{vmatrix}}. \quad (6.8)$$

Les cofacteurs de la première colonne étant des déterminants de Vandermonde, cette expression se réduit à la formule :

$$s = \frac{\sum_{j=0}^k (-1)^j \binom{k}{j} (j+1)^{k-1} \frac{s_j}{R(j)}}{\sum_{j=0}^k (-1)^j \binom{k}{j} (j+1)^{k-1} \frac{1}{R(j)}}. \quad (6.9)$$

Dans le cas général d'une suite s_n qui converge vers s , la relation (6.7) est seulement approchée et il en est de même de la relation (6.9). Suivant le choix de $R(n)$ on obtient plusieurs transformations de Levin.

- Transformation t :

$$R(n) = a_n$$

- Transformation u :

$$R(n) = (n+1)a_n$$

- Transformation v :

$$R(n) = a_n a_{n+1} / (a_n - a_{n+1})$$

Description du programme

Le mode d'emploi de la fonction **levin** et des procédures **levinpt**, **levinpu** ou **levinpv** est décrit à la manière du manuel de référence.

levin(s , x [, trans])

V_fonction Transformation de Levin *trans* (symbolique)

levinpt s , x , N , D

levinpu s , x , N , D

levinpv s , x , N , D

Procédures Transformation de Levin t , u ou v (symbolique)

s

expr

x

littéral

trans

une des lettres t , u ou v (par défaut u)

N, D

variables déclarées par **var** avant l'appel

Ces programmes effectuent la transformation t , u ou v (c'est la lettre *trans* ou la dernière lettre du nom des procédures) sur le développement limité s suivant le littéral x . Le résultat est soit la valeur de la fonction `levin`, soit la fraction rationnelle N/D , où les polynômes N et D sont renvoyés dans les variables `N` et `D`. Le premier argument s doit être un polynôme généralisé en x :

$$s = x^a \sum_{i=0}^k a_i x^i \quad (6.10)$$

où a est un entier qui peut être négatif et où les coefficients a_i , indépendants de x , peuvent contenir des littéraux. Il est indispensable que tous ces coefficients a_i soient non nuls. La transformation est effectuée à l'ordre le plus grand possible, c'est à dire à l'ordre k (t et u) ou $k-1$ (v).

En exemple, nous partons du développement limité s à l'ordre 16 de $\log(1+x)$ (équation (6.1)). Les trois transformations donnent les fractions rationnelles `T`, `U` et `V`. Ces trois expressions, ainsi que s , sont évaluées pour $x = 1, 2, 3, 4$ et 5 puis comparées à la valeur $\log(1+x)$. La table indique donc l'erreur si l'expression est utilisée pour calculer $\log(1+x)$. Alors que le développement d'origine s converge très mal pour $x = 1$, et diverge pour les autres valeurs, les expressions transformées donnent plus de 10 chiffres exacts de $\log(1+x)$ même pour x plus grand que 1.

Décrivons maintenant le programme `levin`. Le nom de la transformation est mis dans la variable `c$`; s'il n'y a pas 3 arguments d'appel c'est u par défaut; sinon, le troisième argument est placé entre guillemets "`@3`", les espaces éventuels sont supprimés par `justl$`, et la lettre est convertie en minuscules par `lower$`. Ensuite, une des trois procédures `levinpt`, `levinpu` ou `levinpv` est appelée. Ces procédures sont très semblables. Tout d'abord s est multiplié par x^{-a} , puis l'ordre k est obtenu comme étant le degré en x de s . Noter que dans le cas v , l'ordre k est diminué d'une unité pour permettre le calcul de $R(k)$. Le terme a_n de la série est donné par `coeff(s, x, n)x^n`. La valeur de $R(n)$ est calculée à partir de a_n pour les transformations t et u et à partir de la fonction génératrice $R = \sum_{n=0}^k R(n)y^n$ où y est un littéral local, par $R(n) = \text{coeff}(R, y, n)$ pour la transformation v . Le numérateur et le dénominateur de l'expression (6.9) sont calculés dans `N` et `D`. Au cours du calcul, la variable `a1` vaut s_i . Le coefficient du binôme est donné par `ppwr(k, i)/ppwr(i)`. Enfin, le résultat est placé dans `VN` et `VD`, en séparant les numérateurs et dénominateurs de `N` et `D` (qui ne sont pas nécessairement des polynômes).

Pourquoi avons-nous choisi de déterminer séparément le numérateur et le dénominateur de la transformation dans les procédures `levinpt`, `levinpu` et `levinpv` plutôt que d'écrire des fonctions qui renverraient leur quotient ? C'est que la plus grande partie du temps de calcul de `levin` provient du calcul de ce quotient, car le Basic essaie de le simplifier. Ainsi dans l'exemple du développement d'ordre 16 de $\log(1+x)$, la transformation t ou u par la fonction `levin` prend 35 secondes, et sur ces 35 secondes, moins de 5 suffisent pour déterminer le

numérateur et le dénominateur tandis que 28 sont passées à effectuer le quotient. Pour de grandes valeurs de k , il sera donc avantageux (pour raccourcir le temps de calcul ou pour éviter une erreur mémoire) d'utiliser les procédures `levinpt`, `levinpu` et `levinpv`, puis d'utiliser N et D séparément sans former leur quotient.

```
precision 30
notilde
s=slog1(x,15)
print "s=";str$(s,/x)
T=levin(s,x,t)
print "T=";T
U=levin(s,x,u)
V=levin(s,x,v)
print "log      s          t          u          v"
"

for n=1,5
  w=log(n+1)
  print using "##";n+1;
  print using "  ##.## ^^^^^";fsubs(s,x=n)-w;fsubs(T,x=
    n)-w;fsubs(U,x=n)-w;fsubs(V,x=n)-w
next n
stop

levin:function(s,x)
  local char c$
  local var N,D
  c$="u"
  if @0=3
    c$=lower$(justl$("@3"))
  endif
  xqt "levinp"&c$&" s,x,N,D"
  value=N/D
  return

levinpt:procedure(s,x,access VN,VD)
  local char c$
  local index i,k,a
  local var a0,a1,N,D
  a=ordf(s,x)
  s=s*x^-a
  k=degf(s,x)
  a1=0
  for i=0,k
    a0=(-1)^i*ppwr(k,i)/ppwr(i)*(i+1)^(k-1)*x^(k-i)/coeff
      (s,x,i)
    a1=a1+coeff(s,x,i)*x^i
  vadd N,a0*a1
```

```

        vadd D,a0
    next i
    ift a>0 N=N*x^a
    ift a<0 D=D*x^-a
    VN=num(N)*den(D)
    VD=den(N)*num(D)
    a=min(ord(VN,x),ord(VD,x))
    VN=dive(VN,x^a)/cont(VD)
    VD=dive(red(VD),x^a)
    return
levinpu:procedure(s,x,access VN,VD)
    local char c$
    local index i,k,a
    local var a0,a1,N,D
    a=ordf(s,x)
    s=s*x^-a
    k=degf(s,x)
    a1=0
    for i=0,k
        a0=(-1)^i*ppwr(k,i)/ppwr(i)*(i+1)^(k-2)*x^(k-i)/coeff
            (s,x,i)
        a1=a1+coeff(s,x,i)*x^i
        vadd N,a0*a1
        vadd D,a0
    next i
    ift a>0 N=N*x^a
    ift a<0 D=D*x^-a
    VN=num(N)*den(D)
    VD=den(N)*num(D)
    a=min(ord(VN,x),ord(VD,x))
    VN=dive(VN,x^a)/cont(VD)
    VD=dive(red(VD),x^a)
    return
levinpv:procedure(s,x,access VN,VD)
    local lit y
    local char c$
    local index i,k,a
    local var R,a0,a1,N,D
    a=ordf(s,x)
    s=s*x^-a
    k=degf(s,x)-1
    a0=coeff(s,x,0)
    for i=0,k
        a1=coeff(s,x,i+1)*x^(i+1)

```

```

vadd R,y^i/(1/a1-1/a0)
a0=a1
next i
a1=0
for i=0,k
  a0=(-1)^i*ppwr(k,i)/ppwr(i)*(i+1)^(k-1)/coeff(R,y,i)
  vadd a1,coeff(s,x,i)*x^i
  vadd N,a0*a1
  vadd D,a0
next i
ift a>0 N=N*x^a
ift a<0 D=D*x^-a
VN=num(N)*den(D)
VD=den(N)*num(D)
a=min(ord(VN,x),ord(VD,x))
VN=dive(VN,x^a)/cont(VD)
VD=dive(red(VD),x^a)
return

```

Sortie (165 s)

```

s= ( 1)*x+( -1/2)*x^2+( 1/3)*x^3+( -1/4)*x^4+( 1/5)*x^5+( -1/6)*x^6
+( 1/7)*x^7+( -1/8)*x^8+( 1/9)*x^9+( -1/10)*x^10+( 1/11)*x^11+( -1
/12)*x^12+( 1/13)*x^13+( -1/14)*x^14+( 1/15)*x^15+( -1/16)*x^16
T= 1/360360* [x]* [x^15 +491520*x^14 +1506635235*x^13 +488552529920*x
^12 +41656494140625*x^11 +1411965508681728*x^10 +23761545357264715*x^
9 +226411434391633920*x^8 +1324909435029066315*x^7 +5005000000000000
00*x^6 +12544276252755199953*x^5 +21030584449310392320*x^4 +232895813
21411294435*x^3 +16334650033570283520*x^2 +6568408355712890625*x +115
2921504606846976]^~1* [6601157*x^14 +1727237521320*x^13 +362702931332
6475*x^12 +884316292693004080*x^11 +59268561245735449335*x^10 +161547
2521753186376280*x^9 +22095029539421508855185*x^8 +171492511407263483
168160*x^7 +813534218016310627538130*x^6 +2461274432615947435807320*x
^5 +4835728261607918760407322*x^4 +6134586795273281955770160*x^3 +484
1347599698413195211820*x^2 +2159258238364635577489320*x +415466793400
123376271360]

```

log	s	t	u	v
2	-0.30 E-1	-0.38 E-19	0.29 E-18	0.13 E-17
3	-0.27 E+4	0.54 E-15	-0.11 E-14	-0.47 E-14
4	-0.20 E+7	-0.20 E-13	-0.13 E-12	-0.29 E-12
5	-0.21 E+9	-0.94 E-12	-0.21 E-12	0.11 E-11
6	-0.79 E+10	-0.56 E-11	0.12 E-10	0.33 E-10

La transformation de Levin (numérique)

La fonction `levin` du paragraphe précédent convient lorsque la série à sommer contient un littéral. C'est par exemple le cas lorsqu'on connaît une fonction $f(x)$ par un développement en série de x (ou de $x - a$ autour du point a), et que l'on cherche un moyen pratique pour évaluer la fonction en plusieurs points, lorsque la série converge lentement. L'expression `T` sortie ci-dessus peut ainsi être utilisée pour calculer $\log(1 + x)$ pour x voisin de zéro.

Il est également possible d'appliquer les transformations de Levin à des séries numériques. Dans ce cas, le calcul de la formule (6.9) est beaucoup plus rapide, pour une même valeur de k . Autrement dit, il devient possible d'utiliser des valeurs élevées de k permettant d'atteindre une grande précision. Les 3 fonctions ci-dessous ont une syntaxe semblable à la fonction `rho_serie`. Avec cette fonction, on dispose ainsi d'un choix de 4 procédés d'accélération de convergence.

levint(N , a)

levinu(N , a)

levinv(N , a)

V_fonctions Transformation de Levin (numérique)

N

entier ($N \geq 3$)

a

tableau de variables déclaré par **var**

Les fonctions `levint`, `levinu` et `levinv` attendent en entrée un tableau de variables contenant les N premières valeurs a_1, \dots, a_N de la série à sommer. Noter que dans ce paragraphe, pour assurer la compatibilité avec `rho_serie`, le premier terme de la série n'est pas noté a_0 comme précédemment, et que donc l'élément `a(0)` du tableau est inutilisé. Les programmes effectuent en flottant le calcul du numérateur A et du dénominateur B de la formule (6.9). La variable s vaut $\sum_{j=1}^i a_j$ et la variable b vaut $(-1)^i \binom{N-1}{i}$ dans la boucle sur i . Il aurait été possible de remplacer les 12 dernières lignes de `levinu` par un `goto` renvoyant sur les 12 dernières lignes identiques de `levint`, et les 7 dernières lignes de `levinv` par un `goto` renvoyant sur les 7 dernières lignes identiques de `levint`. Le `next` dans cette partie commune renverrait sous le `for` de `levint` ou `levinv` selon le cas. Comme ces simplifications rendraient les programmes moins lisibles, en particulier en détruisant l'indentation, nous ne les avons pas effectuées.

Nous appliquons ces transformations au calcul de la série divergente $\sum_{i=0}^{\infty} (-1)^i i!$. Cette série représente la constante de Gompertz :

$$\int_0^{\infty} \frac{e^{-t}}{1+t} dt = 0.59634 \dots \quad (6.11)$$

C'est ce qu'on obtient en effet, quand on développe $(1+t)^{-1} = 1 - t + t^2 - t^3 + \dots$, en intégrant terme à terme, à l'aide de

$$\int_0^{\infty} (-t)^i e^{-t} dt = (-1)^i i! \quad (6.12)$$

sans s'occuper des questions de convergence. De la même façon que dans la section précédente, où nous arrivions à resommer les séries divergentes donnant $\log 3$, $\log 4$, \dots , ici également nous sommes capables de resommer cette série. Nous utilisons $N = 50$ termes (jusqu'au terme $a_{50} = -49! \approx -3 \times 10^{64}$) et une précision de 100 chiffres. L'erreur sur le résultat obtenu est estimée par comparaison avec la transformation effectuée sur 49 termes. La valeur de l'intégrale (6.11) est ainsi calculée avec 24 chiffres significatifs.

```

N=50
local var a(N)
precision 100
format 26
notilde
p=1
for i=1,N
  a(i)=p
  vmul p,-i
next i
tr t
tr u
tr v
print "\TSortie (";justl$(mtimer-25);" ms)"
stop
tr:v=levin@1(N,a)
print "Levin @1=";v;
print using "  err=##.#### ~~~~~";abs(v-levin@1(N-1,a)
)
return
levint:function(N, access a(@1))
local index i,d
d=N-2
local var b,c,s,A,B
s=0
b=1~
for i=1,N

```

```

        c=b*i^d/a(i)
        vadd s,a(i)
        vadd A,c*s
        vadd B,c
        b=-b*(N-i)/i
    next i
    value=A/B
    return
levinu:function(N, access a(@1))
    local index i,d
    d=N-3
    local var b,c,s,A,B
    s=0
    b=1~
    for i=1,N
        c=b*i^d/a(i)
        vadd s,a(i)
        vadd A,c*s
        vadd B,c
        b=-b*(N-i)/i
    next i
    value=A/B
    return
levinv:function(N, access a(@1))
    N=N-1
    local index i,d
    local var b,c,s,A,B
    d=N-2
    s=0
    b=1~
    for i=1,N
        c=b*i^d*(1/a(i+1)-1/a(i))
        vadd s,a(i)
        vadd A,c*s
        vadd B,c
        b=-b*(N-i)/i
    next i
    value=A/B
    return

```

Sortie (27915 ms)

Levin t=	0.5963473623231940743410740	err= 0.9601	E-24
Levin u=	0.5963473623231940743410746	err= 0.1655	E-22
Levin v=	0.5963473623231940743410758	err= 0.2588	E-22

Comparaison des méthodes

Pour comparer les 4 procédés d'accélération de convergence, nous les appliquons à des séries dont la somme est connue. Les fonctions `rho_serie`, `levint`, `levinu` et `levinv` doivent être adjointes à la suite du programme (ou placées dans la bibliothèque). La procédure `test` N, p, a_i, s permet d'effectuer commodément divers essais. Elle traite la série (de somme s) dont le terme général a_i est donné comme une expression en `i` dans le troisième argument. Les transformations accélératrices utilisent N termes de la série et effectuent les calculs en précision p . Pour la série de départ, tronquée à N termes, et pour chaque transformation, la procédure `test` affiche l'erreur et le temps de calcul.

Nous considérons d'abord la série (équation (6.1) pour $x = 1$) :

$$\log 2 = 1 - \frac{1}{2} + \frac{1}{3} - \frac{1}{4} + \dots \quad (6.13)$$

Avec $N = 50$ termes, la série (6.13) ne donne $\log 2$ qu'à 10^{-2} près. Les transformations de Levin, appliquées à ces 50 termes, permettent d'obtenir $\log 2$ avec 60 chiffres exacts, la meilleure précision étant obtenue avec la transformation t . La transformation ρ n'est pas accélératrice de la série (6.13), puisque elle donne une somme moins bonne que la série initiale. Ce résultat nous permet d'attirer votre attention sur le fait qu'une méthode d'accélération de convergence n'est pas tout le temps accélératrice, mais peut aussi dégrader la convergence de certaines séries.

La deuxième série étudiée est :

$$\frac{\pi^2}{6} = 1 + \frac{1}{2^2} + \frac{1}{3^2} + \frac{1}{4^2} + \dots \quad (6.14)$$

La transformation ρ donne la meilleure évaluation de la série, mais si on tient compte des temps de calcul, la transformation u est la triomphatrice de cette épreuve.

La troisième série étudiée est :

$$4 = 1 + \frac{2}{2} + \frac{3}{2^2} + \frac{4}{2^3} + \frac{5}{2^4} + \dots \quad (6.15)$$

Pour cette série, l'équation (6.7) est exacte dans le cas des transformations t et u pour $k \geq 3$. En effet, soit :

$$s_n = \sum_{k=0}^n (k+1)x^k = \frac{1}{(1-x)^2} - \frac{(n+1)x^{n+1}}{1-x} - \frac{x^{n+1}}{(1-x)^2}. \quad (6.16)$$

La série (6.15) s'obtient par passage à la limite $n \rightarrow \infty$ et en posant $x = 1/2$ dans l'équation (6.16). Pour la transformation t , $R(n) = a_n = (n+1)x^n$ conduit

à

$$s_n = s + R(n) \left[-\frac{x}{1-x} - \frac{x}{(n+1)(1-x)^2} \right], \quad (6.17)$$

qui est identique à l'équation (6.7) pour $c_1 = -x/(1-x)$, $c_2 = -x/(1-x)^2$, $c_3 = \dots = c_k = 0$. Pour la transformation u , comme $R(n) = a_n(n+1) = (n+1)^2 x^n$ on a

$$s_n = s + R(n) \left[-\frac{x}{(n+1)(1-x)} - \frac{x}{(n+1)^2(1-x)^2} \right], \quad (6.18)$$

qui est un cas particulier de l'équation (6.7) pour $c_1 = 0$, $c_2 = -x/(1-x)$, $c_3 = -x/(1-x)^2$, $c_4 = \dots = c_k = 0$. Il s'en suit que pour $N = 5$, ces méthodes ne sont limitées que par la précision des calculs.

Pour la dernière série :

$$\frac{\pi}{4} = 1 - \frac{1}{3} + \frac{1}{5} - \frac{1}{7} + \frac{1}{9} + \dots \quad (6.19)$$

les meilleurs résultats sont obtenus avec la transformation v .

```
test 50,100,-(-1)^i/i,log(2)
test 50,100,i^-2,pi^2/6
test 5,1000,2*i/2^i,4
test 50,100,-(-1)^i/(2*i-1),pi/4
stop
test:procedure(N)
  local var a(N)
  precision @2
  notilde
  for i=1,N
    a(i)=@3
  next i
  w=@4
  c$="\.....\ ##.## ~~~~~_ ##### ms"
  print justl$("@3",24);"erreur          temps"
  clear timer
  print using c$;"série";sum(i=1,N of a(i))-w;mtimer
  clear timer
  print using c$;"levin t";levint(N,a)-w;mtimer
  clear timer
  print using c$;"levin u";levinu(N,a)-w;mtimer
  clear timer
  print using c$;"levin v";levinv(N,a)-w;mtimer
  clear timer
  print using c$;"rho";rho_serie(N,a)-w;mtimer
  print
  return
```

Sortie

$-(-1)^i/i$	erreur		temps
série	-0.99	E-2	225 ms
levin t	0.20	E-61	4860 ms
levin u	-0.13	E-60	4885 ms
levin v	-0.65	E-60	4275 ms
rho	0.30	E-1	1059120 ms

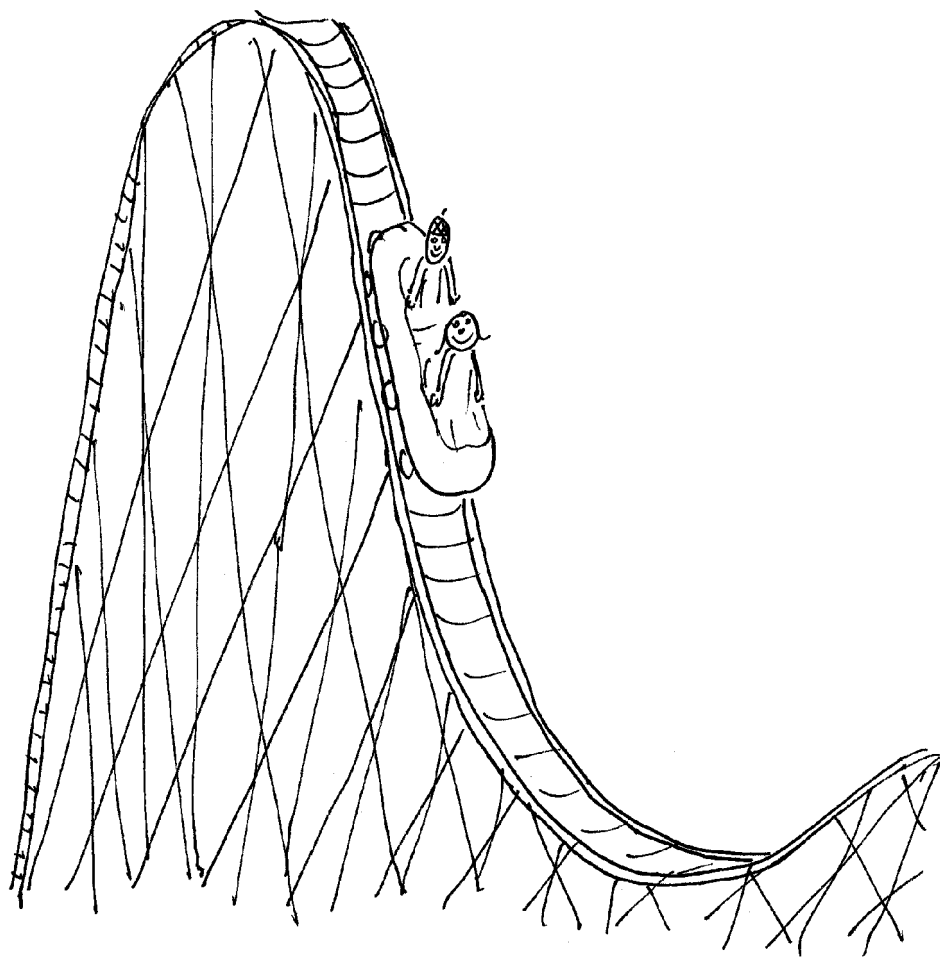
i^{-2}	erreur		temps
série	-0.20	E-1	240 ms
levin t	-0.78	E-3	4965 ms
levin u	0.97	E-46	5015 ms
levin v	0.53	E-44	4360 ms
rho	-0.37	E-50	61140 ms

$2*i/2^i$	erreur		temps
série	-0.44		60 ms
levin t	-0.13	E-1019	10725 ms
levin u	-0.13	E-1019	10750 ms
levin v	0.19	E-1	5830 ms
rho	-0.75		395 ms

$-(-1)^i/(2*i-1)$	erreur		temps
série	-0.50	E-2	245 ms
levin t	-0.16	E-61	4950 ms
levin u	-0.16	E-61	4980 ms
levin v	0.53	E-63	4350 ms
rho	0.16	E-1	1529985 ms

7

Tracé de courbes



Dans ce chapitre, nous décrivons d'abord les procédures **fplot** et **axis** de la bibliothèque MATH avec des exemples de tracés de courbes. Ensuite nous donnons la procédure **qplot**, qui est plus simple à utiliser que les procédures **fplot** et **axis** pour des courbes $y = f(x)$. Pour terminer, nous tracerons des fonctions d'onde de l'atome d'hydrogène pour des grands nombres quantiques. Dans cette application, le Basic 1000d montre sa supériorité sur la plupart des traceurs de courbes qui donnent des résultats déplorables par suite d'une précision insuffisante.

Les procédures **fplot** et **axis**

Voici les procédures **fplot** et **axis** de la bibliothèque MATH.

fplot t1, t2, t3, x0, y0, sx, sy, fx, fy

La procédure **fplot** trace la courbe paramétrée $x = \mathbf{fx}(t)$, $y = \mathbf{fy}(t)$, en utilisant les valeurs $t = t_1, t_1 + t_3, t_1 + 2t_3, \dots$ (sans dépasser t_2). Le pas t_3 peut être négatif. Le point de coordonnées écran **x0, y0** correspond à l'origine mathématique $x = y = 0$ et **sx** (respectivement **sy**) est le nombre de pixels correspondant à l'unité sur l'axe Ox (respectivement Oy).

Dans la procédure **fplot** l'origine graphique est provisoirement modifiée par la commande **origin**, les valeurs précédentes, placées sur la pile par **push**, étant rétablies à la fin de la procédure. Le programme affiche le premier point de la courbe par la commande **plot**, puis des segments de droite par la commande **line to** placée dans une boucle **forv ... nextv**. Les noms des fonctions à tracer donnés en argument de **fplot** sont utilisés directement (passage par nom des arguments @8 et @9). C'est pour éviter des conflits de noms que les variables locales dans **fplot** ont des identificateurs compliqués (**fplot_t1, ...**).

```
fplot:procedure(fplot_t1,fplot_t2,fplot_t3,fplot_x0,fplot_y0,
  fplot_x1,fplot_y1)
  push originy,originx
  origin fplot_x0,fplot_y0
  plot fplot_x1*@8(fplot_t1),-fplot_y1*@9(fplot_t1)
  forv fplot_t1 in [fplot_t1+fplot_t3,fplot_t2,fplot_t3]
    line to fplot_x1*@8(fplot_t1),-fplot_y1*@9(fplot_t1)
  nextv
  origin pop,pop
  return
```


axis x0, y0, x1, y1, x2, y2, dx, dy, x\$, y\$

La procédure **axis** trace des axes *Oxy* (x_0 et y_0 sont les coordonnées écran du point *O*) gradués avec un pas de **dx** ou **dy** pixels. Le tracé est limité au rectangle x_1, y_1, x_2, y_2 . Les noms des axes sont **x\$** et **y\$**.

Les axes sont tracés par la commande **line**, en trait continu (**l_type** 1), d'épaisseur 1 pixel (**l_width** 1) et terminés par une flèche (**l_end** 1, alors que le début des lignes est droit **l_begin** 0). La boucle **forv ... nextv** suivante trace la graduation de l'axe *Ox*. Ce sont des traits de 5 pixels de haut au plus. La valeur à inscrire sous la graduation est placée dans la chaîne **c\$**, et l'appel **vdi(116)** détermine l'extension en pixels d'un rectangle qui encadre cette chaîne (sa hauteur est **ptsout(7)** pixels et sa largeur **ptsout(2)** pixels). La chaîne **c\$** est ensuite affichée par la commande **text**. Le tracé de la graduation de l'axe *Oy* est analogue. Ensuite, après avoir restauré les valeurs initiales des attributs de ligne, le nom des axes est affiché. Remarquer que l'on peut choisir la taille et le style des caractères par **t_height** et **t_type** avant l'appel de la procédure **axis**.

```
axis:procedure(x0,y0,x1,y1,x2,y2,dx,dy,char x$,y$)
  local var x char c$
  push l_type,l_width,l_begin,l_end
  l_type 1
  l_width 1
  l_begin 0
  l_end 1
  line x1,y0 to x2,y0
  line x0,y1 to x0,y2
  l_end 0
  forv x in ([x0,x1,-dx],[x0,x2,dx])
    line x,max(y2,y0-2) to x,min(y1,y0+2)
    c$=justl$(cint((x-x0)/dx))
    vdi §116,c$
    text x-2,y0+ptsout(7),c$
  nextv
  forv x in ([y0,y1,dy],[y0,y2,-dy])
    line max(x1,x0-2),x to min(x2,x0+2),x
    c$=justl$(cint((y0-x)/dy))
    vdi §116,c$
    text x0-ptsout(2)-2,x+2,c$
  nextv
  l_end pop
  l_begin pop
  l_width pop
  l_type pop
  vdi §116,x$
```

```

text x2-5-ptsout(2),y0-5,x$
vdi §116,y$
text x0+5,y2+5+ptsout(7),y$
return

```

La courbe exponentielle

Le programme trace des axes et la courbe $y = e^x$ pour x variant de -3.5 à 2.3 avec un pas de 0.2 . Les fonctions entrées dans `fplot` sont ici les fonctions internes `float` et `exp`. L'origine est le point de coordonnées absolues écran $(384, 370)$. L'unité vaut 120 pixels sur l'axe Ox et 40 pixels sur l'axe Oy . La procédure `axis` trace les axes. Ses arguments d'entrée sont les coordonnées absolues de l'origine $(384, 370)$, une fenêtre de restriction d'affichage ($0 \leq x \leq 639, 399 \geq y \geq 64$), l'unité sur l'axe Ox (120 pixels), l'unité sur l'axe Oy (40 pixels) et les noms des axes. La partie de l'écran sous le menu est effacée par `cls`.

```

cls
axis 384,370,0,399,639,64,120,40,"x","exp(x)"
fplot -3.5,2.3,.2,384,370,120,40,float,exp

```

Sortie (3080 ms)

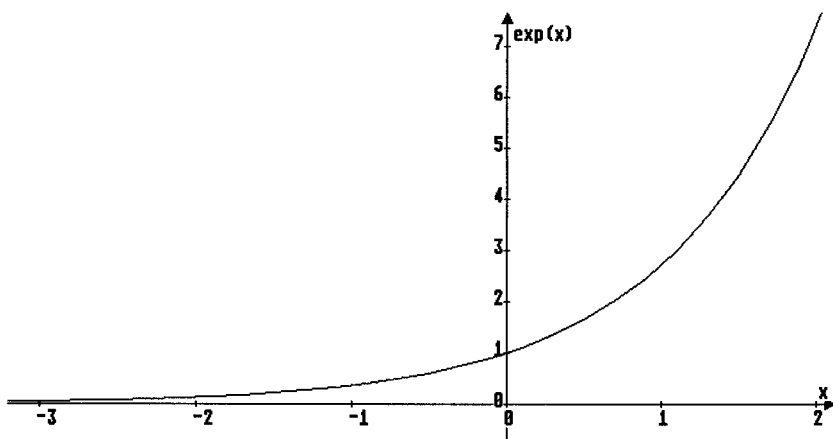


Figure 7.1. La courbe exponentielle

Courbe paramétrée

La procédure `fplot` trace la courbe paramétrée par $x = \sin^2 t$, $y = \sin^3 t$. Les fonctions externes `x` et `y` calculent les coordonnées correspondant à la valeur de t . On fait varier le paramètre t de $-\pi/2$ à $\pi/2$ avec un pas de $\pi/20$. L'origine est le point de coordonnées absolues écran (320, 240). L'unité vaut 150 pixels sur les deux axes. La procédure `axis` trace les axes. Ses arguments d'entrée sont les coordonnées absolues de l'origine (320, 240), une fenêtre de restriction d'affichage ($320 \leq x \leq 520$, $399 \geq y \geq 64$), les unités (150 pixels sur les deux axes) et les noms des axes.

```
axis 320,240,320,399,520,64,150,150,"sin(t)^2","sin(t)^3"
fplot -pi/2,pi/2,pi/20,320,240,150,150,x,y
stop
x:function(t)
  value=sin(t)^2
  return
y:function(t)
  value=sin(t)^3
  return
```

Sortie (3295 ms)

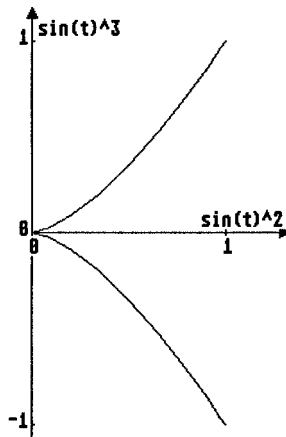


Figure 7.2. Courbe paramétrée

Trajectoire d'un nombre complexe

Le programme trace la trajectoire du nombre complexe $z(t) = 1/(i + t^3)$ lorsque le nombre réel t varie de -4 à 4 avec un pas de 0.1 . Elle s'obtient en traçant la courbe paramétrée par les parties réelles et imaginaires de $z(t)$. Les fonctions `rez` et `imz` calculent les coordonnées correspondant à la valeur de t . L'origine est le point de coordonnées absolues écran $(320, 95)$. L'unité vaut 300 pixels sur les deux axes. La procédure `axis` trace les axes. Ses arguments d'entrée sont les coordonnées absolues de l'origine $(320, 95)$, une fenêtre de restriction d'affichage ($10 \leq x \leq 630$, $399 \geq y \geq 64$), les unités (300 pixels sur les deux axes) et les noms des axes. La partie de l'écran sous le menu est effacée par `cls`.

Lorsque t varie de $-\infty$ à $+\infty$, le nombre complexe $i + t^3$ décrit la droite parallèle à l'axe réel et passant par le nombre complexe i . Le lieu du nombre complexe $z(t)$ s'obtient à partir de cette droite par inversion-symétrie : c'est un cercle. La courbe tracée par le programme est en réalité seulement un arc de ce cercle.

```

cls
complex i
axis 320,95,10,399,630,64,300,300,"x","y"
fplot -4,4,0.1,320,95,300,300,rez,imz
stop
rez:function(t)
  value=re(1~/(i+t^3))
  return
imz:function(t)
  value=im(1~/(i+t^3))
  return

```

Sortie (9185 ms)

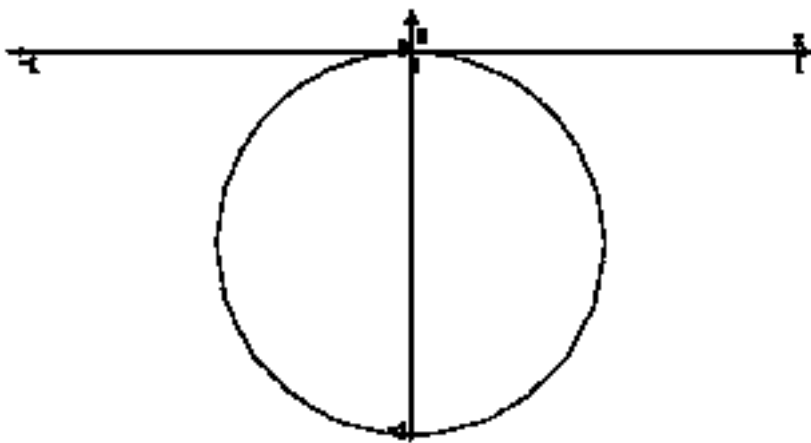


Figure 7.3. Trajectoire d'un nombre complexe

La procédure `qplot`

La procédure `qplot` suivante est beaucoup plus simple à utiliser que les procédures `fplot` et `axis`. Par suite de l'automatisation du choix des échelles et du tracé des axes, on perd en souplesse ce que l'on gagne en facilité de mise en œuvre.

`qplot a, b, N { , fi }`

Procédure Tracé de fonctions

`a, b`

réels

`N`

entier positif

`fi`

nom de fonction (ne commençant pas par `q_`)

La procédure `qplot` permet de tracer plusieurs courbes $y = f_j(x)$ ($j = 0, 1, \dots$) sur le même graphe, simplement en indiquant les valeurs extrêmes a et b de x , et les identificateurs des fonctions à tracer. Les fonctions sont calculées chacune en $N + 1$ points de $[a, b]$. La procédure détermine les échelles horizontale et verticale pour que les courbes soient agrandies le plus possible tout en étant entièrement dans l'écran. Elle trace les axes et leurs graduations.

L'exemple trace les trois courbes $y = x^2 + 1$, $y = x$ et $y = x^3$ pour $-2 \leq x \leq 2$. Chaque courbe est tracée comme une ligne polygonale à 21 sommets.

Nous décrivons le fonctionnement de `qplot` en détail, pour faciliter les modifications comme par exemple le changement des attributs de lignes ou de la taille des lettres et des chiffres (il a été supposé, pour le calage des affichages par `text`, que `t_height` vaut 13). A l'occasion, nous signalerons d'ailleurs des variantes du programme dignes d'intérêt. Les identificateurs des variables locales commencent par la forme exotique `q_` pour éviter des conflits avec les fonctions à tracer (entrées à partir du quatrième argument) qui seront utilisées avec leur propre nom. Les valeurs d'entrée de quelques variables d'état sont poussées dans la pile par `push` avant leur modification; elles seront restaurées en fin du sous-programme par `pop`. La valeur `graphmode` 1 implique que les textes écrits remplacent les bouts de courbes et autres textes placés en dessous. Une modification intéressante consiste à utiliser `graphmode` 2, qui superpose toutes les écritures. Les limites ($a = q_a$ et $b = q_b$) de variation de x sont ordonnées par `exg` si besoin est.

La procédure `qplot` est écrite de sorte qu'il est possible de changer les valeurs de la fenêtre d'affichage $W = (q_wx1, q_wy1, q_wx2, q_wy2)$. Les

valeurs utilisées, qui dépendent de la résolution, correspondent à tout l'écran. On a également donné en remarques un exemple d'autres valeurs avec tracé d'un cadre par `box`. Si on utilise ces valeurs (enlever les marques `'`), rien ne sera écrit en dehors du cadre.

Il est inutile que le nombre de points $q_N = N$ dépasse le nombre de pixels horizontaux $q_wx2 - q_wx1$; q_N est plafonné à cette valeur par `min`. Le pas $c = q_c$ est la distance $x_{i+1} - x_i$ entre les abscisses de 2 points successifs. On calcule les valeurs $q_T(i, j) = f_j(x_i)$ des fonctions $j = 0, 1, \dots$ aux points $x_i = a + i \times c$ ($i = 0, 1, \dots, N$), le tableau de variable q_T ayant été défini avec la taille juste nécessaire. En même temps que les $f_j(x_i)$ on obtient leurs valeurs minimum q_u et maximum q_v qui permettront de déterminer l'échelle verticale. Notons la variante intéressante consistant à initialiser q_u et q_v à 0 au lieu de $\pm 2^{32767}$ (qui représentent $\pm\infty$). Cela force la présence de l'axe $y = 0$ dans le graphe.

La section suivante du programme vise à déterminer où seront écrits les noms des fonctions. On désire qu'ils soient disposés les uns au dessous des autres et près de leur courbe. On cherche ainsi l'indice i qui correspond à une abscisse x_i telle que les $p + 1$ valeurs $f_j(x_i)$ ($j = 0, \dots, p$) soient séparées le mieux possible. Noter que p est donné par q_p . Le nombre maximum de caractères des divers identificateurs de fonction est transformé en unité de variation du premier index du tableau q_T , sachant qu'à une variation d'index de 1 correspondent q_sx pixels horizontaux et qu'un caractère occupe 8 pixels. On obtient ainsi une valeur (mise dans q_in) utilisée pour restreindre la recherche aux abscisses x_i ($q_in \leq i \leq N - q_in$) qui garantissent que les affichages ne dépassent ni à gauche ni à droite du rectangle W . Pour chaque abscisse envisagée, on range les $p + 1$ ordonnées $q_T(i, j)$ ($j = 0, \dots, p$) par la commande `sort`. Les trois premiers arguments de `sort` spécifient les éléments à trier (le troisième argument, $N + 1$, donne la distance entre les éléments). Les valeurs ne sont pas réarrangées, seul l'ordre est renvoyé dans le tableau q_R . Ce classement donne l'intervalle minimum entre ordonnées q_s . Dans la boucle q_i sur les abscisses, on conserve la valeur maximum q_sn de q_s , ainsi que la valeur correspondante q_in de q_i . L'abscisse où écrire les noms des fonctions est, en unités d'index du tableau q_T , la valeur q_in à la sortie de cette boucle.

