Une méthodologie du calcul des fonctions élémentaires

Jean-Michel Muller

THESE

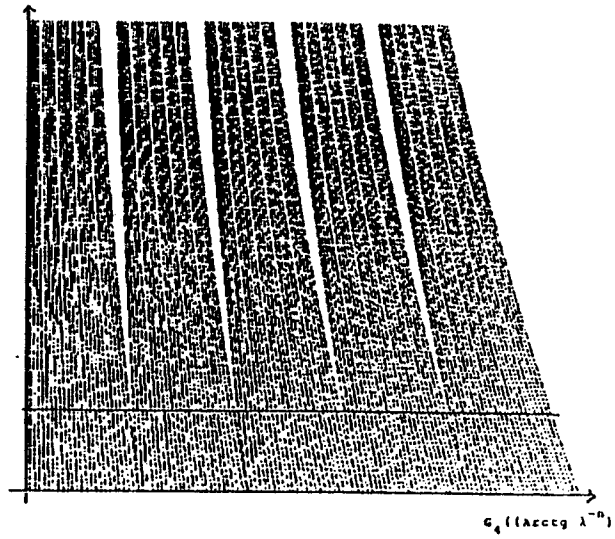
présentée par

Jean - Michel MULLER

pour obtenir le titre de **DOCTEUR**

de l'INSTITUT NATIONAL POLYTECHNIQUE DE GRENOBLE

Mathématiques appliquées



METHODOLOGIES DE CALCUL DES FONCTIONS ELEMENTAIRES

Date de soutenance : 13 Septembre 1985

Composition du jury :

F. Robert	Président
F. Anceau	
M. Cosnard	
J. Gastinel	
A. Guyot	Examinateurs
G. Noguez	

Thèse préparée au sein du laboratoire TIM 3.

THESE

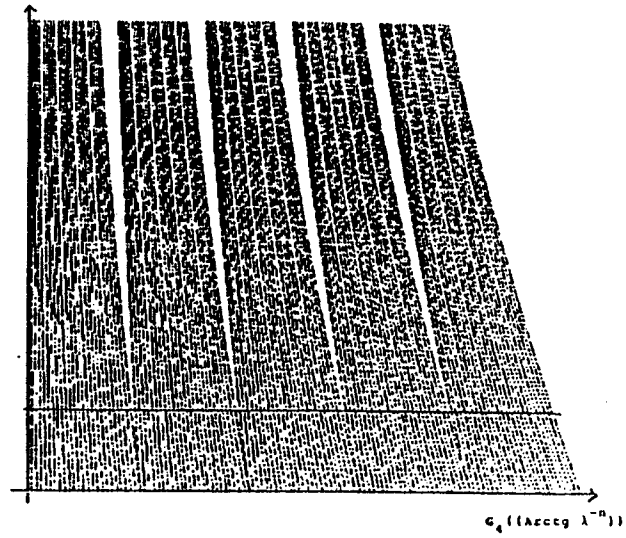
présentée par

Jean - Michel MULLER

pour obtenir le titre de **DOCTEUR**

de l'INSTITUT NATIONAL POLYTECHNIQUE DE GRENOBLE

Mathématiques appliquées



METHODOLOGIES DE CALCUL DES FONCTIONS ELEMENTAIRES

Date de soutenance : 13 Septembre 1985

Composition du jury :

F. Robert	Président
F. Anceau	
M. Cosnard	
J. Gastinel	
A. Guyot	Examinateurs
G. Noguez	

Thèse préparée au sein du laboratoire TIM 3.

INSTITUT NATIONAL POLYTECHNIQUE DE GRENOBLE

Année universitaire 1982-1983

Président de l'Université : D. BLOCH

Vice-Président : René CARRE
Hervé CHERADAME
Marcel IVANES

PROFESSEURS DES UNIVERSITES :

ANCEAU François	E.N.S.I.M.A.G.
BARRAUD Alain	E.N.S.I.E.G.
BAUDELET Bernard	E.N.S.I.E.G.
BESSON Jean	E.N.S.E.E.G.
BLIMAN Samuel	E.N.S.E.R.G.
BLOCH Daniel	E.N.S.I.E.G.
BOIS Philippe	E.N.S.H.G.
BONNETAIN Lucien	E.N.S.E.E.G.
BONNIER Etienne	E.N.S.E.E.G.
BOUVARD Maurice	E.N.S.H.G.
BRISSONNEAU Pierre	E.N.S.I.E.G.
BUYLE BODIN Maurice	E.N.S.E.R.G.
CAVAIGNAC Jean-François	E.N.S.I.E.G.
CHARTIER Germain	E.N.S.I.E.G.
CHENEVIER Pierre	E.N.S.E.R.G.
CHERADAME Hervé	U.E.R.M.C.P.P.
CHERUY Arlette	E.N.S.I.E.G.
CHIAVERINA Jean	U.E.R.M.C.P.P.
COHEN Joseph	E.N.S.E.R.G.
COUMES André	E.N.S.E.R.G.
DURAND Francis	E.N.S.E.E.G.
DURAND Jean-Louis	E.N.S.I.E.G.
FELICI Noël	E.N.S.I.E.G.
FOULARD Claude	E.N.S.I.E.G.
GENTIL Pierre	E.N.S.E.R.G.
GUERIN Bernard	E.N.S.E.R.G.
GUYOT Pierre	E.N.S.E.E.G.
IVANES Marcel	E.N.S.I.E.G.
JAUSSAUD Pierre	E.N.S.I.E.G.
JOUBERT Jean-Claude	E.N.S.I.E.G.
JOURDAIN Geneviève	E.N.S.I.E.G.
LACOUME Jean-Louis	E.N.S.I.E.G.
LATOMBE Jean-Claude	E.N.S.I.M.A.G.

.../...

LESSIEUR Marcel	E.N.S.H.G.
LESPINARD Georges	E.N.S.H.G.
LONGEQUEUE Jean-Pierre	E.N.S.I.E.G.
MAZARE Guy	E.N.S.I.M.A.G.
MOREAU René	E.N.S.H.G.
MORET Roger	E.N.S.I.E.G.
MOSSIERE Jacques	E.N.S.I.M.A.G.
PARIAUD Jean-Charles	E.N.S.E.E.G.
PAUTHENET René	E.N.S.I.E.G.
PERRET René	E.N.S.I.E.G.
PERRET Robert	E.N.S.I.E.G.
PIAU Jean-Michel	E.N.S.H.G.
POLOUJADOFF Michel	E.N.S.I.E.G.
POUPOT Christian	E.N.S.E.R.G.
RAMEAU Jean-Jacques	E.N.S.E.E.G.
RENAUD Maurice	U.E.R.M.C.P.P.
ROBERT André	U.E.R.M.C.P.P.
ROBERT François	E.N.S.I.M.A.G.
SABONNADIERE Jean-Claude	E.N.S.I.E.G.
SAUCIER Gabrielle	E.N.S.I.M.A.G.
SCHLENKER Claire	E.N.S.I.E.G.
SCHLENKER Michel	E.N.S.I.E.G.
SERMET Pierre	E.N.S.E.R.G.
SILVY Jacques	U.E.R.M.C.P.P.
SOHM Jean-Claude	E.N.S.E.E.G.
SOUQUET Jean-Louis	E.N.S.E.E.G.
VEILLON Gérard	E.N.S.I.M.A.G.
ZADWORNY François	E.N.S.E.R.G.

PROFESSEURS ASSOCIES

BASTIN Georges	E.N.S.H.G.
BERRIL John	E.N.S.H.G.
CARREAU Pierre	E.N.S.H.G.
GANDINI Alessandro	U.E.R.M.C.P.P.
HAYASHI Hirashi	E.N.S.I.E.G.

PROFESSEURS UNIVERSITE DES SCIENCES SOCIALES (Grenoble II)

BOLLIET Louis
Chatelin Françoise

PROFESSEURS E.N.S. Mines de Saint-Etienne

RIEU Jean
SOUSTELLE Michel

CHERCHEURS DU C.N.R.S.

FRUCHART Robert	Directeur de Recherche
VACHAUD Georges	Directeur de Recherche

.../...

ALLIBERT Michel	Maître de Recherche
ANSARA Ibrahim	Maître de Recherche
ARMAND Michel	Maître de Recherche
BINDER Gilbert	
CARRE René	Maître de Recherche
DAVID René	Maître de Recherche
DEPORTES Jacques	
DRIOLE Jean	Maître de Recherche
GIGNOUX Damien	
GIVORD Dominique	
GUELIN Pierre	
HOPFINGER Emil	Maître de Recherche
JOUD Jean-Charles	Maître de Recherche
KAMARINOS Georges	Maître de Recherche
KLEITZ Michel	Maître de Recherche
LANDAU Ioan-Dore	Maître de Recherche
LASJAUNIAS J.C.	
MERMET Jean	Maître de Recherche
MUNIER Jacques	Maître de Recherche
PIAU Monique	
PORTESEIL Jean-Louis	
THOLENCE Jean-Louis	
VERDILLON André	

CHERCHEURS du MINISTERE de la RECHERCHE et de la TECHNOLOGIRE (Directeurs et Maîtres de Recherches, ENS Mines de St. Etienne)

LESBATS Pierre	Directeur de Recherche
BISCONDI Michel	Maître de Recherche
KOBYLANSKI André	Maître de Recherche
LE COZE Jean	Maître de Recherche
LALAUZE René	Maître de Recherche
LANCELOT Francis	Maître de Recherche
THEVENOT François	Maître de Recherche
TRAN MINH Canh	Maître de Recherche

PERSONNALITES HABILITEES à DIRIGER des TRAVAUX de RECHERCHE (Décision du Conseil Scientifique)

ALLIBERT Colette	E.N.S.E.E.G.
BERNARD Claude	E.N.S.E.E.G.
BONNET Rolland	E.N.S.E.E.G.
CAILLET Marcel	E.N.S.E.E.G.
CHATILLON Catherine	E.N.S.E.E.G.
CHATILLON Christian	E.N.S.E.E.G.
COULON Michel	E.N.S.E.E.G.
DIARD Jean-Paul	E.N.S.E.E.G.
EUSTAPOPOULOS Nicolas	E.N.S.E.E.G.
FOSTER Panayotis	E.N.S.E.E.G.

.../...

GALERIE Alain	E.N.S.E.E.G.
HAMMOU Abdelkader	E.N.S.E.E.G.
MALMEJAC Yves	E.N.S.E.E.G. (CENG)
MARTIN GARIN Régina	E.N.S.E.E.G.
NGUYEN TRUONG Bernadette	E.N.S.E.E.G.
RAVAINE Denis	E.N.S.E.E.G.
SAINFORT	E.N.S.E.E.G. (CENG)
SARRAZIN Pierre	E.N.S.E.E.G.
SIMON Jean-Paul	E.N.S.E.E.G.
TOUZAIN Philippe	E.N.S.E.E.G.
URBAIN Georges	E.N.S.E.E.G. (Laboratoire des ultra-réfractaires ODEILLON)
GUILHOT Bernard	E.N.S. Mines Saint Etienne
THOMAS Gérard	E.N.S. Mines Saint Etienne
DRIVER Julien	E.N.S. Mines Saint Etienne
BARIBAUD Michel	E.N.S.E.R.G.
BOREL Joseph	E.N.S.E.R.G.
CHOVET Alain	E.N.S.E.R.G.
CHEHIKIAN Alain	E.N.S.E.R.G.
DOLMAZON Jean-Marc	E.N.S.E.R.G.
HERAULT Jeanny	E.N.S.E.R.G.
MONLLOR Christian	E.N.S.E.R.G.
BORNARD Guy	E.N.S.I.E.G.
DESCHIZEAU Pierre	E.N.S.I.E.G.
GLANGEAUD François	E.N.S.I.E.G.
KOFMAN Walter	E.N.S.I.E.G.
LEJEUNE Gérard	E.N.S.I.E.G.
MAZUER Jean	E.N.S.I.E.G.
PERARD Jacques	E.N.S.I.E.G.
REINISCH Raymond	E.N.S.I.E.G.
ALEMANY Antoine	E.N.S.H.G.
BOIS Daniel	E.N.S.H.G.
DARVE Félix	E.N.S.H.G.
MICHEL Jean-Marie	E.N.S.H.G.
OBLED Charles	E.N.S.H.G.
ROWE Alain	E.N.S.H.G.
VAUCLIN Michel	E.N.S.H.G.
WACK Bernard	E.N.S.H.G.
BERT Didier	E.N.S.I.M.A.G.
CALMET Jacques	E.N.S.I.M.A.G.
COURTIN Jacques	E.N.S.I.M.A.G.
COURTOIS Bernard	E.N.S.I.M.A.G.
DELLA DORA Jean	E.N.S.I.M.A.G.
FONLUPT Jean	E.N.S.I.M.A.G.
SIFAKIS Joseph	E.N.S.I.M.A.G.
CHARUEL Robert	U.E.R.M.C.P.P.
CADET Jean	C.E.N.G.
COEURE Philippe	C.E.N.G. (LETI)

.../...

DELHAYE Jean-Marc	C.E.N.G. (STT)
DUPUY Michel	C.E.N.G. (LETI)
JOUVE Hubert	C.E.N.G. (LETI)
NICOLAU Yvan	C.E.N.G. (LETI)
NIFENECKER Hervé	C.E.N.G.
PERROUD Paul	C.E.N.G.
PEUZIN Jean-Claude	C.E.N.G. (LETI)
TAIEB Maurice	C.E.N.G.
VINCENDON Marc	C.E.N.G.

LABORATOIRES EXTERIEURS

DEMOULIN Eric	C.N.E.T.
DEVINE	C.N.E.T. (R.A.B.)
GERBER Roland	C.N.E.T.
MERCKEL Gérard	C.N.E.T.
PAULEAU Yves	C.N.E.T.
GAUBERT C.	I.N.S.A. Lyon

ECOLE NATIONALE SUPERIEURE DES MINES DE SAINT-ETIENNE

Directeur : Monsieur M. MERMET
Directeur des Etudes et de la formation : Monsieur J. LEVASSEUR
Directeur des recherches : Monsieur J. LEVY
Secrétaire Général : Mademoiselle M. CLERGUE

Professeurs de 1ère Catégorie

COINDE	Alexandre	Gestion
GOUX	Claude	Métallurgie
LEVY	Jacques	Métallurgie
LOWYS	Jean-Pierre	Physique
MATHON	Albert	Gestion
RIEU	Jean	Mécanique - Résistance des matériaux
SOUSTELLE	Michel	Chimie
FORMERY	Philippe	Mathématiques Appliquées

Professeurs de 2ème catégorie

HABIB	Michel	Informatique
PERRIN	Michel	Géologie
VERCHERY	Georges	Matériaux
TOUCHARD	Bernard	Physique Industrielle

Directeur de recherche

LESBATS	Pierre	Métallurgie

Maîtres de recherche

BISCONDI	Michel	Métallurgie
DAVOINE	Philippe	Géologie
FOURDEUX	Angeline	Métallurgie
KOBYLANSKI	André	Métallurgie
LALAUZE	René	Chimie
LANCELOT	Francis	Chimie
LE COZE	Jean	Métallurgie
THEVENOT	François	Chimie
TRAN MINH	Canh	Chimie

Personnalités habilitées à diriger des travaux de recherche

DRIVER	Julian	Métallurgie
GUILHOT	Bernard	Chimie
THOMAS	Gérard	Chimie

Professeur à l'UER de Sciences de Saint-Etienne

VERGNAUD	Jean-Maurice	Chimie des Matériaux & chimie industrielle

****

" Le mouvement infini, le point qui remplit tout, le moment de repos : infini sans quantité, indivisible et infini. "

Pascal

A mes parents.

Je tiens à remercier monsieur le professeur François Robert pour l'honneur qu'il me fait en présidant le jury de cette thèse, j'ai pu à maintes reprises apprécier sa compétence scientifique et sa chaleur humaine.

De même, j'adresse toute ma gratitude à Michel Cosnard, qui par son aide constante et attentive m'a guidé tout au long de ce travail de thèse.

Mes remerciements vont également à François Anceau, Gérard Noguez, Jean Gastinel et Alain Guyot, qui ont accepté de faire partie du Jury.

Je remercie encore tous les membres de l'équipe d'algorithmique mathématique de TIM3, et notamment ceux du groupe itérations de cette équipe; ainsi que Claire Dicrescenzo et André Eberhard, dont les conseils m'ont souvent été d'une grande utilité.

Il est impossible d'oublier dans ces remerciements les joyeux lurons membres titulaires ou honoraires du bureau 58 de la tour IRMA: Pascal Klein, Christophe Masse, Julio Medina, Didier Pellegrin et mon vieux complice Yves Paillet; ainsi qu'Edith, Régine et les grandes amies Marie-Hélène et Catherine.

De même, je ne peux oublier le "groupe FELIN", Bertrand Hochet, Hassan Ouaouicha et Eytan Zysman : j'ai eu beaucoup de plaisir à travailler avec eux.

Enfin, je remercie les membres du service reprographie de l'IMAG pour la réalisation matérielle de ce travail.

SOMMAIRE

INTRODUCTION

Le calcul des fonctions mathématiques usuelles (fonctions trigonométriques, exponentielle, logarithme, etc...) constitue la base de tout le calcul scientifique: il est évident que si l'implémentation sur ordinateur de ces fonctions s'avère peu satisfaisante, soit du point de vue de la précision, soit du point de la rapidité, aucun travail correct n'est possible.

Il est donc tout naturel de se pencher sur ce problème critique: c'est le but que je me suis donné dans cette thèse.

Il y a encore peu de temps, l'implémentation de ces fonctions ne se concevait que de manière **logicielle** , mais de nos jours, les possibilités sans cesse croissantes de l'intégration sur silicium nous autorisent à envisager des réalisations **matérielles** . La différence considérable entre ces deux modes d'implémentation induira inévitablement une différence également très grande entre les algorithmes utilisés: les méthodes utilisées traditionnellement en software pour le calcul de ces fonctions, basées généralement sur des approximations par des **polynômes** ou des **fractions rationnelles** ne peuvent plus s'appliquer en matériel, car elles font appel à des multiplications et des divisions en virgule flottante sur un grand nombre de bits, ce qui serait trop coûteux en surface de silicium.

Le seul point commun entre ces diverses techniques réside dans la notation des nombres réels en **virgule flottante** , universellement adoptée en calcul scientifique. Le caractère très particulier de cette notation complique quelque peu les problèmes liés au contrôle et à la maîtrise de l'erreur. De nombreux auteurs ont d'ailleurs étudié cette question, en particulier M. Pichat (70), U.W. Kulisch et W.L.Miranker (49), R.P. Brent (9) et W.J. Cody (22).

Le premier chapitre de cette thèse sera consacré à ce sujet, on y montre en particulier que quelle que soit la qualité de l'algorithme utilisé, il existe un seuil d'erreur **inévitable** , en deça duquel on ne peut descendre, et qui est inhérent à la fonction calculée, ainsi qu'à la **base de numération** du système à virgule flottante utilisé.

Il convient de ne pas croire que cette base de numération est systématiquement 2 : de nombreux ordinateurs travaillent en base 16 (c'est par exemple le cas de l'IBM 370), tandis que la plupart des calculatrices de poche ou des petits ordinateurs "personnels" travaillent en base 10, afin d'éviter les conversions décimal/binaire nécessitées par les entrées/sorties fréquentes sur ce genre de machines. Le micro-ordinateur personnel TI 99 de Texas-Instruments utilise même la base 100 , tandis que l'**ILLIAC III** effectuait certains calculs en base 256!

Le deuxième chapitre discutera de l'implémentation **logicielle** des fonctions élémentaires, ainsi que des problèmes liés au **test** de cette implémentation.

Ce domaine à déjà fait l'objet d'une étude considérable: on connait depuis longtemps des polynômes et des fractions rationnelles d'approximation des principales fonctions mathématiques. En 1968, Cheney, Hart, Lawson, Maehly, Mesztenyi, Rice, Tacher et Witzgall publiaient un ouvrage extrèmement complet à ce sujet (44).

Toutefois, pour l'élaboration de tels algorithmes, on ne se préoccupe de tenir compte des particularités arithmétiques de la machine sur laquelle on travaille (base de numération, mode d'arrondi, existence de chiffres de garde, etc...) que depuis peu de temps. W. Cody ((19), (21), (22)) a donné de nombreux résultats concernant ce problème, notamment dans le livre qu'il a publié en 1980 avec W. Waite (19).

R.P. Brent (9) a également beaucoup travaillé dans cette voie, tout particulièrement en ce qui concerne la **complexité** et la **multiprécision** .

Notre propos sera d'**automatiser** la recherche d'une approximation liée au "contexte machine". Dans ce but, nous reprendrons les idées développées par W. Cody pour le **test** des logiciels de calcul des fonctions élémentaires, en montrant qu'elles peuvent également être utilisées pour la **conception** de programmes.

Les chapitres suivants traitent du point que j'ai le plus approfondi: le calcul **matériel** des fonctions élémentaires. En effet, les progrès spectaculaires de la micro-électronique permettent désormais l'intégration d'algorithmes calculant ces fonctions.

Ce domaine a été abordé en 1959 par J. Volder (82) , qui présenta l'algorithme CORD IC (COordinate Rotations on a DIgital Computer), qui permet de calculer les fonctions sinus, cosinus et arctangente, et ce en n'effectuant que des additions, des soustractions, et des **multiplications par des puissances entières de 2** qui, sur une machine travaillant en binaire, se ramène soit à un **décalage** des bits de mantisse en arithmétique à virgule fixe, soit à une **addition/soustraction** aux bits d'exposant en arithmétique à virgule flottante. Par abus de langage, nous appellerons dans tous les cas cette opération un **décalage** .

En 1971, J. Walther (83) montre que CORDIC, moyennant de petites modifications, peut être étendu au calcul des fonctions hyperboliques, ainsi qu'à l'exponentielle, au logarithme et à la racine carrée. Il constate en outre que les algorithmes usuels de multiplication et de division peuvent s'exprimer comme un cas particulier de CORDIC (il est d'ailleurs à noter que dès 1962, J.E. Meggitt (55) était à même de calculer certaines fonctions élémentaires par des procédés de "pseudo-division" et de "pseudo-multiplication" .

On obtient ainsi un **schéma unifié** permettant de calculer les principales fonctions mathématiques, et qui se prête particulièrement bien à une réalisation matérielle (CORDIC a d'ailleurs été choisi pour la calculatrice HP 35 de

Hewlett-Packard).

L'algorithme CORDIC, dans la version de Walther, est basé sur l'itération suivante:

$$x_{n+1} = x_n - m\, d_n y_n 2^{-n}$$

$$y_{n+1} = y_n + d_n x_n 2^{-n}$$

$$z_{n+1} = z_n - d_n e_n$$

où les choix du paramétre m,des variables d_n et des constantes e_n sont présentés dans la figure 1.

L'exponentielle est obtenue par addition des fonctions sh et ch, et le logarithme est obtenu en effectuant un changement de variable permettant d'utiliser la relation :

$$Argth\ x = 1/2\ .\ Log\ (|1+x| / |1-x|)$$

En 1970, De Lugish (32) étudie une classe d'algorithmes similaires, tandis que Chen (16) développe des idées originales à ce sujet.

On peut constater que tous ces algorithmes reviennent, sur un système travaillant en base 2, à approximer l'argument t de la fonction que l'on désire calculer par une expression de la forme:

$$t \sim d_0 e_0 + d_1 e_1 + \ldots + d_n e_n \qquad d_i = \pm 1, 0$$

$$\begin{cases} K = \prod_{i=0}^{\infty} \mathrm{Cos}\, e_i \\ K' = \prod_{i=0}^{\infty} \mathrm{Ch}\, e_i \end{cases}^{**} \qquad \begin{cases} x_{n+1} = x_n - m\, d_n\, y_n\, 2^{-n} \\ y_{n+1} = y_n + d_n\, x_n\, 2^{-n} \\ z_{n+1} = z_n - d_n\, e_n \end{cases}$$

fonction	valeurs init.	m	e_n	d_n	résultats
sinus/cosinus	$x_0 = K$ $y_0 = 0$	1	$\mathrm{Arctg}\, 2^{-n}$	$\mathrm{sign}\, z_n$	$x_n \longrightarrow \cos z_0$ $y_n \longrightarrow \sin z_0$
Arctg	$z_0 = 0$	1	" " " "	$-\mathrm{sign}\, y_n$	$z_n \longrightarrow \mathrm{arctg}\, y_0 / x_0$
sh/ch **	$x_0 = K'$ $y_0 = 0$	-1	$\mathrm{Argth}\, 2^{-n}$	$\mathrm{sign}\, z_n$	$x_n \longrightarrow \mathrm{ch}\, z_0$ $y_n \longrightarrow \mathrm{sh}\, z_0$
Argth **	$z_0 = 0$	-1	" " " "	$-\mathrm{sign}\, y_n$	$z_n \longrightarrow \mathrm{argth}\, y_0 / x_0$
multiplication	$y_0 = 0$	0	2^{-n}	$\mathrm{sign}\, z_n$	$y_n \longrightarrow x_0 z_0$
division	$z_0 = 0$	0	2^{-n}	$-\mathrm{sign}\, y_n$	$z_n \longrightarrow y_0 / x_0$

N.B. le symbole ** signifie que lors de l'itération comme dans le produit infini, les termes correspondant à i = 4, 13, 40, ... k, ... , 3k+1 doivent être répétés.

Fig. 1 Le schéma CORDIC de Volder et Walther.

où les e_i sont des constantes précalculées.

Notre but est de dégager une théorie générale, valable pour d'autres bases de numération et pour d'autres valeurs des coefficients d_i , afin de pouvoir construire une classe d'algorithmes, englobant ceux que nous venons de présenter succintement. Ceci sera fait dans le chapitre **III**, qui présentera une nouvelle notion: celle de **base discrète**, qui revêt également un intérêt en théorie des nombres, puisqu'elle permet de généraliser le concept de **base de numération**.

Le chapitre **IV** expose les principaux algorithmes déduits de la partie présentée au chapitre **III**. Tous ces algorithmes sont exempts de multiplications et n'utilisent que des **additions/soustractions** et des **décalages**. On présente comme cas particuliers les algorithmes de Volder et Walther, ainsi que de nouveaux algorithmes, comme celui de calcul de **l'exponentielle complexe**.

Au cours du chapitre **V**, on expose des méthodes nouvelles permettant d'effectuer rapidement en matériel la **réduction d'argument**, opération qui consiste à se ramener aux intervalles de convergence des algorithmes utilisés. Par exemple, si l'on désire calculer le sinus de 10000 avec un algorithme qui converge entre 0 et $\pi/2$, la réduction d'argument consiste à trouver le nombre x de l'intervalle $[0 , \pi/2]$ tel que $(10000-x)$ est un multiple de π, soit $x \sim 0.31058...$

On montre en particulier le lien entre un de nos algorithmes de réduction d'argument et les algorithmes dérivés de l'étude du chapitre **III**.

Le dernier chapitre sera consacré à une présentation rapide du coprocesseur **FELIN** (Fonctions ELémentaires INtégrées) de calcul des fonctions élémentaires en virgule flottante que nous sommes en train de réaliser au sein du laboratoire TIM3, en collaboration avec l'équipe d'architecture des ordinateurs. Ce circuit, destiné à être un coprocesseur du MOTOROLA 68000, utilise des algorithmes présentés dans cette thèse.

Chapître I

Conditionnement de fonctions et représentation en virgule flottante des nombres réels.

I - A - Modélisation du contexte machine notations.

Dans cette partie, nous reprendrons tout d'abord la notion d'**arrondi** développée auparavant par Kulisch et Miranker dans (49), ainsi que par Michèle Pichat dans sa thèse d'état (70). Nous présenterons ensuite quelques idées permettant d'évaluer la précision maximale atteignible lors du calcul d'une fonction sur un système à virgule flottante.

I-A-1 Arrondis d'un intervalle I de $\mathbb{R}$ dans un sous ensemble de $\mathbb{R}$.

Soit M un sous ensemble fini et non vide de $\mathbb{R}$, nous appelerons **arrondi** d'un intervalle I de $\mathbb{R}$ dans M une application $\square$ vérifiant les propriétés:

1) $\forall x \in M, \quad \square x = x$
2) $x \leq y \quad \Rightarrow \quad \square x \leq \square y$

On montre aisément que les différentes applications suivantes sont des arrondis:

Arrondi inférieur (existe si I est minoré par un élément de M)

$$\nabla x = Max \{ y \in M / y \leq x \}$$

Si M est l'ensemble des entiers naturels, cette application n'est autre que l'application **partie entière**.

Arrondi supérieur (existe si I est majoré par un élément de M)

$$\Delta x = Min \{ y \in M / y \geq x \}$$

Arrondi vers zéro

$$Zer(x) = \nabla x \text{ si } x \geq 0, \ \Delta x \text{ sinon.}$$

Arrondi vers l'infini

$$Inf(x) = \Delta x \text{ si } x \geq 0, \ \nabla x \text{ sinon.}$$

On peut remarquer que si $\square$ est un arrondi de I dans $\mathbb{R}$, alors pour tout élément x de I, $\square$ x est égal soit à ∇x, soit à Δ x.

Représentations M-correctes d'une application:

On dira que l'application φ de M^n dans M est une représentation **M-correcte** de l'application f de I^n dans I si pour tout n-uplet $V = (x_1, x_2, ..., x_n)$ de M^n, $\varphi(V)$ est inclus dans le duplet $[\nabla f(V), \Delta f(V)]$.

Cette notion recouvre, pour n=2, celle **d'opérations correctes** présentée dans (70).

Notons M^+ (resp. M^-) l'ensemble des éléments positifs ou nuls (resp. négatifs ou nuls) de M.

Pour $x \in M$, si x est distinct du plus grand élément de M (resp. si x est distinct du plus petit élément de M), notons succ (x) (resp pred (x)) le **successeur** de x (resp. le **prédecesseur** de x), c'est à dire le plus petit élément de M strictement supérieur à x (resp. le plus grand élément de M strictement inférieur à x).

Nous définirons les fonctions **epsilon-M à droite** et

epsilon-M à gauche en x respectivement par:

$$\varepsilon^{+}(x) = \Delta x - \nabla x \text{ si } x \in I \setminus M$$
$$= \text{succ}(x) - x \text{ si } x \in M.$$

$$\varepsilon^{-}(x) = \Delta x - \nabla x \text{ si } x \in I \setminus M$$
$$= x - \text{pred}(x) \text{ si } x \in M.$$

Il est à noter que si x n'est pas élément de M, ces deux fonctions sont égales à une même quantité $\varepsilon(x)$ que nous appelerons **epsilon-M en x**.

Aprés ces définition générales, nous allons nous placer dans un contexte plus restreint, destiné à l'étude de la représentation des nombres réels en virgule flottante.

I-A-2 Modélisation d'un système à virgule flottante.

définition: On appelera **nombre à virgule flottante normalisé écrit en base B sur N chiffres de mantisse, et à exposant compris entre expmin et expmax** un nombre de la forme

$$x = \pm f . B^{e}$$

où:

. e est un entier compris entre expmin et expmax.
. $1/B \leq f < 1$.
. f s'écrit en base B avec N chiffres, le zéro précédant la virgule n'étant pas compté.

e sera appelé **exposant** de x, et f sera appelé **mantisse** de x.

On appelera **Espace machine en base B avec N chiffres de mantisse et des exposants compris entre expmin et expmax** l'ensemble M(B,N,expmin,expmax) formé des nombres à virgule flottante normalisés que l'on vient d'expliciter et du nombre zéro. Cet ensemble sera noté M lorsqu'il n'y aura pas de risques de confusion.

On appelera également **Espace approximable**, noté E_a , l'ensemble formé par les nombres réels dont la valeur absolue est inférieure à B^{expmax}.

On se donnera un arrondi de E_a dans M, noté fl. La fonction epsilon-M sur M sera appelée **epsilon-machine**. Nous allons tout d'abord chercher à évaluer cette fonction epsilon-machine.

Evaluation de l'epsilon-machine.

Soit x élément de E_a différent de zéro.

Cherchons tout d'abord la valeur de l'exposant de x.

Nous avons: $\forall x = \pm f . B^e$

donc $Log_B (|\forall x|) = Log_B (f) + e$

donc $\lfloor Log_B(|\forall x|)\rfloor = \lfloor Log_B (f)\rfloor + e$

où $\lfloor u\rfloor$ désigne la partie entière de u.

Or il est clair que, puisque les puissances de B sont éléments de M, $\lfloor Log_B (|\forall x|)\rfloor$ est égal à $\lfloor Log_B (|x|)\rfloor$. On en déduit donc:

$$\lfloor Log_B (|x|)\rfloor = \lfloor Log_B (f)\rfloor + e$$

Cependant, d'après la définition des nombres à virgule flottante normalisés:

$$1/B \leq f < 1$$

par conséquent: $-1 \leq Log_B f < 0$

donc $\lfloor Log_B(f) \rfloor = -1$

donc $e = \lfloor Log_B(|x|) \rfloor + 1$

Or il est évident que la valeur de l'epsilon-machine à droite en x sera:

$$\varepsilon^+(x) = h_0 . B^e$$

avec $h_0 = B^{-N}$.

D'où la relation:

$$\varepsilon^+(x) = h_0 . B^{\lfloor Log_B |x| \rfloor + 1}$$

On pourrait de même montrer:

$$\varepsilon^-(x) = h_0 . B^{\lceil Log_B |x| \rceil}$$

où $\lceil u \rceil$ désigne le plus petit entier supérieur où égal à u

On notera MI (x) (**micro-intervalle** en x) l'ensemble des nombres ayant même "représentation machine" que x, c'est à dire la classe d'équivalence:

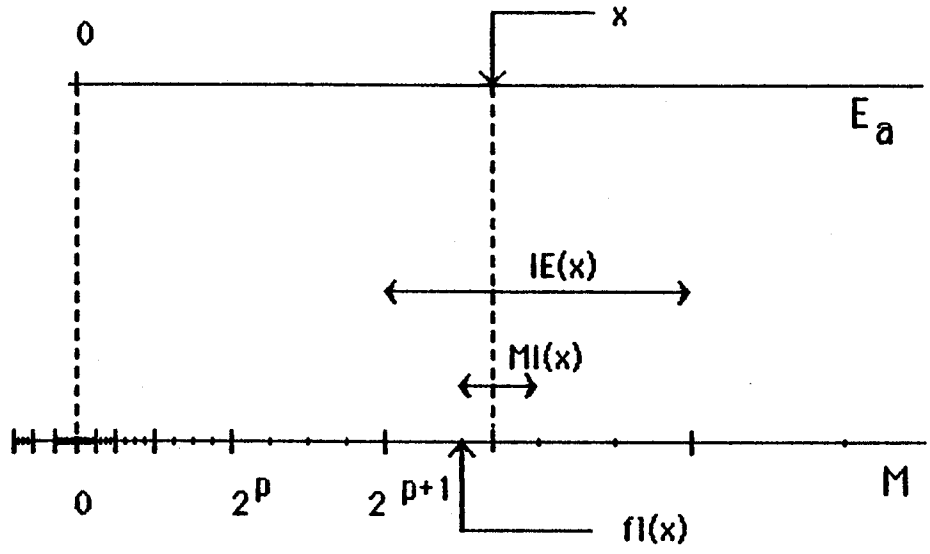
$$MI(x) = fl^{-1}(fl(x))$$

de même, on notera IE (x) (ensemble de **même exposant** que x) la classe d'équivalence:

$$IE(x) = (\varepsilon^+)^{-1}(\varepsilon^+(x))$$

On dira qu'un ensemble est un ensemble **de rupture** s'il contient au moins une puissance entière de B distincte de sa borne inférieure. Les intervalles de rupture sont ceux dans lesquels la fonction ε^+ n'est pas constante.