En outre du rectangle W , défini plus haut, nous introduisons le rectangle $R = (q_x2, q_y2, q_x3, q_y3)$, dans lequel les courbes sont inscrites, et les coordonnées écran $\Omega = (q_x0, q_y0)$ de l'origine mathématique $x = y = 0$. Les deux rectangles R et W sont pris égaux si l'origine Ω ne se trouve pas trop près des bords dans le rectangle W , car on désire disposer de 20 pixels pour annoter les axes. Si $R = W$ entraînerait que Ω soit trop près des bords, la taille du rectangle R est réduite, pour laisser une marge. La procédure `q_1` détermine ainsi les valeurs de q_y2 , q_y3 , q_y0 , et le nombre de pixels q_sy qui correspond à l'unité sur l'axe des y , d'abord avec l'hypothèse $R = W$, c'est-à-dire de sorte qu'au minimum q_u et maximum q_v correspondent respectivement les ordonnées écran q_y3 et q_y2 . Si Ω se trouve alors à moins de 20 pixels au dessus

du bord inférieur q_wy2 de W , la procédure q_1 est rappelée pour remonter le côté inférieur de R de 20 pixels. De façon analogue, le bord supérieur de R peut être abaissé de 20 pixels. Du point de vue de la programmation de la procédure q_1 , nous attirons votre attention sur l'écriture $@2-(@1)$, où on ne peut pas supprimer les parenthèses autour de $@1$, pour que le cas $@1 = q_wy1+20$ soit correctement traité. Les bords gauche et droit de R sont ensuite déterminés.

Vient ensuite le tracé des courbes. L'origine graphique est déplacée par la commande **origin** pour correspondre au point mathématique $(a, 0)$, après avoir sauvegardé les anciennes valeurs. Noter que toutes les coordonnées écran q_wx1, \dots sont données par rapport à cette ancienne origine, supposée en $(0, 0)$ pour définir la fenêtre W . Noter également que la commande **origin** peut provoquer une erreur si **cint**(q_y0) n'est pas un entier*16. Remède : effectuer une translation verticale des fonctions à tracer, ou récrire le programme sans translation d'origine. Le tracé de la ligne polygonale, par **plot** pour le premier point, puis par **line to** pour les N segments, est suivi de l'affichage du nom de la fonction. La valeur q_i vaut -1 si la fonction est croissante à l'abscisse q_in , et 0 sinon (la boucle **while ... wend** lève l'ambiguïté si le point q_in a la même abscisse que le point q_in-1 . Le nom est écrit au point de coordonnées (q_z, q_y1) par la commande **text**, si possible au dessus de la courbe (l'affichage doit rester dans le rectangle W), à gauche si la courbe est croissante et à droite si elle est décroissante. Sinon, l'écriture est faite sous la courbe, mais maintenant à gauche si la courbe est décroissante. Ce processus, répété pour chaque courbe (boucle sur q_p), se termine par le rétablissement de l'ancienne origine (les valeurs sont retirées par **pop** dans l'ordre inverse des valeurs poussées par **push**).

La procédure trace l'axe horizontal d'ordonnée q_y1 et l'axe vertical d'abscisse q_x1 . Ce sont les axes usuels $y = 0$ et $x = 0$ seulement s'ils traversent la fenêtre W . Sinon, d'autres axes coupant R sont choisis.

La commande **line** trace l'axe horizontal dans toute la largeur de la fenêtre R . La valeur $q_sx = q_sx/q_c$ est le nombre de pixels représentant l'unité sur l'axe horizontal et $\nu = q_i$ est un nombre de pixels compris entre 20 et 40 suivant la taille de la fenêtre W . Une nouvelle unité q_f , égale à une puissance de 10, est choisie pour que l'axe puisse être gradué par une suite de nombres entiers multiples de q_f1 (qui peut prendre les valeurs 1, 2 et 5). Cette unité est représentée sur l'écran par μ pixels ($\nu/2 < \mu \leq 5\nu$) : $\mu = q_sx = q_sx \times q_f$. On définit q_f1 de sorte que l'intervalle entre deux graduations successives, $q_f1 \times \mu$, soit compris entre 2ν et 5ν (entre 80 et 200 pixels pour tout l'écran haute résolution). Ensuite, une boucle **do** sur les valeurs q_i de ces graduations est effectuée, avec pour première valeur de q_i le plus petit multiple de q_f1 plus grand ou égal à la borne gauche $x = a$. La graduation est marquée par un petit trait vertical de 6 pixels issu de l'axe à l'abscisse q_z , et la valeur q_i correspondante est affichée en dessous à condition qu'elle puisse être entièrement dans la fenêtre W . Après marquage des graduations, la boucle **do** est terminée par la commande **exit**, puis la valeur de l'unité q_f est affichée (si $q_f \neq 1$), à droite ou à gauche suivant la position de l'axe vertical.

Il n'est pas nécessaire de décrire en détail le tracé de l'axe vertical et de sa graduation qui suivent le même processus que dans le cas de l'axe horizontal, avec la nuance que les graduations sont écrites vers le haut après `t_angle 900`.

```

qplot -2,2,20,F,G,G3
  ift inp(2)
  stop
F:function(x)
  value=x^2+1
  return
G:function(x)
  value=x
  return
G3:function(x)
  value=x^3
  return
qplot:procedure(q_a,q_b,index q_N)
  push format,formatl,formatm,tilde,graphmode
  format -1
  formatl 1
  formatm 1
  notilde
  graphmode 1
  ift q_a>q_b exg q_a,q_b
  local index q_i,q_j,q_p
  ,

  'Fenêtre d'affichage
  local var q_wx1,q_wy1,q_wx2,q_wy2
  q_wx1=0
  q_wy1=0
  q_wx2=639
  q_wy2=399
  ift resolution=0 q_wx2=319
  ift resolution<2 q_wy2=199
  cursh 0
  cls
  ' Exemple de fenêtre d'affichage avec cadre
  ' q_wx1=150
  ' q_wy1=80
  ' q_wx2=480
  ' q_wy2=320
  ' box q_wx1-1,q_wy1-1,q_wx2+1,q_wy2+1
  local var q_c
  q_N=min(q_N,q_wx2-q_wx1)
  q_c=(q_b-q_a)/q_N

```



```

q_p=@0-4
local var q_T(q_N,q_p),q_u,q_v,q_s,q_sn,q_sx,q_sy,q_fl,
  q_f,q_z
local var q_x0,q_x1,q_x2,q_x3,q_y0,q_y1,q_y2,q_y3
local index q_R(q_p),q_in
local char q_c$
,

'Minimum et maximum verticaux
q_u=2^32767
q_v=-q_u
for q_i=0,q_N
  for q_j=4,@0
    q_p=q_j-4
    q_T(q_i,q_p)=@q_j(q_a+q_c*q_i)
    q_u=min(q_u,q_T(q_i,q_p))
    q_v=max(q_v,q_T(q_i,q_p))
  next q_j
next q_i
,

'On écrire les noms ?
q_in=0
for q_j=4,@0
  q_in=max(len("@q_j"),q_in)
next q_j
q_sx=(q_wx2-q_wx1)/q_N
q_in=max(min(gint((8*q_in+10)/q_sx)+1,q_N\3),2)
q_sn=0
if q_p
  for q_i=q_in,q_N-q_in
    sort q_T(q_i,0),q_p+1,q_N+1,q_R(0)
    q_s=2^32767
    for q_j=1,q_p
      q_s=min(q_s,q_T(q_i,q_R(q_j)-1)-q_T(q_i,q_R(q_j-1)
        )-1))
    next q_j
    if q_s>q_sn
      q_sn=q_s
      q_in=q_i
    endif
  next q_i
endif
,

'Fenêtre (q_?2,q_?3) origine (q_?0) et échelles (q_s?)
q_1 q_wy1,q_wy2

```

```

ift cint(q_y0-q_wy2) in [-20,0] q_1 q_wy1,q_wy2-20
ift cint(q_y0-q_wy1) in [0,20] q_1 q_wy1+20,q_wy2
q_f=-q_a/q_c
q_x2=q_wx1
q_x3=q_wx2
q_x0=q_x2+q_f*q_sx
ift cint(q_x0-q_wx1) in [0,20] q_x2=q_wx1+20
ift cint(q_x0-q_wx2) in [-20,0] q_x3=q_wx2-20
q_sx=(q_x3-q_x2)/q_N
q_x0=q_x2+q_f*q_sx
,

'Tracé des courbes
push originy,originx
origin q_x2,q_y0
for q_p=0,@0-4
    plot 0,-q_sy*q_T(0,q_p)
    for q_i=1,q_N
        line to q_sx*q_i,-q_sy*q_T(q_i,q_p)
    next q_i
    'Affiche le nom
    c$="@ (q_p+4) "
    q_i=(q_T(q_in-1,q_p)<q_T(q_in,q_p))
    q_j=q_in-2
    while (not q_i) and q_j and (q_T(q_j+1,q_p)=q_T(q_in,
        q_p))
        q_i=(q_T(q_j,q_p)<q_T(q_in,q_p))
    wend
    q_y1=-q_sy*q_T(q_in,q_p)-3
    q_z=4+q_sx*q_in
    if q_y1+q_y0-13>q_wy1
        ift q_i q_z=q_z-8*len(c$)-8
    else
        q_y1=q_y1+16
        ift not q_i q_z=q_z-8*len(c$)-8
    endif
    text q_z,q_y1,c$
next q_p
origin pop,pop
,

'Coordonnées (q_?1) des axes
q_y1=q_y0
ift q_y1 not in [q_wy1,q_wy2] q_y1=q_wy2-30
q_x1=q_x0
ift q_x1 not in [q_wx1,q_wx2] q_x1=q_wx1+20

```

```
,
'Tracé de l'axe horizontal
line q_x2,q_y1,q_x3,q_y1
q_sx=q_sx/q_c
q_i=max(20,cint((q_wx2-q_wx1)/16))
q_f=10~^lint(log10(5*q_i/q_sx))
q_sx=q_sx*q_f
q_f1=1
ift q_sx<2*q_i q_f1=2
ift q_sx<q_i q_f1=5
q_i=gint(q_a/q_f)
q_i=q_f1*gint(q_i/q_f1)
do
  q_z=cint(q_i*q_sx)+q_x0
  ift q_z>q_x3 exit
  line q_z,q_y1,q_z,q_y1+5
  q_c$=justl$(q_i)
  ift q_z-4*len(q_c$) in [q_wx1,q_wx2-8*len(q_c$)] text
    q_z-4*len(q_c$),q_y1+18,q_c$
  q_i=q_i+q_f1
loop
q_c$=change$("10~"&cint(log10(q_f))," ","")
q_z=q_x3-10-8*len(q_c$)
ift q_x1>q_z-20 q_z=q_x2+10
ift q_f<>1 text q_z,q_y1-3,q_c$
,

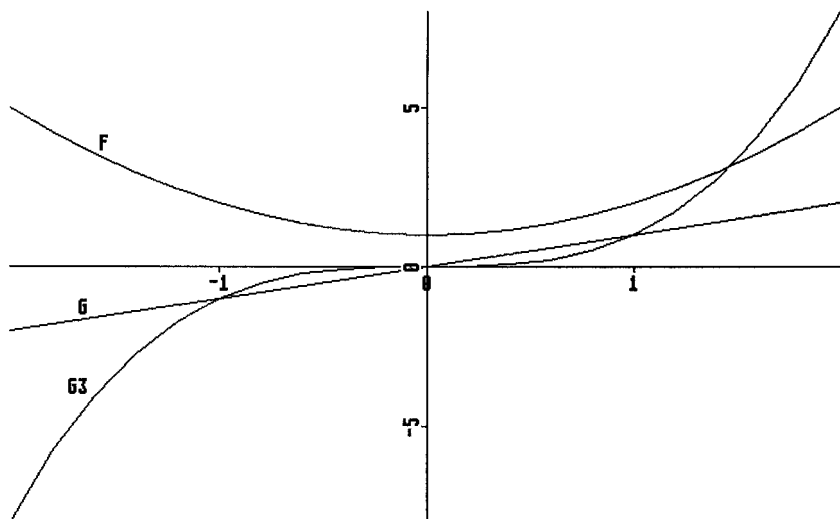
'Tracé de l'axe vertical
line q_x1,q_y2,q_x1,q_y3
q_i=max(20,cint((q_wx2-q_wx1)/16))
q_f=10~^lint(log10(5*q_i/q_sy))
q_sy=q_sy*q_f
q_f1=1
ift q_sy<2*q_i q_f1=2
ift q_sy<q_i q_f1=5
q_i=gint(q_u/q_f)
q_i=q_f1*gint(q_i/q_f1)
t_angle 900
do
  q_z=cint(-q_i*q_sy)+q_y0
  ift q_z<q_y2 exit
  line q_x1-5,q_z,q_x1,q_z
  q_c$=justl$(q_i)
  ift q_z+4*len(q_c$) in [q_wy1+8*len(q_c$),q_wy2] text
    q_x1-7,q_z+4*len(q_c$),q_c$
```

```

    q_i=q_i+q_f1
loop
t_angle 0
q_c$=change$("10~"&cint(log10(q_f))," ","")
ift q_y1-q_wy1<70 q_y2=q_y3-40
q_z=q_x1+5
ift q_z+8*len(q_c$)>q_wx2-5 q_z=q_x1-5-8*len(q_c$)
ift q_f<>1 text q_z,q_y2+20,q_c$
graphmode pop
tilde pop
formatm pop
formatl pop
format pop
hidecm
return
q_1:q_sy=(@2-(@1))/(q_v-q_u)
q_y0=@1+q_v*q_sy
q_y2=@1
q_y3=@2
return

```

Sortie (6430 ms)

Figure 7.4. Les courbes $F(x) = x^2 + 1$, $G(x) = x$ et $G_3(x) = x^3$

Fonctions d'onde de l'hydrogène

Nous nous proposons de calculer les fonctions d'onde radiales $R(r)$ de l'atome d'hydrogène et de tracer leur graphe. Nous utilisons les expressions de ces fonctions données au §36 du livre de Mécanique Quantique de Landau et Lifchitz en unités atomiques : l'unité de longueur est le rayon de Bohr ($4\pi\epsilon_0\hbar^2/me^2 = 0.529 \times 10^{-8}$ cm) et l'unité d'énergie est le Hartree ($me^4/(4\pi\epsilon_0\hbar)^2 = 27.2$ eV). Pour chaque valeur entière $l \geq 0$ du moment cinétique, il existe des états liés d'énergies quantifiées $E = -1/2n^2$, où le nombre quantique n prend toutes les valeurs entières $n > l$, et des états du continuum d'énergie positive $E = k^2/2$ où k est un nombre réel positif quelconque. Les états liés sont décrits par les fonctions d'onde radiales $R_{nl}(r)$ ($R_{nl}(r)^2 r^2 dr$ donne la probabilité de trouver l'électron à une distance du noyau comprise entre r et $r + dr$) :

$$R_{nl}(r) = \frac{2}{n^{l+2}(2l+1)!} \sqrt{\frac{(n+l)!}{(n-l-1)!}} (2r)^l e^{-r/n} F(-n+l+1, 2l+2, 2r/n). \quad (7.1)$$

Les fonctions d'onde du continuum sont :

$$R_{kl} = \frac{C_k}{(2l+1)!} (2kr)^l e^{-ikr} F\left(\frac{i}{k} + l + 1, 2l + 2, 2ikr\right). \quad (7.2)$$

Dans ces expressions, F désigne la fonction hypergéométrique dégénérée, définie par le développement en série :

$$F(a, c, z) = 1 + \frac{a}{c} z + \frac{a(a+1)}{c(c+1)} \frac{z^2}{2!} + \frac{a(a+1)(a+2)}{c(c+1)(c+2)} \frac{z^3}{3!} + \dots, \quad (7.3)$$

qui converge pour tout z . Le nombre C_k est un facteur de normalisation, que nous prenons égal à 1 pour simplifier. Le choix $C_k = 1$ diffère de celui de Landau et Lifchitz :

$$C_k = \sqrt{\frac{2}{\pi}} k e^{\pi/2k} \left| \Gamma\left(l + 1 - \frac{i}{k}\right) \right|, \quad (7.4)$$

qui correspond à la condition de normalisation des fonctions du continuum :

$$\int_0^\infty R_{k'l'} R_{kl} r^2 dr = \delta(k' - k). \quad (7.5)$$

Pour des r très grands, on dispose du développement asymptotique de R_{kl} :

$$R_{kl} \approx C_k \frac{e^{-\pi/2k}}{kr} \operatorname{Re} \left(\frac{e^{-i(kr - \pi(l+1)/2 + \log(2kr)/k)}}{\Gamma(l+1 - i/k)} G\left(l+1 + \frac{i}{k}, \frac{i}{k} - l, -2ikr\right) \right), \quad (7.6)$$

où

$$G(a, b, z) = 1 + \frac{ab}{1!z} + \frac{a(a+1)b(b+1)}{2!z^2} + \dots \quad (7.7)$$

Le premier programme a pour but d'écrire explicitement les fonctions R_{nl} pour les valeurs $n \leq 3$ du nombre quantique principal n , ce qui nous permettra de retrouver la table donnée par Landau et Lifchitz. Nous voulons écrire le coefficient numérique sous la forme $p/q\sqrt{z}$ où p , q et z sont des entiers, et développer la série hypergéométrique, qui est un polynôme de degré $n - l - 1$. Par exemple, la fonction R_{30} sera écrite sous la forme :

$$R_{30} = \frac{2}{3\sqrt{3}} e^{-r/3} \left(1 - \frac{2}{3}r + \frac{2}{27}r^2 \right). \quad (7.8)$$

La fonction `wf$(n, l)` renvoie dans une chaîne l'écriture désirée. Le coefficient numérique, $W\sqrt{X}$, où W et X sont des nombres rationnels, est réduit en utilisant la fonction précédemment décrite `y = sqr(x)`, qui calcule la partie sans facteur carré, y , de l'entier $x = ys^2$. Cette réduction est effectuée en appliquant `sqr` au numérateur `numr(X)` et au dénominateur `denr(X)` du nombre rationnel X , puis en calculant la racine carrée rationnelle à l'aide de la fonction `root`. Le coefficient est écrit sous la forme $p/q/\text{sqr}(z)$ ou simplement p/q si $z = 1$. Le calcul de la fonction hypergéométrique dégénérée est immédiat à l'aide du développement limité à l'ordre $-a$, qui est identique à la fonction hypergéométrique. Ce développement est donné par la fonction `shyg`.

La procédure `printf(n, l)` écrit la fonction d'onde $R_{nl}(r)$. Elle est utilisée dans le programme principal qui écrit la table désirée. Les résultats obtenus permettent de découvrir une erreur de la table de Landau et Lifchitz (pour $R_{21}(r)$).

```
'adjoindre la fonction sqr
for n=1,3
  for l=0,n-1
    printf n,l
  next l,n
stop
printf:procedure(n,l)
  print using "R( #, # )=&" ; n ; l ; wf$(n,l)
  return
wf$:function$(n,l)
  local index a var W,X,z lit r
  a=-n+l+1
  W=2*n^(1+2)/ppwr(2*l+1)*2^1
  X=ppwr(n+1)/ppwr(n-l-1)
  z=sqr(numr(X))*sqr(denr(X))
  value=W*root(X*z,2)
  ift z<>1 cadd value," / sqr(",justl$(z),")"
  if n=1
```

```

      cadd value," * exp(-r)"
    else
      cadd value," * exp(-r/",justl$(n),")"
    endif
    ift 1 cadd value," * ",justl$(r^1)
    W=shyg(2*r/n,-a,r,a,1,2*1+2,-1,1,-1)
    ift W<>1 cadd value," * (" ,justl$(W),")"
    return

```

Sortie (1365 ms)

```

R(1,0)= 2 * exp(-r)
R(2,0)= 1 / sqr(2) * exp(-r/2) * (-1/2*r +1)
R(2,1)= 1/2 / sqr(6) * exp(-r/2) * r
R(3,0)= 2/3 / sqr(3) * exp(-r/3) * (2/27*r^2 -2/3*r +1)
R(3,1)= 8/27 / sqr(6) * exp(-r/3) * r * (-1/6*r +1)
R(3,2)= 4/81 / sqr(30) * exp(-r/3) * r^2

```

Tracé de la fonction 10s

Nous calculons la fonction d'onde $R_{10s}(r)$ pour $n = 10$, $l = 0$ à l'aide de la fonction `wf$` du programme précédent.

```
print wf$(10,0)
```

Sortie (890 ms)

```

1/5 / sqr(10) * exp(-r/10) * (-1/7087500000000*r^9 +1/15750000000*r^
8 -1/87500000*r^7 +1/937500*r^6 -7/125000*r^5 +21/12500*r^4 -7/250*r^
3 +6/25*r^2 -9/10*r +1)

```

Le programme suivant trace (en 97 s) sur l'écran la densité de probabilité $r^2 R_{10s}(r)^2$. Le calcul du polynôme W est effectué par la substitution flottante `fsubs` dans la fonction `R10s`. La fonction `fsubs` est plus rapide lorsque on donne W sous forme d'un produit de facteurs. En effet, chaque point est calculé en 70 ms, mais si on supprime la commande `factor`, W est sous forme polynomiale et le calcul de chaque point est 5 fois plus lent. La commande `s_var 2000` installe suffisamment de variables locales pour permettre le calcul de la fonction en 640 points dans la procédure `qplot`. La représentation de la courbe $r^2 R_{10s}(r)^2$ est une épreuve difficile pour la plupart des traceurs de courbes. Nous avons observé que de tels logiciels, par suite d'une précision insuffisante, donnent une forme inexacte pour r grand et que les minimums de la courbe ne valent pas toujours zéro. Par contre, le Basic 1000d permet de traiter cette application très simplement.

```

'adjoindre la procédure qplot (avec q_1)
s_var 2000
factor
W=r^2/250*(-1/7087500000000*r^9 +1/15750000000*r^8 -1/8
7500000*r^7 +1/937500*r^6 -7/125000*r^5 +21/12500*r^4
-7/250*r^3 +6/25*r^2 -9/10*r +1)^2
develop

```

```

qplot 0,400,640,R10s
texcopy
ift inp(2)
stop
R10s:function(x)
value=exp(-x/5) * fsubs(W,r=x)
return

```

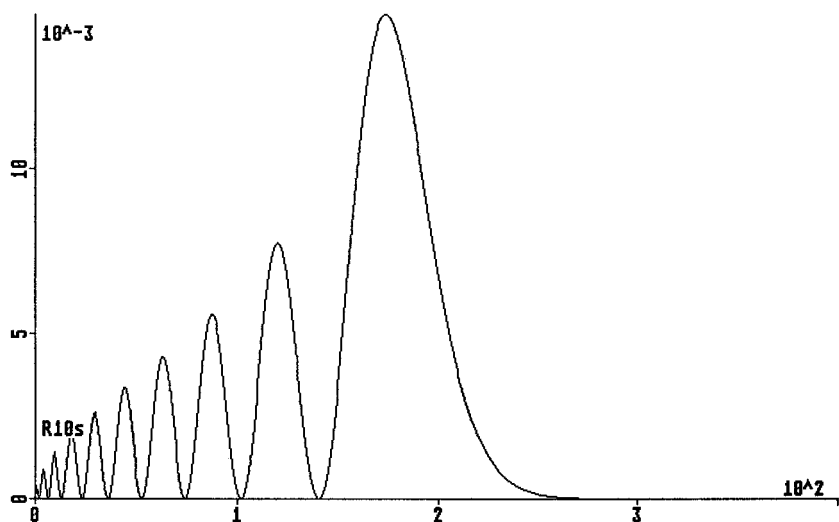


Figure 7.5. Etat 10s

Fonction du continuum

Nous étudions maintenant le calcul de la fonction R_{kl} par la série (7.3), ainsi que la validité de la forme asymptotique (7.6). Nous considérons le cas $l = 0$, $k = 1$ pour des valeurs de r allant jusqu'à 100 ou 200. Examinons le comportement des termes de la série (7.3) lors du calcul de l'expression (7.2) ($z = 2ikr$), par exemple pour $r = 30$. Le module du n ième terme de la série (7.3), qui est de l'ordre de $(2r)^{n-1}/n!$, croît jusqu'à sa valeur maximum $\approx 2 \times 10^{23}$ (obtenue pour $n = 60$), puis décroît ensuite vers 0. Il faut atteindre le 177ième terme pour que son module devienne plus petit que 10^{-10} . La série demande quelques précautions pour être calculée, puisque elle s'obtient par une somme algébrique de grands nombres (de l'ordre de 10^{23} en module) qui se réduit à un petit nombre (de l'ordre de 0.02 en module). Dans le cas $r = 30$, pour obtenir

une dizaine de chiffres significatifs, on faut effectuer les calculs avec une précision de 40 chiffres.

La fonction `hyg(a, c, z)` effectue le calcul de la série (7.3), avec une erreur absolue de 2^{-b} , où $b = \text{precision2}$ ($2^{-b} \approx 10^{-13}$ en précision 10). La valeur W de la série est calculée dans la procédure `hyg_1` : la valeur P du j ème terme est ajoutée à W tant que le module Q de P n'est pas, pour trois fois consécutives, inférieur à la valeur `eps` = ϵ . Le nombre M est la valeur du plus grand module rencontré. La fonction calculant le module d'un nombre complexe, `cabs`, exige que le littéral complexe soit défini et la fonction `abs` exige un argument réel. Pour que `hyg` soit utilisable aussi bien en réel qu'en complexe, on a distingué les deux cas grâce à la variable d'état `complex`. La procédure `hyg_1` est appelée une première fois, avec une faible précision (25 bits), pour déterminer l'ordre de grandeur M des nombres. La procédure `hyg_1` est ensuite appelée à la précision de $b + \text{intlg}(M) + 2$ bits, avec la valeur désirée $\epsilon = 2^{-b}$ de l'erreur absolue. En plus de la valeur de la série, la fonction renvoie le nombre de termes sommés, `hyg_j`, et la précision utilisée, `hyg_prec` (en nombre de bits).

Le programme affiche pour des valeurs de r entre 1 et 200, la précision en bits de la sommation intermédiaire `hyp_prec` (pour $r = 200$, la valeur 606 correspond à un calcul en `precision` 180), le nombre de termes sommés de la série (7.3), la valeur de $rR_{kl}(r)$ calculée par la formule (7.2) en séparant la partie réelle de la partie imaginaire (qui nous fournit une évaluation de l'erreur puisqu'elle est théoriquement nulle) et $rR_{kl}(r)$ calculée par la forme asymptotique (7.6).

Cette dernière valeur est obtenue à l'aide de la fonction `ghyg` qui calcule la série asymptotique $G(a, b, z)$ (7.7). Le module du n ème terme décroît d'abord quand n croît jusqu'à $n \approx |z|$, puis croît ensuite infiniment. Pour évaluer $G(a, b, z)$, on somme la série (7.7) sans aller plus loin que le terme de module minimum (le rapport M de deux termes consécutifs doit rester inférieur à 1 en module); la somme est tronquée plus tôt si le terme général P devient inférieur à $\epsilon = 2^{-\text{precision2}}$.

On observe que pour r grand, la forme asymptotique (de calcul rapide) peut remplacer avantageusement la forme (7.2) (de calcul lent et difficile).

```
complex i
k=1
l=0
'Coefficients
CF=(2*k)^l/ppwr(2*l+1)
CA=exp(-pi/2/k)/k
CA1=exp(i*pi/2*(l+1))/gamma(l+1-i/k)
print "  r  prec. ordre      re(r*Rkl)      im(r*Rkl)
      forme asympt."
notilde
format -9
forv r in ([1,9,1],[10,50,10],100,200)
```

```

W=CF*r^(1+1)*exp(-i*k*r)*hyg(i/k+1+1,2*1+2,2*i*k*r)
print justr$(r,3);justr$(hyg_prec,6);justr$(hyg_j,6);
print using "    ##.#####^~~~_  ##.###^~~~";re(W);i
m(W);
G=ghyg(i/k+1+1,i/k-1,-2*i*k*r)
VA=CA*re(CA1*exp(-i*(k*r+log(2*k*r)/k))*G)
print "    ";VA
nextv
stop
hyg:function(a,c,z)
  index hyg_j,hyg_prec
  local index j,b
  local var W,P,M,eps,Q,Qj
  b=precision2
  precision2 25
  eps=2~^precision2
  hyg_1(a,c)
  eps=2~^-b
  do
    hyg_prec=b+intl(M)
    ift hyg_prec<precision2 exit
    precision2 hyg_prec+2
    hyg_1(a,c)
  loop
  precision2 b
  value=W
  hyg_j=j
  return
hyg_1:procedure(a,c)
  j=0
  W=1~
  P=1~
  M=1~
  Qj=0
  do
    j=j+1
    P=P*z*a/c/j
    W=W+P
    if complex
      Q=cabs(P)
      M=max(cabs(W),Q,M)
    else
      Q=abs(P)
      M=max(abs(W),Q,M)

```

```

endif
if Q<eps
    ift P=0 return
    Qj=Qj+1
    ift Qj=3 return
else
    Qj=0
endif
a=a+1~
c=c+1~
loop
ghyg:function(a,b,z)
    local index j
    local var P,M,eps
    eps=2~^precision2
    j=1
    value=1~
    P=1~
    do
        M=a*b/(j*z)
        ift cabs(M)>1 return
        P=P*M
        value=value+P
        ift cabs(P)<eps return
        a=a+1~
        b=b+1~
        j=j+1
    loop

```

Sortie (1857 s)

r	prec.	ordre	re(r*Rk1)	im(r*Rk1)	forme asympt.
1	43	22	0.20778017	0.111 E-14	0.32228492
2	45	29	-0.23524083	-0.944 E-15	-0.24554557
3	47	36	-0.30182714	-0.158 E-14	-0.30170777
4	49	42	0.78133292 E-1	0.147 E-14	0.78285915 E-1
5	51	48	0.36246326	-0.416 E-14	0.36243319
6	54	54	0.19602061	0.527 E-15	0.19602338
7	57	60	-0.20716801	0.377 E-14	-0.20716798
8	59	65	-0.37333336	0.552 E-14	-0.37333341
9	62	71	-0.11902630	-0.630 E-14	-0.11902629
10	64	77	0.26802073	0.550 E-14	0.26802073
20	92	132	-0.34985101	0.866 E-14	-0.34985101
30	120	186	0.56624285 E-1	0.368 E-13	0.56624285 E-1
40	148	240	0.25780371	0.105 E-12	0.25780371
50	176	295	-0.39394573	-0.253 E-13	-0.39394573

100	319	566	-0.37095351	0.787 E-13	-0.37095351
200	606	1109	-0.34477267	0.266 E-12	-0.34477267

Approximation rationnelle

Nous allons étudier la validité de la transformation de Levin pour le calcul de $R_{kl}(r)$. Si on doit évaluer $R_{kl}(r)$ pour de nombreuses valeurs de r (par exemple pour en tracer la courbe), il est préférable d'effectuer la transformation de Levin en gardant r sous forme littérale. Le programme suivant utilise la procédure `levinpt`, déjà étudiée, pour former, à partir de $N = 15$ termes de la série (7.3), la fraction rationnelle en r , NT/DT, approchant la fonction hypergéométrique, $F(i/k + l + 1, 2l + 2, 2ikr)$ pour $l = 0$, $k = 1$ et r non spécifié. Notons qu'il serait également possible de travailler avec des littéraux k et/ou l , toutefois à un ordre N assez bas, pour obtenir toute une famille d'approximations, et pas seulement l'approximation qui correspond à des valeurs fixées de k et l .

Pour calculer le développement limité (7.3) jusqu'à l'ordre N , W , au lieu d'utiliser :

```
W=shyg(2*i*k*r,N,r,a,1,c,-1,1,-1)
```

nous explicitons la sommation, ce qui fait gagner de la vitesse pour la raison que `shyg` ne tient compte de la simplification $i^2 + 1 = 0$ qu'en fin du calcul, et non à chaque opération. Le programme détermine de même, à partir de $N = 7$ termes de la série (7.7), la fraction rationnelle en $u = 1/r$, NA/DA, approchant la forme asymptotique $G(l + 1 + i/k, i/k - l, -2ikr)$ pour $l = 0$, $k = 1$ et r non spécifié.

Après l'obtention des polynômes NT et DT (en 51 secondes), le calcul des valeurs de NT/DT aux 10 points $x = 1, 2, \dots, 10$, par la fonction de substitution flottante `fsubs`, demande moins de 3 secondes. Il est important de mettre NT et DT sous forme factorisée pour accélérer `fsubs`. Sans cette conversion (à la fin de la procédure `metNTDT`) `fsubs` travaillerait plus lentement (23 secondes pour substituer les 10 valeurs). Le temps total de calcul des 10 valeurs est plus rapide que les 99 secondes de l'évaluation de ces mêmes valeurs par `hyg`.

Nous affichons l'erreur absolue lorsque on remplace le calcul des séries par les approximations NT/DT ou NA/DA. On pourra donc calculer $R_{kl}(r)$ avec une bonne précision en utilisant la forme asymptotique (7.6) pour $r > 4.5$ et la forme (7.2) si $r < 4.5$, les fonctions F et G étant calculées à l'aide de ces approximations rationnelles.

```
'Adjoindre les sous-programmes levinpt, hyg, hyg_1 et g
hyg
complex i
l=0
k=1
var NT,DT,NA,DA
N=15
metNTDT
print "NT=";NT
```

```

print "DT=";DT
print "NT/DT déterminé en";mtimer;" ms"
clear timer
N=7
metNADA
print "NA=";NA
print "DA=";DA
print "NA/DA déterminé en";mtimer;" ms"
N1=10
var P(N1),H(N1),PA(N1),HA(N1)
clear timer
for x=1,N1
    P(x)=F(x)
next x
print N1;" points : (par NT/DT)";mtimer;" ms";
clear timer
for x=1,N1
    H(x)=hyg(i/k+l+1,2*l+2,2*i*k*x)
next x
print "    ou (par hyg)";mtimer;" ms"
clear timer
for x=1,N1
    PA(x)=FA(x)
next x
print "                                (par NA/DA)";mtimer;" ms";
clear timer
for x=1,N1
    HA(x)=ghyg(i/k+l+1,i/k-l,-2*i*k*x)
next x
print "    ou (par ghyg)";mtimer;" ms"
print " r\erreurs      NT/DT                                NA/DA"
for x=1,N1
    print justr$(x,2);using "          ##.####^~~~~~";cabs(H(
        x)-P(x));cabs(HA(x)-PA(x))
next x
stop
metNTDT:a=i/k+l+1
c=2*l+2
y=2*i*k*r
W=1
Wp=1
for j=1,N
    Wp=Wp*y*a/c/j
    W=W+Wp

```

```

        a=a+1
        c=c+1
    next j
    'NT/DT <-- transformation de Levin t
    levinpt W,r,NT,DT
    factor
    NT=NT
    DT=DT
    develop
    return
metNADA:a=i/k+1+1
        b=i/k-1
        y=-u/(2*i*k)
        W=1
        Wp=1
        for j=1,N
            Wp=Wp*y*a*b/j
            W=W+Wp
            a=a+1
            b=b+1
        next j
    levinpt W,u,NA,DA
    factor
    NA=NA
    DA=DA
    develop
    return
F:function(x)
    value=fsubs(NT,r=x)/fsubs(DT,r=x)
    return
FA:function(x)
    value=fsubs(NA,u=1/x)/fsubs(DA,u=1/x)
    return

```

Sortie (178 s)

```

NT= 3143448/5* [213349541289350942976*i*r^14 - ...
DT= [25208068879687925760*i*r^14 + ...
NT/DT déterminé en 53735 ms
NA= -504* [15731607645*i*u^6 -417809079860*i*u^5 -2897781392200*i*u^4
+7761084848400*i*u^3 +6058780043640*i*u^2 -1072105051536*i*u -2369907
58912*i +3907011485*u^6 -546456707730*u^5 +3850945789500*u^4 +6406862
852000*u^3 -5045585440280*u^2 -2174474794848*u +51468304384]
DA= [3744346424000*i*u^6 +102361070366100*i*u^5 -575131610726400*i*u
^4 -4129793850000000*i*u^3 -2301693698304000*i*u^2 +587092604515200*i
*u +119443342491648*i -8357916125*u^7 -3744346424000*u^6 -20472214073

```

```
2200*u^5 -1725394832179200*u^4 -1032448462500000*u^3 +280388141429760
0*u^2 +1023243612652800*u -25940025409536]
```

NA/DA déterminé en 9320 ms

10 points: (par NT/DT) 3425 ms ou (par hyg) 99785 ms
(par NA/DA) 1635 ms ou (par ghyg) 8225 ms

r\erreurs	NT/DT	NA/DA
1	0.5118~ E-15	0.2961~
2	0.5613~ E-15	0.3784~ E-1
3	0.3417~ E-11	0.4742~ E-2
4	0.1194~ E-8	0.5969~ E-3
5	0.1049~ E-6	0.7567~ E-4
6	0.3691~ E-5	0.9622~ E-5
7	0.6704~ E-4	0.1244~ E-5
8	0.7318~ E-3	0.1586~ E-6
9	0.5312~ E-2	0.2037~ E-7
10	0.2756~ E-1	0.3193~ E-8

Tracé de $R_{kl}(r)$

Nous traitons le cas $k = 1$, $l = 0$. Maintenant que nous savons calculer efficacement $R_{kl}(r)$, le tracé de la densité de probabilité $P(r) = r^2 R_{kl}(r)^2$ est assez rapide (275 s pour 640 points entre $r = 0$ et $r = 25$, alors que le calcul de $R_{kl}(r)$ par la fonction `hyg` en ces points prendrait 212 minutes). La fonction `psi(r)` calcule $rR_{kl}(r)$ en utilisant la forme asymptotique (7.6) pour $r > 4.347$ ou la forme (7.2) sinon. Les fonctions F et G sont calculées à l'aide des approximations rationnelles déterminées par le programme précédent. Une étude non présentée ici nous a montré que l'erreur relative sur $rR_{kl}(r)$ est inférieure à 10^{-7} pour tout r , le point de séparation $r \approx 4.347$ minimisant cette erreur. Remarquer la façon d'entrer les données NT, DT, NA et DA, qui optimise le temps : ces polynômes sont décodés en mode `develop` et réel; la transformation en forme factorisée est effectuée en mode réel; on indique ensuite que i est le littéral complexe.

'Adjoindre la procédure `qplot` (avec `q_1`)

```
NT=3143448/5* [213349541289350942976*i*rv^14 -213349698
199952506160*i*rv^13 -511185291583305267552824*i*rv^12
+1015300588887416997445600*i*rv^11 +23043336569054169
3004864520*i*rv^10 -665954574162202096021414060*i*rv^9
-40472708164529231771895900325*i*rv^8 +14880473576198
9133064721123040*i*rv^7 +31928786703790715518009346844
00*i*rv^6 -13952063675042481535476554186100*i*rv^5 -10
7801063671382601686515466894700*i*rv^4 +55756849686639
5309237655366654000*i*rv^3 +10417130577152034717031052
21020800*i*rv^2 -7830873828494723478545545977004500*i*
rv +8047754921196818356819936149504000*i +12297893888*
rv^14 +14401094082594842411040*rv^13 -2160174072419504
```



```

NT=NT
DT=DT
FA=subs(NA,u=1/rv)/subs(DA,u=1/rv)
develop
complex i
k=1
l=0
CF=(2*k)^1/ppwr(2*l+1)
CA=exp(-pi/2/k)/k
CA1=exp(i*pi/2*(l+1))/gamma(l+1-i/k)
qplot 0,25,640,P
ift inp(2)
stop
psi:function(x)
  local datav complex var i
  if x>4.347
    value=fsubs(FA,rv=x)
    value=CA*re(CA1*exp(-i*(x+log(2*x))))*value)
    return
  else
    value=fsubs(NT,rv=x)/fsubs(DT,rv=x)
    value=value*CF*x*exp(-i*x)
    return
  endif
P:function(r)
  value=re(psi(r))^2
  return
  'Adjoindre les sous-programmes hyg (avec hyg_1)
  'et qplot (avec q_1)
  complex i
  k=1
  l=0
  CF=(2*k)^1/ppwr(2*l+1)
  qplot 0,25,640,P
P:function(r)
  value=(Rkl(r)*r)^2
  return
Rkl:function(r)
  value=CF*r^l*exp(-i*k*r)*hyg(i/k+l+1,2*l+2,2*i*k*r)
  value=re(value)
  return

```

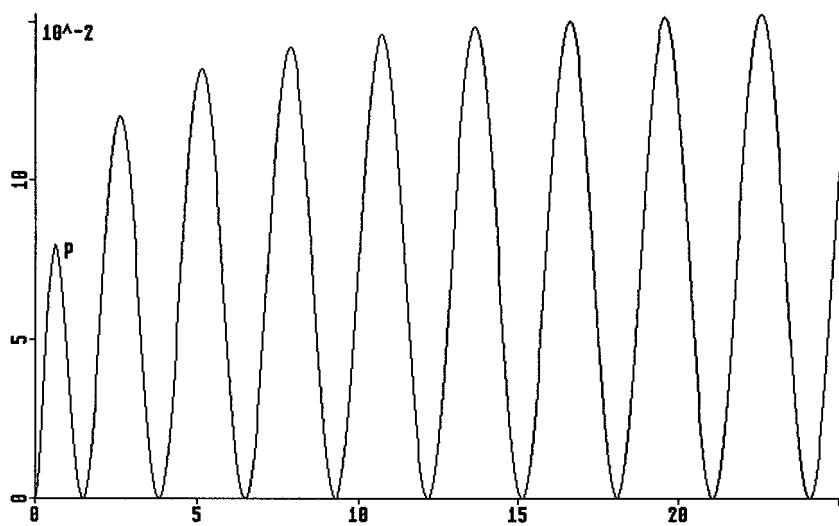
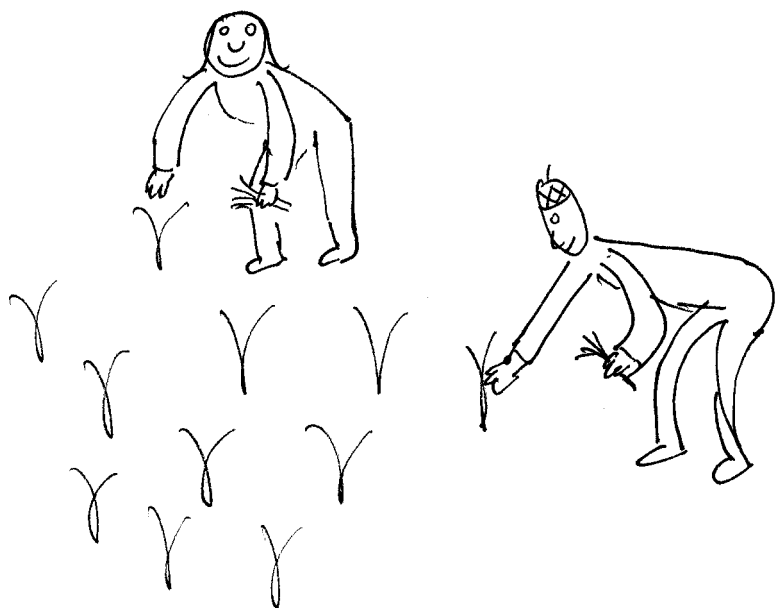


Figure 7.6. Etat du continuum

8

Fonction Gamma



La fonction spéciale $\Gamma(x)$, x étant un nombre réel ou complexe, est disponible dans la bibliothèque MATH. Sa précision est limitée à 166 chiffres et son nom `gamma(x)` doit être écrit en minuscules. Ce chapitre fournit des exemples d'utilisations et une description du fonctionnement de la fonction `gamma`, puis considère certaines questions en rapport avec ce sujet. Comme référence citons le Handbook of Mathematical Functions.