La figure suivante représente une partie des ensembles M et E_a, pour B=2, et $fl = \nabla$.



I - B - Problèmes de précision liés au calcul en machine d'une fonction.

Soit f une fonction définie et continue sur un intervalle I inclus dans E_a. On notera f_m la fonction de M dans M qui la représente en machine. (f_m peut être assimilée au *programme* qui calcule f).

L'idéal serait d'avoir:

$$\forall x \in I,\ f_m(fl(x)) = fl(f(x))$$

Cette condition est hélas utopique, sauf pour de rares

fonctions, car elle ne peut être réalisée dès que deux éléments d'un même micro-intervalle ont leurs images dans deux micro-intervalles différents.

En fait, dans le meilleur des cas, f_m ne peut être que M-correcte.

La condition de M-correction est une condition portant sur **l'algorithme** calculant f_m , nous allons donner à présent une autre condition, **indépendante de f_m** , qui ne dépend que de f et de B.

Voici cette condition, que nous appelerons condition de **perte d'au plus k intervalles machine**, sur $J \subset I$, tel que:

- J et f(J) ne sont pas des ensembles de rupture.
- J est ouvert.

$$(1)\ \forall x_0 \in J,\ \exists\ \alpha>0,\ \forall x \in [x_0-\alpha\ ,\ x_0+\alpha],$$
$$|f(x)-f(x_0)| \,/\, \varepsilon^+(f(x_0)) \leq k.|x-x_0| \,/\, \varepsilon^+(x_0).$$

On dira que f est **k-régulière** sur I si cette condition est vérifiée pour tous les intervalles J inclus dans I et tels que J et f(J) ne sont pas des ensembles de rupture.

En gros, si cette condition est vérifiée, alors on pourra calculer f **à k intervalles-machine près**, c'est-à-dire avec une **perte d'au plus $\lceil Log_B(k) \rceil$ chiffres significatifs de mantisse.**

Le théorème suivant donne une condition nécessaire et suffisante pour que (1) soit vérifiée.

Théorème: Soit J un intervalle ouvert inclus dans I, tel que J et f(J) ne sont pas des ensembles de rupture. Si f est continûment dérivable sur J, alors (1) est vraie si et seulement si

$$\forall x \in J,\ |f'(x)| \leq k\ .\ \varepsilon^+(f(x)) \,/\, \varepsilon^+(x)$$

Démonstration

a - La condition est suffisante.

Supposons: $\forall x \in J,\ |f'(x)| \leq k\varepsilon^+(f(x)) / \varepsilon^+(x)$
Soient x et x_0 deux éléments distincts de J.

D'après le théorème des accroissements finis, il existe ζ compris entre x et x_0 tel que:

$$|(f(x) - f(x_0))/(x - x_0)| = |f'(\zeta)|$$

C'est à dire tel que:

$$|(f(x) - f(x_0))/(x - x_0)| \leq k.\varepsilon^+(f(\zeta)) / \varepsilon^+(\zeta)$$

Or, puisque ζ est compris entre x et x_0, et que nous savons que J et f(J) ne sont pas des ensembles de rupture, nous avons:

$$\varepsilon^+(\zeta) = \varepsilon^+(x_0)$$

et $\varepsilon^+(f(\zeta)) = \varepsilon^+(f(x_0))$

Par conséquent:

$$|(f(x) - f(x_0))/(x - x_0)| \leq k\varepsilon^+(f(x_0)) / \varepsilon^+(x_0)$$

Ce qu'il fallait démontrer.

b - La condition est nécessaire.

La démonstration est triviale.

Corrollaire immédiat: une fonction continûment dérivable f est k-régulière sur I si et seulement si

$$\forall x \in I,\ |f'(x)| \leq k.\varepsilon^+(f(x)) / \varepsilon^+(x).$$

I - B - Fonctions de conditionnement.

Le théorème précédent montre que la perte de précision liée au calcul de f au voisinage de x_0, mesurée en nombre de micro-intervalles est:

$$\Psi_f(x_0) = |f'(x_0)| . \varepsilon^+(x_0) / \varepsilon^+(f(x_0)).$$

Cette fonction Ψ_f sera appelée **fonction de conditionnement de f.**

Il est aisé de vérifier:

$$\Psi_f(x) = |f'(x)| . B^{(\lfloor \log_B |x| \rfloor - \lfloor \log_B |f(x)| \rfloor)}$$

Cette fonction peut être délicate à étudier. Lorsqu'on ne désire en connaître qu'un ordre de grandeur, où lorsque l'on ne connaît pas la base dans laquelle travaille la machine utilisée, on pourra se contenter de l'approximation de Ψ:

$$\Phi_f(x) = |f'(x).x/f(x)| = \Psi_f(x) . e$$

Avec $1/B^2 \leq e \leq B^2$.

Cette fonction Φ_f est connue en économie sous le nom d'**élasticité** de f.

Le nombre de chiffres de mantisse que l'on ne pourra plus garantir lors de l'évaluation de f au voisinage de x sera:

$$\lceil \log_B(\Psi_f(x)) \rceil = \lceil \log_B(\Phi_f(x)) \rceil \pm 2.$$

Propriétés élémentaires des fonctions de conditionnement.

a - Calcul de f^{-1}.

$$\Psi_f{}^{-1}(f(x)) = 1/\Psi_f(x)$$

$$\Phi_f{}^{-1}(f(x)) = 1/\Phi_f(x)$$

En gros, ces deux relations signifient que si f se calcule "bien" au voisinage de x, alors f^{-1} se calculera "mal" au voisinage de f(x).

b - Composition d'applications.

$$\Psi_{gof}(x) = \Psi_g(f(x)) \cdot \Psi_f(x)$$

$$\Phi_{gof}(x) = \Phi_g(f(x)) \cdot \Phi_f(x).$$

Au vu de ces propriétés, on constate que Ψ et Φ se comportent un peu comme des **dérivées**. Ceci n'a rien de surprenant, car la condition du théorème précédent n'est en fait qu'une adaptation de la condition de Lipschitz à la structure particulière de E_a. On peut d'ailleurs remarquer que si les éléments de E_a étaient équidistants (c'est-à-dire si la fonction ε^+ était constante), alors la fonction de conditionnement Ψ_f de f coïnciderait avec la fonction dérivée de f.

Fonctions Φ de fonctions usuelles.

f(x)	$\Phi_f(x)$
sin x	|x cotg x|
cos x	|x tg x|
tg x	|2x / sin (2x)|
arctg x	$x / ((1 + x^2)$ arctg x)
sh x	x coth x
e^x	|x|
ln x	|1 / ln x|
x^α	|α|
k.x (k≠0)	1

En conclusion, les domaines où l'on peut - moyennant bien sûr un bon algorithme - espérer calculer f en commettant une erreur inférieure à k micro-intervalles sont les domaines :

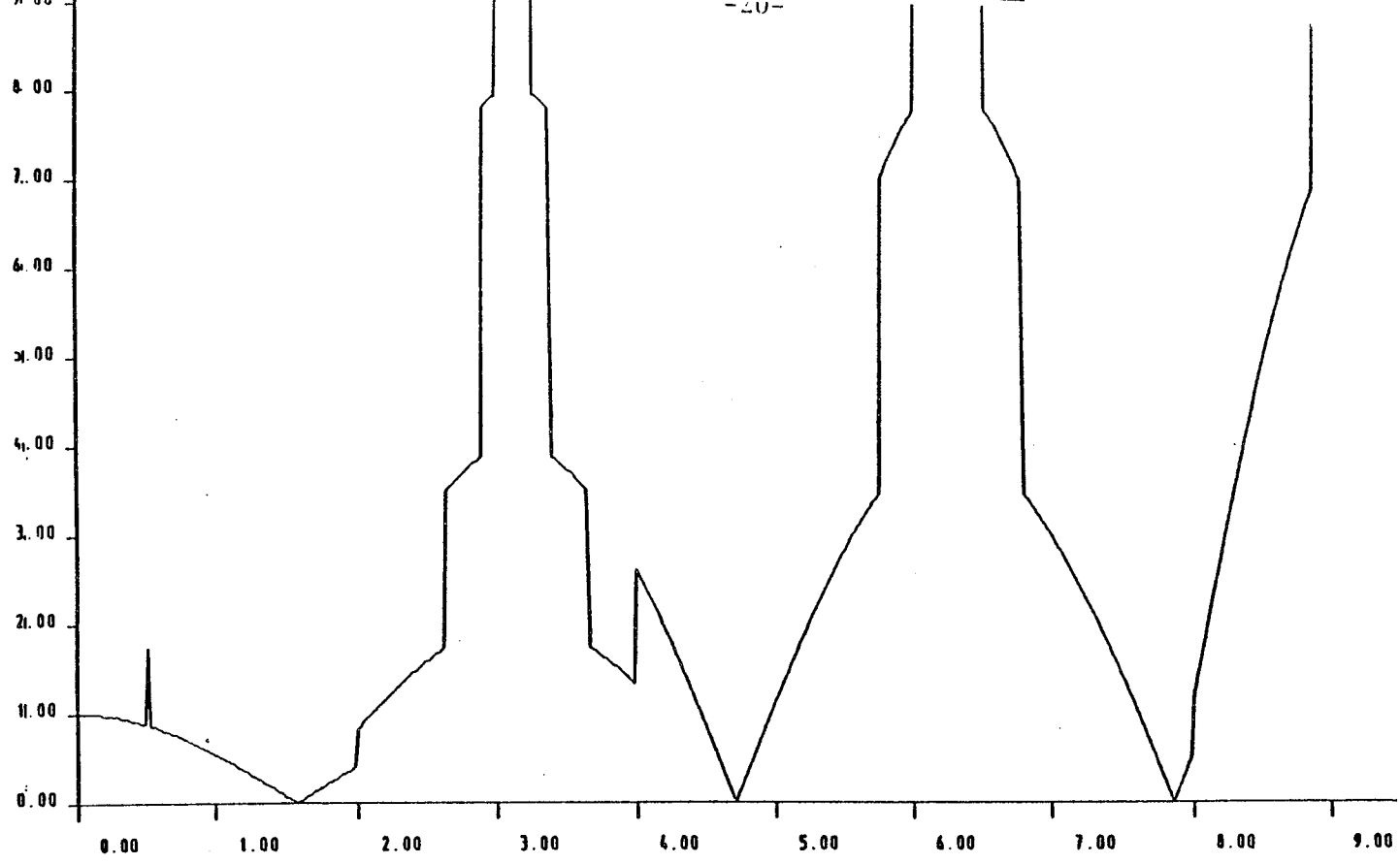
$$D_k(f) = \Psi_f^{-1}([0, k]).$$

Très difficiles à déterminer théoriquement, ils peuvent toutefois être évalués numériquement.

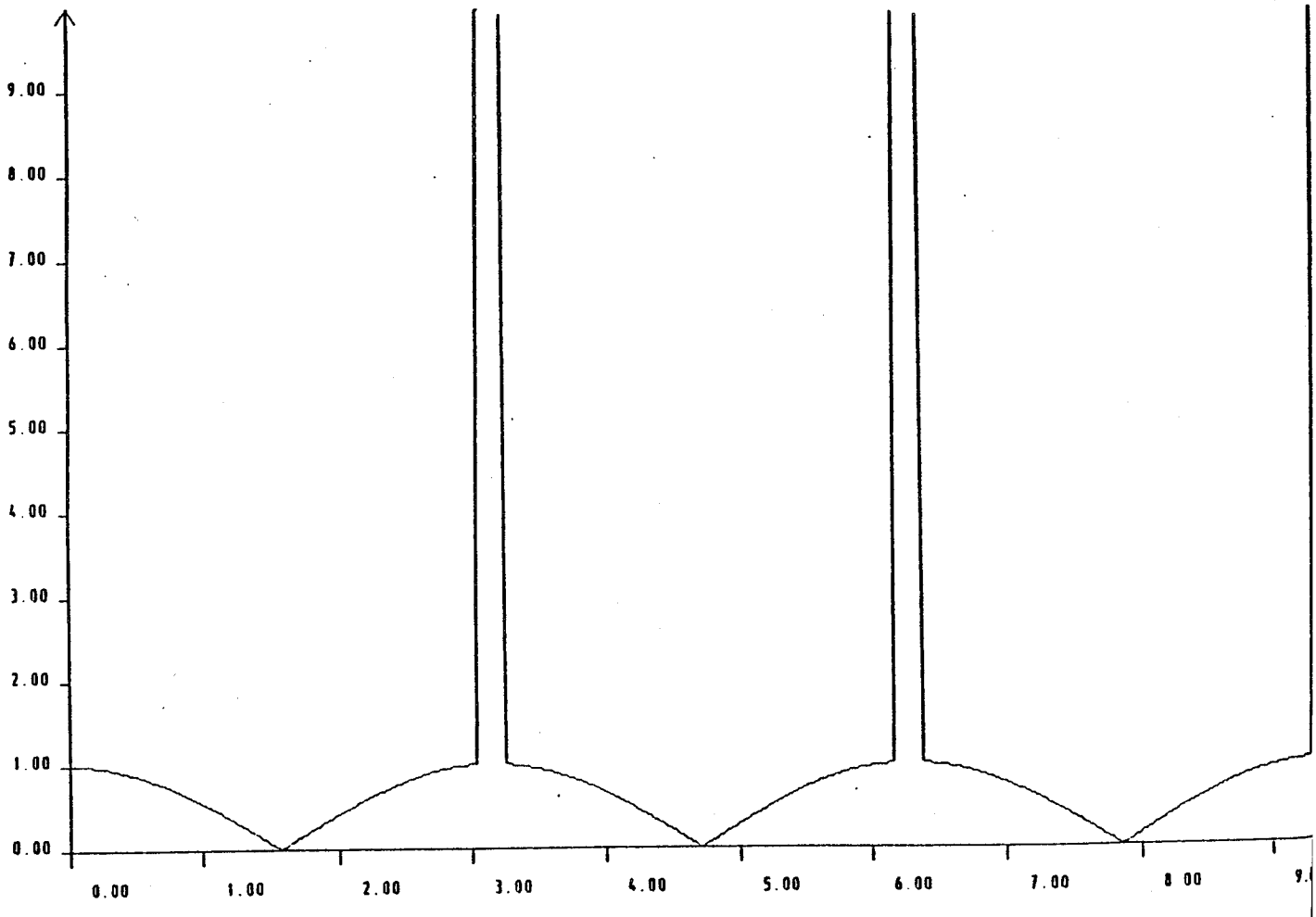
Nous possédons donc un moyen d'information important pour l'élaboration ou le test de programmes ou de circuits intégrés de calcul de fonctions. En effet, la précision que nous

serons en droit d'exiger des algorithmes sur E_a est subordonnée à la valeur que prend la fonction de conditionnement de la fonction calculée sur l'intervalle considéré.

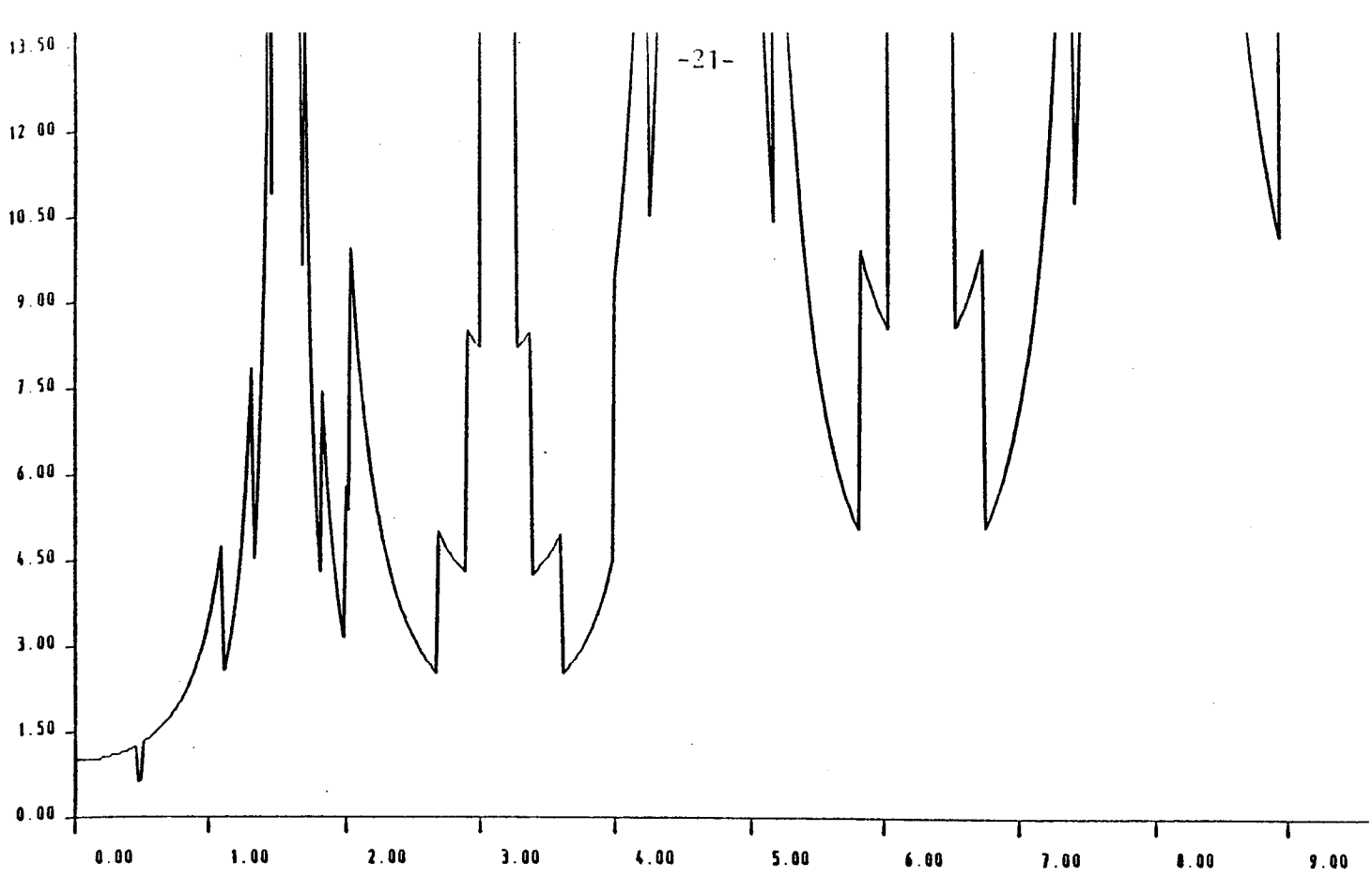
Par exemple, les courbes suivantes, qui donnent les tracés des fonctions de conditionnement de fonctions usuelles dans diverses bases de numération, prouvent que l'on peut être très exigeant vis à vis d'un algorithme de calcul de la racine carrée ou de l'arctangente, tandis qu'il est tout à fait inutile de raffiner exagérément un algorithme de calcul de la tangente.



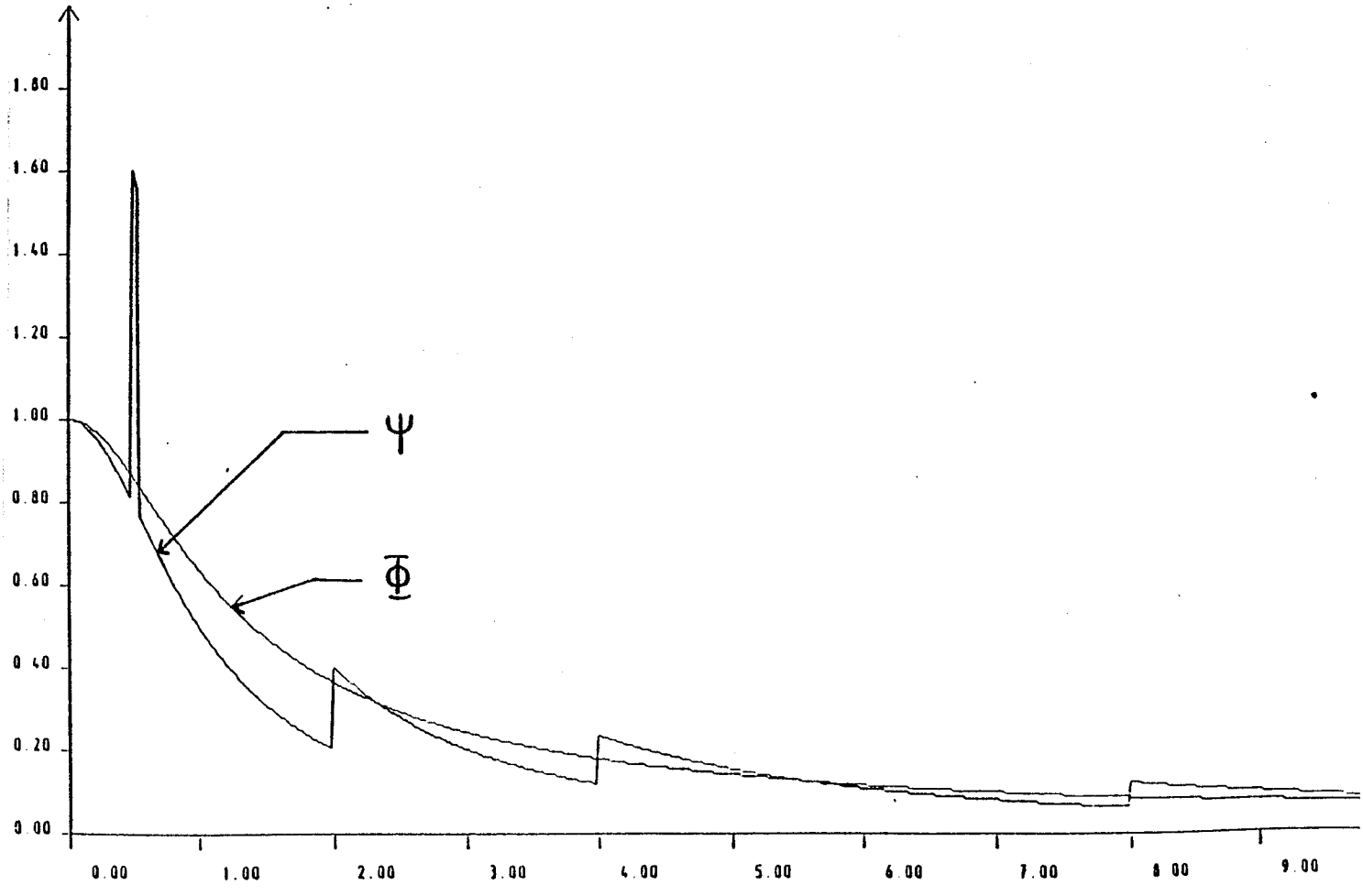
Conditionnement de f(x) = sin x BASE 2



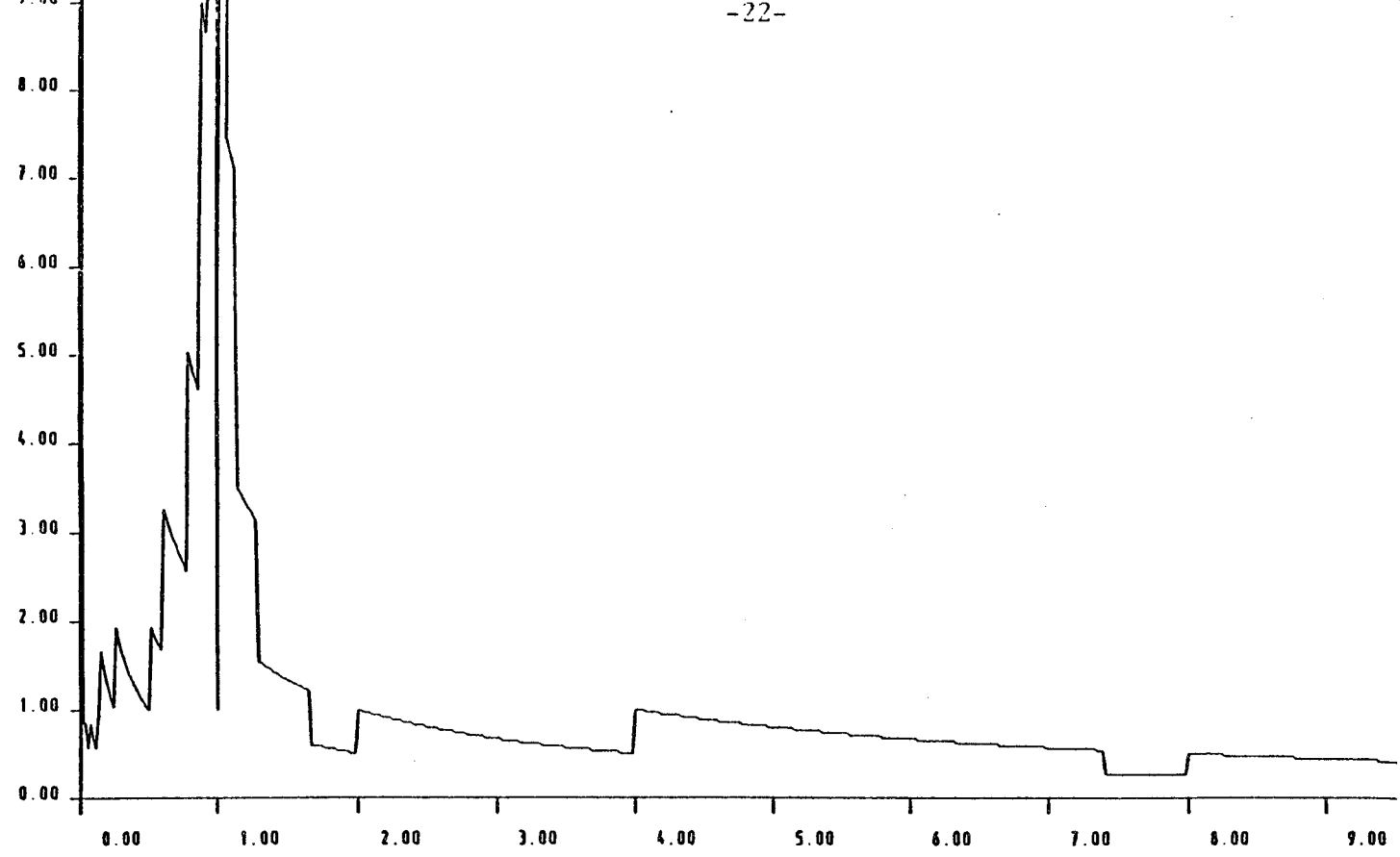
Conditionnement de f(x) = sin x BASE 10



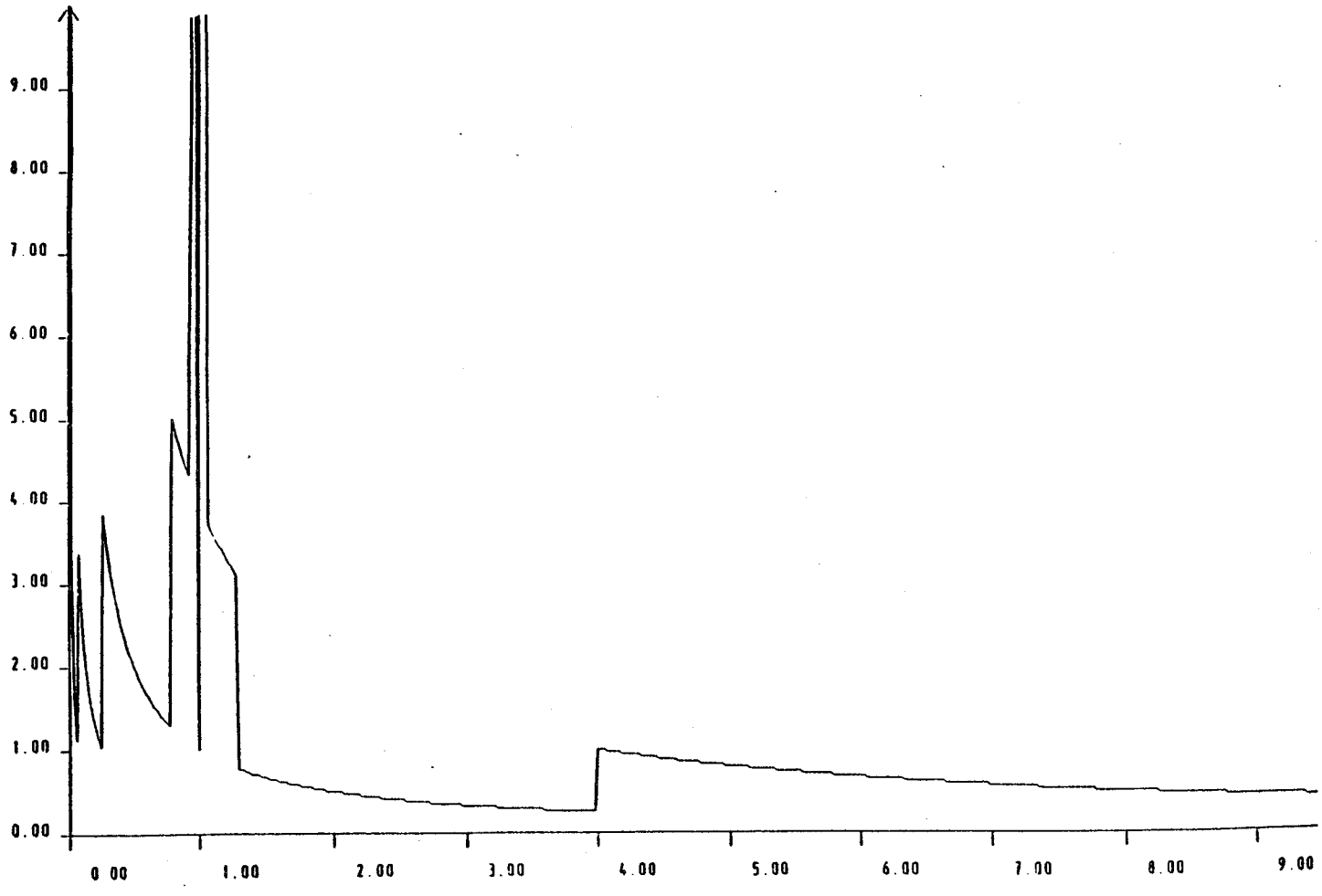
Conditionnement de f(x) = tg x BASE 2



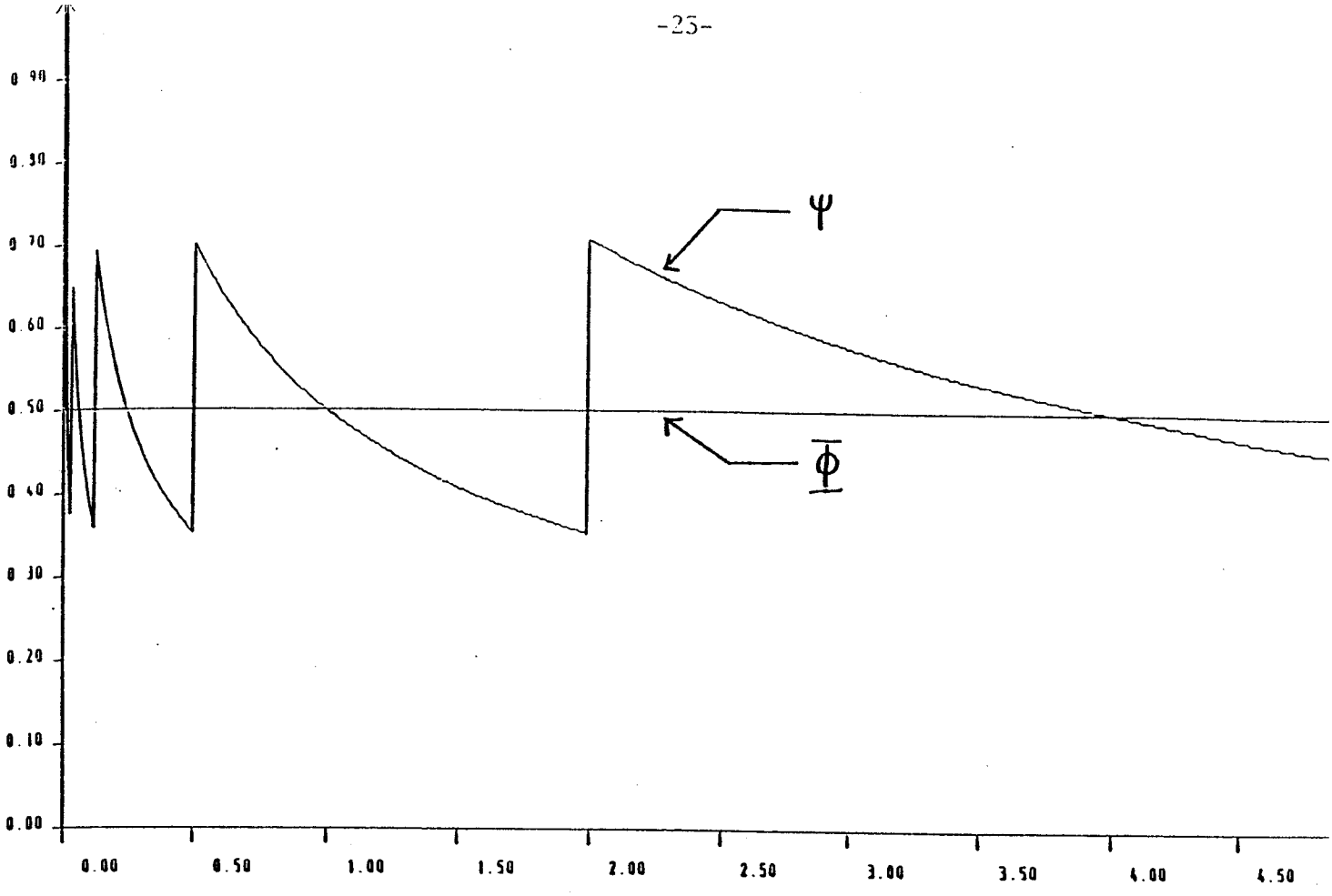
Conditionnement de f(x) = arctan x BASE 2



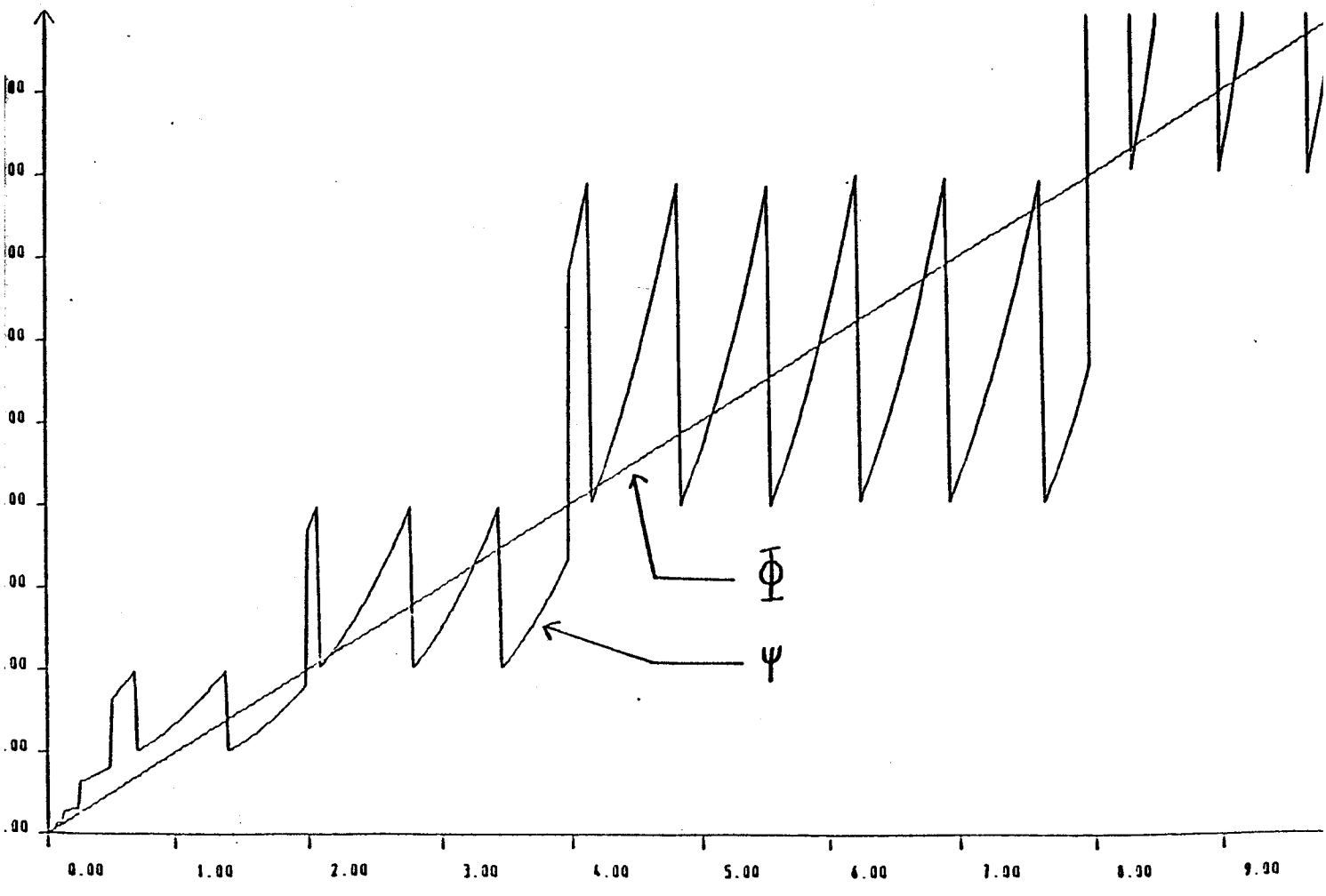
Conditionnement de f(x) = ln x BASE 2



Conditionnement de f(x) = ln x BASE 4



Conditionnement de f(x) = $\sqrt{x}$ BASE 2



Conditionnement de f(x) = exp x BASE 2

Domaines D_k de quelques fonctions usuelles pour un système binaire (B=2).