Fonction Gamma et factorielle

Pour n entier, on a $\Gamma(n+1) = n!$, mais `gamma(n+1)` donne une valeur approchée de $n!$ à la différence de `ppwr(n)` qui calcule $n!$ exactement. Pour z réel positif, $\Gamma(z)$ est la valeur de l'intégrale d'Euler :

$$\Gamma(z) = \int_0^{\infty} t^{z-1} e^{-t} dt. \quad (8.1)$$

L'exemple calcule $\Gamma(z)$ pour $z = n+1$ et en $z = 1/2$:

$$\Gamma(1/2) = 2 \int_0^{\infty} e^{-t^2} dt = \sqrt{\pi} \approx 1.772. \quad (8.2)$$

```
for n=1,10
  print using "##_ #####_ #####.#####";n;
  ppwr(n);gamma(n+1)
next n
print "gamma(1/2)=";gamma(1/2); " pi^(1/2)=";pi^(1/2)
```

Sortie (5690 ms)

1	1	1.0000000000~
2	2	2.0000000000~
3	6	6.0000000000~
4	24	24.0000000000~
5	120	120.0000000000~
6	720	720.0000000000~
7	5040	5039.9999999999~
8	40320	40319.9999999984~
9	362880	362879.9999999814~
10	3628800	3628799.9999999311~

```
gamma(1/2)= 0.1772453851~ E+1 pi^(1/2)= 0.1772453851~ E+1
```

Graphe de la fonction Γ

Le programme trace la courbe $y = \Gamma(x)$ pour x variant de -5 à 4.2 . La procédure **axis** trace le système d'axes Oxy . Comme la fonction Γ possède des pôles aux points $x = 0, -1, -2, -3, -4$ et -5 , le tracé est effectué, par la procédure **fplot**, sur les intervalles $[j - 1 + \epsilon, j - \epsilon]$ (où $\epsilon = 0.01$) pour j prenant les valeurs entières de -4 à 0 , puis sur l'intervalle $[\epsilon, 4.2]$, en évitant les pôles qui produisent une erreur dans la fonction **gamma**. La commande **line** trace les asymptotes $x = j - 1$ ($j = -4, \dots, 0$). L'origine est le point de coordonnées absolues écran (320,200). L'unité vaut 60 pixels sur l'axe Ox et 30 pixels sur l'axe Oy .

```

cursh 0
cls
axis 320,200,20,399,585,0,60,30,"x","y"
for j=-4,0
  x0=260+60*j
  line x0,0,x0,399
  fplot j-0.99,j-0.01,0.05,320,200,60,30,,gamma
next j
fplot 0.1,4.2,0.05,320,200,60,30,,gamma
ift inp(2)
stop

```

Sortie (80 s)

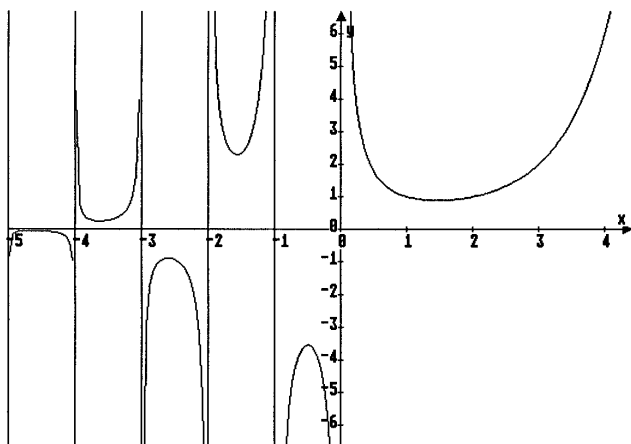


Figure 8.1. La fonction $y = \Gamma(x)$

Fonction Gamma complexe

Le module du nombre complexe $\Gamma(l+1+ia)$, où l est un entier $l \geq 0$ et a un nombre réel, peut s'exprimer à l'aide de fonctions élémentaires. En effet, en utilisant les propriétés de la fonction Γ :

$$\overline{\Gamma(z)} = \Gamma(\bar{z}), \quad \Gamma(z+1) = z\Gamma(z), \quad \Gamma(z)\Gamma(1-z) = \frac{\pi}{\sin \pi z}, \quad (8.3)$$

on a :

$$\begin{aligned} \Gamma(l+1+ia) &= (l+ia) \dots (1+ia)ia\Gamma(ia), \\ \Gamma(l+1-ia) &= (l-ia) \dots (1-ia)\Gamma(1-ia), \\ |\Gamma(l+1+ia)| &= \sqrt{\Gamma(l+1+ia)\Gamma(l+1-ia)} \\ &= \sqrt{\frac{\pi}{a \sinh \pi a}} \prod_{s=0}^l \sqrt{s^2 + a^2}. \end{aligned} \quad (8.4)$$

Le programme suivant calcule $\Gamma(5+3i)$ et vérifie que son module est donné par la formule ci-dessus. Le premier appel de `gamma` est plus long par suite du remplissage d'un tableau de valeurs. Les appels suivants de `gamma` sont plus rapides (par exemple, le calcul de $\Gamma(5+3i)$ prend 1370 ms au premier appel et 640 ms ensuite).

```
complex i
a=3
l=4
g=gamma(l+1+i*a)
print "gamma(";l+1+i*a;")=";g
print "err=";1-sqr(pi/a/sinh(pi*a)*prod(s=0,l of s^2+a^
2))/cabs(g)
```

Sortie (1630 ms)

```
gamma( 3*i +5)= 0.1604188274~ E-1 -i*0.9433293290~ E+1
err= -0.5329070518~ E-14
```

Exercice 8.1. gamma

Vérifier pour des valeurs numériques du nombre réel y les formules :

$$|\Gamma(1/2+iy)|^2 = \frac{\pi}{\cosh \pi y} \quad (8.5)$$

$$\Gamma(1/4+iy)\Gamma(3/4-iy) = \frac{\pi\sqrt{2}}{\cosh \pi y + i \sinh \pi y}. \quad (8.6)$$

Trajectoires complexes

Une façon de représenter graphiquement la fonction $\Gamma(z)$ consiste à tracer la trajectoire dans le plan complexe de $\Gamma(z)$ lorsque $z = x + iy$ décrit un réseau de courbes. Le réseau de départ est formé des segments de droites :

$$\begin{aligned} x = \alpha, \quad 0 \leq y \leq 4 \\ y = \beta, \quad 0.4 \leq x \leq 3. \end{aligned} \tag{8.7}$$

Le premier programme trace les segments de droites verticaux (étiquetés par les lettres a–h) pour $\alpha = 0.5$ (a), 0.75 (b), 1 (c), 1.25 (d), $x_m \approx 1.46$ (e), 2 (f), 2.5 (g) et 3 (h), puis les segments horizontaux (A–F) pour $\beta = 0.25$ (A), 0.5 (B), 0.75 (C), 1 (D), 1.5 (E) et 2 (F). La valeur x_m , qui correspond au minimum de $\Gamma(x)$ pour x réel positif, sera calculée plus bas à l'aide de la fonction `digamma`.

Comme la transformation conforme $z \rightarrow \Gamma(z)$ préserve les angles, le réseau transformé est formé de courbes orthogonales. C'est bien ce qui apparaît à la sortie du deuxième programme. L'écriture des deux programmes est semblable. L'origine est le point de coordonnées écran (`x0`, `y0`), et les unités valent `sx` et `sy` pixels sur les axes Ox et Oy respectivement. La procédure `axis` trace le système d'axes Oxy . Chaque passage dans les boucles `forv` correspond à une courbe différente. La procédure `fplot` trace la trajectoire du nombre complexe $a + bt$ ou $\Gamma(a + bt)$, lorsque t parcourt l'intervalle réel spécifié dans les deux premiers arguments de `fplot`, avec le pas donné en troisième argument. Les deux derniers arguments de `fplot` sont les fonctions qui calculent les parties réelles et imaginaires du nombre complexe $a + bt$ ou $\Gamma(a + bt)$. La procédure `lettre z` affiche le nom de la courbe (une des lettres a–h, A–F de code ASCII `l`), près du nombre complexe z ou $\Gamma(z)$; dans le plan $\Gamma(z)$, on déplace les lettres a et c pour plus de lisibilité.

```
complex i
cursh 0
cls
x0=20
y0=350
sx=160
sy=80
xm=1.461632145
axis x0,y0,0,399,639,0,sx,sy,"x","y"
b=i
l=$61
forv a in (0.5, 0.75, 1, 1.25, xm, 2, 2.5, 3)
    fplot 0,4,4,x0,y0,sx,sy,xd,yd
```

```

    lettre a+0.1*i
nextv
b=1
l=$41
forv a in (i/4, i/2, 3*i/4, i, 1.5*i, 2*i)
    fplot 0.4, 3, 2.6,x0,y0,sx,sy,xd,yd
    lettre 0.3+a
nextv
ift inp(2)
stop
lettre:z=@1
y=y0-sy*im(z)+4
x=x0+sx*re(z)+4
text x,y,chr$(l)
l=l+1
return
xd:function(t)
    value=re(a+b*t)
    return
yd:function(t)
    value=im(a+b*t)
    return

```

Sortie (4395 ms)

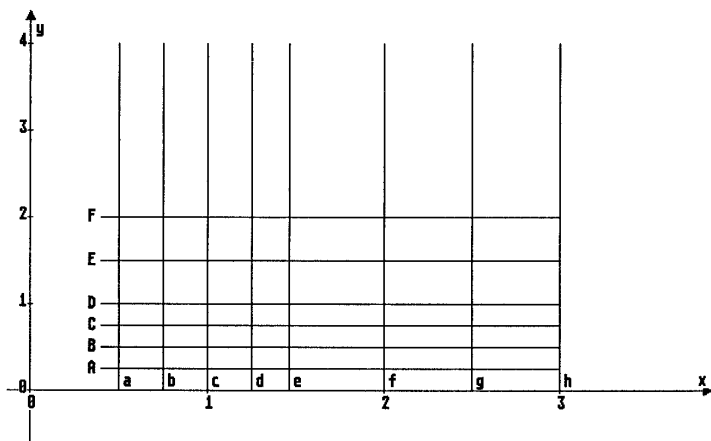


Figure 8.2. Le réseau du plan z


```

'Réseau transformé
complex i
cursh 0
cls
x0=120
y0=200
sx=200
sy=200
xm=1.4616321449
axis x0,y0,0,399,639,0,sx,sy,"x","y"
b=i
l=$61
forv a in (0.5, 0.75, 1, 1.25, xm, 2, 2.5, 3)
    fplot 0,4,0.02,x0,y0,sx,sy,xg,yg
    lettre a+0.1*i
nextv
b=1
l=$41
forv a in (i/4, i/2, 3*i/4, i, 1.5*i, 2*i)
    fplot 0.4, 3, 0.01,x0,y0,sx,sy,xg,yg
    lettre 0.42+a
nextv
ift inp(2)
stop

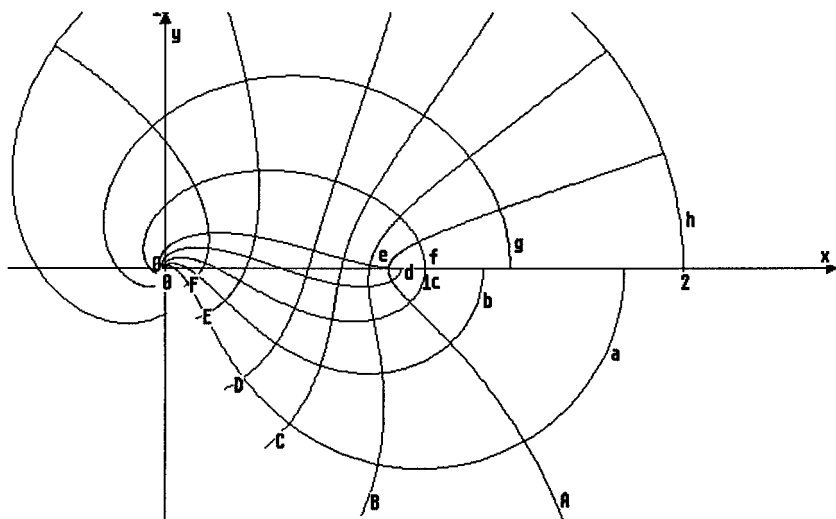
lettre:z=gamma(@1)
y=y0-sy*im(z)+4
x=x0+sx*re(z)+4
ift l=$63 x=x+3
if l=$65
    y=y-10
    x=x-16
endif
text x,y,chr$(l)
l=l+1
return

xg:function(t)
    value=re(gamma(a+b*t))
    return

yg:function(t)
    value=im(gamma(a+b*t))
    return

```

Sortie (4501 s)

Figure 8.3. Le réseau du plan $\Gamma(z)$

Description du programme gamma

Nous utilisons le développement asymptotique :

$$\log \Gamma(z) \sim \left(z - \frac{1}{2}\right) \log z - z + \frac{1}{2} \log(2\pi) + \frac{1}{12z} - \frac{1}{360z^3} + \frac{1}{1260z^5} - \frac{1}{1680z^7} + \dots \quad (8.8)$$

pour $z \rightarrow \infty$ avec $|\arg z| < \pi$. Le terme général de (8.8) est donné par :

$$T_{n-1} = \frac{B_{2n}}{2n(2n-1)z^{2n-1}} \sim \frac{1}{e\sqrt{n\pi}} \left(\frac{n}{\pi ez}\right)^{2n-1}, \quad (n > 0) \quad (8.9)$$

en fonction des nombres de Bernoulli B_{2n} , la forme équivalente étant valable pour $n \rightarrow \infty$. La variable `psi` contient $N = 31$ termes du développement sous la forme d'un polynôme en $x = 1/z$:

$$\psi = \sum_{i=1}^N \frac{B_{2i} x^{2i-1}}{2i(2i-1)}. \quad (8.10)$$

Le coefficient de x^{2n-1} de ce polynôme est placé dans `gamma_psic(n-1)` pour $1 \leq n \leq 31$ lors du premier appel de la fonction `gamma` par appel de la procédure `gamma_it`. Pour z donné, les termes T_n (équation (8.9)) décroissent en fonction de n jusqu'à $n \sim \pi z$ puis croissent ensuite. Supposons que $\log \Gamma(z)$ soit calculé par le développement (8.8) sommé jusqu'au terme T_m . L'erreur sur $\log \Gamma(z)$ est alors inférieure à la valeur absolue du terme T_{m+1} pour z réel. Comme on désire que cette erreur soit inférieure à $\epsilon = 2^{-a}$, où $a = \text{precision2}$, on ne calculera le développement (8.8) que pour des valeurs de z telles que $\text{Re}(z) > y$. Le nombre réel y est déterminé de sorte que T_m soit en module d'une part le plus petit des termes calculés ($y \geq m/\pi \approx \lceil m/3 \rceil$), et d'autre part inférieur à ϵ . La valeur m choisie,

$$m \approx -\frac{1}{2} \log \epsilon \approx 9a/26, \quad (8.11)$$

dans la limite des coefficients précalculés $m \leq 30$, rend y le plus petit possible. Pour calculer $\Gamma(x)$ on détermine un entier $k \geq 0$ tel que $\text{Re}(x+k) \geq y$, puis on utilise :

$$\Gamma(x+k) = (x+k-1)(x+k-2) \cdots (x+1)x\Gamma(x), \quad (8.12)$$

$\Gamma(x+k)$ étant évalué par la forme asymptotique (8.8) et $(x+k-1)(x+k-2) \cdots (x+1)x$ par `ppwr(x+k-1, k)`. Pour éviter un dépassement dans le calcul de `ppwr`, y a été plafonné à la valeur 2000. Cela revient à limiter l'erreur ϵ à la valeur qui correspond à la précision 166.

Comme la fonction `re` n'est utilisable que lorsque le littéral complexe est défini, on évite de prendre la partie réelle en mode réel, en interposant le test `ift complex`. Le décodage du gros polynôme `psi`, dans la procédure `gamma_it`, est effectué en mode `develop`. Le temps de décodage est alors 450 ms, tandis qu'avec l'option `factor` 31 secondes seraient nécessaires. Bien entendu, l'option à l'entrée de la procédure est poussée sur la pile (`push factor`) avant la commande `develop`, pour pouvoir être rétablie en fin de procédure par la commande `factor pop`. On pourrait calculer le développement (8.8) par :

```
fsubs(psi,x=1/tf)
```

`psi` et `x` n'étant plus déclarés locaux dans `gamma_it`. Avec `psi` sous forme factorisée, on gagne alors jusqu'à 300 ms à chaque appel de `gamma`. Cette méthode n'a pas été utilisée parce que la conversion de `psi` à la forme factorisée allongerait de 5 secondes le temps du premier appel de `gamma`.

```
gamma:function(tf)
  var gamma_psic(30)
  tf=float(tf)
  local index i,m
  local var k,y,tfi,tff
  ift gamma_psic(0)<>1/12 gamma_it
  m=min(cint(precision2*9/26),30)
  y=precision2
  push precision2
```

```

precision2 43
y=max((abs(gamma_psic(m))/2~^-y)^(.5/(m+.5)),gint(m/3))
y=min(y,2000)
k=tf
ift complex k=re(tf)
k=max(gint(y-k),0)
precision2 pop
tf=tf+k
value=(tf-.5)*log(tf)-tf
tfi=1/tf
tff=tfi^2
for i=0,m
  vadd value,tfi*gamma_psic(i)
  vmul tfi,tff
next i
value=sqr(2*pi)*exp(value)/ppwr(tf-1,k)
return
gamma_it:local var psi lit x index i
push factor
develop
psi= 396793078518930920708162576045270521/732*x^61 -12
15233140483755572040304994079820246041491/201025024200
*x^59 +2913228046513104891794716413587449/40356*x^57 -
354198989901889536240773677094747/382800*x^55 +2914996
3634884862421418123812691/2283876*x^53 -61628132164268
458257532691681/324360*x^51 +1980228820964318592849910
1/6468*x^49 -5609403368997817686249127547/104700960*x^
47 +25932657025822267968607/25380*x^45 -25302972344819
11294093/118680*x^43 +1520097643918070802691/3109932*x
^41 -261082718496449122051/21106800*x^39 +154210205991
661/444*x^37 -26315271553053477373/2418179400*x^35 +15
1628697551/396*x^33 -7709321041217/505920*x^31 +172316
8255201/2492028*x^29 -3392780147/93960*x^27 +657931/30
0*x^25 -236364091/1506960*x^23 +77683/5796*x^21 -17461
1/125400*x^19 +43867/244188*x^17 -3617/122400*x^15 +1/
156*x^13 -691/360360*x^11 +1/1188*x^9 -1/1680*x^7 +1/1
260*x^5 -1/360*x^3 +1/12*x
for i=30,0
  gamma_psic(i)=coef(psi,x,2*i+1)
next i
factor pop
return

```

Exercice 8.2. Super gamma_it

Comment rendre plus rapide la procédure d'initialisation `gamma_it` ?

La procédure `stirling`

La procédure `stirling` m de la bibliothèque MATH calcule la forme explicite du développement asymptotique de $\log \Gamma(x)$ (8.8) jusqu'au terme en $1/z^{2m-1}$. Les nombres de Bernoulli sont calculés à partir de leur fonction génératrice :

$$w = \frac{z}{e^z - 1} = \sum_{n=0}^{\infty} \frac{B_n}{n!} z^n. \quad (8.13)$$

Le développement limité de w est calculé à l'ordre $2m$ par `sexp` et `taylor`. Le coefficient de degré $2i$ de w (donné par `coef`), $B_{2i}/(2i)!$, est utilisé pour former le polynôme `psi` (8.10). C'est ce polynôme qui apparaît dans `gamma_it`, avec le même nom `psi` et à l'ordre $6l$,

Le développement (8.8) donne par exponentiation la formule de Stirling :

$$\Gamma(z) \sim e^{-z} z^{z-1/2} \sqrt{2\pi} \left(1 + \frac{1}{12z} + \frac{1}{288z^2} - \dots \right). \quad (8.14)$$

La procédure `stirling` détermine cette formule en effectuant le développement limité de e^{psi} par `sexp`.

```
stirling:procedure
  m=@1
  clear timer
  char gx
  gx=chr$( $2)&"(x)"
  print "Sortie du développement de Stirling de log ";gx;
    " à l'ordre ";2*m-1
  print "log ";gx;"= (x-1/2)*log(x)-x+log(2*pi)/2+ psi +0
    (1/x^";justl$(2*m+1);")"
  w=taylor(x/(sexp(x,2*m+1)-1),2*m)
  psi=formd(sum(i=1,m of x^(2*i-1)*coef(w,x,2*i)*ppwr(2*i
    -2)))
  print "psi=";change$(str$(psi,/x),"*x","/x")
  print "timer=";timer
  print "Voici aussi le développement de Stirling de ";gx
  sti=formd(sexp(psi,2*m-1,x))
  print chr$( $2);"(x)=exp(-x)*x^(x-1/2)*sqr(2*pi)*{";cha
```

```

    nge$(str$(sti,/x),"*x","/x");"  +0(1/x^";justl$(2*m);"
  ) }"
print "timer=";timer
return

```

La procédure bernoulli

La procédure **bernoulli** peut être appelée après que la procédure **stirling** ait calculé le développement (8.13), w . Elle affiche les nombres de Bernoulli, tels qu'ils ont été définis par Jakob Bernoulli. Ils diffèrent de la notation moderne, définie par l'équation (8.13), par la valeur de l'index et par le signe.

```

bernoulli:procedure
  var m
  if m=0
    print "Il faut d'abord appeler stirring"
    return
  endif
  print "Nombres de Bernoulli"
  for k=1,m
    print "B";justl$(k);"=";coef(w,x,2*k)*ppwr(2*k)*(-1)^(k+1)
  next k
  return

```

Les polynômes et nombres de Bernoulli

Dans ce paragraphe, nous donnons une méthode de calcul des nombres de Bernoulli plus efficace que le calcul du développement limité du programme **stirling**. Les polynômes de Bernoulli $B_n(x)$ sont définis par la fonction génératrice :

$$\frac{ze^{xz}}{e^z - 1} = \sum_{n=0}^{\infty} \frac{B_n(x)}{n!} z^n. \quad (8.15)$$

Voici quelques propriétés de ces polynômes. En faisant $x = 0$ dans (8.15) on obtient la fonction génératrice (8.13) des nombres de Bernoulli. Il en résulte la relation :

$$B_n = B_n(0). \quad (8.16)$$

Les polynômes de Bernoulli $B_n(x)$ ($n \geq 1$) satisfont l'équation aux différences :

$$B_n(x+1) - B_n(x) = nx^{n-1}, \quad (8.17)$$

et permettent de calculer les sommes de puissances :

$$\sum_{k=1}^m k^n = \frac{B_{n+1}(m+1) - B_{n+1}(0)}{n+1}. \quad (8.18)$$

Ils sont donnés en fonction des nombres de Bernoulli B_n par :

$$B_n(x) = \sum_{k=0}^n \binom{n}{k} B_{n-k} x^k. \quad (8.19)$$

On a pour $n \geq 2$:

$$B_n(1) = B_n(0) = B_n. \quad (8.20)$$

Les deux premiers nombres de Bernoulli sont $B_0 = 1$ et $B_1 = -1/2$. Les nombres suivants B_n ($n > 0$) peuvent être calculés par récurrence à l'aide de la relation :

$$B_n = \sum_{k=0}^n \binom{n}{k} B_k, \quad n \geq 2. \quad (8.21)$$

La procédure `bnum(N, B2)` détermine les nombres de Bernoulli de rang pair $B2(n) = B_{2n}$ pour $n \leq N$. Le tableau `B2`, utilisé en `access` doit être déclaré par `var` préalablement à l'appel. La relation (8.21) est utilisée en tenant compte du fait que les nombres de rang impair, B_{2n+1} , sont nuls pour $n > 0$. L'exemple affiche les nombres non nuls de B_0 à B_{10} . Les nombres de Bernoulli apparaissent également dans la valeur de la fonction Zêta de Riemann ζ aux points $2n$, n étant un entier positif :

$$\zeta(2n) = \sum_{k=1}^{\infty} \frac{1}{k^{2n}} = \frac{(2\pi)^{2n}}{2(2n)!} |B_{2n}|. \quad (8.22)$$

Les valeurs obtenues $B_2 = 1/6$ et $B_4 = -1/30$, et la formule (8.22) donnent :

$$\begin{aligned} \zeta(2) &= 1 + \frac{1}{2^2} + \frac{1}{3^2} + \cdots = \frac{\pi^2}{6} \\ \zeta(4) &= 1 + \frac{1}{2^4} + \frac{1}{3^4} + \cdots = \frac{\pi^4}{90} \end{aligned} \quad (8.23)$$

```
N=5
var B2(N)
bnum N,B2
print "B0= 1"
```

```

print "B1= -1/2"
for n=1,N
    print using "B#=#";2*n;B2(n)
next n
stop
bnum:procedure(N, access B(@1))
    'B(i) est le nombre de Bernoulli B2i.
    'B1=-1/2
    'B2i+1=0 (i>0)
    B(0)=1
    ift N=0 return
    local index n,k
    local var P
    for n=1,N
        P=1
        B(n)=- (2*n+1)/2
        for k=0,n-1
            vadd B(n),B(k)*P
            vmul P,(2*n+1-2*k)*(n-k)/(2*k+1)/(k+1)
        next k
        vmul B(n),-1/(2*n+1)
    next n
    return

```

Sortie (565 ms)

```

B0= 1
B1= -1/2
B2=1/6
B4=-1/30
B6=1/42
B8=-1/30
B10=5/66

```

Exercice 8.3. Polynômes de Bernoulli

Ecrire un programme calculant les polynômes de Bernoulli $B_n(x)$.

Exercice 8.4. psi

Utiliser la procédure `bnum` pour calculer le polynôme `psi`, donné par l'équation (8.10), utilisé par la fonction `gamma`.

Exercice 8.5. Somme finie sur un polynôme

Nous avons vu que les polynômes de Bernoulli permettent de calculer les sommes de puissances (8.18). De façon plus générale, on peut calculer la somme finie :

$$S = \sum_{x=a}^b f(x) \quad (8.24)$$

lorsque $f(x) = \sum_n a_n x^n$ est un polynôme en x , a et b étant des entiers $a \leq b$. En effet, le polynôme

$$F(x) = \sum_n a_n \frac{B_{n+1}(x)}{n+1} \quad (8.25)$$

vérifie, d'après l'équation aux différences (8.17),

$$S = \sum_{x=a}^b f(x) = F(b+1) - F(a). \quad (8.26)$$

La bibliothèque MATH contient la fonction `dsun(f, x, a, b)` qui détermine la somme S par une méthode d'identification. Ecrire une fonction `dsunb(f, x, a, b)`, de syntaxe analogue à `dsun` qui somme $f(x)$ par l'intermédiaire des polynômes de Bernoulli.

Les polynômes et nombres d'Euler

Les polynômes d'Euler $E_n(x)$ (pour $n = 0, 1, \dots$) sont définis par la fonction génératrice :

$$\frac{2e^{xz}}{e^z + 1} = \sum_{n=0}^{\infty} \frac{E_n(x)}{n!} z^n, \quad (8.27)$$

et les nombres d'Euler E_n sont les nombres $E_n = 2^n E_n(1/2)$. Ces polynômes permettent de calculer les sommes alternées de puissances :

$$\sum_{k=1}^m (-1)^{m-k} k^n = \frac{E_n(m+1) + (-1)^m E_n(0)}{2}. \quad (8.28)$$

Les nombres d'Euler sont des entiers et $E_{2n+1} = 0$ pour $n = 0, 1, \dots$.

La procédure `enum(N, E2)` détermine les nombres d'Euler de rang pair $E2(n) = E_{2n}$ pour $n \leq N$. Le tableau `E2`, utilisé en `access` doit être déclaré par `var` préalablement à l'appel. Les nombres d'Euler sont calculés par $E_0 = 1$ et en utilisant la relation (pour $n \geq 1$) :

$$E_{2n} = - \sum_{k=0}^{n-1} \binom{2n}{2k} E_{2k}. \quad (8.29)$$

```
N=10
var E2(N)
enum N,E2
for n=0,10
  print using "E#=#";2*n;E2(n)
```

```

next n
stop
enum:procedure(N, access E(@1))
local var P
local index n,k
E(0)=1
ift N=0 return
for n=1,N
  P=1
  E(n)=0
  for k=0,n-1
    vsub E(n),P*E(k)
    P=P*(n-k)*(2*n-2*k-1)/((2*k+1)*(k+1))
  next k
next n
return

```

Sortie (1365 ms)

```

E0=1
E2=-1
E4=5
E6=-61
E8=1385
E10=-50521
E12=2702765
E14=-199360981
E16=19391512145
E18=-2404879675441
E20=370371188237525

```

Exercice 8.6. enum_bis

Ecrire un programme qui calcule les nombres d'Euler à l'aide de la relation (pour $n \geq 1$) :

$$E_{2n} = (2n)! \left[\sum_{k=0}^n (2^{2n+2-k} - 1)(2^{1-2k} - 1) \frac{2^{2k} B_{2k}}{(2k)!} \frac{2^{2n+2-2k} B_{2n+2-2k}}{(2n+2-2k)!} \right]. \quad (8.30)$$

Exercice 8.7. Polynômes d'Euler

Ecrire un programme calculant les polynômes d'Euler $E_n(x)$ à l'aide de la formule :

$$E_n(x) = \sum_{k=0}^n \binom{n}{k} \frac{E_k}{2^k} \left(x - \frac{1}{2} \right)^{n-k}. \quad (8.31)$$

On pourra tester le programme en vérifiant l'équation :

$$E_n(x+1) + E_n(x) = 2x^n \quad (8.32)$$

satisfaite par les polynômes d'Euler.

Permutations d'André

Représentation graphique

Une permutation p_1, p_2, \dots, p_N des entiers $1, 2, \dots, N$ est appelée permutation d'André si aucun nombre p_k ($2 \leq k \leq N-1$) n'a une valeur comprise entre p_{k-1} et p_{k+1} . Ces permutations sont aussi appelées permutations en zigzag, car en effet la ligne brisée qui joint les points (i, p_i) forme un zigzag. Le programme suivant détermine une permutation d'André aléatoire des $N = 10$ premiers entiers. Pour cela, une permutation aléatoire est créée par la fonction `nextperm` appelée avec un troisième argument négatif. A la sortie de la boucle `for`, l'index k vaut N si et seulement si la permutation est en zigzag. Le système d'axes et la ligne brisée sont ensuite tracés par les procédures `axis` et `fplot`,

```

N=10
index P(N)
do
  ift nextperm(N,P(1),-1)
  for k=2,N-1
    ift P(k) in [P(k-1),P(k+1)] exit
    ift P(k) in [P(k+1),P(k-1)] exit
  next k
  ift k=N exit
loop
x0=20
y0=380
sx=600/N
sy=360/N
cursh 0
cls
axis x0,y0,0,399,639,0,sx,sy,"n","Pn"
fplot 1,N,1,x0,y0,sx,sy,,P
ift inp(2)
stop

```

Sortie (6 s)

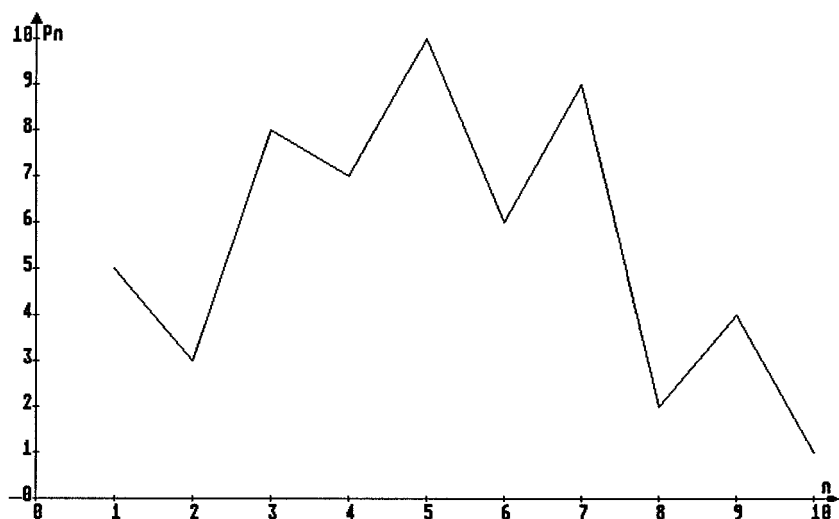


Figure 8.4. Permutation en zigzag

Exercice 8.8. Liste André

Ecrire un programme qui dresse la liste des permutations d'André de $N = 4$ éléments.

La formule d'André

Le mathématicien André a donné une méthode de calcul du nombre des permutations en zigzag. Pour les démonstrations, on peut consulter le livre de Dörrie. Le nombre de permutations en zigzag de n éléments est donné pour $n \geq 2$ par l'entier pair $2A_n$, où A_n peut se calculer par récurrence avec la formule :

$$2A_n = \sum_{k=0}^{n-1} \binom{n-1}{k} A_k A_{n-1-k}, \quad n \geq 2 \quad (8.33)$$

à partir de $A_0 = A_1 = 1$.

La procédure `andre(N, A)` détermine les nombres d'André $A(n) = A_n$ pour $0 \leq n \leq N$. Le tableau `A`, utilisé en `access` doit être déclaré par `var` préalablement à l'appel. L'exemple calcule et affiche les entiers A_n pour $0 \leq n \leq N = 8$, qui représentent la moitié du nombre de permutations en zigzag de n éléments (pour $n \neq 0, 1$).

```

N=8
var A(N)
andre N,A

```

```

for n=0,N
    print using "A#=#";n;A(n)
next n
stop
andre:procedure(N, access A(@1))
    '2*A(i) est le nombre de permutations d'André de i
    éléments
    A(0)=1
    A(1)=1
    local index n,k
    for n=2,N
        P=1
        A(n)=0
        for k=0,n-1
            vadd A(n),P*A(k)*A(n-1-k)
            P=P*(n-1-k)/(k+1)
        next k
        vdiv A(n),2
    next n
    return

```

Sortie (935 ms)

```

A0=1
A1=1
A2=1
A3=2
A4=5
A5=16
A6=61
A7=272
A8=1385

```

Exercice 8.9. André Bernoulli Euler

Les nombres de Bernoulli et d'Euler de rang pair sont reliés aux nombres d'André par les relations :

$$\begin{aligned}
 A_{2n-1} &= \frac{(-1)^{n-1} 2^{2n} (2^{2n} - 1) B_{2n}}{2n}, \\
 A_{2n} &= (-1)^n E_{2n}.
 \end{aligned}
 \tag{8.34}$$

Ecrire un programme qui calcule les nombres de Bernoulli et d'Euler à l'aide de la procédure **andre**, et étudier si cette méthode est plus rapide que les procédures **bnum** et **enum**.

Exercice 8.10. stan

Les développements en série de $\tan x$ et $\sec x = 1/\cos x$ pour $|x| < \pi/2$ peuvent s'écrire en fonction des nombres d'André A_n :

$$\begin{aligned}\tan x &= \frac{A_1}{1!}x + \frac{A_3}{3!}x^3 + \frac{A_5}{5!}x^5 + \frac{A_7}{7!}x^7 + \dots \\ \sec x &= \frac{A_0}{0!} + \frac{A_2}{2!}x^2 + \frac{A_4}{4!}x^4 + \frac{A_6}{6!}x^6 + \dots\end{aligned}\tag{8.35}$$

En utilisant les procédures **bnun** et **enum**, écrire des fonctions **stan**(p, k [, x]) et **sssec**(p, k [, x]), de syntaxe analogue aux fonctions **ssin** ou **scos**, qui calculent le développement limité de $\tan p$ et $\sec p$ à l'ordre k en x .

La lemniscate de Bernoulli

Cette section introduit la constante de la lemniscate que nous calculerons avec 1000 chiffres significatifs dans la section suivante. La spirale sinusoidale d'équation polaire $r^n = a^n \cos n\theta$ est composée de n lobes identiques. Le périmètre de la courbe est donné par :

$$2^{(1-n)/n} a \frac{[\Gamma(1/2n)]^2}{\Gamma(1/n)}.\tag{8.36}$$

Pour $n = 2$, on obtient la lemniscate de Bernoulli de périmètre $2a\varpi$ où :

$$\varpi = 2 \int_0^1 \frac{dx}{\sqrt{1-x^4}} = \frac{[\Gamma(1/4)]^2}{2\sqrt{2\pi}},\tag{8.37}$$

est la constante de la lemniscate. La constante ϖ est pour la lemniscate l'analogue de π pour le cercle (qui d'ailleurs s'obtient comme la spirale sinusoidale d'ordre $n = 1$).