(On se limite à l'intervalle I = [0 , 4])

Fonction sinus.

D_1 = [0 , 2.09...]

D_2 = [0 , 2.61...]

D_3 = [0 , 2.61...] U [3.66... , 4]

D_4 = [0 , 2.88...] U [3.39... , 4]

Fonction exponentielle.

D_1 = [0 , 0.5]

D_2 = [0 , 2]

D_3 = [0 , 2] U [2.33... , 2.48...] U [2.77... , 3.17...] U [3.46... , 3.87...]

D_4 = [0 , 4]

Fonction logarithme népérien.

D_1 = [0 , 0.5] U [1.64... , 4]

D_2 = [0 , 0.60...] U [1.28... , 4]

D_3 = [0 , 0.77...] U [1.28... , 4]

D_4 = [0 , 0.77...] U [1.13... , 4]

Ces domaines ont été évalués en utilisant une méthode de dichotomie.

Chapitre II

Algorithmes destinés à une implémentation logicielle des fonctions élémentaires.

Introduction : L'existence de bibliothèques performantes de calcul des principales fonctions mathématiques constitue une nécessité primordiale dans le domaine du calcul scientifique. Il n'est donc guère étonnant que de nombreux auteurs aient étudié la question. En particulier, W. Cody (19) et R.P. Brent (10) ont donné de nombreux résultats à ce sujet.

Les algorithmes utilisés dans la plupart des logiciels sont basés sur des approximations **polynômiales** ou **rationnelles** des fonctions que l'on désire calculer. Dans la première partie, nous rappellerons les principales méthodes d'obtention de ces approximations.

Nous préciserons ensuite quelques problèmes inhérents à la notation des nombres en machine, puis, après avoir présenté les méthodes de tests de W. Cody (21), nous les utiliserons pour construire des programmes **tenant compte** de l'arithmétique du système utilisé. Nous illustrerons notre propos en présentatn la bibliothèque FELIMAG de calcul des fonctions élémentaires.

II - A - Méthodes classiques d'obtention d'approximations.

II - A -1 Approximations polynômiales.

Notons P_N l'ensemble des polynômes de degré inférieur ou égal à N. Pour approcher, sur l'intervalle [a , b], une fonction f par un élément p^* de P_N , deux critères d'approximation seront traités ici: l'approximation au sens des **moindres carrés** et l'approximation au sens de la norme de la **convergence uniforme**.

Dans le contexte étudié, le deuxième peut sembler plus satisfaisant, mais le premier nous permet d'obtenir plus aisément des résultats.

Approximation polynômiale au sens des moindres carrés.

Nous cherchons à obtenir :

$$\int_a^b w(x)\,(f(x) - p^*(x))^2\,dx = \min_{p \in P_N} \int_a^b w(x)\,(f(x) - p(x))^2\,dx$$

où w est une **fonction poids** positive, et telle que pour toute fonction g continue sur [a,b], w.g est intégrable sur [a,b].

La marche à suivre est la suivante :

- en appliquant le procédé d'orthogonalisation de Schmidt à la famille $(x^p)_{p \le N}$, on construit une **base orthogonale** $(T_p)_{p \le N}$ de polynômes pour le produit scalaire :

$$\langle f,g \rangle = \int_a^b w(x)f(x)g(x)\,dx$$

telle que T_p est de degré p.

- on calcule les coefficients $a_i = \langle f,T_i \rangle / \langle T_i,T_i \rangle$

- on obtient $p^* = \sum_{i=0}^{N} a_i T_i$.

En effet, soit $p_1 = \sum_{i=0}^{N} b_i T_i$ une autre approximation de f,

le produit scalaire $d^2(f,p1) = \langle f - p_1, f - p_1 \rangle$ est égal à :

$$\langle f,f \rangle - 2 \langle f , \sum_{i=0}^{N} b_i T_i \rangle + \langle \sum_{i=0}^{N} b_i T_i , \sum_{i=0}^{N} b_i T_i \rangle$$

c'est à dire à :

$$\langle f,f \rangle + \sum_{i=0}^{N} (b_i^2 - 2 a_i b_i) \langle T_i , T_i \rangle$$

Par conséquent, la quantité :

$$d^2(f,p_1) - d^2 (f , \sum_{i=0}^{N} a_i T_i)$$

vaut : $\sum_{i=0}^{N} (b_i^2 - 2 a_i b_i + a_i^2) \langle T_i , T_i \rangle$

c'est à dire : $\sum_{i=0}^{N} (b_i - a_i)^2 \langle T_i , T_i \rangle$

Cette quantité est donc positive (et même **strictement positive** s'il existe i tel que $b_i \neq a_i$). Ce qu'il fallait démontrer.

Le point crucial de cette méthode est le calcul des coefficients a_i. Ce calcul peut se faire analytiquement (avec éventuellement l'utilisation d'un système de calcul formel) si les fonctions f et w sont suffisament simples, mais si ce n'est pas le cas, il faut faire appel à un calcul numérique précis, en se servant éventuellement d'un logiciel multi-précision.

Certaines familles de polynômes orthogonaux sont connues depuis longtemps, par exemple, pour a = -1 et b = 1, on trouve :

- pour $w(x) = 1$: les polynômes de **Legendre**.
- pour $w(x) = \sqrt{(1 - x^2)}$: les polynômes de **Chebyshev**.
- pour $w(x) = (1-x)^{\alpha}(1+x)^{\beta}$: des polynômes de **Jacobi**.

Avec a = 0 et b = $+\infty$, pour $w(x) = e^{-x}$, on obtient les polynômes de **Laguerre**.

Avec a = $-\infty$ et b = $+\infty$, pour $w(x) = e^{-(x)^2}$, on obtient les polynômes de **Hermite**.

Approximation polynômiale au sens de la norme de la convergence uniforme.

Supposons que nous désirions calculer une fonction continue f sur l'intervalle [a,b]. Nous citerons sans les démontrer les deux théorèmes suivants :

Théorème d'approximation de Weierstrass : Pour tout $\varepsilon>0$, il existe un polynôme P tel que :

$$\sup_{[a,b]} |f(x) - p(x)| \leq \varepsilon.$$

Théorème d'alternance de l'erreur d'approximation :
Le polynôme P^*_N de meilleure approximation de f sur [a,b] pris parmi les polynômes de degré inférieur ou égal à N est entièrement caractérisé par l'existence de N+2 points :

$$a \leq x_0 < x_1 < x_2 < \ldots < x_{n+1} \leq b$$

vérifiant :

$$P^*_N(x_i) - f(x_i) = (-1)^i (p^*_N(x_0) - f(x_0)) = \pm \sup_{[a,b]} |p^*_N(x) - f(x)|$$

Un algorithme, dû à Remes (44) utilise cette dernière propriété pour construire une suite de polynômes convergeant vers p^*_N. D'autres méthodes existent, qui, si elles ne donnent pas forcément la **meilleure approximation** au sens de la norme de la convergence uniforme, en donnent toutefois une **bonne**.

La plus répandue d'entre elles consiste à développer f en série de **polynômes de Chebyshev**, ce qui nous ramène à l'étude précédente des approximations polynômiales au sens des moindres carrés.

II - A - 2 Approximation par des fractions rationnelles.

Le principal inconvénient des approximations polynômiales est l'existence de fonctions qui se laissent "mal" approcher, c'est à dire pour lesquelles, si l'on désire avoir une bonne précision de calcul, il faut utiliser un polynôme de degré très élevé.

Il ne faudrait d'ailleurs pas croire que de telles fonctions ne sont que des "exemples d'école" et que dans la pratique tout se passera de manière satisfaisante. Hart (44) montre par exemple que la fonction Arcsinus ne peut être approximée sur [-1 , +1] avec une erreur inférieure à 10^{-8} que par un polynôme de degré 10000 !

On ne peut d'ailleurs guère avoir idée a priori du degré necessaire des polynômes employés : tout dépend de la fonction étudiée, comme le montre le théorème suivant.

Théorème de Bernstein : Pour toute suite décroissante (ε_n) de réels strictement positifs convergeant vers zéro, il existe une fonction continue f telle que la meilleure approximation polynômiale uniforme de degré n p^*_n de f sur [a,b] vérifie :

$$\sup_{[a,b]} |p^*_n(x) - f(x)| = \varepsilon_n.$$

Pour évaluer des fonctions qui se laissent "mal" approcher par des polynômes, la seule ressource est **d'utiliser des approximations rationnelles** : en effet, les opérations arithmétiques dont nous disposons sur ordinateur étant l'addition, la soustraction, la multiplication et la division, l'ensemble des fonctions d'une variable réelle que l'on peut calculer est égal à l'ensemble des fractions rationnelles. On ne peut hélas plus construire de méthodes semblables à celles utilisant des polynômes orthogonaux, même si, d'après le théorème suivant, dû à Chebyshev, on sait caractériser théoriquement les meilleures approximations uniformes rationnelles. Ce théorème est l'équivalent pour les fractions rationnelles du théorème d'alternance de l'erreur d'approximation rencontré auparavant.

Théorème de Chebyshev : La meilleure approximation $R = P/Q$ de f sur $[a,b]$, prise parmi les fractions rationnelles irréductibles dont le numérateur est de degré inférieur ou égal à p et le dénominateur de degré inférieur ou égal à q, est entièrement caractérisée par l'existence de

$N = 2 + \text{Max}\,(p + \text{degré}\,(P), q + \text{degré}\,(Q))$ points :

$$a \leq x_0 < x_1 < \ldots < x_{N-1} \leq b$$

tels que :

$$R(x_i) - f(x_i) = (-1)^i (R(x_0) - f(x_0)) = \pm \sup_{[a,b]} |R(x) - f(x)|$$

En pratique, même si on ne dispose pas d'algorithmes permettant de trouver la **meilleure approximation** de f, on sait en trouver de **bonnes**, par exemple en "retravaillant" des **approximants de Padé**.

En utilisant les méthodes que nous venons d'explorer rapidement, nous avons oublié un facteur important : nous travaillons sur un ordinateur calculant avec un nombre **fini** de chiffres de mantisse, disposant d'opérations arithmétiques **qui ne sont pas exactes** et utilisant des modes d'arrondi qui peuvent varier.

Certes, si l'on ne désire pas une très grande précision, on pourra se contenter d'arrondir les coefficients des polynômes ou des fractions rationnelles obtenus au nombre de chiffres disponibles. Mais si l'on désire obtenir des résultats avec une précision **proche de la précision machine**, on ne peut se contenter d'une telle politique, car le meilleur approximant d'une fonction pris parmi les "polynômes machine" ou les "fractions rationnelles machine" n'est pas forcément obtenu en arrondissant à la précision machine prés le meilleur approximant théorique.

Là encore, pour notre recherche d'un "approximant machine", il est hors de question de trouver un **meilleur approximant**, car ce problème est bien plus difficile à résoudre que les précédents, on se contentera donc de trouver une **bonne approximation**, et ceci de deux manières possibles :

1) En construisant une suite d'approximants, le premier terme

de la suite n'étant autre que l'arrondi de l'approximant théorique (voire même un développement en série tronqué ou un approximant de Padé), et chaque itéré étant "meilleur" que les précédents. Cette méthode implique que nous soyons en mesure de **décider** qu'un approximant est meilleur qu'un autre, ce que nous pourrons faire après le paragraphe consacré au **test** de logiciels. Ce principe sera appliqué par notre programme **Bestpol** d'amélioration d'une approximation polynômiale, que nous étudierons ultérieurement, et qui pourrait aisément être adapté à l'amélioration d'une fraction rationnelle.

2) En "trichant" pour simuler une précision supérieure à la précision machine : c'est ce que nous appellerons la méthode de "rattrappage de l'erreur de troncature". En voici un exemple tiré de la bibliothèque FELIMAG :

Pour calculer la fonction sinus, on part d'une approximation polynômiale donnée par Cody et Waite (19) et dont les coefficients de degrés 1 et 0 sont respectivement :

$$8{,}3333333333331650 \cdot 10^{-3} \text{ et } 1{,}66666666666666665 \cdot 10^{-1}.$$

Plutôt que de les tronquer à 15 décimales (ce programme est destiné à être utilisé en double précision sur un MICRAL 90.50), on choisit de les arrondir à un nombre **exactement représentable en binaire** (pour supprimer l'erreur due à la traduction décimal/binaire et à la troncature qui s'en suit), ce qui nous donne les constantes $8{,}30078125 \cdot 10^{-3}$ et $1{,}66015625 \cdot 10^{-1}$, puis de calculer la différence entre ces nouveaux coefficients et les coefficients originaux, et ensuite, lors du calcul du polynôme (en employant le schéma de Horner), d'effectuer :

```
... + 8.30078125E-3) * x - 1.66015625E-1)
  + 3.255208333331650E-5 * x
  - 6.51041666666665E-4 ;
```

En effectuant ceci, on réduit très nettement l'erreur due à la conversion décimal/binaire, et en fait on **simule** une machine travaillant sur quelques bits de plus (étant donné qu'après la réduction d'argument x est faible, il suffit d'appliquer cette méthode aux coefficients de degrés 0 et 1).

Cette méthode repose sur une assise théorique assez faible,

mais en pratique elle donne d'excellents résultats, et ce au prix d'une perte de temps négligeable.

Nous commencerons tout d'abord par exposer les problèmes inhérents au **test** d'un logiciel de calcul des fonctions élémentaires, ce qui nous permettra ensuite de présenter une approche particulière de la conception d'algorithmes d'approximation d'une fonction.

II - B - Problèmes liés au test.

Comme le souligne W. Cody (21), le test d'un programme destiné à calculer une fonction que l'on désire évaluer avec une précision proche de la précision machine est problématique.

En effet, il est hors de question d'effectuer une comparaison avec des valeurs prises dans une table, sauf peut-être avec une machine travaillant en base 10 (ce qui est rare si l'on excepte le cas des calculatrices de poche et de certains micro-ordinateurs "personnels"). Ceci est dû à une constatation simple : la conversion décimal/binaire apparaissant lors de la lecture des coefficients de la table entraîne une erreur d'arrondi ou de troncature de l'ordre de la précision machine, ce qui peut fausser le test si l'implémentation de la fonction est très précise, c'est à dire si l'erreur d'approximation est du même ordre de grandeur.

Lorsque l'on désire tester un logiciel destiné à un calcul en **simple précision**, on peut à la rigueur le comparer à un logiciel **double précision**, mais ce procédé devient irréalisable si l'on désire utiliser la précision maximale que peut fournir le système, à moins d'utiliser un logiciel multiprécision, ce qui est fastidieux et coûteux.

W. Cody (21) conclut que le seul moyen acceptable pour tester une fonction consiste à utiliser une **identité** vérifiée par cette fonction. Il convient toutefois de se méfier car cette identité peut être utilisée par le programme qui calcule cette fonction : il est fréquent, lorsque l'on désire tester une calculatrice, de vérifier pour plusieurs valeurs de x si l'on a bien :

$$(1)\ \mathrm{Sin}^2 x + \mathrm{Cos}^2 x = 1$$

Sans se douter que, peut-être, la machine n'évalue directement que la fonction sinus, le cosinus étant obtenu par :

$$\mathrm{Cos}\ x = \pm \sqrt{(1 - \mathrm{Sin}^2 x)}$$

L'utilisation de (1) pour le test porterait alors à croire - pour peu que la racine carrée soit bien implémentée - que les fonctions trigonométriques de la calculatrice sont excellentes, même si ce

n'est pas le cas.

L'identité utilisée pour effectuer le test doit donc vérifier trois propriétés :

1) On doit pouvoir être certain qu'elle n'est pas utilisable par un logiciel de calcul, ce qui écarte toutes les identités permettant de déduire une fonction d'une autre.

2) Sa vérification ne doit pas entrainer d'erreur.

3) Elle doit apporter non seulement une information qualitative mais aussi une information quantitative : elle doit en fait fournir une mesure de l'erreur commise lors de l'évaluation de la fonction, de manière à ce que l'on puisse comparer des logiciels.

Ce dernier aspect est fondamental : il nous permettra ultérieurement d'écrire un programme d'amélioration automatique d'une approximation polynômiale.

Par exemple, pour testerla fonction sinus, Cody propose d'utiliser l'identité :

$$\text{Sin } x = 3 \text{ Sin } (x/3) - 4 \text{ Sin}^3 (x/3)$$

en montrant que si x est dans un intervalle de la forme $[3n\,\pi\,,\,(3n + 1/2)\,\pi]$, alors la quantité :

$$E = |(\text{Sin } x - 3 \text{ Sin } (x/3) + 4 \text{ Sin}^3 (x/3)) / \text{Sin } x|$$

est une bonne valeur approchée de l'erreur relative commise sur le calcul de la fonction sinus en x. Le point 3) est donc satisfait. Pour que le point 2) le soit aussi, il faut que x et x/3 puissent être tous deux exactement représentables en machine; pour ceci, Cody propose, à partir d'un réel x généré de manière pseudo-aléatoire, d'effecuer :

```
y := x / 3.0 ;
y := (x + y) - x ;
x := 3.0 * y ;
```

La deuxième ligne est destinée à placer un zéro du coté des chiffres de poids faibles de la mantisse de y. Après exécution de

cette séquence d'instruction, x est peut-être très légèrement différent de sa valeur initiale, mais on a **exactement** x = 3y.

Le programme suivant teste la fonction sinus, et est obtenu à partir de l'algorithme de Cody.

Pour tester les autres fonctions élémentaires, on se sert d'identités similaires. Par exemple, le logarithme est testé en utilisant la relation :

$$\ln(K.x) = \ln(K) + \ln(x) \quad (2)$$

où K est une constante dont le logarithme est connu avec une grande précision, et qui s'écrit sur **un petit nombre de chiffres dans la base de numération du système**, de sorte que pour tout réel x, le produit K.x puisse s'évaluer avec une perte minimale de précision. En outre, il ne faut pas que K soit entier, car dans ce dernier cas, la relation (2) pourrait être utilisée lors de la réduction d'argument.

Sur une machine binaire, les programmes de tests que nous avons utilisés prennent K = 17/16.

Les figures suivantes donnent les résultats des tests des fonctions sin, cos et ln du logiciel TRANCEND (logiciel d'origine du MICRAL 90.50), de notre logiciel FELIMAG (implémenté sur MICRAL) et du logiciel Pascal MULTICS.

Il est à noter que l'exécution du programme de test donné ici requiert la connaissance de la **base de numération** dans laquelle travaille la machine utilisée. Le lecteur se persuadera aisément qu'elle peut être obtenue à l'aide de la fonction PASCAL qui suit :

```
function Base : integer ;
    Var A,B : real ;
    Begin
        A := 1.0 ;
        while ((A + 1.0) - A) - 1.0 = 0
            do A := 2.0 * A ;
        B := 1.0 ;
        while ((A + B) - A) - B <> 0 do B := B + 1.0 ;
        Base := round (B)
    End ;
```

De même, le nombre de digits de mantisse de la représentation

```
program codysin(input,output);

(* test des fonctions sinus et cosinus d'apres l'algorithme de
    W. Cody, en tenant compte des caracteristiques arithmetiques
    du systeme *)

var super,egal,infer,ind1,ind2,exposant,taille,ibeta,n:integer;
    logbeta,c,pas,errmax,errmoy,errcour,x,xl,xi,val2,decalage,a,b,beta,y,val1:re
al;


external function hasard: real ;

   (* hasard est une fonction de bibliotheque
      qui doit fournir des reels pseudo-aleatoires
      uniformement repartis sur [0,1]              *)

external function intpower (x: real ; n: integer): real ;

  (* intpower est une fonction de FELIMAG
     qui calcule avec une grande precision
     x puissance n *)

begin
  beta:=2.0;
  writeln ('Nombre de chiffres de mantisse dans la representation flottante ?');
  read (taille);
  writeln ('NOMBRE DE TIRAGES ALEATOIRES ?');
  read (n);
  ibeta:=round (beta);
  logbeta:=ln(beta);
  a:=0.0;
  b:=1.57079632679489661;c:=b;
  for ind1:=1 to 3 do
    begin
      super:=0;infer:=0;
      x1:=0.0;errmax:=0.0;errmoy:=0.0;
      pas:=(b-a)/n;
      xl:=a;
       for ind2:=1 to n do
          begin
            x:=pas*hasard+xl;
            y:=x/3.0;
            y:=(x+y)-x;
            x:=3.0*y;
            if ind1<>3 then
              begin
                val1:=sin (x);val2:=sin (y);
                errcour:=1.0;
                if val1<>0.0 then errcour:=(val1-val2*(3.0-4.0*val2*val2))/val1;
              end
               else
                begin
                  val1:=cos(x);
                  val2:=cos (y);
                  errcour:=1.0;
                  if val1<>0.0 then errcour:=(val1+val2*(3.0-4.0*val2*val2))/val
1;
                end;
            if errcour>0.0 then super:=super+1
```

```
            else if errcour<0.0 then infer:=infer+1;
           errcour:=abs(errcour);
           if errcour>errmax then
            begin
             errmax:=errcour;
             x1:=x;
            end;
            errcour := errcour * 1.0e20 ;
             errmoy := errmoy + errcour*errcour ;
           x1:=x1+pas;
       end;
   egal:=n-infer-super;
   errmoy:=sqrt (errmoy/n)/1.0e20 ;
   writeln;
 if ind1<>3 then
     begin
       writeln ('TEST DU SINUS VIA 3*SIN(X/3)-4*SIN(X/3)**3 ');
       writeln;
       writeln ('TESTS ALEATOIRES PORTANT SUR ',n,'ECHANTILLONS');
       writeln ('DE L''INTERVALLE [',a,' , ',b,']');
       writeln;
       writeln (' le sinus a ete trop grand ',super,' fois');
       writeln ('                correct    ',egal,' fois');
       writeln ('                trop petit ',infer,' fois');
     end
      else
     begin
       writeln ('TEST DU COSINUS VIA 4*COS(X/3)**3-3*COS(X/3) ');
       writeln;
       writeln ('TESTS ALEATOIRES PORTANT SUR ',n,'ECHANTILLONS');
       writeln ('DE L''INTERVALLE [',a,' , ',b,']');
       writeln;
       writeln (' le cosinus a ete trop grand ',super,' fois');
       writeln ('                correct    ',egal,' fois');
       writeln ('                trop petit ',infer,' fois');
     end;
  writeln;
  errcour:=-999.0;
  if (errmax<>0.0) then errcour:=ln (abs(errmax))/logbeta;
  writeln ('QUALITE DE L''APPROXIMATION AU SENS UNIFORME:');
  writeln ('    erreur relative maximale: ',errmax,' = ',ibeta,'**',errcour);
  writeln ('    atteinte au point: ',x1);
  if taille+errcour>0.0 then errcour:=taille+errcour else errcour:=0.0;
  writeln('    perte de precision en chiffres de base ',ibeta:2,': ',errcour);
  errcour:=-999.0;
  if errmoy<>0.0 then errcour:=ln (abs(errmoy))/logbeta;
  writeln;
  writeln ('QUALITE DE L''APPROXIMATION AU SENS DES MOINDRES CARRES:');
  writeln('    erreur relative: ',errmoy,' = ',ibeta,'**',errcour);
  if taille + errcour>0.0 then errcour:=taille+errcour else errcour:=0.0;
  writeln ('    perte de precision en chiffres de base ',ibeta:2,':',errcour);
  a:=18.8495559215387594;
  if ind1=2 then a:=b+c;
  b:=a+c;
end;
end.
```

TEST DE LA FONCTION SINUS
Sur l'intervalle

[6.PI , 6.5 PI]

(tous les systèmes étudiés travaillent en base 2)

LOGICIEL	TRANCEND	FELIMAG	MULTICS
Err. relative Max.	$5.9\ 10^{-4}$	$3.4\ 10^{-16}$	$3.2\ 10^{-14}$
Nombre max de bits perdus	41	1	25
Err. relative Moyenne	$1.0\ 10^{-4}$	$1.2\ 10^{-16}$	$1.8\ 10^{-17}$
Nombre moyen de bits perdus	38.7	0.0	18.7

TEST DE LA FONCTION SINUS
Sur l'intervalle

[0 , PI/2]

(tous les systèmes étudiés travaillent en base 2)

LOGICIEL	TRANCEND	FELIMAG	MULTICS
Err. relative Max.	$5.9\ 10^{-4}$	$3.4\ 10^{-16}$	$4.7\ 10^{-19}$
Nombre max. de bits perdus	41	1	9
Err. relative Moyenne	$1.0\ 10^{-4}$	$1.0\ 10^{-16}$	$1.47\ 10^{-19}$
Nombre moyen de bits perdus	38.7	0.0	8.4

TEST DE LA FONCTION COSINUS
Sur l'intervalle

[7 PI , 7.5 PI]

(tous les systèmes étudiés travaillent en base 2)

LOGICIEL	TRANCEND	FELIMAG	MULTICS
Err. relative Max.	$1.6\ 10^{-6}$	$3.3\ 10^{-16}$	$8.6\ 10^{-15}$
Nombre max. de bits perdus	33	1	23
Err. relative Moyenne	$3.6\ 10^{-8}$	$1.2\ 10^{-16}$	$1.5\ 10^{-16}$
Nombre moyen de bits perdus	27	0.0	17.4

TEST DE LA FONCTION LOGARITHME NEPERIEN
Sur l'intervalle

$[\ 1-\epsilon\ ,\ 1+\epsilon\ ]\quad \epsilon = 2^{-17}$

(tous les systèmes étudiés travaillent en base 2)

LOGICIEL	TRANCEND	FELIMAG	MULTICS
Err. relative Max.	39.2	$1.7\ 10^{-15}$	$6.8\ 10^{-11}$
Nombre max. de bits perdus	58	4	36
Err. relative Moyenne	$8.8\ 10^{-1}$	$1.3\ 10^{-15}$	$1.2\ 10^{-12}$
Nombre moyen de bits perdus	52	3.5	30.5

TEST DE LA FONCTION LOGARITHME NEPERIEN
Sur l'intervalle

[16 , 240]

(tous les systèmes étudiés travaillent en base 2)

LOGICIEL	TRANCEND	FELIMAG	MULTICS
Err. relative Max.	$1.6\ 10^{-8}$	$2.2\ 10^{-16}$	$2.2\ 10^{-19}$
Nombre max. de bits perdus	27	1	8
Err. relative Moyenne	$5.8\ 10^{-9}$	$3.4\ 10^{-17}$	$9.8\ 10^{-20}$
Nombre moyen de bits perdus	25.6	0.0	6.9

en virgule flottante des nombres réels dans le système utilisé peut être obtenu grâce à la fonction :

```
Function taille_mantisse : integer ;
    Var i,b : integer ;
        k : real ;
    Begin
        i := 1 ; b := base ; k := b ;
        while ((1.0 + k) - k) - 1.0 = 0 do
            begin
                i := i + 1 ;
                k := k * b
            end ;
        taille_mantisse := i
    End ;
```

(Pour plus de détails sur ce type d'algorithmes, voir (19) et (40)).

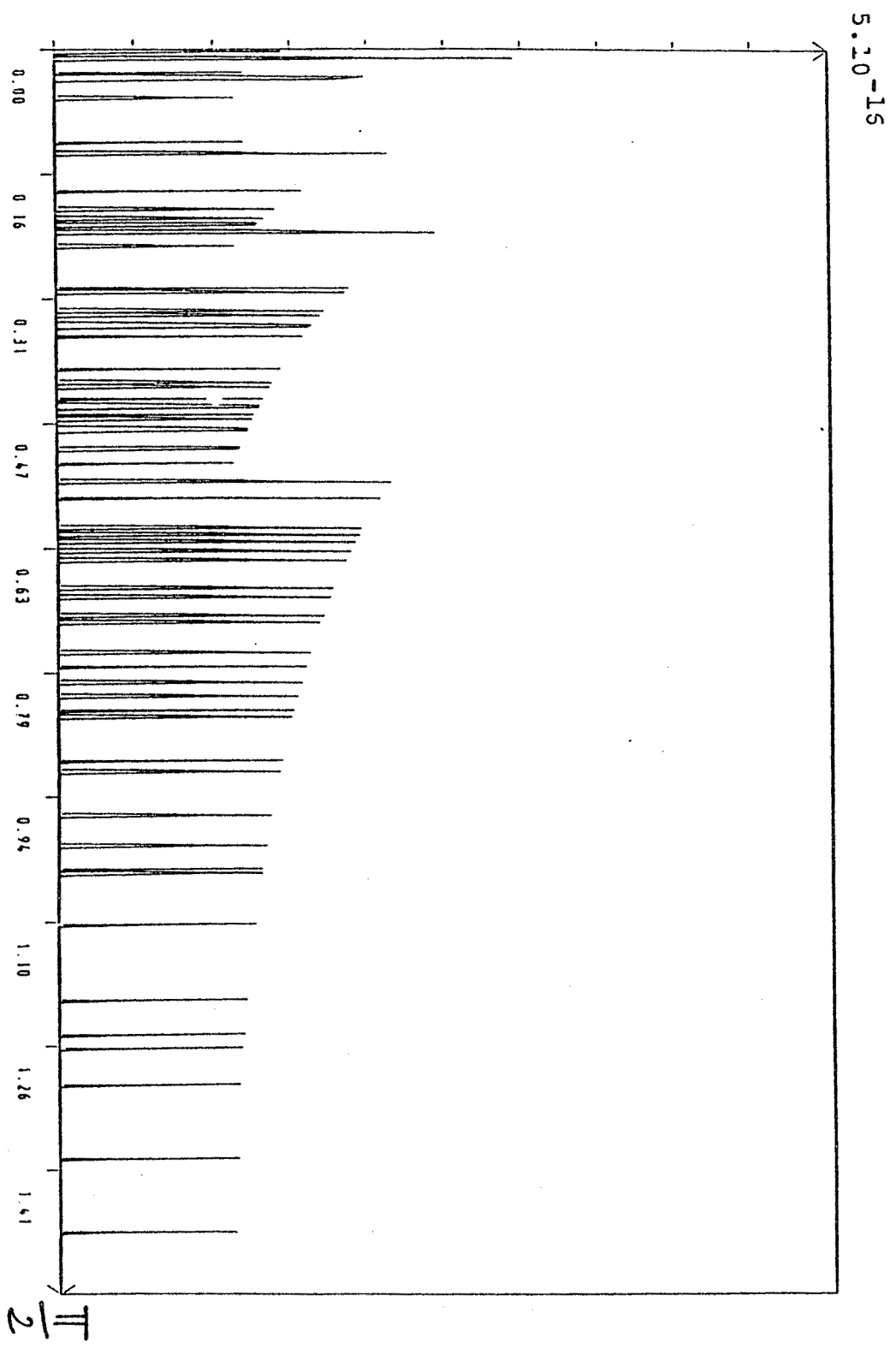
Nous donnons ci-après la "courbe d'erreur" (c'est à dire la valeur E vue auparavant) de la fonction sinus obtenue avec les logiciels FELIMAG et TRANCEND sur MICRAL 90.50.

II - C Un algorithme d'amélioration d'une approximation polynômiale.