La lemniscate est donnée par l'équation polaire :

$$r^2 = a^2 \cos 2\theta,\tag{8.38}$$

par l'équation cartésienne :

$$(x^2 + y^2)^2 = a^2(x^2 - y^2),\tag{8.39}$$

ou par la représentation paramétrique :

$$\begin{cases} x = a \frac{\cos t}{1 + \sin^2 t} \\ y = a \frac{\sin t \cos t}{1 + \sin^2 t} \end{cases} \quad -\pi \leq t \leq \pi.\tag{8.40}$$

Le programme suivant montre comment passer de l'équation paramétrique à l'équation cartésienne, puis à l'équation polaire. Les polynômes P_x et P_y correspondent aux équations (8.40), les littéraux $ct = \cos t$ et $st = \sin t$ étant liés par la relation $P = \sin^2 t + \cos^2 t - 1 = 0$. On obtient l'équation (8.39) en éliminant ct et st entre les trois polynômes P_x , P_y et P : l'élimination de ct donne Q_x et Q_y qui dépendent de st par son carré st^2 ; l'élimination de st^2 donne Q , qui est réduit par `red` pour supprimer les facteurs numériques ou dépendant de a . La substitution $x = r \cos \theta$ et $y = r \sin \theta$ dans l'équation cartésienne (8.39) est effectuée à l'aide des littéraux r , $c = \cos \theta$ et $s = \sin \theta$, en imposant la condition $\cos^2 \theta + \sin^2 \theta = 1$. La forme polaire (8.38) découle facilement du polynôme U en utilisant $\cos 2\theta = 2 \cos^2 \theta - 1$.

```
Px=x*(1+st^2)-a*ct
Py=y*(1+st^2)-a*st*ct
P=st^2+ct^2-1
Qx=elim(Px,P,ct)
Qy=elim(Py,P,ct)
Qx=subsr(Qx,st^2=st2)
Qy=subsr(Qy,st^2=st2)
Q=elim(Qx,Qy,st2)
Q=red(Q,x,y)
print Q
cond s^2+c^2-1
U=subs(Q,x=r*c,y=r*s)
print formf(U)
```

Sortie (660 ms)

```
x^4 -x^2*a^2 +2*x^2*y^2 +a^2*y^2 +y^4
- [r]^2* [2*a^2*c^2 -a^2 -r^2]
```

Le programme suivant utilise l'équation paramétrique (8.40) pour tracer la lemniscate.

```
cursh 0
cls
axis 320,200,0,350,639,50,250,250,"x","y"
fplot -pi,pi,pi/50,320,200,250,250,x,y
ift inp(2)
stop
x:function(t)
value=cos(t)/(1+sin(t)^2)
return
y:function(t)
value=sin(t)*cos(t)/(1+sin(t)^2)
return
```

Sortie (25 s)

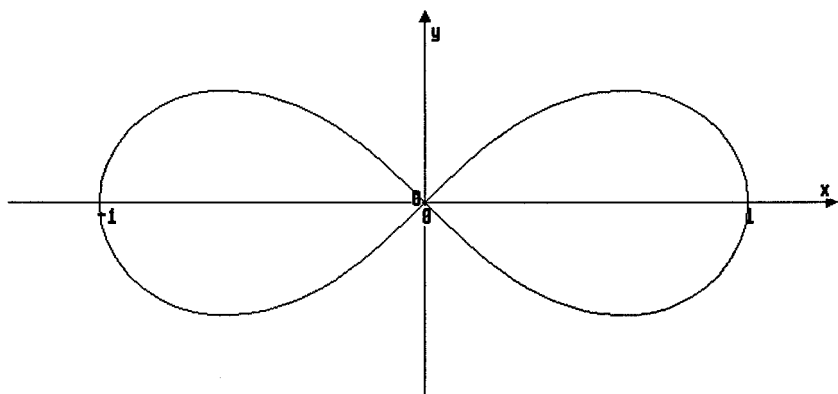


Figure 8.5. Lemniscate de Bernoulli

Fonction Gamma en précision 1000

La fonction `gp(x)` calcule $\Gamma(x)$ avec une précision quelconque jusqu'à la limite des 1230 chiffres des calculs en flottant. La méthode de calcul est la même que pour `gamma`, mais maintenant le développement asymptotique (8.8) est calculé jusqu'au terme z^{-2m+1} , où $m = 210$ est la valeur qui minimise le temps de calcul en précision 1000. La procédure `gp_it`, appelée seulement au premier appel de `gp`, calcule les coefficients du développement (8.8) à partir de la table des nombres de Bernoulli créée par la procédure `bnum`. A la différence de `gamma`, la borne inférieure y de z peut dépasser 2000 (en précision 1000, $y \approx 5891.7$). Le calcul direct du produit $(x+k-1)(x+k-2)\cdots(x+1)x$ dans l'équation (8.12) par `ppwr(x+k-1, k)` provoquerait en général un dépassement, le nombre de facteurs du produit étant de l'ordre de y . Il est calculé par blocs de 2000 facteurs dans la boucle `while ...wend`. L'exemple détermine la constante de la lemniscate ϖ (voir équation (8.37)) avec 1000 chiffres significatifs. Noter

la valeur élevée donnée à `pack` pour éviter une erreur mémoire, car, en précision 1000 les fonctions `exp` et `log` exigent une grande place libre.

```
'adjoindre la procédure bnum
pack 500000
precision 1000
print gp(1/4)^2/sqr(8*pi)
stop
gp:function(tf)
  local datai 210 index mm
  local var gp_psic(mm+1)
  tf=float(tf)
  local index i,m
  local var k,y,tfi,tff
  ift gp_psic(0)<>1/12 gp_it
  m=min(cint(precision2*9/26),mm)
  y=precision2
  push y
  precision2 43
  y=max((abs(gp_psic(m))/2~-y)^(.5/(m+.5)),gint(m/3))
  k=tf
  ift complex k=re(tf)
  k=max(gint(y-k),0)
  precision2 pop
  tf=tf+k
  value=log(tf)
  value=(tf-.5)*value-tf
  tfi=1/tf
  tff=tfi^2
  for i=0,m
    vadd value,tfi*gp_psic(i)
    vmul tfi,tff
  next i
  while k>2000
    tff=ppwr(tf-1,2000)
    value=value-log(tff)
    k=k-2000
    tf=tf-2000
  wend
  value=exp(value)
  value=value/ppwr(tf-1,k)
  value=sqr(2*pi)*value
  return
gp_it:local index i
  bnum mm+1,gp_psic
```

```

for i=1,mm+1
  gp_psic(i-1)=gp_psic(i)/(2*i*(2*i-1))
next i
return

```

Sortie (1667 s)

```

0.262205755429211981046483958989111941368275495143162316281682170380
079058707041425023029553296142909344613575267178321805560895690139393
569470111943477523584042264149716490695193689997993214607238312139081
020622189742960085655453977230536954971028888832552648702132901209754
083312856851172975222921429669243051396845645553943288141538133173510
840922631213247666763341450998860342942147922471448796390787256418952
181102725251662996433338466067933363509313980852623773940914262648984
803480457254147704617542125634212995586312998022405460901209149913989
788564531248097110114966507506054209384172388690004027478538962548303
058030394632478321955832552297303719134191898359219991422953667256910
686113093813498072555291301509373033261108704581424076578188653076693
247694076162672163624954948006676096138812232247692559101870577574361
464891287983268666203073137331356210761263637924578580178136410536130
609356347202502259231204120266827045772304460837895331135700294057744
2011806826257962983642671092116198597- E+1

```

Fonctions digamma et polygamma

Les fonctions $\text{loggamma}(x)$, $\text{digamma}(x)$ et $\text{polygamma}(x, n)$ calculent respectivement $\log \Gamma(x)$, $\psi(x) = d \log \Gamma(x)/dx$ et $d^n \psi(x)/dx^n$, pour n entier positif ou nul. La forme $\text{polygamma}(x, -1)$ calcule également $\log \Gamma(x)$. Le calcul est basé sur les développements asymptotiques de ces fonctions pour $z \rightarrow \infty$, avec $|\arg z| < \pi$, qui s'obtiennent par dérivation du développement asymptotique (8.8) :

$$\begin{aligned}
 \psi(z) &\sim \log z - \frac{1}{2z} - \frac{1}{12z^2} + \frac{1}{120z^4} - \frac{1}{252z^6} + \cdots \\
 \psi'(z) &\sim \frac{1}{z} + \frac{1}{2z^2} + \frac{1}{6z^3} - \frac{1}{30z^5} + \frac{1}{42z^7} + \cdots \\
 \psi''(z) &\sim -\frac{1}{z^2} - \frac{1}{z^3} - \frac{1}{2z^4} + \frac{1}{6z^6} - \frac{1}{6z^8} + \cdots \\
 &\dots
 \end{aligned} \tag{8.41}$$

Comme dans le cas de la fonction Γ , la fonction ψ et ses dérivées sont calculées aux petites valeurs de x à partir des développements ci-dessus appliqués à un

grand nombre $z = x + k$ (k entier), en utilisant la formule de récurrence :

$$\psi^{(n)}(z+1) - \psi^{(n)}(z) = \frac{(-1)^n n!}{z^{n+1}}. \quad (8.42)$$

On remarquera la ressemblance entre (8.42) et l'équation aux différences des polynômes de Bernoulli (8.17). La programmation est analogue à celle de la fonction `gamma`. Noter d'ailleurs que la procédure `gamma_it` et le tableau `gamma_psic` sont partagés par les fonctions `gamma` et `polygamma`. Le nombre de fois, k , que la relation (8.42) est appliquée n'est pas limité à 2000 comme dans la fonction `gamma`. Il en résulte que la précision peut dépasser 166 chiffres (précision maximum de `gamma` qui correspond à $k \approx 2000$), mais, comme augmenter la précision de 30 chiffres multiplie k par 10, les calculs deviennent rapidement trop lents. Nous donnerons plus bas des modifications qui fonctionnent rapidement en précision 1000. L'exemple calcule $\log \Gamma(1/2) = (\log \pi)/2$, l'opposé de la constante d'Euler $\psi(1) = -\gamma$ et $\psi'(1) = \pi^2/6$.

```

format 11
g=loggamma(1/2)
print "loggamma(1/2) =";g;using "  err=###^";g-log(pi
)/2
print "digamma(1)  =";digamma(1)
g=polygamma(1,1)
print "polygamma(1,1)=";g;using "  err=###^";g-pi^2/6
stop

loggamma:function(x)
  value=polygamma(x,-1)
  return

digamma:function(x)
  value=polygamma(x,0)
  return

polygamma:function(tf,index q)
  q=q+1
  ift q<0 erreur_q
  var gamma_psic(30)
  tf=float(tf)
  local index i,m
  local var k,y,tfi,tff
  ift gamma_psic(0)<>1/12 gamma_it
  m=min(cint(precision2*9/26),30)
  y=precision2
  push precision2
  precision2 43
  y=(abs(gamma_psic(m)*ppwr(2*m+q,q))/2^-y)
  ift q>1 y=y/ppwr(q-2)
  y=max(y^(.5/(m+1)),gint(m/3))

```

```

k=tf
ift complex k=re(tf)
k=max(gint(y-k),0)
precision2 pop
tf=tf+k
tfi=1/tf
tff=tfi^2
select q
case=0
  value=(tf-.5)*log(tf)-tf+.5*log(2*pi)
case=1
  value=log(tf)-.5*tfi
  tfi=-tff
case others
  tfi=(-tfi)^q
  value=tfi*ppwr(q-2)*(tf+(q-1)*.5)
  tfi=tfi/tf
endselect
for i=0,m
  vadd value,tfi*gamma_psic(i)*ppwr(2*i+q,q)
  vmul tfi,tff
next i
if k
  select q
  case=0
    while k>2000
      tff=ppwr(tf-1,2000)
      value=value-log(tff)
      k=k-2000
      tf=tf-2000
    wend
    value=value-log(ppwr(tf-1,k))
  case others
    tff=sum(i=1,k of (tf-i)^-q)
    value=value+(-1)^q*ppwr(q-1)*tff
  endselect
endif
return

```

Sortie (2680 ms)

```

loggamma(1/2) = 0.5723649429~ err=0.711~ E-14
digamma(1)    = -0.5772156649~
polygamma(1,1)= 1.6449340668~ err=0.178~ E-14

```

Graphe de la fonction digamma $\psi(x)$

Le programme trace la courbe

$$y = \psi(x) = \frac{d \log \Gamma(x)}{dx} \quad (8.43)$$

pour x variant de -5 à 4 . Le programme est identique à celui qui nous a permis de tracer $y = \Gamma(x)$, mis à part que le nom de la fonction a été remplacé par digamma.

```
'adjoindre les fonctions digamma et polygamma
cursh 0
cls
axis 320,200,20,399,585,0,60,30,"x","y"
for j=-4,0
  x0=260+60*j
  line x0,0,x0,399
  fplot j-0.99,j-0.01,0.05,320,200,60,30,,digamma
next j
fplot 0.1,4.2,0.05,320,200,60,30,,digamma
ift inp(2)
stop
```

Sortie (111 s)

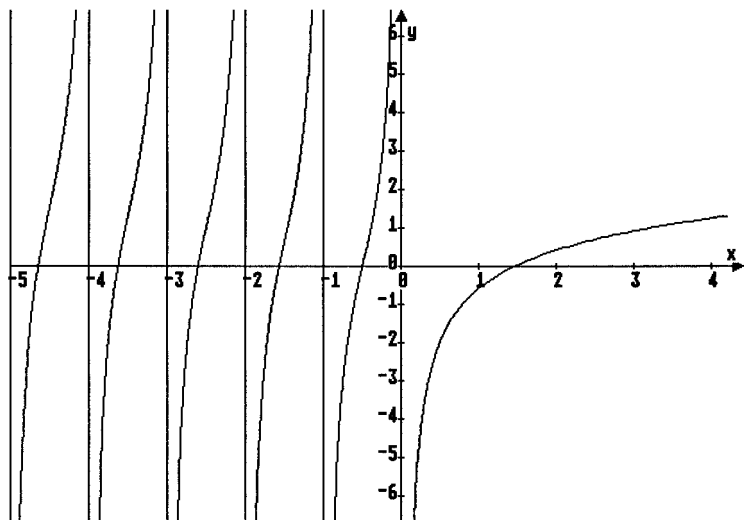


Figure 8.6. La fonction digamma $y = \psi(x)$

Minimum de la fonction $\Gamma(x)$

Le petit programme suivant calcule le minimum de $\Gamma(x)$ pour $x > 0$ en déterminant le zéro de la fonction digamma $\psi(x)$ par la méthode de dichotomie. Les nombres a et b encadrent le zéro cherché, la longueur $|b - a|$ est divisée par deux à chaque itération et le programme s'arrête lorsque $|b - a| < 10^{-10}$.

```
'adjoindre les fonctions digamma et polygamma
a=1~
b=2~
while b-a>10~-10
  c=(a+b)/2
  if digamma(c)<0
    a=c
  else
    b=c
  endif
wend
print a
stop
```

Sortie (19945 ms)

0.1461632145~ E+1

Calcul de $\zeta(n)$

Une application intéressante de la fonction polygamma est la sommation des séries rationnelles. Comme premier exemple, nous calculons $\zeta(n)$, pour $n = 2, 3, \dots, 9$, à l'aide de la relation suivante, valable pour les entiers naturels $n \geq 2$, qui découle immédiatement de l'équation (8.42) :

$$\zeta(n) = 1 + \frac{1}{2^n} + \frac{1}{3^n} + \dots = \frac{(-1)^n \psi^{(n-1)}(1)}{(n-1)!}. \quad (8.44)$$

```
'adjoindre la fonction polygamma
precision 20
c$="#####.##### ##### ##### #"
for n=2,9
  print n;using c$;polygamma(1,n-1)*(-1)^n/ppwr(n-1)
next n
stop
```

Sortie (8645 ms)

```
2      1.64493 40668 48226 43647~
3      1.20205 69031 59594 28540~
4      1.08232 32337 11138 19152~
5      1.03692 77551 43369 92633~
6      1.01734 30619 84449 13971~
7      1.00834 92773 81922 82684~
8      1.00407 73561 97944 33938~
```

9 1.00200 83928 26082 21442~

Sommation de $1/(1+n^4)$

La somme :

$$\sum_{n=1}^{\infty} \frac{1}{1+n^4} \quad (8.45)$$

peut se calculer par la méthode suivante. On décompose $1/(1+n^4)$ en éléments simples :

$$\frac{1}{1+n^4} = 2\operatorname{Re} \left[\frac{e^{-3i\pi/4}}{4} \left(\frac{1}{n - e^{i\pi/4}} - \frac{1}{n + e^{i\pi/4}} \right) \right], \quad (8.46)$$

puis à l'aide de la relation (8.42) on obtient :

$$\sum_{n=1}^{\infty} \frac{1}{1+n^4} = 2\operatorname{Re} \left\{ \frac{e^{-3i\pi/4}}{4} \left[-\psi(1 - e^{i\pi/4}) + \psi(1 + e^{i\pi/4}) \right] \right\}. \quad (8.47)$$

L'élément simple de $1/(1+n^4)$ correspondant au pôle v est $1/4v^3(n-v)$. Le programme suivant montre ce résultat. Les éléments simples de la fraction $f(n) = P/Q$ se rapportant à la racine en $n = v$ du dénominateur Q s'obtiennent par `psing(f, n, v, y)` lorsque v est un nombre rationnel, y étant un littéral qui représente $n - v$. Lorsque v est un nombre algébrique, comme ici, `psing` peut également reconnaître que v est un pôle.

```
cond v^4+1
f=1/(n^4+1)
S=psing(f,n,v,x)
print S
```

Sortie (125 ms)

```
1/4* [x]^-1* [v]^-3
```

La décomposition (8.46) en éléments simples de $1/(1+n^4)$ est la somme sur les 4 racines quatrièmes de -1 de $1/4v^3(n-v)$. Le programme suivant vérifie d'abord cette formule (8.46). Le littéral v représente $e^{i\pi/4}$. On lui impose donc la condition $v^2 = i$. Par suite de la présence de v dans l'expression g , il n'est pas possible de calculer sa partie réelle par la fonction `re` (qui suppose tous les littéraux réels). Ici on forme le complexe conjugué de g en effectuant les substitutions $i \rightarrow -i$ et $v \rightarrow -iv$, qui par addition à g donne le double de la partie réelle de g . La première sortie nulle atteste que la vérification a réussi. La seconde sortie donne le résultat :

$$\sum_{n=1}^{\infty} \frac{1}{1+n^4} \approx 0.5784775797. \quad (8.48)$$

```
'adjoindre les fonctions digamma et polygamma
complex i
cond v^2-i,v
g=1/4/v^3*(1/(n-v)-1/(n+v))
gp=g+subs(g,i=-i,v=-i*v)-1/(n^4+1)
```

```

print gp
V=exp(i*pi/4)
G=1/4/V^3*(-digamma(1-V)+digamma(1+V))
print 2*re(G)
stop

```

Sortie (10280 ms)

0

0.5784775797~

Sommation des séries rationnelles

La fonction `rsum`(f) calcule la somme de la série rationnelle $\sum_{n=1}^{\infty} f(n)$ lorsque $f(n)$ est une fraction rationnelle telle que $f(n) = O(1/n^2)$ pour $n \rightarrow \infty$ et dont les racines du dénominateur sont toutes rationnelles. La première condition est nécessaire pour que la série converge. La deuxième condition est une restriction qui rend l'écriture du programme facile. La méthode est analogue à celle utilisée pour calculer la somme de $1/(n^4 + 1)$ au paragraphe précédent (mais la fraction $1/(n^4 + 1)$ qui a des pôles irrationnels ne peut être sommée par `rsum`). On décompose f en éléments simples, qui sont sommés par les fonctions `polygamma`. L'exemple calcule d'abord la somme de :

$$f(n) = \frac{6n^2 - 1}{2n^3(2n + 1)} = \frac{1}{n} + \frac{1}{n^2} - \frac{1}{2n^3} - \frac{1}{n + 1/2} \quad (8.49)$$

par la formule :

$$\sum_{n=1}^{\infty} f(n) = -\psi(1) + \psi'(1) + \frac{\psi''(1)}{4} + \psi(3/2) \approx 1.6576. \quad (8.50)$$

On calcule ensuite la somme

$$\sum_{n=1}^{\infty} \frac{1}{(n^2 + 1)(4n^2 + 9)} \approx 0.0503, \quad (8.51)$$

ce qui nécessite l'utilisation de nombres complexes pour permettre la factorisation complète du dénominateur.

Voici quelques détails sur la programmation de `rsum`. Le littéral n de $f(n)$, qui peut avoir un nom quelconque, est déterminé par analyse du dénominateur Q de $f(n)$, en effectuant une boucle sur les `polyln`(Q) littéraux de Q jusqu'à obtenir un littéral autre que le littéral complexe. Ce littéral est conservé dans la variable x . La procédure `err_sum`, qui affiche un message d'erreur, est appelée si x n'a pas pu être obtenu, ou si $f(n) \neq O(1/n)$ pour $n \rightarrow \infty$ (les fractions d'ordre $1/n$ à l'infini donnent une série divergente, mais `rsum` est utilisable pour calculer $\sum_{n=1}^N f(n)$). Après factorisation de Q par `formf`, on effectue une boucle sur ses facteurs $S = \text{factorp}(Q, i)$ (pour $i = 1, \text{factorn}(Q)$). Seulement deux types de facteurs sont admis, soit S est un nombre (réel ou complexe), soit S est du premier degré en x . Le produit des facteurs du premier type donne le nombre flottant K . Dans le cas d'un facteur du deuxième type, la racine de S ,

u , est donnée par `sroot`, et la partie de la décomposition en éléments simples concernant le pôle u s'obtient par `psing(f, x, u, y)`. Chaque élément simple

$$A/(x-u)^s$$

contribue au résultat par le nombre flottant

$$A(-1)^{s-1}\psi^{(s-1)}(1-u)/(s-1)!.$$

Les éléments simples considérés sont retranchés de f pour fournir une vérification (en fin de calcul f doit être nul sinon la commande `ift f ?` sort en erreur).

```
'adjoindre la fonction polygamma
factor
print rsum((6*n^2 -1)/(2*n^3*(2*n +1)))
complex i
print rsum(1/((n^2+1)*(4*n^2+9)))
stop
rsum:function(f)
  local index i,j
  local var Q,x,S,u,H,T,K
  local lit y
  push factor
  factor
  Q=denf(f)
  for i=1,polyln(Q)
    x=polyl(Q,i)
    ift x<>complex exit
    x=0
  next i
  ift x=0 err_rsum le dénominateur existe
  ift deg(num(f),x)>=deg(Q,x) err_rsum f(x)=0(1/x) pour x
  grand
  Q=formf(Q)
  K=1~
  develop
  for i=1,factorn(Q)
    S=factorp(Q,i)
    select deg(S,x)
    case=0
      K=S^factore(Q,i)*K
    case=1
      u=sroot(S,x)
      S=psing(f,x,u,y)
      H=-1
      for j=-1,ordf(S,y),-1
        T=coeff(S,y,j)
```

```

        ift T value=value+polygamma(1-u,-1-j)*T*H
        H=H/j
        f=f-T*(x-u)^j
    next j
case others
    err_rsum le dénominateur se factorise complètement
endselect
next i
value=value*K
ift f ?
factor pop
return
err_rsum:print "*ERREUR* rsum. Il faut que @1f."
stop
Sortie (9310 ms)
0.1657611254~ E+1
0.5030593764~ E-1 +i*0~

```

L'exemple suivant montre que $\text{rsum}(f)$ est utilisable lorsque $f(n)$ est en $1/n$ à l'infini. Les sommes :

$$\sum_{n=1}^{\infty} \frac{(-1)^{n+1}}{n} = \log 2$$

$$\sum_{n=1}^{\infty} \frac{(-1)^{n+1}}{2n-1} = \frac{\pi}{4}$$
(8.52)

s'obtiennent en séparant les termes pairs et impairs (qui forment des séries divergentes). On calcule ensuite :

$$s_N = \left(\sum_{n=1}^N \frac{1}{n} \right) - \log N,$$
(8.53)

pour $N = 10, 100, 1000, \dots, 10^{10}$. Noter d'ailleurs que

$$g = \text{rsum}(1/n) = -\psi(1) = \gamma$$
(8.54)

est la limite de s_N lorsque $N \rightarrow \infty$.

```

'adjoindre les fonctions rsum (avec err_rsum)
' et polygamma
print rsum(1/(2*n-1))-rsum(1/2/n);log(2)
print rsum(1/(4*n-3))-rsum(1/(4*n-1));pi/4
g=rsum(1/n)
for k=1,10
    print k;g-rsum(1/(n+10^k))-log(10^k)
next k
stop

```

Sortie (11360 ms)

```
0.6931471806~ 0.6931471806~
0.7853981634~ 0.7853981634~
1 0.6263831610~
2 0.5822073317~
3 0.5777155816~
4 0.5772656641~
5 0.5772206649~
6 0.5772161649~
7 0.5772157149~
8 0.5772156699~
9 0.5772156654~
10 0.5772156650~
```

Exercice 8.11. super rsum

Récrire la fonction `rsum(f)`, en apportant les améliorations suivantes. Si la somme $\sum_{n=1}^{\infty} f(n)$ peut être calculée exactement par l'algorithme de Gosper (voir `dsum` de la bibliothèque MATH), `rsum` devra renvoyer la valeur exacte. Par exemple, pour $f(n) = (2n-1)/n(n+1)(n+2)$, le résultat doit être $3/4$ et non pas un nombre flottant. Étendre la validité de la fonction `rsum` à toutes les fractions rationnelles ($f(n) \sim O(1/n)$ pour $n \rightarrow \infty$). On devra pouvoir entrer des fractions comme $1/(n^4+1)^2$ ou $1/(n^2+2)$ ayant des points singuliers irrationnels.

Constante d'Euler

Nous calculons ici la constante d'Euler $\gamma = -\psi(1)$ avec 1000 décimales exactes. Les fonctions `loggamma(x)`, `digamma(x)` et `polygamma(x, n)` ont été modifiées pour permettre une grande précision. Les nouvelles fonctions, nommées `loggammap`, `digammap` et `polygammap`, utilisent un plus grand nombre, `mm`, de termes dans leurs développements asymptotiques, les coefficients étant calculés par les procédures `gp_it` et `bnum` (voir la fonction `gp`) qu'il faut adjoindre au programme. La valeur `mm = 280` minimise le temps de calcul de $\psi(1)$ en précision 1000.

```
'adjoindre les procédures gp_it et bnum
pack 500000
precision 1000
print -digammap(1)
print "\TSortie (";justl$(mtimer-655);" ms)"
stop
```

```

loggammap:function(x)
    value=polygammap(x,-1)
    return
digammap:function(x)
    value=polygammap(x,0)
    return
polygammap:function(tf,index q)
    q=q+1
    ift q<0 erreur_q
    local datai 280 index mm
    local var gp_psic(mm+1)
    tf=float(tf)
    local index i,m
    local var k,y,tfi,tff
    ift gp_psic(0)<>1/12 gp_it
    m=min(cint(precision2*9/26),mm)
    y=precision2
    push y
    precision2 43
    y=(abs(gp_psic(m)*ppwr(2*m+q,q))/2~^-y)
    ift q>1 y=y/ppwr(q-2)
    y=max(y^(.5/(m+1)),gint(m/3))
    k=tf
    ift complex k=re(tf)
    k=max(gint(y-k),0)
    precision2 pop
    tf=tf+k
    tfi=1/tf
    tff=tfi^2
    select q
    case=0
        value=(tf-.5)*log(tf)-tf+.5*log(2*pi)
    case=1
        value=log(tf)-.5*tfi
        tfi=-tff
    case others
        tfi=(-tfi)^q
        value=tfi*ppwr(q-2)*(tf+(q-1)*.5)
        tfi=tfi/tf
    endselect
    for i=0,m
        vadd value,tfi*gp_psic(i)*ppwr(2*i+q,q)
        vmul tfi,tff
    next i

```

```

if k
  select q
  case=0
    while k>2000
      tff=ppwr(tf-1,2000)
      value=value-log(tff)
      k=k-2000
      tf=tf-2000
    wend
    value=value-log(ppwr(tf-1,k))
  case others
    tff=sum(i=1,k of (tf-i)^-q)
    value=value+(-1)^q*ppwr(q-1)*tff
  endselect
endif
return

```

Sortie (3911 s)

```

0.577215664901532860606512090082402431042159335939923598805767234884
867726777664670936947063291746749514631447249807082480960504014486542
836224173997644923536253500333742937337737673942792595258247094916008
735203948165670853233151776611528621199501507984793745085705740029921
354786146694029604325421519058775535267331399254012967420513754139549
111685102807984234877587205038431093997361372553060889331267600172479
537836759271351577226102734929139407984301034177717780881549570661075
010161916633401522789358679654972520362128792265559536696281763887927
268013243101047650596370394739495763890657296792960100901512519595092
224350140934987122824794974719564697631850667612906381105182419744486
783638086174945516989279230187739107294578155431600500218284409605377
243420328547836701517739439870030237033951832869000155819398804270741
154222781971652301107356583396734871765049194181230004065469314299929
777956930310050308630341856980323108369164002589297089098548682577736
4288253954925873629596133298574739302~

```

Fonction Gamma incomplète

Pour des nombres réels $a > 0$ et $x \geq 0$, la fonction `gammap(a, x)` de la bibliothèque MATH calcule la fonction Gamma incomplète :

$$P(a, x) = \frac{1}{\Gamma(a)} \int_0^x e^{-t} t^{a-1} dt. \quad (8.55)$$

Pour a donné, $P(a, x)$ croît de 0 à 1 lorsque x varie de 0 à ∞ . La fonction peut être appelée sous la forme `gammap(a, x, *)`, où le troisième argument est arbitraire; on obtient alors $1 - P(a, x)$. Comme pour la fonction `gamma`, la précision est limitée à 166 chiffres. Pour $x < \min(8, a)$, la fonction est calculée par le développement en série (6.3) de la fonction hypergéométrique F à partir de :

$$P(a, x) = e^{-x} x^a F(1, a+1, x) / \Gamma(a+1). \quad (8.56)$$

Sinon, le développement en fraction continue suivant est utilisé :

$$(1 - P(a, x)) \Gamma(a) e^x x^{-a} = \frac{1}{x + \frac{1-a}{1 + \frac{1}{x + \frac{2-a}{1 + \frac{2}{x + \dots}}}}} \quad (8.57)$$

On peut calculer les convergents :

$$\frac{p_n}{q_n} = a_0 + \frac{b_1}{a_1 + \frac{b_2}{a_2 + \frac{b_3}{a_3 + \frac{b_4}{\dots + \frac{b_n}{a_n}}}}} \quad (8.58)$$

par les relations de récurrence analogues à (4.3) :

$$\begin{cases} p_{-1} = 1 \\ q_{-1} = 0 \end{cases} \quad \begin{cases} p_0 = a_0 \\ q_0 = 1 \end{cases} \quad \begin{cases} p_n = a_n p_{n-1} + b_n p_{n-2} \\ q_n = a_n q_{n-1} + b_n q_{n-2} \end{cases} \quad (8.59)$$

Après la commande `for`, dans la boucle sur n , les valeurs des convergents (8.58) d'ordres $2n$ et $2n+1$ respectivement sont p/q et r/s . Les nombres p , q , r et s croissent rapidement, aussi une fois tous les 20 passages on les renormalise par division par s . Le calcul de la fraction continue est arrêtée lorsque deux valeurs successives de r/s diffèrent moins que l'erreur souhaitée.

```
gammap_err:print "*erreur* gammap"
stop
gammap:function(a,x)
a=float(a)
x=float(x)
local datav 2^-precision2 var f
ift a<=0 gammap_err
ift x<0 gammap_err
ift x=0 return
if x<max(8~,a)
```

```

value=1
local datav a,1~ var b,c
do
  vadd b,1
  vmul c,x/b
  vadd value,c
  ift abs(c)<=value*f exit
loop
vmul value,exp(-x+a*log(x))/gamma(a+1)
ift @0<>2 value=1~-value
return
else
  local datav 0~,1~,1~,x var p,q,r,s,b index n
  for n=1,2^30
    p=r+p*(n-a)
    q=s+q*(n-a)
    r=x*p+n*r
    s=x*q+n*s
    if s
      b=value
      value=r/s
      ift abs(value-b)<=f*abs(value) exit
      if modr(n,20)=0
        vdiv p,s
        vdiv q,s
        vdiv r,s
        s=1~
      endif
    endif
  next n
  vmul value,exp(-x+a*log(x))/gamma(a)
  ift @0=2 value=1~-value
  return
endif

```

Graphe

Le programme trace les courbes $y = P(a, x)$ pour $a = 1, 2$ et 3 .

```

'adjoindre la procédure qplot (avec q_1)
qplot 0,10,100,P1,P2,P3
ift inp(2)
stop
P1:function(x)
  value=gammap(1~,x)
  return

```

```

P2:function(x)
    value=gammap(2~,x)
    return
P3:function(x)
    value=gammap(3~,x)
    return

```

Sortie (289 s)

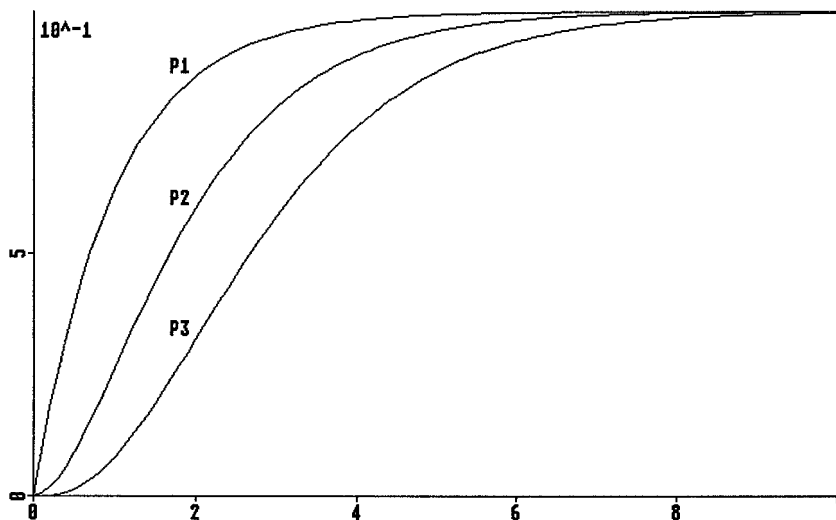


Figure 8.7. La fonction gamma incomplète

Fonction d'erreur

La fonction d'erreur **erf**(x) est définie par l'intégrale :

$$\text{erf}(x) = \frac{2}{\sqrt{\pi}} \int_0^x e^{-t^2} dt \quad (8.60)$$

et la fonction de répartition normale **phistar**(x) par :

$$\Phi^*(x) = \frac{1}{\sqrt{2\pi}} \int_{-\infty}^x e^{-t^2/2} dt. \quad (8.61)$$

Ces fonctions de la bibliothèque MATH sont calculées facilement à l'aide de la fonction **gammap**, par exemple **erf**(x) est donné par $P(1/2, x^2)$.

```

erf:function(x)
    value=sgn(x)*gammap(0.5~,float(x)^2)
    return
phistar:function(x)

```



```

if x>=0
    value=0.5~*(1~+gammap(0.5~,0.5~*float(x)^2))
else
    value=0.5~*gammap(0.5~,0.5~*float(x)^2,*)
endif
return

```

Graphe

Le programme trace les courbes $y = \operatorname{erf}(x)$ et $y = \Phi^*(x)$ à l'aide de la procédure `qplot`. On notera que `qplot` n'a pas réussi à tracer la graduation sur l'axe Oy ($\operatorname{erf}(x)$ varie de -1 à $+1$ quand x varie de $-\infty$ à $+\infty$).

```

'adjoindre la procédure qplot (avec q_1)

qplot -3,3,50,erf,phistar

ift inp(2)

stop

```

Sortie (100 s)

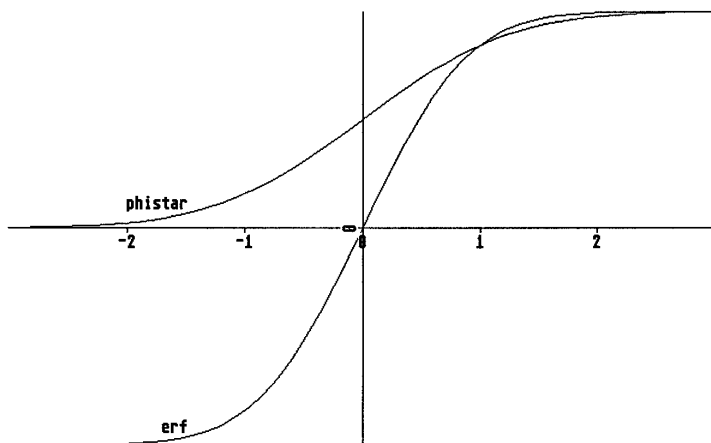


Figure 8.8. Les fonctions $\operatorname{erf}(x)$ et $\Phi^*(x)$

Exercice 8.12. Fresnel

Ecrire une nouvelle version de la fonction d'erreur $\text{erf}(z)$ acceptant un argument z complexe. Le programme $\text{erf}(z)$ de la bibliothèque MATH est limité à z réel, mais il est intéressant de l'étendre aux valeurs complexes de z . En effet on pourra alors calculer les intégrales de Fresnel :

$$\begin{aligned} C(x) &= \int_0^x \cos\left(\frac{\pi}{2}t^2\right)dt \\ S(x) &= \int_0^x \sin\left(\frac{\pi}{2}t^2\right)dt \end{aligned} \quad (8.62)$$

par la relation :

$$C(x) + iS(x) = \frac{1+i}{2} \text{erf}\left(\frac{\sqrt{\pi}}{2}(1-i)x\right). \quad (8.63)$$

On dispose d'équations analogues à (8.56) et (8.57) :

$$\text{erf}(z) = \frac{2}{\sqrt{\pi}} e^{-z^2} \sum_{n=0}^{\infty} \frac{2^n}{1 \cdot 3 \cdot 5 \cdots (2n+1)} z^{2n+1} \quad (8.64)$$

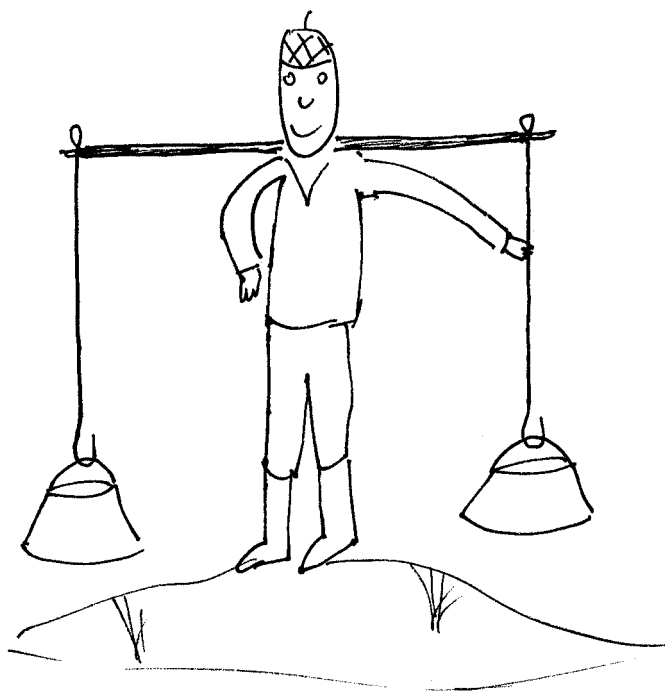
et :

$$\sqrt{\pi} e^{z^2} (1 - \text{erf}(z)) = \frac{1}{z + \frac{1/2}{z + \frac{1}{z + \frac{3/2}{z + \frac{2}{z + \cdots}}}}}, \quad (\text{Re}(z) > 0). \quad (8.65)$$

On étudiera quelle formule (8.64) ou (8.65) utiliser suivant la valeur de z .

9

Etats quasistationnaires



Nous considérons le mouvement à une dimension d'une particule en présence d'une barrière de potentiel. La particule se déplace sur un demi-axe Or dans le potentiel $U(r)$ suivant :

$$U(r) = \begin{cases} 1, & \text{si } \alpha \leq r \leq \beta; \\ 0, & \text{autrement.} \end{cases} \quad (9.1)$$

Les propriétés de ce modèle ont été étudiées par Damburg et Ponomarev (1989) en tant que modèle simplifié de la photoionisation des atomes en présence d'un champ électrique uniforme. Les résultats de cet article repris ici vont nous fournir des exemples de traitement exact d'expressions contenant des fonctions transcendantes (résolution d'un système linéaire, calculs de dérivées et de développements limités).

Figure du potentiel

Comme hors-d'œuvre, le programme suivant trace la figure du potentiel (9.1), puis effectue une copie d'écran. Il faut un moniteur haute résolution, et `resolution = 2`, sinon il y a arrêt de l'exécution après un avis affiché par la commande `message`. La totalité de l'écran est vidée par `print/c/`, après `cursh 0`. Les textes sont simplement écrits par `print`, après spécification de la ligne par `cursl`. Les coordonnées écran (`0x`, `0y`) de l'origine ont été placées à trois pixels à droite et au dessus du 0 qui marque ce point sur le graphe. Les abscisses écran `R0`, `R1` et `Fx` correspondent aux indications sous l'axe des abscisses. De même les ordonnées écran `Uy` et `x2y` correspondent aux indications sur l'axe des ordonnées.