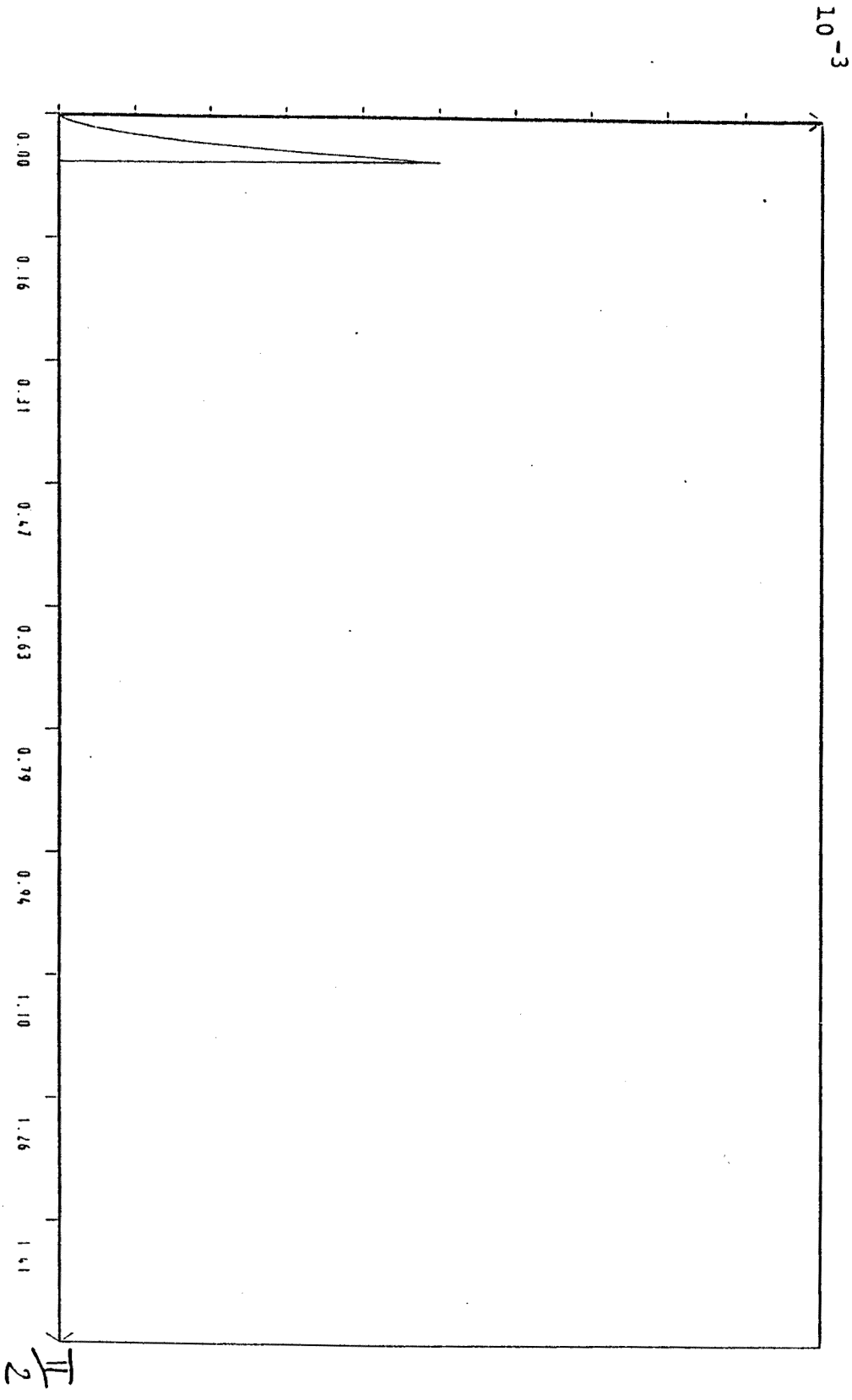
Dans ce paragraphe, notre propos est simple : il s'agit de trouver une bonne approximation polynômiale d'une fonction, tenant compte du "contexte-machine" , en partant d'une approximation théorique donnée par une méthode exposée au paragraphe II - A.

Nous procèderons en construisant une suite de polynômes d'approximation, dont les coefficients sont exactement représentables en machine, et tels que chaque terme de la suite est "meilleur" que les précédents. La notion d'approximant "meilleur" qu'un autre peut maintenant revêtir un sens si l'on connaît une identité vérifiée par la fonction qui satisfait aux trois exigences présentées dans le paragraphe consacré au test.

En effet, si l'on sait **mesurer** l'erreur commise lors du calcul de f(x), il suffit de calculer la **moyenne** de cette erreur sur l'intervalle de test pour avoir une bonne appréciation de la qualité



COURBE D'ERREUR DE LA FONCTION SINUS Logiciel FELIMAG ymax = 5.0e-16



COURBE D'ERREUR DE LA FONCTION SINUS Logiciel TRANCEND ymax = 1.0e-3

de l'approximation.

Notre algorithme utilisera une méthode de "relaxation" : si les (a_j) sont les coefficients du polynôme (ou de la fraction rationnelle), et si $M(a_1 , a_2 , \dots , a_N)$ représente l'estimation de l'erreur sur le calcul de f en employant les coefficients a_j, alors à l'étape i de l'algorithme, on cherchera à remplacer a_i par a_i^*, vérifiant :

$$M(a_1 , a_2 , \dots , a_{i-1} , a_i^* , a_{i+1} , \dots , a_N)$$

$$< M(a_1 , a_2 , \dots , a_{i-1} , a_i , a_{i+1} , \dots , a_N)$$

et on répètera plusieurs fois la boucle entière (i allant de 1 à N).

Le choix de a_i^* se fera en écrivant :

$$a_i^* = a_i \cdot \prod_{j=0}^{P} (1 + 2^{-j})^{d_j}$$

où P est le nombre de digits de mantisse disponibles (sur un système binaire), ou ce nombre multiplié par Log (B) / Log (2) sur un système en base B. On cherchera pour chaque j (de 1 à P) la valeur de d_j qui minimise :

$$M(a_1 , \dots , a_{i*} \prod_{k=0}^{j} (1 + 2^{-k})^{d_k} , \dots , a_N)$$

Nous présentons ici le programme **Bestpol** qui utilise cet algorithme pour améliorer des **approximations polynômiales**. Une petite modification permettrait d'écrire un programme similaire pour les approximations rationnelles.

La fonction **mesure** utilisée dans ce programme est la fonction $M(a_1 , \dots , a_N)$ de l'algorithme. Les résultats sont stockés dans deux fichiers PASCAL, l'un de type TEXT, l'autre de type FILE OF REAL.

Nous donnons ensuite une trace d'éxécution de ce programme.

```
program bestpol;

(* amelioration d'une approximation polynomiale *)


 var coeffin,coeffout:text;
     outbin:file of real;
     coeff:array [0..20] of real;
     degre,nombval,i,j,k,kmin,expmin:integer;
     x,y,sigma,oldsigma,tampon,a,b,diam:real;
     epsilon: array [0..50] of real;
     fini:boolean;
     reponse:char;

     external function hasard:real;
     external function mesure (x:real):real;

procedure initialise;
 var i:integer;
  begin
    writeln ('degre du polynome?');
    read (degre);
    for i:=0 to degre do read (coeffin,coeff[i]);
    writeln ('bornes de l''intervalle de tests');
    read (a,b);
    diam:=b-a;
    epsilon [0]:=1.0;
    for i:=1 to 50 do epsilon [i]:=(epsilon[i-1]/2.0);
    writeln ('terme a partir duquel on s''autorise a modifier les coefficients?'
);
    read (kmin);
    writeln ('puissance de 2 a partir de laquelle on modifie?');
    read (expmin);
    writeln ('nombre de valeurs testees?');
    read (nombval);
    writeln;
    fini:=false;
  end;

function poly (x:real):real;

  var r:real;
      n:integer;

begin
  r := coeff [degre] ;
  for n:= (degre - 1) downto 0 do r := r * x + coeff [n] ;
  poly := r
end ;

procedure stat;
var x:real;
    i:integer;
 begin
  sigma:=0.0;
  for i:=1 to nombval  do
    begin
      x:=a+diam*hasard;
      sigma:=sigma+sqr(mesure (x));
    end;
```

```
  sigma:=sqrt (sigma/nombval);
 end;


begin
 assign (coeffin,'b:coeff.in');
 reset (coeffin);
 assign (coeffout,'b:coeff.out');
 rewrite (coeffout);
 assign (outbin,'b:bin.out');
 rewrite (outbin);
 initialise;
 stat;
 k:=1;
 oldsigma:=sigma;
 while not fini do
  begin
   writeln ('ITERATION NUMERO ',k);
   for j:=kmin to degre do if coeff[j] <> 0.0 then
       begin
         i:=expmin;
         writeln ('je travaille sur le coeff. ',j:3,' iteration ',k:3);
         writeln ('valeur actuelle de sigma: ',oldsigma);
         while i<=50 do
           begin
             tampon:=coeff[j];
             coeff [j]:=coeff [j]+ epsilon [i]*coeff[j];
             stat;
             if sigma <oldsigma then oldsigma:=sigma
              else begin
                     coeff [j]:=tampon/(1.0+epsilon[i]);
                     stat;
                     if sigma <oldsigma then oldsigma:=sigma
                      else coeff [j]:=tampon;
                   end;
              i:=i+1;
           end;
      end;
    writeln ('coefficients actuels du polynome: ');
    for j:=0 to degre do writeln ('degre ',j:3,':',coeff[j]);
    writeln ('voulez-vous continuer (o/n)?');
    read (reponse);
    if (reponse<>'o') and (reponse<>'O') then fini:=true else k:=k+1;
   end;
writeln;
writeln ('COEFFICIENTS MODIFIES DU POLYNOME: ');
for j:=0 to degre do writeln ('degre ',j:3,': ',coeff[j]);
for j:=0 to degre do writeln (coeffout,coeff[j]);
for j:=0 to degre do
   begin
    outbin^:=coeff[j];
    put (outbin);
   end;
close (outbin,i);
close (coeffout,i);
close (coeffin,i);
end.
```

TRACE D'EXECUTION DU PROGRAMME BESTPOL
d'amélioration d'une approximation polynômiale.
(Sur MICRAL 90.50)

La fonction que l'on désire approximer est la fonction sinus.

Les coefficients de départ du polynôme sont ceux du développement en série entière de la fonction.

La variable "sigma" désigne la valeur courante de la fonction qui mesure l'erreur d'approximation. Elle constitue une valeur approchée de l'erreur relative moyenne sur l'intervalle de test.

A chaque étape, les chiffres modifiés sont soulignés.

Il importe de bien réaliser que si pour des raisons de clarté nous donnons les coefficients en décimal, c'est en fait la sortie en binaire qu'il conviendrait d'utiliser.

```
degre du polynome?
17
bornes de l'intervalle de tests
 0.00000000000000E+000  1.57000000000000E+000
terme a partir duquel on s'autorise a modifier les coefficients?
3
puissance de 2 a partir de laquelle on modifie?
46nombre de valeurs testees?
500
ITERATION NUMERO 1
je travaille sur le coeff.   3 iteration   1
valeur actuelle de sigma:  1.06182859401580E-016
je travaille sur le coeff.   5 iteration   1
valeur actuelle de sigma:  9.79791067371322E-017
je travaille sur le coeff.   7 iteration   1
valeur actuelle de sigma:  9.70285025215751E-017
je travaille sur le coeff.   9 iteration   1
valeur actuelle de sigma:  9.70285025215751E-017
je travaille sur le coeff.  11 iteration   1
valeur actuelle de sigma:  9.70285025215751E-017
je travaille sur le coeff.  13 iteration   1
valeur actuelle de sigma:  9.70285025215751E-017
je travaille sur le coeff.  15 iteration   1
valeur actuelle de sigma:  9.66881727694985E-017
je travaille sur le coeff.  17 iteration   1
valeur actuelle de sigma:  9.66881727694985E-017
coefficients actuels du polynome:
degre   0: 0.00000000000000E+000
degre   1: 1.00000000000000E+000
degre   2: 0.00000000000000E+000
degre   3: 1.66666666666669E-001
degre   4: 0.00000000000000E+000
degre   5: 8.33333333333321E-003
degre   6: 0.00000000000000E+000
degre   7:-1.98412698412018E-004
degre   8: 0.00000000000000E+000
degre   9: 2.75573192101528E-006
degre  10: 0.00000000000000E+000
degre  11:-2.50521067982746E-008
degre  12: 0.00000000000000E+000
degre  13: 1.60589364903713E-010
degre  14: 0.00000000000000E+000
degre  15:-7.64291780689105E-013
degre  16: 0.00000000000000E+000
degre  17: 2.72047909578888E-015
ITERATION NUMERO 2
je travaille sur le coeff.   3 iteration   2
valeur actuelle de sigma:  9.66881727694985E-017
je travaille sur le coeff.   5 iteration   2
valeur actuelle de sigma:  9.66881727694985E-017
je travaille sur le coeff.   7 iteration   2
valeur actuelle de sigma:  9.66881727694985E-017
je travaille sur le coeff.   9 iteration   2
valeur actuelle de sigma:  9.56656267140417E-017
je travaille sur le coeff.  11 iteration   2
valeur actuelle de sigma:  9.56656267140417E-017
je travaille sur le coeff.  13 iteration   2
valeur actuelle de sigma:  9.56656267140417E-017
je travaille sur le coeff.  15 iteration   2
valeur actuelle de sigma:  9.56656267140417E-017
```

```
je travaille sur le coeff.  17 iteration   2
valeur actuelle de sigma:  9.56656267140417E-017
coefficients actuels du polynome:
degre   0: 0.00000000000000E+000
degre   1: 1.00000000000000E+000
degre   2: 0.00000000000000E+000
degre   3: 1.66666666666669E-001
degre   4: 0.00000000000000E+000
degre   5: 8.33333333333321E-003
degre   6: 0.00000000000000E+000
degre   7:-1.98412698412017E-004
degre   8: 0.00000000000000E+000
degre   9: 2.75573192101528E-006
degre  10: 0.00000000000000E+000
degre  11:-2.50521067982746E-008
degre  12: 0.00000000000000E+000
degre  13: 1.60589364903713E-010
degre  14: 0.00000000000000E+000
degre  15:-7.64291780689105E-013
degre  16: 0.00000000000000E+000
degre  17: 2.72047909578888E-015
ITERATION NUMERO 3
je travaille sur le coeff.   3 iteration   3
valeur actuelle de sigma:  9.56656267140417E-017
je travaille sur le coeff.   5 iteration   3
valeur actuelle de sigma:  9.32786018094098E-017
je travaille sur le coeff.   7 iteration   3
valeur actuelle de sigma:  9.32786018094098E-017
je travaille sur le coeff.   9 iteration   3
valeur actuelle de sigma:  9.32786018094098E-017
je travaille sur le coeff.  11 iteration   3
valeur actuelle de sigma:  9.32786018094098E-017
je travaille sur le coeff.  13 iteration   3
valeur actuelle de sigma:  9.32786018094098E-017
je travaille sur le coeff.  15 iteration   3
valeur actuelle de sigma:  9.32786018094098E-017
je travaille sur le coeff.  17 iteration   3
valeur actuelle de sigma:  9.32786018094098E-017
coefficients actuels du polynome:
degre   0: 0.00000000000000E+000
degre   1: 1.00000000000000E+000
degre   2: 0.00000000000000E+000
degre   3: 1.66666666666669E-001
degre   4: 0.00000000000000E+000
degre   5: 8.33333333333321E-003
degre   6: 0.00000000000000E+000
degre   7:-1.98412698412017E-004
degre   8: 0.00000000000000E+000
degre   9: 2.75573192101528E-006
degre  10: 0.00000000000000E+000
degre  11:-2.50521067982746E-008
degre  12: 0.00000000000000E+000
degre  13: 1.60589364903713E-010
degre  14: 0.00000000000000E+000
degre  15:-7.64291780689105E-013
degre  16: 0.00000000000000E+000
degre  17: 2.72047909578888E-015
```

II - D Un problème parfois délicat : La réduction d'argument.

Nous avons étudié des algorithmes permettant de calculer une fonction dans un intervalle. Or, la plupart des fonctions mathématiques sont définies sur un domaine non borné : notre problème consiste donc à nous **ramener** à l'intervalle de convergence; cette opération est appelée **réduction d'argument**.

Exemple : on désire calculer l'exponentielle de 100 en utilisant une approximation rationnelle qui converge sur [0 , ln (2)]. On cherchera à évaluer **le plus précisément possible** le réel x compris entre 0 et ln (2) tel que :

$$100 = x + N \cdot \ln(2) \qquad (N \text{ entier})$$

Et le résultat désiré sera alors :

$$e^{100} = e^{x} \cdot 2^{N}$$

La quantité 2^N étant évaluée par multiplications successives (il faudra environ $\text{Log}_2 N$ multiplications).

L'étape de réduction d'argument peut parfois paraître triviale, notamment pour les fonctions trigonométriques. Il n'en est rien : l'expérience prouve que de nombreux logiciels effectuent très mal cette opération, qui doit être menée avec beaucoup de soin. Prenons par exemple les résultats, examinés auparavant, des tests de la fonction sinus de MULTICS. On voit que la fonction est assez bien approximée sur [0 ,π/2] (on perd au plus 9 bits), mais que l'implémentation laisse passablement à désirer sur [6π , 6.5π]. On voit donc que c'est la réduction d'argument qui est en cause.

Il est assez facile de cerner l'origine du problème : lors de la réduction **additive** d'argument, c'est à dire lorsque l'argument réduit est obtenu en additionnant ou en soustrayant à l'argument d'origine un multiple d'une constante (π par exemple pour les fonctions trigonométriques), si l'argument d'origine est grand, **cette opération revient à soustraire deux nombres de taille comparable**, ce qui est souvent source d'erreur.

Cherchons à éliminer au moins en partie cet inconvénient :

supposons donc que l'on désire soustraire N.c (N est entier et c est la constante de la réduction d'argument) à l'argument d'origine x, afin d'obtenir un argument réduit compris entre 0 et c, ou entre -c/2 et c/2. Soit M le nombre de chiffres de mantisse dans la base B du système utilisé, et soit p le nombre de chiffres communs entre N.c et x. (plus x est grand, plus p est proche de M). Il est clair que si l'on effectue directement :

$$y := x - N * c ;$$

Alors, on n'aura plus que M - p digits de base B significatifs.

Si par contre on écrit $c = c_1 + c_2$, où c_2 s'écrit **de façon exacte** en base B, sur un nombre de chiffres d suffisament petit pour que, pour des valeurs "raisonnables" de N, $n.c_1$ s'écrive **exactement** en base B sur moins de M digits, alors l'éxécution de :

$$y_1 := x - N * c_1 ;$$

N'entraine **aucune erreur de calcul**.

Il reste ensuite à effectuer :

$$y := y_1 - N * c_2 ;$$

et le nombre de chiffres significatifs sur le résultat sera alors de l'ordre de M - p + d : tout se passe comme si l'on avait **simulé un système de plus grande précision pour stocker c sur M + d digits**. Cette idée originale semble due à W. Cody (21).

II - E Le logiciel FELIMAG.

FELIMAG a tout d'abord été écrit sur MICRAL 90.50 pour remplacer la bibliothèque d'origine de calcul des fonctions élémentaires, TRANCEND, fournie avec le système PASCAL.

Les premières fonctions implémentées ont donc été les fonctions PASCAL standard : Sinus, Cosinus, Logarithme népérien, Exponentielle et Arctangente.

Notre travail a ensuite été axé sur deux aspects :

- L'écriture **d'autres fonctions** que les fonctions PASCAL standard, par exemple les fonctions Sinus hyperbolique, Cosinus hyperbolique, Tangente et Arg. tangente hyperboliques, élévation d'un nombre réel à une puissance réelle quelconque (power), Gamma, Erf, Erfc, les fonctions de Bessel, ainsi que quelques fonctions de répartition statistiques.

- La conception d'un logiciel pour d'**autres micro-ordinateurs** que le MICRAL 90.50 (en particulier l'IBM PC). L'étude précédente montre s'il en était besoin qu'un programme de calcul performant **ne peut pas** être entièrement indépendant de la machine sur laquelle on travaille, et que par conséquent, on ne doit pas se contenter de recopier exactement un logiciel qui donne satisfaction sur un autre système, mais on doit le modifier afin de l'adapter au nouveau contexte.

Dans ce style d'étude, le calcul de l'approximant **théorique** peut se faire sur n'importe quel système (les nôtres sont pour la plupart tirés de (20) et (44)).

Par contre, que l'on utilise la méthode de rattrappage de l'erreur de troncature où que l'on fasse appel à un programme d'amélioration d'approximant du style de notre programme Bestpol, on ne doit plus utiliser que le système sur lequel sera exécuté le programme de calcul de la fonction désirée.

Les programmes de la bibliothèque FELIMAG sont donnés en annexe, dans la version la plus récente au moment de la frappe de ce texte (versions 5.0 ou 6.0 suivant les fonctions). Auparavant, nous décrivons rapidement les différentes méthodes utilisées pour chaque programme.

Chapitre III

Les bases discrètes: un outil pour le calcul matériel des fonctions élémentaires.

III - Bases discrètes

III - A - Bases discrètes additives.

III- A - 1 - Premières définitions.

Nous noterons S le cône des suites réelles, strictement positives, décroissantes et sommables ($(e_n) \in S \Longrightarrow \sum_{n=0}^{\infty} e_n < +\infty$).

Définition 1. Soit $E = (e_n)$ un élément de S, on appellera **Ensemble généré d'ordre p de E** l'ensemble:

$$G_p(E) = \{ \sum_{n=0}^{\infty} d_n e_n \ / \ d_n \in \{0, 1, \ldots, p\}\}.$$

Soit $x \in G_p(E)$, si x peut s'écrire $\sum_{n=0}^{\infty} d_n e_n$ on dira que la suite (d_n) est un **système de coordonnées** (non nécessairement unique) de x sur E.

Propriété 1. Pour tout élément E de S, $G_p(E)$ a la puissance du continu, et n'a pas de points isolés.

Démonstration.

a - $G_p(E)$ n'a pas de points isolés.

Posons $E = (e_n)$.

Soit $x \in G_p(E)$, $x = \sum_{n=0}^{\infty} d_n e_n$.

Montrons que pour tout $\varepsilon > 0$, il existe $y \in G_p(E)$, $y \neq x$, tel que $|x - y| < \varepsilon$.

E est sommable, donc $\lim_{n \to \infty} e_n = 0$, et par conséquent, pour tout $\varepsilon > 0$, il existe N tel que :

$$n > N \Longrightarrow 0 < e_n < \varepsilon/p.$$

Soit y défini par $y = \sum_{n=0}^{\infty} d'_n e_n$, où:

$$\begin{cases} - \ d'_n = d_n \text{ si } n \neq N. \\ - \ d'_N = 0 \text{ si } d_N \neq 0, \ d'_N = 1 \text{ sinon.} \end{cases}$$

Nous avons $|y - x| = |d'_N - d_N| \ e_n$

soit $0 < |y - x| < p e_n$

soit $0 < |y - x| < \varepsilon$.

Ce qu'il fallait démontrer.

b - $G_p(E)$ a la puissance du continu.

Etant donnée l'inclusion évidente $G_q(E) \subset G_p(E)$ pour tout $q \leq p$, il suffit de montrer que $G_1(E)$ a la puissance du continu. Pour ceci, remarquons que si $E' = (e'_n)$ est une suite extraite de $E = (e_n)$, alors $G_1(E') \subset G_1(E)$.

Construisons une suite (e'_n) comme suit:

$e'_0 = 0$.

$\lim_{n\to\infty} e_n = 0$ donc il existe N tel que $e_N < e_0 / 2$.

posons $e'_1 = e_N$.

de même, il existe $M > N$ tel que $e_M < e'_1 / 2$, et nous choisirons $e'_2 = e_M$.

En itérant ce processus, nous construisons une suite $E' = (e'_n)$ extraite de (e_n) telle que pour tout n, $e'_{n+1} < e'_n / 2$. (1).

Montrons que $G_1(E')$ a la puissance du continu.

Pour ceci, établissons que l'application de $\{0, 1\}^{\mathbb{N}}$ vers $G_1(E')$ qui à (d_n) fait correspondre $\sum_{n=0}^{\infty} d_n e'_n$ est injective.

Soit $x = \sum_{n=0}^{\infty} d_n e'_n$. Supposons qu'il existe (d'_n) différente de (d_n) et telle que $x = \sum_{n=0}^{\infty} d'_n e'_n$.

En notant k le plus petit entier vérifiant $d_k \neq d'_k$, il vient:

$$(d_k - d'_k)\, e'_k = \sum_{n=k+1}^{\infty} (d_n - d'_n)\, e'_n .$$

Soit:

$$e'_k = \left| \sum_{n=k+1}^{\infty} (d_n - d'_n)\, e'_n \right| \leq \sum_{n=k+1}^{\infty} \left| (d_n - d'_n) \right| e'_n$$

$$\leq \sum_{n=k+1}^{\infty} e'_n \quad (2).$$

Or, (1) implique que pour tout $n>k$, $e'_n < 2^{-(n-k)} e'_k$.

Par conséquent $\sum_{n=k+1}^{\infty} e'_n < e'_k \sum_{n=k+1}^{\infty} 2^{-(n-k)} = e'_k$.

Ce qui contredit (2).

D'où le résultat.

Propriété 2. (unicité d'écriture)

Si pour tout entier naturel n, $e_n > p \sum_{k=n+1}^{\infty} e_k$, alors tout élément de $G_p(E)$ admet un système de coordonnées unique sur E.

Démonstration: Si x s'écrit $x = \sum_{i=0}^{\infty} d_i e_i = \sum_{i=0}^{\infty} d'_i e_i$ $(d_i) \neq (d'_i)$, alors, en notant r le plus petit entier tel que $d_r \neq d'_r$, il vient:

$$(d_r - d'_r)\, e_r = \sum_{i=r+1}^{\infty} (d'_i - d_i)\, e_i .$$

Et par conséquent: $e_r < |(d_r - d'_r)\, e_r| < \sum_{i=r+1}^{\infty} |d_i - d'_i| e_i$

$$< p \sum_{i=r+1}^{\infty} e_i$$

Ce qui contredit l'hypothèse.

Remarque: Si l'on a seulement $e_n > p \sum_{i=n+1}^{\infty} e_i$, alors, dans la démonstration précédente, il ne peut y avoir égalité entre $\sum_{i=0}^{\infty} d_i e_i$ et $\sum_{i=0}^{\infty} d'_i e_i$ que si:

$$d_r - d'_r = 1 \quad \text{et} \quad \forall\, n > r,\ d_n = 0 \text{ et } d'_n = p.$$

ou:

$$d_r - d'_r = -1 \text{ et } \forall\, n > r,\ d_n = p \text{ et } d'_n = 0.$$

Ce qui exige d'ailleurs que e_r soit égal à $p \sum_{i=r+1}^{\infty} e_i$.

On en déduit de plus que dans ce cas, un élément de $G_p(E)$ admet au plus deux systèmes de coordonnées sur E. Ceci est à rapprocher de la convention 0.99999999... = 1 de l'écriture décimale des nombres.

III - A - 2 **Bases discrètes additives: Définition et propriétés**

Définition 2: Soit E un élément de S. On dira que E est une **base discrète (additive) d'ordre p** si $G_p(E)$ est un intervalle. C'est alors l'intervalle $[0 , p_n \sum_{n=0}^{\infty} e_n]$.

Théorème 1: $E = (e_n)$ est une base discrète d'ordre p si et seulement si pour tout entier n, $e_n \leq p \sum_{k=n+1}^{\infty} e_k$. (A)

Démonstration:

a - La condition est nécessaire.

Supposons qu'il existe $n \in \mathbb{N}$ tel que $e_n > p \sum_{k=n+1}^{\infty} e_k = r_n$, et posons $I =] r_n , e_n [$. Montrons que l'intersection de I avec $G_p(E)$ est vide.

Soit $a \in G_p(E)$, $a = \sum_{i=0}^{\infty} d_i e_i$.

1) S'il existe $k<n$ tel que $d_k \neq 0$, alors $a > d_k e_k > e_k > e_n$ puisque la suite (e_i) décroit.

2) Si pour tout $k<n$, $d_k = 0$, alors $a \leq p \sum_{k=n+1}^{\infty} e_k$.

Par conséquent, dans tout les cas, $a \notin I$.

b - La condition est suffisante.

Pour établir ceci, montrons que si (A) est vérifiée, alors pour tout $a \in I = [0 , p_n \sum_{n=0}^{\infty} e_n]$, la suite (d_n) définie comme suit nous assure: $\sum_{n=0}^{\infty} d_n e_n = a$.

$$\begin{cases} a_0 = 0 \\ d_i = \text{Max} \{ j \in \{ 0, 1, \ldots , p \} / a_i + j e_i \leq a \} \\ a_{i+1} = a_i + d_i e_i . \end{cases}$$

Il est clair que notre problème revient à démontrer que a est la limite de la suite a_n.

Dans ce but, établissons par récurrence le résultat suivant:

$$|a_n - a| \leq p \sum_{k=n}^{\infty} e_k \qquad (B)$$

- Cette propriété est vraie si n = 0, puisque a ∈ I.

- Supposons qu'elle soit vraie pour un entier $n > 0$, et évaluons la quantité $|a_{n+1} - a|$.

1) Si $d_n < p$, alors, puisque d_n est égal à

Max { j ∈ { 0, ..., p } / $a_n + d_n e_n \leq a$ } , nous avons:

$$a_{n+1} = a_n + d_n e_n \leq a \leq a_n + (d_n + 1)\, e_n$$

donc, par conséquent: $a_{n+1} \leq a \leq a_{n+1} + e_n$.

donc, en utilisant (A), $|a_{n+1} - a| \leq p \sum_{k=n+1}^{\infty} e_k$.

2) Si $d_n = p$, alors $a_{n+1} = a_n + pe_n \leq a$,

donc $|a_{n+1} - a| = |a - a_n| - pe_n \leq p \sum_{k=n}^{\infty} e_k - pe_n$

(d'aprés l'hypothèse de récurrence)

$$\leq p \sum_{k=n+1}^{\infty} e_k .$$

Ce qu'il fallait démontrer.

Dans de nombreux cas, la condition (A est difficile à vérifier directement. Nous donnons ci aprés un théorème permettant de prouver plus simplement que certaines suites de S sont des bases discrètes.

Théorème 2: Si E = (e_n) ∈ S est une base discrète d'ordre p, si f est une fonction à variable réelle vérifiant:

1) f' est continue sur $I = [0 , p \sum_{n=0}^{\infty} e_n]$.
2) f(0) = 0.
3) f est concave et strictement croissante sur I.

Alors f(E) = (f(e_n)) est une base discrète d'ordre p.

Démonstration:

a - (f(e_n)) est une suite décroissante de réels strictement positifs.

Ceci provient de la croissance stricte de f. Pour tout entier naturel n, $e_n > e_{n+1} > 0$, donc $f(e_n) > f(e_{n+1}) > f(0) = 0$.

b - $\sum_{n=0}^{\infty} f(e_n) < +\infty$.

Pour ceci, il suffit d'établir que pour tout entier naturel n:

$$f(e_n) < e_n f'(0).$$

En effet, f étant concave et f' étant continue sur I, f' est donc décroissante sur I. Par conséquent:

$$f(e_n) = \int_0^{e_n} f'(x)\,dx < f'(0) \int_0^{e_n} dx = e_n f'(0)$$

c - (f(e_n)) vérifie (A).

La concavité de f sur I implique que pour tout (a_1 , a_2 ,..., a_n) $\in I^n$ ($a_i > 0$) tel que $p(a_1 + a_2 + \ldots + a_n)$ est inclus dans I, nous avons :

$$f\,[\,p(a_1 + \ldots + a_n)\,] < p\,[\,f(a_1) + \ldots + f(a_n)\,]$$

Par conséquent, f étant continue, si la série $p\sum_{i=0}^{\infty} a_i$ converge vers un élément de I ($a_i \in I$), alors:

$$f\left(p\sum_{i=0}^{\infty} a_i\right) < p\sum_{i=0}^{\infty} f(a_i).$$

Or, pour tout entier naturel n, nous avons $e_n < p\sum_{i=n+1}^{\infty} e_i$,

par conséquent, $f(e_n) < f\left(p\sum_{i=n+1}^{\infty} e_i\right)$

$$< p\sum_{i=n+1}^{\infty} f(e_i).$$

Donc $(f(e_n))$ est une base discrète d'ordre p.

Exemples.

a - Suites géométriques.

Le théorème 1 nous permet de constater que la suite (e_n) définie par $e_n = K\,a^{-n}$ est une base discrète d'ordre p si

et seulement si $1 < a \leq p+1$.

De telles bases, étudiées par A. Renyi ([71], [72]), constituent une première généralisation de la notion usuelle de bases de numération ($a \in \mathbb{N}$).

On peut, par exemple écrire "en base π " $1/2 =$ "0.11211202...", ce qui signifie qu'il existe une suite (d_n) (non nécessairement unique), telle que $1/2 = \sum_{n=0}^{\infty} d_n \pi^{-n}$, et vérifiant $d_0 = 0$, $d_1 = 1$, $d_2 = 1$, $d_3 = 2$, etc...

Nous étudierons ultérieurement des algorithmes permettant de calculer les termes d_i.

Propriété 3: (Caractère Fractal de $G_p(E)$ [56]) Si E est une suite géométrique de raison strictement inférieure à 1 alors toute intersection non vide de $G_p(E)$ avec un intervalle ouvert contient un sous ensemble homothétique à $G_p(E)$.

En effet, posons $e_n = ca^{-n}$ soit $I =]\alpha, \beta[$ un intervalle ouvert d'intersection non vide avec $G_p(E)$, soit x un élément de cette intersection, définissons la quantité $\varepsilon = \min \{ x - \alpha , \beta - x \}$.

La suite $\sum_{k=n}^{\infty} e_k$ tend vers zéro lorsque n tend vers l'infini, par conséquent, il existe N tel que:

$$n>N \implies p_k \sum_{k=n}^{\infty} e_n < \varepsilon$$

Si x s'écrit $\sum_{n=0}^{\infty} d_n e_n$, avec $d_n \in \{0, \ldots p\}$, alors si l'on considère l'ensemble:

$$H_p^N(E) = \{ \sum_{n=0}^{\infty} d'_n e_n \ / \ d'_n = d_n \text{ si } n<N, \ d'_n \in \{0, \ldots p\} \text{ sinon} \}.$$

On a l'inclusion évidente: $H_p^N(E) \subset I$. Or on a manifestement $H_p^N(E) = f(G_p(E))$ où f est l'application:

$$y \rightarrow a^{-N} y + x_N$$

Où $x_N = \sum_{i=0}^{N} d_i e_i$.

Ce qu'il fallait démontrer.

Propriété 4: (séparation de $G_p(E)$ en p+1 branches translatées).

Il existe un ensemble A_0 tel que $G_p(E) = A_0 \cup A_1 \cup \ldots \cup A_p$, où $A_i = A_0 + ie_0$.

La démonstration est triviale si l'on pose $A_i = \{ \sum_{n=0}^{\infty} d_n e_n \ / \ d_0 = i \}$.

Les figures 1 à 3 donnent des exemples d'ensembles $G_p(E)$ pour E géométrique.

Suites asymptotiquement géométriques.

Le théorème 2 nous prouve que les suites:

$$e_n = \text{Arctg}\ (a^{-n})$$
$$e'_n = \text{Log}\ (1 + a^{-n})$$

Sont des bases discrètes d'ordre p si et seulement si $1 < a < p+1$.

Par exemple, en "base (Log$(1 + \pi^{-n})$)", Log (2) peut s'écrire "1.00000...", et 2 peut s'écrire "2.2012202...".

On peut qualifier ces suites d'"asymptotiquement géométriques" puisqu'elles sont toutes deux équivalentes à a^{-n}.

Les figures 4 et 5 donnent des exemples d'ensembles $G_p(E)$ pour E asymptotiquement géométrique. On constatera la similitude (prévisible) avec le cas où E est géométrique.

On a donc caractérisé les éléments E de S tels que $G_p(E)$ est un intervalle. On est alors en droit de chercher à quelles conditions $G_p(E)$ **contient** un intervalle I, car nous verrons que cette propriété pourrait nous permettre de calculer des fonctions sur I. Le théorème suivant répond partiellement à cette interrogation.

Théorème 3: Si la suite $E = (e_k)$ vérifie:

$$(C) \quad \forall\, n \in \mathbb{N}, \quad e_n > p \sum_{k=n+1}^{\infty} e_k$$

Alors $G_p(E)$ ne contient aucun intervalle.

Démonstration:

Soit a un élément quelconque de $G_p(E)$.

Soit ε un réel strictement positif.

Montrons qu'il existe un réel b n'appartenant pas à $G_p(E)$ tel que $|b - a| < \varepsilon$.

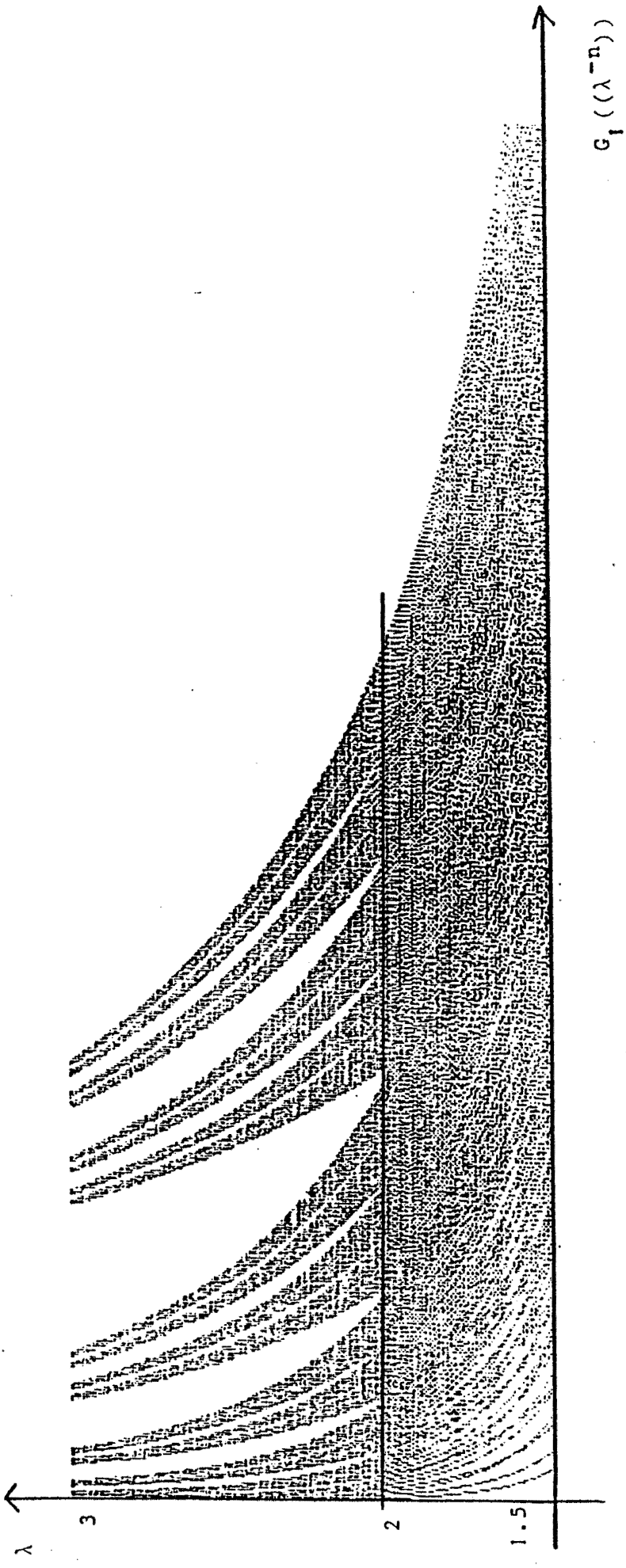


Fig. 1 Ensembles $G_1((\lambda^{-n}))$ pour différentes valeurs du paramètre λ. On voit clairement que la valeur $\lambda=2$ apparait comme un point critique.

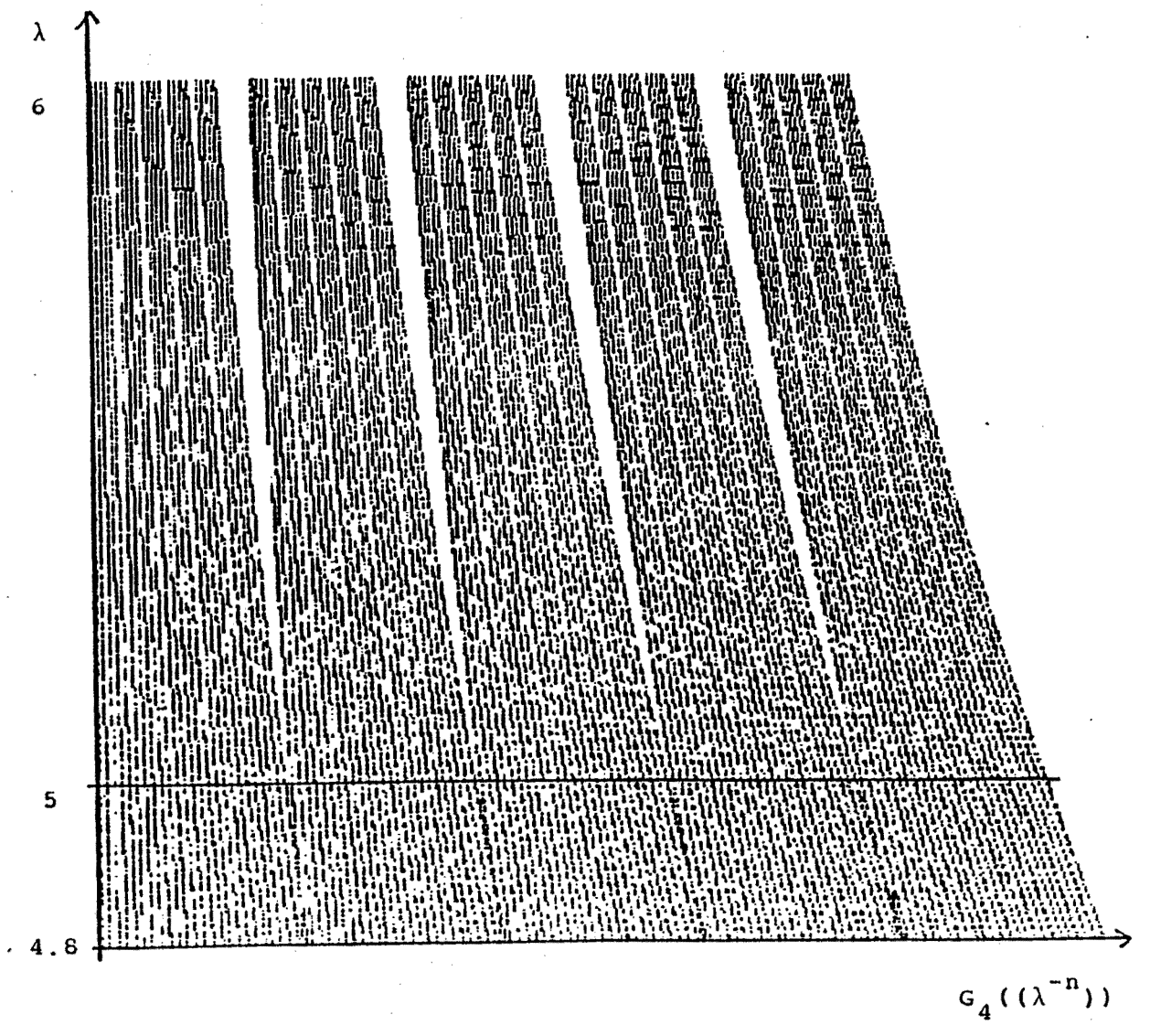


Fig. 2 Ensembles $G_4((\lambda^{-n}))$ tracés pour λ variant continûment entre 4.8 et 6.

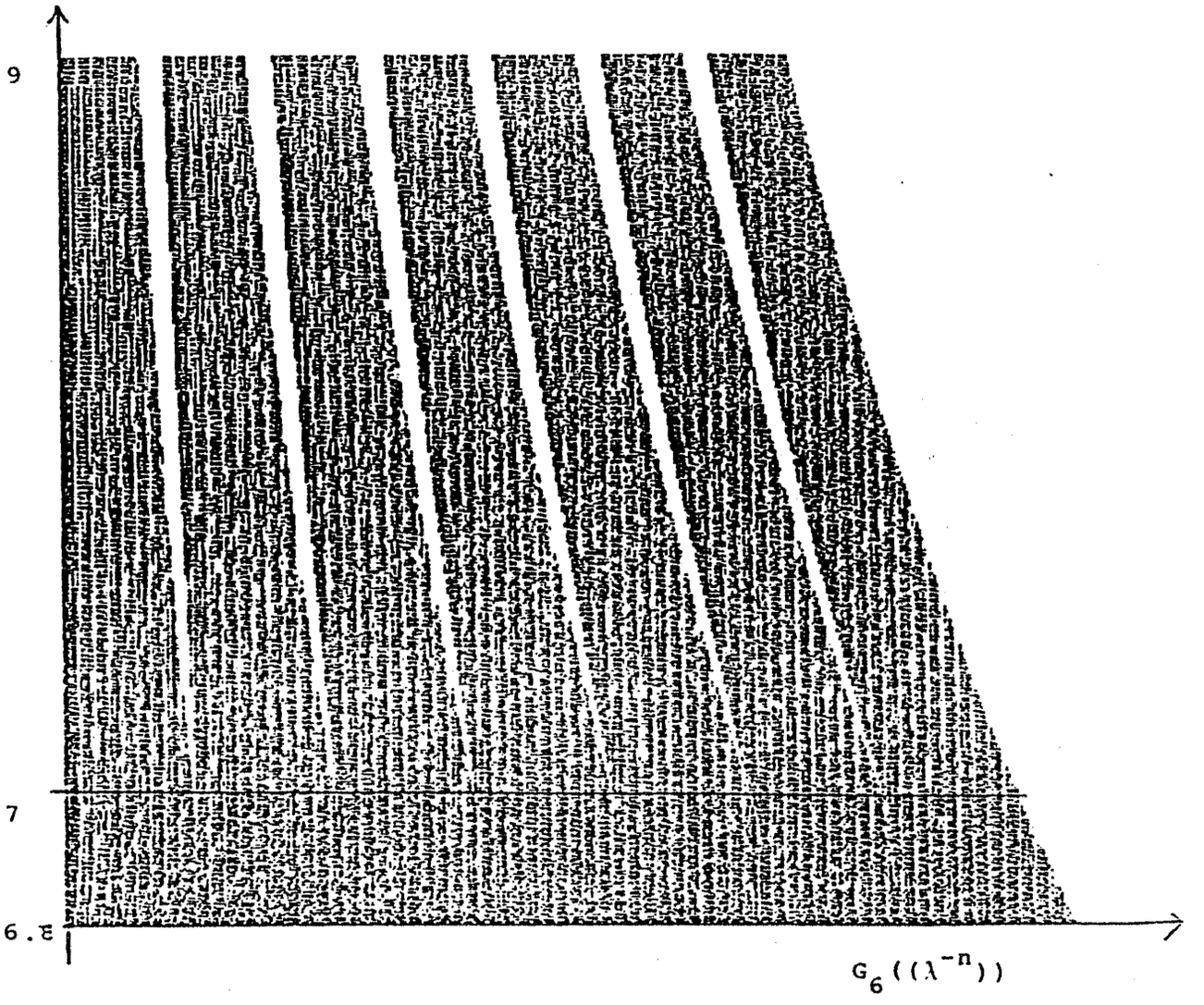


Fig. 3 Ensembles $G_6((\lambda^{-n}))$ tracés pour λ variant continûment entre 6.8 et 9.

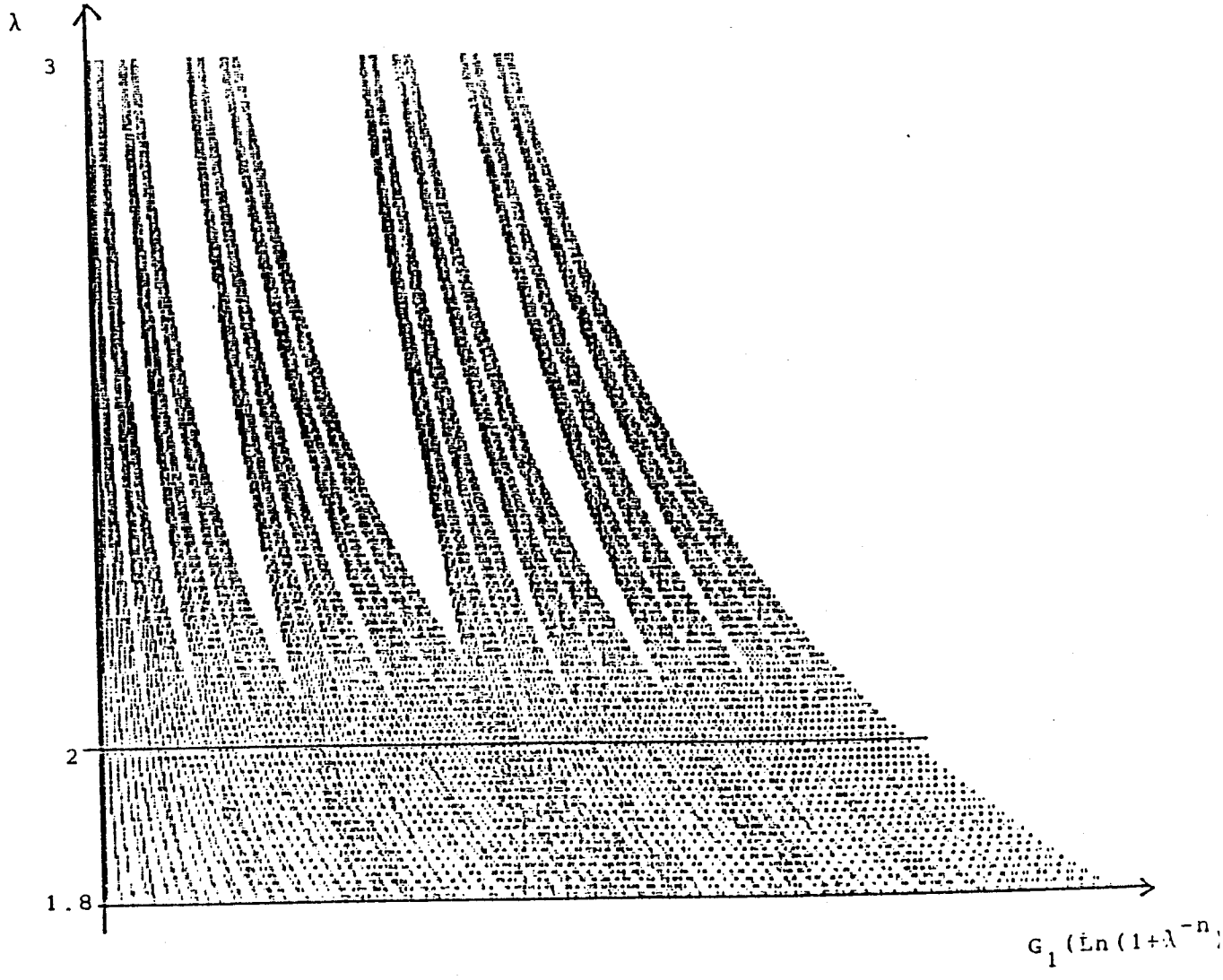


Fig. 4 Tracé des ensembles $G_1(\mathrm{Ln}(1+\lambda^{-n}))$, λ variant continûment de 1.8 à 3.

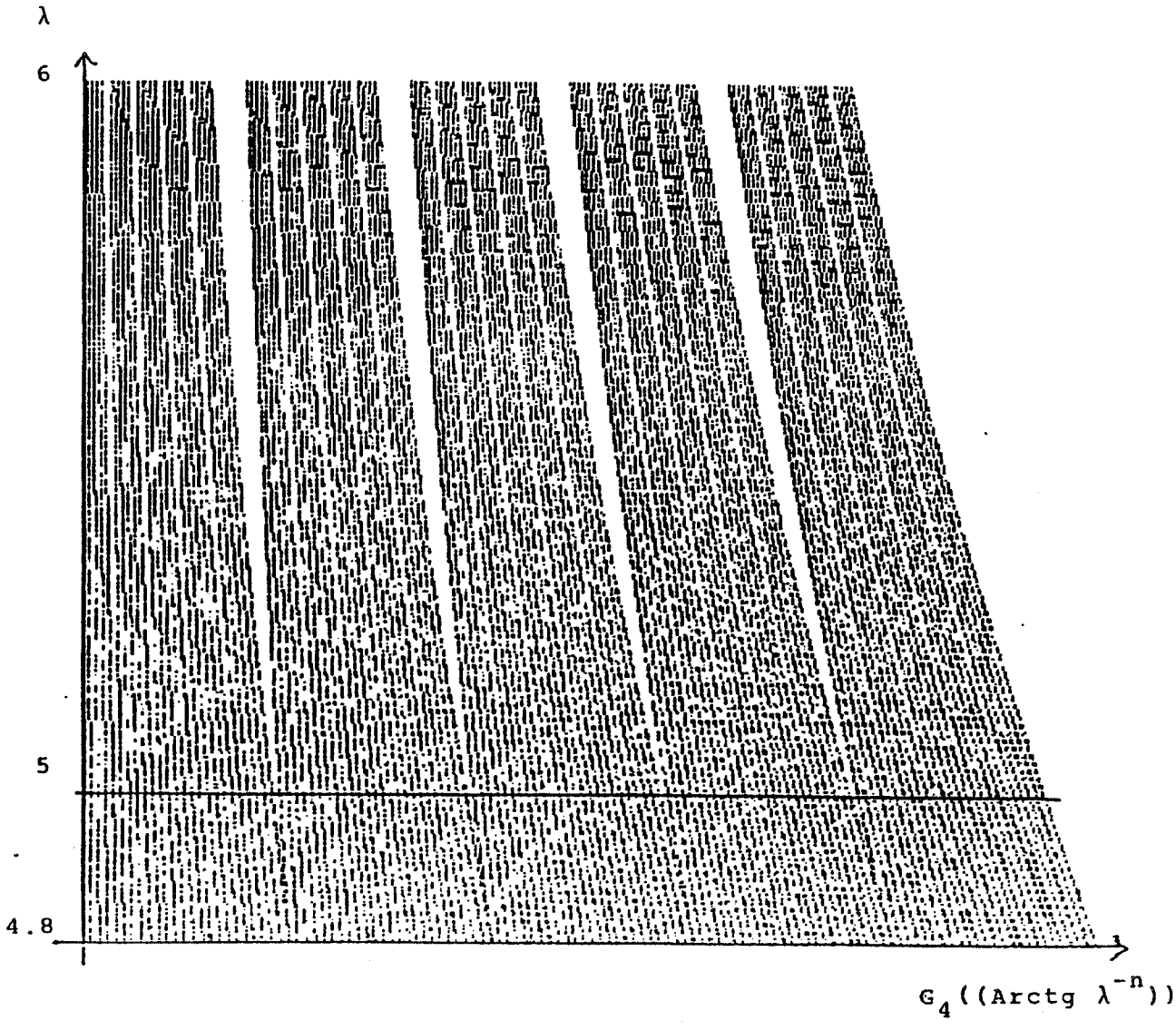


Fig. 5 Ensembles $G_4(\text{Arctg}(\lambda^{-n}))$ tracés pour λ variant continûment entre 4.8 et 6.

Posons $a = \sum_{n=0}^{\infty} d_n e_n$, $d_n \in \{0, 1, \ldots, p\}$.

La série (e_n) étant convergente, nous avons: $\lim_{n\to\infty} e_n = 0$, par conséquent il existe $n \in \mathbb{N}$ tel que pour tout $n > N$, on ait $pe_n < \varepsilon$.

Posons $c = \sum_{n=0}^{\infty} d'_n e_n$, où

$$\begin{cases} d'_n = d_n \text{ si } n < N \\ d'_N = 0 \\ d'_n = p \text{ si } n > N. \end{cases}$$

et $d = \sum_{n=0}^{\infty} d''_n e_n$, où

$$\begin{cases} d''_n = d_n \text{ si } n < N \\ d''_N = 1 \\ d''_n = 0 \text{ si } n > N. \end{cases}$$

(C) implique $c < d$. Soit b un élément de l'intervalle $]c , d[$.

Supposons que b puisse s'écrire $\sum_{n=0}^{\infty} d'''_n e_n$, avec $d'''_n \in \{0, 1, \ldots, p\}$. Soit alors r le plus petit entier naturel tel que d'''_r est différent de d_r.

1) Si $r < N$, alors:

- si $d'''_r < d_r$ alors $c - b = \sum_{n=0}^{\infty} (d'_n - d'''_n) e_n$

$$= (d_r - d'''_r) e_r + \sum_{n=r+1}^{\infty} (d'_n - d'''_n) e_n$$

> 0 puisque $e_r > p \sum_{n=r+1}^{\infty} e_n$.

- si $d'''_r > d_r$ alors on montre de la même façon que $b-d > 0$.

2) Si $r > N$ alors:

- si $d'''_N = 0$ alors $b < c$
- si $d'''_N > 1$ alors $b > d$ par construction.

D'où contradiction, ce qui démontre le résultat énoncé.

Lorsque (C) est vérifiée, l'ensemble $G_p(E)$ vérifie d'autres propriétés intéressantes, comme le montrent les résultats suivants:

Propriété 5: Si E e S vérifie (C) alors $G_p(E)$ est fermé.

Démonstration: Soit (x_n) une suite d'éléments de $G_p(E)$ convergeant dans $\mathbb{R}$, Montrons que sa limite appartient à $G_p(E)$.

Posons $x_n = \sum_{i=0}^{\infty} d_i^n e_i$. Nous allons exhiber une suite (d_i), i e $\{0, 1, \ldots, p\}$ telle que pour tout entier naturel i, il existe N e $\mathbb{N}$ tel que si n est supérieur ou égal à N, alors $d_i^n = d_i$. (Ceci n'est qu'une façon d'exprimer une "convergence" de la suite des coordonnées des x_n).

En effet, supposons:

$$\exists\, j \in \mathbb{N},\ \forall\, N\ \exists p > N,\ \exists n > N,\ d_j^n \neq d_j^p .$$

Nous avons: $\left|x_n - x_p\right| = \left|\sum_{i=0}^{\infty} (d_i^n - d_i^p)\, e_i\right|$

Soit $\varepsilon_j = \min_{i<j} \left\{ e_i - p_k \sum_{i+1}^{\infty} e_k \right\}$. D'aprés (C), $\varepsilon_j > 0$.

Soit r le plus petit entier naturel tel que $d_r^p \neq d_r^n$, il vient:

$$\begin{aligned} \left|x_n - x_p\right| &= \left|(d_r^n - d_r^p)\, e_r - \sum_{k=r+1}^{\infty} (d_k^p - d_k^n)\, e_k\right| \\ &> \left|e_r - \left|\sum_{k=r+1}^{\infty} (d_k^n - d_k^p)\, e_k\right|\right| \\ &> \left|e_r - p_k \sum_{r+1}^{\infty} e_k\right| > \varepsilon_j \end{aligned}$$

On en déduit donc:

$$\exists\, \varepsilon = \varepsilon_j\ \forall\, N \in \mathbb{N},\ \exists n > N\ \ \exists p > N\ \ \left|x_n - x_p\right| > \varepsilon$$

Donc la suite (x_n) n'est pas une suite de Cauchy, ce qui est évidemment en contradiction avec le fait qu'elle soit convergente. Nous avons par conséquent:

$$\forall\, j \in \mathbb{N}\ \exists N_j \in \mathbb{N}\ \forall\, n > N_j\ ,\ \forall\, p > N_j\ ,\ d_j^n = d_j^p .$$

Soit la suite (d_j) définie par $d_j = d_j^{N_j}$, et soit x le réel égal à $\sum_{j=0}^{\infty} d_j e_j$. Il nous reste à montrer que x est la limite de la suite (x_n) . Pour ceci, posons $M_j = \operatorname{Sup}_{k<j} N_k$.

Il vient:

$$\forall\, j \in \mathbb{N} \quad \exists M_j \in \mathbb{N} \quad \forall\, n > M_j \ , \ \forall\, i < j \ , \ d_i^n = d_i .$$

D'où:

$$\forall\, j \in \mathbb{N} \quad \exists M_j \in \mathbb{N}, \ \forall\, n > M_j \ , \quad \left| x - x_n \right| < p \sum_{i=j+1}^{\infty} e_i .$$

Or, puisque la série (e_i) est convergente, nous avons:

$$\lim_{j \to \infty} \sum_{i=j+1}^{\infty} e_i = 0.$$

Donc x est bien la limite de la suite (x_n) , donc, puisque x est élément de $G_p(E)$ par construction, la propriété 5 est démontrée.

Propriété 6: Si $E \in S$ est une suite asymptotiquement géométrique de la forme (a^{-n}) ,avec a p+1,et vérifiant (C), alors la mesure de Lebesgue de $G_p(E)$ est nulle.

Avant de démontrer cette propriété, énonçons le lemme suivant:

Lemme 1: Si l'on note

$$B_{N,p}^{(d_0, d_1, \ldots d_N)}(E) = \left[\sum_{i=0}^{N} d_i e_i \ , \ \sum_{i=0}^{N} d_i e_i + p \sum_{i=N+1}^{\infty} e_i \right]$$

$$\Omega_p(E) = \bigcap_{N \in \mathbb{N}} \left[\bigcup_{(d_0, \ldots, d_N) \in \{0,1,\ldots,p\}^{N+1}} \left(B_{N,p}^{(d_0, d_1, \ldots, d_N)}(E) \right) \right]$$

Alors:

1) $\forall\, E \in S, \quad G_p(E) \subset \Omega_p(E)$

2) Si $E \in S$ vérifie (C), alors, $G_p(E) = \Omega_p(E)$, et tous les intervalles $B_{N,p}^{D}(E)$, où $D = (d_0, \ldots, d_N) \in \{0,1,\ldots,p\}^{N+1}$ sont disjoints.

N.B. Pour démontrer la proposition 6, 1) suffit.

Démonstration du lemme:

- 1) est trivialement vraie.

- Pour prouver 2), nous attirons l'attention du lecteur sur le fait que la disjonction des $B^D_{N,p}(E)$ est une conséquence immédiate de (C), il nous suffit donc de montrer que $\Omega_p(E)$ est inclus dans $G_p(E)$. Pour ceci, considérons un élément x de $\Omega_p(E)$ et montrons qu'il appartient à $G_p(E)$:

Soit $N \in \mathbb{N}$, puisque les $B^D_{N,p}$ sont disjoints, il existe un unique $(d_0^x, d_1^x, \ldots, d_N^x)$ tel que $x \in B^{(d_0^x, d_1^x \ldots, d_N^x)}_{N,p\ (E)}$

Considérons alors $y = \sum_{n=0}^{\infty} d_n^x e_n$. Il est clair que pour tout $N \in \mathbb{N}$ il existe $D \in \{0, 1, \ldots, p\}^N$ tel que x et y appartiennent au même $B^D_{N,p}(E)$. Or, la longueur de $B^D_{N,p}(E)$ décroit vers zéro avec N. Par conséquent, on a nécessairement $x = y$.

Démonstration de la propriété 6:

Posons:

$$A_{N,p}(E) = \bigcup_{(d_0, \ldots d_N) \in \{0, \ldots, p\}^{N+1}} \left[B^{(d_0, d_1, \ldots, d_N)}_{N,p\ (E)} \right]$$

(Nous avons alors $\Omega_p(E) = \bigcap_{N \in \mathbb{N}} A_{N,p}(E)$)

Or, la mesure de $B^D_{N,p}(E)$ ne dépend pas de D, et vaut $\sum_{n=0}^{\infty} e_n$, par conséquent la mesure de $A_{N,p}(E)$ est majorée par:

$$\sum_{(d_0, \ldots, d_N) \in \{0, \ldots, p\}} \left(\sum_{k=N+1}^{\infty} e_k \right)$$

$$= (p+1)^{(N+1)} p \sum_{n=N+1}^{\infty} e_n$$

Mais ce dernier terme est équivalent, lorsque N tend vers l'infini, à:

$$((p+1)/a)^N\ p(p+1)/(a-1)$$

Or, cette quantité tend vers zéro lorsque N tend vers l'infini.

Donc la mesure de Lebesgue de $G_p(E)$ est nulle.

Propriété 7: Si E e S est une suite géométrique de raison a, a > p+1, alors la dimension de Hausdorff de $G_p(E)$ est égale à Log (p+1) / Log (a)

Exemple: On retrouve comme cas particulier de ce résultat l'ensemble triadique de Cantor, dont la dimension de Hausdorff est bien connue et vaut Log (2) / Log (3).

Démonstration:

Considérons la **mesure de Hausdorff dans la dimension d**, définie comme suit:

Soit G un sous-ensemble compact de $\mathbb{R}$, considérons un recouvrement Δ^ρ de G par des intervalles ouverts Δ_i^ρ (i e $\mathbb{N}$), de diamètre inférieur ou égal à ρ. Le terme:

$$H_\rho^d(G) = \inf_{\Delta^\rho}\left(\sum_{i=0}^{\infty} \text{Diam } (\Delta_i^\rho)^d \right)$$

Admet une limite (éventuellement infinie) quand ρ tend vers zéro par valeur supérieure, cette limite, notée H(G), sera appelée **mesure de Hausdorff de G dans la dimension d.**

On montre les propriétés suivantes:

i) Si G est réunion de deux compacts disjoints G_1 et G_2, alors $H^d(G) = H^d(G_1) + H^d(G_2)$.

ii) Si F est homothétique de G dans le rapport k, alors $H^d(F) = k^d H^d(G)$.

iii) Il existe une quantité Dim (G) e [0 , 1] vérifiant:

$$\text{Dim } (G) = \inf \{d \text{ e } [0 , 1] / H^d(G) = 0\}$$
$$= \sup \{d \text{ e } [0 , 1] / H^d(G) = +\infty\}.$$

Cette quantité Dim (G) est appelée **Dimension de Hausdorff** de G. On trouvera plus de précisions sur cette dimension dans [45] et [80].

iv) Si la mesure de Lebesgue de G est non nulle, alors Dim (G) = 1.

Dans le cas qui nous intéresse, le résultat est une

conséquence des propriétés 3 et 4. En effet, $G_p(E)$ peut alors s'écrire comme réunion de p+1 compacts disjoints A_0 , ..., A_p , qui lui sont homothétiques dans le rapport 1/a. Par conséquent, $G_p(E)$ est ce que Kahane et Salem appellent un **Parfait homogène de type (p+1,1/a)**, donc, d'aprés leur étude (voir [45]), sa dimension de Hausdorff est égale à Log (p+1) / Log (a). On peut s'en persuader en écrivant que:

$$H^d(G_p(E)) = H^d(A_0) + \ldots + H^d(A_p)$$
$$= (p+1)\, H^d(A_0)$$
$$= (p+1)\, a^{-d} H^d(G_p(E))$$

Par conséquent, si $(p+1)a^{-d} \neq 1$, alors $H^d(E)$ est nul ou infini. Il suffit alors de montrer que pour $d = Log(p+1)/Log(a)$, la mesure de Hausdorff de $G_p(E)$ dans la dimension d est finie et non nulle. La propriété iii donne alors immédiatement le résultat. Nous conjecturons que ce résultat reste vrai si E est seulement **asymptotiquement géométrique.**

III - A - 3 Problème de minimalité de la décomposition.

Introduction: Il est évident que, si (e_n) est une base discrète géométrique ou asymptotiquement géométrique (par exemple une base usuelle de numération), alors si l'on désire approximer l'élément $x = \sum_{n=0}^{\infty} d_n e_n$ de $G_p(E)$ par la quantité $x^* = \sum_{n=0}^{N} d_n e_n$, le nombre N de chiffres nécessaires pour avoir $|x - x^*| < \varepsilon$ est proportionnel à $Log(1/\varepsilon)$.

Nous cherchons ici à déterminer si cette approximation est optimale, c'est à dire s'il existe une base discrète (e_n) telle que :

$$\forall A > 0 \quad \exists N \in \mathbb{N}, \quad n>N \implies p \sum_{k=n+1}^{\infty} e_k < e^{-nA} \qquad (D)$$

Ceci serait d'autant plus intéressant que, nous le verrons ultérieurement, le temps de calcul des algorithmes du chapitre IV est proportionnel au nombre N de chiffres nécessaires à la décomposition approchée de x. La réponse à la question matérialisée par (D) est non. En fait, nous donnons ici un résultat plus général concernant la représentation des nombres.

Théorème 4: Soit (f_n) une famille de "codages approchés d'éléments d'un intervalle I sur N digits de $\{0, 1, \ldots, p\}$ ", c'est à dire une application de $\{0, 1, \ldots, p\}^N$ dans I, alors il existe deux constantes strictement positives K et c vérifiant:

$$\exists x \in I, \forall D \in \{0, \ldots, p\}^N , \left| f_N(d) - x \right| > ce^{-KN} .$$

Démonstration: Soit R_N l'ensemble $\{ f_N(D) / D \in \{0, \ldots, p\}^N\}$. Le cardinal de R_N est inférieur ou égal à $(p+1)^N$, par conséquent, I contient au plus $(P+1)^N$ éléments de R_N. Notons ces éléments:

$$x_1 < x_2 < \ldots < x_{(p+1)^N}$$

Notons aussi x_0 la borne inférieure de I et $x_{(p+1)^N + 1}$ sa borne supérieure. Nous montrerons d'abord les résultats intermédiaires suivants:

Lemme 3: Il existe $i \in \{0, 1, \ldots, (p+1)^N\}$ tel que

$$x_{i+1} - x_i \geqslant d / ((p+1)^N + 1)$$

où l'on note $d = x_{(p+1)^N + 1} - x_0$.

En effet, sinon on aurait :

$$d = \sum_{j=0}^{(p+1)^N} (x_{j+1} - x_j) < ((p+1)^N + 1) \cdot d / ((p+1)^N + 1)$$

D'où le résultat.

Lemme 4: Il existe x appartenant à I tel que pour tout élément D de $\{0, 1, \ldots p\}^N$, on ait:

$$\left| x - f_N(D) \right| > d / [2 ((p+1)^N + 1)] > d (p+2)^{-N} / 2$$

La démonstration de ce résultat est immédiate: il suffit de prendre, pour le i du lemme 3, $x = (x_{i+1} - x_i) / 2$.

Le théorème découle de ce dernier résultat, en posant $c = d/2$ et $K = \text{Log } (p+2)$.

On déduit de ce résultat l'optimalité de la décomposition des réels dans les bases géométriques ou asymptotiquement géométriques. Ceci est trés important pour les algorithmes que nous étudierons ultérieurement: en effet, le temps de calcul nécessaire pour obtenir une précision donnée ε en abcisse sera proportionnel au nombre de d_i nécessaires à l'expression de l'argument, et donc à Log ($1/\varepsilon$).

III - B - Bases discrètes multiplicatives.

Nous nous intéressons maintenant à l'écriture de nombres réels non plus comme **somme** de termes prédéfinis, mais comme **produit** de ces termes. Un résultat, le théorème 5, montre qu'une étude exhaustive de ce cas est inutile et que la plupart des résultats concernant les bases additives peuvent être étendus aux bases multiplicatives. Nous verrons ultérieurement que ceci est vrai également pour les algorithmes.

III - B - 1 - Définitions.

Dans tout ce qui suit, on notera M l'ensemble des suites réelles (e_n) dont les termes sont strictement supérieurs à 1 et décroissent vers 1, et telles que $\prod_{n=0}^{\infty} e_n$ est fini.

Définition 3: On appelera **Généré multiplicatif d'ordre p** de $E = (e_n) \in M$, l'ensemble:

$$GM_p(E) = \left\{ \prod_{n=0}^{\infty} e_n^{d_n} \;/\; d_n \in \{0, \ldots, p\} \right\}$$

Définition 4: On dira que $E \in M$ est une **Base discrète multiplicative d'ordre p** si $GM_p(E)$ est un intervalle.