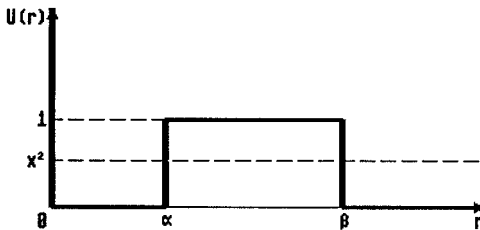
Le graphe est formé de quelques lignes, tracées par la commande `line`. L'épaisseur de ligne (1 ou 3 pixels) est donnée par `l_width`. La ligne est continue (1) ou pointillée (2) suivant la valeur de `l_type`. L'extrémité est normale (0) ou fléchée (1) selon `l_end`. Noter comment la commande `line` permet de tracer une ligne brisée. La souris et le curseur sont rendus invisibles par `hidecm`. Le système est configuré pour une copie d'écran sur imprimante Epson (960 points par ligne) par la fonction `xbios(33)`. Dans le cas d'une imprimante Atari (1280 points par ligne), on utilisera `xbios(33,0)`. Si on ne désire pas imprimer le graphe, on remplacera la commande `hardcopy` par `ift inp(2)`, qui attend l'appui sur une touche.

```
if resolution<>2
    message "Haute résolution|nécessaire"
    stop
endif
```

```

cursh 0
print /c/"U(r)"
cursl 5
print "    1"
cursl 7
print "  x"&chr$( $FD)
cursl 10
print "    0                "&chr$( $E0)&"                "&
      chr$( $E1)&"                r"
print "  Figure : Potentiel U(r)"
Ox=35
Oy=157
Fx=368
R0=122
R1=258
x2y=120
Uy=88
l_type 1
l_width 1
l_end 1
line Ox,Oy, Ox,1
line Ox,Oy, Fx,Oy
l_end 0
l_width 3
line Ox,1,Ox,Oy,R0,Oy,R0,Uy,R1,Uy,R1,Oy,Fx,Oy
l_type 2
l_width 1
line Ox,Uy,R0,Uy
line Ox,x2y,Fx,x2y
hidecm
ift xbios(33,4)
hardcopy' ou ift inp(2)

```

Figure 9.1. Potentiel $U(r)$

La fonction $W(x)$

Une particule d'énergie $x^2 < 1$ ne peut pas franchir le barrière de potentiel en mécanique classique. Elle se déplace soit dans le puits de potentiel ($0 \leq r \leq \alpha$) soit dans la région à droite de la barrière ($\beta \leq r \leq \infty$). En mécanique quantique, par contre, même quand son énergie est inférieure à 1, la particule peut sortir du puits de potentiel par effet tunnel. Nous nous attaquons maintenant à la résolution de l'équation de Schrödinger :

$$\left(\frac{d^2}{dr^2} - U(r) + x^2 \right) \psi(r) = 0. \quad (9.2)$$

En posant $y = \sqrt{1 - x^2}$, la fonction d'onde $\psi(r)$ solution de l'équation (9.2) peut s'écrire :

$$\psi(r) = \begin{cases} \sin xr, & \text{si } 0 \leq r \leq \alpha; \\ ae^{y(r-\alpha)} + be^{-y(r-\alpha)}, & \text{si } \alpha \leq r \leq \beta; \\ A \sin xr + B \cos xr, & \text{si } \beta \leq r \leq \infty. \end{cases} \quad (9.3)$$

Les constantes a , b , A et B s'obtiennent par raccordement de la fonction $\psi(r)$ en α et β , c'est-à-dire par résolution du système des équations qui expriment la continuité de $\psi(r)$ et de sa dérivée :

$$\begin{aligned} \sin \alpha x &= a + b \\ x \cos \alpha x &= y(a - b) \\ A \sin \beta x + B \cos \beta x &= ae^{Sy} + be^{-Sy} \\ xA \cos \beta x - xB \sin \beta x &= y(ae^{Sy} - be^{-Sy}), \end{aligned} \quad (9.4)$$

où nous avons posé $S = \beta - \alpha$.

Le programme suivant effectue la résolution de ce système d'équations, puis calcule l'intensité de l'onde à droite de la barrière $W(x) = A^2 + B^2$. Le résultat obtenu est exprimé en fonction de $t = \tan \alpha x$:

$$\begin{aligned} W(x) &= \frac{t^2}{t^2 + 1} \times \frac{(e^{Sy} - e^{-Sy})^2 + 4x^2}{4x^2} \\ &+ \frac{t}{t^2 + 1} \times \frac{e^{2Sy} - e^{-2Sy}}{2xy} \\ &+ \frac{1}{t^2 + 1} \times \frac{(e^{Sy} + e^{-Sy})^2 - 4x^2}{4y^2}. \end{aligned} \quad (9.5)$$

Les inconnues a , b , A et B du système d'équations (9.4) sont placées dans le tableau **z**(3). Les 4 expressions placées dans le tableau **f**(3) transcrivent les

équations (9.4). Les littéraux ont les équivalences suivantes :

$$\begin{aligned}
 \mathbf{sa} &\equiv \sin \alpha x \\
 \mathbf{ca} &\equiv \cos \alpha x \\
 \mathbf{sb} &\equiv \sin \beta x \\
 \mathbf{cb} &\equiv \cos \beta x \\
 \mathbf{eSy} &\equiv e^{Sy}
 \end{aligned} \tag{9.6}$$

La procédure `sleq` de la bibliothèque MATH résout le système (9.4). Les valeurs des inconnues a , b , A et B sont renvoyées dans le tableau `vz(3)`, ce qui permet de calculer $W = A^2 + B^2$. L'expression obtenue de W peut être simplifiée car les divers littéraux ne sont pas indépendants. Par exemple, la relation $\sin^2 \beta x + \cos^2 \beta x = 1$ permet d'éliminer \mathbf{sb} et \mathbf{cb} de W . Cette élimination est effectuée en remplaçant \mathbf{cb}^2 par $1 - \mathbf{sb}^2$ dans W . L'expression de W s'écrit de façon compacte en fonction de $\tan \alpha x$, représenté ici par le littéral \mathbf{t} . A l'affichage du résultat, W est décomposé pour faire apparaître à peu de choses près la formule (9.5).

```

var z(3),f(3),vz(3)
z(0)=a
z(1)=b
z(2)=A
z(3)=B
f(0)=sa-a-b
f(1)=x*ca-y*(a-b)
f(2)=A*sb+B*cb-(a*eSy+b/eSy)
f(3)=x*A*cb-x*B*sb-y*(a*eSy-b/eSy)
sleq f,z,vz,3
W=vz(2)^2+vz(3)^2
W=subsr(W,cb^2=1-sb^2)
W=subs(W,sa=t*ca)
W=subsr(W,ca^2=1/(1+t^2))
W=subsr(W,y^2=1-x^2)
factor
WD=denf(W)
WN=num(W)
print "W="
for i=deg(WN,t),0
  print using "+#",formf(coef(WN,t,i)/WD,0);
  ift i print "* ";justl$(t^i)
next i

```

Sortie (4360 ms)

```

W=
+1/4* [eSy]^-2* [x]^-2* [t^2 +1]^-1* [4*x^2*eSy^2 +eSy^4 -2*eSy^2 +1]*
[t]^2
-1/2* [eSy]^-2* [y]* [x]^-1* [t^2 +1]^-1* [eSy^4 -1]* [x^2 -1]^-1* [t]

```

$$+1/4* [eSy]^{-2*} [t^2 + 1]^{-1*} [x^2 - 1]^{-1*} [4*x^2*eSy^2 - eSy^4 - 2*eSy^2 - 1]$$

Etats quasistationnaires

Lorsque $S = \infty$, c'est à dire lorsque la barrière est infinie, pour $(n - 1/2)\pi < \alpha < (n + 1/2)\pi$ le puits de potentiel $U(r)$ supporte n états liés. Les énergies x^2 de ces états s'obtiennent par résolution de l'équation :

$$\cot \alpha x + \sqrt{1 - x^2}/x = 0 \quad (9.7)$$

qui est la forme limite de l'équation (9.5) quand $S \rightarrow \infty$. En effet, en négligeant e^{-Sy} devant e^{Sy} , l'équation (9.5) écrit que l'expression suivante est nulle :

$$\text{print formf}(t^2/x^2+2*t/x/y+1/y^2)$$

Sortie (220 ms)

$$[y]^{-2*} [x]^{-2*} [t*y + x]^2$$

et (9.7) exprime que $t*y+x$ est nul.

Lorsque S est fini, les états liés se transforment en états quasistationnaires d'énergie x_0^2 . L'intensité de l'onde à l'infini $W(x)$ passe par un minimum $W(x_0)$ pour la valeur x_0 de x correspondant à un état quasistationnaire. Au voisinage de x_0 on peut écrire (formule de Breit-Wigner) :

$$W(x) \approx F((x - x_0)^2 + \Gamma^2/4) \quad (9.8)$$

où F et Γ sont des constantes, Γ étant la largeur de l'état quasistationnaire (ou l'inverse de la durée de vie).

Les solutions x de l'équation (9.7) sont inférieures à 1. Les oscillations de la fonction $W(x)$ continuent pour $x > 1$, mais dans ce cas les minimums correspondent à des états (appelés états quasistationnaires oscillatoires) qui ne tendent plus vers des états liés dans la limite $S \rightarrow \infty$. La différence physique des deux sortes d'états quasistationnaires est reflétée par la probabilité de présence de la particule dans la barrière, qui est grande dans le cas oscillatoire ($x > 1$) et décroît presque exponentiellement dans le cas $x < 1$. Cependant les deux sortes d'états quasistationnaires sont très similaires puisque en variant S et α on peut transformer une sorte en l'autre.

Dans le programme suivant, pour la valeur $\alpha = 7$ on détermine les états liés ($S = \infty$), puis pour $S = 10$ on trace la courbe $\log W(x)$. D'abord les états liés sont déterminés par résolution de l'équation (9.7) par la méthode de dichotomie. Il y a exactement une racine en x de (9.7) dans chaque intervalle de la forme $((j - 1/2)\pi/\alpha, j\pi/\alpha)$, où j est entier tel que $(j - 1/2)\pi/\alpha < 1$, et toutes les racines s'obtiennent ainsi. Pour $\alpha = 7$, il y a $n = 2$ racines, et pour

α quelconque il y en a $n = \lfloor (\alpha + \pi/2)/\pi \rfloor$. La commande `print /c/` vide tout l'écran (après `cursh 0`), et le résultat est affiché à droite de l'écran en utilisant la fonction `justr$` (on suppose ici aussi l'utilisation de la haute résolution). Pour calculer la racine x_j de l'équation (9.7) ($j = 1, \dots, n$) le programme part des valeurs $a = (1 - \epsilon)(j - 1/2)\pi/\alpha$ et $b = (1 - \epsilon)j\pi/\alpha$ (ou $b = 1$ si $j = n$), où $\text{eps} = \epsilon = 2^{-\text{precision}2}$ est la précision des calculs en flottants. Le nombre différent de zéro $\epsilon \ll 1$ est utilisé pour éviter une division par zéro dans le calcul de la fonction $G(x)$, qui désigne le membre de gauche de l'équation (9.7). Si $G(a)G(b) \geq 0$, c'est que la racine est b à ϵ près. Sinon, a et b encadrent la racine cherchée, et la méthode de dichotomie peut commencer. L'intervalle $(a, b]$ est remplacé par l'intervalle de longueur moitié $(a, (a+b)/2]$ ou $((a+b)/2, b]$ qui contient la racine. Ce remplacement est répété tant que la longueur de $(a, b]$ reste plus grande que ϵ . La table `bs` contient les valeurs des racines.

Le calcul de $W(x)$ est effectué dans la fonction `FW` par substitution de valeurs flottantes dans l'expression formelle `W`. Pour les valeurs $x > 1$ il suffit d'effectuer les calculs en nombres complexes. On ne peut pas substituer $x = 0$ et $x = 1$ directement dans `W` car cela provoque une division par zéro. Il est possible de calculer les valeurs en ces points par passage à la limite, ce que nous ferons d'ailleurs pour $x \rightarrow 1$ dans la suite. Ici une méthode beaucoup plus simple est utilisée. On remplace x par des valeurs très proches de 0 ou 1, tout en effectuant les calculs en précision 20. Nous traçons la courbe de $F(x) = (\log W(x))/10$ qui permet de mieux distinguer les oscillations que $W(x)$. Noter que par suite des erreurs numériques dues à la précision finie des calculs flottants, le résultat de `WF`, pour $x > 1$, contient une petite partie imaginaire alors que $W(x)$ est réel positif. La fonction `cabs` qui calcule le module d'un nombre complexe permet de se débarrasser de cette partie imaginaire. Les minimums très étroits en $x \approx x_1$ et $x \approx x_2$ rendent le tracé de la courbe un peu délicat. Pour bien rendre ces minimums, la courbe est tracée en trois morceaux, pour x variant de 0 à x_1 , puis de x_1 à x_2 et enfin de x_2 à 1.5. Le pas du tracé `p1` dans un intervalle donné est choisi voisin de la valeur `p` et tel les extrémités de l'intervalle soient distantes d'un nombre entier de pas. Le tracé complet prend environ 2 minutes pour un pas `p1` qui correspond à 2 pixels sur l'écran.

```
alpha=7~
S=10
cursh 0
n=(alpha+pi/2)\pi
print /c/justr$("alpha."&alpha&"          S."&S,79)
print justr$(n&" états liés (pour S infini):",79)
var bs(n+1)
if n
  eps=2~^-precision2
  for j=1,n
    a=(j-1/2)*pi*(1-eps)/alpha
    b=min(j*pi*(1-eps)/alpha,1)
```

```

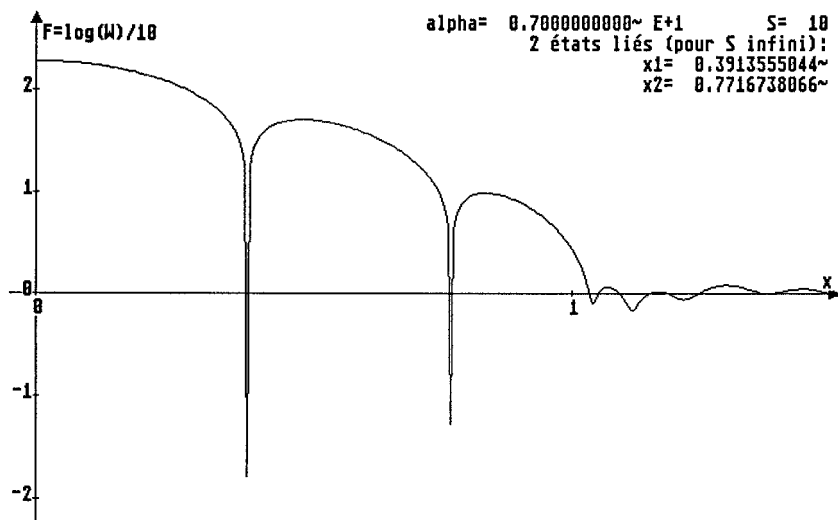
A=G(a)
B=G(b)
ift A*B>=0 a=b
while b-a>eps
  c=(a+b)/2
  C=G(c)
  if C*B>=0
    b=c
    B=C
  else
    a=c
    A=C
  endif
wend
bs(j)=b
c$="x"&justl$(j)
print justr$(c$&"="&b,79)
next j
endif
W=t^2/[t^2 +1]*[(eSy -1/eSy)^2+4*x^2]/4/x^2
W=W+t/[t^2 +1]*(eSy^2 -1/eSy^2)/2/x/y
W=W+1/[t^2 +1]*[(eSy +1/eSy)^2-4*x^2]/4/y^2
complex i
bs(0)=0~
bs(n+1)=1.5~
sx=620/bs(n+1)
sy=80
p=2/sx
x0=20
y0=220
axis x0,y0, 0,399,640,0, sx,sy, "x","F=log(W)/10"
hidecm
for j=0,n
  p1=(bs(j+1)-bs(j))/cint((bs(j+1)-bs(j))/p)
  fplot bs(j),bs(j+1)+p1/2,p1, x0,y0, sx,sy, ,F
next j
ift inp(2)
stop
G:function(u)
  value=1/tan(alpha*u)+sqr(1-u^2)/u
  return
F:function(u)
  value=log(cabs(FW(u)))/10
  return

```

```

FW:function(u)
    local var vy
    if abs(u)<2^-500
        precision 20
        ift u=0 u=2^-500
    endif
    if abs(u-1)<2^-30
        precision 20
        ift u=1 u=1-2^-40
    endif
    vy=sqr(1-u^2)
    value=fsubs(W,x=u,y=vy,eSy=exp(S*vy),t=tan(alpha*u))
    precision 10
    return

```

Figure 9.2. Etats liés et courbe $(\log W(x))/10$

Etude de $W(x)$ au voisinage de $x = 1$

Lorsque $x = 1$, $W(x)$ et ses dérivées $W'(x) = dW(x)/dx$ et $W''(x) = d^2W(x)/dx^2$ sont des formes indéterminées $0/0$. Le programme suivant calcule $W'(x)$ et $W''(x)$ ainsi que leurs limites quand $x \rightarrow 1$. Le calcul de la valeur limite de $W = WN/WD$, où $WN = \text{num}(W)$ et $WD = \text{den}(W)$ sont des polynômes, s'obtient en déterminant des expressions équivalentes $WD \sim Dy^k$, et $WN = Ny^k + o(y^k)$ pour $y \rightarrow 0$. Pour cela, dans la fonction `lim`, on effectue des développements limités de WD en $y = 0$ suivant les ordres croissants $k = 0, 1, \dots$, jusqu'à trouver un équivalent non nul Dy^k . $H = 1 - y^2/2 - y^4/8 + \dots$ est le développement limité de $x = \sqrt{1 - y^2}$. La fonction `lim1` effectue le développement limité à l'ordre k , en supprimant dès que possible tous les termes contenant y^{k+1} . Ces termes sont éliminés de W par `mod(W, y^{k+1})` (cette méthode est plus rapide que `W=subsr(W,y^(k+1)=0)`). Les termes `eSyj` sont remplacés par le développement limité de e^{jSy} (on gagne en rapidité par rapport au seul remplacement de `eSy`).

La dérivée par rapport à x d'une expression est calculée par la fonction `derx`, en tenant compte du fait que les littéraux `y`, `eSy` et `t` dépendent de `x`. Le littéral `a` représente α .

```

W=t^2/[t^2 +1]*[(eSy -1/eSy)^2+4*x^2]/4/x^2
W=W+t/[t^2 +1]*(eSy^2 -1/eSy^2)/2/x/y
W=W+1/[t^2 +1]*[(eSy +1/eSy)^2-4*x^2]/4/y^2
print "W(x=1)=";lim(W)
D=derx(W)
print "W'(x=1)=";lim(D)
DS=derx(D)
print "W''(x=1)=";lim(DS)
stop
lim:function(W)
    local var WD,WS,H
    local index i,j,k
    WD=den(W)
    k=0
    do
        H=shyg(y^2,k,y,-1/2,1,1,-1)
        WS=lim1(WD)
        ift WS exit
        k=k+1
    loop
    value=lim1(num(W))/WS
    return

```

```

lim1:function(W)
  W=subs(W,x=H)
  W=mod(W,y^(k+1))
  do
    j=deg(W,eSy)
    ift j=0 exit
    W=subsr(W,eSy^j=sexp(j*S*y,k,y))
    W=mod(W,y^(k+1))
  loop
  value=mod(W/y^k,y)
  return
derx:function(W)
  value=der(W,x)
  vadd value,-(der(W,y)+der(W,eSy)*S*eSy)*x/y
  vadd value,der(W,t)*(1+t^2)*a
  value=subsrr(value,y^2=1-x^2)
  return

```

Sortie (120 s)

```

W(x=1)= [t^2 +1]^-1* [t^2 +2*t*S +S^2 +1]
W'(x=1)= -2/3* [S]* [t^2 +1]^-1* [3*t^2*S +3*t^2*a +4*t*S^2 +3*t*S*a +
3*t +S^3 -3*a]
W''(x=1)= 2/45* [S]* [t^2 +1]^-1* [60*t^2*S^3 +120*t^2*S^2*a +45*t^2*
S*a^2 +135*t^2*S +90*t^2*a +48*t*S^4 +60*t*S^3*a +60*t*S^2 -180*t*S*a
-180*t*a^2 +90*t +8*S^5 -15*S^3 -120*S^2*a -45*S*a^2 -90*a]

```

Etats quasistationnaires en $x = 1$

La condition d'existence d'un état quasistationnaire pour $x = 1$ est donnée par les conditions :

$$\left. \frac{dW}{dx} \right|_{x=1} = 0 \quad \text{et} \quad \left. \frac{d^2W}{dx^2} \right|_{x=1} > 0. \quad (9.9)$$

Dans la limite $S \rightarrow \infty$, on obtient $\alpha = \alpha_n = (n + 1/2)\pi$ (n entier). Nous nous proposons d'obtenir le développement en série de Laurent des valeurs de α en fonction de S :

$$\alpha = \alpha_n + \frac{1}{S} - \frac{11}{6S^3} + \frac{3\alpha_n}{4S^4} + \left(\frac{233}{40} - \frac{9\alpha_n^2}{8} \right) \frac{1}{S^5} + \dots \quad (9.10)$$

Le résultat de la section précédente montre que la condition $\left. \frac{dW}{dx} \right|_{x=1} = 0$ s'exprime par $M = 0$ en fonction de $s = 1/S$. Nous posons $\alpha = \alpha_n + u$ et cherchons

le premier terme du développement (9.10) $u = a_1 s + \dots$. Ce développement et celui de $t = \tan \alpha = -\cot u = -1/u + \dots$ sont substitués au premier ordre dans M. En annulant le terme de plus bas degré en s , on obtient l'équation $(a_1 - 3)(a_1 - 1) = 0$. La racine $a_1 = 3$ correspond aux maximums de la fonction $W(x)$ et seule la racine $a_1 = 1$ qui correspond aux minimums doit être retenue.

```
M=3*a*t^2*s^3 +3*a*t*s^2 -3*a*s^3 +3*t^2*s^2 +3*t*s^3 +
4*t*s +1
W=subs(M,a=a0+u,t=-1/u,u=a1*s)
print formf(coeff(W,s,ordf(W,s)))
```

Sortie (450 ms)

```
[a1]^(-2)* [a1 -3]* [a1 -1]
```

Nous cherchons maintenant les termes suivants du développement (9.10). La fonction `st(n)` calcule le développement d'ordre n de $-\cot u$. Par exemple, pour $n = 3$ on obtient :

$$-\frac{1}{u} + \frac{u}{3} + \frac{u^3}{45}. \quad (9.11)$$

La variable `dev` contient la partie calculée du développement de u en fonction de s . Nous partons du résultat précédent `dev = s`. Lorsque `dev` est connu à l'ordre $n - 1$, le terme suivant du développement s'obtient en portant $u = \text{dev} + a_n s^n$ dans M et en annulant le terme de degré le plus bas en s . L'écriture de M en fonction de s est réalisée en plusieurs étapes. D'abord on porte α et t , écrits en fonction de u , dans M et on rejette les termes d'ordre supérieur à u^n . Ensuite la substitution de u en fonction de s donne une équation du premier degré en a_n . La solution de cette équation, effectuée par la fonction `sroot`, détermine le terme cherché P du développement.

```
M=3*a*t^2*s^3 +3*a*t*s^2 -3*a*s^3 +3*t^2*s^2 +3*t*s^3 +
4*t*s +1
dev=s
print "a=a0 + s";
for n=2,5
    W=subs(M,a=a0+u,t=st(n))
    W=num(W)
    W=subsr(W,u^(n+1)=0)
    W=subs(W,u=dev+an*s^n)
    m=ord(W,s)
    P=coef(W,s,m)
    ift deg(P,an)<>1 erreur
    P=sroot(P,an)*s^n
    ift P print using " +#",P;
    dev=dev+P
next n
stop
st:function(n)
    value=-taylor(scot(u,n+2)/ssin(u,n+2),n+2)
```

return

Sortie (7130 ms)

a=a0 + s -11/6*s^3 +3/4*s^4*a0 -9/8*s^5*a0^2 +233/40*s^5

Largeur des états quasistationnaires en $x = 1$

La largeur Γ définie par la formule de Breit-Wigner (9.8) est donnée par :

$$\Gamma^2 = \frac{8W}{W''} \Big|_{x=1}. \quad (9.12)$$

Nous nous proposons d'établir le développement en série de Laurent de Γ en fonction de S :

$$\Gamma = 3 \frac{1}{S^3} - \frac{9\alpha_n}{2} \frac{1}{S^4} + \frac{9(30\alpha_n^2 - 61)}{40} \frac{1}{S^5} + \dots \quad (9.13)$$

Nous substituons dans $G2 = (8W/W'')|_{x=1}$ le développement (9.10) de α en fonction de s . La variable u est une variable qui contient le développement de $\alpha - \alpha_n$. Noter le terme supplémentaire $I1?*s^6$ qui indique que les termes d'ordres supérieurs n'ont pas été calculés. Le littéral $I1?$, conjointement avec les littéraux $I2?$, $I3?$ et $I4?$ utilisés de façon analogue, nous permettra de déterminer jusqu'à quel ordre est valable le développement obtenu de Γ^2 . En effet, l'expression $G2$ étant compliquée, il est pénible de déterminer cet ordre à la main. Les variables $um1$, $u1$ et $u3$ contiennent les développements limités de $1/u$, u et u^3 jusqu'aux termes en s^3 . On en tire le développement limité Dt de $\tan \alpha = -\cot u$. Les variables Da , $Da2$ et $Da3$ sont initialisées avec les développements limités de α , α^2 et α^3 . La réduction de $G2$ modulo M ($M=0$ correspond à l'équation $\frac{dW}{dx}|_{x=1} = 0$) permet de simplifier $G2$ avant la substitution de Dt à t par élimination des termes en t^2 qui sont très coûteux à calculer. Après les substitutions, le numérateur de $G2$, $G2N = 9s^6 + \dots$, se trouve être valable jusqu'au terme en s^8 . La troncation de $G2N$ et du dénominateur $G2D$, puis le développement de leur rapport donnent le développement limité de Γ^2 . Le développement limité de $\sqrt{1-f(s)}$ autour de $s = 0$ d'une expression rationnelle $f(s)$ telle que $f(s) = 0$ pour $s = 0$ s'obtient à l'ordre k par `shyg(f(s),k,s,-1/2,1,1,-1)`. Le développement limité $G1$ de Γ s'obtient ainsi à partir de celui de $\Gamma^2 = (3s^3)^2(1-f(s))$. L'impression par `str$` est ordonnée suivant les puissances croissantes de s .

```
u=s-11/6*s^3 +3/4*s^4*a0 -9/8*s^5*a0^2 +233/40*s^5+I1?*
s^6
um1=taylor(1/u,4,s)
u1=mod(u,s^4)
```

```

u3=mod(u^3,s^4)
Dt= (-1)*um1+( 1/3)*u1+( 1/45)*u3 +I2?*s^4
Da=a0 + u
Da2=mod(Da^2,s^6)+I3?*s^6
Da3=mod(Da2*Da,s^6)+I4?*s^6
S=1/s
M=3*a*t^2*s^3 +3*a*t*s^2 -3*a*s^3 +3*t^2*s^2 +3*t*s^3 +
  4*t*s +1
W0= [t^2 +1]^-1* [t^2 +2*t*S +S^2 +1]
W2= 2/45* [S]* [t^2 +1]^-1* [45*a^2*t^2*S -180*a^2*t -
  45*a^2*S +120*a*t^2*S^2 +90*a*t^2 +60*a*t*S^3 -180*a*t
  *S -120*a*S^2 -90*a +60*t^2*S^3 +135*t^2*S +48*t*S^4 +
  60*t*S^2 +90*t +8*S^5 -15*S^3]
G2=8*W0/W2
G2=mod(G2,M,t)
G2=subs(G2,t=Dt)
G2=subsr(G2,a^3=Da3)
G2=subsr(G2,a^2=Da2)
G2=subsr(G2,a=Da)
G2N=num(G2)
G2D=den(G2)
G2N=mod(G2N,s^9)
G2D=mod(G2D,s^3)
G2=taylor(G2N/G2D,2,s)
G1=3*s^3*shyg(1-G2/9/s^6,2,s,-1/2,1,1,-1)
print str$(G1,/s)

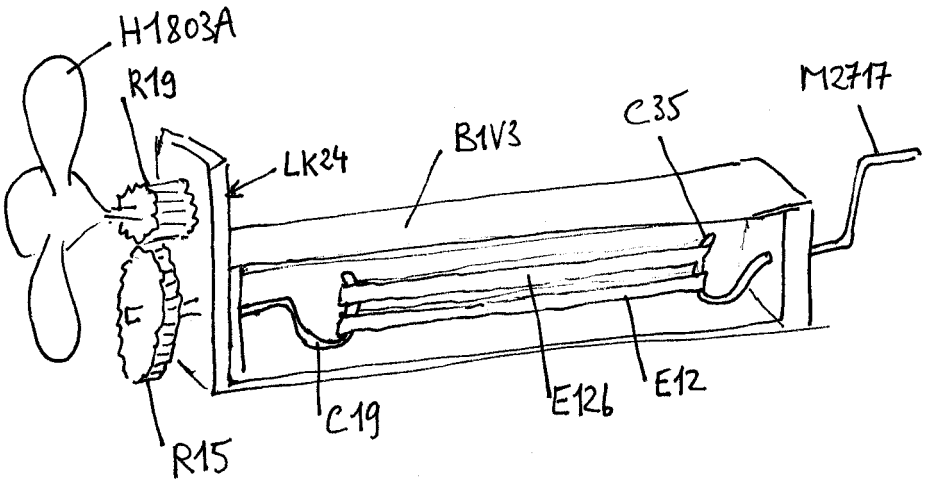
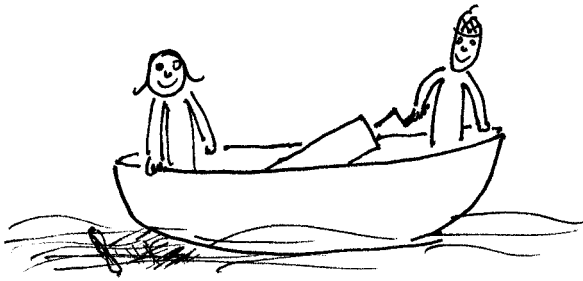
```

Sortie (25 s)

```
( 3)*s^3+( -9/2* [a0])*s^4+( 9/40* [30*a0^2 -61])*s^5
```


10

Etude d'un moteur



Le problème traité ici donne un exemple de l'aide que peut apporter notre Basic à l'art de l'ingénieur. Il s'agit de la modélisation d'un moteur asynchrone à rotor bobiné, à couplage étoile, alimenté par un onduleur triphasé. On se donne les tensions appliquées et la charge du moteur. L'objectif est de décrire le régime permanent du moteur. Comme nous nous intéressons ici seulement à l'aspect mathématique du problème, notre point de départ est l'équation donnant les courants I_{sa} , I_{sb} , I_{rd} et I_{rq} dans divers bobinages du stator et rotor. Cette équation est une équation matricielle qui ressemble à la loi d'Ohm $U = RI$. C'est :

$$U = R_E I - p\Omega X_E L_E I + L_E \frac{dI}{dt}, \quad (10.1)$$

où U (grandeur connue) et I (l'inconnue) sont des vecteurs :

$$U = \begin{pmatrix} U_1 \\ U_2 \\ 0 \\ 0 \end{pmatrix} \quad I = \begin{pmatrix} I_{sa} \\ I_{sb} \\ I_{rd} \\ I_{rq} \end{pmatrix} \quad (10.2)$$

et où R_E (résistance), X_E (connexions) et L_E (inductance) sont des matrices :

$$R_E = \begin{pmatrix} R_s & 2R_s & 0 & 0 \\ -2R_s & -R_s & 0 & 0 \\ 0 & 0 & R_r & 0 \\ 0 & 0 & 0 & R_r \end{pmatrix} \quad X_E = \begin{pmatrix} 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & -1 & 0 \end{pmatrix}$$

$$L_E = \begin{pmatrix} L_s & 2L_s & 0 & -\sqrt{2}M \\ -2L_s & -L_s & -\sqrt{3/2}M & M/\sqrt{2} \\ \sqrt{3/2}M & 0 & L_r & 0 \\ -M/\sqrt{2} & -\sqrt{2}M & 0 & L_r \end{pmatrix}. \quad (10.3)$$

Les tensions U_1 et U_2 sont des fonctions en escalier périodiques déphasées de $2\pi/3$, de fréquence $f = 50$ Hz et de période $T = 1/f$. Elles sont constantes sur des paliers de durée $T/6$. Les valeurs successives pendant les 6 paliers d'une période sont $(-H, 0, H, H, 0, -H)$ pour U_1 et $(0, -H, -H, 0, H, H)$ pour U_2 où $H = 510$ V.

Les résistances et inductances des bobinages sont $R_s = 1$ ohm, $R_r = 0.12$ ohm, $L_s = 0.24$ mH, $L_r = 14.7$ mH et $M = 56.2$ mH.

La vitesse de rotation Ω est liée au couple par l'équation du moment cinétique :

$$J \frac{d\Omega}{dt} = C_e - C_r. \quad (10.4)$$

où le couple moteur est :

$$C_e = pM \left[3I_{rq}I_{sa} + I_{rd}\sqrt{3}(2I_{sb} + I_{sa}) \right] / \sqrt{6}, \quad (10.5)$$

$C_r = 10$ Nm est un couple résistant constant, $J = 0.2$ kg·m² est le moment d'inertie. Le nombre de paires de pôles est $p = 2$.

Ecriture formelle des équations

Nous nous proposons de récrire les équations différentielles modélisant le moteur sous la forme

$$dy/dt = f(t, y), \quad (10.6)$$

où t désigne le temps et où

$$y = (\Omega, I_{sa}, I_{sb}, I_{rd}, I_{rq}) \quad (10.7)$$

regroupe les inconnues du problème.

Le programme suivant effectue cette réécriture pour les courants, d'abord de façon entièrement symbolique, puis pour les valeurs numériques. La méthode utilise quelques calculs matriciels simples, notamment l'inversion de la matrice L_E qui est effectuée par la procédure `invm` de la bibliothèque MATH. On notera que les racines carrées de 2 et 3 sont représentées exactement par les littéraux `sqr2` et `sqr3` qui vérifient les conditions `sqr2^2=2` et `sqr3^2=3`.

```

rem -----
rem Matrice R_E
rem -----
dim R_E(3,3)
for i=0 to 3
  for j=0 to 3
    read R_E(i,j)
  next j
next i
data  Rs, 2*Rs, 0, 0
data -2*Rs, -Rs, 0, 0
data  0, 0, Rr, 0
data  0, 0, 0, Rr
rem -----
rem Matrice X_E
rem -----
dim X_E(3,3)
for i=0 to 3
  for j=0 to 3
    read X_E(i,j)
  next j
next i
data 0, 0, 0, 0
data 0, 0, 0, 0
data 0, 0, 0, 1

```

```

data 0, 0,-1, 0
rem -----
rem Matrice L_E
rem -----
rem sqr2=sqr(2), sqr3=sqr(3)
cond sqr2^2-2
cond sqr3^2-3
dim L_E(3,3)
for i=0 to 3
  for j=0 to 3
    read L_E(i,j)
  next j
next i
data      Ls,      2*Ls,      0, -sqr2*M
data      -2*Ls,      -Ls, -sqr3*sqr2/2*M, sqr2/2*M
data  sqr3*sqr2/2*M,      0,      Lr,      0
data      -sqr2/2*M, -sqr2*M,      0,      Lr
rem -----
rem Vecteur U
rem -----
dim U(3)
for i=0 to 3
  read U(i)
next i
data  U1, U2, 0, 0
rem -----
rem Vecteur I
rem -----
dim I(3)
for i=0 to 3
  read I(i)
next i
data  Isa, Isb, Ird, Irq
rem -----
rem Calcul de l'inverse LI de la matrice L_E
rem -----
dim LI(3,3)
invm L_E,LI,3
rem -----
rem Impression de LI
rem -----
print "Inverse de la matrice L_E"
print "-----"
for i=0,3

```

```

    for j=0,3
        print "LI(";justl$(i);",";justl$(j);")=";LI(i,j)
    next j
next i
rem
rem Ci-dessous le point "." note le produit matriciel.
rem Transformation de l'équation
rem  $U = R_E \cdot I - p \cdot \omega \cdot X_E \cdot L_E \cdot I + L_E \cdot dI/dt$ 
rem en la forme résolue en  $dI/dt$ :
rem  $dI/dt = DerI$  où
rem  $DerI = LI \cdot U - LI \cdot R_E \cdot I + p \cdot \omega \cdot LI \cdot X_E \cdot$ 
rem  $L_E \cdot I$ 
rem -----
rem
rem Calcul de DerI
rem -----
dim DerI(3)
for i=0,3
    DerI(i)=sum(j=0,3 of LI(i,j)*U(j))
    DerI(i)=DerI(i)-sum(j=0,3 of LI(i,j)*sum(k=0,3 of R_E
        (j,k)*I(k)))
    DerI(i)=DerI(i)+p*omega*sum(j=0,3 of LI(i,j)*sum(k=0,
        3 of X_E(j,k)*sum(l=0,3 of L_E(k,l)*I(l)))
next i
rem
rem Impression des équations transformées
rem (forme littérale)
print "-----"
print "Equations des courants"
print "-----"
for i=0,3
    print "d(";justl$(I(i));)/dt=";DerI(i)
    print
next i
rem
rem Substitutions numériques
rem -----
rem p=2
rem Rs=1
rem Rr=0.12
rem Ls=0.24
rem Lr=0.0147
rem M=0.0562
rem  $sqr2=v2=appr(\sqrt{2})$ 

```

```

rem sqr3=v3=appr(sqr(3))
rem -----
rem
v2=appr(sqr(2))
v3=appr(sqr(3))
for i=0,3
    DerI(i)=subs(DerI(i),p=2,Rs=1,Rr=0.12,Ls=0.24,Lr=0.01
        47,M=0.0562)
    DerI(i)=subs(DerI(i),sqr2=v2,sqr3=v3)
next i
rem
rem Impression des équations
rem (forme partiellement numérisée)
rem -----
print "-----"
print "Equations seminumériques des courants"
print "-----"
notilde
formatx -11
for i=0,3
    print "d(";justl$(I(i));)/dt=";DerI(i)
    print
next i
formatx 0
stop

```

Sortie (22 s)

Inverse de la matrice L_E

```

-----
LI(0,0)= -1/3* [Lr]* [Ls*Lr -M^2]^-1
LI(0,1)= -2/3* [Lr]* [Ls*Lr -M^2]^-1
LI(0,2)= -1/3* [M]* [sqr3]* [sqr2]* [Ls*Lr -M^2]^-1
LI(0,3)= 0
LI(1,0)= 2/3* [Lr]* [Ls*Lr -M^2]^-1
LI(1,1)= 1/3* [Lr]* [Ls*Lr -M^2]^-1
LI(1,2)= 1/6* [M]* [sqr3]* [sqr2]* [Ls*Lr -M^2]^-1
LI(1,3)= 1/2* [M]* [sqr2]* [Ls*Lr -M^2]^-1
LI(2,0)= 1/6* [M]* [sqr3]* [sqr2]* [Ls*Lr -M^2]^-1
LI(2,1)= 1/3* [M]* [sqr3]* [sqr2]* [Ls*Lr -M^2]^-1
LI(2,2)= [Ls]* [Ls*Lr -M^2]^-1
LI(2,3)= 0
LI(3,0)= 1/2* [M]* [sqr2]* [Ls*Lr -M^2]^-1
LI(3,1)= 0
LI(3,2)= 0
LI(3,3)= [Ls]* [Ls*Lr -M^2]^-1

```

Equations des courants

$$d(I_{sa})/dt = -1/3 * [L_s * L_r - M^2]^{-1} * [3 * R_s * L_r * I_{sa} - R_r * \text{sqr2} * \text{sqr3} * M * I_{rd} + \text{sqr2} * \text{sqr3} * M * L_r * I_{rq} * p * \omega - \text{sqr3} * M^2 * I_{sa} * p * \omega - 2 * \text{sqr3} * M^2 * I_{sb} * p * \omega + L_r * U_1 + 2 * L_r * U_2]$$

$$d(I_{sb})/dt = -1/6 * [L_s * L_r - M^2]^{-1} * [6 * R_s * L_r * I_{sb} + R_r * \text{sqr2} * \text{sqr3} * M * I_{rd} + 3 * R_r * \text{sqr2} * M * I_{rq} - \text{sqr2} * \text{sqr3} * M * L_r * I_{rq} * p * \omega + 3 * \text{sqr2} * M * L_r * I_{rd} * p * \omega + 4 * \text{sqr3} * M^2 * I_{sa} * p * \omega + 2 * \text{sqr3} * M^2 * I_{sb} * p * \omega - 4 * L_r * U_1 - 2 * L_r * U_2]$$

$$d(I_{rd})/dt = 1/6 * [L_s * L_r - M^2]^{-1} * [3 * R_s * \text{sqr2} * \text{sqr3} * M * I_{sa} - 6 * R_r * L_s * I_{rd} + \text{sqr2} * \text{sqr3} * M * U_1 + 2 * \text{sqr2} * \text{sqr3} * M * U_2 - 3 * \text{sqr2} * L_s * M * I_{sa} * p * \omega - 6 * \text{sqr2} * L_s * M * I_{sb} * p * \omega + 6 * L_s * L_r * I_{rq} * p * \omega]$$

$$d(I_{rq})/dt = -1/2 * [L_s * L_r - M^2]^{-1} * [R_s * \text{sqr2} * M * I_{sa} + 2 * R_s * \text{sqr2} * M * I_{sb} + 2 * R_r * L_s * I_{rq} + \text{sqr2} * \text{sqr3} * L_s * M * I_{sa} * p * \omega - \text{sqr2} * M * U_1 + 2 * L_s * L_r * I_{rd} * p * \omega]$$