La fonction exponentielle étant un isomorphisme de groupes de $(\mathbb{R}, +)$ dans $(\mathbb{R}^{+*}, \times)$, on déduit le résultat suivant, fondamental pour l'étude des ensembles $GM_p(E)$.

Propriété 8: Soit $E = (e_n)$ un élément de M, la suite $Log(E) = (Log\ (e_n))$ est un élément de S, et la fonction exponentielle est une bijection de $G_p(Log(E))$ dans $GM_p(E)$.

On en déduit immédiatement, grace à l'étude des suites additives, que $GM_p(E)$ a la puissance du continu, n'a pas de points isolés; et que si pour tout $n \in \mathbb{N}$, $e_n > \left(\prod_{k=n+1}^{\infty} e_k\right)^p$, alors $GM_p(E)$ ne contient aucun intervalle, et tous ses éléments admettent un système de coordonnées unique sur E. Une autre conséquence de la propriété 8, plus importante pour notre étude est le théorème suivant:

III - C Algorithmes

III - C - 1 Algorithmes de calcul des d_i.

Dans ce premier paragraphe, nous cherchons à calculer rapidement les premiers termes des coordonnées d'un réel donné dans une base discrète. Nous étudierons surtout le cas des bases additives, celui des bases multiplicatives pouvant rapidement en être déduit.

Théorème 6 (algorithme unidirectionnel) Si $E = (e_n)$ est une base discrète d'ordre p, alors pour tout élément t de $[0 , p \sum_{n=0}^{\infty} e_n]$, les suites (t_n) et (d_n) définies comme suit vérifient $t = \lim_{n\to\infty} t_n = \sum_{n=0}^{\infty} d_n e_n$.

$$\begin{cases} t_0 = 0 \\ d_n = \text{Max} \{ j < p , t_n + j e_n < t \} \\ t_{n+1} = t_n + d_n e_n \end{cases}$$

Démonstration: En fait le théorème 6 a déjà été énoncé et démontré, il constitue la condition suffisante du théorème 1. Nous l'avons replacé ici pour plus de clarté.

Théorème 7 (algorithme bidirectionnel) Si E est une base discrète d'ordre p, alors pour tout élément t de $[- p\sum_{n=0}^{\infty} e_n , + p\sum_{n=0}^{\infty} e_n]$, les suites (t_n) et (d_n) définies comme suit vérifient $t = \sum_{n=0}^{\infty} d_n e_n = \lim_{n\to\infty} t_n$.

$$\begin{cases} t_0 = 0 \\ \text{si } t_n < t \text{ alors} \\ \quad d_n = \text{Max} \{ 1 < j < p , t_n + (j-1) e_n < t \} \\ \text{sinon } d_n = \min \{ -p < j < -1 , t_n + (j+1) e_n > t \} \\ t_{n+1} = t_n + d_n e_n . \end{cases}$$

Pour établir ce dernier résultat, comme dans la condition suffisante du théorème 1, on montrera aisément par récurrence la relation:

$$\left| t_n - t \right| \leq p \sum_{k=n}^{\infty} e_k .$$

On peut de la même manière exhiber des algorithmes multiplicatifs bidirectionnel et unidirectionnel.

III - C - 2 Application au calcul d'une classe de fonctions.

a - L' algorithme primal.

Soit f une fonction vérifiant une équation de la forme:

$$f\ (x\ T\ y) = g\ (f(x)\ ,\ f(y))$$

où le symbole T représente l'addition ou la multiplication.

exemples:

logarithme: $f(x) = \ln(x)$

$$\begin{cases} T = * \\ g(u,v) = u + v \end{cases}$$

exponentielle: $f(x) = e^x$

$$\begin{cases} T = + \\ g(u,v) = uv \end{cases}$$

sinus/cosinus: $f(x) = e^{ix}$

$$\begin{cases} T = x \\ g(u,v) = uv \end{cases}$$

Supposons qu'il existe une base discrète d'ordre p, (e_n), additive si T = + et multiplicative si T = *, et telle que pour tout u, $g(u,a_n)$ puisse se calculer uniquement à l'aide d'additions et de décalages; où, si l'on note z l'élément neutre de la loi T, on à:

soit $a_n \in \{f(e_n) , z\}$
(en ce cas nous utiliserons l'algorithme unidirectionnel)

soit $a_n \in \{f(e_n) , f(e_n^*) , z\}$, où e_n^* désigne l'inverse de e_n pour la loi T
(en ce cas nous choisirons l'algorithme bidirectionnel).

Lorsque nous désirons calculer f(t), si nous écrivons

$$t = \sum_{n=0}^{\infty} d_n e_n \qquad (T = +)$$

$$\text{ou } t = \prod_{i=0}^{\infty} e_n^{d_n} \qquad (T = *)$$

nous obtenons:

$f_i \to t$, f_i étant défini par
$i \to \infty$

$f_0 = z$

$f_{i+1} = g(f_i , a_i)$, où a_i vaut

$$\begin{cases} f(e_i \ T \ e_i \ T \ \dots \ T \ e_i) & (k \text{ fois}) \text{ si } d_i = k \quad (k > 0) \\ f(e_i^* \ T \ \dots \ T \ e_i^*) & (k \text{ fois}) \text{ si } d_i = -k \quad (k > 0) \\ z \text{ si } d_i \text{ est nul.} \end{cases}$$

Cette méthode de calcul de f(t) sera appelée algorithme **primal**

par opposition avec l'algorithme que nous allons maintenant étudier, et qui sera appelé algorithme **dual**.

b - L'algorithme dual.

Soit f une fonction **strictement croissante**, dont nous connaissons **un algorithme primal** de calcul sur un intervalle I.

Nous allons montrer ici qu'alors une **petite modification** de cet algorithme permet d'obtenir un algorithme de calcul de f^{-1} sur f(I), qui sera appelé **algorithme dual** de l'algorithme précédent.

En effet, lorsque nous calculons f(t) en construisant une suite (t_n) qui converge vers t, engendrée par un algorithme bidirectionnel ou unidirectionnel, nous obtenons une suite (f_n) qui converge vers f(t) (f_n est égal à $f(t_n)$).

Le calcul de t_{n+1} en fonction de t_n se fait en posant:

$$t_{n+1} = t_n + d_n e_n$$

où $d_n = \text{Max} \{ j < p \ , \ t_n + je_n < t \}$ (1)

(Algorithme unidirectionnel sur base discrète d'ordre p)

ou bien $d_n = \text{Max} \{ 1 < j < p, \ t_n + (j-1)e_n < t \}$

si $t_n < t$ (2)

$= \min \{ -p < j < -1, \ t_n + (j+1)\, e_n > t \}$

sinon.

Or, on peut remarquer que, en posant $u = f(t)$

et $u_n = f(t_n)$, si l'on remplace (1) par:

$$d_n = \text{Max} \{ j < p , f(t_n + je_n) < u \} \quad (1')$$

ou (2) par:

$$d_n = \text{Max} \{ 1 < j < p , f(t_n + (j-1)e_n) < u \}$$

si $u_n < u$.

$$= \min \{ -p < j < -1 , f(t_n + (j+1) e_n) > u \}$$

sinon, alors puisque f est strictement croissante, on obtient exactement la même suite (t_n) que précedemment: on a **ainsi mis en évidence un algorithme permettant de calculer t connaissant u**, c'est à dire un **algorithme de calcul de f^{-1}.**

Donnons tout de suite par souci de clarté un exemple, qui sera développé de manière plus approfondie dans le chapître suivant:

Supposons que, sur une machine travaillant en base 2, nous désirions calculer la fonction exponentielle en utilisant l'algorithme unidirectionnel appliqué sur la base discrète d'ordre 1:

$$(\text{Ln}(1 + 2^{-n}))$$

L'algorithme primal qui calcule cette fonction est le suivant:

(variable d'entrée: l'argument t, variable de sortie: la valeur finale de "exp").

(on pose $e_n = \ln (1 + 2^{-n})$)

```
Début
   x := 0 ; exp := 1 ; k := 0 ;
   Tant que (k < N) faire
      début
(L)      tant que (x + e_k > t) et (k < N) faire k := k+1 ;
         exp := exp + exp . 2^-k ;
         x := x + e_k ;
         k := k + 1
      fin
Fin.
```

Le nombre N de pas à effectuer en fonction de la précision désirée et l'intervalle de convergence de cet algorithme seront explicités au chapître suivant.

L'algorithme dual de cet algorithme, qui calcule le logarithme, est obtenu en remplaçant la ligne (L) par:

tant que $((exp + exp.2^{-k}) > u)$ et (k < N) faire k := k + 1.

La variable d'entrée est alors u, et la variable de sortie est la valeur finale de x.

Il est intéressant de constater que cet algorithme dual peut s'exprimer comme un algorithme primal utilisant l'algorithme unidirectionnel de décomposition sur la base discrète

multiplicative d'ordre 1:

$$(1 + 2^{-n}).$$

Il convient toutefois de ne pas croire que tout algorithme dual est aussi un algorithme primal: un contre exemple immédiat est l'algorithme de calcul de l'arctangente présenté au cours du prochain chapître.

Chapitre IV

Algorithmes destinés au calcul matériel des fonctions élémentaires.

IV - A - Calcul de l'exponentielle et du logarithme.

Nous allons ici étudier deux applications des algorithmes présentés à la fin du chapitre précédent : le calcul de la fonction exponentielle à l'aide de l'algorithme unidirectionnel **additif**, et le calcul du logarithme à l'aide de l'algorithme unidirectionnel **multiplicatif** .

IV - A - 1 - Calcul de l'exponentielle.

Nous avons vu au chapitre III que si a est un réel vérifiant :

$$1 < a \leq p+1 \qquad (p \in \mathbb{N})$$

alors la suite $(\ln (1 + a^{-n}))$ est une **base discrète additive d'ordre p**. L'idée de base de l'algorithme que nous étudions ici est de décomposer un nombre réel x sous la forme :

$$(1) \quad x = \sum_{n=0}^{\infty} d_n \ln (1 + a^{-n}) \quad d_n \in \{0, \ldots, p\}$$

en utilisant l'algorithme unidirectionnel, afin d'obtenir comme résultat :

$$(2) \quad e^x = \prod_{n=0}^{\infty} (1 + a^{-n})^{d_n}$$

Or, on sait que sur une machine travaillant en base B, les multiplications par une puissance de B peuvent s'éxécuter d'une manière très simple, à l'aide d'un **décalage**. Ici, on aura donc avantage à prendre a = B, et donc à utiliser la base discrète additive d'ordre B-1 $(\ln (1 + B^{-n}))$. En effet, une multiplication par $(1 + B^{-n})$ se ramènera à une addition et une multiplication par B^{-n}. Il est évident que la série (1) et le produit infini (2) seront tronqués à un rang N dépendant de la précision ε désirée.

Nous présentons tout d'abord l'algorithme obtenu, et nous expliciterons ensuite la relation entre N et ε.

Exponentielle.

```
(* Cet algorithme calcule l'exponentielle de t.
                                        ∞
Il converge pour t ∈ [0 , (B-1).Σ ln (1 + B^-i)]
                                       i=0

précision relative : environ B^-N
résultat : la valeur finale de la variable exp    *)

Début
    x := 0 ;
    exp := 1 ;
    pour k := 0 jusqu'à N faire
      début
        d := 0 ;
        u := 0 ;
        tant que (u < t) et (d < B - 1) faire
          début
            u := x + ln (1 + B^-k) ;
            si u ≤ t alors
              début
                x := u ;
                exp := exp + exp.B^-k
              fin ;
            d := d + 1
          fin
      fin
Fin.
```

Evaluation du nombre N de pas nécessaires :

Nous savons que lorsque nous approchons une variable t à l'aide d'une suite (t_n) générée par l'algorithme unidirectionnel de décomposition sur une base discrète d'ordre p (e_n), l'erreur au $n^{ième}$ pas, $|t_n - t|$, est majorée par $p\Sigma_{i>n} e_i = r_n$.

Dans le cas qui nous concerne, $e_i = \ln (1 + B^{-i})$ est strictement inférieur à B^{-i} , par conséquent le terme r_n est majoré par $(B-1).\sum_{i>n} B^{-i} = B^{-n}$.

On en déduit que l'erreur relative sur le résultat "exp" de l'algorithme précédent, égale à :

$$|\exp (t_n) - \exp (t)| / \exp (t)$$

sera majorée par $\exp (B^{-N}) - 1 \sim B^{-N}$.

IV - A - 1 - Calcul du logarithme.

Sur une machine travaillant en base B, l'idée fondamentale de cet algorithme est de décomposer un nombre réel x sur la base discrète multiplicative d'ordre B-1 $(1 + B^{-n})$:

$$x = \prod_{n=0}^{\infty} (1 + B^{-n})^{d_n} \qquad d_n \in \{0, ..., B-1\}.$$

en utilisant l'algorithme unidirectionnel multiplicatif et d'obtenir :

$$\ln (x) = \sum_{n=0}^{\infty} d_n \ln (1 + B^{-n}).$$

voici l'algorithme :

IV - B Un algorithme de calcul de l'exponentielle complexe.

Nous allons maintenant utiliser la théorie présentée auparavant pour élaborer un algorithme matériel permettant de calculer l'exponentielle complexe dans un domaine que nous expliciterons ultérieurement. Par souci de simplification, nous nous cantonnerons au calcul sur une machine binaire, bien qu'une généralisation à d'autres bases de numération soit tout à fait envisageable.

Nous retrouverons comme cas particuliers de cet algorithme ceux de calcul de l'exponentielle et du logarithme réels exposés précedemment, ainsi que l'algorithme CORDIC de calcul des fonctions trigonométriques introduit par Jack Volder en 1959 (82).

IV - B - 1 - Les équations

Nous désirons calculer les nombres réels a et b tels que:

$$a + ib = e^{x + iy}$$

où x et y sont donnés.
Nous avons la relation bien connue:

$$e^{x + iy} = (\cos y + i.\sin y) . e^{x}$$

Par conséquent, si nous désirons décomposer x et y sur des bases discrètes additives facilitant le calcul, nous aurons à satisfaire deux exigences:

- La base discrète $(e_n^{\,x})$ associée à la variable x doit être choisie de sorte qu'une multiplication par exp $(e_n^{\,x})$ puisse s'effectuer aisément pour tout n. Nous choisirons donc $e_n^{\,x} = \ln (1 + 2^{-n})$. On sait (voir chapitre III) que c'est une base discrète additive d'ordre 1.

logarithme.

(* calcule le logarithme de t

$$\text{où } t \in [1\ ,\ (\prod_{i=0}^{\infty} (1 + B^{-i}))^{(B-1)}]$$

résultat : la valeur finale de la variable L *)

```
Début
  x := 1 ;
  L := 0 ;
  pour k := 0 jusqu'à N faire
   début
     d := 0 ;
     u := 0 ;
     tant que (u < t) et (d < B-1) faire
        début
          u := x + x.B^-k ;
          si (u ≤ t) alors
             début
               x := u ;
               L := L + ln (1 + B^-k)
             fin ;
          d := d + 1 ;
        fin
   fin
Fin.
```

Evaluation du nombre N de pas nécessaires :

On peut montrer, de même manière que précedemment, qu'après exécution de l'algorithme :

$$|L - \ln(t)| \leq B^{-N}.$$

Remarque : Il est important de noter que les deux algorithmes précédents peuvent fort bien servir à calculer les fonctions β^t et Log_β , où β est un réel strictement supérieur à 1. Il suffit pour cela d'utiliser les bases discrètes :

$$Log_\beta\ (1 + B^{-n}) \quad \text{et} \quad (1 + B^{-n}).$$

On obtient donc :

$$f_{n+1} = f_n(1 + 2^{-n})^{d_n^x} (\cos(e_n^y) + id_n^y \sin(e_n^y))$$

$$= \cos(e_n^y) f_n (1 + 2^{-n})^{d_n^x} (1 + id_n^y . \mathrm{tg}(e_n^y))$$

$$= \eta_n f_n (1 + 2^{-n})^{d_n^x} (1 + id_n^y . 2^{-n})$$

Avec les choix :

$$\eta_n = \cos(e_n^y)$$

$$e_n^y = \mathrm{arctg}\, 2^{-n}.$$

En notant $a_n + ib_n = f_n$, nous avons alors :

$$a_{n+1} = \eta_n(1 + 2^{-n})^{d_n^x} (a_n - b_n d_n^y 2^{-n})$$

$$b_{n+1} = \eta_n(1 + 2^{-n})^{d_n^x} (b_n + a_n d_n^y 2^{-n})$$

Par conséquent, avec les choix :

$x_0 = y_0 = b_0 = 0$

$a_0 = 1$

$d_n^x = 1$ si $x_n + e_n^x \leq x$ sinon 0

$d_n^y = 1$ si $y_n < y$ sinon -1.

- La base discrète (e_n^y) associée à la variable y doit faciliter les calculs trigonométriques. Nous constaterons ultérieurement que le choix $e_n^y = \text{Arctg}(2^{-n})$ est particulièrement judicieux. c'est également une base **discrète additive d'ordre 1.**

Posons $t = x + iy$.

Nous allons construire une suite $t_n = x_n + iy_n$ convergeant vers t et définie par:

$$x_{n+1} = x_n + d_n^x e_n^x$$

$$y_{n+1} = y_n + d_n^y e_n^y$$

où d_n^x est donné par l'algorithme unidirectionnel (décomposition de x), et d_n^y par l'algorithme bidirectionnel.

Posons également $f_n = \exp(t_n)$

Il vient :

$$f_{n+1} = \exp(x_{n+1} + iy_{n+1})$$

$$= \exp((x_n + iy_n) + (d_n^x e_n^x + id_n^y e_n^y))$$

$$= f_n \exp(d_n^x e_n^x)(\cos(d_n^y e_n^y) + i.\sin(d_n^y e_n^y))$$

Mais nous savons que :

$$\exp(e_n^x) = 1 + 2^{-n}$$

$d_n^y = \pm 1$, par conséquent,

$$\cos(d_n^y e_n^y) = \cos(e_n^y)$$

$$\sin(d_n^y e_n^y) = \sin(e_n^y)$$

(algorithmes unidirectionnel et bidirectionnel)

Nous sommes assurés d'obtenir :

$$\lim_{n\to\infty} a_n + ib_n = e^{x + iy}$$

Pour peu que $x + iy$ soit dans un domaine que nous allons bientôt préciser. Hélas, nous avons dans nos équations une multiplication par un facteur η_n qui demanderait trop de temps pour que l'algorithme soit viable.

C'est d'ailleurs la seule "vraie" multiplication apparaissant dans ces équations, puisque celle par $(1 + 2^{-n})$ sera effectuée au moyen d'un décalage et d'une addition.

Nous pouvons éviter ce problème en effectuant un changement de variable : en effet, si nous convenons de noter

$$b'_n = b_n K_n$$

$$a'_n = a_n K_n$$

Où K_n est égal à $\prod_{j=n}^{\infty} \eta_j = (\prod_{j=n}^{\infty} (1 + 2^{-2j}))^{-1/2}$

il vient alors immédiatement :

$$a'_{n+1} = (1 + 2^{-n}) \ (a'_n - b'_n d_n^{\,y} e_n^{\,y})^{d_n^{\,x}}$$

$$b'_{n+1} = (1 + 2^{-n}) \ (b'_n + a'_n d_n^{\,y} e_n^{\,y})^{d_n^{\,x}}$$

Il suffit alors d'une petite modification : la valeur de a'_0 doit être $K_0 = 0.6072529350088812...$ au lieu de 1.

IV - B - 2 - Le domaine de convergence D.

Nous avons vu au chapître III que si (e_n) est une base discrète d'ordre 1, et si nous notons $E = \sum_{n=0}^{\infty} e_n$, alors l'algorithme unidirectionnel converge sur [0,E] tandis que l'algorithme bidirectionnel converge sur [-E,+E].

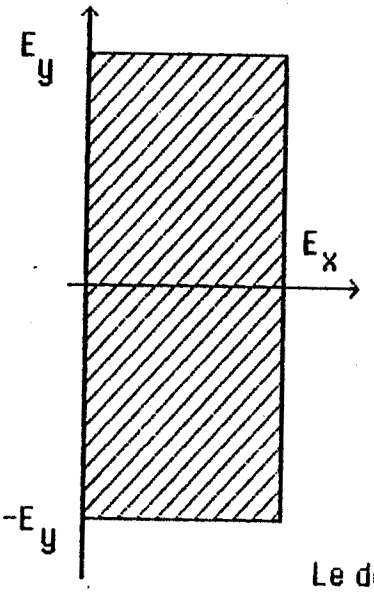
Par conséquent, dans le cas qui nous préoccupe, notre algorithme converge vers l'exponentielle complexe sur le pavé :

$$[0 , E_x] \times [-E_y , +E_y]$$

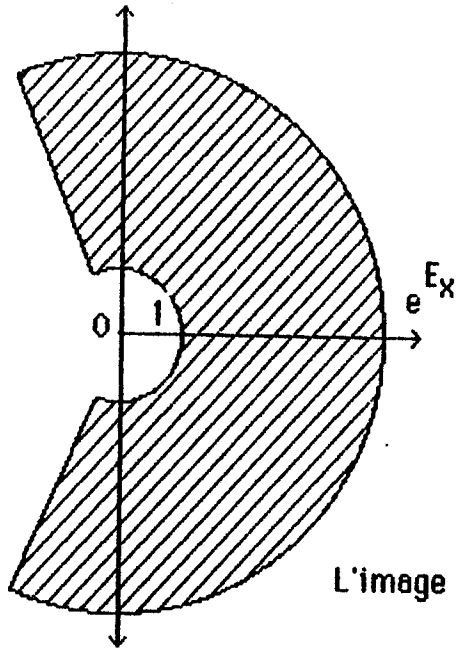
avec $E_x = \sum_{n \in \mathbb{N}} \ln (1 + 2^{-n}) = 1.562023833...$

et $E_y = \sum_{n \in \mathbb{N}} \text{arctg} (2^{-n}) = 1.743286621... > \pi/2$

La figure ci-dessous représente ce domaine D ainsi que son image D' par l'exponentielle complexe.



Le domaine de convergence D



L'image D' de D.

IV - B - 3 - Simplification des équations.

L'inconvénient majeur des équation précédentes réside dans l'évaluation de d_n^X par le biais de la relation :

$$d_n^X = 1 \text{ si } e_n^X + x_n \leq x, \text{ zéro sinon}$$

car celle-ci nous oblige à effectuer une addition (le résultat de cette opération ne resservira que si d_n^X est égal à 1). Ceci peut être évité si nous remplaçons les variables x_n et y_n par $x - x_n$ et $y - y_n$. L'algorithme revient alors à construire une suite qui converge vers zéro au lieu de $x + i.y$.

d_n^X et d_n^Y sont alors obtenus par :

$$d_n^X = 1 \text{ si } x_n > e_n^X, 0 \text{ sinon}$$

$$d_n^Y = 1 \text{ si } y_n > 0, -1 \text{ sinon.}$$

Avec les points de départ $x_0 = x$ et $y_0 = y$.

IV - B - 4 -Nombre de pas nécessaires.

Si nous construisons une suite (t_n) qui converge vers t en utilisant l'algorithme unidirectionnel ou l'algorithme bidirectionnel de décomposition sur une base discrète additive d'ordre 1 (e_n), nous avons trivialement :

$$|t_n - t| \leq \sum_{k > n} e_k$$

Ici, les deux bases discrètes utilisées sont **asymptotiquement géométriques** : elles vérifient

$$e_n \sim 2^{-n}$$

Cette propriété présente deux avantages non négligeables : tout d'abord, il ne sera pas nécessaire de stocker dans une mémoire morte (ROM) un grand nombre de constantes, puisque à partir d'un certain rang, e_n est égal à 2^{-n} à la précision machine près; ensuite, le terme :

$$r_n = \sum_{k>n} e_k$$

est lui aussi équivalent à 2^{-n}, ce qui indique que si nous notons nos variables en virgule fixe, **le nombre de pas nécessaires sera de l'ordre du nombre de bits significatifs désirés**.

IV - B - 5 - Extension du domaine de convergence.

La relation bien connue :

$$e^{z + ni\pi} = (-1)^n e^z$$

nous autorise, étant donné que la quantité $E_y = \sum_{i=0}^{\infty} \text{arctg } 2^{-i}$ est supérieure à $\pi/2$ (elle vaut environ 1.74), à effectuer une **réduction d'argument symétrique** (voir le chapitre V) vers l'intervalle $[-\pi/2 , +\pi/2]$ pour la variable y.

Pour la réduction d'argument de la variable x, on effectuera une réduction **positive** d'argument vers l'intervalle [0 , ln(4)], puisque la quantité :

$$E_x = \sum_{i=0}^{\infty} \ln (1 + 2^{-i})$$

est supérieure à ln (4).

Dans un but de clarté, dans l'algorithme décrit ci-dessous, nous nous contenterons d'effectuer la réduction d'argument par **soustractions successives**. Il est évident que pour une implémentation matérielle réaliste, il conviendrait d'opter pour l'algorithme de réduction d'argument en temps logarithmique décrit dans le prochain chapitre.

IV - B - 6 - L'algorithme.

Nous désirons calculer $a + ib = e^{x + iy}$.

Nous supposerons que les constantes suivantes ont été calculées à l'avance, et qu'elles sont stockées dans une ROM.

ln (4) = 1.38629436111989061833446 4 en décimal
= 1.30562057737216367465362 3 en octal.

π = 3.14159265358979323846264338 en décimal
= 3.11037552421026430215142306 en octal.

$e_i^x = \ln(1 + 2^{-i})$ i=0...N

$e_i^y = \text{arctg}\, 2^{-i}$ i=0...N

où N est égal au nombre de bits significatifs que l'on désire avoir sur le résultat final.

Ces dernières constantes sont données en octal en annexe.

Voici l'algorithme :

Début
a := K_0 ; b := 0 ;
(* première étape : réduction d'argument *)
nx := 0 ; ny := 0 ;
si x < 0 **alors tant que** x < 0 **faire**
début
nx := nx - 1 ;
x := x + ln (4) ;
fin
sinon tant que X > E_x **faire**
début
nx := nx + 1 ;
x := x - ln (4)
fin ;
si y < -E_y **alors tant que** y < -E_y **faire**
début
ny := ny - 1 ;
y := y + π
fin
sinon tant que y > E_y **faire**
début
ny := ny + 1 ;
y := y - π
fin ;

(* deuxième étape : calcul sur l'argument réduit *)

```
pour i := 0 jusqu'à N faire
  début
    si x ≥ e_i^X alors d^X := 1 sinon d^X := 0 ;
    si y ≥ 0 alors d^Y := 1 sinon d^Y := -1 ;
    x := x - d^X e_i^X ;
    y := y - d^Y e_i^Y ;
    a' := a - b.d^Y.2^-i ;
    b := b + a.d^Y.2^-i ;
    a := a' ;
    si d^X = 1 alors
      début
        a := a + a.2^-i ;
        b := b + b.2^-i
      fin
  fin ;
```

(* troisième étape : obtention de l'exponentielle de l'argument d'origine *)

```
si ny mod 2 = 1 alors
  début
    a := -a ;
    b := -b
  fin ;
si nx ≠ 0 alors
  début
    a := a . 2^2nx ;
    b := b . 2^2nx
  fin
Fin.
```

(* après exécution de cet algorithme, a + ib constitue l'exponentielle recherchée *)

IV - C - L'algorithme CORDIC.

Dans sa version initiale, due à J. Volder (82), CORDIC peut se présenter comme un cas particulier de l'algorithme de calcul de l'exponentielle complexe présenté auparavant. Il suffit pour s'en convaincre de constater que si l'on désire calculer l'exponentielle complexe d'un nombre imaginaire pur $i\theta$, alors l'itération associée à l'algorithme de la partie IV - B - 6 peut s'écrire :

$$x_{n+1} = x_n - d_n y_n 2^{-n}$$

$$y_{n+1} = y_n + d_n x_n 2^{-n}$$

$$z_{n+1} = z_n - d_n e_n$$

avec :

$$e_n = \text{arctg } 2^{-n}$$

$$d_n = \text{sign } (z_n)$$

L'étude menée au paragraphe IV-B permet de conclure que nous aurons alors :

$$x_n \mapsto \cos z_0 \qquad y_n \mapsto \sin z_0$$
$$n \mapsto \infty \qquad n \mapsto \infty$$

Pour peu que la valeur absolue de z_0 soit inférieure ou égale à $\sum_{i=0}^{\infty} e_i$ et que l'on ait choisi les valeurs initiales :

$$x_0 = \prod_{i=0}^{\infty} \cos e_i = \left(\prod_{i=0}^{\infty} (1 + 2^{-2i})\right)^{-1/2}$$

$$y_0 = 0$$

On peut montrer que l'erreur commise en s'arrêtant après N itérations est inférieure à 2^{-N}, tant pour le sinus que pour le cosinus.

Ce schéma itératif est connu sous le nom de **mode rotation** de l'algorithme de Volder.

L'algorithme dual de cet algorithme est obtenu en remplaçant l'affectation :

$$d_n = \text{sign}(z_n)$$

par : $d_n = -\text{sign}(y_n)$

on obtient alors :

$$z_n \mapsto \text{arctg}(y_0/x_0) + z_0$$
$$n \mapsto \infty$$

Cet algorithme est appelé **mode vectoring** de CORDIC.

En 1971, J. Walther (83) chercha à calculer les fonctions ch, sh et argth par un algorithme similaire obtenu en remplaçant la suite e_n de l'algorithme précédent par la suite (argth 2^{-n}) et en effectuant dans le schéma itératif les changements de signes appropriés.

Son but était de calculer l'exponentielle par addition du cosinus et du sinus hyperbolique, et le logarithme par le biais de la relation :

$$\ln x = 2\,\text{argth}((x-1)/(x+1))$$

Il constata que les modes rotation ($d_n = \text{sign}(z_n)$) et vectoring ($d_n = -\text{sign}(y_n)$) présentés auparavant ne donnent les résultats

souhaités **que si l'on répète les itérations numéros** 4, 13, 40, 121, ..., k, 3k+1, ...

Ceci peut s'expliquer d'une manière très simple : la suite (e_n) = (argth 2^{-n}) **n'est pas une base discrète d'ordre 1**, par contre, en étudiant les variations de la fonction :

$$f(x) = \text{argth}(x) - x - \text{argth}(x^3/2)$$

sur l'intervalle [0 , 1/2], on peut montrer la relation :

$$\text{argth } 2^{-i} \leq \sum_{k>i} \text{argth } 2^{-k} + \text{argth } 2^{-(3i+1)}$$

qui nous permet d'établir que la suite (a_n) définie par :

$$a_0 = e_0 , a_1 = e_1, ..., a_4 = e_4, a_5 = e_4 , a_6 = e_5 ...$$

obtenue en répétant les termes de la suite (e_n) d'indices 4, 13, 40, 121, ... (ces indices sont les termes de la suite récurrente $u_0 = 4$, $u_{n+1} = 3u_n + 1$), est **une base discrète additive d'ordre 1.**

Walther constata en outre qu'en choisissant $e_n = 2^{-n}$, le schéma itératif décrit ci-dessus pouvait s'apparenter aux algorithmes usuels de **multiplication** (mode rotation) et de **division** (mode vectoring).

L'algorithme ainsi obtenu est décrit dans l'introduction de cette thèse.

IV - D - Autres algorithmes.

IV - D - 1 Calcul de la racine carrée.

Nous supposerons ici que nous travaillons en base B.
Au chapitre **III** , nous avons pu constater que la suite :

$$(1 + B^{-n})$$

est une base discrète multiplicative d'ordre (B-1). On peut aisément en déduire que la suite :

$$((1 + B^{-n})^2)$$

est également une base discrète multiplicative d'ordre (B-1).

L'algorithme qui suit consiste, en utilisant l'**algorithme unidirectionnel multiplicatif**, à décomposer un nombre réel x sous la forme :

$$x = \prod_{n=0}^{\infty} ((1 + B^{-n})^2)^{d_n}$$

afin d'obtenir :

$$\sqrt{x} = \prod_{n=0}^{\infty} (1 + B^{-n})^{d_n}$$

L'algorithme qui suit utilise cette méthode, en tronquant les produits infinis au rang N. On peut montrer comme dans le paragraphe **IV** - A que l'erreur relative ε_r commise vérifie :

$$\varepsilon_r \leq \exp(B^{-N}) - 1 \sim B^{-N}$$

Racine carrée.

```
(* Cet algorithme calcule la racine carrée de t.
                                        ∞
  Il converge pour t ∈ [1 , ( Π  (1 + B^-i) )^2 ]
                                       i=0

  précision relative : environ B^-N
  résultat : la valeur finale de la variable sq   *)


début
   x := 1 ;
   sq := 1 ;
   pour k := 0 jusqu'à N faire
     début
        d := 0 ;
        u := 0 ;
        tant que (u ≤ t) et (d < B-1) faire
           début
              u := x + B^-2k x + B^-k x + B^-k x ;
              si u ≤ t alors
                 début
                    x := u ;
                    sq := sq + sq.B^-k
                 fin ;
              d := d + 1
           fin
     fin
Fin.
```

Cet algorithme peut facilement être transformé en algorithme de calcul de la racine $k^{ième}$ d'un nombre en utilisant la base discrète multiplicative d'ordre (B-1) :

$$((1 + B^{-n})^k)$$

IV - D - 2 Inversion d'un nombre.

L'algorithme présenté ci-dessous utilise une idée simple :
Si, pour une machine travaillant en base B, nous notons

$$K = \prod_{n=0}^{\infty} (1 + B^{-n})^{(B-1)}$$

alors, après avoir décomposé, à l'aide de l'algorithme unidirectionnel multiplicatif, un nombre réel $x \in [1 , K]$, sous la forme :

$$x = \prod_{n=0}^{\infty} (1 + B^{-n})^{d_n} \qquad d_n \in [0 , (B-1)]$$

nous obtiendrons :

$$K/x = \prod_{n=0}^{\infty} (1 + B^{-n})^{(B-1)-d_n}$$

Voici l'algorithme obtenu :

Inverse.

```
(* cet algorithme calcule l'inverse de t
   pour t ∈ [1 , K]
   avec une précision relative de l'ordre de B^-N
   résultat : la valeur finale de la variable inv *)

début
   x := 1 ;
   inv := 1/K ;
 (* La valeur 1/K est stockée en mémoire *)
   pour i := 0 jusqu'à N faire
     début
       d := 0 ;
       u := x + x . B^-i ;
       tant que (u < t) et (d < B-1) faire
         début
           x := u ;
           d := d + 1 ;
           u := x + x . B^-i
         fin ;
       tant que (d < B-1) faire
         début
           inv := inv + inv . B^-i ;
           d := d + 1
         fin
     fin
Fin.
```

IV - D - 3 - Division.

Cet algorithme, dont la mise en oeuvre est plus complexe que celle de l'algorithme de division CORDIC, est ici uniquement pour illustrer la notion d'algorithme dual.

En effet, on peut très aisément construire un algorithme de multiplication comme suit :

- on décompose une des opérandes s sur la base discrète multiplicative $(1 + B^{-i})$, afin d'obtenir :

$$s = \prod_{i=0}^{\infty} (1 + B^{-i})^{d_i} \qquad d_i = 0, \ldots, B-1$$

- on obtient le produit s.t par additions et décalages successifs :

$$st = t \prod_{i=0}^{\infty} (1 + B^{-i})^{d_i}$$

L'algorithme de multiplication ainsi obtenu est :

multiplication

(* effectue le produit de t par s
si s appartient à l'intervalle : $[1 , (\prod_{i=0}^{\infty} (1 + B^{-i}))^{(B-1)}]$

précision relative : environ B^{-N}
résultat : la valeur finale de la variable prod *)

```
Début
  x := 1 ;
  prod := t ;
  pour k := 0 jusqu'à N faire
    début
      d := 0 ;
      u := 1 ;
        tant que (u < s) et (d < B-1) faire          (1)
          début
            u := x + x . B^-k ;                      (2)
            si (u ≤ s) alors                         (3)
              début
                x := u ;                             (4)
                prod := prod + prod . B^-k           (5)
              fin ;
            d := d + 1
          fin
    fin
Fin.
```

On obtient alors un algorithme de division, dual de l'algorithme précédent, en remplaçant :

- la ligne (1) par "**tant que** (prod < num) **et** (d < B-1) **faire**"
- la ligne (2) par "u := prod + prod . B^{-k}"
- la ligne (3) par "**si** (u ≤ num) **alors**"
- la ligne (4) par "prod := u"
- la ligne (5) par "x := x + x . B^{-k}".

Après éxécution de cet algorithme, x sera égal à num/t avec une imprécision relative de l'ordre de B^{-N}.

Chapître V

La réduction d'argument par matériel.

V - La réduction d'argument par matériel.

Présentation du problème: Tous les algorithmes matériels que nous avons étudiés ne convergent que sur un petit intervalle, que nous appellerons **intervalle de convergence restreinte**, aussi il convient d'effectuer une **réduction d'argument** avant d'éxécuter ces algorithmes.

Généralement, lorsqu'un circuit intégré calcule des fonctions élémentaires, il n'effectue pas cette réduction d'argument: l'utilisateur doit s'en charger **par programme** (c'est par exemple le cas du coprocesseur 8087 d'INTEL). On arrive alors à une situation qui peut sembler paradoxale : le petit nombre d'opérations arithmétiques élémentaires nécessaires à la réduction d'argument demande autant de temps, sinon plus, que le calcul qui suit de la fonction trancendante.

Il convient donc de rechercher des algorithmes simples, susceptibles d'être intégrés, permettant d'effectuer la réduction d'argument au niveau du matériel. Comme précédemment, nous nous interdirons les multiplications et les divisions ne pouvant pas se ramener à un petit nombre d'additions/soustractions et de décalages.

Deux classes distinctes de réduction d'argument apparaissent lors du calcul des fonctions usuelles:

- La réduction **additive**, lors de laquelle l'argument réduit y est obtenu à partir de l'argument d'origine x par le biais de l'opération:

$$y = x - N.c$$

où c est une constante positive inhérente à la fonction

calculée (par exemple un multiple de $\pi/2$ pour les fonctions trigonométriques), et où N est un entier relatif.

- La réduction **multiplicative**, lors de laquelle l'argument réduit y est déduit de l'argument d'origine x par le biais de l'opération:

$$y = x.c^{-N}$$

où ici encore c est une constante positive dépendant de la fonction calculée, et N un entier relatif.

Lors de la réduction d'argument multiplicative, deux cas de figure peuvent se produire: soit on peut choisir c puissance entière de la base de numération B du système, **ce qui est le cas pour les fonctions élémentaires usuelles** (logarithme, racine carrée), auquel cas la réduction d'argument s'effectue de manière triviale, soit on ne le peut pas, et dans ce dernier cas il semble difficile d'éviter les multiplications et les divisions. Nous nous cantonnerons donc à l'étude de la réduction **additive** d'argument, en soulignant encore qu'en pratique, pour les fonctions usuelles, la réduction multiplicative ne pose pas de problèmes.

En pratique, lors de la réduction d'argument additive, l'intervalle de convergence restreinte contient soit l'intervalle [-c/2 , +c/2], dans ce cas nous parlerons de réduction **symétrique** d'argument, soit l'intervalle [0 , c/2] : ce sera la réduction **positive** d'argument.

Par souci de simplification, nous étudierons tout d'abord le cas de la réduction d'argument sur des machines travaillant en base 2. Le cas général sera abordé par la suite.

A - Divers algorithmes de réduction d'argument additive en base 2.

Si nous désirons éviter les multiplications et les divisions pour une implémentation matérielle, la première idée qui vient à l'esprit, pour obtenir l'argument réduit y correspondant à un argument d'origine x est d'éxécuter les algorithmes:

```
y := x ;
s := sign (x) ;
tant que |y| > c/2 faire y := y - s.c
```

(pour la réduction d'argument symétrique)

```
y := x ;
s := sign (x) ;
tant que ((y < 0) ou (y > c)) faire y := y - c.s
```

(pour la réduction d'argument positive)

le principal désavantage de ces algorithmes apparaît immédiatement: leur temps d'éxécution est proportionnel à |x| (on effectue environ |x/c| opérations arithmétiques), ce qui est rhédibitoire pour de grandes valeurs de x.

Une deuxième idée consiste à utiliser la fonction:

$$E(x) = \lfloor Log_2 |x| \rfloor$$

où $\lfloor u \rfloor$ désigne la partie entière de u.

Cette fonction est très aisément calculable puisqu'on l'obtient immédiatement à partir :

- de l'exposant de la représentation en virgule flottante de x.

- de l'emplacement du premier "1" à partir des poids forts dans la représentation en virgule fixe de x.

Par conséquent, si nous notons N l'entier tel que:

$$2^{N-1} \leq c < 2^{N} \quad \text{(N est égal à E(c) + 1)}$$

Il vient:

(R) $\quad 2^{E(x)-N} . c \leq |x| \leq 2^{E(x)-N+2} . c.$

Donc l'algorithme:

(J) $y := x$;
tant que $E(y) \geq N$ ***faire*** $y := y - c.2^{E(y)-N} . sign(y)$

nous assure une valeur finale de y comprise entre -2^N et $+2^N$, et ce en un nombre de passages dans la boucle de (J) majoré par 4.(E(x)-N), puisque la relation (R) implique que:

1) y reste toujours du même signe.
2) Il faut au plus 4 passages dans la boucle de (J) pour faire décroître strictement la valeur de E(y).

Par conséquent, après éxécution de l'algorithme (J), il ne reste plus qu'à effectuer:

- **pour la réduction symétrique:**

Si $|y| > c/2$ ***alors*** $y := y - c.sign(y)$

- **pour la réduction positive:**

Si $((y > c)$ ***ou*** $(y < 0))$ ***alors*** $y := y - c.sign(y)$

après quoi y est égal à l'argument réduit recherché.
Il est à noter que la borne 4.(E(x)-N) qui majore le nombre d'opérations arithmétiques effectuées peut fort bien être atteinte. On peut le montrer en considérant l'exemple suivant:

$c = 2.1$ et $x = 15$

qui nous donne comme valeurs successives de la variable y:

$15, 10.8, 6.6, 4.5$ et 2.4

on passe donc 4 fois dans la boucle de (J), la borne est par conséquent atteinte.

On peut par contre se demander si **pour toute valeur de c** le nombre maximal d'opérations arithmétiques nécessaires à l'éxécution de cet algorithme est égal à 4(E(x)-N). Ce n'est pas le cas, on s'en persuadera aisément en constatant que si $c.2^{-N}$ est supérieur à 2/3, alors pour toute valeur de y, $c.2^{E(y)-N}$ est supérieur à y/3, ce qui implique qu'il faut au plus 3 passages dans la boucle de (J) pour faire diminuer strictement E(y), et donc que le nombre d'opérations arithmétiques nécessitées par l'algorithme est majoré par 3(E(x)-N).

Il est par contre aisé de constater que pour tout c distinct d'une puissance entiére de 2, il existe une infinité de nombres x, aussi grands que l'on veut, tels que la réduction d'argument de x par cet algorithme nécessite un nombre de passages dans la boucle supérieur ou égal à E(x)-N : il suffit pour le constater de considérer un nombre x de la forme :

$$x = c + 2c + 4c + 8c + \ldots + 2^{p}c = c(2^{p+1}-1)$$

Cet algorithme, s'il est facilement implémentable en virgule flottante, est moins intéressant en virgule fixe car il nécessite à chaque pas la connaissance de E(y), qui ne peut être obtenu qu'en scrutant y bit à bit, ce qui constitue une perte de temps. Il est d'ailleurs à noter que le calcul **interne** peut être effectué en virgule fixe même si les entrées/sorties se font en virgule flottante (c'est le cas de notre coprocesseur **FELIN** : voir le chapître **IV** de cette thèse).

On utilisera donc de préférence l'algorithme suivant, qui ne demande qu'une seule fois l'utilisation de la fonction E, et qui nécessite **exactement** E(x)-N+3 opérations arithmétiques.

Début
y := x ;
K := E(x) - N + 2 ;
pour *i := K* ***jusqua*** *0* ***pas*** *-1* ***faire*** *y := y - sign (y).c.2^{i} ;*
Fin ;

La convergence de cet algorithme peut être établie en démontrant par récurrence qu'à chaque pas, la valeur absolue de la variable y est majorée par c.2^{i}.

En effet, si nous notons y_j la valeur de y avant éxécution de la boucle "pour" correspondant à i=j , et y_{-1} la valeur finale de cette variable y, alors :

- nous avons bien $|y_K| \leq c.2^{K+1}$

(c'est une conséquence immédiate de la relation $|x| \leq 2^{E(x)-N+2}$)

- si i=j ≥ 0, alors supposons que nous ayons

$$|y_j| \leq c.2^{j+1}$$

en ce cas, après exécution de la boucle d'indice j, nous aurons :

$$|y_{j-1}| = |y_j| - c.2^j \leq c.2^j$$

ce qu'il fallait démontrer.
Par conséquent, il vient :

$$-c \leq y_{-1} \leq +c.$$

Il restera donc, après éxécution de cet algorithme, à effectuer :

- pour la réduction d'argument **symétrique**:

si *|y| > c/2* ***alors*** *y := y - sign (y) . c*

- pour la réduction d'argument **positive**:

si *y < 0* ***alors*** *y := y + c*

A titre d'exemple, nous donnons l'algorithme obtenu en ajoutant la réduction d'argument par la méthode que nous venons de présenter à l'algorithme de calcul de l'exponentielle étudié auparavant, qui convergeait sur un intervalle contenant [0 , ln (4)].

Cet algorithme utilise la réduction **positive** d'argument.

On suppose que l'on désire calculer l'exponentielle de la variable t avec une précision relative de l'ordre de 2^{-p}. On supposera également les constantes $e_i = \ln (1 + 2^{-i})$, i=1 ... p et ln(4) tabulées.

```
Début
  z := t ;
  compteur := 0 ;
  K := E(z) + 1 ;
  pour i:= K jusqu'à 0 pas -1 faire
     début
       compteur := compteur + sign (z) .2 i;
       z := z - sign(z). ln(4).2 i
     fin ;
  si z < 0 alors
     début
       compteur := compteur - 1 ;
       z := z + ln (4)
     fin ;
  x := 1 ;
  pour i := 0 jusqu'à p faire
     si  z ≥ e i alors
           début
             z := z - e i ;
             x := x + x . 2 -i
           fin ;

  x := x . 2 2.compteur
Fin.
```

Après éxécution de cet algorithme, x est égal à $e^t (1 + \varepsilon)$, avec $|\varepsilon| \leq 2^{-p}$.

B - Extension à des bases de numération différentes de 2

Nous allons maintenant rechercher un algorithme matériel de réduction d'argument qui puisse s'appliquer à des systèmes travaillant dans une autre base de numération que la base 2, ce qui est fréquent tant sur de gros systèmes, qui calculent souvent en base 16, que sur des calculatrices de poche ou de petits ordinateurs "personnels", qui travaillent généralement en base 10 pour éviter les conversions décimal/binaires apparaissant lors des entrées/sorties.

Nous supposerons donc que nous calculons en base B, et que nous désirons effectuer une réduction **additive** d'argument, positive ou symétrique. Nous supposerons également que cette réduction doit se faire par additions/soustractions d'une constante positive c, qui vérifie :

$$B^{N-1} \leq c < B^{N}$$

Nous définirons la fonction :

$$E(x) = \lfloor Log_{B} |x| \rfloor$$

Là encore, cette fonction E peut s'obtenir immédiatement soit à partir de l'**exposant** de la représentation en virgule flottante de x, soit à partir de l'**emplacement du premier chiffre différent de zéro à partir des poids forts** de la représentation en virgule fixe de x.

Considérons l'algorithme suivant:

```
Début
  y := x ;
  K := E(x) - N + 2 ;
  pour i := K jusqu'à 0 pas -1 faire
       pour j := 1 jusqu'à (B-1) faire
            y := y - sign (y) . c . B^i
Fin.
```

Tout comme dans la partie consacrée aux algorithmes de réduction d'argument en base 2, nous pouvons montrer par récurrence que la valeur finale de la variable y vérifie:

$$-c \leq y \leq +c$$

Mais ce résultat peut être déduit immédiatement de la théorie exposée au chapitre **III** : en effet, l'algorithme que nous venons de présenter n'est rien d'autre que **l'algorithme bidirectionnel additif** appliqué à la **base discrète d'ordre 1** (e_n), où

$$e_n = B^K - [N/(B-1)]$$

(La suite e_n est obtenue en répétant (B-1) fois les termes de la suite (B^{K-n})).

L'éxécution de cet algorithme requiert K.(B-1) additions ou soustractions, le temps nécessaire à la réduction de l'argument x en base B est donc de l'ordre de:

$$ln(X).B/ln(B).$$

Conclusion: Les algorithmes de réduction d'argument présentés ici rendent possible la réalisation **matérielle** et non plus seulement **logicielle** de cette opération, ce qui doit assurer un **gain considérable de temps**, ainsi qu'une **souplesse d'utilisation** accrue, puisque le programmeur n'aura plus à se soucier de savoir si ses variables sont bien dans le domaine de convergence des algorithmes fournis par le concepteur du circuit de calcul utilisé.

Ces algorithmes seront implémentés, en base 2, dans le coprocesseur **FELIN** (voir chapitre **VI**) qui est en cours de conception.

Chapitre VI

Le coprocesseur FELIN

Le coprocesseur FELIN.

Introduction : Les possibilités croissantes d'intégration sur silicium autorisent désormais l'implémentation matérielle d'algorithmes de plus en plus complexes.

En particulier, l'arithmétique à virgule flottante, qui était autrefois entièrement gérée par logiciel, est de plus en plus souvent sujette à une implémentation matérielle.

Les microprocesseurs, qui sont d'emplois très généraux, ne peuvent traiter des fonctions appartenant à des domaines trop particuliers (gestion de terminaux graphiques, calcul numérique, etc...). Cette tâche est dévolue à des circuits spécialisés, travaillant en parallèle avec le processeur principal, recevant les mêmes informations que lui, mais n'exécutant que les instructions qui les concernent; ces circuits sont appelés **coprocesseurs** , ou EPU (Extended Processing Units).

Dans le domaine du calcul numérique en virgule flottante, certains coprocesseurs, comme le futur Z 8070 de Zilog (coprocesseur du microprocesseur Z 8000), n'effectuent que les opérations arithmétiques et les conversions de format. D'autres, comme le coprocesseur INTEL 8087 des microprocesseurs INTEL 8086 et 8088, calculent de surcroît certaines fonctions élémentaires, sans toutefois effectuer la réduction d'argument.

Ces circuits traitent en général trois formats différents de nombres réels à virgule flottante, en base 2, standardisés par la **norme IEEE**. Ces différents formats sont:

- La **simple précision** .

Les nombres sont écrits sur 32 bits, répartis comme suit:

- 1 bit de signe
- 23 bits de mantisse

(le premier "1" de la mantisse est **implicite** : il n'est pas

stocké, ce qui permet de gagner 1 bit en précision)
- 8 bits d'exposant.

- La **double précision** .

Les nombres sont écrits sur 64 bits, répartis comme suit:
- 1 bit de signe
- 52 bits de mantisse

(le premier "1" de la mantisse est **implicite**)
- 11 bits d'exposant.

- La **précision étendue** .

Les nombres sont écrits sur 80 bits, répartis comme suit:
- 1 bit de signe
- 63 bits de mantisse

(le premier "1" de la mantisse est **explicite**)
- 16 bits d'exposant.

Notre circuit **FELIN** (Fonctions ELémentaires INtégrées), fruit de la collaboration entre les équipes d'algorithmique mathématique et d'architecture des ordinateurs du laboratoire TIM3, sera un coprocesseur virgule flottante du microprocesseur MOTOROLA MC68000.

Il sera réalisé en technologie **CMOS à deux couches d'aluminium** : la technologie CMOS présente l'avantage de consommer moins de courant électrique, ce qui entraîne une **moindre dissipation de chaleur** par effet Joule; quant à la présence d'une deuxième couche d'aluminium, elle facilite grandement les connexions.

Ce circuit acceptera les trois formats du standard IEEE et effectuera, outre les 4 opérations arithmétiques de base et les diverses conversions de format, le calcul, **réduction d'argument comprise** , des fonctions suivantes:

- racine carrée, carré
- sinus, cosinus, tangente
- arctangente, arcsinus, arccosinus
- logarithme et exponentielle, dans les bases 2, *e* et 10
- élévation d'un nombre à une puissance quelconque.
- sinus, cosinus et tangente hyperboliques, ainsi que leurs fonctions réciproques.

Une partie de ces fonctions constituera un "noyau", calculé directement à partir des algorithmes présentés dans cette thèse, il s'agit des fonctions:

- carré, racine carrée
- sinus, cosinus, arctangente
- logarithme et exponentielle en base *e*

Les autres fonctions seront évaluées à partir des précédentes, en utilisant des relations comme:

$$Argth\ (x) = (Ln\ ((1+x)/(1-x)))/2$$

Architecture générale du circuit.

FELIN sera constitué de deux parties:

- une **machine à calcul**, chargée de l'exécution proprement dite des algorithmes, et tout particulièrement du

calcul d'expressions de la forme

$$A \pm B \cdot 2^{-i}$$

car on peut constater aisément que, en base 2, tous nos

algorithmes peuvent se ramener à une séquence de calculs de ce type.

- une **machine à format**, chargée des entrées/sorties, des conversions de format, du traitement des exceptions et du **contrôle** de tout le circuit (décodage des instructions, séquencement des opérations, etc...).

Format interne choisi pour les calculs.

Le grand avantage de la représentation des nombres réels en virgule flottante réside dans son aptitude à traiter, avec une précision relative à peu près constante, de très grands et de très petits nombres.

Toutefois, ce mode de représentation revêt également un inconvénient non négligeable: il rend les additions et les soustractions nettement plus coûteuses, tant en **temps** qu'en **surface de silicium** que dans la représentation en virgule fixe.

On s'en persuadera en examinant quelle est la marche à suivre pour effectuer une addition en virgule flottante:

1) il faut tout d'abord **aligner** les mantisses, c'est à dire se ramener, pour les deux opérandes, à un **même exposant**, par **décalage à droite** de la mantisse du plus petit nombre (en valeur absolue), d'une quantité égale à la différence entre les deux exposants.

2) On effectue ensuite l'addition des deux mantisses.

3) On **renormalise** le résultat: cette opération consiste à **décaler** sa mantisse vers la gauche de manière à ce que son premier bit non nul apparaisse tout à gauche (cas des représentations avec premier "1" explicite) ou disparaisse (cas des représentations avec premier "1" implicite). puis à **décrémenter** son exposant d'une quantité égale à celle dont on à décalé la mantisse.

Nos algorithmes nécessitant beaucoup d'additions et de soustractions, pour éviter la perte de temps due à ces trois étapes, nous avons opté pour une **représentation interne en virgule fixe sur 120 bits**, le bit de poids le plus fort correspondant à 2^{19}.

Toutefois, il convient alors de traiter à part, au niveau de la machine à format, le cas des petits ou des grands nombres, qui ne peuvent se représenter en virgule fixe. Ceci se fait par une prise en charge rigoureuse des **cas d'exception** , comme nous allons le constater au cours du prochain paragraphe.

Traitement des exceptions.

Il s'agit ici, en plus de signaler les cas d'erreur théorique ou d'overflow/underflow, de traiter directement les nombres trop grands ou trop petits pour être représentés avec précision dans le format en virgule fixe de la machine à calcul. Nous donnons ci-après quelques exemples de ce traitement.

a - traitement des erreurs théoriques.

Il s'agit de la partie la plus facile à mettre en place: on interdit les calculs n'ayant aucun sens mathématique, c'est à dire:

- La division par zéro.
- La tangente d'un nombre de la forme $\pi/2 + k\pi$.
- La racine carrée d'un nombre strictement négatif.
- Le logarithme d'un nombre négatif ou nul.

b - cas d'overflow/underflow.

Lors du calcul de l'exponentielle d'un nombre dont l'exposant est supérieur ou égal à 14, il y aura overflow si ce nombre est positif, et underflow s'il est négatif. Cette valeur critique se trouve en considérant que, puisqu'en précision étendue, les exposants sont codés sur 16 bits, on ne peut calculer l'exponentielle d'un nombre dont la valeur absolue est supérieure à $ln(2^{32768})$. (N.B. : $32768 = 2^{16}$).

c - traitement des grands nombres.

Ce problème ne se pose pas dans le cas de fonctions telles que le logarithme ou la racine carrée. Par exemple, pour calculer le logarithme d'un nombre écrit en virgule flottante, on calcule le logarithme de sa mantisse, et on ajoute à ce résultat l'exposant multiplié par le logarithme de 2.

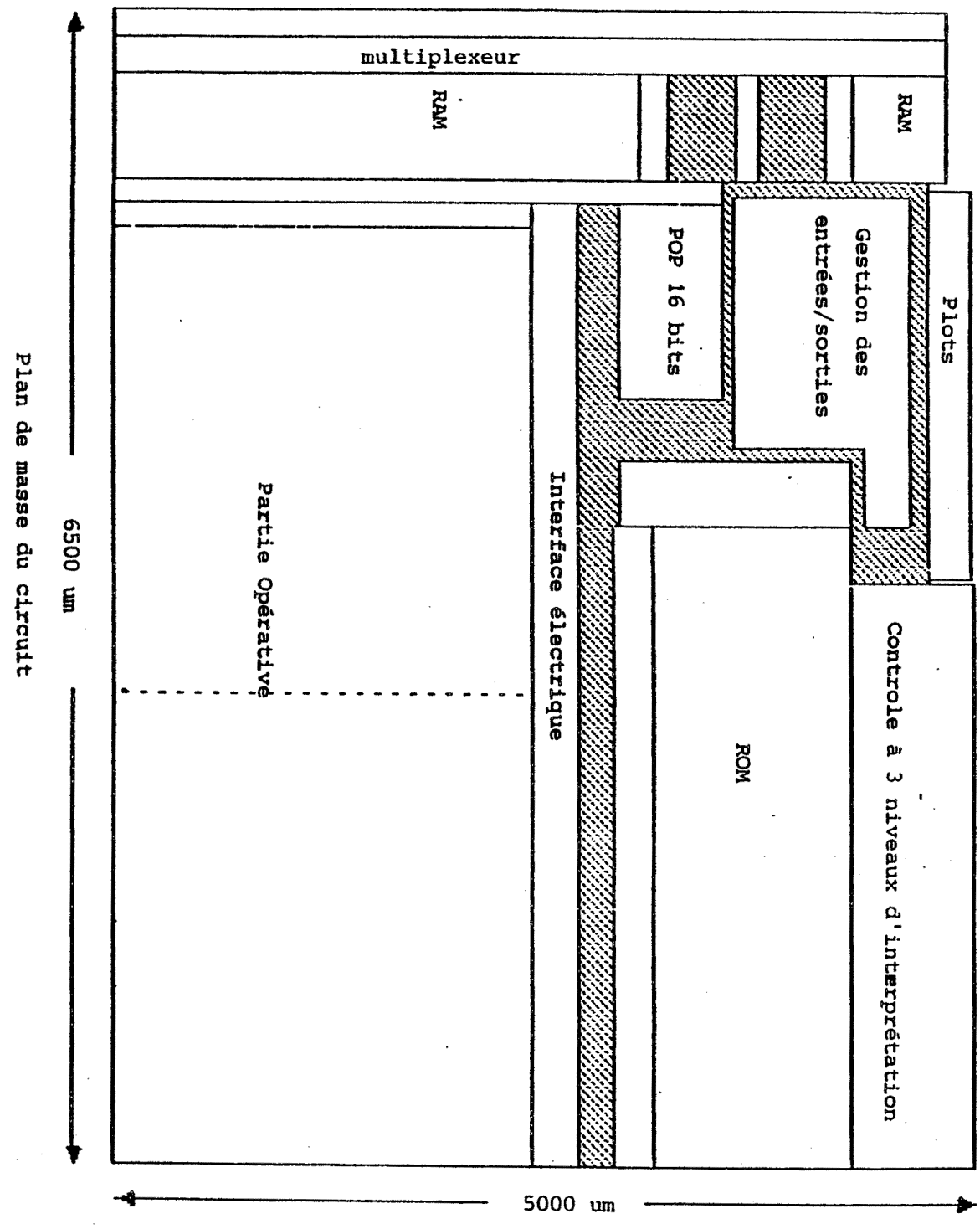
Pour les fonctions sinus, cosinus et tangente, il suffit de réaliser que si x est un nombre très grand devant $\pi/2$, il est clair que, bien qu'elles soient **théoriquement définies**, les fonctions trigonométriques sont **incalculables** en x, car une faible imprécision relative sur x peut entrainer une imprécision relative **arbitrairement grande, voire infinie**, sur le résultat du calcul.

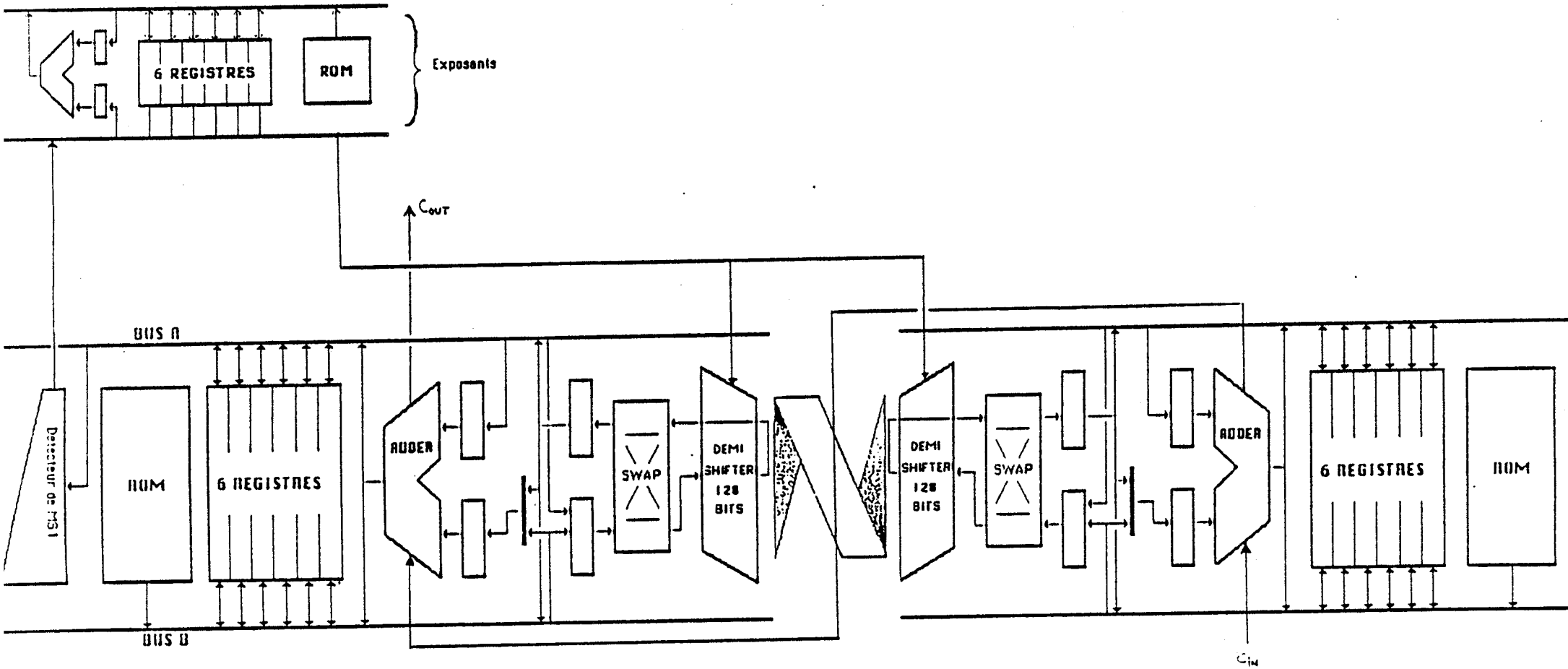
On résoud ce problème en refusant de calculer les rapports trigonométriques de nombres supérieurs à 2^{20}.

d - traitement des petits nombres.

Là, la méthode employée est triviale: on se ramène à des approximations au **premier ordre** des fonctions calculées. Par exemple, si l'exposant de x est inférieur ou égal à -32, on remplace sin (x) par x, cos (x) par 1 et e^x par 1 + x.

Les figures suivantes présentent le plan de masse du circuit et le schéma de la machine à calcul, dans leur version la plus récente au moment de la frappe de cette thèse.


multiplexeur
RAM
RAM
Plots
Gestion des entrées/sorties
POP 16 bits
Interface électrique
Partie Opérative
ROM
Controle à 3 niveaux d'interprétation
6500 um
5000 um
Plan de masse du circuit



Partie opérative du circuit.

CONCLUSION

Après avoir fait le point sur les méthodes **logicielles** courament utilisées pour le calcul des fonctions élémentaires, nous avons abordé dans cette thèse l'étude des méthodes **matérielles** , étude imposée par l'essor de la technologie dans ce domaine.

Dans le chapître **IV**, nous donnons plusieurs algorithmes de calcul, mais la partie qui nous semble la plus importante est l'étude menée au chapître **III**, qui, nous l'espérons, devrait permettre de trouver encore d'autres algorithmes.

Notre coprocesseur FELIN, qui utilise les méthodes présentées dans cette thèse (en particulier celles de réduction d'argument), et qui est le fruit d'un important travail d'équipe, devrait permettre de concilier efficacement les exigences souvent contradictoires de rapidité et de précision.

ANNEXE 1

Constantes $\text{Arctg } 2^{-1}$

et $\ln(1 + 2^{-1})$

(Données en octal)

i	Arctan 2^{-i} (Octal)
0	6.2207 73250 42055 06043 23046 14612 E-1
1	3.5530 63405 30335 51732 12677 44213 E-1
2	1.7533 35374 45440 15654 25333 43636 E-1
3	7.7526 72465 26057 33343 13104 57142 E-2
4	3.7752 55667 13176 75161 55454 47445 E-2
5	1.7775 25335 65135 42225 02357 16567 E-2
6	7.7775 25267 35562 46533 57474 30531 E-3
7	3.7777 52525 56735 22713 20030 43243 E-3
8	1.7777 75252 53356 73345 13542 25373 E-3
9	7.7777 75252 52673 56724 56246 53365 E-4
10	3.7777 77525 25255 67356 67122 71320 E-4
11	1.7777 77752 52525 33567 35651 34513 E-4
12	7.7777 77752 52525 26735 67355 62456 E-5
13	3.7777 77775 25252 52556 73567 35227 E-5
14	1.7777 77777 52525 25253 35673 56733 E-5
15	7.7777 77777 52525 25252 67356 73567 E-6
16	3.7777 77777 75252 52525 25567 35673 E-6
17	1.7777 77777 77525 25252 52533 56735 E-6
18	7.7777 77777 77525 25252 52526 73567 E-7
19	3.7777 77777 77752 52525 25252 55673 E-7
20	1.7777 77777 77775 25252 52525 25335 E-7
21	7.7777 77777 77775 25252 52525 25267 E-10
22	3.7777 77777 77777 52525 25252 52525 E-10
23	1.7777 77777 77777 75252 52525 25252 E-10
24	7.7777 77777 77777 75252 52525 25252 E-11
25	3.7777 77777 77777 77525 25252 52525 E-11
26	1.7777 77777 77777 77752 52525 25252 E-11
27	7.7777 77777 77777 77752 52525 25252 E-12
28	3.7777 77777 77777 77775 25252 52525 E-12
29	1.7777 77777 77777 77777 52525 25252 E-12
30	7.7777 77777 77777 77777 52525 25252 E-13
31	3.7777 77777 77777 77777 75252 52525 E-13
32	1.7777 77777 77777 77777 77525 25252 E-13
33	7.7777 77777 77777 77777 77525 25252 E-14
34	3.7777 77777 77777 77777 77752 52525 E-14
35	1.7777 77777 77777 77777 77775 25252 E-14
36	7.7777 77777 77777 77777 77775 25252 E-15
37	3.7777 77777 77777 77777 77777 52525 E-15
38	1.7777 77777 77777 77777 77777 75252 E-15
39	7.7777 77777 77777 77777 77777 75252 E-16
40	3.7777 77777 77777 77777 77777 77525 E-16
41	1.7777 77777 77777 77777 77777 77752 E-16
42	7.7777 77777 77777 77777 77777 77752 E-17
43	3.7777 77777 77777 77777 77777 77775 E-17
44	1.7777 77777 77777 77777 77777 77777 E-17

i	$\ln(1 + 2^{-i})$ (Octal)
0	5.4271 02775 75071 73632 57117 07316 E-1
1	3.1746 21754 57714 11374 53321 53546 E-1
2	1.6217 75743 63246 42065 41135 40367 E-1
3	7.4234 07334 25362 71364 75241 77774 E-2
4	3.7024 30300 04261 24630 27631 13426 E-2
5	1.7602 46607 17014 60011 47617 60236 E-2
6	7.7402 50541 76003 70747 63666 47232 E-3
7	3.7700 25153 04063 61077 03355 10071 E-3
8	1.7760 02521 26102 74011 20206 55264 E-3
9	7.7740 02523 25420 71474 24743 50251 E-4
10	3.7770 00252 42530 41757 00232 02431 E-4
11	1.7776 00025 24652 61040 63170 53305 E-4
12	7.7774 00025 25052 54210 27360 04670 E-5
13	3.7777 00002 52515 25304 20714 63612 E-5
14	1.7777 60000 25252 12526 10417 56740 E-5
15	7.7777 40000 25252 32525 42104 06314 E-6
16	3.7777 70000 02525 24252 53042 10273 E-6
17	1.7777 76000 00252 52465 25261 04207 E-6
18	7.7777 74000 00252 52505 25254 21041 E-7
19	3.7777 77000 00025 25251 52525 30421 E-7
20	1.7777 77600 00002 52525 21252 52610 E-7
21	7.7777 77400 00002 52525 23252 52542 E-10
22	3.7777 77700 00000 25252 52425 25253 E-10
23	1.7777 77760 00000 02525 25246 52523 E-10
24	7.7777 77740 00000 02525 25250 52525 E-11
25	3.7777 77770 00000 00252 52525 15252 E-11
26	1.7777 77776 00000 00025 25252 52125 E-11
27	7.7777 77774 00000 00025 25252 52325 E-12
28	3.7777 77777 00000 00002 52525 25242 E-12
29	1.7777 77777 60000 00000 25252 52524 E-12
30	7.7777 77777 40000 00000 25252 52525 E-13
31	3.7777 77777 70000 00000 02525 25252 E-13
32	1.7777 77777 76000 00000 00252 52525 E-13
33	7.7777 77777 74000 00000 00252 52525 E-14
34	3.7777 77777 77000 00000 00025 25252 E-14
35	1.7777 77777 77600 00000 00002 52525 E-14
36	7.7777 77777 77400 00000 00002 52525 E-15
37	3.7777 77777 77700 00000 00000 25252 E-15
38	1.7777 77777 77760 00000 00000 02525 E-15
39	7.7777 77777 77740 00000 00000 02525 E-16
40	3.7777 77777 77770 00000 00000 00252 E-16
41	1.7777 77777 77776 00000 00000 00025 E-16
42	7.7777 77777 77774 00000 00000 00025 E-17
43	3.7777 77777 77777 00000 00000 00002 E-17
44	1.7777 77777 77777 60000 00000 00000 E-17
45	7.7777 77777 77777 40000 00000 00000 E-20
46	3.7777 77777 77777 70000 00000 00000 E-20
47	1.7777 77777 77777 76000 00000 00000 E-20
48	7.7777 77777 77777 74000 00000 00000 E-21
49	3.7777 77777 77777 77000 00000 00000 E-21
50	1.7777 77777 77777 77600 00000 00000 E-21
51	7.7777 77777 77777 77400 00000 00000 E-22
52	3.7777 77777 77777 77700 00000 00000 E-22
53	1.7777 77777 77777 77760 00000 00000 E-22
54	7.7777 77777 77777 77740 00000 00000 E-23
55	3.7777 77777 77777 77770 00000 00000 E-23
56	1.7777 77777 77777 77776 00000 00000 E-23
57	7.7777 77777 77777 77774 00000 00000 E-24
58	3.7777 77777 77777 77777 00000 00000 E-24
59	1.7777 77777 77777 77777 60000 00000 E-24
60	7.7777 77777 77777 77777 40000 00000 E-25

i	Ln (1 + 2^{-i}) (Octal)
61	3.7777 77777 77777 77777 70000 00000 E-25
62	1.7777 77777 77777 77777 76000 00000 E-25
63	7.7777 77777 77777 77777 74000 00000 E-26
64	3.7777 77777 77777 77777 77000 00000 E-26
65	1.7777 77777 77777 77777 77600 00000 E-26
66	7.7777 77777 77777 77777 77400 00000 E-27
67	3.7777 77777 77777 77777 77700 00000 E-27
68	1.7777 77777 77777 77777 77760 00000 E-27
69	7.7777 77777 77777 77777 77740 00000 E-30
70	3.7777 77777 77777 77777 77770 00000 E-30
71	1.7777 77777 77777 77777 77776 00000 E-30
72	7.7777 77777 77777 77777 77774 00000 E-31
73	3.7777 77777 77777 77777 77777 00000 E-31
74	1.7777 77777 77777 77777 77777 60000 E-31
75	7.7777 77777 77777 77777 77777 40000 E-32
76	3.7777 77777 77777 77777 77777 70000 E-32
77	1.7777 77777 77777 77777 77777 76000 E-32
78	7.7777 77777 77777 77777 77777 74000 E-33
79	3.7777 77777 77777 77777 77777 77000 E-33
80	1.7777 77777 77777 77777 77777 77600 E-33
81	7.7777 77777 77777 77777 77777 77400 E-34
82	3.7777 77777 77777 77777 77777 77700 E-34
83	1.7777 77777 77777 77777 77777 77760 E-34
84	7.7777 77777 77777 77777 77777 77740 E-35
85	3.7777 77777 77777 77777 77777 77770 E-35
86	1.7777 77777 77777 77777 77777 77776 E-35
87	7.7777 77777 77777 77777 77777 77774 E-36
88	3.7777 77777 77777 77777 77777 77777 E-36

ANNEXE 2

Les programmes de la bibliothèque FELIMAG

Sinus/Cosinus/Tangente
(Module sincos)

Nous présentons ici deux versions différentes: la première est celle dont nous avons donné les résultats de tests précedemment, elle utilise la méthode de rattrappage de l'erreur d'arrondi. La deuxième, un peu moins précise mais légèrement plus rapide utilise le résultat de notre programme "Bestpol".