Equations seminumériques des courants

$$d(I_{sa})/dt = -0.1325901072 \text{ E}+2 * U_1 - 0.2651802143 \text{ E}+2 * U_2 + 0.9868633966 \text{ E}+1 * I_{sa} * \omega - 0.3977703215 \text{ E}+2 * I_{sa} + 0.1973726793 \text{ E}+2 * I_{sb} * \omega + 0.1490002420 \text{ E}+2 * I_{rd} - 0.3650505928 \text{ E}+1 * I_{rq} * \omega$$

$$d(I_{sb})/dt = 0.2651802143 \text{ E}+2 * U_1 + 0.1325901072 \text{ E}+2 * U_2 - 0.1973726793 \text{ E}+2 * I_{sa} * \omega - 0.9868633966 \text{ E}+1 * I_{sb} * \omega - 0.3977703215 \text{ E}+2 * I_{sb} - 0.3161430870 \text{ E}+1 * I_{rd} * \omega - 0.7450012098 \text{ E}+1 * I_{rd} + 0.1825252964 \text{ E}+1 * I_{rq} * \omega - 0.1290379947 \text{ E}+2 * I_{rq}$$

$$d(I_{rd})/dt = 0.6208343415 \text{ E}+2 * U_1 + 0.1241668683 \text{ E}+3 * U_2 - 0.5161519788 \text{ E}+2 * I_{sa} * \omega + 0.1862503024 \text{ E}+3 * I_{sa} - 0.1032303958 \text{ E}+3 * I_{sb} * \omega - 0.7793051196 \text{ E}+2 * I_{rd} + 0.1909297543 \text{ E}+2 * I_{rq} * \omega$$

$$d(I_{rq})/dt = 0.1075316623 \text{ E}+3 * U_1 - 0.8940014517 \text{ E}+2 * I_{sa} * \omega - 0.1075316623 \text{ E}+3 * I_{sa} - 0.2150633245 \text{ E}+3 * I_{sb} - 0.1909297543 \text{ E}+2 * I_{rd} * \omega - 0.7793051196 \text{ E}+2 * I_{rq}$$

Résolution numérique

Le système $dy/dt = f(t, y)$ est intégré numériquement par une méthode Runge-Kutta d'ordre 4. La dépendance en t de f est contenue dans les tensions U_1 et U_2 . Pendant le temps $T/6$ d'un palier, la fonction $f(t, y) = g(y)$ ne dépend pas explicitement du temps t . La méthode d'intégration calcule la valeur $y + \Delta y$ au temps $t + h$ en fonction de la valeur y au temps t par les formules :

$$\begin{cases} k_1 = hg(y) \\ k_2 = hg(y + k_1/2) \\ k_3 = hg(y + k_2/2) \\ k_4 = hg(y + k_3) \\ \Delta y = (k_1 + 2k_2 + 2k_3 + k_4)/6. \end{cases} \quad (10.8)$$

En partant de $y = (0, 0, 0, 0, 0)$ à l'instant $t = 0$, on atteint le régime stationnaire après une centaine de périodes. Le programme ci-dessous utilise une valeur initiale de y qui correspond au régime stationnaire. La variation de I_{sa} pendant une période est représentée graphiquement par appel de la procédure `qplot`.

```
rem adjoindre la procédure qplot
notilde'supprime ~ dans la sortie des flottants
rem valeurs numériques
rem -----
rem f=fréquence en Hz
rem T=période en secondes =1/f
rem H=saut de tension en Volts
rem -----
f=50
T=1/f
H=510
rem -----
rem Les 6 paliers de tension pendant une période
rem -----
dim u1(5),u2(5)
for i=0 to 5
  read u1(i)
  u2(modr(i+2,6))=u1(i)
next i
data -H,0,H,H,0,-H
rem -----
rem N=nombre de pas d'intégration par palier
rem h=T/(6*N) pas d'intégration
rem num_per=1, 2, ... numéro de la période
rem Les vecteurs suivants donnent
```



```

rem  (omega,Isa,Isb,Ird,Irq).
rem E valeur à un instant donné, entrée de la procédure
    calcul_accr
rem S accroissement, sortie de la procédure calcul_accr
rem k1, k2, k3, k4 accroissements utilisés pour la
    méthode Runge-Kutta
rem y(.,j) (j de 0 à 6*N) valeurs aux temps j*h d'une
    période
rem On intègre sur une période.
rem Si y(.,6*N) est voisin de y(.,0), le régime
    stationnaire est atteint.
rem Sinon, on répète l'intégration sur la période
    suivante.
rem -----
N=20
h=float(T/6/N)
print "pas d'intégration=";h;" s"
dim E(4),S(4),k1(4),k2(4),k3(4),k4(4)
dim y(4,6*N)
rem -----
rem                               Valeurs initiales
rem ces valeurs correspondent au régime stationnaire
rem -----
for i=0,4
    read y(i,0)
next i
data 0.1535703294~ E+3~, -0.8472457711~ E+1~, 0.141024
    1904~ E+1~, 0.1984716097~ E+2~, -0.1676977701~ E+2~
num_per=0
rem -----
rem boucle sur les périodes jusqu'au régime
    stationnaire
rem -----
do
    une_période'intègre sur une période
    v=sqr(sum(i=0,4 of (y(i,0)-y(i,6*N))^2))
    num_per=num_per+1
    print "Période=";num_per;" Non stationnarité=";v;"
        timer=";timer
    ift v<1.E-5 exit
    for i=0,4
        y(i,0)=y(i,6*N)
    next i
loop

```

```

rem -----
rem Tracé de la courbe Isa
rem -----
qplot 0,T,6*N,I_sa
ift inp(2)
stop
I_sa:function(x)
value=y(1,x*6*N/T)
return
rem -----
rem          procédure une_période
rem          =====
rem Intègre sur une période
rem k=numéro du palier (temps de k*T/6 à (k+1)*T/6)
rem j=numéro du point (temps j*h)
rem -----
une_période:procedure
print "  j      omega      Isa      Isb      Ird
      Irq      Ce"
for k=0 to 5
  for j=k*N,(k+1)*N-1
    for i=0,4
      E(i)=y(i,j)
    next i
    calcul_accr
    for i=0,4
      k1(i)=S(i)
      E(i)=y(i,j)+k1(i)/2
    next i
    calcul_accr
    for i=0,4
      k2(i)=S(i)
      E(i)=y(i,j)+k2(i)/2
    next i
    calcul_accr
    for i=0,4
      k3(i)=S(i)
      E(i)=y(i,j)+k3(i)
    next i
    calcul_accr
    for i=0,4
      k4(i)=S(i)
      y(i,j+1)=y(i,j)+(k1(i)+2*k2(i)+2*k3(i)+k4(i))/6
    next i
  
```

```

        print justr$(j,3);using "#####.#####",y(0,j+1),y(1,
            j+1),y(2,j+1),y(3,j+1),y(4,j+1),Ce
    next j
next k
return
rem -----
rem          procédure  calcul_accr
rem          =====
rem  Entrée E=(omega,Isa,Isb,Ird,Irq)
rem  Sortie S= h . d(E)/dt accroissement de E pendant le
rem          temps h
rem  Ce=couple (Les coefficients dans l'expression sont
rem  0.7947880221~ E-1~ = p*M/sqr(2)
rem  0.1732050808~ E+1~ =sqr(3)
rem  Dans S(0), 10=Cr et le facteur 5=1/J
rem  -----
calcul_accr:procedure
    omega=E(0)
    Isa=E(1)
    Isb=E(2)
    Ird=E(3)
    Irq=E(4)
    U1=u1(k)
    U2=u2(k)
    S(1)=h*(-0.1325901072~ E+2*U1 -0.2651802143 E+2*U2 +0.9
        868633966 E+1*Isa*omega -0.3977703215 E+2*Isa +0.19737
        26793 E+2*Isb*omega +0.1490002420 E+2*Ird -0.365050592
        8 E+1*Irq*omega)
    S(2)=h*(0.2651802143~ E+2*U1 +0.1325901072 E+2*U2 -0.19
        73726793 E+2*Isa*omega -0.9868633966 E+1*Isb*omega -0.
        3977703215 E+2*Isb -0.3161430870 E+1*Ird*omega -0.7450
        012098 E+1*Ird +0.1825252964 E+1*Irq*omega -0.12903799
        47 E+2*Irq)
    S(3)=h*(0.6208343415~ E+2*U1 +0.1241668683 E+3*U2 -0.51
        61519788 E+2*Isa*omega +0.1862503024 E+3*Isa -0.103230
        3958 E+3*Isb*omega -0.7793051196 E+2*Ird +0.1909297543
        E+2*Irq*omega)
    S(4)=h*(0.1075316623~ E+3*U1 -0.8940014517 E+2*Isa*omeg
        a -0.1075316623 E+3*Isa -0.2150633245 E+3*Isb -0.19092
        97543 E+2*Ird*omega -0.7793051196 E+2*Irq)
    Ce=0.7947880221~ E-1~*(0.1732050808~ E+1~*Irq*Isa+Ird*(
        2*Isb+Isa))
    S(0)=5*h*(Ce-10)
return

```

Sortie (410 s)

pas d'intégration= 0.1666666667 E-3 s

j	omega	Isa	Isb	Ird	Irq	Ce
0	153.57035	-7.16151	0.72564	13.63640	-15.88815	9.47641
1	153.56955	-5.97065	0.10272	8.04923	-14.99670	8.63956
2	153.56817	-4.89751	-0.46460	3.07361	-14.12512	8.10128
3	153.56645	-3.93946	-0.98239	-1.30374	-13.30204	7.82713
4	153.56461	-3.09365	-1.45662	-5.09731	-12.55503	7.78190
5	153.56281	-2.35703	-1.89319	-8.32271	-11.91055	7.92978

...

115	153.56459	-5.60264	0.19920	7.20657	-19.64979	12.17569
116	153.56640	-6.19522	0.41502	9.70982	-19.07736	12.13083
117	153.56808	-6.87152	0.69185	12.65330	-18.38964	11.87793
118	153.56946	-7.63090	1.02512	16.03386	-17.61213	11.39077
119	153.57033	-8.47246	1.41024	19.84716	-16.76978	10.64484

Période= 1 Non stationnarité= 0.6799213953 E-5 timer= 410

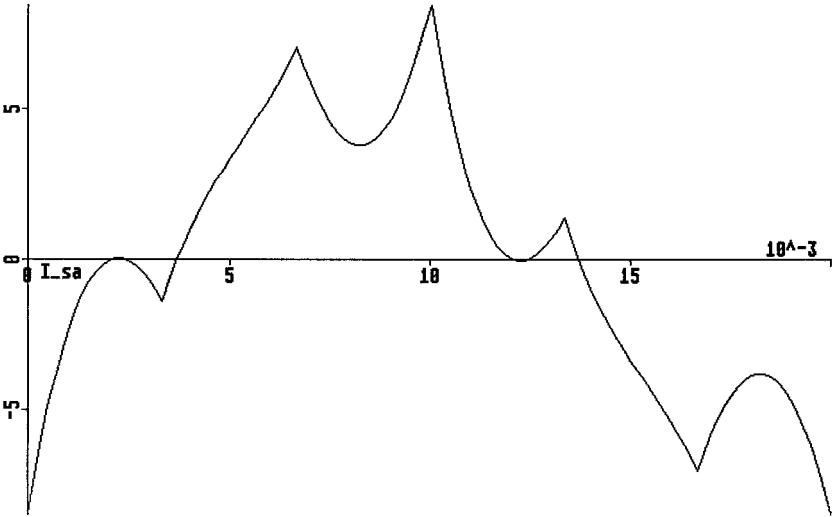
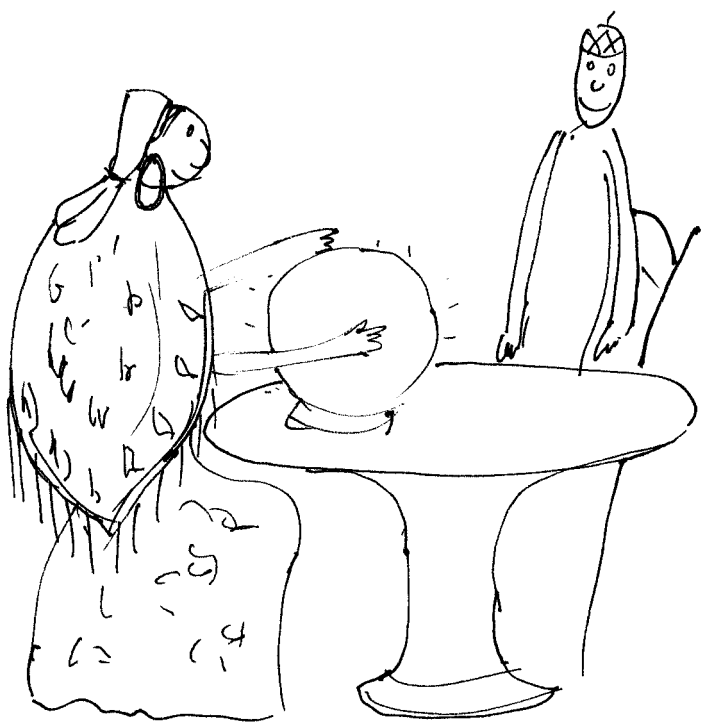


Figure 10.1. Courant statorique I_{sa} en régime permanent

11

Algèbres non commutatives



Il arrive souvent d'avoir à manipuler des symboles K, L, \dots qui se multiplient de façon non-commutative ($KL \neq LK$). Par exemple, K et L peuvent désigner des matrices carrées, avec lesquelles on désire travailler sans expliciter les éléments $K_{i,j}$ et $L_{i,j}$ à l'aide de tableaux. Il n'est pas possible d'utiliser directement le Basic 1000d pour effectuer des opérations sur les symboles K, L, \dots , puisque les expressions $K \star L$ et $L \star K$ sont codées par le même polynôme.

Les sous-programmes introduits ici vont nous permettre d'étendre le Basic 1000d à des calculs non-commutatifs. Nous utiliserons, en tant qu'exemple, l'algèbre enveloppante U de l'algèbre de Lie $\mathfrak{su}(2)$. Nous notons L_x, L_y et L_z des générateurs de l'algèbre de Lie $\mathfrak{su}(2)$ qui vérifient la relation de commutation

$$[L_x, L_y] = L_x L_y - L_y L_x = i L_z \quad (11.1)$$

ainsi que les relations qui s'en déduisent par permutations circulaires des indices x, y et z . Un élément a de U peut en général s'écrire de plusieurs façons en fonction des générateurs et de l'unité. Par exemple

$$a = 3 + L_x L_y = 3 + i L_z + L_y L_x = 3 + i L_z L_y^2 - i L_y L_z L_y = \dots \quad (11.2)$$

Définissons une relation d'ordre sur les générateurs, par exemple

$$L_x < L_y < L_z. \quad (11.3)$$

Nous dirons alors qu'une expression comme $L_x^3 L_z + L_y L_z$, où les générateurs sont ordonnés de gauche à droite dans les produits, est une forme normale. Tout élément de U possède une et une seule forme normale. Nous pouvons donc représenter tout élément de U par sa forme normale que nous coderons simplement par un polynôme du Basic, les générateurs L_x, L_y et L_z étant des littéraux. Si on a pris soin de définir les générateurs en suivant la relation d'ordre, ce polynôme s'écrit (par `print`) sous la forme normale.

Les opérations d'addition, soustraction et multiplication par une constante coïncident avec les opérations usuelles sur les polynômes, mais la multiplication est plus compliquée, puisque il faut transformer le résultat en forme normale. Cette transformation s'effectue à l'aide des relations de commutation. Par exemple, chaque fois que le produit non ordonné $L_y L_x$ apparaît, il est remplacé par la somme $L_x L_y - i L_z$.

Décrivons maintenant le programme.

Initialisations

Tout d'abord, on déclare le littéral complexe i et les $\mathbf{NG} = 3$ générateurs, qui sont enregistrés dans le tableau \mathbf{G} pour faciliter la généralisation à d'autres algèbres. L'ordre des générateurs est $\mathbf{G}(1) < \mathbf{G}(2) < \mathbf{G}(3)$. Ensuite, on initialise les éléments $\mathbf{GG}(i_1, i_2)$ du tableau \mathbf{GG} , pour $\mathbf{NG} \geq i_1 > i_2 \geq 1$, avec les commutateurs $[\mathbf{G}(i_1), \mathbf{G}(i_2)]$. Ce sont ces éléments qui servent à récrire $\mathbf{G}(i_1) \mathbf{G}(i_2)$ sous forme normale :

$$\mathbf{G}(i_1) \mathbf{G}(i_2) = \mathbf{G}(i_2) \mathbf{G}(i_1) + \mathbf{GG}(i_1, i_2). \quad (11.4)$$

Calcul d'un produit

Le calcul du produit AB , où A et B sont des formes normales s'obtient à l'aide de la fonction `PR` par `PR(A, B)`. La fonction `PR` décompose A et B en sommes de monômes et appelle la fonction `PRM(M, N)` qui transforme le produit MN de deux monômes en forme normale. Ainsi pour $a = \sum_j m_j$ et $b = \sum_k n_k$, où m_j et n_k sont des monômes, `PR(a, b)` est calculé par $\sum_{j,k} \text{PRM}(m_j, n_k)$.

Le calcul de `PRM(A, B)` se réduit parfois au produit usuel $A * B$ qui est calculé par la sortie `PRM1` de la fonction. C'est le cas lorsque A (ou B) ne dépend pas des générateurs. Lorsque $A = A_p G(i_1)$ et $B = G(i_2) B_p$ contiennent tous deux des générateurs, on détermine le générateur $G(i_2)$ le plus à gauche de B et le générateur $G(i_1)$ le plus à droite de A . Si $i_1 \leq i_2$, le produit s'obtient également par la sortie `PRM1` (dans ce cas, la sortie `PRM1` est effectuée sans déterminer i_1). Sinon, le produit est $AB = A_p(G(i_2)G(i_1) + [G(i_1), G(i_2)])B_p$ qui est calculé par la ligne :

```
value=PR(PRM(AP,G(i2)),PRM(G(i1),BP))+PR(PR(AP,GG(i1,i2)),BP)
```

Comme des appels récursifs sont utilisés, les index et variables des fonctions `PR` et `PRM` ont été déclarés par `local`.

Commutateurs et anticommutateurs

La fonction `COM(A, B)` calcule le commutateur $[A, B] = AB - BA$ et la fonction `ACOM(A, B)` calcule l'anticommutateur $\{A, B\} = AB + BA$. On trouvera aussi les procédures `PRCOM` et `PRACOM` qui calculent et écrivent ces mêmes expressions. L'exemple vérifie les relations :

$$[L_x + iL_y, L_x - iL_y] = 2L_z, \quad \{L_x + iL_y, L_x - iL_y\} = 2L_x^2 + 2L_y^2, \quad (11.5)$$

$$[\mathbf{L}^2, L_x] = [\mathbf{L}^2, L_y] = [\mathbf{L}^2, L_z] = 0, \quad (11.6)$$

où $\mathbf{L}^2 = L_x^2 + L_y^2 + L_z^2$.

```
Initialisations
PRCOM Lx+i*Ly,Lx-i*Ly
PRACOM Lx+i*Ly,Lx-i*Ly
L2=Lx^2+Ly^2+Lz^2
print "L2=";L2
for i1=1,NG
  print "[ L2,";G(i1);" ] =" ;COM(L2,G(i1))
next i1
stop
```

Initialisations:

```
complex i
rem -----
rem Algèbre U
rem Les NG générateurs G(1), ..., G(NG)
rem -----
NG=3
var G(NG)
```

```

G(1)=Lx
G(2)=Ly
G(3)=Lz
rem -----
rem Les commutateurs [ G(i1) , G(i2) ]=GG(i1,i2)
rem pour i1>i2
rem -----
var GG(NG,NG)
GG(2,1)=-i*Lz
GG(3,1)=i*Ly
GG(3,2)=-i*Lx
return
PR:function(A,B)
  local index i1,i2
  local var M,N
  for i1=1,polymn(A)
    for i2=1,polymn(B)
      vadd value,PRM(polym(A,i1),polym(B,i2))
    next i2,i1
  return
PRM:function(A,B)
  ift polyln(A)=0 or polyln(B)=0 goto PRM1
  local index i1,i2
  for i2=1,NG-1
    if deg(B,G(i2))
      for i1=NG,i2+1,-1
        if deg(A,G(i1))
          local var AP,BP
          AP=A/G(i1)
          BP=B/G(i2)
          value=PR(PRM(AP,G(i2)),PRM(G(i1),BP))+PR(PR(AP,
            GG(i1,i2)),BP)
          return
        endif
      next i1
      goto PRM1
    endif
  next i2
PRM1:value=A*B
return
COM:function(A,B)
  value=PR(A,B)-PR(B,A)
  return
PRCOM:procedure(A,B)

```



```

    print "[";A;" ,";B;" ] =" ;COM(A,B)
    return
ACOM:function(A,B)
    value=PR(A,B)+PR(B,A)
    return
PRACOM:procedure(A,B)
    print "{";A;" ,";B;" } =" ;ACOM(A,B)
    return

```

Sortie (10290 ms)

```

[ i*Ly +Lx , -i*Ly +Lx ] = 2*Lz
{ i*Ly +Lx , -i*Ly +Lx } = 2*Lx^2 +2*Ly^2
L2= Lx^2 +Ly^2 +Lz^2
[ L2, Lx ] = 0
[ L2, Ly ] = 0
[ L2, Lz ] = 0

```

Programme de décodage

La fonction **PR** permet d'effectuer tous les calculs dans l'algèbre U . Cependant, l'écriture des expressions peut devenir laborieuse. Par exemple pour calculer $[ABCD + E, F]$ on écrit une expression volumineuse comme **COM**(**PR**(A , **PR**(B , **PR**(C , D))) + E , F) où il faut être sûr que chaque argument de **PR** ou **COM** est une forme normale. Pour simplifier l'entrée des données, nous avons écrit la fonction **NFORM**. L'expression précédente se calcule par **NFORM**("[A*B*C*D+E,F]"). Explicitons la syntaxe souhaitée. Nous désirons décoder une expression contenant les opérateurs $+$, $-$, $*$, $/$ (pour diviser par une constante) et $^$. De plus on veut pouvoir écrire les commutateurs et anticommutateurs avec la notation habituelle. Formellement, nous posons les définitions suivantes.

expr

[signe] terme { signe terme }

La notation [signe] indique que le premier signe peut être omis. Les accolades ici ne doivent pas être écrites. Elles indiquent que la partie "signe terme" peut être répétée plusieurs fois.

signe

+

| -

terme

fact

```

| terme*fact
| terme/fact

fact
  primaire
  | primaire ^ [signe] primaire

primaire
  nombre
  | nomi
  | (expr)
  | [expr, expr]
  | {expr, expr}

```

Ci-dessus les crochets `[]` et accolades `{ }` représentent le commutateur et l'anticommutateur. Ils doivent être écrits. La forme *nomi* désigne le nom, qui peut être indicé, d'un littéral, index, variable ou fonction.

Le programme de décodage a été basé sur ces définitions. En entrée, `n_c` est la chaîne à décoder. Le décodage s'effectue de gauche à droite, et, au fur et à mesure qu'il progresse, la partie gauche déjà décodée de `n_c` est supprimée. Ainsi après le décodage des caractères de 1 à `n_i - 1` de `n_c`, on coupe `n_c` par :

```
n_c=mid$(n_c,n_i)
```

Notons que la chaîne initiale est conservée dans `n_cm` pour pouvoir localiser les erreurs de syntaxe en terminant le programme par la sortie `n_err`.

Chacune des fonctions `n_expr`, `n_term`, `n_fact` et `n_primaire` s'attend à trouver en début de la chaîne `n_c` une forme qui correspond aux définitions données plus haut. La fonction `decode` est utilisée plusieurs fois pour examiner si `n_c` commence par un caractère donné sans tenir compte des espaces. Ainsi dans `n_expr`, si `n_c="-Lx"`, `decode(n_c,"-")` prend la valeur 2, attestant la présence du signe "-".

n_expr

L'index local `n_j` vaut 0 lors du décodage du premier terme, et 1 pour les termes suivants. On peut ainsi traiter le premier terme, qui est obligatoire mais avec un signe facultatif, et les autres termes qui sont nécessairement précédés d'un signe. L'absence de signe est interprétée comme la fin de l'*expr*.

n_term

On décode le premier *fact*, puis tant qu'il y a un opérateur `*` ou `/`, les *fact* suivants. Au début de chaque nouvelle itération de la boucle `do`, la variable `value` contient la valeur de la partie du *terme* décodée jusque là (cette partie a été supprimée de la chaîne `n_c`). Lorsque **fact* suit, `value` est multiplié à droite par la valeur de *fact*, en utilisant la fonction `PR`. Noter que l'on ne pouvait pas écrire pour cela :

```
value=PR(value,n_fact)
```

mais qu'il est nécessaire d'utiliser une variable auxiliaire avant l'appel de `PR` :

```
w=value
```

```
value=PR(w,n_fact)
```

En effet, il ne faut pas oublier la règle qui interdit d'utiliser **value** comme argument de fonction externe. Cette interdiction n'est pas une vraie interdiction, puisque la forme **PR(value,n_fact)** est admise. Mais l'argument **value** est alors utilisé avec sa valeur locale dans **PR**, et non avec sa valeur au moment de l'appel.

Dans le cas de l'opérateur **/**, on vérifie à l'aide de la fonction **complexp** que le *fact* qui suit est un nombre (qui peut être complexe).

n_fact

La forme **a^k** est admise avec **k** entier*16 négatif seulement si **a** est un nombre complexe. Pour **k > 1**, son calcul est effectué par **k - 1** appels de la fonction **PR**.

n_primaire

Le décodage et calcul des commutateurs et anticommutateurs utilise la procédure **n_paire** appelée avec l'argument **COM** ou **ACOM**. Noter que le résultat du décodage dans **n_paire** est mis dans la variable locale **value** qui correspond au niveau de sous-programme de la fonction **n_primaire**.

Les formes autres que (*expr*) et que les commutateurs et anticommutateurs sont décodées comme expressions du Basic. Pour cela, la fonction **decodex** est utilisée, mais il faut préalablement limiter la chaîne à décoder, de sorte que des opérations **+**, **-**, *****, **/** et **^** ne soient pas effectuées. La fonction **n_instr(x)** renvoie la position du caractère **x** dans la chaîne **n_c**, ou si le caractère est absent, la longueur de **n_c** augmentée de 1. Cette fonction permet de déterminer jusqu'où au plus on peut décoder **n_c** par **decodex**. Le programme de décodage est ainsi très simple, mais les indices des formes indicées ne peuvent pas contenir des opérations. Une forme comme **G(1+1)** n'est donc pas admise.

Exemple

La procédure **PRNFORM** écrit une chaîne et sa valeur après décodage par **NFORM**. On vérifie l'équation (11.2) et on effectue de nouveau le calcul des commutateur et anticommutateur (11.5) à l'aide de **PRNFORM**.

```
rem Adjoindre les programmes Initialisations,
rem PR, PRM, COM et ACOM
Initialisations
PRNFORM "3+Lx*Ly"
PRNFORM "3+i*Lz+Ly*Lx"
PRNFORM "3+i*Lz*Ly^2-i*Ly*Lz*Ly"
PRNFORM "[ Lx+i*Ly, Lx-i*Ly]"
PRNFORM "{ Lx+i*Ly, Lx-i*Ly}"
stop
PRNFORM:procedure(char c$)
  print c$;"=";NFORM(c$)
  return
NFORM:function(char n_c)
```

```

    local char n_cm
    n_cm=n_c
    local index n_i
    value=n_expr
    ift len(n_c)=0 return
n_err:print "*ERREUR* NFORM"
    print n_cm
    print left$(n_cm,len(n_cm)-len(n_c))&"?"
    stop
n_expr:function
    local index n_j
    do
        n_i=decode(n_c,"-")
        if n_i
            n_c=mid$(n_c,n_i)
            vsub value,n_term
        else
            n_i=decode(n_c,"+")
            ift n_j ift n_i=0 return
            ift n_i n_c=mid$(n_c,n_i)
            vadd value,n_term
        endif
        n_j=1
    loop
n_term:function
    local var w
    value=n_fact
    do
        n_i=decode(n_c,"*")
        if n_i
            n_c=mid$(n_c,n_i)
            w=value
            value=PR(w,n_fact)
        else
            n_i=decode(n_c,"/")
            ift n_i=0 return
            n_c=mid$(n_c,n_i)
            w=1/n_fact
            ift not complexp(w) n_err
            vmul value,w
        endif
    loop
n_fact:function
    value=n_primaire

```

```

n_i=decode(n_c,"^")
ift n_i=0 return
n_c=mid$(n_c,n_i)
local var v,w
n_i=decode(n_c,"-")
if n_i
    w=-1
else
    w=1
    n_i=decode(n_c,"+")
endif
ift n_i n_c=mid$(n_c,n_i)
vmul w,n_primaire
ift not integerp(w) n_err
n_i=w
w=value
if complexp(w) or (n_i<2 and polyp(w))
    value=w^n_i
    return
endif
ift n_i<2 n_err
v=w
for n_i=2,n_i
    v=PR(v,w)
next n_i
value=v
return
n_primaire:function
n_i=decode(n_c,"(")
if n_i
    n_c=mid$(n_c,n_i)
    value=n_expr
    n_i=decode(n_c,")")
else
    n_i=decode(n_c,"[")
    if n_i
        n_paire COM
        n_i=decode(n_c,"]")
    else
        n_i=decode(n_c,"{")
        if n_i
            n_paire ACOM
            n_i=decode(n_c,"}")
        else

```

```

        n_i=min(n_instr(+),n_instr(-),n_instr(*),n_instr(
            /),n_instr(^))
        ift n_i n_i=decodex(left$(n_c,n_i-1),1,value)
    endif
endif
endif
ift n_i=0 n_err
n_c=mid$(n_c,n_i)
return
n_instr:function
value=instr(n_c,"@1")
ift value return
value=len(n_c)+1
return
n_paire:procedure
local var v,w
n_c=mid$(n_c,n_i)
v=n_expr
n_i=decode(n_c,"")
ift n_i=0 n_err
n_c=mid$(n_c,n_i)
w=n_expr
value=@1(v,w)
return

```

Sortie (10320 ms)

```

3*Lx*Ly=  Lx*Ly +3
3+i*Lz+Ly*Lx=  Lx*Ly +3
3+i*Lz*Ly^2-i*Ly*Lz*Ly=  Lx*Ly +3
[ Lx+i*Ly, Lx-i*Ly]=  2*Lz
{ Lx+i*Ly, Lx-i*Ly}=  2*Lx^2 +2*Ly^2

```

Autre algèbre

L'adaptation du programme à une autre algèbre s'effectue simplement en modifiant seulement l'initialisation des tableaux **G** et **GG**. Nous donnons ici l'algèbre enveloppante de l'algèbre de Lie $\mathfrak{so}(4)$. Nous utilisons les 6 générateurs L_x, L_y, L_z, A_x, A_y et A_z . Les relations de commutations sont :

$$[L_x, L_y] = iL_z, \quad [L_x, L_z] = -iL_y,$$

$$[L_x, A_x] = 0, \quad [L_x, A_y] = iA_z, \quad [L_x, A_z] = -iA_y, \quad (11.7)$$

$$[A_x, A_y] = iL_z, \quad [A_x, A_z] = -iL_y,$$

ainsi que les relations qui s'en déduisent par permutations circulaires des indices x , y et z . Les générateurs sont copiés dans le tableau **G**, et leurs commutateurs dans le tableau **GG**. Comme exemple de calculs dans cette algèbre, nous montrons que les invariants de Casimir,

$$C_1 = \mathbf{L}^2 + \mathbf{A}^2 = L_x^2 + L_y^2 + L_z^2 + A_x^2 + A_y^2 + A_z^2, \quad (11.8)$$

$$C_2 = \mathbf{L} \cdot \mathbf{A} = L_x A_x + L_y A_y + L_z A_z, \quad (11.9)$$

commutent avec tous les générateurs. Ensuite, le programme calcule la forme normale de l'expression

$$(A_z^3 + A_z A_y^2) A_x. \quad (11.10)$$

```
rem Adjoindre les programmes
rem PR, PRM, COM et ACOM, PRNFORM, NFORM,
rem n_err, n_expr, n_term, n_fact, n_primaire,
rem n_instr et n_paire
Initialisations
,
'calcul de commutateurs avec les invariants de Casimir
L^2+A^2 et L.A
,
C1=Lx^2+Ly^2+Lz^2+Ax^2+Ay^2+Az^2
C2=Lx*Ax+Ly*Ay+Lz*Az
print "C1=";C1
print "C2=";C2
for i1=1,NG
  print "[ C1,";G(i1);" ] =";COM(C1,G(i1));
  print "[ C2,";G(i1);" ] =";COM(C2,G(i1))
next i1
PRNFORM "(Az^3+Az*Ay^2)*Ax"
stop
```

Initialisations:

```
complex i
rem -----
rem Algèbre U so(4)
rem Les NG générateurs G(1), ..., G(NG)
rem -----
NG=6
var G(NG)
G(1)=Lx
G(2)=Ly
G(3)=Lz
```

```

G(4)=Ax
G(5)=Ay
G(6)=Az
rem -----
rem Les commutateurs [ G(i1) , G(i2) ]=GG(i1,i2)
rem pour i1>i2
rem -----
var GG(NG,NG)
for i1=2,NG
  for i2=1,i1-1
    read U
    GG(i1,i2)=i*U
  next i2,i1
data -Lz
data  Ly,-Lx
data  0, Az,-Ay
data -Az, 0, Ax,-Lz
data  Ay,-Ax, 0, Ly,-Lx
return

```

Sortie (66 s)

```

C1=  Lx^2 +Ly^2 +Lz^2 +Ax^2 +Ay^2 +Az^2
C2=  Lx*Ax +Ly*Ay +Lz*Az
[ C1,  Lx ] = 0          [ C2,  Lx ] = 0
[ C1,  Ly ] = 0          [ C2,  Ly ] = 0
[ C1,  Lz ] = 0          [ C2,  Lz ] = 0
[ C1,  Ax ] = 0          [ C2,  Ax ] = 0
[ C1,  Ay ] = 0          [ C2,  Ay ] = 0
[ C1,  Az ] = 0          [ C2,  Az ] = 0
(Az^3+Az*Ay^2)*Ax= -2*i*Lx*Ax*Ay +i*Ly*Ay^2 +3*i*Ly*Az^2 -2*i*Lz*Ay*Az
-2*Lx*Lz +Ax*Ay^2*Az +Ax*Az^3 +3*Ax*Az

```

Exercice 11.1. $[L^2, \rho^2]$

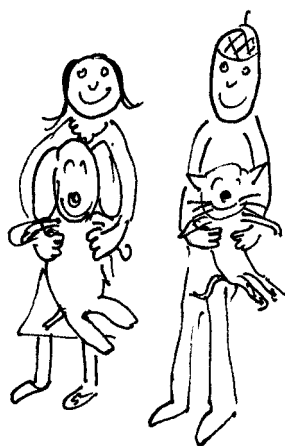
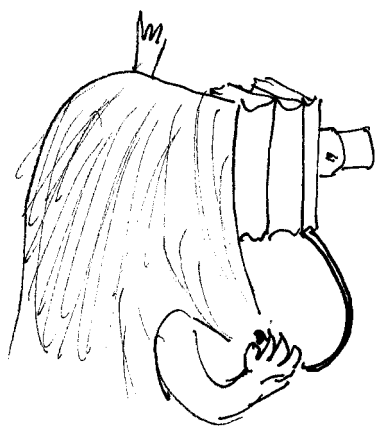
Soit l'algèbre engendrée par les 6 générateurs x, y, z, L_x, L_y et L_z , et définie par les relations de commutations :

$$\begin{aligned}
 [L_x, L_y] &= iL_z, & [L_x, L_z] &= -iL_y, \\
 [L_x, x] &= 0, & [L_x, y] &= iz, & [L_x, z] &= -iy, \\
 [x, y] &= 0, & [x, z] &= 0,
 \end{aligned}
 \tag{11.11}$$

ainsi que les relations qui s'en déduisent par permutations circulaires des lettres x, y et z . Modifier la procédure **Initialisations** précédente pour pouvoir effectuer des calculs dans cette algèbre. On calculera par exemple le commutateur

$$[L_x^2 + L_y^2 + L_z^2, x^2 + y^2]. \tag{11.12}$$

Appendice



Solution des exercices

Exercice 1.1. Nombres rationnels

On obtient pour la somme (1.1) :

$$\frac{70}{125} + \frac{763}{3325} = \frac{15}{19}.$$

```
print 70/125+763/3325
```

Sortie (20 ms)

15/19

Exercice 1.2. Fractions rationnelles

L'expression (1.2) se simplifie en :

$$\frac{a}{x-a} + \frac{x(a-1) + a^2 + a}{x^2 - a^2} = \frac{1}{x+a}.$$

```
print a/(x-a)-(x*(a-1)+a^2+a)/(x^2-a^2)
```

Sortie (130 ms)

[a +x]^-1

Exercice 1.3. Racines

Les racines de l'équation

$$x^2 - 4x + 13 = 0$$

sont les nombres complexes $2 \pm 3i$.

```
complex i
```

```
print formf(x^2-4*x+13)
```

Sortie (290 ms)

```
- [3*i -x +2]* [3*i +x -2]
```

Exercice 1.4. Equations

Le système d'équations :

$$\begin{cases} x + y = 1 \\ 98x - 14y = 2 \end{cases}$$

a pour solution

$$\begin{cases} x = 1/7 \\ y = 6/7. \end{cases}$$

```
var eq1(1),z(1),vz(1)
```

```
z(0)=x
```

```
z(1)=y
```

```

eq1(0)=x+y-1
eq1(1)=98*x-14*y-2
sleq eq1,z,vz,1
print "x=";vz(0);" y=";vz(1)

```

Sortie (305 ms)

```
x= 1/7 y= 6/7
```

Exercice 1.5. Autres équationsLe système d'équations en x , y , z et t :

$$\begin{cases} x + y - z - t = a + 2 \\ x^3 + y^3 + z^3 + t^3 = a^3 + 8 \\ x^2 + y^2 + z^2 + t^2 = a^2 + 6 \\ x + y + z + t = a + 2 \end{cases}$$

a 4 solutions :

$$\begin{cases} x = 2 \\ y = a \\ z = 1 \\ t = -1 \end{cases} \quad \begin{cases} x = 2 \\ y = a \\ z = -1 \\ t = 1 \end{cases} \quad \begin{cases} x = a \\ y = 2 \\ z = 1 \\ t = -1 \end{cases} \quad \begin{cases} x = a \\ y = 2 \\ z = -1 \\ t = 1. \end{cases}$$

```

var eq(3),q(3)
eq(0)=x+y-z-t-a-2
eq(1)=x^3+y^3+z^3+t^3-a^3-8
eq(2)=x^2+y^2+z^2+t^2-a^2-6
eq(3)=x+y+z+t-a-2
q(0)=x
q(1)=y
q(2)=z
q(3)=t
c$=sgeq(3,3,eq,q)

```

Sortie (9640 ms)

```

2 cas pour x
Cas 1 pour x
x= 2
y= a
2 cas pour z
Cas 1 pour z
z= 1
t= -1
Cas 2 pour z
z= -1
t= 1
Cas 2 pour x

```

```

x= a
y= 2
2 cas pour z
Cas 1 pour z
z= 1
t= -1
Cas 2 pour z
z= -1
t= 1

```

Exercice 1.6. Dérivation

La dérivée par rapport à x de l'expression

$$\frac{x}{\cos(x^2 + ax)}$$

est :

$$\frac{\cos(x^2 + ax) + (2x^2 + ax) \sin(x^2 + ax)}{\cos^2(x^2 + ax)}.$$

```

c$="x/cos(x^2+a*x)"
c$=trigox(c$)
print "La dérivée en x de ";trigop(c$)
print "est ";trigop(dertrigo(c$,x))

```

Sortie (755 ms)

```

La dérivée en x de [x]* [cos( x^2 +x*a )]^~1
est [cos( x^2 +x*a )]^~2* [cos( x^2 +x*a ) +2*sin( x^2 +x*a )*x^2 +sin
( x^2 +x*a )*x*a]

```

Exercice 1.7. Intégrale

Le programme montre que l'intégrale

$$\int \frac{1}{(x^2 + 3ax + 2a^2)^3} dx$$

a pour valeur :

$$\frac{12x^3 + 54ax^2 + 76a^2x + 33a^3}{2a^4(x^2 + 3ax + 2a^2)^2} + \frac{6}{a^5} \log(x + a) - \frac{6}{a^5} \log(x + 2a).$$

```

p=(x^2+3*x*a+2*a^2)^~3
intg1 p,x

```

Sortie (3805 ms)

La partie rationnelle de l'intégrale est

$$\frac{1}{2} * [a]^{-4} * [x^2 + 3*x*a + 2*a^2]^{-2} * [12*x^3 + 54*x^2 * a + 76*x*a^2 + 33*a^3]$$

La partie logarithmique de l'intégrale est le produit de

$$6 * [a]^{-4}$$

et de la somme de

```
[a]^(-1) * log( x + a )
et de
- [a]^(-1) * log( x + 2*a )
```

Exercice 1.8. Développement limité

Le développement limité de $\exp(\sin x)$ au voisinage de $x = 0$, à l'ordre 10, est donné par :

$$1 + x + \frac{1}{2}x^2 - \frac{1}{8}x^4 - \frac{1}{15}x^5 - \frac{1}{240}x^6 + \frac{1}{90}x^7 + \frac{31}{5760}x^8 + \frac{1}{5670}x^9 - \frac{2951}{3628800}x^{10} + \dots$$

```
k=10
```

```
y=sexp(ssin(x,k),k,x)
```

```
print "exp(sin(x))=";str$(y,/x);" + ..."
```

Sortie (1735 ms)

```
exp(sin(x))= ( 1)+( 1)*x+( 1/2)*x^2+( -1/8)*x^4+( -1/15)*x^5+( -1/2
40)*x^6+( 1/90)*x^7+( 31/5760)*x^8+( 1/5670)*x^9+( -2951/3628800)*
x^10 + ...
```

Exercice 2.1. Arrondi de l'unité

La valeur de l'arrondi de l'unité dépend de $q = \text{precision2}$ par la relation

$$\delta = 2^{-k} = 2^{-q - \lfloor \sqrt{q} \rfloor - 1}.$$

La fonction `intsqr(q)` calcule exactement $\lfloor \sqrt{q} \rfloor$.

```
print "precision      k      delta"
forv p in (10,20,100,1000,1230)
  precision p
  format -3
  k=precision2+intsqr(precision2)
  while 1~+2^-k>1~
    k=k+1
  wend
  while 1~+2^-k=1~
    k=k-1
  wend
  print just$(p,5);just$(k,9);"      ";2^-k
nextv
```

Sortie (13615 ms)

precision	k	delta
10	50	0.89~ E-15
20	85	0.26~ E-25
100	361	0.21~ E-108
1000	3389	0.64~ E-1020
1230	4159	0.10~ E-1251

Exercice 2.2. Somme des chiffres

La fonction `schiffre(x)` calcule la somme des chiffres de l'entier x . On ajoute à `value` (qui à l'entrée de la fonction vaut 0) le dernier chiffre de x , donné par `modr(x,10)`, puis on remplace x par $x \setminus 10 = \lfloor x/10 \rfloor$. On itère ensuite, tant que $x \neq 0$. Le nombre $x = (10^{22} - 1)/23$ est écrit en deux blocs de 11 chiffres (le premier étant précédé de 0), pour mettre en évidence la propriété que les chiffres d'un bloc sont les complémentaires à 9 des chiffres de l'autre ($0 + 9 = 9$, $4 + 5 = 9$, $3 + 6 = 9$, ...). La somme des chiffres de x est donc $9 \times 11 = 99$, ce que confirme `schiffre`.

```
x=(10^22-1)/23
print using "0#####" #####";x
print schiffre(x)
stop

schiffre:function(x)
  ift not integerp(x) err_schiffre
  do
    ift x=0 return
    value=value+modr(x,10)
    x=x\10
  loop
Sortie (260 ms)
04347826086 95652173913
99
```

Autre solution

La fonction `schiffre` s'écrit plus simplement par programmation récursive.

```
schiffre:function(x)
  ift x<>0 value=schiffre(x\10)+modr(x,10)
  return
```

Exercice 2.3. 1989

À l'aide de 8 boucles `forc` imbriquées, on forme les 4^8 expressions contenant les chiffres de 1 à 9 dans l'ordre croissant et les signes $+$, $*$ et $-$. Ces expressions sont écrites dans la chaîne de caractères `f9$` et calculées par une commande `xqt`. Par exemple, lors du premier passage, la commande `xqt` exécute l'instruction : `u=1+2+3+4+5+6+7+8+9`. Le programme montre qu'il y a une et une seule solution.

```
e$=cset$("+", "-", "*", "-")
forc c1 in e$
  f2$="1"&c1&"2"
forc c2 in e$
  f3$=f2$&c2&"3"
forc c3 in e$
  f4$=f3$&c3&"4"
forc c4 in e$
  f5$=f4$&c4&"5"
```

```

      forc c5 in e$
        f6$=f5$&c5&"6"
      forc c6 in e$
        f7$=f6$&c6&"7"
      forc c7 in e$
        f8$=f7$&c7&"8"
      forc c8 in e$
        f9$=f8$&c8&"9"
        xqt "u="&f9$
        ift u=1989 print f9$
      nextc
    nextc
  nextc
nextc
nextc
nextc
nextc
nextc

```

Sortie (1111 s)

1+2+34*56-7+89

Exercice 2.4. Total=100

Solution 1

Cette première solution est analogue à celle de l'exercice 1989. A l'aide de 9 boucles **forc** imbriquées, on forme les 2×3^8 expressions contenant les chiffres de 1 à 9 dans l'ordre croissant et les signes + et -. Ces expressions sont écrites dans la chaîne de caractère **c** et testées par une commande **xqt**. Par exemple, lors du premier passage, la commande **xqt** exécute l'instruction :

```
ift 100=-1-2-3-4-5-6-7-8-9 print c
```

Remarquer que nous avons utilisé partout la même variable de boucle **c**. Cela est parfaitement correct. En effet une boucle **forc** ou **forv** calcule d'abord l'ensemble des valeurs sur lesquelles la boucle doit être effectuée, puis à chaque nouveau passage la variable de boucle reçoit la valeur suivante. A la différence des boucles **for**, la modification de la variable de boucle à l'intérieur de la boucle n'influe ni sur le nombre de passages, ni sur la valeur de la variable de boucle au début de chaque passage.

```

forc c in ("-1","1")
  forc c in(c&"-2",c&"2",c&"2")
    forc c in(c&"-3",c&"3",c&"3")
      forc c in(c&"-4",c&"4",c&"4")
        forc c in(c&"-5",c&"5",c&"5")
          forc c in(c&"-6",c&"6",c&"6")
            forc c in(c&"-7",c&"7",c&"7")
              forc c in(c&"-8",c&"8",c&"8")

```

```

        forc c in(c&"-9",c&"9",c&"9")
        xqt "ift 100="&c&" print c"
        nextc
        nextc
        nextc
        nextc
        nextc
        nextc
        nextc
        nextc
        nextc

```

Sortie (209 s)

```

-1+2-3+4+5+6+78+9
12-3-4+5-6+7+89
123-4-5-6-7+8-9
123-45-67+89
123+4-5+67-89
123+45-67+8-9
12+3-4+5+67+8+9
12+3+4+5-6-7+89
1+23-4+56+7+8+9
1+23-4+5+6+78-9
1+2+3-4+5+6+78+9
1+2+34-5+67-8+9

```

Solution 2

La procédure `somme` est utilisée de façon récursive pour former toutes les expressions possibles formées des chiffres de 1 à 9 dans l'ordre, avec des signes + et -. Pendant une suite de 8 appels emboîtés, le deuxième argument n croît de 2 à 9, et le chiffre n est rajouté au premier argument, précédé soit du signe -, soit de rien du tout ou soit du signe +. Ensuite, à la profondeur de 9 appels emboîtés, n vaut 10 et la chaîne de l'expression, qui est entièrement construite, est testée dans la commande `xqt`. Par exemple, pour la première fois, la commande `xqt` exécute l'instruction :

```
ift 100=-1-2-3-4-5-6-7-8-9 print c$
```

La sortie, identique à celle de la solution 1, n'est pas répétée.

```

somme "-1",2
somme "1",2
stop

```

```

somme:procedure(char c$,index n)
if n<10
    somme c$&justl$(-n),n+1
    somme c$&justl$(n),n+1
    somme c$&"+"&justl$(n),n+1

```



```

else
    xqt "ift 100="&c$&" print c$"
endif
return

```

Sortie (444 s)

Exercice 2.5. `cube+cube=carré`

Le programme effectue une boucle sur les nombres i de 2 chiffres, pour une base quelconque (de 2 à 36). Le nombre j est obtenu en inversant les deux chiffres de i . La fonction `root`, qui renvoie 0 si $i^3 + j^3$ n'est pas le carré d'un entier, permet de vérifier si le nombre i est une solution. Noter que les valeurs 2 et 3 sont précédées de § pour forcer un décodage décimal, ce qui permet au programme de tourner même si la base est 2 ou 3.

```

print "Dans quelle base voulez-vous résoudre le problèm
e ?"
input k
base k
for i=11,100-1
    j=modr(i,10)*10+divr(i,10)
    ift i<=j ift root(i^§3+j^§3,§2) print i;using "  (#_
3_ + #_ ^3 = #_ ^2)";i;j;root(i^§3+j^§3,§2)
next
base ten
stop

```

Exemple de sortie (2375 ms)

Dans quelle base voulez-vous résoudre le problème ?

INPUT >

10

56 (56^3 + 65^3 = 671^2)

Exercice 2.6. Premiers jumeaux

Le programme suivant effectue une boucle sur les nombres premiers p_1 à l'aide de la fonction `prime` qui détermine le nombre premier suivant un entier donné. Pour déterminer ensuite si $(p_1, p_1 + 2)$ sont premiers jumeaux, on teste la primalité de $p_1 + 2$ par `prtst`. Si oui, la somme de leurs inverses et ajoutée à B et on affiche le numéro d'ordre de la paire n , les nombres p_1 et $p_1 + 2$, la valeur de B et le temps en secondes.

La constante B est très difficile à calculer par suite de la croissance très lente de la série. En une dizaine d'heures, le programme obtient $B \approx 1.72$, ce qui est encore loin de la valeur exacte qui est connue à 10^{-5} près ($B \approx 1.90195$).

```

B=0
n=1
p1=2
do
    p1=prime(p1+1)

```

```

    if prtst(p1+2)
      vadd B,1~/p1+1~/ (p1+2)
      print using "### (#####_,#####) ##.#####_   # s"
        ,n,p1,p1+2,B,timer
      n=n+1
    endif
  loop

```

Sortie

```

1 (   3,   5) 0.53333333~   0 s
2 (   5,   7) 0.87619048~   0 s
3 (  11,  13) 1.04402264~   0 s
4 (  17,  19) 1.15547775~   0 s
5 (  29,  31) 1.22221858~   0 s
...
15399 (2081249,2081251) 1.72064947~ 36498 s
15400 (2081351,2081353) 1.72065043~ 36501 s
15401 (2081921,2081923) 1.72065139~ 36512 s
15402 (2082131,2082133) 1.72065235~ 36516 s

```

Exercice 2.7. 2/97 égyptien

La v_fonction **egypt**(y, m) affiche toutes les représentations du nombre rationnel y par une somme de m fractions égyptiennes :

$$y = \frac{1}{a_1} + \frac{1}{a_2} + \cdots + \frac{1}{a_m},$$

où les entiers a_i vérifient les inégalités strictes : $1 < a_1 < a_2 < \cdots < a_m$. Elle renvoie le nombre de décompositions trouvées, et 0 s'il n'y en a pas. La procédure **part**(x, b, n), appelée de façon récursive, permet d'effectuer la boucle sur tous les entiers naturels a_1, a_2, \dots, a_{m-1} . vérifiant $1 < a_1 < a_2 < \cdots < a_{m-1}$ et

$$y > \frac{1}{a_1} + \frac{1}{a_2} + \cdots + \frac{1}{a_{m-1}}.$$

Lors de l'appel avec la valeur $n \neq m$ du troisième argument les nombres a_1, a_2, \dots, a_{n-1} ont été choisis, et il s'agit de définir la boucle sur a_n . Le premier argument x , qui vaut

$$x = y - \frac{1}{a_1} - \frac{1}{a_2} - \cdots - \frac{1}{a_{n-1}},$$

donne une limite supérieure pour a_n puisque x doit être égal à la somme de $m - n + 1$ fractions plus petites ou égales à $1/a_n$: $a_n < (m - n + 1)/x$. Le deuxième argument b , qui vaut a_{n-1} , donne une limite inférieure pour a_n . Une autre limite inférieure pour a_n est donnée par la condition $x > 1/a_n$.

L'appel le plus profond de la procédure **part**, avec $n = m$, détermine si x donne la dernière fraction égyptienne $1/a_m$. Si c'est le cas, la solution est affichée et la variable **value**, qui correspond à la valeur de la fonction **egypt**,

est incrémentée. La commande `print egypt(2/97,3)` a pour effet d'écrire les solutions de l'exercice proposé (par les `print` dans la procédure `part`), puis le nombre de solutions trouvées (la valeur renvoyée par la fonction `egypt`).

```

print egypt(2/97,3); " solutions"
stop
egypt:function(y,index m)
  local index i char c$ var a(m)
  part y,1,1
  ift value print
  return
part:procedure(x,b, index n)
  ift x=0 return
  b=max(gint(1/x)-1,b)
  if n=m
    ift not integerp(1/x) return
    ift 1/x<=b return
    ift value=0 print y;
    c$=x
    ift m>1 c$=conc$(i=1,m-1 of 1/a(i)&"")&c$
    print "=";change$(c$," ","");
    value=value+1
    return
  else
    a(n)=int((m-n+1)/x)
    ift a(n)=(m-n+1)/x a(n)=a(n)-1
    do
      ift a(n)<=b return
      part x-1/a(n),a(n),n+1
      a(n)=a(n)-1
    loop
  endif
endif

```

Sortie (767 s)

```

2/97=1/97+1/98+1/9506=1/91+1/104+1/70616=1/73+1/146+1/14162=1/70+1/1
58+1/268205=1/68+1/170+1/32980=1/66+1/194+1/3201=1/66+1/183+1/390522=
1/65+1/194+1/12610=1/60+1/291+1/1940=1/60+1/255+1/32980=1/56+1/679+1/
776=1/56+1/392+1/4753=1/56+1/388+1/5432=1/56+1/364+1/70616=1/54+1/582
+1/2619=1/54+1/486+1/23571=1/54+1/485+1/26190=1/54+1/477+1/277614=1/5
3+1/582+1/30846=1/52+1/776+1/10088=1/52+1/728+1/70616=1/51+1/1164+1/6
596=1/51+1/1020+1/32980=1/51+1/990+1/1632510=1/50+1/2450+1/4753=1/50+
1/2425+1/4850=1/50+1/1940+1/9700=1/50+1/1825+1/14162=1/50+1/1746+1/21
825=1/50+1/1700+1/32980=1/50+1/1650+1/80025=1/50+1/1649+1/82450=1/50+
1/1625+1/315250=1/50+1/1620+1/785700=1/50+1/1618+1/1961825=1/50+1/161
7+1/7842450=1/49+1/7154+1/14162=1/49+1/5432+1/38024=1/49+1/5096+1/706
16=1/49+1/4850+1/237650=1/49+1/4802+1/465794=1/49+1/4760+1/3232040=1/

```

49+1/4754+1/22595762

43 solutions

La fraction $2/97$ possède également une et une seule décomposition en deux fractions égyptiennes, comme le montre l'appel ci-dessous de la fonction `egypt` précédente. Elle fait cependant intervenir une fraction plus compliquée que la décomposition $2/97 = 1/56 + 1/679 + 1/776$.

`ift egypt(2/97,2)`

Sortie (2165 ms)

$2/97=1/49+1/4753$

Exercice 2.8. Stein

La procédure `egd(x)` affiche la décomposition du nombre rationnel x . Dans la boucle `for`, x est le reste non encore décomposé et z est le dénominateur de la dernière fraction égyptienne. Le dénominateur de la fraction égyptienne suivante est le plus petit nombre impair plus grand que $1/x$ et différent de z . La variable `c$` est utilisée pour afficher le signe $+$ entre deux fractions ou un espace devant la première fraction. La boucle se termine lorsque le reste x est nul, ou sur un dépassement mémoire (essayer $5/139$).

```

    egd 6/11
    egd 4/13
    stop
egd:procedure(x)
    print x;" =";
    c$=" "
    z=1
    for m=1,2^31-1
        y=gint(1/x)
        ift even(y) y=y+1
        ift y=z y=y+2
        z=y
        print c$;justl$(1/y);
        c$=" + "
        x=x-1/y
        ift x=0 exit
    next m
    print " [somme de";m;" fractions]"
    return

```

Sortie (1095 ms)

6/11 = 1/3 + 1/5 + 1/83 + 1/13695 [somme de 4 fractions]
 4/13 = 1/5 + 1/11 + 1/61 + 1/2567 + 1/4664989 + 1/16848090073171 + 1/157135755580772061841245625 + 1/131688776970347403615775095308511151
 31046845668962709 + 1/26012900969918851043179315333197435458121451559
 4663379683885827184094388323372007037885351529128379965313 + 1/135334
 203374161020268963206516585593880933159712363358518522008780313914088

```
768181196414253145643002723363753390122566182017600763813267384940215
661898534001220925164529500240526357556899650403533766405786410625  [
somme de 10 fractions]
```

Exercice 2.9. $16/64 = 1/4$

Le programme suivant cherche les chiffres a , b et c de 1 à 9 tels que :

$$\frac{ac}{cb} = \frac{a\cancel{c}}{\cancel{c}b} = \frac{a}{b}.$$

Toutes les possibilités sont examinées au moyen de trois boucles **for**. À l'impression des solutions, les espaces devant les nombres sont supprimés au moyen de la fonction **justl\$**.

```
for a=1,9
  for b=1,9
    for c=1,9
      if (10*a+c)/(10*c+b)=a/b
        print a;justl$(c);"/";justl$(c);justl$(b);"=";a
          ;"/";justl$(b)
      endif
    next c,b,a
```

Sortie (9585 ms)

```
11/11= 1/1
16/64= 1/4
19/95= 1/5
22/22= 2/2
26/65= 2/5
33/33= 3/3
44/44= 4/4
49/98= 4/8
55/55= 5/5
66/66= 6/6
77/77= 7/7
88/88= 8/8
99/99= 9/9
```

Exercice 2.10. $6729/13458 = 1/2$

Le programme recherche les fractions de la forme

$$\frac{p_1p_2p_3p_4}{p_5p_6p_7p_8p_9} = \frac{1}{2},$$

où p_1, p_2, \dots, p_9 est une permutation de 1, 2, ..., 9, et où les notations $p_1p_2p_3p_4$ et $p_5p_6p_7p_8p_9$ représentent l'écriture en base 10 de deux nombres de 4 et 5 chiffres. La solution proposée est très brutale (boucle sur tous les cas), et le programme prend 3 heures de calcul. Par contre, l'écriture du programme et sa mise au point ont été très rapides. La boucle **while ...wend** est parcourue $9! = 362880$

fois : le tableau **P**, qui contient les chiffres p_i , est initialisé par le premier appel de **nextperm** avec la permutation $p_i = i$ ($i = 1, \dots, 9$); à chaque passage, une nouvelle permutation est placée dans le tableau **P** par l'appel de **nextperm**; lorsque toutes les permutations ont été générées, **nextperm** renvoie la valeur 0, ce qui termine la boucle. Lorsque le chiffre p_9 est impair, la fraction ne peut valoir $1/2$ et on se contente de passer à la permutation suivante. Pour toute autre permutation, la fraction est écrite dans la chaîne **c\$**. Au premier passage, par exemple, la chaîne calculée par **conc\$(i=1,4 of P(i))** vaut " 1 2 3 4", avec des espaces doubles devant chaque chiffre; ces espaces sont supprimés par **change\$**, ce qui fait que la variable **c\$** prend pour valeur la chaîne "1234/56798"; la commande **xqt** effectue alors l'instruction :

```
ift 1234/56798=1/2 print c$
```

comme on peut le vérifier facilement à l'aide du débogueur. Remarquer qu'on aurait pu aussi utiliser, au lieu de cette commande **xqt** :

```
xqt "if 1/2="&c$æ&"print c$æ&"endif"
```

où le symbole æ ([a] Z) représente le code ASCII 0. Dans ce cas, au premier passage, la commande **xqt** exécute le code Basic :

```
if 1/2=1234/56798
  print c$
endif
```

Par contre, il n'est pas recommandé de remplacer la commande **xqt** par les lignes suivantes :

```
xqt "if 1/2="&c$
print c$
endif
```

En effet, supposons que le programme a été modifié de cette façon. Le **if** est caché dans la chaîne du **xqt**, tandis que le **endif** est visible. Il en résulte que lorsque la condition **if even(P(9))** est fausse, c'est après cet **endif** que l'exécution se poursuit, et non pas après la véritable sortie de la condition (la commande **endif** appareillée à **if even(P(9))**). Dans le cas présent, comme les deux **endif** se suivent, le programme reste correct (des commandes **endif** en trop ne gênent pas le Basic).

```
index P(9)
d$="=1/2 print c$"
k=nextperm(9,P(1),0)
while k
  if even(P(9))
    c$=change$(conc$(i=1,4 of P(i)) & "/" & conc$(i=5,9
      of P(i))," ","")
    xqt "ift "&c$&d$
  endif
  k=nextperm(9,P(1))
wend
```

Sortie (11242 s)

6729/13458
 6792/13584
 6927/13854
 7269/14538
 7293/14586
 7329/14658
 7692/15384
 7923/15846
 7932/15864
 9267/18534
 9273/18546
 9327/18654

Exercice 4.1. Nombre de fractions

Il y a $\varphi(q)$ fractions réduites p/q ayant le dénominateur $q > 1$ et telles que $0 < p/q < 1$. Le nombre de fractions réduites $p/q \in (0, 1)$ ayant leur dénominateur $q \leq 100$ est donc

$$\varphi(2) + \varphi(3) + \cdots \varphi(100) = 3043,$$

comme le calcule le programme suivant :

```
print sum(i=2,100 of euler_phi(i))
```

Sortie (4300 ms)

3043

Exercice 4.2. Suite aliquote

En partant d'un entier x , la procédure **amiable** effectue le remplissage du tableau **S** avec la suite $\mathbf{S}(1) = x$, $\mathbf{S}(2) = A(x)$, \dots , $\mathbf{S}(N) = A(\mathbf{S}(N-1))$, la somme des diviseurs de n étant calculée par la fonction **sigma**(n , 1). On interrompt le calcul dans les cas suivants : si $y = \mathbf{S}(N)$ prend une valeur inférieure à x (si on étudie les entiers $x = 2, 3, \dots$, alors le nombre y a déjà été considéré); si N devient plus grand que la taille NM du tableau **S**; si $y = \mathbf{S}(N)$ est un entier plus grand que $\mathbf{VM} = 10^{12}$ (le calcul de $A(y)$ est très long si y est difficile à factoriser); si $\mathbf{S}(N)$ est égal à un des nombres précédents $\mathbf{S}(N_1)$ de la suite. Le dernier cas nous donne un cycle amiable de longueur $N - N_1$, et dont le plus petit nombre est $\mathbf{S}(N_0)$. Les autres cas sont attestés par $N_1 = 0, -1$ ou -2 .

Le tableau **P** donne l'ordre croissant des éléments du tableau **S**. Pour déterminer si le nouveau nombre $\mathbf{S}(N)$ est déjà un nombre de la suite, on utilise la fonction **search** qui donne son rang k dans le tableau. Le rang $k = 0$ indique que $\mathbf{S}(N)$ est le plus petit nombre de la suite (on arrête donc). Si le nombre de rang k , $\mathbf{S}(\mathbf{P}(k))$, est égal à $\mathbf{S}(N)$ on a trouvé un cycle, avec $N_1 = \mathbf{P}(k)$, et on obtient N_0 comme étant le premier nombre de la suite $\mathbf{P}(1), \mathbf{P}(2), \dots$ plus grand ou égal à N_1 . Si, par contre, $\mathbf{S}(n)$ est un nouveau nombre, le tableau **S** est de nouveau trié pour déterminer le nouvel ordre (donné par **P**). Ce tri est effectué par **sort** en indiquant que seulement le nombre $\mathbf{S}(N)$ n'est pas encore rangé.

La procédure `pamiable1`, appelée après `amiable`, affiche le résultat uniquement si un cycle amiable a été trouvé. La procédure `pamiable(x)`, qui effectue l'appel de la procédure `amiable` et l'affichage de la suite calculée, permet d'étudier un nombre particulier. Dans l'exemple nous l'appelons pour $n = 12496$ qui donne un cycle amiable de longueur 5. Ensuite on effectue une recherche systématique à partir de $x = 2$, en appelant seulement `amiable` (un appel de `pamiable` est nécessaire pour déclarer les tableaux). Le programme trouve la plus petite paire de nombres amiables, 220 et 284 en quelques minutes. La suite $A^i(276)$ est le premier exemple d'un des nombreux cas où on obtient des nombres $> 10^{12}$. Le comportement de cette suite est, à notre connaissance, un problème non résolu (on sait que $A^{469}(276)$ est un nombre de 45 chiffres). La suite aliquote 1064, 1336, 1184, 1210, 1184, ... fait apparaître les nombres amiables 1184 et 1210 avant l'étude de la suite $A^i(1184)$.

```
'adjoindre la fonction sigma
pamiable(12496)
x=1
do
    x=x+1
    amiable
    pamiable1
loop
pamiable:procedure(x)
    NM=5000
    VM=10^12
    var S(NM)
    index P(NM)
    amiable
    for i=1,N
        ift i=N1 print "  [";
        print S(i);
        ift i=N0 print "*";
    next i
    if N1>0
        print " ]"
    else N1=0
        print " déjà vu"
    else
        print
    endif
endif
pamiable1:ift N1=0 return
if N1>0
    ift N0<>1 print "[";S(1);" --> ]";
    print S(N0);
    ift N-N1=1 print " parfait";
```



```

    ift N-N1=2 print "  et";S(N0+1);" amiables";
    ift N-N1>2 print " --> cycle amiable de longueur";N-N
      1;
  else N1=-1
    print S(1);" --> trop grand";
  else N1=-2
    print S(1);" --> suite trop longue";
  endif
  print "  (temps";timer;" s)"
  return
amiable:procedure
  S(1)=x
  P(1)=1
  N0=0
  N1=0
  for N=2,NM
    S(N)=sigma(S(N-1),1)-S(N-1)
    if S(N)>VM
      N1=-1
      return
    endif
    k=search(S(1),N,1,P(1))
    ift k=0 return
    if S(P(k))=S(N)
      N1=P(k)
      for k0=1,k
        ift P(k0)>=N1 exit
      next k0
      N0=P(k0)
      return
    endif
    sort S(1),N,1,P(1),N-1
  next N
  N1=-2
  return
[ 12496* 14288 15472 14536 14264 12496 ]
12496 --> cycle amiable de longueur 5 (temps 1 s)
6 parfait (temps 2 s)
28 parfait (temps 4 s)
220 et 284 amiables (temps 277 s)
276 --> trop grand (temps 460 s)
...
[ 1064 --> ] 1184 et 1210 amiables (temps 2010 s)
...
```

```

1184 et 1210 amiables (temps 2521 s)
[ 1188 --> ] 2620 et 2924 amiables (temps 2522 s)
...
[ 2856 --> ] 14316 --> cycle amiable de longueur 28 (temps 10635
s)

```

Exercice 4.3. Réduire 48828/89077

On désire décomposer une fraction m/ab , dont le dénominateur est le produit de deux nombres a et b premiers entre-eux, en une somme $x/a + y/b$. On doit avoir $bx + ay = m$; donc x vérifie la congruence $bx \equiv m \pmod{a}$, qui peut se résoudre à l'aide de `prinv`; on aura ensuite $y = (m - bx)/a$. Pour la fraction demandée, le programme trouve la solution :

$$\frac{48828}{89077} = \frac{123}{317} + \frac{45}{281}.$$

```

m=48828
a=317
b=281
x=modr(m*prinv(b,a),a)
y=(m-b*x)/a
print using "#=#+";x/a+y/b;x/a;y/b

```

Sortie (70 ms)

```
48828/89077=123/317+45/281
```

Exercice 4.4. Racine m ième

Nous réduisons le degré de la congruence (4.26) suivant une méthode classique. Le degré de la congruence

$$x^m \equiv a \pmod{p}, \quad (*)$$

où p est un nombre premier, peut être abaissé au degré $d = (m, p-1)$ de la façon suivante. Soit s une solution de $ms \equiv d \pmod{p-1}$. En prenant la puissance s ième de $(*)$, on obtient, d'après le théorème de Fermat (4.3), $x^{ms} \equiv x^d \equiv a^s \pmod{p}$. Pour l'équation (4.26), $d = 3$ est calculé par `gcdr` et une solution s est obtenue par `prinv`. Comme $s > 2^{15}$, il est hors de question de calculer a^s ; on calcule seulement son résidu modulo p par `mdpwre`. La factorisation de $x^d - a^s$ donne les trois solutions de l'équation (4.26) :

$$\begin{aligned}
x &\equiv -11756755 \\
x &\equiv -8933865 \pmod{p} \\
x &\equiv -8933865.
\end{aligned}$$

```

p=31415971
m=27182817
a=2
d=gcdr(p-1,m)
s=prinv(m/d,(p-1)/d)

```

```
print "d=";d;" s=";s
w=x^d-mdpwre(2,s,p)
print "x^d-2^s=";w;"=";mdff(w,p)
```

Sortie (3100 ms)

```
d= 3 s= 5599069
```

```
x^d-2^s= x^3 -3180876= [x +11756755]* [x +10725351]* [x +8933865]
```

On peut vérifier les solutions obtenues à l'aide de la fonction `mdpwre`.

```
print mdpwre(-11756755,27182817,31415971);
print mdpwre(-10725351,27182817,31415971);
print mdpwre(-8933865,27182817,31415971)
```

Sortie (205 ms)

```
2 2 2
```

Exercice 5.1. Variation pollardOn peut remplacer l'initialisation de `x` et `xp`,

```
x=5
xp=2
```

par :

```
x=@3
xp=x
x=@2
```

et le calcul de x_m en fonction de x_{m-1} ,

```
x=modr(x^2+1,n)
```

par :

```
x=modr(@2,n)
```

Le deuxième argument doit être un polynôme en x , par exemple :

```
pollard 2^67-1,x^2+36*67,1
```

Sortie (24 s)

```
147573952589676412927= 193707721 * 761838257287
```

qui factorise $2^{67} - 1$ en 235 itérations. Cette modification peut être utile lorsque $f(x) = x^2 + 1$ conduit à un échec. Knuth recommande d'utiliser $f(x) = x^2 + c$, en évitant $c = 0$ et $c = -2$.

Exercice 5.2. Fermat

La fonction `fermat$(n[, k])` effectue une factorisation de n en appliquant la méthode de Fermat au nombre kn (par défaut $k = 1$). Après la commande `do` de la boucle, $1 = 2t + 1$ et $\mathbf{np} = kn + t^2$. La fonction `root(np,2)` vaut 0 tant que \mathbf{np} n'est pas un carré parfait. Les facteurs non premiers sont écrits entre parenthèses. Pour x , nous utilisons $k = 3$. Par comparaison, la fonction `prfact$` factorise les nombres w et x en 131 et 63 secondes respectivement.

```
w=284620201979
print w;" = ";fermat$(w)
x=226077651799
print x;" = ";fermat$(x,3)
stop
```

```

fermat$:function$(n)
  local var np,l,k
  local index i
  k=1
  if @0=2
    k=@2
  endif
  np=n*k
  ift not odd(np) err_fermat$
  l=1
  do
    ift root(np,2) exit
    vadd np,l
    vadd l,2
  loop
  np=root(np,2)-(l-1)/2
  np=np/gcd(np,k)
  push$ ""," * "
  for i=1,2
    if prtst(np)
      cadd value,justl$(np)&pop$
    else
      cadd value," (&justl$(np)&")"&pop$
    endif
    np=n/np
  next i
  return
end

```

Sortie (22 s)

284620201979 = 531457 * 535547

226077651799 = 823547 * 274517

Exercice 5.3. Fraction continue

La procédure `fcont` x [ϵ] décompose x en fraction continue et affiche les quotients incomplets a_n et les convergents p_n/q_n , jusqu'à ce que $|x - p_n/q_n| \leq \epsilon$ (par défaut $\epsilon = 2^{-\text{precision}}$ correspond à la précision en cours). Les calculs sont faits en exact, de sorte qu'en donnant $\epsilon = 0$ on peut déterminer le développement d'un nombre rationnel. Les variables `p`, `q`, `P` et `Q` qui ont pour valeurs respectives p_n , q_n , p_{n-1} et q_{n-1} sont calculées par les relations de récurrences (5.3). La fonction interne `appr` fonctionne de la même manière que `fcont`, mais elle renvoie seulement le dernier convergent. Noter la régularité des quotients incomplets de e et $\sqrt{7}$.

```

print "pi=";pi
fcont pi

```

```

print "e=";exp(1)
fcont exp(1)
print "sqr(7)=";sqr(7)
fcont sqr(7)
stop
fcont:procedure(u)
local var x,y,P,Q,p,q,eps
eps=2~precision2
if @0=2
  eps=@2
endif
u=exact(u)
y=int(u)
x=u-y
P=1
Q=0
p=y
q=1
do
  print justl$(y,8);p/q
  ift abs(u-p/q)<=eps exit
  x=1/x
  y=int(x)
  x=x-y
  push y*p+P,y*q+Q
  P=p
  Q=q
  q=pop
  p=pop
loop
ift x print "... "
return

```

Sortie (5745 ms)

```

pi= 0.3141592654~ E+1
3      3
7      22/7
15     333/106
1      355/113
292    103993/33102
1      104348/33215
1      208341/66317
1      312689/99532
2      833719/265381
1      1146408/364913

```

```

3      4272943/1360120
1      5419351/1725033
...
e= 0.2718281828~ E+1
2      2
1      3
2      8/3
1      11/4
1      19/7
4      87/32
1      106/39
1      193/71
6      1264/465
1      1457/536
1      2721/1001
8      23225/8544
1      25946/9545
1      49171/18089
10     517656/190435
1      566827/208524
1      1084483/398959
12     13580623/4996032
...
sqr(7)= 0.2645751311~ E+1
2      2
1      3
1      5/2
1      8/3
4      37/14
1      45/17
1      82/31
1      127/48
4      590/223
1      717/271
1      1307/494
1      2024/765
4      9403/3554
1      11427/4319
1      20830/7873
1      32257/12192
4      149858/56641
1      182115/68833
1      331973/125474
1      514088/194307

```

4	2388325/902702
1	2902413/1097009
1	5290738/1999711
1	8193151/3096720
...	

Exercice 5.4. $\sqrt{2(2^{67}-1)}$

Le nombre x_0 ayant les quotients incomplets :

$$a_0 = N - 1, \quad a_1 = 1, \quad a_2 = N - 2, \quad a_3 = 1, \quad a_4 = 2(N - 1),$$

$$a_k = a_{k-4} \text{ pour } k > 4,$$

vérifie l'équation :

$$x_0 = a_0 + x, \quad \text{où } x = \frac{1}{a_1 + \frac{1}{a_2 + \frac{1}{a_3 + \frac{1}{a_4 + x}}}}.$$

Le programme suivant détermine l'équation $w = 0$ vérifiée par x , puis par x_0 . L'équation, simplifiée en prenant la partie réduite par **red** du numérateur de la fraction rationnelle w , montre que

$$x_0 = \sqrt{N^2 - 2}.$$

L'exemple considéré correspond à $N = 2^{34}$ et $x_0 = \sqrt{2(2^{67} - 1)}$.

```

a0=N-1
a1=1
a2=N-2
a3=1
a4=2*N-2
w=x-1/(a1+1/(a2+1/(a3+1/(a4+x))))
w=subs(w,x=x0-a0)
w=red(num(w),x0)
print w

```

Sortie (225 ms)

$$-N^2 + x_0^2 + 2$$

Exercice 5.5. Racine multiple

Le polynôme $P = x^3 + ax + b$ a des racines multiples s'il a des racines communes avec $dP/dx = 3x^2 + a$, c'est-à-dire si et seulement si $4a^3 + 27b^2 = 0$. Ce résultat est obtenu par le programme suivant en éliminant x entre les équations $P = 0$ et $dP/dx = 0$ à l'aide de la fonction **elim**.

```

P=x^3+a*x+b
print elim(P,der(P,x),x)

```

Sortie (80 ms)

$$4a^3 + 27b^2$$

Exercice 5.6. Courbe elliptique

La courbe elliptique $y^2 = x^3 - 8x + 8$ coupe l'axe Ox aux points $A = -1 - \sqrt{5}$, $B = -1 + \sqrt{5}$ et $C = 2$. Le programme (monochrome) suivant trace la figure 5.1 du texte. Chaque moitié de la courbe au dessus et au dessous de l'axe Ox est tracée séparément, à l'aide de `fplot` en fonction du paramètre x . Pour que la courbe soit bien tracée au voisinage de l'axe Ox , on introduit les nombres $A_1 = -3$, $B_1 = 1$ et $C_1 = 2.3$ proches de A , B et C respectivement. La courbe est tracée pour les arcs $A \leq x \leq A_1$, $A_1 \leq x \leq B_1$, $B_1 \leq x \leq B$, $C \leq x \leq C_1$ et $C_1 \leq x \leq D$ ($D = 4.5$) avec plusieurs appels de `fplot`. Le nombre de points calculés pour chaque arc est $p + 1 = 26$ (noter que l'on donne à p une valeur très légèrement plus grande que 25 pour éviter que le dernier point ne soit pas pris en compte par suite de l'erreur d'arrondi dans le calcul du pas). L'origine a pour coordonnées écran $x_0 = 270$, $y_0 = 200$ et les unités suivant les axes Ox et Oy valent respectivement $s_x = 80$ et $s_y = 25$ pixels. Les coordonnées (mathématiques) $x_1 = 34/9$, $y_1 = 152/27$ sont celles du point R . La droite PQ , d'équation $y = (5x - 2)/3$, et la droite verticale passant par R sont également tracées par `fplot`. La procédure `axis` trace les axes Oxy . La procédure `textxy(x, y, t)`, qui affiche le texte t au point de coordonnées mathématiques x , y , est utilisée pour écrire les noms des points P , Q , R et $P + Q$.

```

cursh 0
cls
A=-1-sqr(5)
A1=-3
B1=1
B=-1+sqr(5)
C=2
C1=2.3
D=4.5
p=25+10^-9
x0=270
sx=80
y0=200
sy=25
x1=34/9
y1=152/27
fplot B,B1,(B1-B)/p,x0,y0,sx,sy,,fym
fplot B1,A1,(A1-B1)/p,x0,y0,sx,sy,,fym
fplot A1,A,(A-A1)/p,x0,y0,sx,sy,,fym
fplot A,A1,(A1-A)/p,x0,y0,sx,sy,,fyp
fplot A1,B1,(B1-A1)/p,x0,y0,sx,sy,,fyp
fplot B1,B,(B-B1)/p,x0,y0,sx,sy,,fyp

```



```

fplot D,C1,(C1-D)/p,x0,y0,sx,sy,,fym
fplot C1,C,(C-C1)/p,x0,y0,sx,sy,,fym
fplot C,C1,(C1-C)/p,x0,y0,sx,sy,,fyp
fplot C1,D,(D-C1)/p,x0,y0,sx,sy,,fyp
fplot A,D,D-A,x0,y0,sx,sy,,fdr
fplot -y1,y1,2*y1,x0,y0,sx,sy,fvt,
axis x0,y0,0,390,640,10,sx,sy,"x","y"
textxy -2.05,-4.7,P
textxy 0.95,1.2,Q
textxy x1-.1,y1+.3,R
textxy x1+.1,-y1-.1,P+Q
ift inp(2)
stop
textxy:text x0+sx*(@1),y0-sy*(@2),"@3"
return
fvt:function
value=x1
return
fdr:function(x)
value=5*x/3-2/3
return
fym:function(x)
value=-sqr(x^3-8*x+8)
return
fyp:function(x)
value=sqr(x^3-8*x+8)
return

```

Exercice 5.7. Suite aliquote 2

La procédure `seraliq(n)` est une modification des programmes décrits dans l'exercice 3.2. La suite des nombres $A^i(n)$ est affichée avec leurs factorisations, déterminées par `prfactb`. Les résultats sont enregistrés sur disque. Après une coupure de courant, on peut faire repartir le programme à partir des données sauvegardées sur disque en changeant n en $-n$ dans l'appel `seraliq(n)` (ligne 4). Pour la suite partant de 276, dont le comportement est inconnu, en 304 heures de calcul, nous avons obtenu $A^{466}(276) =$

$$= 11327678427397701485014114448280258508115880$$

$$= 2^3 \times 3^4 \times 5 \times 23604019 \times 137868819677 \times 1074345300387662864599$$

```

'adjoindre prfactb
s_var 5000
s_pro 100000
seraliq(276)
'seraliq(-276)

```

```

print
pamiable1
stop
pamiable1:ift N1=0 return
if N1>0
  ift N0<>1 print "[";S(1);" --> ]";
  print S(N0);
  ift N-N1=1 print " parfait";
  ift N-N1=2 print " et";S(N0+1);" amiables";
  ift N-N1>2 print " --> cycle amiable de longueur";N-N
    1;
else N1=-1
  print S(1);" --> trop grand";
else N1=-2
  print S(1);" --> suite trop longue";
endif
print
return
seraliq:procedure(x)
nf$=left$(justl$(abs(x)),8)&".dat"
NM=5000
var S(NM)
index P(NM),N
N0=0
N1=0
total=0
if x>0
  N=1
  S(1)=x
  P(1)=1
  print " 0";x;
  save$ nf$,vset$(S(N))
else
  dat$=load$(nf$)
  N=elementn(dat$)\3+1
  for Ni=1,N-1
    S(Ni)=elementv(dat$,3*Ni-2)
    t0=elementv(dat$,3*Ni-1)
    total=total+t0
    print Ni-1;S(Ni);"=";element$(dat$,3*Ni);
    print " (";t0;" s tot=";(timer+total)\3600;" h) "
  next Ni
  S(N)=elementv(dat$,3*N-2)
  print N-1;S(N);

```

```

    dat$=
    sort S(1),N,1,P(1)
endif
for N=N+1,NM
    t0=timer
    w=prfactb(S(N-1))
    t0=timer-t0
    w$=change$(w,"*","", "+","* ")
    print w$;"    (" ;t0;" s    tot=";(timer+total)\3600;" h)
    "
    S(N)=1
    for i=1,polymn(w)
        u=polym(w,i)
        p=norm(u)
        vmul S(N),(p^(deg(u)+1)-1)/(p-1)
    next i
    S(N)=S(N)-S(N-1)
    h$=vset$(t0)&csset$(w$)&vset$(S(N))
    on break next
    open "a",#1,nf$
    print #1;h$;
    close #1
    on break stop
    print justr$(N-1,3);S(N);
    ift S(N)<2 return
    k=search(S(1),N,1,P(1))
    if k>0 and S(P(k))=S(N)
        N1=P(k)
        for k0=1,k
            ift P(k0)>=N1 exit
        next k0
        N0=P(k0)
        return
    endif
    sort S(1),N,1,P(1),N-1
next N
N1=-2
return

```

Exercice 6.1. Euler

La fonction `euler(N, a)` attend en entrée le tableau de variables `a` contenant les N premières valeurs a_1, \dots, a_N (la valeur a_i étant placée dans la variable `a(i)` et la variable `a(0)` étant inutilisée). Elle renvoie la somme (6.2) calculée par le procédé de sommation d'Euler limité aux N premiers termes de (6.3).