Ces deux versions sont bâties à partir d'un approximant théorique du à W. Cody (19).

Exponentielle/sh/ch/Logarithme
Argth
(Modules hyperb et ln)

Nous employons pour l'exponentielle une fraction rationnelle due à W.Cody. Le logarithme utilise la méthode de rattrappage de l'erreur d'arrondi. Les fonctions sh, ch et argth sont obtenues par:

$$sh(x) = 0.5 * (exp(x) - 1.0/exp(x))$$
$$ch(x) = 0.5 * (exp(x) + 1.0/exp(x))$$
$$argth(x) = 0.5 * ln((1.0 + x)/(1.0 - x))$$

Arctangente
Intpower/power/gamma
(Modules Arctan et Speciale)

La fonction Arctan est obtenue en utilisant la méthode de rattrappage de l'erreur d'arrondi, elle est basée sur un approximant théorique donné par W. Cody.

La fonction Intpower (calcul de x^n , n entier) est obtenue par une méthode équivalente à la méthode de la "chaîne chinoise", elle utilise récursivement la propriété:

$$x^n = (x^{\lfloor n/2 \rfloor})^2 * p \quad , p \in \{ 1 , x \}$$

La fonction power calcule x^y chaque fois que cette expression a un sens. Elle est plus précise que l'appel direct de exp (y * ln (x)) bien qu'elle utilise elle aussi les fonctions ln et exp de FELIMAG.

La fonction gamma provient d'une approximation tirée de Hart&Cheney (44). La réduction d'argument utilise la propriété:

$$\Gamma(1+X) = x\,\Gamma(x)$$

ERF/ERFC/GAUSS/ARCGAUSS
(Module stat 1)

Ce module est le premier module de fonctions statistiques de FELIMAG. D'autres modules sont en préparation, donnant les fonctions de répartition des principales lois statistiques, continues ou discrètes.

Ne pouvant effectuer de réduction d'argument, on se trouve réduit à approximer la fonction erfc par deux fractions rationnelles différentes, l'une étant utilisée sur [0,8] et l'autre sur [8,25]. Ces fractions sont tirées de Hart&Cheney (44).

Pour des valeurs négatives de l'argument, on utilise:

erfc (-x) = 2 - erfc (x).

Si x est supérieur à 25, il y a overflow.

La fonction erreur erf est obtenue par: erf = 1 - erfc (x).
(Pour limiter l'erreur de calcul, on remplace cette expression par:

$2x.\pi^{-1/2}$ si |x| est inférieur à 10^{-8}.
1 si x > 7
-1 si x < -7).

La fonction de répartition de la loi de Gauss est obtenue par le biais de la relation:

$$\text{Gauss}(x) = 0.5 * \text{erfc}(-x * 2^{-1/2})$$

Et sa fonction réciproque arcgauss est obtenue en utilisant la méthode de Newton.

```
module sincos5;

(* J.M. Muller - Logiciel FELIMAG Version 5.0                Mars 1984 *)

function sin (x:real):real;

  const c1=3.1416015625;
        c2=-8.90891020676154E-06;     (* c1 + c2 = pi *)
        pi=3.141592653589793;

  var y,r:real;
      n:integer;
      negatif:boolean;

 begin
   if x<0.0 then
     begin
       negatif:=true;
       x:=-x;
     end
      else negatif:=false;
   n:=round (x/pi);
   if n mod 2 = 1 then negatif:= not negatif;
   x:= (x-n*c1)-n*c2;     (* reduction soignee de l'argument *)
   y:=x*x;
   r:=(((((((2.72047909578888E-15*y-7.64291780689105E-13)*y
           +1.60589364903716E-10)*y-2.50521067982746E-8)*y
           +2.75573192101528E-6)*y-1.98412698412018E-4)*y
           +8.30078125E-3)*y-1.66015625E-1)
           +3.25520833331650E-5*y-6.51041666666665E-4;
   (* les 2 derniers termes sont ceux de la methode de compensation d'erreur *
   y:=r*y*x+x;
    if negatif then sin:=-y else sin:=y;
 end;

 function cos (x:real):real;

  const c1=3.1416015625;
        c2=-8.90891020676154E-06;   (* c1 + c2 = pi *)
        pisur2=1.570796326794896;
        pi=3.14159265358979;

  var y,xn,r:real;
      n:integer;
 begin
   if x = 0.0 then cos := 1.0
   else
   begin
     x:=abs (x);
     y:=x+pisur2;
     n:=round (y/pi);
     xn:=n-0.5;
     x:= (x-xn*c1)-xn*c2;
     y:=x*x;
     r:=(((((((2.72047909578888E-15*y-7.64291780689105E-13)*y
             +1.60589364903716E-10)*y-2.50521067982746E-8)*y
             +2.75573192101528E-6)*y-1.98412698412018E-4)*y
             +8.30078125E-3)*y-1.66015625E-1)
             +3.25520833331650E-5*y-6.51041666666665E-4;
       (* compensation d'erreur de degre 1 *)
```

```
    y:=r*y*x+x;
    if (n mod 2 = 1) then cos:=-y else cos:=y;
  end;
end;

function tg (x:real):real;
 begin
   tg:=sin(x)/cos(x);
 end;

modend.
```

```
module sincos5;

(* J.M. Muller - Logiciel FONCTELEM Version 5.0                Mars 1984 *)

function sin (x:real):real;

  const c1=3.1416015625;
        c2=-8.90891020676154E-06;     (* c1 + c2 = pi *)
        pi=3.141592653589793;

  var y,r:real;
      n:integer;
      negatif:boolean;

 begin
   if x< 0.0 then
     begin
       negatif:=true;
       x:=-x;
     end
      else negatif:=false;
   n:=round (x/pi);
   if n mod 2 = 1 then negatif:= not negatif;
   x:= (x-n*c1)-n*c2;     (* reduction soignee de l'argument *)
   y:=x*x;
   r:=((((((2.72047909578888E-15*y-7.64291780689105E-13)*y
          +1.60589364903713E-10)*y-2.50521067982746E-8)*y
          +2.75573192101528E-6)*y-1.98412698412017E-4)*y
          +8.33333333333321E-3)*y-1.66666666666669E-1) ;
   y:=r*y*x+x;
    if negatif then sin:=-y else sin:=y;
 end;

 function cos (x:real):real;

  const c1=3.1416015625;
        c2=-8.90891020676154E-06;  (* c1 + c2 = pi *)
        pisur2=1.570796326794896;
        pi=3.14159265358979;

  var y,xn,r:real;
      n:integer;
 begin
   if x = 0.0 then cos := 1.0
   else
   begin
     x:=abs (x);
     y:=x+pisur2;
     n:=round (y/pi);
     xn:=n-0.5;
     x:= (x-xn*c1)-xn*c2;
     y:=x*x;
   r:=((((((2.72047909578888E-15*y-7.64291780689105E-13)*y
          +1.60589364903713E-10)*y-2.50521067982746E-8)*y
          +2.75573192101528E-6)*y-1.98412698412017E-4)*y
          +8.33333333333321E-3)*y-1.66666666666669E-1) ;
     y:=r*y*x+x;
     if (n mod 2 = 1) then cos:=-y else cos:=y;
   end;
 end;
```

```
function tg (x:real):real;
 begin
   tg:=sin(x)/cos(x);
 end;

modend.
```

```
module  hyperb5;

(* J.M. Muller  Logiciel FELIMAG version 5.0              Mars 1984 *)

  function exp (x:real):real;

   const c1=0.693359375;
         c2=-2.12194440054691E-4;      (* c1 + c2 = ln (2) *)
         unsurlog2=1.442695040888963;

   var i:integer;
       z,p,d,facteur,xq:real;
       q:integer;

 begin
    q:=round (x*unsurlog2);
    xq:=q;
    x:=(x-xq*c1)-xq*c2;  (* reduction soignee d'argument *)
    facteur:=1.0;
    if q>0 then for i:=1 to q do facteur:= 2.0*facteur
       else if q<0 then for i:=1 to (-q) do facteur:=0.5*facteur;
    z:=x*x;
    p:=((1.65203300268279E-5*z+6.94360001511793E-3)*z+2.5E-1)*x;
    d:=(4.95862884905441E-4*z+5.55538666969001E-2)*z+5.0E-1;
   exp:= facteur + 2.0*p/(d-p)*facteur;
 end;


function sh (x:real):real;
 var z:real;
  begin
   if abs(x)<1.0E-08 then sh:=x
   else
   begin
    z:=exp (x);
    sh:=0.5*(z-1.0/z);
   end;
  end;

function ch (x:real):real;
 var z:real;
  begin
    z:=exp (x);
    ch:=0.5*(z+1.0/z);
  end;

modend.
```

```
module ln5;

(* J.M. Muller - Logiciel FELIMAG version 5.0          Mars 1984 *)

function ln (x:real):real;

  const c1=6.93359375E-1;
        c2=-2.12194440054691E-4;  (* c1 + c2 = ln (2) *)

  var i,p:integer;
      y,r1,r2:real;

   begin
    if x<=0.0 then
        begin
          writeln ('ERREUR: LOGARITHME D''UN REEL NEGATIF');
          ln:=-1.0E+300;
        end
    else
   begin
     p:=0;
     while x<7.07106E-1 do
        begin
          x:=x*2.0;
          p:=p-1;
        end;
     while x>1.4142136 do
       begin
         x:=x*5.0E-1;
         p:=p+1;
       end;
 x:=(x - 1.0)/(x+1.0);
 y:=x*x;
    r1:=(((((1.6948212488E-1*y+1.811136267967E-1)*y
            +2.2223823332791E-1)*y+2.85714091590488E-1)*y
            +4.00000001206045E-1)*y+6.66015625E-1)*y
            +2.0;
    r2:=6.51041663366089E-4*y+2.61007E-15;
    ln:=(p*c2+(r1+r2)*x)+p*c1;  (* compensation d'erreur *)
  end;
end;


function argth (w:real):real;
 begin
   if abs (w)>=1.0 then
     begin
       writeln ('ERREUR: ARGTH D''UN REEL DE MODULE SUPERIEUR A 1');
       argth:=1.0E+300;
     end
    else if abs (w)<1.0E-007 then argth:=w
    else argth:=5.0E-1*ln ((1.0+w)/(1.0-w));
 end;

modend.
```

```
module arctan5;

(* J.M. Muller  Logiciel FELIMAG version 5.0          Mars 1984 *)

function arctan (x:real):real;

  var g,r,f:real;
      supl:boolean;

   begin
     f:=abs (x);
     if f > 1.0 then
        begin
          supl:=true;
          f:=1.0/f;
        end
         else supl:=false;
    g:=f*f;
     r:=f*((((((9.76272159171763E-2*g + 1.13221594116765E+1)*g
             +1.92579201448156E+2)*g+ 1.11412907284553E+3)*g
             +2.76171982461388E+3)*g+ 3.03107459561151E+3)*g
             +1.20974700175809E+3)
     /((((((g+3.99178842486538E+1)*g+ 4.23071646480905E+2)*g
             +1.82160033929185E+3)*g+ 3.66454495632837E+3)*g
             +3.43432359619754E+3)*g+ 1.20974700175809E+3);
      if supl then r:= (1.5703125 - r) +4.8382679489661923E-4;
       (* compensation d'erreur sur la soustraction *)
    if x<0.0 then  arctan:=-r
     else arctan:=r;
end;

modend.
```

```
module speciale5;

(* J.M. Muller - Logiciel FELIMAG version 5.0          Mars 1984 *)

function intpower (x:real;n:integer):real;

(* calcule une puissance entiere d'un nombre *)

 var p:integer;
     res:real;

  begin
    if n<0 then intpower:=1.0/intpower (x,-n)
    else if n=0 then intpower:=1.0 else if n=1 then intpower:=x
    else
      begin
        p:=n div 2;
        res:=sqr (intpower (x,p));
        if 2*p <> n then intpower:=res*x else intpower:=res;
      end;
  end;

function power (x,y:real):real;

(* calcule x puissance y des que cette expression est definie *)

 var n:integer;
     r:real;

begin
  n:=trunc (y);
  r:=y - n;
  if (x<0.0) and (r<>0.0) then
       begin
         writeln ('FONCTION POWER INDEFINIE');
       end
  else if r=0.0 then power:=intpower (x,n)
  else power:=intpower (x,n)* exp (r*ln(x));
end;


function gamma (x:real):real;
begin
 if abs (x)>171.0 then
   begin
    writeln (x:7:2,': ARGUMENT TROP GRAND POUR LA FONCTION GAMMA');
    gamma:=1.0E+307;
   end
 else
  if (x<=0.0) and (round(x)=x) then
    begin
      writeln ('GAMMA N''EST PAS DEFINIE POUR LES ENTIERS NEGATIFS OU NULS');
      gamma:=1.0E+307;
    end
   else
 begin
    if x>=3.0 then gamma:=(x-1.0)*gamma(x-1.0)
    else if x<2.0 then gamma:=gamma(x+1.0)/x
     else
       begin
```

```
            x:=x-2.0;
            gamma:=((((((-6.74495072459253*x-5.010869375279010E+1)*x
                    -4.39330444060026E+2)*x-2.00852740130728E+3)*x
                    -8.76271029785215E+3)*x-2.08868617892699E+4)*x
                    -4.23536895097441E+4)
              /(((((((x-2.30815515245801E+1)*x+1.89498234157028E+2)*x
                      -4.99028526621439E+2)*x-1.52860727377952E+3)*x
                      +9.94030741508277E+3)*x-2.98038533092566E+3)*x
                      -4.23536895097441E+4);
          end;
 end;
end;
modend.
```

```
program statl;

(* J.M. Muller  logiciel FELIMAG    Version 6.0          Decembre 1984 *)
(* Premier module de statistiques                      fonctions erreur *)

  var x,y: real;

 const munsurrac2 = -7.0710678118654752E-1;
       deuxsurracpi = 1.12837916709551;
       rac2pi = 2.50662827463100;

function ratl (x: real): real;

(* donne erfc pour x c [0 , 8] *)
    var num, den: real;
  begin
    num := (((((((5.641895867618136E-1 *x + 1.006485897490954E+1)*x
                 +8.608276221194859E+1)*x + 4.562614587060926E+2)*x
                 +1.631760268753714E+3)*x + 4.032267010830049E+3)*x
                 +6.758216964110485E+3)*x + 7.113663246954049E+3)*x
                 +3.723507981554806E+3;

    den := (((((((x + 1.783949843913955E+1)*x
                 +1.530777107503622E+2)*x + 8.176223863045440E+2)*x
                 +2.968004901482308E+3)*x + 7.542479510193475E+3)*x
                 +1.334934656128445E+4)*x + 1.580253599940204E+4)*x
                 +1.131519208185441E+4)*x + 3.723507981554807E+3;

    ratl := num/den

  end;


function rat2 (x: real): real;

(* donne erfc sur [8 , 100] *)
var num,den: real;
  begin
    num := ((((5.641895835477551E-1 *x + 1.275366644729966)*x
              +5.019049726784267)*x + 6.160209853109631)*x
              +7.409740605964742)*x + 2.978865626393993;

    den := (((((x + 2.260528520767327)*x
              +9.396034016235054)*x + 1.204895192785513E+1)*x
              +1.708144074746600E+1)*x + 9.608965327192788)*x
              +3.369075206982753;

    rat2 := num/den

  end;


function erfc (x:real):real;
  var negatif: boolean;
      tampon: real;
 begin
   if x<0 then begin
                x := -x;
                    negatif := true
```

```
                end
            else negatif := false;
        if x<8 then tampon := rat1 (x)* exp (-sqr (x))
    else  if x < 25 then tampon := rat2 (x)* exp (-sqr (x))
     else
         tampon := 1.0E-307;
    if negatif then erfc := 2 - tampon else erfc := tampon
end;

function erf (x:real):real;
  begin
    if abs (x) < 1.0E-8 then erf := deuxsurracpi*x
    else if x>7 then erf := 1 else if x<-7 then erf := -1
    else erf := 1 - erfc (x)
  end;

function gauss (x:real):real;
  begin
    gauss := 0.5 * erfc (munsurrac2*x)
  end;

function arcgauss (x:real): real;
  (* le calcul se fait par la methode de newton *)
 var i,max: integer;
     y,rayon: real;
  begin
    rayon := abs (x - 0.5);
    if abs (rayon)>=0.5 then writeln ('ERREUR: L''argument de arcgauss doit etr
 compris entre 0 et 1')
     else
       begin
         if rayon < 0.25 then max := 5
           else if rayon < 0.999 then max := 10
             else if rayon < 1.0E-6 then max := 20
              else max := 100;
         y := 0;
         for i := 1 to max do
            y := y - (gauss (y) - x)*rac2pi*exp(sqr(y)/2);
         arcgauss := y
       end
  end;
```

BIBLIOGRAPHIE

(1) M. Abramowitz and I.A. Stegun, Handbook of Mathematical Functions with formulas, graphs, and mathematical tables, Nat. Bur. Standards, Appl. Math. Series, 55, Washington D.C., 1964.

(2) H.M. Ahmed, J-M. Delosme, M. Morf, Highly concurrent computing structures for matrix arithmetic and signal processing, Computer, Jan. 1982.

(3) F. Anceau, Architecture and design of Von Neumann microprocessors, Nato advanced summer institute, July 1980.

(4) F. Anceau, Layout strategies for NMOS-CMOS VLSI, R.R. IMAG, Grenoble, France.

(5) F. Anceau, Règles de dessin pour le projet CMP, R.R. IMAG, Grenoble, France.

(6) M. Andrews and T. Mraz, Unified elementary function generator, Microprocessors and Microsystems, Vol. 2 n° 5, Oct.1978, pp 270-274.

(7) P.W. Baker, More efficient radix-2 algorithms for some elementary functions. IEEE Trans. on computers, vol. c-24 n° 11, Nov. 1975, pp 1049-1054.

(8) P.W. Baker, Suggestion for a fast binary Sine/Cosine generator, IEEE Trans. on Computers, Nov. 1976, pp 1134-1136.

(9) R.P. Brent, Multiple-precision zero-finding methods and the complexity of elementary function evaluation, Analytic Computational Complexity (Ed. by J.F. Traub), Academic Press, New York, 1975, pp 151-176.

(10) R.P. Brent, Fast multiple-precision evaluation of elementary functions, J. ACM 23, 1976, pp 242-251.

(11) R.P. Brent, A Fortran multiple-precision arithmetic package, ACM Trans. Math. Software 4, 1978, pp 57-70.

(12) R.P. Brent, Unrestricted algorithms for elementary and special functions, Information Processing 80, S.H. Lavington ed., North-Holland Publishing Comp., pp 613-619.

(13) W.S. Brown, A simple but realistic model of floating-point computation, ACM Trans. on Math. software, Vol. 7 n° 4, Dec. 1981, pp 445-480.

(14) L.B. Bushard, A minimum table size result for higher radix nonrestoring division, IEEE Trans. on Computers, Vol. c-32 n° 6, June 1983.

(15) T.H. Chan and O.H. Ibarra, On the space and time complexity of functions computable by sample programs, Siam J. Comput., Vol. 12, n° 4, Nov. 1983.

(16) T.C. Chen, Automatic computation of exponentials, logarithms, ratios and square roots. IBM J. Res. and Developpement, Vol. 16,July 1972, pp 380-388.

(17) C.W. Clenshaw, Mathematical tables, Vol. 5, Chebyshev Series for Math. functions, Her Majesty's Stationery Office, London, 1962.

(18) C.W. Clenshaw and F.W.J. Olver, Beyond floating point, J. of the ACM, Vol. 31 n° 2, April 1984, pp. 319-328.

(19) W. Cody & W. Waite, Software manual for the elementary functions, Prentice-Hall, inc.,Englewood cliffs, New-Jersey,1980.

(20) W. Cody, A Generalization of the proposed IEEE standard for floating point arithmetic, Math. & Computer sci. division, Argonne Nat. laboratory.

(21) W. Cody, Implementation and testing of function software, Ibid.

(22) W. Cody, Basic concepts for computational software, Ibid.

(23) W. Cody, The FUNPACK package of special function subroutines, TOMS1, 1975, pp 13-25.

(24) W. Cody, FUNPACK, A Package of special function subroutines, Technical memorandum No. 385, Argonne Nat. Laboratory, Argonne, Illinois, 1981.

(25) W. Cody, Performance testing of function subroutines, AFIPS Conf. Proc., Vol. 34, 1969 SJCC, AFIPS Press, Montvale, N.J.,1969, pp 759-763.

(26) W. Cody, An overview of software development for special functions, Lecture Notes in Mathematics, 506, Numerical Analysis Dundee 1975, G.A. Watson (ed.), Springer Verlag, Berlin, 1976, pp. 1-23.

(27) W. Cody, Observations on the mathematical software effort, Proc. of the 1981 army numerical analysis and computer conference.

(28) J.T. Coonen, An implementation guide to a proposed standard for floating-point arithmetic, IEEE Computer, Jan. 1980.

(29) J.M. Delosme, VLSI implementation of rotations in pseudo-euclidian spaces, proc. 1983 IEEE Int. Conf. on ASSP, Boston, April 1983, pp 927-930.

(30) J.M. Delosme, The matrix exponential approach to elementary operations, Depart. of Electrical Engineering, Yale Univ., New Haven.

(31) J.E. Gentle & W.J. Kennedy, Statistical computing, Marcel Dekker, inc. New York-Basel, 1980.

(32) B. De Lugish, A class of algorithms for automatic evaluation of certain elementary functions in a binary computer, Ph.D. dissertation, Dep. Computer sci., Univ. of illinois, Urbana, June 1970.

(33) B. Derrida, A. Gervois, Y. Pomeau, Iteration of endomorphisms on the real axis and representation of numbers. Commissariat à l' énergie atomique, Service de physique théorique, CEN Saclay.

(34) Alvin M. Despain, Fourier transform computers using CORDIC iterations, IEEE Trans. on Computers, Vol. c-23 n° 10, Oct. 1974.

(35) A.M. Despain, Pipeline and parallel-pipeline FFT Processors for VLSI implementations, IEEE Trans on Computers, Vol. c-33 n° 5, May 1984.

(36) S.W. Ellacot, On the faber transform and efficient numerical rational approximation, SIAM J. Numer. Anal., Vol. 20 n° 5, Oct. 1983.

(37) M.D. Ercegovac, Radix-16 evaluation of certain elementary functions, IEEE Trans. on Computers, Vol. c-22 n°16, June 1973.

(38) M.D. Ercegovac, A General method for evaluation of functions in a digital computer, Computer sci. dep., School of Engineering & Applied science, Univ. of California, Los Angeles, California 90024.

(39) C.T. Fike, Computational evaluation of math. functions, Prentice-Hall, Englewood cliffs, New-Jersey, 1968.

(40) W.M. Gentleman, More on algorithms that reveal properties of floating-point arithmetics units, Comm. of the ACM, Vol.17, n° 5, May 1974.

(41) G.W. Gerrity, Computer representation of real numbers, IEEE Trans. Computers, Vol. c-31 n° 8,Aug. 1982.

(42) M.H. Gutknecht and L.N. Trefethen, Real polynomial chebyshef approximation by the ctheodory-fejer method, SIAM J. Numerical Analysis, Vol. 19, n° 2, April 1982.

(43) G.H. Haviland and A. A. Tuszynsky, A CORDIC arithmetic processor chip, IEEE Trans. on Computers, Vol. c-29 n° 2, Feb. 1980.

(44) J.F. Hart, E.W. Cheney, C.L. Lawson, H.J. Maehly, C.K. Mesztenyi, J.R. Rice, H.C. Tacher, Jr. and C. Witzgall, Computer Approximations, Wiley, N.Y., 1968.

(45) J.P. Kahane & R.Salem, Ensembles parfaits et séries trigonométriques, Actualités scientifiques et industrielles 1301, Hermann Paris,1963.

(46) A. H. Karp, Exponential and logarithm by sequential squaring, IEEE Trans. on Computers, Vol. c-33, n° 5, May 1984, pp 462-464.

(47) D.E. Knuth, The art of computer programming, Vol. 2, Addison Wesley, Reading, Mass. , 1969.

(48) J. Kropa, Calculator algorithms, Math. Mag., Vol. 51 n° 2, March 1978, pp 106-109.

(49) U.V. Kulish and W.L. Miranker, Computer arithmetic in theory and practice, Acad. Press, New-York, 1981.

(50) M.A. Malcolm, Algorithms to reveal properties of floating-point arithmetic, Comm. of the A.C.M., Vol. 15 n° 11, Nov. 1972.

(51) J.D. Marasa and D.W. Matula, A simulated study of correlated error propagation in various finite-precision arithmetic, IEEE Trans. on Computers, Vol. c-22, n° 6, June 1973.

(52) C. Masse, L'iteration de Newton: convergence et chaos, thèse de troisième cycle, Université Grenoble I, Oct. 1984.

(53) D. W. Matula, Basic digit sets for radix representation, J. of the A.C.M., Vol. 29 n° 4, Oct. 1982, pp. 1131-1143.

(54) C. Mead & L. Conway, Introduction to VLSI systems, Addison-Wesley publishing Company.

(55) J.E. Meggitt, Pseudo Division and Pseudo Multiplication Processes, IBM J. of Res. and Dev., Vol. 6, April 1962, pp 210-227.

(56) B. Mendelbrot, The fractal geometry of nature, Freeman, 1982.

(57) J.M. Muller, Discrete basis and computation of elementary functions, R.R. Math. App. n° 442, Grenoble, 1984, à paraître dans IEEE Transactions on Computers.

(58) J.M. Muller, Conditionnement de fonctions et representation flottante des nombres reels, RR. Math. App. N° 453, Grenoble, 1984.

(59) J.M. Muller, A hardware algorithm for computing the complex exponential function, R.R. Math. App. n° 467, Grenoble, 1984.

(60) J.M. Muller, Algorithmes rapides pour le calcul de fonctions élémentaires, Proc. Congres National d'Analyse Numérique, Bombannes, France, 1984, pp 153-156.

(61) J.M. Muller, Une méthodologie du calcul des fonctions élémentaires, R.R. Math. App. N° 503, Grenoble, 1985.

(62) J.M. Muller, FELIMAG, Un logiciel de calcul des fonctions mathématiques de base, R.R. Math. App. N° 519, Grenoble, 1985.

(63) J.M. Muller, Hardware computation of elementary functions including range reduction, R.R. Math. App. N° 523, Grenoble, 1985.

(64) J.M. Muller, représentation des nombres réels et calcul des fonctions élémentaires, à paraître dans les annales du séminaire de théorie des nombres de Bordeaux de 1985.

(65) A. Naseem & P.D. Fisher, A modified CORDIC Algorithm, Preprint Dept. of Electrical Engineering and Systems Science, Michigan State Univ. , East Lansing, Michigan 48824.

(66) A.M. Pezechke, Vers une conception régulière des circuits intégrés à l'aide de cellules pré-caractérisées paramètrables, thèse de docteur-ingénieur, Université Paris 6, Janv. 1984.

(67) F. W. J. Olver, A new approach to error arithmetic, SIAM J. Numer. Analysis, Vol. 15 n° 2, April 1978.

(68) G. Paul and W. Wayne Wilson, Should the elementary function library be incorporated into computer instruction sets, ACM Trans. on Math. Software, Vol. 2 N° 2, June 1976, pp 132-142.

(69) W. Parry, On the β - expansion of real numbers, Acta math. acad. sci. Hung.,11, 1960, pp 401-416.

(70) M. Pichat, Contribution à l'étude des erreurs d'arrondi en arithmétique à virgule flottante, thése d'état, Grenoble, France, 1976

(71) A. Renyi, Representations for real numbers and their ergodic functions, Acta. Math. Acad. Sci. Hungary, 1957, pp 477-493.

(72) A Renyi, On the distribution of the digits in Cantor's series, Mat. Lapok 7, 1956, pp. 77-100.

(73) F. Robert, Iteration machine d'une fonction affine, rr. Math. App. n° 440, IMAG, Grenoble, France.

(74) B.P. Sarkar and E.V. Krishnamurthy, Economic pseudodivision processes for obtaining square root, logarithm and arctan, IEEE Trans. on Computers, Dec. 1971, pp.1589-1593.

(75) T. Sasaki, An arbitrary precision real arithmetic package in Reduce, Dept. of computer science, The univ. of Utah, Salt Lake City, Utah 84112, UUCS -79-105.

(76) C. W. Schelin, Calculator function approximation, Amer. Math. Monthly 90,5 ,May 1983.

(77) H. Schmid and A. Bogocki, Use decimal CORDIC for generation of many trancendental functions, Electrical design mag., Feb. 1973, pp 64-73.

(78) O. Spaniol, Computer arithmetic and design, J. Wiley & Sons, 1981.

(79) W.H. Specker, A Class of algorithms for ln (x), exp (x), sin (x), cos (x), arctan (x) and arcot (x), IEEE Trans. on electronic computers, Vol. ec-14, 1965, pp 85-86.

(80) C. Tricot, Mesures et dimensions, Thèse d'état, Université Paris-sud, centre d'Orsay, Paris, Dec. 1983.

(81) J.M. Trio, Microprocesseurs 8086-8088 Architecture et programmation, Coprocesseur de calcul 8087, Editions Eyrolles, Paris, 1984.

(82) J. Volder, The CORDIC Computing technique, IRE Trans. on Computers, Vol. ec-8, Sept. 1959, pp 330-334.

(83) J. Walther, A Unified algorithm for elementary functions, Joint Computer Conference Proceedings, Vol. 38, pp 379-385.

(84) E.H. Wold, Pipeline and parallel-pipeline FFT processors for VLSI implementations, IEEE Trans. on Computers, Vol. c-33 n° 5, May 1984.

(85) J.M. Yohe, Roundings in floating-point arithmetic, IEEE Trans. on Computers, Vol c-22 n° 6, June 1973.

AUTORISATION de SOUTENANCE

VU les dispositions de l'article 15 Titre III de l'arrêté du 5 juillet 1984 relatif aux études doctorales

VU les rapports de présentation de Messieurs

. G. NOGUEZ, Professeur
. F. ROBERT, Profeseur

Monsieur Jean-Michel MULLER

est autorisé à présenter une thèse en soutenance en vue de l'obtention du diplôme de DOCTEUR de L'INSTITUT NATIONAL POLYTECHNIQUE DE GRENOBLE, spécialité "Mathématiques Appliquées".

Fait à Grenoble, le 9 juillet 1985

Le Président de l'I.N.P.-G

D. BLOCH
Président
de l'Institut National Polytechnique
de Grenoble

P.O. le Vice-Président,

RESUME

Après avoir fait le point sur les méthodes **logicielles** couramment utilisées pour le calcul des fonctions élémentaires, nous abordons dans cette thèse l'étude des méthodes **matérielles**, étude imposée par l'essor de la technologie dans ce domaine.

A l'aide d'un outil mathématique nouveau, les **bases discrètes**, nous retrouvons et justifions théoriquement des algorithmes connus, et nous en mettons au point de nouveaux.

Nous présentons ensuite le coprocesseur FELIN, qui utilise les méthodes présentées dans cette thèse (en particulier celles de réduction d'argument), et qui est le fruit d'un important travail d'équipe au sein du laboratoire TIM3.

Mots-clés: Fonctions élémentaires, CORDIC, Coprocesseurs.