La variable $R(i)$ (pour i de 1 à $N - 1$) contient successivement les différences $-\Delta a_i = a_i - a_{i+1}$, $\Delta^2 a_i = \Delta a_i - \Delta a_{i+1}$, ..., $(-1)^{N-i} \Delta^{N-i} a_i$. La somme (6.3) est obtenue à partir des valeurs successives de $R(1)$. Avec $N = 15$ termes, on obtient $\log 2$ à 0.5×10^{-11} près. Il faudrait sommer environ 10^{11} termes de la série (6.13) pour obtenir la même précision.

```
s=7
N=15
var a(N)
for i=1,N
    a(i)=1~/(i+s)
next i
ift s S=-sum(i=1,s of (-1)^i/float(i))
S=S+(-1)^s*euler(N,a)
print S;using "    err=##.###^~~~~";abs(S-log(2))
stop
euler:function(index N, access a(@1))
    local var R(N-1),F
    local index k
    value=a(1)/2
    for k=1,N-1
        R(k)=a(k)-a(k+1)
    next k
    value=value+R(1)/4
    F=1/8~
    for N=N-2,1
        for k=1,N
            R(k)=R(k)-R(k+1)
        next k
        value=value+R(1)*F
        F=F/2
    next N
    return
```

Sortie (1235 ms)

```
0.6931471806~  err= 0.586~ E-11
```

Exercice 8.1. gamma

Le programme calcule les deux membres des formules (8.5) et (8.6) pour un nombre y aléatoire ($0 \leq y < 1$).

```
complex i
y=rnd
g=gamma(1/2+i*y)
print cxnorm(g);pi/cosh(pi*y)
print gamma(1/4+i*y)*gamma(3/4-i*y);pi*sqr(2)/(cosh(pi*
y)+i*sinh(pi*y))
```

Sortie (3115 ms)

```
0.7999008997~ 0.7999008997~
```

```
0.5845638391~ -i*0.5652978679~ 0.5845638391~ -i*0.5652978679~
```

Exercice 8.2. Super gamma_it

En remplaçant la procédure `gamma_it` par la procédure ci-dessous qui initialise le tableau `gamma_psic` élément par élément, on gagne 340 ms sur le temps d'initialisation. Il est inutile dans ce cas de passer en mode réel ou `factor` (d'ailleurs l'exécution est très légèrement plus rapide (de 15 ms) en mode `factor` qu'en mode `develop`).

```
gamma_it:
```

```
gamma_psic(30)=396793078518930920708162576045270521/732
gamma_psic(29)=-121523314048375557204030499407982024604
1491/201025024200
gamma_psic(28)=2913228046513104891794716413587449/40356
gamma_psic(27)=-354198989901889536240773677094747/38280
0
gamma_psic(26)=29149963634884862421418123812691/2283876
gamma_psic(25)=-61628132164268458257532691681/324360
gamma_psic(24)=19802288209643185928499101/6468
gamma_psic(23)=-5609403368997817686249127547/104700960
gamma_psic(22)=25932657025822267968607/25380
gamma_psic(21)=-2530297234481911294093/118680
gamma_psic(20)=1520097643918070802691/3109932
gamma_psic(19)=-261082718496449122051/21106800
gamma_psic(18)=154210205991661/444
gamma_psic(17)=-26315271553053477373/2418179400
gamma_psic(16)=151628697551/396
gamma_psic(15)=-7709321041217/505920
gamma_psic(14)=1723168255201/2492028
gamma_psic(13)=-3392780147/93960
gamma_psic(12)=657931/300
gamma_psic(11)=-236364091/1506960
gamma_psic(10)=77683/5796
gamma_psic(9)=-174611/125400
gamma_psic(8)=43867/244188
gamma_psic(7)=-3617/122400
gamma_psic(6)=1/156
gamma_psic(5)=-691/360360
gamma_psic(4)=1/1188
gamma_psic(3)=-1/1680
gamma_psic(2)=1/1260
gamma_psic(1)=-1/360
gamma_psic(0)=1/12
```

return

Lorsqu'on doit écrire des programmes qui comportent des données compliquées (mais calculables par le Basic), comme la procédure `gamma_it` ci-dessus, il vaut mieux effectuer une écriture automatique. Dans le cas présent, nous avons utilisé le petit programme suivant. Les données sont dans la table `gamma_psic` après l'appel de `gamma`. Le programme crée la chaîne `c$` qui contient le texte de la procédure. Les fins de lignes sont indiquées par le symbole `æ` (touche [a] Z). La fonction `change$` est utilisée pour supprimer tous les espaces, puis `c$` est sauvegardé sur disque sous forme de fichier de type Z, les lignes étant terminées par un octet nul.

```
ift gamma(1)
c$="gamma_it:"æ
for i=30,0
  c$=c$&"gamma_psic("&i&")"&"&gamma_psic(i)æ
next i
c$=c$&"return"æ
c$=change$(c$," ","")
save$ "g.z",c$
```

Exercice 8.3. Polynômes de Bernoulli

Le programme détermine d'abord les nombres de Bernoulli $B(n) = B_n$ pour $n \leq N$ (la valeur $N = 4$ peut être modifiée), en utilisant la relation (8.21). On aurait pu utiliser de façon plus efficace la procédure `bnum`, qui ne calcule pas les nombres nuls. Les polynômes de Bernoulli $B_n(x)$ sont ensuite calculés, pour $n = 0, 1, \dots, N$, à l'aide de l'équation (8.19). En fin de programme, on vérifie pour $n = 4$, l'équation aux différences (8.17) satisfaite par ces polynômes.

```
N=4
var B(N),PB(N)
B(0)=1
print "B0= 1"
for n=1,N
  P=1
  for k=0,n-1
    vadd B(n),B(k)*P
    vmul P,(n+1-k)/(k+1)
  next k
  vmul B(n),-1/P
  print using "B#=#";n;B(n)
next n
for n=0,N
  P=1
  for k=0,n
    vadd PB(n),P*B(n-k)
    vmul P,(n-k)/(k+1)*x
```

```

next k
print using "B#(x)=#";n;PB(n)
next n
print "B4(x+1)-B4(x)=";subs(PB(4),x=x+1)-PB(4)

```

Sortie (750 ms)

```

B0= 1
B1=-1/2
B2=1/6
B3=0
B4=-1/30
B0(x)=1
B1(x)=x -1/2
B2(x)=x^2 -x +1/6
B3(x)=x^3 -3/2*x^2 +1/2*x
B4(x)=x^4 -2*x^3 +x^2 -1/30
B4(x+1)-B4(x)= 4* [x]^3

```

Exercice 8.4. psi

La fonction `fastpsi(31)` calcule le polynôme de degré 61 utilisé par la fonction `gamma` en 13 secondes, alors que `stirling 31` est beaucoup plus lent. L'exemple ci-dessous calcule et affiche `psi` à l'ordre 5.

```

'adjoindre la procédure bnum
psi=fastpsi(3)
print "psi=";psi
stop

fastpsi:function(N)
local var B2(N)
local index i
bnum N,B2
value=formd(sum(i=1,N of x^(2*i-1)*B2(i)/(2*i*(2*i-1)))
)
return

```

Sortie (325 ms)

```
psi= 1/1260*x^5 -1/360*x^3 +1/12*x
```

Exercice 8.5. Somme finie sur un polynôme

La fonction `sdpolyb(f, x)` renvoie le polynôme $F(x)$, équation (8.25), les polynômes de Bernoulli étant déterminés par la relation (8.19). Les nombres de Bernoulli de rang pair $B_2(n) = B_{2n}$ sont calculés par la procédure `bnum`, et le seul nombre non nul de rang impair est $B_1 = -1/2$. Noter que la fonction `sdpoly` de la bibliothèque MATH renvoie une solution $G(x)$ de l'équation $G(x) - G(x-1) = f(x)$, alors que `sdpolyb` résout $F(x+1) - F(x) = f(x)$. La fonction `dsumb` ci-dessous équivaut à la fonction `dsum` de la bibliothèque MATH, tout en étant plus rapide lorsque f est un polynôme en x . Ainsi, l'exemple prend 2725 ms au lieu de 1980 ms si on remplace `dsumb` par `dsum`.

```

'adjoindre bnum
forv f in (1,x,x^2,2*x-1,x^3)
  print "Somme de 1 à n de";f;tab(27);"=";
  print formf(dsumb(f,x,1,n))
nextv
print "\TSortie (";justl$(mtimer-595);" ms)"
stop
dsumb:function(f,x,a,b)
  local datav den(f) var g
  if ord(g,x)=-1
    f=sdpolyb(num(f),x)
    value=(subs(f,x=b+1)-subs(f,x=a))/g
  else
    f=sdfrac(f,x)
    value=subs(f,x=b)-subs(f,x=a-1)
  endif
  return
sdpolyb:function(f,x)
  local index a,b,n,k
  a=degf(f,x)
  b=ordf(f,x)
  ift b<0 err_sdpolyb
  n=lint((a+1)/2)
  local var B2(n),T,S,P
  bnum n,B2
  for n=b,a
    T=coeff(f,x,n)
    if T
      S=-(n+1)/2*x^n
      P=1
      for k=0,(n+1)\2
        S=S+P*B2(k)*x^(n+1-2*k)
        P=P*(n+1-2*k)*(n-2*k)/((2*k+1)*(2*k+2))
      next k
      value=value+S*T/(n+1)
    endif
  next n
  return

```

Sortie (1980 ms)

Somme de 1 à n de	1	=	[n]
Somme de 1 à n de	x	=	1/2* [n]* [n +1]
Somme de 1 à n de	x^2	=	1/6* [n]* [2*n +1]* [n +1]
Somme de 1 à n de	2*x -1	=	[n]^2
Somme de 1 à n de	x^3	=	1/4* [n]^2* [n +1]^2

Exercice 8.6. enum_bis

La procédure `enum_bis(N, E2)` détermine les nombres d'Euler de rang pair $E2(n) = E_{2n}$ pour $n \leq N$. Le tableau `E2`, utilisé en `access` doit être déclaré par `var` préalablement à l'appel. La procédure `bnum` qui calcule les nombres de Bernoulli doit être adjointe au programme. La procédure `enum` est nettement plus rapide que `enum_bis`.

```

      'adjoindre bnum
      N=10
      var E2(N)
      enum_bis N,E2
      for n=0,10
        print using "E#=#";2*n;E2(n)
      next n
      stop

enum_bis:procedure(N, access E(@1))
  local var B(N+1),P
  local index n,k
  bnum N+1,B
  P=1
  for n=0,N+1
    vdiv B(n),P
    P=P*(n+1)*(n+1/2)
  next n
  E(0)=1
  P=2
  for n=1,N
    E(n)=sum(k=0,n of (2^(2*n+2-2*k)-1)*(2^(1-2*k)-1)*B(k)
      )*B(n+1-k))*P
    P=P*(2*n+1)*(2*n+2)
  next n
  return

```

Sortie (4805 ms)

```

E0=1
E2=-1
E4=5
E6=-61
E8=1385
E10=-50521
E12=2702765
E14=-199360981
E16=19391512145
E18=-2404879675441
E20=370371188237525

```

Exercice 8.7. Polynômes d'Euler

Les polynômes d'Euler $PE(n) = E_n(x)$ sont déterminés pour $n = 0, 1, \dots, N$ (la valeur $N = 5$ peut être modifiée). Le programme utilise la procédure `enum` pour déterminer d'abord les nombres d'Euler $E2(n) = E_{2n}$ pour $n \leq N' = \lfloor N/2 \rfloor$. Ensuite, on met $PE(n) = (x - 1/2)^n/n!$ et $E2(n) = E_{2n}/2^{2n}(2n)!$, ce qui permet de calculer l'équation (8.31) sous la forme :

$$E_n(x) = n! \sum_{k=0}^{\lfloor n/2 \rfloor} E2(k)PE(n-2k),$$

par valeurs de n décroissantes (puisque $E_n(x)$ est placé dans $PE(n)$). L'équation (8.32) est vérifiée pour tout n .

```
'adjoindre enum
N=5
NP=lint(N/2)
var E2(NP),PE(N),P
enum NP,E2
PE(0)=1
for n=1,N
  PE(n)=PE(n-1)*(x-1/2)/n
next n
P=1
for n=1,NP
  P=P*(2*n-1)*8*n
  E2(n)=E2(n)/P
next n
for n=N,0
  PE(n)=sum(k=0,lint(n/2) of E2(k)*PE(n-2*k))*ppwr(n)
  print using "E#(x)=#";n;PE(n)
  ift subs(PE(n),x=x+1)+PE(n)-2*x^n err_Euler
next n
stop
```

Sortie (1030 ms)

```
E5(x)=x^5 -5/2*x^4 +5/2*x^2 -1/2
E4(x)=x^4 -2*x^3 +x
E3(x)=x^3 -3/2*x^2 +1/4
E2(x)=x^2 -x
E1(x)=x -1/2
E0(x)=1
```

Exercice 8.8. Liste André

Le programme suivant affiche les 10 permutations d'André pour $N = 4$. La boucle `while ...wend` effectue une boucle sur toutes les permutations de 1,

2, ..., N . Si la permutation est en zigzag, elle est affichée et le compteur A est incrémenté.

```

N=4
index k,P(N)
A=0
k=nextperm(N,P(1),0)
while k
  for k=2,N-1
    ift P(k) in [P(k-1),P(k+1)] exit
    ift P(k) in [P(k+1),P(k-1)] exit
  next k
  if k=N
    print conc$(k=1,N of P(k))
    A=A+1
  endif
  k=nextperm(N,P(1))
wend
print A;" permutations"

```

Sortie (1425 ms)

```

1 3 2 4
1 4 2 3
2 1 4 3
2 3 1 4
2 4 1 3
3 1 4 2
3 2 4 1
3 4 1 2
4 1 3 2
4 2 3 1
10 permutations

```

Exercice 8.9. André Bernoulli Euler

La procédure `abenum`(N , $B2$, $E2$) détermine les nombres de Bernoulli et d'Euler de rang pair $B2(n) = B_{2n}$ et $E2(n) = E_{2n}$, pour $n \leq N$, en utilisant les nombres A_n ($0 \leq n \leq 2N$) calculés par la procédure `andre`. Les tableaux `AB2` et `AE2`, utilisés en `access`, sont déclarés par `var` préalablement à l'appel de `abenum`. La méthode est comparée aux procédures `bsum` et `esum` pour diverses valeurs de N . Il apparaît que les procédures `bsum` et `esum` sont nettement plus rapides.

```

'adjoindre andre, bnum et enum
N=128
var B2(N),AB2(N),E2(N),AE2(N)
print "  N      abenum      bnum      enum"
forv N in (2,4,8,16,32,64,128)
  print justr$(N,3);

```

```

clear timer
abenum N,AB2,AE2
print justr$(mtimer,10);
clear timer
bnum N,B2
print justr$(mtimer,10);
for n=0,N
    ift AB2(n)<>B2(n) ???
next n
clear timer
enum N,E2
print justr$(mtimer,10)
for n=0,N
    ift AE2(n)<>E2(n) ???
next n
nextv
stop
abenum:procedure(N, access B(@1),E(@1))
    local var A(2*N),P,d2n
    local index n
    andre 2*N,A
    for n=0,N
        E(n)=(-1)^n*A(2*n)
    next n
    B(0)=1
    for n=1,N
        B(n)=-(-1)^n*2*n*A(2*n-1)/2^(2*n)/(2^(2*n)-1)
    next n
    return

```

Sortie

N	abenum	bnum	enum
2	295	115	90
4	895	285	235
8	3040	910	760
16	11380	3180	2735
32	45900	12065	10530
64	202830	50095	43595
128	1167175	227920	199780

Exercice 8.10. stan

Le développement en série de $\tan x$ peut s'écrire en fonction des nombres de Bernoulli :

$$\tan x = x + \frac{x^3}{3} + \frac{2x^5}{15} + \cdots + \frac{(-1)^{n-1} 2^{2n} (2^{2n} - 1) B_{2n}}{(2n)!} x^{2n-1} + \cdots,$$

et celui de $\sec x$ en fonction des nombres d'Euler :

$$\sec x = 1 + \frac{x^2}{2} + \frac{5x^4}{24} + \cdots + \frac{(-1)^n E_{2n}}{(2n)!} x^{2n} + \cdots.$$

Si le troisième argument de **stan** est omis, on utilise pour x le premier littéral du polynôme p . Si une erreur de donnée est détectée dans **sverif**, un arrêt se produit sur la commande illégale **stan_err**. Le calcul du développement limité à l'ordre k de $\tan x$ peut être aussi effectué par :

```
taylor(ssin(x,k)/scos(x,k),k-1)
```

mais **stan** est plus rapide pour $k \geq 10$ (pour $k = 21$ on gagne 8 secondes). Le développement limité de $\sec x$ effectué par :

```
taylor(1/scos(x,k),k)
```

est moins rapide que **ssec** pour $k \geq 14$ (de 2 secondes pour $k = 20$).

```
'adjoindre les procédures bnum et enum
print "tan x=";str$(stan(x,21),/x);" +..."
print "sec x=";str$(ssec(x,20),/x);" +..."
print "\TSortie (";justl$(mtimer-580);" ms)"
stop
stan:function(p,k)
  if @0=3
    local datav @3f var x
  else
    local var x
    x=poly1(p)
  endif
  sverif tan
  local index N,i
  N=(k/ord(p,x)+1)\2
  local var q,B(N)
  bnum N,B
  p=mod(p,x^(k+1))
  q=-mod(p^2,x^(k+1))
  p=2*p
  for i=1,N
    ift p=0 exit
    value=value+B(i)*p*(2^(2*i)-1)
    p=mod(p*q,x^(k+1))/(i+1/2)/(i+1)
  next
  return
ssec:function(p,k)
  if @0=3
    local datav @3f var x
  else
    local var x
```

```

    x=poly1(p)
endif
sverif sec
local index N,i
N=(k/ord(p,x)+1)\2
local var q,E(N)
enum N,E
q=-mod(p^2,x^(k+1))
p=1
for i=0,N
    ift p=0 exit
    value=value+E(i)*p
    p=mod(p*q,x^(k+1))/((2*i+1)*(2*i+2))
next i
return
sverif:ift not polyp(p) s@1_err
ift not litp(x) s@1_err
ift ord(p,x)<1 s@1_err
ift not integerp(k) s@1_err
ift k<0 s@1_err
return

```

Sortie (5035 ms)

```

tan x= ( 1)*x+( 1/3)*x^3+( 2/15)*x^5+( 17/315)*x^7+( 62/2835)*x^9
+( 1382/155925)*x^11+( 21844/6081075)*x^13+( 929569/638512875)*x^15
+( 6404582/10854718875)*x^17+( 443861162/1856156927625)*x^19+( 18
888466084/194896477400625)*x^21 +...
sec x= ( 1)+( 1/2)*x^2+( 5/24)*x^4+( 61/720)*x^6+( 277/8064)*x^8+
( 50521/3628800)*x^10+( 540553/95800320)*x^12+( 199360981/87178291
200)*x^14+( 3878302429/4184557977600)*x^16+( 2404879675441/64023737
05728000)*x^18+( 14814847529501/97316080327065600)*x^20 +...

```

Exercice 11.1. $[L^2, \rho^2]$

Le programme montre que $[L^2, x^2 + y^2 + z^2] = 0$ et que

$$[L^2, x^2 + y^2] = -4ixzL_y + 4iyzL_x + 2x^2 + 2y^2 - 4z^2.$$

```

rem Adjoindre les programmes
rem PR, PRM, COM et ACOM, PRNFORM, NFORM,
rem n_err, n_expr, n_term, n_fact, n_primaire,
rem n_instr et n_paire
Initialisations
L2=Lx^2+Ly^2+Lz^2
print "L2=";L2
PRNFORM "[L2,x^2+y^2+z^2]"
PRNFORM "[L2,x^2+y^2]"

```

```

        stop
Initialisations:
        complex i
        rem -----
        rem Les NG générateurs G(1), ..., G(NG)
        rem -----
        NG=6
        var G(NG)
        G(1)=x
        G(2)=y
        G(3)=z
        G(4)=Lx
        G(5)=Ly
        G(6)=Lz
        rem -----
        rem Les commutateurs [ G(i1) , G(i2) ]=GG(i1,i2)
        rem pour i1>i2
        rem -----
        var GG(NG,NG)
        for i1=2,NG
            for i2=1,i1-1
                read U
                GG(i1,i2)=i*U
        next i2,i1
        data 0
        data 0, 0
        data 0, z, -y
        data -z, 0, x, -Lz
        data y, -x, 0, Ly, -Lx
        return

```

Sortie (51 s)

$$L2 = Lx^2 + Ly^2 + Lz^2$$

$$[L2, x^2 + y^2 + z^2] = 0$$

$$[L2, x^2 + y^2] = -4*i*x*z*Ly + 4*i*y*z*Lx + 2*x^2 + 2*y^2 - 4*z^2$$

Bibliographie

- E H Bareiss, *Sylvester Identity and Multistep Integer Preserving Gaussian Elimination*, *Math Comp* **22** (1968) 565
- S Bhowmick, R Bhattacharya & D Roy, *Iterations of Convergence Accelerating Nonlinear Transforms*, *Comp Phys Comm* **54** (1989) 31
- C Brezinski, *Accélération de la Convergence en Analyse Numérique* (Springer-Verlag 1977)
- J Brillhart & M A Morrison, *A Method of Factoring and the Factorisation of F_7* , *Math Comp* **29** (1975) 183
- J Davenport, Y Siret & E Tournier, *Calcul Formel* (Masson 1987)
- H Dörrie, *100 Great Problems of Elementary Mathematics* (Dover 1965)
- R W Gosper, *Proc Nat Acad Sci USA* **75** (1978) 40
- J Grotendorst, *MAPLE Programs for Converting Series Expansions to Rational Functions Using the Levin Transformation*, *Comp Phys Comm* **55** (1989) 325–335
- Handbook of Mathematical Functions*, édité par M Abramowitz et I A Stegun (Dover 1965)
- Hua Loo Keng, *Introduction to Number Theory* (Springer-Verlag 1982)
- D E Knuth, *The Art of Computer Programming*, vol 2 (Addison-Wesley 1981)
- N Koblitz, *Introduction to Elliptic Curves and Modular Forms* (Springer-Verlag 1982)
- N Koblitz, *A Course in Number Theory and Cryptography* (Springer-Verlag 1987)
- J J Labarthe, *Basic 1000d Manuel de Référence* (1990)
- L Landau & E Lifchitz, *Mécanique Quantique* (Mir Moscou 1966)
- H W Lenstra, Jr., *Factoring Integers with Elliptic Curves*, *Ann Math* **126** (1987) 649
- F Le Lionnais, *Les Nombres Remarquables* (Hermann 1983)
- D Levin, *Int J comput Math B* **3** (1973) 371
- J M Muller, *Arithmétique des Ordinateurs* (Masson 1989)
- J M Pollard, *A Monte-Carlo Method for Factorisation*, *BIT* **15** (1975) 331
- P Ribenboim, *The Book of Prime Number Records* (Springer-Verlag 1989)
- H Riesel, *Prime Numbers and Computer Methods for Factorization* (Birkhäuser 1985)
- D Wells, *The Penguin Dictionary of Curious and Interesting Numbers* (Penguin Books 1986)
- J H Wilkinson, *The Evaluation of the Zeros of Ill-conditioned Polynomials*, *Num Math* **1** (1959) 150

Notations

[a] Z	Désigne l'appui simultané sur les touches Alternate et Z
$d n$	L'entier d divise l'entier n
(a, b)	Plus grand commun diviseur (pgcd) des entiers a et b
$\lfloor x \rfloor$	Plus grand entier inférieur ou égal à x
$\lceil x \rceil$	Plus petit entier plus grand ou égal à x
$\log x$	Logarithme naturel de x
$\tan x$	Tangente de x

Index

!	12, 160	aligne	57
@k	117, 121, 132, 139	aliquote	64, 113, 251–4, 261–3
γ	10, 116, 183, 191–3	amiable	64
$\Gamma(x)$	159–98	André	176
$\zeta(x)$	171, 186	andre	176–7
π	10, 21, 88, 256	anticommutateur	227
ρ (factorisation)	83	appr	21, 256
ρ (transformation)	116–9	approximation	21
abenum	271–2	polynomiale	57–8
abs	149	rationnelle	152–8
accélération de la convergence	115–30, 152–8	argument	28
access	171, 173, 176, 269, 271	arrondi de l'unité	11
ACOM	227–9	asin	7
affichage	8–9	assignation	35, 48
aire	54	axes	133–4
aléatoire	29–30	axis	133–6, 161, 163, 175, 185, 260
algèbre		Bareiss	45
de Lie	226	barrière de potentiel	200
enveloppante	226	Basalg	2
non-commutative	225–36	base	13–4
algorithme d'Euclide étendu	65–6	base de facteurs	91
		Basic Algébrique	2

bernoulli 170
 Bernoulli 170, 178
 bibliographie 276
 binome 13
 binome\$ 13
 bnum 171–2, 180, 191, 269
 Bohr 145
 box 138
 Breit-Wigner 204
 Brillhart 17, 91
 brison 17, 82, 99–100
 brison_1 92–6
 Brun 18
 bsum 271
 cabs 28, 149, 205
 calcul
 conditionnel 47–8
 formel 31
 matriciel \rightarrow matrice
 modulaire 56
 puissance 56
 racine carrée 19, 72–5
 calendrier 68
 carg 28
 Casimir 235
 cc 28, 77
 cdr\$ 77, 107
 cercle 178
 change\$ 106, 250, 266
 charn 107
 checker 21
 chinois 67
 chinois1 67
 chinois2 67
 chinoiseq 19, 66, 72
 cint 139
 clear timer 14
 cls 134, 136
 coef 169
 coeff 36, 38, 121
 coefficient 36
 du binôme 13
 COM 227–9
 commutateur 227
 complex 47, 76, 149, 238
 complexp 231
 comput 68
 conc\$ 44, 250
 cond 47
 condition 47–8
 congruences 19, 65
 non linéaires 71–5
 constante
 d'Euler 10, 116, 183, 191–3
 de Brun 18
 de Gompertz 126
 contf 40
 continuum 145, 148, 158
 convergence 115–30
 convergent 21, 88
 copie écran 200
 copy 93, 117
 cos 53
 courbe 131–58, 179–80
 digamma 185
 elliptique 17, 100–5
 fonction γ 161
 orthogonales 163–6
 paramétrée 135
 Cramer 119
 cursh 200
 cursl 113, 200
 cxcmp 77
 cxfact 76–80
 cxfact\$ 78
 cxfact_c 78
 cxfact_p 78
 cxgcd 27, 76
 cxint 28
 cxmod 77
 cxnorm 77
 cycle amiable 64, 252–4
 cycle solaire 68
 c_fonctions 13
 dabord_prfact 83, 85–6, 99, 103
 data 44
 décodage 229–34
 decode 230

- decodex 231
- décorticage 36–7
- défilement de l'écran 21
- deg 17
- degf 36, 38
- degré 36
- den 208
- denf 40
- denr 146
- der 52
- dériver 52–3, 240
- dertrigo 53, 240
- déshomogénéiser 37
- det 44
- déterminant 44–6
- devdet 45
- develop 34–5, 107, 155, 167
- développement
 - asymptotique 145, 148, 166–7, 169, 182
 - limité 41–3, 119, 209–12, 241, 273–4
 - sec x 178
 - tan x 178
 - multipolaire 43
 - polynôme 34
- digamma 182–6
- digammap 191–3
- Disquisitiones Arithmeticae 19, 72
- div 38
- diviseur 63
- division 11
 - puissances croissantes 38
 - euclidienne 38
- divr 11–2, 84
- do 21
- droite 57
- dsum 55, 173, 267
- dsumb 173, 267–8
- e 20, 88, 256
- effet tunnel 202
- egd 248
- egypt 23, 246–7
- Egyptien 23–5
- élément primitif 61
- elementn 78
- éléments simples 40–1, 187, 189
- elementv 77–8
- elementy 77–8
- elim 49, 259
- élimination 49–51, 179
- endif 250
- enum 173–4, 269
- enum_bis 269
- équation 49–51
 - aux différences 173, 183
 - cartésienne 178–9
 - différentielle 214–24
 - linéaires 202–3, 238–9
 - non-linéaire 239–40
 - modulaire 19, 65–8
 - paramétrique 178–9
 - polaire 178–9
 - de Schrödinger 202
- erf 196–7
- erreur
 - division 65
 - instruction illégale 32
 - mémoire 181
 - nombre complexe 33
- eset\$ 77, 106
- esum 271
- Euler 10, 68, 75, 116, 160, 191
- euler_phi 60
- exact 21
- exg 14, 137
- exit 22, 139
- exp 28, 134, 181
- exp1 10
- exponentiation modulaire 19
- exposant complexe 28
- extension quadratique 73–5
- factor 34–5, 147, 167
- factorielle 12, 160
- factorisation
 - des entiers 16, 81–114
 - Brillhart et Morrison 91–100
 - Fermat 86
 - Legendre 87

- Lenstra 100–5
- Monte-Carlo 83
- méthode ρ 83
- entier de Gauss 75–80
- polynôme 34–6
- factorn** 188
- factorp** 188
- fastpsi** 267
- fcont** 256–9
- Fermat 61, 75, 86
- fermat\$** 93, 255–6
- Fibonacci 14
- float** 10
- fonction 13
 - arithmétique multiplicative 63
 - d'erreur 196–7
 - d'onde 145–58
 - digamma 182–93
 - Gamma incomplète 193–6
 - hypergéométrique dégénérée 43, 145, 149–58, 194
 - polygamma 182–93
 - trigonométrique 52–3
 - Zêta 171, 186
 - Γ 159–98
- fonctions symétriques 39–40
- for** 10–1, 243
- forc** 242–3
- format** 8
- formatl** 9
- formatm** 9
- formatx** 20
- formf** 34–6, 38, 40, 188
- formule de Stirling 169
- forv** 21, 243
- fplot** 132, 134–6, 161, 163, 175, 185, 206, 260
- fraction 64
 - continue 87–91, 194, 256–9
 - égyptienne 23–5, 246–9
 - rationnelle 33
- Fresnel 198
- fsubs** 147, 152
- gamma** 160, 162, 167–9, 180
- gammap** 194–6
- gamma_it** 167–9, 183
- gamma_psic** 167–8, 183
- Gauss 10, 19, 27, 72, 75
- gcd** 38
- gcdr** 11–2, 29, 61, 254
- géométrie plane 56–7
- ghyg** 149–52
- gint** 22
- Gompertz 126
- Gosper 191
- gp** 180–2
- gp_it** 191
- graphmode** 137
- groupe 100
 - cyclique 61
- hardcopy** 200
- Hartree 145
- Hasse 101
- hidecm** 200
- homog** 37
- homogénéiser 37
- hydrogène 145–58
- hyg** 149–52, 155
- hyperbole 52
- identificateur 32
- im** 28
- imprimante 200
- indicateur d'Euler 60–3
- indiction 68
- input** 22
- instabilité 25–7, 50
- integerp** 13, 17
- intégration
 - numérique 10
 - formelle 53–5, 240–1
- intg** 54–5
- intg1** 54, 240
- intlq** 149
- introot** 17
- intsqr** 12, 92, 241
- inv** 48–9
- invariant de Casimir 235
- inversion d'une matrice 44, 46–8

- invm 46–7, 215
- irrationnel quadratique 88
- irréductible 35
- Jacobi 68–70
- justc\$
- justl\$ 14, 121
- justr\$ 205, 249
- kmult 97–9
- Knuth 83
- Kraïtchik 91
- Labarthe 276
- Lagrange 61, 88
- Legendre 68–70, 87
- legendre 69, 74, 97
- lemniscate 178–80
- len 14
- Lenstra 100
- lenstra 17, 82, 102–5
- Levin 119
- levin 120–4
- levinpt 120–4, 152
- levinpu 120–4
- levinpv 120–4
- levint 125–30
- levinu 125–30
- levinv 125–30
- line 132–3, 139, 161, 200
- lit 52
- litp 38
- littéral 32
 - complexe 27
- local 45
- log 28, 181
- log1 117
- loggamma 182–4
- loggamma 191–3
- loi de composition 100
- lower\$ 121
- l_begin 133
- l_end 133, 200
- l_type 133, 200
- l_width 133, 200
- MATH 3
- matrice 43–7, 214–9
 - déterminant 44–6
 - inversion 44, 46–7, 215
 - polynôme caractéristique 45
- mdff 56, 71–2
- mdgcd 71
- mdinv 74
- mdmod 74
- mdpwr 56, 74
- mdpwre 19, 61, 65, 254
- mediatrice 57
- Mersenne 15
- message 22, 200
- mid\$ 230
- milieu 57
- min 13, 138
- minimum d'une fonction 186
- mod 38, 48–9, 208
- modr 11–2, 18
- mods 18
- module 28
- modulo 18
- Morrison 17, 91
- moteur 213–24
- mtimer 14
- Neper 20
- nextperm 20, 30, 45, 175, 250
- NFORM 229–34
- nombre
 - aléatoire 29–30
 - algébrique 47–9
 - inverse 48
 - complexe 27–8
 - aléatoire 29
 - conjugué 28, 187
 - partie imaginaire 28
 - partie réelle 28
 - d'or 68
 - des diviseurs 63
 - entier 11–9
 - aléatoire 29
 - de Gauss 27, 75
 - flottant 6–11
 - modulaire 18–9, 56
 - parfait 63

- premier 15–8
- rationnel 19–25
- nombres amiables 64, 252–4
 - d'André 176–7
 - d'Euler 173–5, 177
 - de Bernoulli 166, 169–73, 177
 - de Mersenne 15
 - premiers jumeaux 18, 245–6
- non-commutatif 225–36
- non-résidu quadratique 68–70
- norm 17, 37, 60, 84, 93
- notations 277
- notilde 9
- num 208
- numr 146
- odd 17
- Ohm 214
- opérateurs non-commutatifs 225–36
- ordf 36, 38
- ordre 36
- origin 132, 139
- pack 181
- partfrac\$ 40
- particule 200
- partie imaginaire 28
 - réelle 28
 - singulière 40
- périmètre 178
- permutation 20, 249–51, 270–1
 - aléatoire 29–30
 - d'André 175–8
 - en zigzag 175–8
- perpinf 57
- pgcd 11, 27, 38, 277
- phantom 107
- phistar 197
- pi 6
- plot 132, 139
- plus grand commun diviseur → pgcd
- pôle 161, 187, 189
- pollard 16, 82–6, 99
- Pollard 83
- pollard_iter 84
- polyappr 57
- polygamma 182–6
- polygammap 191–3
- polyl n 188
- polym 17, 37, 93
- polymn 17, 37, 60, 84
- polynôme 33
 - de Bernoulli 170–3
 - caractéristique 45
 - d'Euler 173–5
 - forme produit 33
 - modulaire 56
 - racines 51–2, 238
- pop 137, 139
- potentiel 200
- ppwr 12–3, 121, 160, 167, 180
- PR 227–9
- précision 6, 8
- precision 6, 8, 10, 149
- precision2 149, 241
- prf 106–12
- prfact 17, 60, 72, 77, 82–3, 93
- prfact\$ 16, 82, 255
- prfactb 82, 106–12, 261
- prfactb\$ 82, 106–12
- primalité 16
- prime 16, 245
- primitif 61
- print 13
- prinv 65, 103, 254
- PRM 227–9
- PRNFORM 231–4
- produit de polynômes 33
- programmation récursive 23–4, 26–7, 242, 246
- prsq r 19, 72
- prsq r1 73
- prtst 15–6, 84, 245
- psing 40–1, 187, 189
- ptsout 133
- puits de potentiel 202
- push 113, 132, 137, 139
- qplot 137–44, 147, 197, 220
- quadratfrei 17
- quantique 145–58, 199–212

- quotient 38
- quotient incomplet 88
- racine *kième* 51
 - carrée 7, 89
 - modulaire 19, 72–5
 - multiple 102
 - d'un polynôme 51–2, 238
- random 22, 29, 78
- randomize 30
- ratnump 22
- re 28, 167, 187
- read 44
- réciprocité quadratique 69
- récurrence 23–4, 26–7, 242, 246
- red 51, 179, 259
- réduite 88
- remember 26
- rep-units 16
- répartition normale 197
- repeat 22
- réseau de courbes 163–6
- résidu 61
 - quadratique 68–70
- resolution 200
- reste 18
 - modulaire 48
- restes chinois 19
- return 13
- rho_serie 117–9, 128–30
- rnd 29
- romberg 10
- root 17, 51, 146, 245, 255
- rsum 188–90
- Runge-Kutta 219–24
- Schrödinger 202
- scos 42
- scrolling 21
- sdpolyb 267–8
- search 251
- select 26, 117
- série 115–30, 149
 - de Laurent 209
 - rationnelle 186–91
- sexp 42, 169, 241
- sgeq 50–1, 239
- shyg 43, 146, 152, 211
- sigma 63
- sin 7, 53
- sinh 7
- sleq 50, 203, 239
- slog1 42
- so(4) 234
- sommation
 - en termes finis 55–6
 - séries rationnelles 186–91
- somme de deux carrés 75
 - des diviseurs 63
- sort 138, 251
- sortie des nombres 8–9
 - disque 266
- spirale sinusoïdale 178
- sqr 7, 10, 28
- sqr1 17, 146
- square-free 17
- sroot 40–1, 189, 210
- ssec 178, 273–4
- ssin 42, 241
- stack 113
- stan 178, 273–4
- statistique 196
- Stein 24–5, 248
- stirling 169
- stop 10, 22
- str\$ 14, 34, 41–3, 211, 241
- su(2) 226
- subs 37, 41
- subsr 37, 48, 208
- subsrr 48
- substitution 37, 48
- suite aliquote 64, 113, 251–4, 261–3
- sum 20, 55
- sumsq 75–6
- sure? 22
- symbole de Jacobi 68–70
 - de Legendre 68–70
- symétrie 39
- symf 39–40
- s_pro 51, 106, 112

s_var 106, 112, 147
tangente 52
taylor 41, 169
temps de calcul 8, 12, 14
text 133, 137, 139
théorème des restes chinois 66, 72
tilde 6, 9
tilde 9
timer 14, 21–2
tracé de courbes 131–58
trajectoire 136
 complexe 163–6
transformation
 conforme 163–6
 d'Euler 116
 de Levin 119–27, 152–8
 ρ 116–9
trigop 53, 240
trigox 53, 240
type 32
t_angle 140
t_ensemble 76, 106–8
t_height 133, 137
t_type 133
unité 27
until 22
using 9, 36, 40
vadd 14, 77
value 10, 13, 230, 246
Vandermonde 119
var 44
variable 32
vdi 133
vset\$ 53
v_ensemble 53
v_fonction 13
while 20, 249
xbios(33) 200
xqt 242–4, 250
zerop 51
zigzag 175